TIPE 2018-2019

Résolution d'équations aux dérivées partielles par l'usage de réseaux de neurones

Thomas Pouplin (n° 21633)

Problématique :

Comment utiliser les réseaux de neurones afin de mettre en oeuvre une méthode de résolution des équations aux dérivées partielles ?

Comment peut-on mettre en pratique une telle méthode dans le cadre de la résolution du modèle LWR ?

Table des matières :

Définition générale

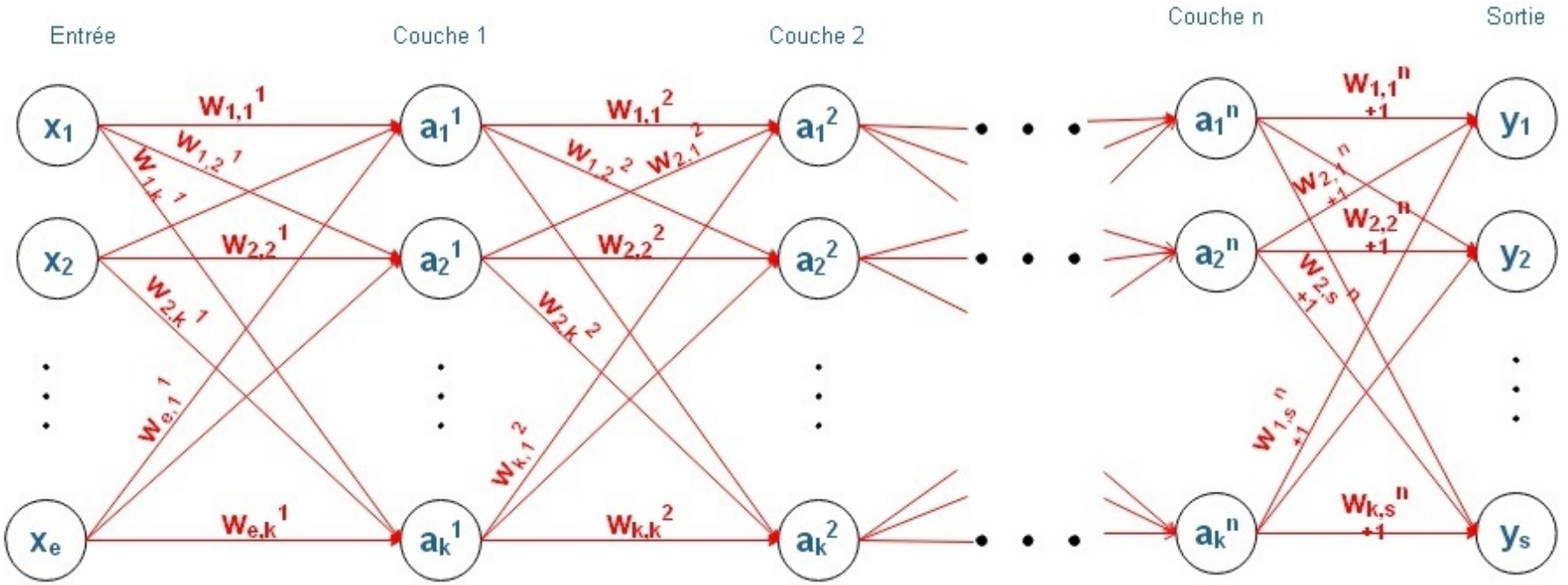
Un réseau de neurones permet de réaliser une fonction non linéaire de ses entrées.

Theorem

Théorème d'approximation universelle (König) : un réseau de neurones contenant un nombre fini de neurones peut approximer des fonctions continues sur des sous-ensembles compacts et ceci à n'importe quel degré de précision.

Notation

- a_i^j *est le neurone i de la couche j*
- $w_{i,j}^l$ *est le poids de* a_i^{l-1} *à* a_j^l
- b_i^j *est le biais attaché à* a_i^j (non représenté sur le graphe)
- Fonction d'activation : $\sigma : \mathbb{R} \to \mathbb{R}$
- Les neurones $a_i^j = \sigma(\sum_{m=1}^{k} w_{m,i}^j a_i^{j-1} + b_i^j)$



Utilisation des réseaux de neurones

$$NN : \begin{cases} \mathbb{R}^e \longrightarrow \mathbb{R}^s \\ \begin{bmatrix} x1 \\ x2 \\ \vdots \\ xe \end{bmatrix} \mapsto \begin{bmatrix} y1 \\ y2 \\ \vdots \\ ys \end{bmatrix} \end{cases}$$

avec
$yi = \sigma(\sum_{m=1}^{k} w_{m,i}^{n}(\sigma(\sum_{n=1}^{k} w_{n,m}^{(n-1)}(\dots (\sigma(\sum_{u=1}^{k} w_{u,p}^{1} x_m + b_p^1))) + b_m^{(n-1)}) + b_i^n)$.

But de la méthode proposée : on cherche à modifier NN de sorte qu'elle satisfasse une équation aux dérivées partielles.

Les hypothèses

- Une route unidimensionnelle, sans intersection.
- Un modèle macroscopique : le trafic est un fluide incompressible et on fait l'hypothèse d'un milieu continu.
- Densité : $\rho(x,t)$; Vitesse moyenne : $v(x,t)$
- La vitesse moyenne dépend uniquement de la densité, selon la relation : $v(\rho) = v_{max}(1 - exp(\frac{-c}{v_{max}}(1 - \frac{\rho_{max}}{\rho})))$.
  $v_{max} = c = \rho_{max} = 1 \Rightarrow \underline{v(\rho) = (1 - exp(1 - \frac{1}{\rho}))}$.
- Flux de voitures : $\underline{f = \rho v}$

Loi de conservation

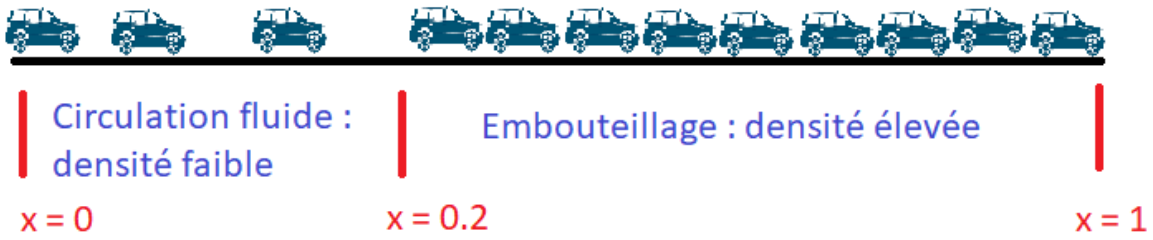
Theorem

- *Loi de conservation :* $\frac{d}{dt}\int_{x_1}^{x_2}\rho(x,t)dx = f(x_1,t) - f(x_2,t)$
- *Forme locale :* $\frac{\partial \rho}{\partial t} + \frac{\partial f\circ\rho}{\partial x} = 0$

Le modèle LWR : Situation initiale

On se place sur le carré $[0,1]^2$.

