Rapport de stage de L3
Étude comparative des réseaux de neurones pour l'apprentissage des paramètres d'une équation différentielle

Arthur Barjot

Mai 2023

Encadrants de stage : Roberta TITTARELLI, Noura DRIDI NAFOUSSI, Zeina AL MASRY

Table des matières

1 Introduction

L'objectif de mon stage de L3 était d'investiguer la résolution l'équation de la chaleur (sous différentes formes), à l'aide des réseaux de neurones, qui sont de plus en plus employés en calcul numérique [3]. Ces travaux ont pour but de tester de nouvelles idées pour voir lesquelles sont fécondes, dans le but d'être utilisées dans des domaines de recherche concrets tels que la détection du cancer. En effet des études ont montré que le développement d'une tumeur dans le sein conduisait à une augmentation locale de la température à l'endroit où se trouve la tumeur (la tumeur agit comme une source de chaleur), l'enjeu est donc de s'appuyer sur des simulations numériques afin, par exemple, de détecter une source anormale de chaleur. J'ai notamment eu l'occasion de participer à des réunions avec d'autres stagiaires ou chercheurs dans le médical, ou dans les sciences de données, pour écouter leurs résultats et pour leur présenter les miens, tout cela dans le but de converger vers des projets communs.

Ce stage n'a pas ressemblé à ce que je peux faire au cours de l'année, il a en grande partie été plus appliqué, j'ai notamment fait beaucoup de programmation. Il m'a permis de prendre du temps pour exprimer ma créativité et mes aptitudes de recherche.

Les différentes parties du rapport suivent le déroulement du stage : j'ai d'abord travaillé sur des méthodes classiques de résolution numérique d'EDP (équation aux dérivées partielles), puis j'ai découvert les réseaux de neurones, et enfin je les ai utilisés pour résoudre notre problème.

2 Résolution d'EDP par approximation numérique

Pour me familiariser avec la résolution numérique des EDP, j'ai d'abord étudié la méthode des Différences Finies [2]. Elle consiste à discrétiser le problème en approximant les dérivées par des taux d'accroissement, ainsi nous transformerons une EDP linéaire en un système d'équations linéaires complètement résoluble. Explicitons la méthode sur l'exemple de l'équation de la chaleur monodimensionnelle sans terme source. Nous cherchons $T = T(t, x)$ tel que :

$$\begin{cases} \forall x \in]0,1[, \forall t \in]0, t_f[, & \frac{\partial T}{\partial t}(t,x) = \kappa \frac{\partial^2 T}{\partial x^2}(t,x), \\ \forall t \in [0, t_f], & T(t,0) = T(t,1) = 0, \\ \forall x \in [0,1], & T(0,x) = sin(2\pi x). \end{cases}$$

En suivant [2] nous fixons N, M$\in \mathbb{N}$ tels que $\Delta t = \frac{t_f}{N}$ et $\Delta x = \frac{1}{M}$ seront nos pas de discrétisation en temps et en espace respectivement. On note $t_i = i\Delta t$ et $x_k = k\Delta x$. De plus nous allons approcher la solution aux noeuds $T(t_i, x_k)$ par T_k^i.

Alors nous approximons l'équation :

$$\begin{cases} \forall k \in [\![1, M-1]\!], \forall i \in [\![0, N-1]\!], & \frac{T_k^{i+1} - T_k^i}{\Delta t} = \kappa \frac{\frac{T_{k+1}^i - T_k^i}{\Delta x} - \frac{T_k^i - T_{k-1}^i}{\Delta x}}{\Delta x}, \\ \forall i \in [\![0, N]\!], & T_0^i = T_M^i = 0, \\ \forall k \in [\![1, M-1]\!], & T_k^0 = sin(2\pi x_k). \end{cases}$$

Remarquons la condition initiale est compatible avec la condition aux bords au temps 0 et t_f. Ainsi nous avons un système de $(M+1)(N+1)$ équations libres, à $(M+1)(N+1)$ inconnues. Une possibilité est de calculer la solution par récurrence (grâce aux formules ci-dessus). Mais plus généralement nous écrivons ce système sous forme matricielle par : $T^{i+1} = AT^i$, $i \in [\![0, N-1]\!]$, où

$$A = \begin{bmatrix} (1-2\lambda) & \lambda & 0 & \cdots & 0 \\ \lambda & (1-2\lambda) & \lambda & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & \lambda & (1-2\lambda) & \lambda \\ 0 & \cdots & 0 & \lambda & (1-2\lambda) \end{bmatrix} \in \mathcal{M}_{M-1}(\mathbb{R}), \text{ et } \lambda = \frac{\kappa \Delta t}{\Delta x^2}.$$

Sans rentrer dans les détails, nous prenons en compte que cette méthode sera convergente (lorsque $\Delta t, \Delta x \to 0$) sous la condition de stabilité : $\Delta t < \frac{(\Delta x)^2}{2\kappa}$.

J'ai pu voir aussi quelques idées sur la méthode des Éléments Finis sur un problème similaire, elle est théoriquement très intéressante et adaptée à des problèmes plus généraux, mais pour le problème qui nous intéresse, la méthode des Différences Finies sera suffisante.

