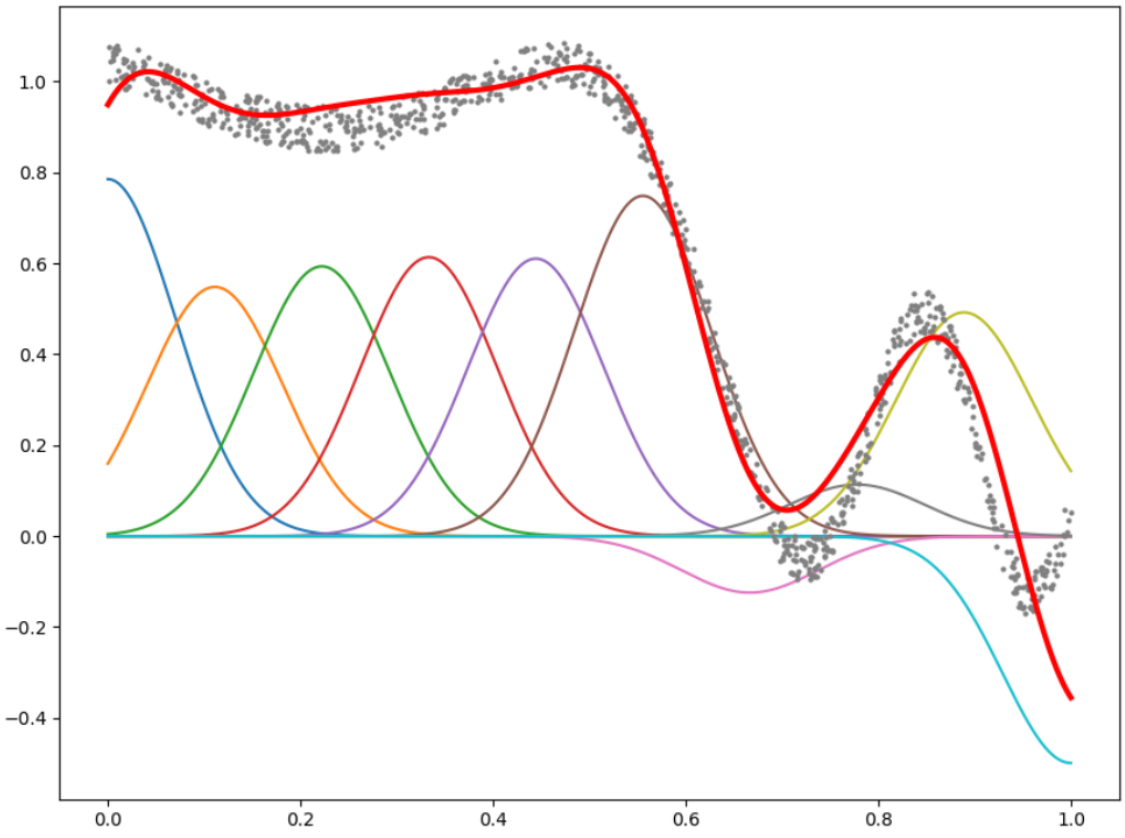
Robotique et apprentissage, TME 1

ABITBOL Ethan, 3804139 et DUFOURMANTELLE Jeremy, 21104331

February 2022

REGRESSION



Sommaire

1 Introduction **3**

2 Batch Linear Least squares - Moindres carrés linéaire **3**
2.1 Study part : 3

3 Ridge Regression **5**
3.1 Study part : 5

4 Batch RBFNs (Radial Basis Function Networks) : **6**
4.1 Première perspective : 6
4.2 Deuxième perspective: 7
4.3 Study part : 7
4.3.1 Première partie de réponse 7
4.3.2 Deuxième partie de réponse 8

5 Incremental RBFNs **10**
5.1 Study part 1 : 10
5.2 Study part 2 : 12

6 Locally Weighted Regression **13**
6.1 Study part : 13

7 Regression with Neural Networks (in pytorch) **14**
7.1 Study part 1: 14
7.2 Study part 2: 14
7.3 Study part 3: 15

1 Introduction

Dans ce TME, nous allons étudier les algorithmes basiques de la régression linéaire et non linéaire. L'objective de la régression ou des fonctions d'approximations est de créer un modèle de données observées. Le modèle a une structure fixe avec des paramètres et la régression consiste a ajuste ces paramètres pour s'adapter aux données.
Ce document va répondre aux différentes questions posées dans le fichier notebook fournis avec, 'DUFOURMANTELLE_ABITBOL_regression_lab_2022.ipynb'.

2 Batch Linear Least squares - Moindres carrés linéaire

Pour cette partie, on utilise la fonction suivante :

$$\theta^* = (\bar{\mathbf{X}}^\intercal \bar{\mathbf{X}})^{-1} \bar{\mathbf{X}}^\intercal \mathbf{y}.$$

avec y un vecteur de dimension N et $\bar{\mathbf{X}} = \begin{pmatrix} \mathbf{x}_{1,1} & \mathbf{x}_{1,2} & \cdots & \mathbf{x}_{1,d} & 1 \\ \mathbf{x}_{2,1} & \mathbf{x}_{2,2} & \cdots & \mathbf{x}_{2,d} & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \mathbf{x}_{N,1} & \mathbf{x}_{N,2} & \cdots & \mathbf{x}_{N,d} & 1 \end{pmatrix}$. la design matrice.

on a donc implémenté la partie Train de cette maniére :

```
def train(self, x_data, y_data):
    # Finds the Least Square optimal weights
    # Fill this part
    x = bar_design(x_data) # matrice avec colonne de 1
    self.theta = np.dot(np.linalg.inv(( np.dot(x.T , x) )),np.dot(x.T,y_data))
```

On transforme dabord les x_data en design matrice et ensuite on applique le fonction pour obtenir le modéle optimal.

2.1 Study part :

Does your train(self, x_data, y_data) function provide exactly the same results as the train_from_stats(self, x_data, y_data) function? Call both functions 2000 times. Is one faster than the other?