On étudie une situation où on arrive dans un embouteillage :



Condition initiale : $\rho(x,0) = g(x)$ avec

$$g : \begin{cases} [0,1] \to [0,1] \\ x \mapsto \begin{cases} 0.65 \; si \; x \leq 0.2 \\ 0.9 \; sinon \end{cases} \end{cases}$$

Fonction de coût

- Soit θ l'éventuelle solution de notre problème.
- Soit φ la fonction de coût. Elle représente une quantité que l'on cherche à minimiser.
- $\varphi \backsimeq NN - \theta$

Remarque : $\varphi = \varphi(w^1_{1,1}, w^1_{1,2} \ldots, w^{n+1}_{k,k}, b^1_1, \ldots, b^{n+1}_k)$

Implémentation de la fonction de coût

Pour satisfaire l'équation, on minimise
$\forall x, t \in [0,1]^2 \ \frac{\partial NN(x,t)}{\partial t} + \frac{\partial f \circ NN(x,t)}{\partial x}$.

Pour $I \subset [0,1]^2$: $\varphi = \sum_{x,t \in I} \left(\frac{\partial NN(x,t)}{\partial t} + \frac{\partial f \circ NN(x,t)}{\partial x} \right)^2$.

Choix de I : un nombre fixé de points pris au hasard.

Remarque: on a besoin de $\frac{\partial NN}{\partial t}$ et $\frac{\partial NN}{\partial x}$.

Implémentation des conditions initiales : 1ère implementation

On satisfait les conditions initiales par construction.

$\forall x, t \in [0,1]^2\ NN^* = A(x) + t \cdot NN(x,t)$ avec

$$A : \begin{cases} \mathbb{R} \to \mathbb{R} \\ x \mapsto \begin{cases} 0.65 \text{ si } x \leq 0.2 \\ 0.9 \text{ sinon} \end{cases} \end{cases}$$

On a alors $\varphi = \sum_{x,t \in I} \left(\frac{\partial NN^*(x,t)}{\partial t} + \frac{\partial f \circ NN^*(x,t)}{\partial x} \right)^2$.

Implémentation des conditions initiales : 2ème implementation

On ajoute les conditions initiales dans le coût.

$$\varphi = \sum_{x,t \in I} \left[\left(\frac{\partial NN(x,t)}{\partial t} + \frac{\partial f \circ NN(x,t)}{\partial x} \right)^2 + ci \cdot B(x,t)^2 \right]$$

$$B : \begin{cases} \mathbb{R}^2 \to \mathbb{R} \\ x, t \mapsto \begin{cases} 0 \; si \; t \neq 0 \\ NN(x,t) - 0.65 \quad si \; t = 0 \; et \; x \leq 0.2 \\ NN(x,t) - 0.9 \quad si \; t = 0 \; et \; x \geq 0.2 \end{cases} \end{cases}$$

$ci \in \mathbb{R}$ est un coefficient contrôlant le poids des conditions initiales dans le coût final.

Inconvénient de la seconde méthode : contrainte sur I

Algorithme du gradient

Algorithme du gradient :

- Calcul de $\nabla\varphi(w^1_{1,1}, w^1_{1,2} \ldots, w^{n+1}_{k,k}, b^1_1, \ldots, b^{n+1}_k)$; i.e $\frac{\partial\varphi}{\partial w^l_{i,j}}$ *et* $\frac{\partial\varphi}{\partial b^l_i}$;
- "Descente du gradient" : $\begin{cases} w^l_{i,j} \leftarrow w^l_{i,j} - v \cdot \frac{\partial\varphi}{\partial w^l_{i,j}} \\ b^l_i \leftarrow b^l_i - v \cdot \frac{\partial\varphi}{\partial b^l_i} \end{cases}$;

  $v \in \mathbb{R}^+$ *est la vitesse d'apprentissage*

Remarque :

- L'algorithme du gradient permet d'arriver à un minimum local mais rien ne prouve que c'est un minimum global.
- L'utilisation de l'algorithme nécessite σ dérivable.

Pseudo-code

Algorithmic 1 pseudocode

```
1: Mise en place du réseau
2: Initialisation des poids et biais
3: N = nombre de périodes d'apprentissage
4: Définition de la fonction coût
5:
6: for j = 1 to N do
7:     I = Ensemble des points d'apprentissage pris au hasard
8:     φ = 0
9:
10:    for i in I do
11:        φ += coût(i)
12:    end for
13:    Calcul des dérivées partielles de φ
14:    Descente du gradient
15: end for
```

Principe Général

Rappel : $\frac{\partial \rho(x,t)}{\partial t} + \frac{\partial f \circ \rho(x,t)}{\partial x} = 0 \Leftrightarrow \frac{\partial \rho(x,t)}{\partial t} + f'(\rho) \cdot \frac{\partial \rho(x,t)}{\partial x} = 0$

Méthode de résolution partielle.

Theorem

ρ solution de l'équation est constante le long de la courbe
$X : t \mapsto (x(t), t) \Rightarrow \forall t \in [0,1]\, x'(t) = f'(\rho(x(t), t))$

Preuve

Soit ρ solution constante le long de la courbe

$\Rightarrow \forall t \in [0,1] \ \frac{d\rho \circ X}{dt}(t) = 0$

$\Rightarrow \frac{\partial \rho}{\partial x}(x(t),t) \cdot x'(t) + \frac{\partial \rho}{\partial t}(x(t),t) = 0$

Puis ρ solution de l'équation $\Rightarrow \frac{\partial \rho}{\partial x}(x(t),t).f'(\rho) + \frac{\partial \rho}{\partial t}(x(t),t) = 0$

Donc $\forall t \in [0,1] \ \frac{\partial \rho}{\partial x}(x(t),t) \cdot (x'(t) - f'(\rho)) = 0$

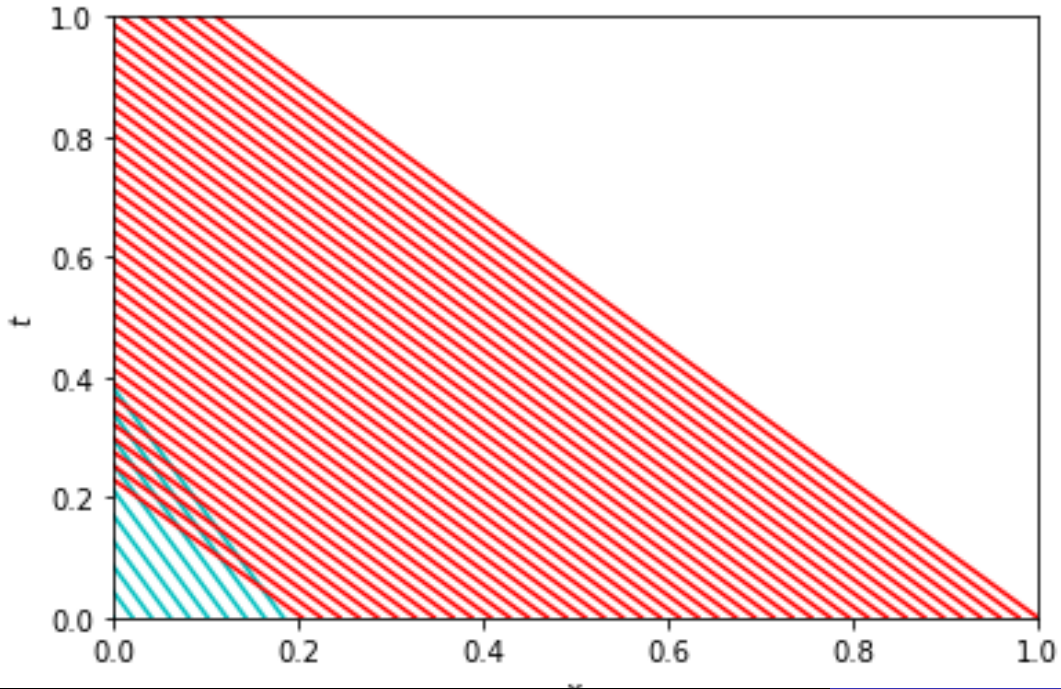
D'où en supposant $\frac{\partial \rho}{\partial x}(x(t),t) \neq 0$, *on a* $\underline{x'(t) = f'(\rho(x(t),t))}$.