3 Réseau de neurones

3.1 Principe général

Un réseau de neurones [1] est une fonction qui a pour but d'estimer une fonction inconnue f^* résolvant un problème. Pour nous ici, un réseau de neurones fournira une approximation de la solution de notre EDP. Le réseau de neurones est déterminé par plusieurs paramètres : une fonction "d'activation" σ, un nombre N de couches, des nombres $(n_i)_{1 \leq i \leq N}$ de neurones par couches, des poids $(w_{j,l}^{(i)})$ et des biais $(b_l^{(i)})$. Il est souvent représenté par un graphe orienté pondéré, comme dans la figure 1. Sur cette figure nous avons $N = 4$ couches, et $n_1 = n_2 = n_3 = 2, n_4 = 1$ neurones sur les couches respectives.

Le vecteur $(a_{1,1}, a_{1,2}) \in \mathbb{R}^2$ est l'entrée et le scalaire $a_{4,1} \in \mathbb{R}$ est la sortie. Chaque couche i est calculée de la manière suivante :

$$\forall i \in [\![2, N]\!], \ \forall j \in [\![1, n_i]\!], \quad a_{i,j} = \sigma \left(b_j^{(i-1)} + \sum_{k=1}^{n_{i-1}} w_{k,j}^{(i-1)} a_{i-1,k} \right).$$

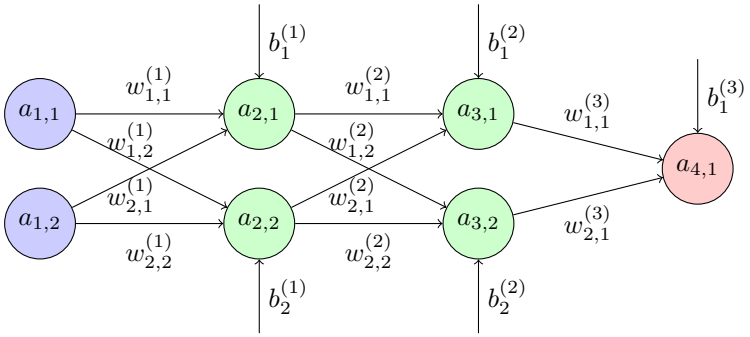

FIGURE 1 – Réseau de neurones

Nous noterons les poids en terme matriciel : $W^{(i)} = (w_{k,j}^{(i)})_{k,l}$ et $B^{(i)} = (b_j^{(i)})_j$. Et $W = \{W^{(i)}, B^{(i)} | 1 \leq i \leq N\}$ représente une pondération du réseau de neurones. On notera $x \mapsto f(W, x)$ la fonction associée au réseau pour la pondération W. Par exemple sur la figure 1, x est choisi dans $\mathbb{R}^2$ et $f(W, x)$ sera dans $\mathbb{R}$, et plus généralement, un réseau de neurones est une application de $\mathbb{R}^n$ dans $\mathbb{R}^m$ (n entrées et m sorties). La fonction σ étant choisie au début de la création du réseau de neurones, l'enjeu sera de trouver les bons poids pour que le réseau colle au mieux avec la fonction qu'on lui demande d'approximer (notée f^*). Pour cela le réseau de neurones doit subir une phase d'entraînement.

Tout d'abord nous devons constituer un ensemble de valeurs $(x_i)_{1 \leq i \leq M}$ en lesquels nous connaissons la valeur de f^* (cette base de données peut être constituée expérimentalement, pour nous elle sera obtenue à partir de la méthode d'approximation numérique de la partie précédente). Ensuite nous choisirons arbitrairement des valeurs pour les poids, et nous calculerons ce que renvoie notre réseau de neurones pour les entrées (x_i). Il va s'agir de "corriger" le réseau pour qu'au prochain test il renvoie de meilleures valeurs en les (x_i).

Pour cela nous allons minimiser une fonction de perte $\mathcal{L}$, par exemple la fonction $x \mapsto \sum_{k=1}^{M} \| f(x_i) - f^*(x_i) \|^2$ (erreur quadratique). Nous utiliserons des méthodes du type descente de gradient. L'idée de base de ce type de méthode consiste à calculer les dérivées partielles de la fonction de perte par rapport aux différents poids et modifier ces poids proportionnellement à ces gradients : on se fixe $\alpha \in \mathbb{R}_+^*$, qui est appelé le "learning rate", et est noté lr, et on fait la mise à jour suivante : $w_{j,l}^{(i)} \leftarrow w_{j,l}^{(i)} - \alpha \frac{\partial \mathcal{L}}{\partial w_{j,l}^{(i)}}(w_{j,l}^{(i)})$. Enfin nous itérons les étapes de test et de correction, on dit qu'une phase de test et de correction est une "époque". La méthode de descente de gradient est particulièrement adaptée aux fonctions convexes, quand ce n'est pas le cas, le risque est que l'algorithme converge quand $\mathcal{L}$ est proche d'un minimum local, mais nous avons tout de même des techniques pour remédier à ce problème, comme le "batching" qui consiste à entraîner le réseau sur une partie seulement des données à chaque époque, pour créer plus d'aléatoire dans l'entraînement.

La puissance des réseaux de neurones est notamment dans la simplicité du

calcul du gradient de $\mathcal{L}$ avec un algorithme de "backpropagation" [1] : nous calculons la dérivée partielle de $\mathcal{L}$ par rapport aux poids sur la couche la plus profonde (la plus à droite du schéma). Ensuite récursivement nous pouvons calculer les dérivées partielles par rapport à la couche i en fonction de ceux de la couche i+1 en utilisant la règle de la chaîne pour la dérivation de fonction composée.