Pour répondre a cette question, nous avons implémenté la fonction suivante :

```
import pandas as pd

def comparaison():
    liste_theta = []
    liste_time = []

    for i in range(2000):
        batch = Batch()
        size = 50
        batch.make_linear_batch_data(size)
        model = LinearModel(size)

        x = np.array(batch.x_data)
        y = np.array(batch.y_data)

        start1 = time.time()
        model.train(x, y)
        theta11 = round(model.theta[0],4)
```

```
        theta12 = round(model.theta[1],4)
        end1 = time.time()
        time1 = end1 - start1

        start2 = time.time()
        model.train_from_stats(x, y)
        theta21 = round(model.theta[0],4)
        theta22 = round(model.theta[1],4)
        time2 = time.time() - start2

        if theta11== theta21 and theta12 == theta22 :
            liste_theta.append(0)
        else:
            liste_theta.append(1)

        if round(time1,10) > round(time2,10) :
            liste_time.append(1)
        else :
            liste_time.append(2)
    return liste_theta, liste_time
```

Cette fonction execute 2000 iterations et a chaque it eration on calcul les thetas des deux fonctions et leur temps d'ex ecutions. Ensuite, on compare theta et si on obtient le mˆeme theta on append 0 a la liste theta sinon 1. De même pour le temps, si cest la fonction train qui est plus rapide on append 1 sinon si c'est la fonction train_from_stats on append 2. Enfin on retourne les deux listes liste_time et liste_theta.

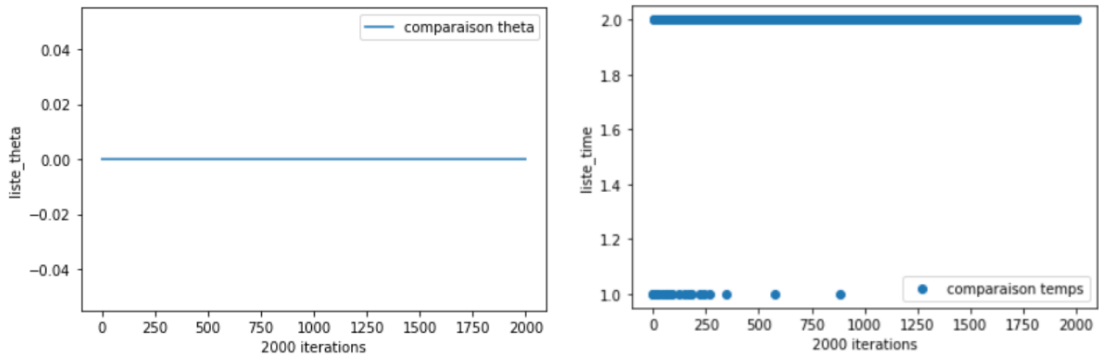
Pour observer le resultat on a donc implémenté une fonction affichage qui permet d'observer des figures du nombred'itérations en fonction de liste_theta et liste_time.

```
def affichage(liste_theta, liste_time):
    plt.figure(1)
    plt.plot(np.arange(2000),np.array(liste_theta), label = 'comparaison theta')
    plt.xlabel('2000 iterations')
    plt.ylabel('liste_theta')
    plt.legend()
    plt.show()
    plt.figure(2)
    plt.scatter(np.arange(2000),np.array(liste_time), label = 'comparaison temps')
    plt.xlabel('2000 iterations')
    plt.ylabel('liste_time')
    plt.legend()
    plt.show()
```

En exécutant les trois lignes suivantes:

```
liste_theta , liste_time = comparaison()
print(pd.Series(liste_time).value_counts())
affichage(liste_theta ,liste_time)
```

On obtient les graphiques suivant :



A gauche on observe une ligne droite constante égale a 0 qui signifie que on obtient le même theta pour les deux fonctions mais a droite on voit bien que le nombre de points à 2 est très important

(1900/2000) 90% correspondant au temps obtenu avec la deuxieme fonction (train_from_stats) et environ 100 fois avec la première fonction (train). Ce qui signifie que nos calculs sont corrects mais que la fonction implémenté train est moins rapide.

3 Ridge Regression

Dans cette partie on a implémenté la fonction suivante :

$$\boldsymbol{\theta}^* = (\lambda \boldsymbol{I} + \mathbf{X}^\intercal \mathbf{X})^{-1} \mathbf{X}^\intercal \mathbf{y}$$

avec lambda un coef, I la matrice identité, X la design matrix et y un vecteur, de la manière suivante :

```
def train_regularized(self, x_data, y_data, coef):
    # Finds the regularized Least Square optimal weights
    x = bar_design(x_data) # design matrice
    I = np.identity(x.shape[1]) # matrice id ntit  , x.shape[1] car on
    self.theta = np.dot(np.linalg.inv((coef * I + np.dot(x.T,x))),np.dot(x.T,y_data))
```

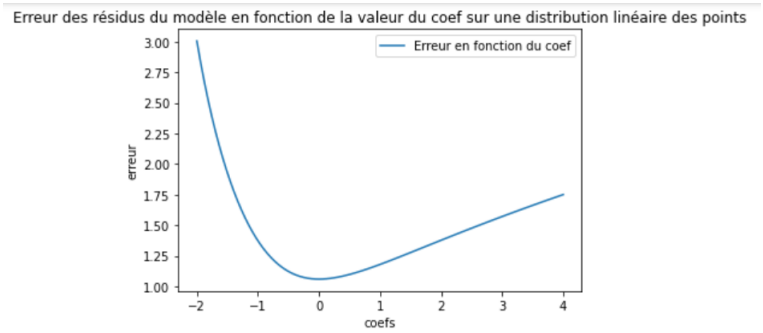
3.1 Study part :

For a batch of 50 points, study with the train_regularized(self,x_data,y_data,coef) function how the residuals degrade as you increase the value of coef. A good idea would be to make a picture with coef in the x axis and the residuals in the y axis, and then to comment it.