Application de la méthode

$$\begin{cases} x'(t) = f'(\rho(x(t),t)) \\ \rho(x(t),t)\ \textit{constante} \end{cases} \quad \Rightarrow x'\ \textit{constante}$$

$$\Rightarrow \textit{la courbe}\ (x(t),t)\ \textit{est une droite}$$

La caractéristique partant de x_0 à pour équation $x(t) = f'(\rho(x_0,0)) \cdot t + x_0$



Solution faible : introduction

Soit ρ une solution de $\frac{\partial \rho}{\partial t} + \frac{\partial f(\rho)}{\partial x} = 0$

$$\forall \varphi \in C^1(\mathbb{R}\times\mathbb{R}^+, \mathbb{R}) \text{ à support compact},$$
$$\int_{\mathbb{R}^+}\int_{\mathbb{R}} (\frac{\partial \rho}{\partial t} + \frac{\partial f(\rho)}{\partial x})\varphi dxdt = 0.$$
$$\Rightarrow \int_{\mathbb{R}^+}\int_{\mathbb{R}} (\rho\frac{\partial \varphi}{\partial t} + f(\rho)\frac{\partial \varphi}{\partial x})dxdt + \int_{\mathbb{R}} g(x)\varphi(x,0) = 0$$

Solution faible définition

Definitions

Une fonction ρ intégrable sur $\mathbb{R} \times \mathbb{R}^+$ est une solution faible si et seulement si pour toute fonction $\varphi \in C^1(\mathbb{R}\times\mathbb{R}^+, \mathbb{R})$ à support compact, elle respecte l'identité
$\int_{\mathbb{R}^+} \int_{\mathbb{R}} (\rho \frac{\partial \varphi}{\partial t} + f(\rho)\frac{\partial \varphi}{\partial x}) dx dt + \int_{\mathbb{R}} g(x)\varphi(x,0)dx = 0$

Condition de Rankine-Hugoniot

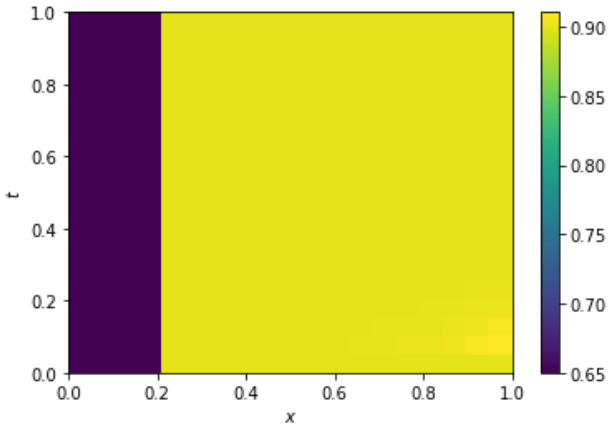
Soit λ la dérivée de la courbe de discontinuité, ρ^+ *et* ρ^- les valeurs de ρ de part et d'autre de la discontinuité.

Theorem

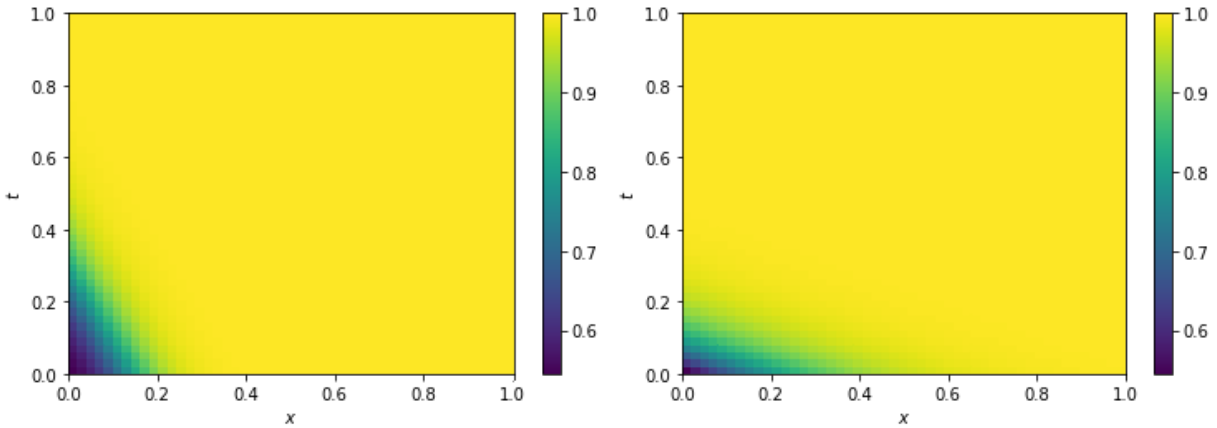
Condition de Rankine-Hugoniot : $\lambda = \frac{f(\rho^+) - f(\rho^-)}{\rho^+ - \rho^-}$

Présentation des résultats : $\rho(x,t)$ sur $[0,1]^2$:

1ère implémentation des conditions initiales :



2ème implémentation des conditions initiales:

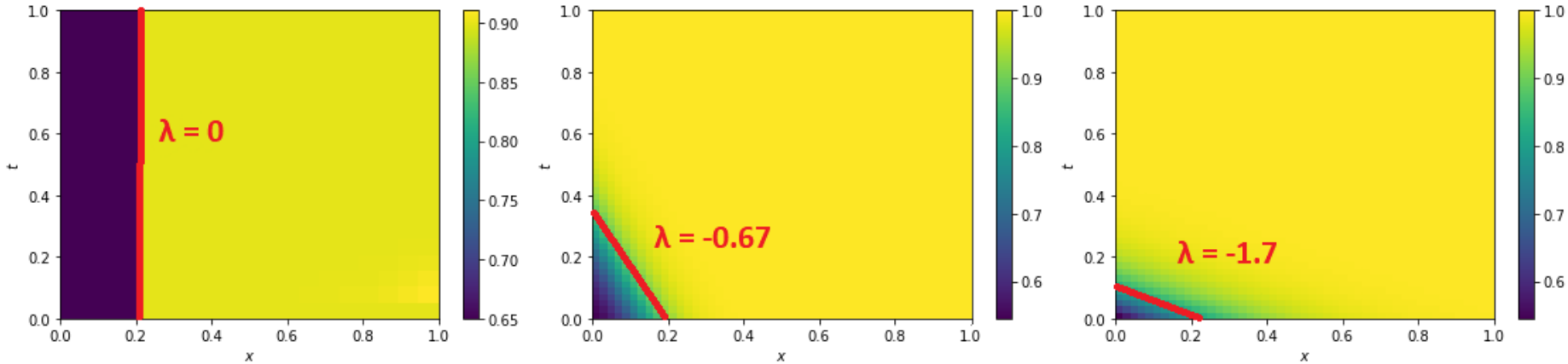


Evaluation des résultats

- Condition de Rankine-Hugoniot : $\lambda = \frac{f(\rho^+) - f(\rho^-)}{\rho^+ - \rho^-}$

Application numérique :

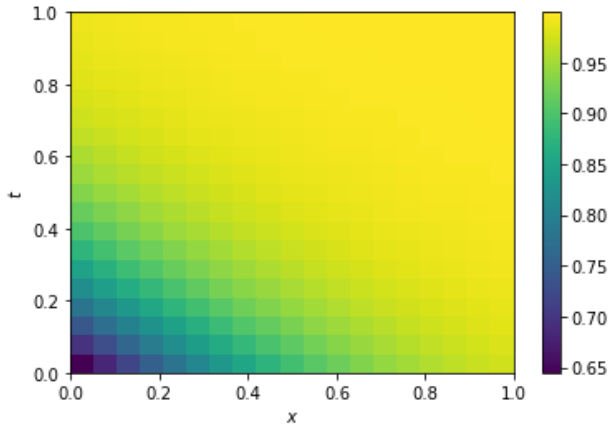
$f(0.65) = 0.27$ *et* $f(0.9) = 0.095 \Rightarrow \lambda = -0.70$



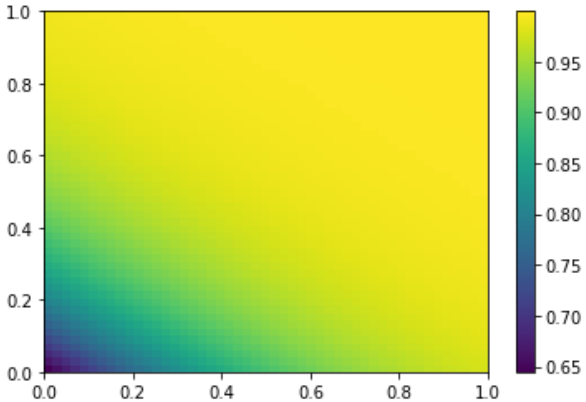
Principe de généralisation

- Intéret théorique: pas besoin de discrétiser
- Intéret pratique : le résultat obtenu est bien une fonction et non le graphe de celle-ci.

Exemple de généralisation 1



Entrainement sur un quadrillage de $[0,1]^2$avec 20*20 points



Résultat sur un quadrillage de $[0,1]^2$avec 50*50 points

Complexité temporelle

On note :

- N le nombre de périodes d'apprentissage
- k la somme du nombre de poids et du nombre de biais
- i le cardinal de l'ensemble I contenant les points utilisés pour une période d'apprentissage

La complexité temporelle de l'algorithme est alors en $O(Nki)$

Justification de la complexité temporelle

Algorithmic 2 pseudocode

```
1: Mise en place du réseau
2: Initialisation des poids et biais
3: N = nombre de périodes d'apprentissage
4: Définition de la fonction coût
5: for j = 1 to N do
6:     I = Ensemble des points d'apprentissage
7:     φ = 0
8:     for i in I do
9:         φ += coût(i)
10:    end for
11:    Calcul des dérivées partielles de φ
12:    Desente du gradient
13: end for
```

Paralélisation des calculs

La partie la plus couteuse en temps est le calcul du coût total et de ses dérivées partielles par rapport aux poids et biais.

Solution : Décomposition du domaine d'apprentissage en sous domaine, chacun traité par un processeur différent.

On note H le nombre de processeurs utilisés.

La nouvelle complexité temporelle est en $O(\frac{Nik}{H})$.

Dépendance aux hyper-paramètres

Exemples d'hyper-paramètres : n nombre de couches du réseau, k nombre de neurones par couche, N nombre de périodes d'apprentissage, v vitesse d'apprentissage, initialisation des poids etc...

- Recalibrage des hyper-paramètres à chaque fois que l'on change d'équation.
- L'aléatoire de la réussite des apprentissages dépend aussi de ses hyper-paramètres.

Conclusion

Avantage :

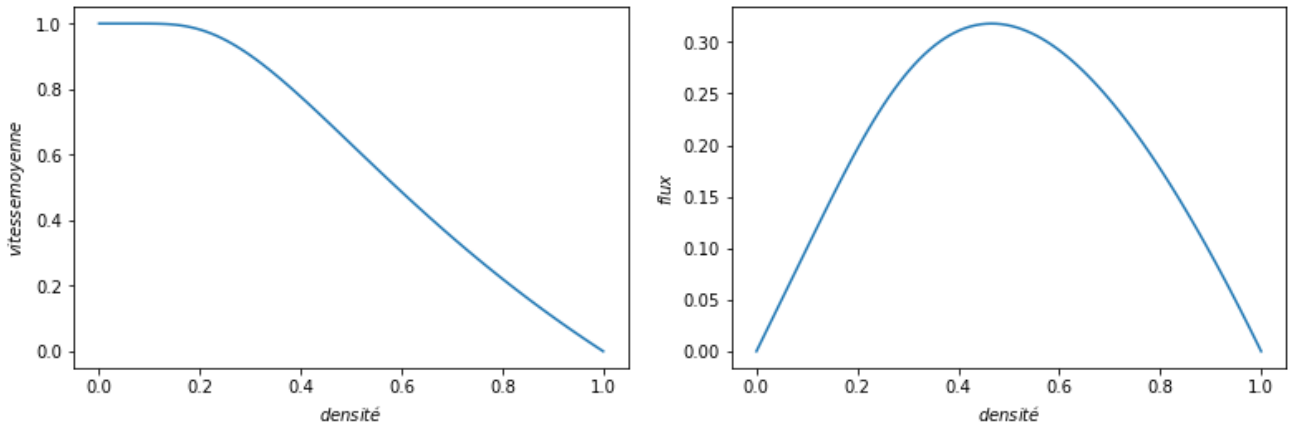
- Temps de calcul
- Résultat facile à manipuler.
- Nombreuses améliorations possibles.