3.2 Application à la résolution des EDP

Nous allons utiliser un réseau de neurones pour résoudre numériquement une équation différentielle en utilisant la méthode PINN (Physics-Informed Neural Network) [6] présentée ci-dessous. Considérons un problème générale non-stationnaire de la forme :

$$\begin{cases} \forall x \in \Omega, \forall t \in]0,T[, & \frac{\partial u}{\partial t}(t,x) = F(u(t,x), \frac{\partial u}{\partial x}(t,x), ..., \frac{\partial^k u}{\partial x^k}(t,x)), \\ \forall t \in [0,T], \forall x \in \Gamma & u(t,x) = 0, \\ \forall x \in \Omega, & u(0,x) = u_0(x), \end{cases}$$

où Ω est un domaine compact de $\mathbb{R}^d$, Γ est sa frontière, F est une fonction de $\mathbb{R}^{k+1}$ dans $\mathbb{R}$et u est notre fonction inconnue, de $\Omega \times [0,T]$ dans $\mathbb{R}$. Vu l'ampleur du sujet, nous n'aborderons pas les réflexions sur les EDP en général, comme par exemple sous quelles hypothèses nous avons un problème bien posé. Supposons juste que cela soit le cas.

Notons u la fonction associée à notre réseau de neurones et u^* la solution exacte du problème. Imaginons que nous ayons constitué une base de données de valeurs en lesquelles nous connaissons u^* (en prenant des mesures par exemple si nous traitons un problème de physique). On note donc $(x_i^{data})_{1\leq i\leq n}$, $(t_i^{data})_{1\leq i\leq n}$ et $(u_i^{*,data})_{1\leq i\leq n}$ des données telles que : $u^*(t_i^{data}, x_i^{data}) = u_i^{*,data}$. Nous définissons alors une fonction de perte associée aux données :

$$\mathcal{L}_{DATA}(u) = \sum_{k=1}^{n} \left\| u(t_i^{data}, x_i^{data}) - u_i^{*,data} \right\|^2$$

. Nous pouvons donc entraîner notre réseau sur ces données, mais cela ne suffit pas pour une bonne convergence.

Le cœur de la méthode PINN est d'intégrer l'équation différentielle directement dans la fonction de perte en ajoutant

$$\mathcal{L}_{EDP}(u) = \sum_{i=1}^{n} \left\| \frac{\partial u}{\partial t}(t_i^{edp}, x_i^{edp}) - F(u(t_i^{edp}, x_i^{edp}), \frac{\partial u}{\partial x}(t_i^{edp}, x_i^{edp}), ..., \frac{\partial^k u}{\partial x^k}(t_i^{edp}, x_i^{edp})) \right\|^2,$$

où les $(x_i^{edp})_{1\leq i\leq n}$ sont des points choisis aléatoirement sur Ω et les $(t_i^{edp})_{1\leq i\leq n}$ sont des points choisis aléatoirement dans $]0,T[$. Ainsi à chaque époque nous

corrigerons notre réseau de neurones pour qu'il vérifie au mieux l'équation différentielle. De même nous créons une fonction de perte visant à forcer le réseau de neurones à vérifier les conditions initiales et aux bords respectivement :

$$\mathcal{L}_{CB}(u) = \sum_{k=1}^{n} \left\| u(t_i^{bord}, x_i^{bord}) - 0 \parallel^2, \quad \mathcal{L}_{CI}(u) = \sum_{k=1}^{M} \parallel u(0, x_i^{init}) - u_0(x_i^{init}) \right\|^2,$$

où les $(x_i^{bord})_{1\leq i\leq n}$ sont des points choisis aléatoirement sur Γ, les $(x_i^{init})_{1\leq i\leq n}$ sont des points choisis aléatoirement dans Ω et les $(t_i^{bord})_{1\leq i\leq n}$ sont des points choisis aléatoirement dans $]0, T[$. En considérant une combinaison de ces quatre fonctions de perte, nous pouvons entraîner un réseau de neurones.

Nous pouvons donc construire une fonction de perte tenant compte de l'équation et des données :

$$\mathcal{L}(u) = \alpha_{EDP}\mathcal{L}_{EDP}(u) + \alpha_{CB}\mathcal{L}_{CB}(u) + \alpha_{CI}\mathcal{L}_{CI}(u) + \alpha_{DATA}\mathcal{L}_{DATA}(u) \quad (1)$$

Où les coefficients α sont à choisir. Nous avons proposé une mise à jour de ces coefficients au cours de l'entraînement en fonction de quelle fonction de perte est la moins minimisée [5].

La résolution numérique présentée dans la section 2 est efficace pour résoudre numériquement une EDP, et nous disposons de résultats de convergence, mais ces méthodes ont des limites qui motivent l'utilisation des réseaux de neurones :

Par exemple nous pouvons nous intéresser à un problème dans lequel certains paramètres sont inconnus. Par exemple, notre équation décrivant un phénomène physique peut tenir compte d'un paramètre propre au milieu étudié, si ce paramètre n'est pas une constante de l'espace, alors il peut être très dur à estimer. Ainsi les méthodes de résolution classique sont obsolètes car elles exploitent le déterminisme de l'équation, qui n'est plus si l'on ne connaît plus tous les paramètres. Encore une fois, la méthode utilisant un réseau de neurones nous permettra de résoudre le problème, en apprenant les paramètres inconnus via un réseau de neurones, nous verrons ceci dans la section 5.

4 Équation de la chaleur, problème direct

Le cœur de mon stage consistait à trouver une solution à l'équation de la chaleur en utilisant la méthode PINN (voir section 3.2). Je parle dans cette partie de problème "direct" car nous connaissons tous les paramètres du problème, ensuite dans la sectino 5 nous nous intéresserons au problème "inverse" dans lequel certains paramètres de l'équation sont inconnus.