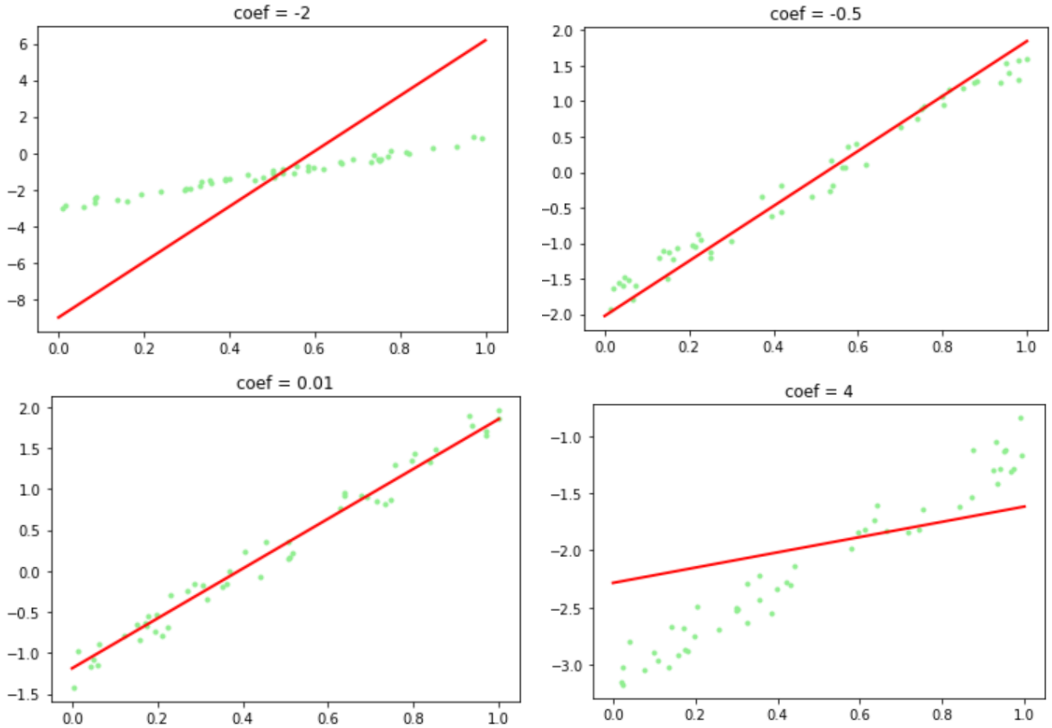
Pour cela, on a codé la fonction 'degradationLineaire()' qui permet de faire 100 itérations sur 100 coefs différents entre -2 et 4. A chaque itération, on calcul l'erreur en fonction du coef et une fois la boucle terminé on renvoi l'affichage sous forme de scatter de l'erreur en fonction du coef.

```
def degradationLineaire(affichage = True):
    batch = Batch()
    size = 50
    batch.make_linear_batch_data(size)
    model = LinearModel(size)
    liste_error = []
    points = np.linspace(-2,4,100)
    for i in points:
        start = time.process_time()
        model.train_regularized(batch.x_data, batch.y_data, coef=i)
        error = model.compute_error(batch.x_data, batch.y_data)
        liste_error.append(error)

    if affichage == True :
        plt.figure()
        plt.plot(points,liste_error, label = "Erreur en fonction du coef")
        plt.xlabel("coefs")
        plt.ylabel("erreur")
        plt.title("Erreur des r sidus du mod le en fonction de la valeur du coef sur une distribution lin aire des points")
        plt.legend()
        plt.show()
    return points, liste_error
```

On obtient donc l'affichage suivant :



On remarque que lorsqu'on a un coef négatif, la somme des résidus est élevé. Plus on se rapproche de zéro, plus l'erreur diminue et la courbe atteint son minimum, cela peut être expliqué du fait que on se retrouve avec la formule de base ($\lambda * I = 0$ et θ^* = formule question 2) . Enfin, lorsqu'on augmente le coef encore plus, l'erreur augmente aussi. Donc le meilleur coefficient se trouve proche de zéro.



4 Batch RBFNs (Radial Basis Function Networks) :

Il y a 2 perspectives a propos de la méthode des batch moindres carrés:

4.1 Première perspective :

Ici, on considère que le RBFN est utilisé pour projeté d'un espace d'entrée vers un espace de fonctionnalités.

$$\boldsymbol{\theta}^* = (\mathbf{G}^\intercal \mathbf{G})^{-1} \mathbf{G}^\intercal \mathbf{y}$$

avec G la gram matrix.

```
def train(self, x_data, y_data):
        G = self.phi_output(x_data).T #gram matrix
        self.theta = np.dot(np.linalg.inv(np.dot(G.T,G)),np.dot(G.T,y_data))
```

4.2 Deuxième perspective:

Ici, on effectue l'ensemble du calcul a partir de zero et la solution générale est obtenu sous la forme :

$$\theta = A^{\sharp} b$$

$$\mathbf{A} = \left(\sum_{i=1}^{N} \phi(\mathbf{x}^{(i)}) \phi(\mathbf{x}^{(i)})^{\mathsf{T}} \right) \quad \mathbf{b} = \sum_{i=1}^{N} \phi(\mathbf{x}^{(i)}) y^{(i)}$$

cela donne en python :

```
def train(self, x_data, y_data):
    A = np.zeros(shape=(self.nb_features, self.nb_features))
    b = np.zeros(self.nb_features)

    for i in range(len(x_data)):
        A = A + np.dot(self.phi_output(x_data[i]),self.phi_output(x_data[i]).transpose())
        b = b + (self.phi_output(x_data[i]).T[0] * y_data[i])

    self.theta = np.linalg.solve(A,b)
```