Inconvénient :

- Difficile à implémenter en pratique

Remarques :

- $f \in C^\infty]0,1]$ (*et prolongeable continument en* 0)
- $\forall x \in]0,1]\ f''(x) = \frac{-1}{x^3} < 0 \Rightarrow$ *f est strictement concave*



Comparaison au méthode classique

Exemple de méthode par discrétisation : méthode des éléments finis

Temps de calcul (en seconde) selon la dimension n d'une solution sur $[0,1]^n$avec une méthode de discrétisation :

Dimension :	1	2	10	100
Pas : 10^{-1}	10^{-8}	10^{-7}	10	10^{91}
Pas : 10^{-3}	10^{-6}	10^{-3}	10^{21}	10^{291}
Pas : 10^{-5}	10^{-4}	10	10^{41}	10^{491}

(pour un processeur à 1Ghz)

Tensorflow

- Code plus lisible, plus flexible
- Temps de calcul significativement amélioré

Plan d'un code utilisant Tensorflow :

- Construction de la structure d'un graphe
- Envoi de données dans le graphe

Condition de Rankine-Hugoniot : Preuve

Soit ρ une solution faible admettant une discontinuité.

- $(X(t), t)$ la courbe sur laquelle se situe la discontinuité, $(\lambda(t), 1)$ sa dérivée.
- $\Omega = [0,1]^2$; $\Omega^+ = [0,1] \times [X(t), 1]$; $\Omega^- = [0,1] \times [0, X(t)]$
- $\rho_{|\Omega^+} = \rho^+$; $\rho_{|\Omega^-} = \rho^-$

Soit $\varphi \in C^1(\mathbb{R}\times\mathbb{R}^+, \mathbb{R})$ à support compact telle que
$\forall x \in \mathbb{R}\, \varphi(x,0) = 0$, alors $\int\int_\Omega (\rho\frac{\partial\varphi}{\partial t} + f(\rho)\frac{\partial\varphi}{\partial x})dxdt = 0$

$$\Rightarrow \int\int_{\Omega^+} (\rho\frac{\partial\varphi}{\partial t} + f(\rho)\frac{\partial\varphi}{\partial x})dxdt + \int\int_{\Omega^-} (\rho\frac{\partial\varphi}{\partial t} + f(\rho)\frac{\partial\varphi}{\partial x})dxdt = 0$$

Condition de Rankine-Hugoniot : Preuve

Theorem

(1) $div(\varphi \overrightarrow{v}) = \varphi div(\overrightarrow{v}) + \overrightarrow{v}.\overrightarrow{grad}(\varphi)$
(2) $\iiint_\Omega div(\overrightarrow{v})dV = \iint_{\partial\Omega} \overrightarrow{v}.\overrightarrow{ds}$ *avec* $\overrightarrow{ds}$ *un vecteur unitaire normal à la frontière dirigé vers l'extérieur.*

$$\begin{aligned}
\int\int_{\Omega^-} \rho\frac{\partial\varphi}{\partial t}dxdt &= \int\int_{\Omega^-} \rho div(0,\varphi)dxdt \\
&= \int\int_{\Omega^-} div(\rho(0,\varphi))dxdt - \int\int_{\Omega^-} (0,\varphi).\overrightarrow{grad}(\rho)dxdt \quad (1) \\
&= \int_{\partial\Omega^-} \rho(0,\varphi).\overrightarrow{dv} - \int\int_{\Omega^-} (0,\varphi).\overrightarrow{grad}(\rho)dxdt \quad (2) \\
&\text{avec } \overrightarrow{dv} = \alpha(1,-\lambda)dv \text{ et } \alpha = (1+\lambda^2)^{-\frac{1}{2}} \\
&= -\int_{\partial\Omega^-} \alpha\rho\varphi\lambda(t)dv - \int\int_{\Omega^-} \frac{\partial\rho}{\partial t}\varphi dxdt
\end{aligned}$$

Condition de Rankine-Hugoniot : Preuve

Ainsi $\int \int_{\Omega^-} (\rho \frac{\partial \varphi}{\partial t} + f(\rho) \frac{\partial \varphi}{\partial x}) dxdt =$
$- \int \int_{\Omega^-} \varphi (\frac{\partial \rho^-}{\partial t} + \frac{\partial f \rho^-}{\partial x}) dxdt + \int_{\partial \Omega^-} (-\rho^- \varphi \lambda(t) + f(\rho^-)\varphi) \alpha dv$

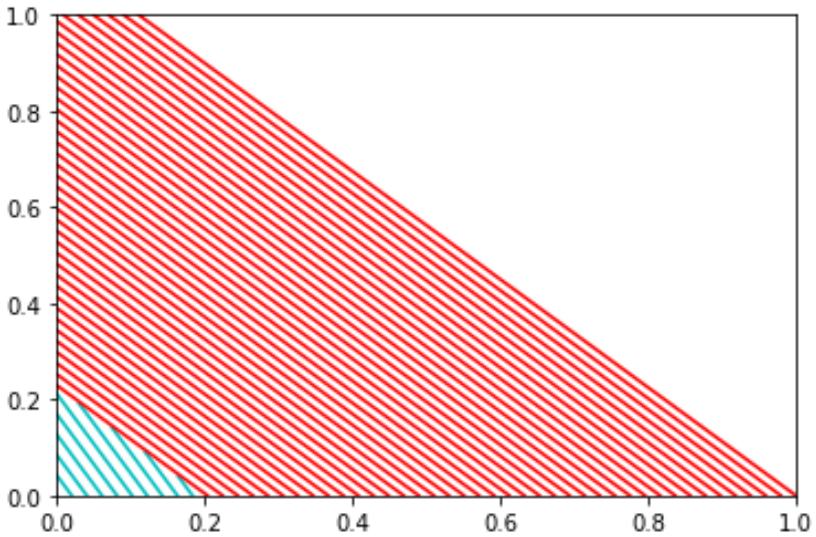
De même, $\int \int_{\Omega^+} (\rho \frac{\partial \varphi}{\partial t} + f(\rho) \frac{\partial \varphi}{\partial x}) dxdt =$
$- \int \int_{\Omega^+} \varphi (\frac{\partial \rho^+}{\partial t} + \frac{\partial f \rho^+}{\partial x}) dxdt + \int_{\partial \Omega^-} (-\rho^+ \varphi \lambda(t) + f(\rho^+)\varphi) \alpha dv$

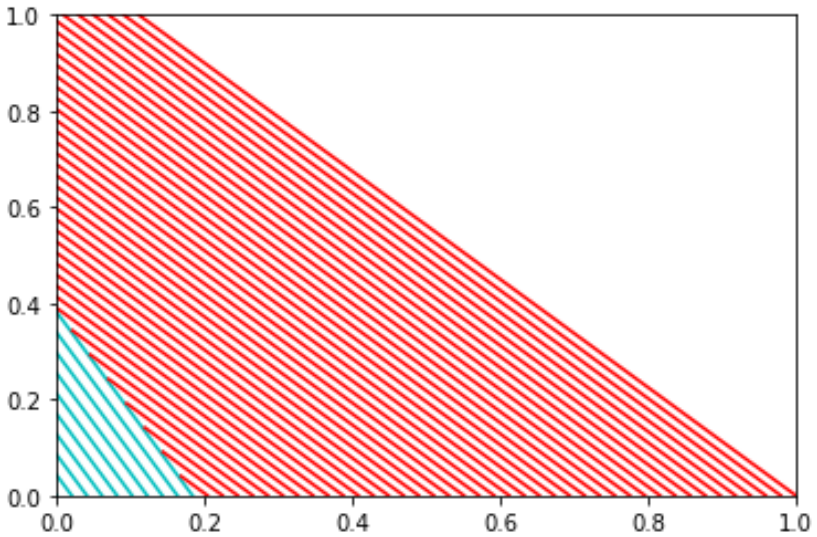
$$\Rightarrow - \int_{\partial \Omega^-)} (-\rho^+ \lambda(t) + f(\rho^+)) \varphi \alpha dv + \int_{\partial \Omega^-} (-\rho^- \lambda(t) + f(\rho^-)) \varphi \alpha dv = 0$$