Pour ce cas d'étude simple, nous savons calculer analytiquement la solution (voir l'annexe). Cependant ici nous nous intéressons à l'approche numérique qui permet de généraliser l'étude à des cas où la solution analytique sera trop difficile à calculer (ou expliciter).

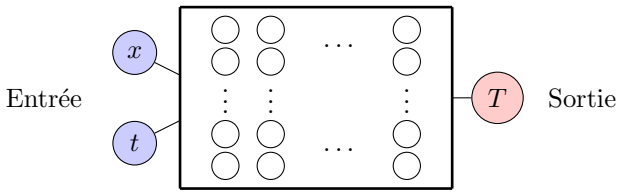

FIGURE 2 – Architecture du réseau de neurones

Nous avons vu que dans un cas si simple (équation linéaire), l'utilisation d'un réseau de neurones n'est pas nécessaire, mais cette partie du stage m'a permis de programmer et tester les notions théoriques de la partie précédente.

Voici le premier problème que je souhaite résoudre avec mon réseau de neurones (équation de la chaleur 1D avec terme source constant) : trouver $T = T(t, x)$ tel que

$$(\mathcal{E}) \begin{cases} \forall x \in]0,1[, \forall t \in]0, t_{max}], & \frac{\partial T}{\partial t}(t,x) = \kappa \frac{\partial^2 T}{\partial x^2}(t,x) + S, \\ \forall t \in]0, t_{max}], \forall x \in \{0,1\} & T(t,x) = 0, \\ \forall x, y \in [0,1], & T(0,x) = u_0(x), \end{cases}$$

Où S sera constante égale à 1, κ constante égale à 0.5, et u_0 est une fonction infiniment dérivable valant 0 sur les bords et 1 au centre du segment :

$$u_0 : x \mapsto \sin(\pi x).$$

4.1 Présentation du programme

Pour ce faire j'ai implémenté un programme python exploitant la méthode PINN, en utilisant principalement la librairie Pytorch, simulant un réseau de neurones, dont je pouvais modifier de nombreux paramètres. Je me suis inspiré d'un code résolvant une équation plus simple trouvé sur Github [4].

Sur la figure 2 on peut voir que T est une fonction scalaire, le nombre exacte de neurones sera donné au moment de la présentation des résultats. Chacune des couches cachées aura le même nombre de neurones.

Pour ce qui est de la fonction d'activation, j'ai choisi la fonction ELU :

$$ELU : x \mapsto \begin{cases} x & \text{si } x > 0, \\ \exp(x) - 1 & \text{si } x \leq 0. \end{cases}$$

Cette fonction a plusieurs avantages, tout d'abord le calcul de ses dérivées est très simple, ce qui rend moins complexe la descente de gradient. Ensuite ses dérivées ne sont pas nulles, ce qui n'est pas le cas pour la fonction RELU qui a une dérivée seconde nulle, ce qui est problématique car nous devons obtenir une solution de l'équation différentielle (qui est d'ordre deux).

On implémente alors une méthode des Différences Finies résolvant le problème (méthode que j'ai présenté dans la section 2). Ceci nous fournit les données d'entraînement, si notre modèle fonctionne bien alors nous pourrons utiliser de vraies données expérimentales pour l'entraîner plutôt que d'utiliser une résolution numérique.

L'implémentation utilise des fonctions de la librairie Pytorch, nous ne reprogrammons donc pas une descente de gradient ou un algorithme de "back-propagation". Ainsi nous commençons par une définition des paramètres du problème et du réseau de neurones, ensuite nous implémentons notre équation différentielle, c'est à dire une fonction qui prend en entrée le réseau de neurones et un point (x, t), et qui renvoie en sortie $\frac{\partial T}{\partial t}(t, x) - -\kappa \frac{\partial^2 T}{\partial x^2}(t, x) - S$ (on utilise la différentiation automatique de Pytorch). On implémente aussi la fonction u_0.

Enfin, il ne reste plus qu'à créer la boucle d'entraînement, comprenant la descente de gradient et la "back-propagation". Puis nous utilisons matplotlib pour produire la solution et des courbes contenant des informations sur l'apprentissage.

Remarque : Améliorations du programme

Plusieurs améliorations ont été faites pour nous permettre d'obtenir de bons résultats et dans un temps raisonnable.

Tout d'abord j'ai ajouté une fonction mettant à jour les pondérations α_{EDP}, α_{CB}, α_{CI}, α_{DATA} (voir la fin de la partie 3.2) [5]. Ainsi toutes les 10 époques, nous rectifions ces coefficients pour que nous entraînions le réseau de neurones sur ses faiblesses, ainsi chacun des critères (l'EDP, les conditions initiales et aux bords, et les points de datas) seront minimisés autant que les autres.

Ensuite j'ai ajouté du "batching". En fait nous allons tester et mettre à jour le réseau bien plus souvent mais à chaque étape il s'entraînera sur moins de donnés.

Enfin j'ai ajouté une mise à jour du learning rate, pour obtenir une meilleure précision. En effet, arrivée à un certain nombre d'époques, la fonction de perte ne bouge plus car les corrections que nous lui apportons sont trop grossières, ainsi il faut affiner cette correction. Pour ce faire j'utilise encore une fois des fonctions de la librairie Pytorch permettant par exemple de diviser par deux le learning rate toutes les 20000 époques.

4.2 Simulation et résultats

L'obtention de bons résultats a été une partie très éprouvante du stage car cela peut prendre énormément de temps. Plusieurs fois, j'ai dû entraîner le réseau de neurones pendant des heures pour tester une amélioration qui en fait ne conduisait à aucun résultat concluant.