4.3 Study part :

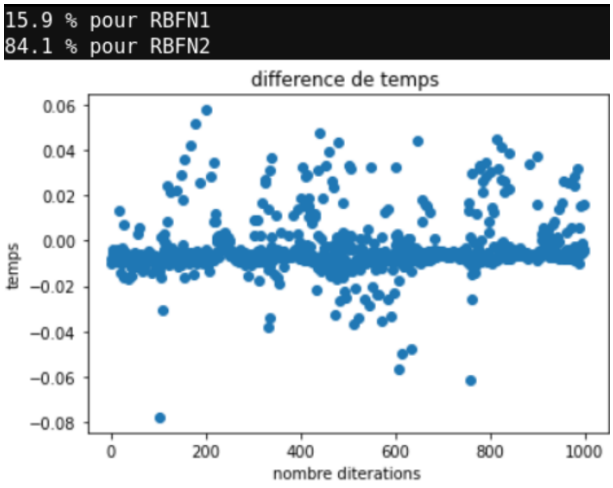
4.3.1 Première partie de réponse

Calling the train functions at least 1000 times, comment on the difference in computation time between the first and the second perspective.
Pour montrer la différence de temps, on a implémenté les fonctions suivantes:

```
def difference_temps(nombre_iter):
    liste_temps = []
    for i in range(nombre_iter):
        batch = Batch()
        batch.make_nonlinear_batch_data(size=60)

        model = BatchRBFN1(nb_features=10)
        start1 = time.process_time()
        model.train(batch.x_data, batch.y_data)
        rbfn1 = time.process_time() - start1

        model = BatchRBFN2(nb_features=10)
        start2 = time.process_time()
        model.train(batch.x_data, batch.y_data)
        rbfn2 = time.process_time() - start2

        liste_temps.append(round(rbfn1,4) - round(rbfn2,4))
    return liste_temps

def rbfn_affichage(nombre_iter, liste_temps):
    plt.figure(1)
    plt.scatter(np.arange(nombre_iter),liste_temps)
    plt.title('difference de temps ')
    plt.xlabel('nombre diterations')
    plt.ylabel('temps')
    plt.show()

nombre_iter = 1000
```

```
liste_temps = difference_temps(nombre_iter)
print(((pd.Series(liste_temps)>0).sum() / 1000)*100,"% pour RBFN1" )
print(((pd.Series(liste_temps)<0).sum() / 1000)*100,"% pour RBFN2" )
rbfn_affichage(nombre_iter , liste_temps)
```

Pour obtenir comme figure, la figure suivante:



Sur cette figure, nous voyons bien que la majorité des points (84,1 %) sont concentrés en dessous de la valeur 0 (RBFN1 - RBFN2 < 0) ce qui indique que la version RBFN1 est plus efficace.

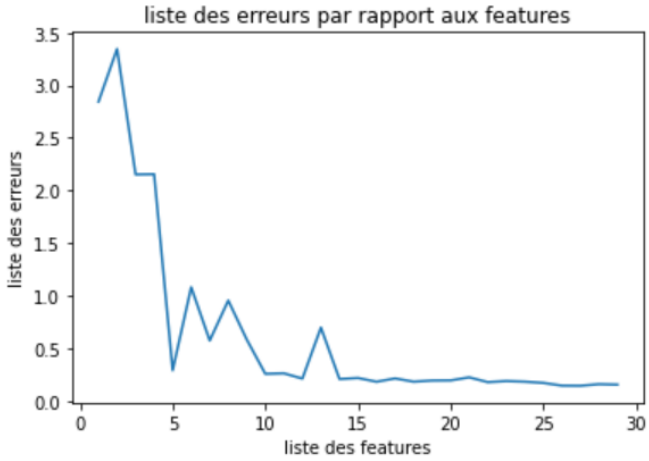
4.3.2 Deuxième partie de réponse

Study the evolution of the error as a function of the number of features. A good idea would be to make a picture with the number of features in the x axis and the residuals in the y axis, and then to comment it. Have a look at the model when the number of features is large (e.g. 30). What is happening?

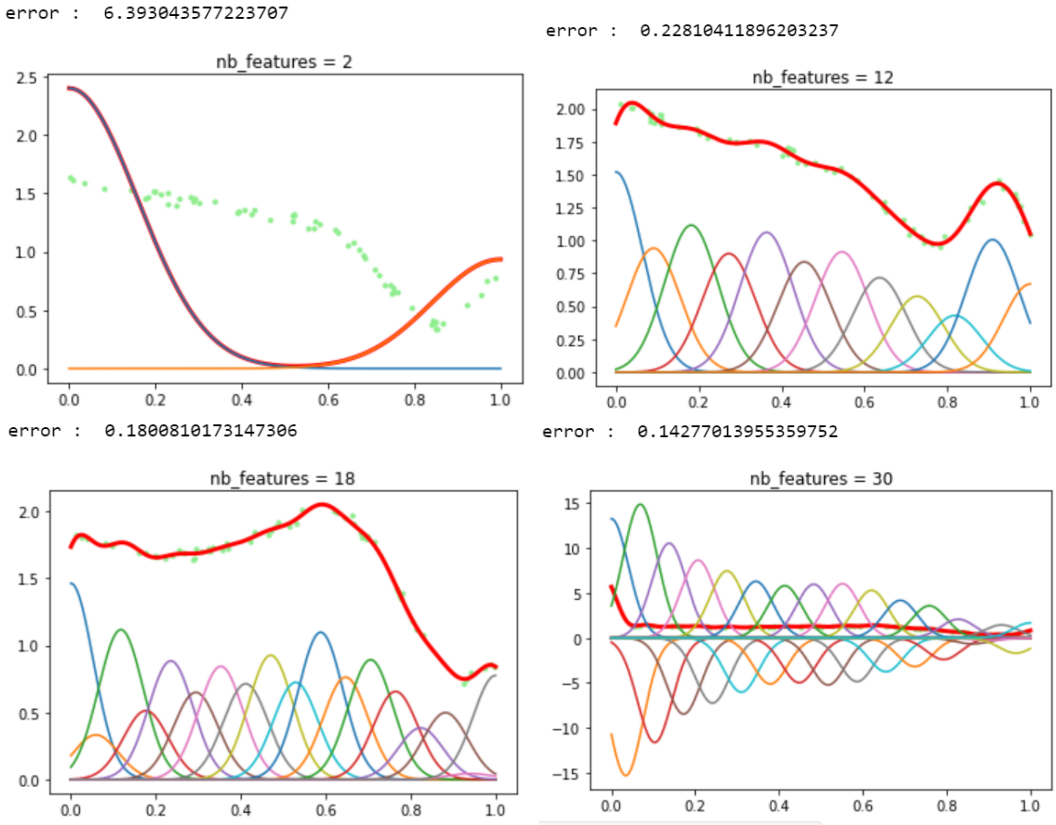
Pour pouvoir étudier l'évolution de l'erreur en fonction des features, on a implémenté la fonction suivante :

```
def feautures_(features = 30):
    liste_error = []
    liste_features = []
    for i in range(1, features):
        liste_features.append(i)
        batch = Batch()
        batch.make_nonlinear_batch_data(size=60)

        model = BatchRBFN2(nb_features=i)
        model.train(batch.x_data, batch.y_data)
        error = model.compute_error(batch.x_data, batch.y_data)
        liste_error.append(error)

    return liste_features ,liste_error

def rbfn_affichage_error(liste_features, liste_error):
    plt.figure()
    plt.plot(liste_features,liste_error)
    plt.xlabel('liste des features')
    plt.ylabel('liste des erreurs')
    plt.title('liste des erreurs par rapport aux features')

features = 30
liste_features , liste_error = feautures_(features)
rbfn_affichage_error(liste_features, liste_error)
```