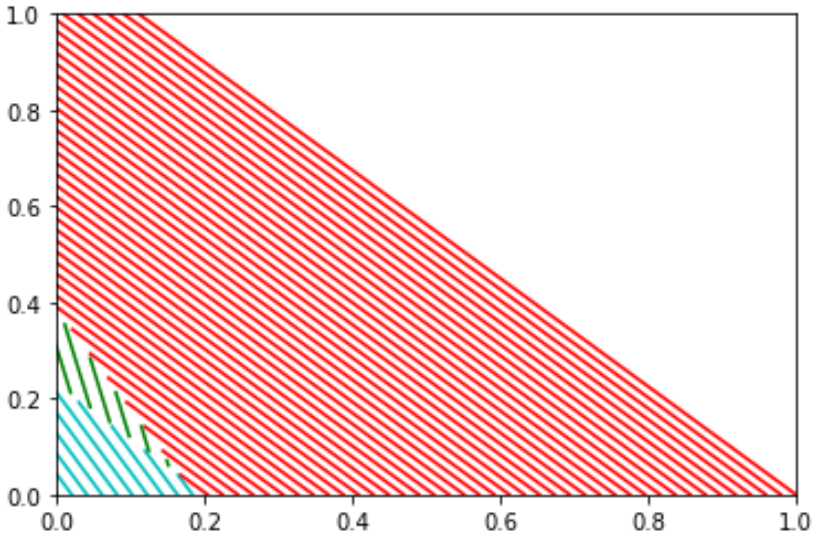
$$\Rightarrow \int_{\partial \Omega^-} ((\rho^+ - \rho^-)\lambda(t) + (f(\rho^-) - f(\rho^+))) \alpha \varphi dv = 0$$

$$\Rightarrow (\lambda(t)(\rho^+ - \rho^-) + (f(\rho^-) - f(\rho^+)) = 0$$

Lax entropy condition : notion d'entropie







Lax entropy condition : la condition

Theorem

Condition d'entropie de Lax : $f'(\rho^-) > \lambda > f'(\rho^+)$

Rappel :

- L'équation d'une caractéristique x partant de x_0 pour $t = 0$ est $x(t) = f'(\rho(x_0, 0)) \cdot t + x_0$
- Le flux f est strictement concave.

Remarque :

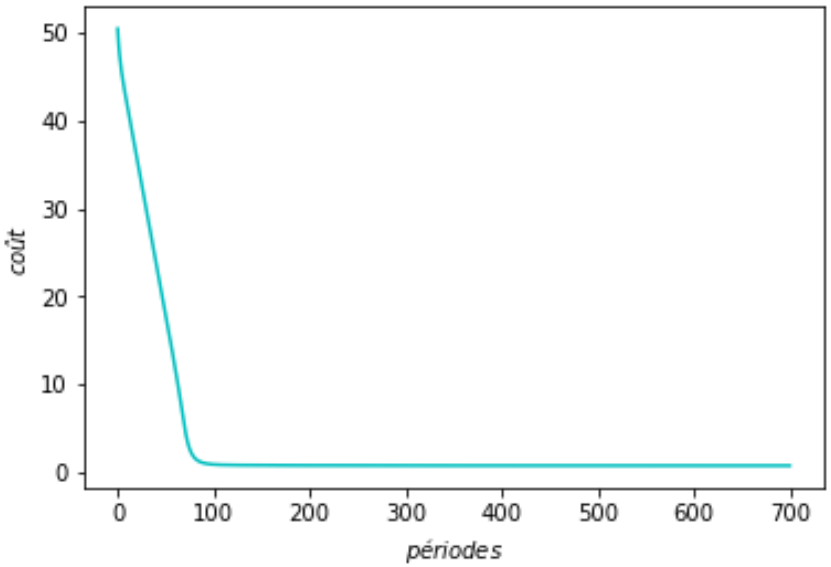
$f'(\rho^-) > f'(\rho^+) \Leftrightarrow \rho^- < \rho^+$