Pour le choix des paramètres, j'ai expérimenté de nombreuses possibilités, mais voici mon choix final :

Intervalle de temps	[0,0.8] pour atteindre un état stationnaire
Nombres de points de data	150
Nombres de points d'entraînement pour l'EDP	500
Nombre d'époques	100 000
Learning rate	0.02, divisé par deux toutes les 45 000 époques
Architecture	16 couches de 13 neurones
Nombre de lot pour le batching	5

L'entraînement a duré environ 7h30 et nous avons obtenu une fonction perte de l'ordre de 10^{-2}, et une erreur entre le réseau et la méthode des différences finies de l'ordre de 10^{-2}. J'ai aussi programmé un réseau qui résout un problème stationnaire, celui-ci renvoie des meilleurs résultats : une fonction de perte de l'ordre de 10^{-5}, et une erreur de l'ordre de 10^{-3}.

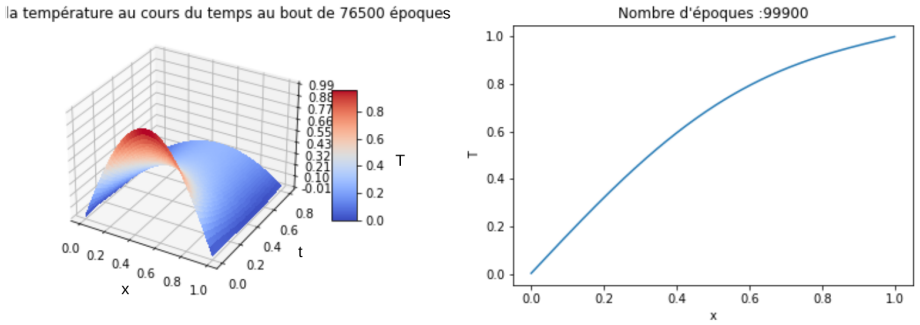

FIGURE 3 – À gauche la courbe obtenue pour une condition initiale sinusoïdale et conditions aux bords nulles, à droite l'état stationnaire pour des conditions aux bords de 0 et 1.

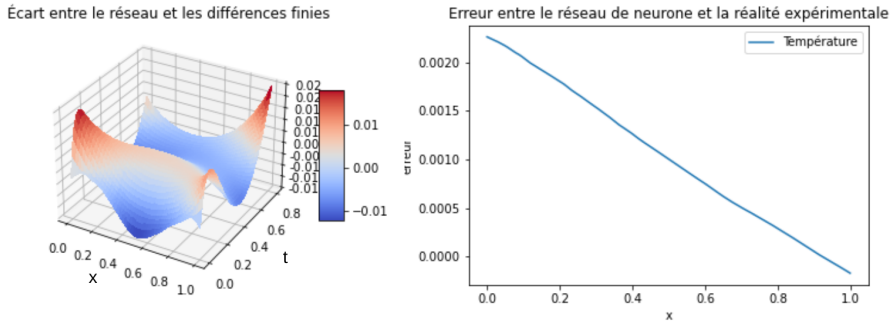

FIGURE 4 – Les écarts entre les courbes de la figure 3 et les courbes obtenues avec la méthode des Différences Finies pour ces deux problèmes respectifs.

4.3 Le cas 2D, terme source constant

J'ai facilement su généraliser le cas 1D au cas 2D, puis j'ai passé une très longue partie de mon stage à tenter d'avoir des résultats concluants en 2D, mais l'entraînement était beaucoup trop long, et les réseaux de neurones utilisées n'avait peut-être pas assez de neurones. Voici tout de même ce à quoi ressemble mes résultats :

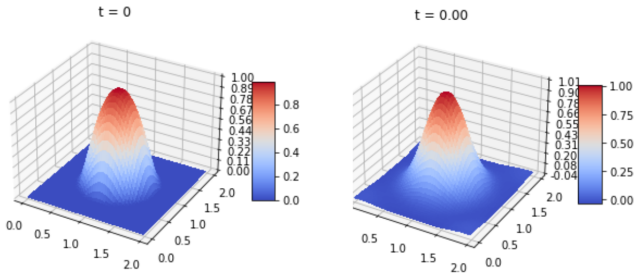

FIGURE 5 – À gauche, la courbe souhaitée, à droite la courbe du réseau de neurones après 300000 époques, toutes deux pour l'instant $t = 0$.

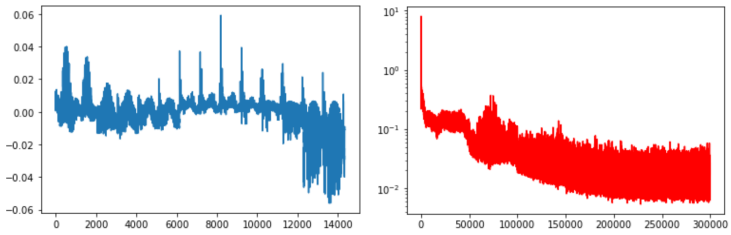

FIGURE 6 – À gauche, l'erreur en différents points de l'espace-temps $[0, 1]^2 \times [0, 0.8]$, à droite la fonction de perte en fonction des époques.

Nous avons obtenu une erreur allant jusqu'à 0.06 entre l'estimation par réseau de neurones et la méthode des différences finies.