Lorsqu'on exécute cette fonction, on obtient le graphique suivant :



On remarque que avec peu de features $[1, 4]$ l'erreur est assez élevé mais a partir de 5 on diminue progressivement pour arriver à une erreur constante aux alentours de 0.3, pour un nombre de features supérieurs a 15.
Mais que cela représente t'il sur notre modèle ?



Ces 4 graphiques illustre bien ce qui se passe lorsqu'on fait varier le nombres de features. Sur l'image en haut a gauche, on peut voir que avec un nombre de feature = 2 l'erreur renvoyé est de 6,3, cela est du aux nombre insuffisant de features. Quant aux 3 autres figures, on voit que l'erreur est approximativement la même or les courbes obtenus sont pas du tout similaire. On remarque que pour 12 et 18 features les poids associées aux gaussiennes sont en cohérence avec nos points alors que pour 30 feautures les poids associées aux gaussiennes sont beaucoup plus grande que les points.

Donc, avec peu de features environ entre [0,8], on est en underfitting et avec trop de features, 25 et plus on est en overfitting.Ainsi, le meilleur compromis entre les deux est donc entre [10,20].

5 Incremental RBFNs

Nous allons dans cette partie comparer deux approches incrémentale de l'algorithme RBFN. La premiere approche consiste à faire une descente de gradient (**GDRBFN**) dont la formule est :

$$\theta^{(t+1)} = \theta^{(t)} + \alpha(y^{(t+1)} - \phi(\mathbf{x}^{(t+1)})^\intercal \theta^{(t)})\phi(\mathbf{x}^{(t+1)})$$

et l'autre se base sur la méthode des moindres carrés recursif (**RLSRBFN**) en utilisant cette formule :

$$\theta^{(t+1)} = (\mathbf{A}^{(t+1)})^\sharp b^{(t+1)}$$

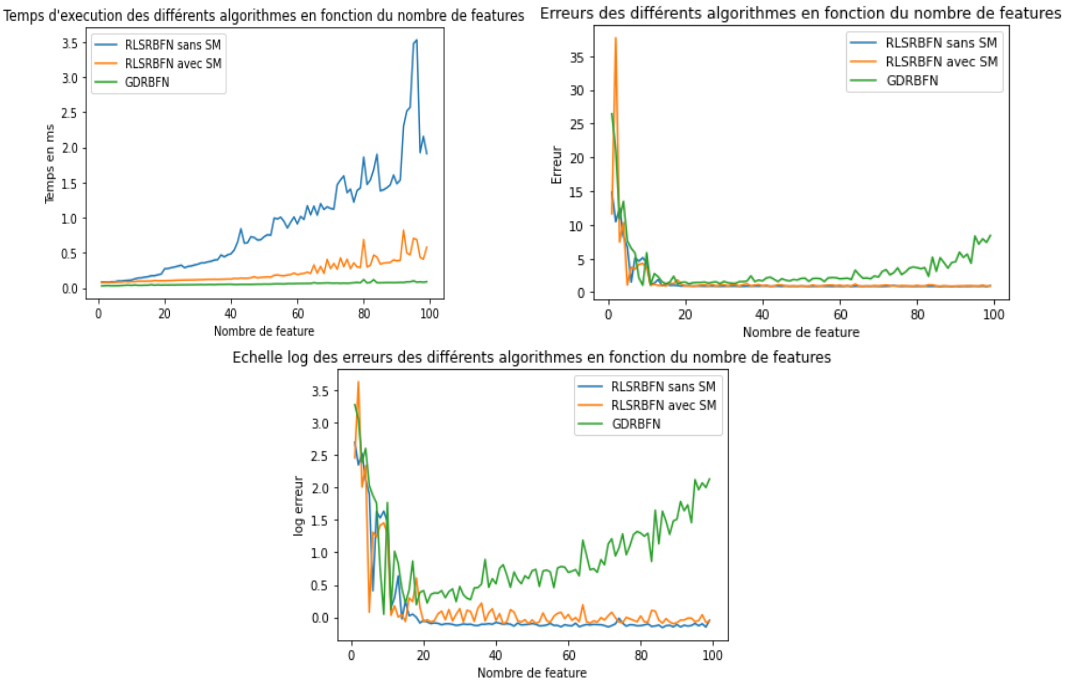
Avec :

$$\mathbf{A}^{(t+1)} = \mathbf{A}^{(t)} + \phi(\mathbf{x}^{(t+1)})\phi(\mathbf{x}^{(t+1)})^\intercal \text{ et } \mathbf{b}^{(t+1)} = \mathbf{b}^{(t)} + \phi(\mathbf{x}^{(t+1)})y^{(t+1)}$$

5.1 Study part 1 :

By varying the number of features, the number of samples and the amount of noise in the data generator, compare both recursive variants (with and without the Sherman-Morrison formula) and gradient descent. Which is the most precise? The fastest? Using graphical displays where you are varying the above parameters is strongly encouraged. To change the amount of noise in the generated data, look in the sample_generator.py file.