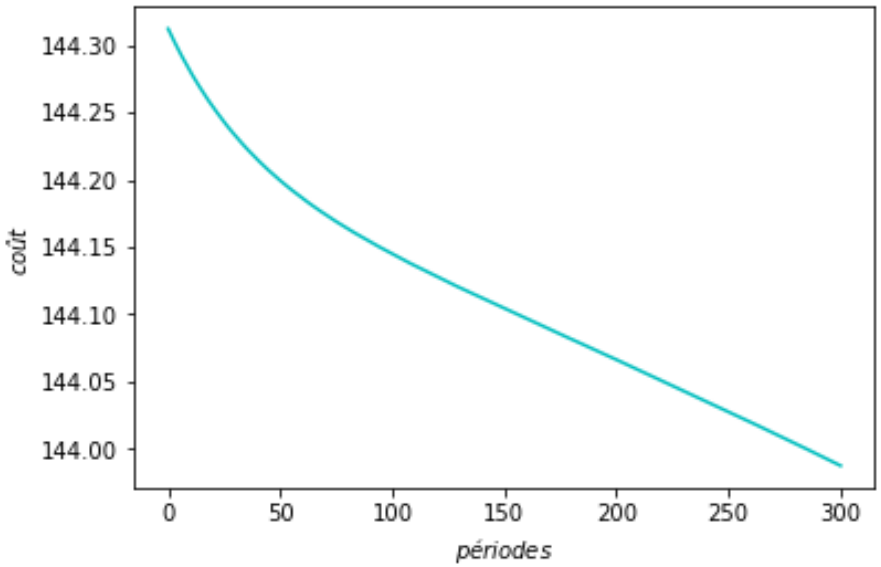
Pseudo-code après parallélisation

Algorithmic 3 pseudocode

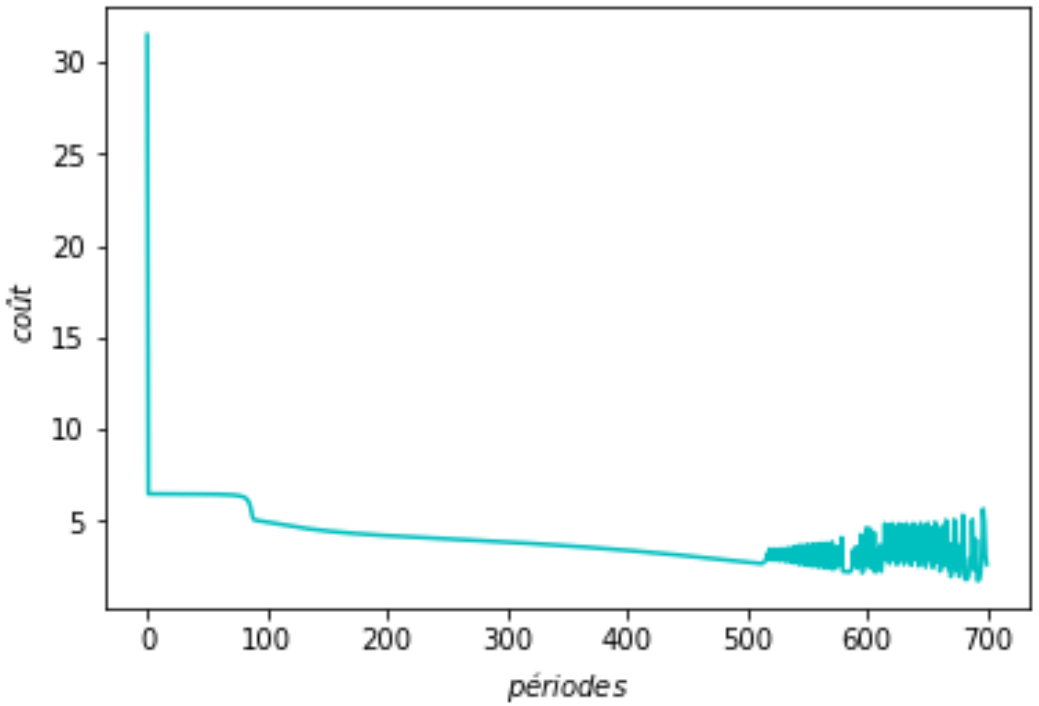
```
1: Mise en place du réseau
2: Initialisation des poids et biais
3: N = nombre de périodes d'apprentissage
4: Définition de la fonction coût
5: H = nombre de processeurs utilisés
6: for j = 1 to N do
7:     for h = 1 to H do
8:         Ih = Ensemble des points d'apprentissage pour le processeur h
9:     end for
10:    φ = 0
11:    for h = 1 to H do {en parallèle}
12:        for i in Ih do
13:            φ += coût(i)
14:        end for
15:        Calcul des dérivées partielles de φ
16:        Desente du gradient
17:    end for
```

Difficulté : nombre de périodes d'apprentissage

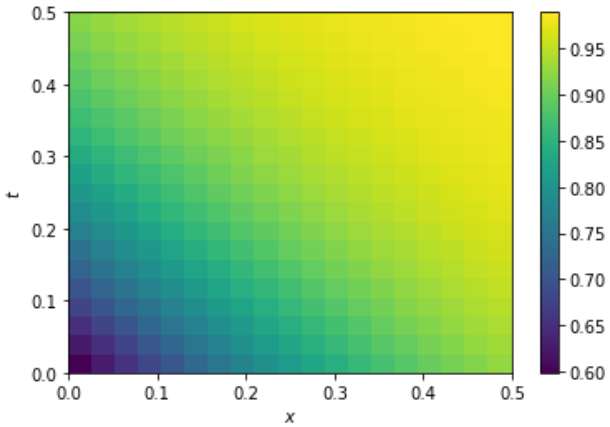




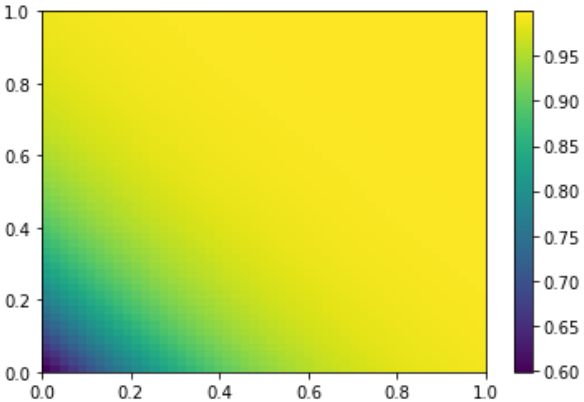
Difficulté : vitesse d'apprentissage



Autre exemple de généralisation



Entrainement sur un quadrillage de $[0, 0.5]^2$avec 20*20 points



Résultat sur un quadrillage de $[0, 1]^2$avec 50*50 points

Code

```python
# -*- coding: utf-8 -*-
import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np

def B(x): # Condition initiale
    cond = tf.cast(x <= 0.2, tf.float32)
    return (0.65 * cond) + ((1-cond) * 0.9)


def df(x): # Dérivé du flux
    return 1-tf.exp(1-1/x) + (-1/(x))*tf.exp(1 - 1/x)

def Init(x): # permet de séparer les points qui sont contraints par les conditions initiales
    return (tf.cond(x <= 0, lambda: [1.], lambda:[0.]))


# Définition de l'espace :
nx = 20
ny = 20

x_space = (np.linspace(0., 1., nx))
y_space = (np.linspace(0., 1., ny))


# Neurones par couche ( on a 2 couches) :
n_nodes_hl1 = 20
n_nodes_hl2 = 20


x = tf.placeholder('float', [None, 2])
```

Code suite

```python
def neural_network_model(data, grad_coef=10): # Initialisation du réseau
    # Mise en place du réseau :
    hidden_1_layer = {'weights': tf.Variable(tf.random_normal([2, n_nodes_hl1])),
                      'biases': tf.Variable(tf.random_normal([n_nodes_hl1]))}

    hidden_2_layer = {'weights': tf.Variable(tf.random_normal([n_nodes_hl1, n_nodes_hl2])),
                      'biases': tf.Variable(tf.random_normal([n_nodes_hl2]))}

    output_layer = {'weights': tf.Variable(tf.random_normal([n_nodes_hl2, 1])),
                    'biases': tf.Variable(tf.random_normal([1]))}

    l1 = tf.add(tf.matmul(data, hidden_1_layer['weights']),hidden_1_layer['biases'])
    l2 = tf.add(tf.matmul(l1, hidden_2_layer['weights']),hidden_2_layer['biases'])
    output = tf.add(tf.matmul(l2, output_layer['weights']),output_layer['biases'])

    output = tf.sigmoid(output)

    trial = output

    # Implémentation du coût :
    d = tf.gradients(trial, data)[0]
    cost = tf.add(tf.multiply(d[:, 0], f(trial)), d[:, 1])
    cost = tf.square(cost)
    cost = tf.reduce_mean(cost)

    # Contrainte des conditions initiales :
    additional_cost =[tf.reduce_sum(tf.multiply(tf.transpose([tf.map_fn(aux,data[:,1])]),(tf.square(trial-B(tf.transpose([data[:,0]])))))]

    # Coût final :
    cost += grad_coef * additional_cost

    return [trial,cost]


def train_neural_network(x): # Apprentissage

    net_out, cst= neural_network_model(x)
    optimizer = tf.train.GradientDescentOptimizer(0.0001).minimize(cst) # Choix de l'algorithme d'optimisation
    hm_epochs = 1001 #Nombre de période d'apprentissage

    fd = {x: [[xi, yi]
          for xi in x_space for yi in y_space]}

    with tf.Session() as sess: # Boucle d'apprentissage
        sess.run(tf.global_variables_initializer())
        for epoch in range(hm_epochs):
            _, epoch_loss = sess.run([optimizer, cst],feed_dict=fd)
            print(sum(epoch_loss))
```