5 Détermination de paramètres, problème inverse

On note $\mathcal{D}$ l'ensemble de données (data) générées avec la méthode des Différences Finies, mais que nous appellerons "données expérimentales". Nous nous plaçons dans le cadre du problème monodimensionnel car la généralisation à de plus grandes dimensions n'est pas l'étape la plus difficile, mais elle demande

une durée d'entraînement décuplée. Pour les mêmes raisons nous n'abordons que le problème stationnaire $\frac{\partial T}{\partial t} = 0$.

Cependant ici nous ne considérons plus le terme source S constant (S dépend de x), ni même connu.

Nous considérons des températures extérieures au système constantes, les conditions aux bords valant T_0 et T_1, nous voudrions tenter de trouver $T = T(x)$ et $S = S(x)$ tels que :

$$(\mathcal{V}) \begin{cases} \forall x \in]0,1[, & -\kappa \Delta T(x) = S(x), \text{(I)} \\ & T(0) = T_0, \text{ et } T(1) = T_1, \text{(II)} \\ \forall (x,u) \in \mathcal{D}, & T(x) = u, \text{(III)} \end{cases}$$

Avec $\kappa > 0$ donné. Si S est "assez régulier", nous pouvons montrer qu'il existe une unique solution à (I) et (II). Or lorsque S est inconnu nous voulons investiguer des manières pour trouver (au moins) un couple (T,S) qui soit admissible avec les données (III).

Dans la suite nous allons nous donner des formes connues pour S avec des paramètres inconnus. L'enjeu sera de déterminer ces paramètres et la solution T à partir des données et de l'EDP. Pour cela, nous appelons ce problème un problème inverse.

5.1 Terme source sinusoïdal

Dans cette sous partie nous cherchons S de la forme $\alpha \sin(\beta x)$ où α et β sont à déterminer. C'est un cas plutôt simple car l'équation est facilement résoluble mais il va bien introduire les idées que j'ai pu développer :

$$T(x) = \frac{\alpha}{\beta^2 \kappa} \left(\sin(\beta x) - \sin(\beta) x\right) + \underbrace{(T_1 - T_0)x + T_0}_{C(x)} \tag{2}$$

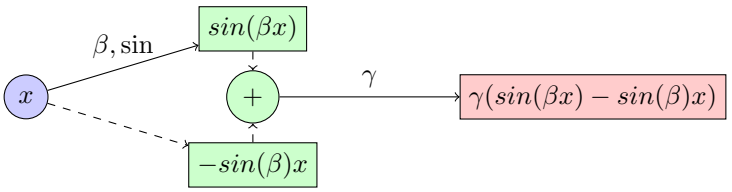

FIGURE 7 – Architecture du réseau

Ce n'est pas rigoureusement un réseau de neurones car le neurone nommé $\boxed{-\sin(\beta)x}$ n'est pas vraiment un neurone, pour le calculer nous extrayons le poids β reliant les deux premiers neurones, et l'entrée x. Aucun nouveau poids ou biais n'est mis en jeu. Ensuite nous ajoutons les deux "neurones" de la couche cachée (neurones vert sur la figure 7) et lui appliquons un poids γ. Le poids β

est bien une estimation du paramètre β que nous cherchons, le poids γ quant à lui à la fin de l'entraînement nous donnera un estimation de la quantité $\frac{\alpha}{\beta^2\kappa}$, donc si nous connaissons ces deux poids, nous trouvons facilement α et β.

Pour trouver la température, en suivant l'équation 2, il ne reste plus qu'à ajouter $C(x)$ qui est connu.

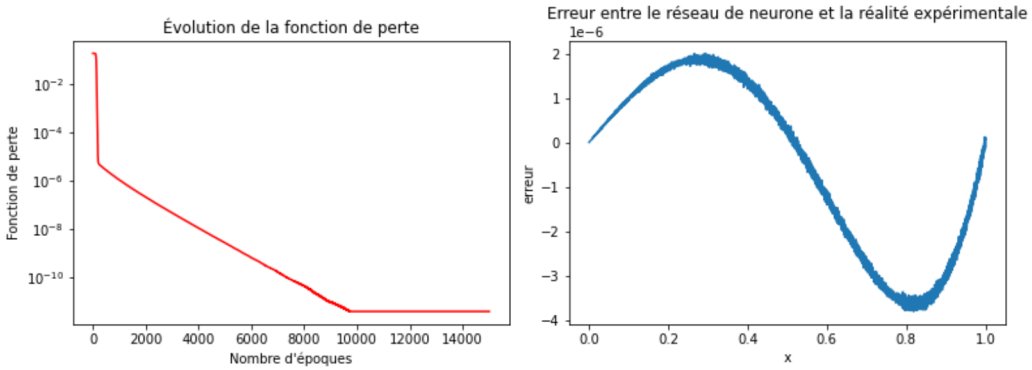

FIGURE 8 – Les courbes de la fonction de perte et de l'erreur entre notre réseau et la courbe souhaitée

Nous obtenons des résultats très satisfaisant pour un entraînement très court, moins d'une minute pour 10 000 époques. En choisissant $\alpha = 3$, $\beta = 2$ on trouve :

```
alpha =  2.9999079196694702
beta =  2.000131130218506
époque :  10000 Traning Loss: 8.24466363236187e-12
Temps d'exécution :  50.28628373146057  secondes
```

Notons que pour une autre initialisation des poids, le réseau peut converger vers $\alpha = -3$, $\beta = -2$, ce qui est en fait la même solution par imparité du sinus.