Dans cette partie, nous allons analyser l'évolution des performances en terme de temps et d'erreur des versions incrémentales des algorithmes **RBFNs**. En particulier une méthode par descente de gradient (**GDRBFN**), et une autre avec la méthode des moindres carrés récursif (**RLSRBFN**). Nous allons aussi analyser les deux méthodes que nous disposons pour calculer les paramètres du modèle **RLSRBFN**. En effet, nous pouvons aussi nous servir de la formule de **Sherman-Morrisson** dans nos calculs.
On commence par faire varier **le nombre de features:**

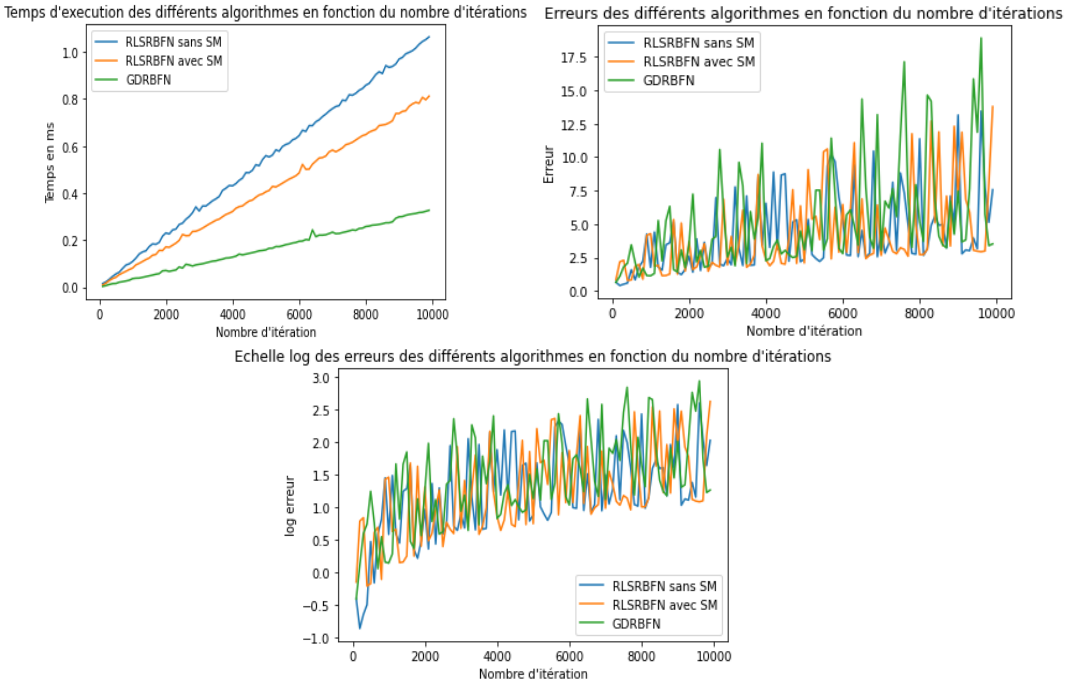


Les tests ci-dessus ont été réalisés en fixant le nombre d'itération à 1000 et avec une valeur de bruit à $\sigma = 0.5$. Nous avons fait varier le nombre de features de 0 à 100 par pas de 1. Dans ces images les courbes bleus et oranges représentent respectivement **RLSRBFN** sans et avec la méthode de **Sherman-Morrisson**, et la courbe verte représente **GDRBFN**.

Nous constatons que les temps d'exécutions (figure de gauche) des versions **RLSRBFN** sont nettement supérieur à **GDRBFN** en faisant varié le nombre de feature. Cela est dû au faite que les méthodes **RLS** doivent inverser une matrice en complexité $\Theta(n^3)$. Nous remarquons aussi que la version avec le calcul **Sherman-Morrisson** permet de gagner du temps de calcul par rapport à la version sans.

Concernant les erreurs (figure de droite), nous voyons que la version **GDRBFN** est assez sensible à l'augmentation du nombre de feature, donc une augmentation d'erreur, ce qui est très visible avec une échelle logarithmique sur les erreurs (figure du bas). L'augmentation de l'erreur en fonction d'une augmentation du nombre de feature est clairement un cas d'**overfitting** (sur apprentissage) et la forte quantité d'erreur au début des courbes est le témoins d'un cas d'**underfitting** (sous apprentissage).

On fait varier maintenant **le nombre d'itérations :**



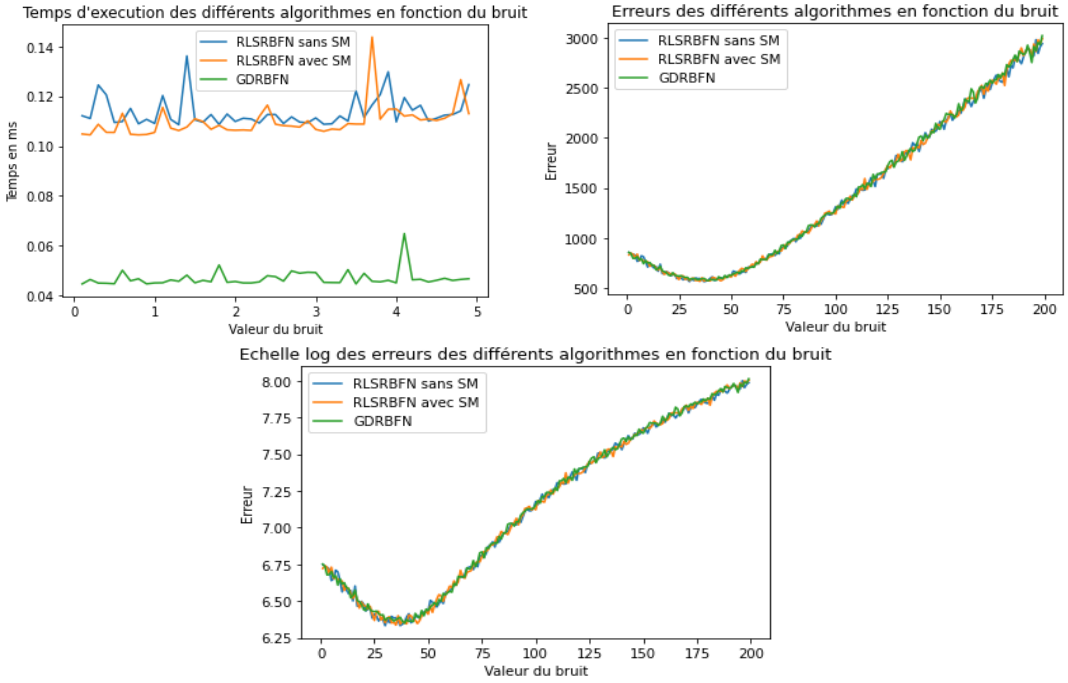
Les tests ci-dessus ont été réalisés en fixant le nombre de feature à 10 et avec une valeur de bruit à $\sigma = 0.5$. Nous avons fait varier le nombre d'itération de 100 à 10000 par pas de 100. Dans ces images les courbes bleus et oranges représentent respectivement **RLSRBFN** sans et avec la méthode de **Sherman-Morrisson**, et la courbe verte représente **GDRBFN**.