Code dérivées à la main

```python
import numpy as np
from matplotlib import pyplot as plt
nx = 10
ny = 10
dx = 1. / nx
dy = 1. / ny
x_space = np.linspace(0, 1, nx)
y_space = np.linspace(0, 1, ny)

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

def d_sigmoid(x):
    return np.exp(-x) / ((1 + np.exp(-x))**2)

def d2_sigmoid(x):
    return ((-np.exp(-x) / ((1 + np.exp(-x))**2)) + (2*np.exp(-2*x) / ((1+ np.exp(-x))**3)))

def carre(x):
    return x**2

def produit(x,y): #Fait le produit terme à terme de deux vecteurs
    k = []
    for i in range(len(x)):
        k.append(x[i]*y[i])
    return np.asarray(k)

def f(x):
    return x*(1-np.exp(1-1/x))

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

def neural_network(W, x):
    a1 = sigmoid(np.dot(x, W[0]))
    return np.dot(a1, W[1])

#Les dérivées premières et seconde du réseau de neurones pour les variables de l'espace x et y.
def d2x_neural_network(W, x):
    a1 = d2_sigmoid(np.dot(x, W[0]))
    a1 = produit(a1, np.apply_along_axis(carre,0,W[0][0]))
    return np.dot(a1, W[1])

def d2y_neural_network(W, x):
    a1 = d2_sigmoid(np.dot(x, W[0]))
    a1 = produit(a1, np.apply_along_axis(carre,0,W[0][1]))
    return np.dot(a1, W[1])

def dx_neural_network(W,x):
    a1 = d_sigmoid(np.dot(x, W[0]))
    a1 = produit(a1, W[0][0])
    return np.dot(a1, W[1])

def dy_neural_network(W,x):
    a1 = d_sigmoid(np.dot(x, W[0]))
    a1 = produit(a1, W[0][1])
    return np.dot(a1, W[1])

def dvi_err(W,x,e,net_out,dx): #La dérivé du coût par rapport aux poids de la deuxième couche
    l = np.zeros((10,1))
    for i,vi in enumerate(W[1]):
        de = sigmoid(np.dot(x, W[0]))[i] + x[1]*W[0][1][i]*d_sigmoid(np.dot(x, W[0]))[i] + \
        (x[1]*W[0][0][i]*d_sigmoid(np.dot(x, W[0]))[i])*(df(psy_trial(x,net_out))) + \
        (x[1]*(dx))*((x[1])* (sigmoid(np.dot(x, W[0]))[i])*(1 - np.exp(1-1/(psy_trial(x,net_out))))\
        - psy_trial(x,net_out) * ((x[1])*(sigmoid(np.dot(x, W[0]))[i])) * (1/(psy_trial(x,net_out)**2))*np.exp(1-1/psy_trial(x,net_out)))
        dE = 2*de*e
        l[i] = dE
    return l
```

Code dérivées à la main suite

```
69
70 def dwij_err(W,x,e,net_out,dx): #La dérivé du coût par rapport aux poids de la première couche
71     l = np.zeros((2,10))
72     for i, w0i in enumerate(W[0][0]):
73         de = x[0]*W[1][i]*d_sigmoid(np.dot(x, W[0]))[i] + x[1]*(x[0]*W[1][i]*W[0][1][i]*d2_sigmoid(np.dot(x, W[0]))[i])\
74         + x[1]*(x[0]*W[1][i]*W[0][0][i]*d2_sigmoid(np.dot(x, W[0]))[i] + W[1][i]*d_sigmoid(np.dot(x, W[0]))[i])*f(psy_trial(x,net_out))\
75         + x[1]*(dx)*x[1]*x[0]* W[1][i]*d_sigmoid(np.dot(x, W[0]))[i]*(1 - np.exp(1-1/psy_trial(x,net_out)) - \
76         psy_trial(x,net_out)*(1/(psy_trial(x,net_out)**2))*np.exp(1-1/psy_trial(x,net_out)))
77         dE = 2*de*e
78         l[0][i] = dE
79
80     for i, w0i in enumerate(W[0][1]):
81         de = x[1]*W[1][i]*d_sigmoid(np.dot(x, W[0]))[i] + x[1]*(x[1]*W[1][i]*W[0][1][i]*d2_sigmoid(np.dot(x, W[0]))[i]\
82         + W[1][i]*sigmoid(np.dot(x, W[0]))[i] ) + x[1]*(x[1]*W[1][i]*W[0][0][i]*d2_sigmoid(np.dot(x, W[0]))[i])*f(psy_trial(x,net_out))\
83         + x[1]*(dx)*x[1]*x[1]* W[1][i]*d_sigmoid(np.dot(x, W[0]))[i]*(1 - np.exp(1-1/psy_trial(x,net_out)) -\
84         psy_trial(x,net_out)*(1/(psy_trial(x,net_out)**2))*np.exp(1-1/psy_trial(x,net_out)))
85         dE = 2*de*e
86         l[1][i] = dE
87     return l
88
89 def A(x): #Condition initiale
90     if x[0] <= 0.2: a = 0.65
91     else: a = 0.9
92     return a
93
94 def NN_trial(x, net_out): #Implémentation des conditions initiales
95     return A(x) + x[1]*net_out
96
97 def loss_function(W, x, y): # Calcul du coût et des dérivées partielles de celui-ci
98     loss_sum = 0.
99     grad = [np.zeros((2,10)), np.zeros((10,1))]
100    for xi in x:
101        for yi in y:
102            input_point = np.array([xi, yi])
103            net_out = neural_network(W, input_point)[0]
104            dx = dx_neural_network(W, input_point)[0]
105            dy = dy_neural_network(W, input_point)[0]
106
107            gradient_of_trial_dx = yi*(dx)
108            gradient_of_trial_dy = net_out + yi*dy
109
110            cost = (gradient_of_trial_dx*df(net_out) + gradient_of_trial_dy)
111
112          grad[0] += dwij_err(W,input_point,cost,net_out,dx)
113            grad[1] += dvi_err(W,input_point,cost,net_out,dx)
114
115            cost = ((gradient_of_trial_dx*df(net_out) + gradient_of_trial_dy))**2
116            loss_sum += cost
117
118    return grad
119
120 def apprentissage(W,vitesse): # Descente du gradient
121     for i in range(100):
122         loss_grad =  loss_function(W,x_space,y_space)
123
124         W[0] = W[0] - vitesse * loss_grad[0]
125         W[1] = W[1] - vitesse * loss_grad[1]
126
127 W = [np.random.random((2, 10)), np.random.random((10, 1))]
```