5.2 Terme source en cloche

Dans cette sous partie nous cherchons S de la forme $\gamma \exp(\alpha(x-\beta)^2)$ où $\alpha < 0$, $\beta \in]0,1[$ et $\gamma > 0$ sont à déterminer. Le problème est bien plus complexe que le précédent car l'équation n'est pas résoluble : on ne connaît de primitive utilisant les fonctions classiques à une telle fonction.

Nous avons eut plusieurs idées plus ou moins concluantes pour résoudre ce problème. Nous allons en exposer trois.

1. Nous avons proposé un algorithme PINN couplant deux réseaux de neurones, un premier réseau R_S décrit S, et exploite dans son architecture le fait que l'on connaisse la forme de S, et un autre réseau R_T décrit T. Cependant il n'est pas vrai que si R_T est proche de T alors sa dérivée seconde sera très proche de la dérivée seconde de T, donc on ne peut pas exploiter les données dans l'entraînement de R_S, j'ai donc abandonné cette idée.

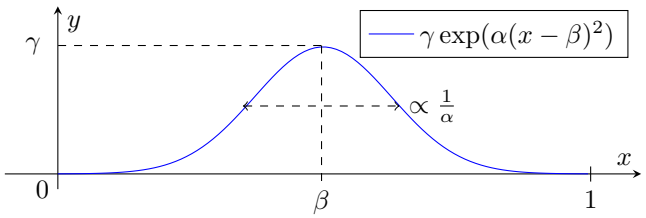

FIGURE 9 – Forme du terme source S

2. Nous avons envisagé un seul réseau, celui décrivant S, ainsi nous pouvons jouer sur son architecture car nous connaissons la forme de S, et pour l'entraîner sur les données, nous revisitons la méthode PINN en considérant la forme intégrale de l'EDP (faisant apparaître le symbole $\int$ plutôt que des dérivations). Cette méthode laisse présager une bonne convergence car elle ne commet pas les même inexactitudes que celles de la première idée, mais l'intégration numérique est trop lourde en temps de calcul (pour calculer une primitive seconde), or nous devons l'utiliser de très nombreuses fois (à chaque époques), donc j'ai choisi d'abandonner cette idée.
3. Si l'on fait une synthèse des deux idées précédentes, nous avons vu qu'il est très intéressant de considérer la forme intégrale de l'EDP. Or les fonctions considérées étant de classe $\mathcal{C}^\infty([0,1],\mathbb{R})$, nous pouvons approcher S par son développement limité (à un ordre k nous satisfaisant), puis par la formule de Taylor-Young, nous obtenons un développement de T à l'ordre k+2 (seuls les deux premiers termes du développement sont à déterminer mais ceci est simple grâce aux conditions aux bords),ce qui donne l'approximation suivante :

$$T(x) \approx T(\beta) + (x-\beta)T'(\beta) + ... + (x-\beta)^k \underbrace{T^{(k)}(\beta)}_{\frac{-1}{\kappa}S^{(k-2)}(\beta)},$$

Avec β comme dans la figure 9.

Pour utiliser cette méthode, nous avons exploité le fait que la forme choisie pour S soit très simple à développer en série entière, et un calcul simple permet de dire qu'avec un développement de S à l'ordre 12 conduit à une erreur d'approximation inférieure à 10^{-11} en norme infinie sur [0,1] tout entier. Nous cherchons donc T sous la forme :

$$T(x) = \gamma \left(C_1 x + C_2 + \underbrace{\sum_{i=0}^{12} (\alpha(x-\beta)^2)^i (x-\beta)^2 \frac{1}{i!(2i+1)(2i+2)}}_{=:\Sigma} \right), \qquad (3)$$

Notre réseau de neurones va estimer la somme (Σ), puis nous trouverons C_1 et C_2 en évaluant la formule 3 en 0 et en 1 ($T(0) = T_0$ et $T(1) = T_1$).

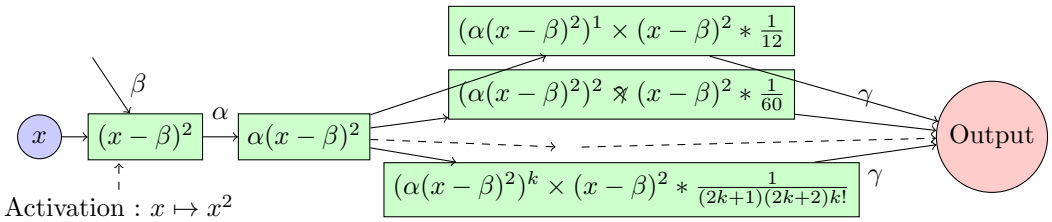

FIGURE 10 – Architecture du réseau

Comme dans les parties précédentes, l'équation différentielle et les conditions aux bords seront automatiquement vérifiées grâce à l'architecture choisit pour le réseau donc nous n'allons entraîner le réseau que sur les données.