Concernant l'évolution du temps (figure de gauche), nous voyons que les deux méthodes **RLSRBFN** mettent beaucoups plus de temps que **GDRBFN** comme nous l'avons analysée précedemment. Cela n'a évidemment pas changer avec un nombre d'itération croissant.

Cependant, nous voyons sur la figure de droite, que plus le nombre d'itération augmente plus l'erreur augmente. Ce qui est d'autant plus visible en échelle logarithmique (figure du bas). Nous estimons que l'erreur est minimal avec un nombre d'itération autour de 2000, donc 2000 points de données. Nous expliquons que cette hausse d'erreur est dû au faite que plus on rajoute de donnée pour un modèle à feature constant plus on lui rajoute du bruit et donc notre modèle et de moins en moins

précis.

Enfin, **le bruit :**



Les tests ci-dessus ont été réalisés en fixant le nombre de feature à 10 et un nombre d'itération à 1000. Nous avons fait varier la valeur du bruit de 0 à 200 par pas de 1. Dans ces images les courbes bleus et oranges représentent respectivement **RLSRBFN** sans et avec la méthode de **Sherman-Morrisson**, et la courbe verte représente **GDRBFN**.

Concernant l'évolution du temps (figure de gauche), nous voyons que les deux méthodes **RLSRBFN** mettent beaucoups plus de temps que **GDRBFN** comme nous l'avons analysée précedemment.Le temps reste constant peu importe la valeur du bruit σ hormis sur certaine valeur dû à l'alea des données.

Pour ce qui est des erreurs, comme nous nous attendions, plus on augmente la dispersion, plus l'apprentissage d'un modèle performant est compliqué. Les figures en bas (Passage au log de l'erreur) et à droite illustrent très bien ce phénomène : lorsque on augmente la valeur $sigma >>$ alors l'erreur augmente aussi.

5.2 Study part 2 :

Using RBFNs, comment on the main differences between incremental and batch methods. What are their main advantages and disadvantages? Explain how you would choose between an incremental and a batch method, depending on the context.

Dans le cadre des RBFNs, on a pu observer plusieurs méthodes de résolution : **la méthode incrémentale** et **la méthode Batch**. A travers nos expérimentations, nous avons constaté d'importantes différences entre les modèles.

Tout d'abord, la méthode Batch était plus précise que la méthode incrémentale mais elle était plus chère en terme de temps d'autant plus qu'on a vu que avec beaucoup de features elle tend a un over-fitting. Au contraire, la méthode incrémentale est moins précise mais elle permet d'obtenir un résultat pour un nombre important de feature.

Donc, lorsque l'on a un nombre raisonnable de features (pas beaucoup), on préférera utiliser la batch méthode pour sa précision mais des lors que le nombre de features augmente, on favoriseras la méthode incrementale.

6 Locally Weighted Regression

Le 3e modèle des RBFNs est Locally Weighted Least Squares. Cette algorithme utilise une somme de poids de M modèles linéaires locales. La formule de thêta est la suivante :

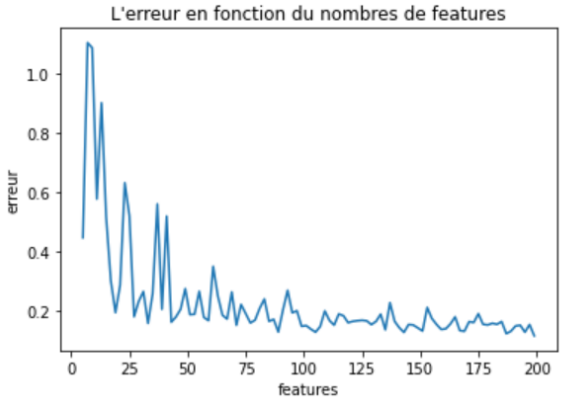
$$\theta_k = A_k^{\sharp} b_k$$

Avec :

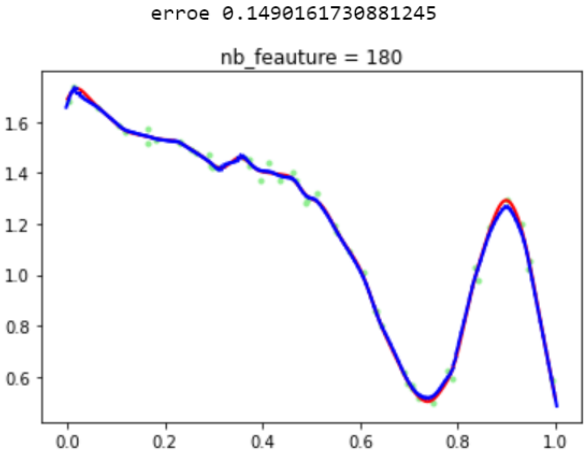
$$\mathbf{A}_k = \sum_{i=1}^{N} \phi_k(\mathbf{x}^{(i)}) \bar{\mathbf{x}}^{(i)} (\bar{\mathbf{x}}^{(i)})^{\intercal} \quad \text{et} \quad \mathbf{b}_k = \sum_{i=1}^{N} \phi_k(\mathbf{x}^{(i)}) \bar{\mathbf{x}}^{(i)} y^{(i)}$$