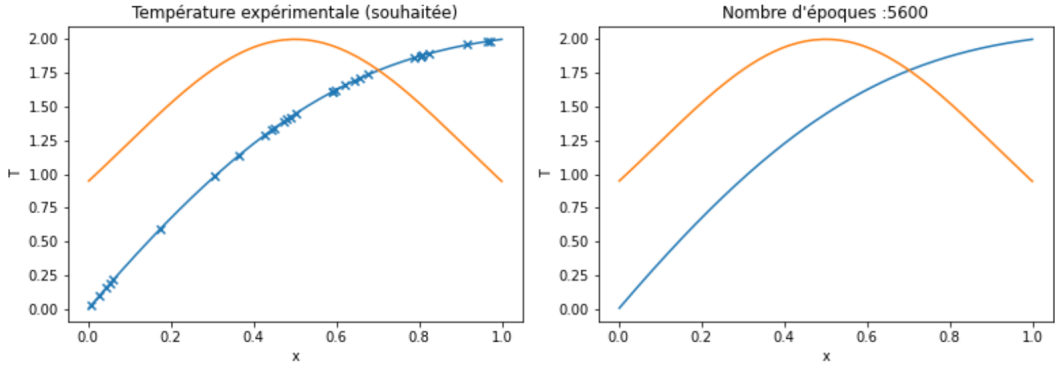

FIGURE 11 – La courbe objectif, et la courbe obtenue après entraînement

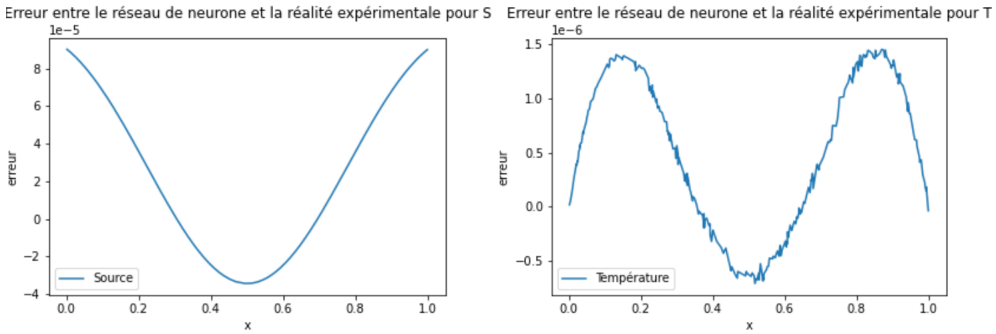

FIGURE 12 – Les courbes d'erreur pour S et T.

L'entraînement converge très rapidement, et cette solution est très efficace (du moins pour ce problème précis). On peut voir ci-dessus la courbe que l'on souhaite atteindre avec le réseau (nous utilisons la méthode des Différences Finies), on choisit des paramètres $\alpha = -3$, $\beta = 0.5$ et $\gamma = 2$, sur certaines courbe vous pouvez voir écrit $\beta = -0.5$ car dans le code, β est un biais qui correspond en fait au paramètre $-\beta$. Les croix bleues sur le premier graphique sont les points de prise de data pour l'entraînement du réseau.

Avec seulement 30 points de data, et 13 minutes d'entraînement, le réseau a une fonction de perte de l'ordre de 10^{-13}, et approxime la température à 10^{-6} près.

5.3 Pour aller plus loin

La dernière méthode présentée est enthousiasmante car on peut se dire qu'elle est généralisable à des formes de S bien plus complexe, par exemple : $x \mapsto (\sin(\alpha x)+\beta)\exp(\gamma(x-\delta)^2)$. Nous pouvons calculer le développement limité de cette fonction (cela devient très calculatoire à la main mais nous pouvons utiliser des programmes de calculs formels), mais le problème est que contrairement au cas précédent il est difficile d'estimer l'erreur d'approximation. Quoi qu'il en soit j'ai tester de tracer sur des exemples les courbes d'erreur entre le DL et la vraie fonction. Le problème est que localement, au point où nous faisons le DL, la fonction est très bien approchée, mais ce n'est pas le cas ailleurs. Le temps limité de mon stage ne m'a pas permis de poursuivre cette étude mais j'ai tout de même quelques idées pour résoudre ce problème. Par exemple nous pouvons découper l'intervalle d'intégration en plusieurs petits intervalles, sur lesquels un DL avec un nombre raisonnable de termes permet de bien approcher la fonction, puis nous pourrions créer un réseau de neurones pour chaque intervalle.

6 Conclusion

Ce stage a été une très bonne première expérience dans le milieu de la recherche. J'ai pu découvrir durant ce stage les notions de résolution numérique d'EDP et de réseaux de neurones, puis nous les avons utilisées pour proposer de potentielles idées dans un domaine de recherche novateur. Ces notions que j'ai pu aborder m'ont beaucoup intéressées, et le fait que mon travail pourra peut être avoir une utilité m'a encouragé dans ce travail. Tout du moins les pistes ne conduisant à rien que j'ai pu aborder pourront peut-être faire gagner du temps à d'autres. Je pense que les réseaux de neurones sont un outil très puissant, qui avec une bonne démarche scientifique peut conduire à encore beaucoup de progrès, et notamment ici dans des domaines médicaux. J'ajoute un grand remerciement à tous mes encradrants et collègues qui ont été très présents et m'ont permis de découvrir le milieu de la recherche.

Références

[1] Julie Delon. "Introduction aux réseaux de neurones et à l'apprentissage profond". In : (2018).

[2] Alfio Quarteroni · Fausto Saleri · Paola Gervasio. "Calcul Scientifique". In : (2010).

[3] Phil Kim. "MATLAB Deep Learning With Machine Learning, Neural Networks and Artificial Intelligence". In : (2017).

[4] nanditadoloi. *PINN*. 2021. url : `https://github.com/nanditadoloi/PINN/blob/main/solve_PDE_NN.ipynb`.

[5] Keith D. Humfeld Navid Zobeiry. "A Physics-Informed Machine Learning Approach for Solving Heat Transfer Equation in Advanced Manufacturing and Engineering Applications". In : (2020).

[6] N. Thuerey · P. Holl · M. Mueller · P. Schnell · F. Trost · K. Um. "Physics-based Deep Learning". In : (2022).