6.1 Study part :

Study the impact of nb_features on the accuracy of LWR. As usual, make a drawing where you measure the residuals as a function of the number of features.
On calcul l'erreur pour un nombre de feature qui varie entre 0 et 200 et on obtient la courbe suivante:



On remarque, avec peu de feature < 5 on a une erreur élevé et quand on atteint environ 25-30 features l'erreur chute de beaucoup et plus le nombre de feature augmente et plus l'erreur diminue cependant dans ce cas, contrairement a batch RBFN pour un nombre de feature élevé on a :



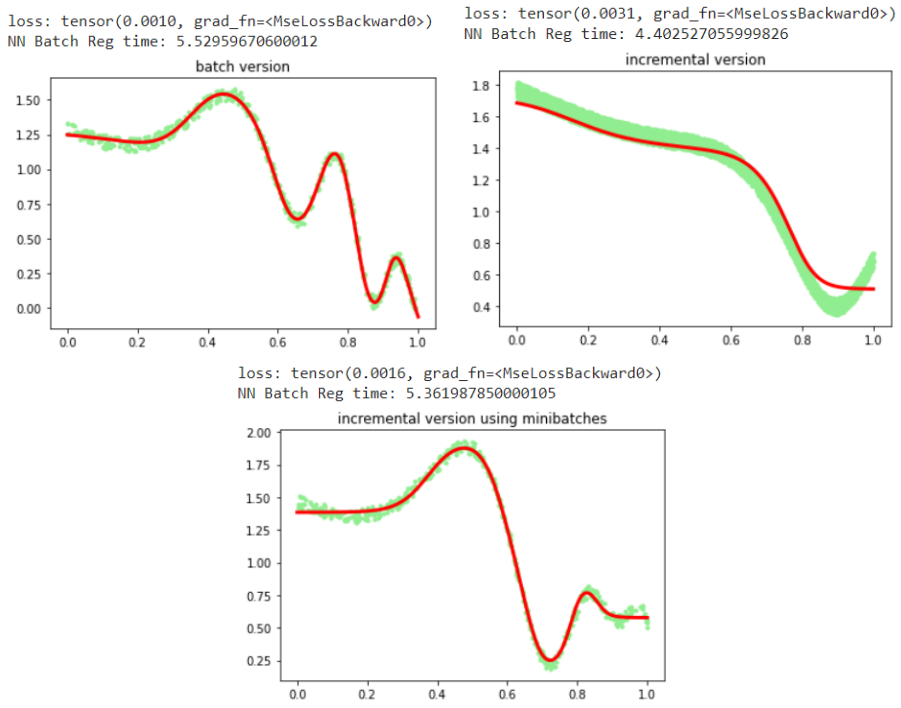
On peut voir que avec 180 feature, on obtient une courbe avec une erreur de 0.14 sans avoir le phénomène de over-fitting, ce qui montre l'efficacité de LWR avec un nombre important de features.

7 Regression with Neural Networks (in pytorch)

Cette partie est consacré a faire les régressions a l'aide des réseaux de neurones en utilisant pytorch.

7.1 Study part 1:

Comment on the difference of results between the batch version, the incremental version and the incremental version with mini-batches.
Pour comparer les trois implémentations, on choisis le même nombre d'itérations égale a 5000, le même réseau de neurones ($NeuralNetwork(1, 10, 20, 50, 1, 0.05)$) et la même size égale a 500, ainsi que 50 la taille des minibatches on obtient :



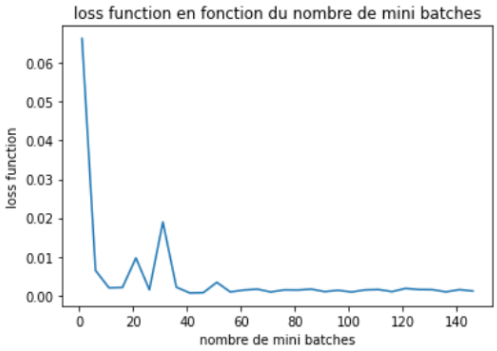
Avec ces trois courbes, On remarque que la méthode la plus rapide est la version incrementale suivi de la version incrementale utilisant des minibatches et enfin la version batch. En ce qui concerne la loss function, la version Batch renvoi la plus petite valeur, suivis de minibatch et enfin de l'incrementale.
Ces résultats ce justifie car la version batch utilise toutes les données pour calculer le gradient donc elle va minimiser le loss mais elle va devoir prendre plus de temps pour le faire.
On en déduit que la la version incrementale utilisant les minibatches est un bon compromis Temps/loss entre les deux versions (incrementale et batch) et c'est donc ce qui est utilisé en pratique.

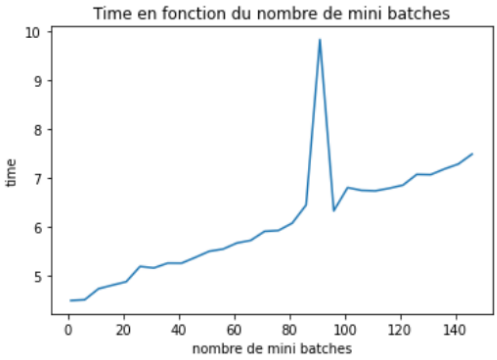
7.2 Study part 2:

By measuring the loss function, study the evolution of the final loss and the training time depending on the size of the minibatch, for a constant budget of iterations.

Tout d'abord, on choisis 5000 comme nombre d'itérations et on choisis comme réseau de neurone: $NeuralNetwork(1, 30, 50, 50, 1, learning\_rate = 0.03)$

En faisant varier le nombre de mini-batches on obtient deux courbes :





On remarque que lorsque le nombre de mini batches est bas la loss function est élevé et le temps est bas. Plus on augmente le nombre de mini batches plus le temps augmente mais la loss function reste stable a partir de 50 batches autour d'une valeur de 0.0015.

7.3 Study part 3:

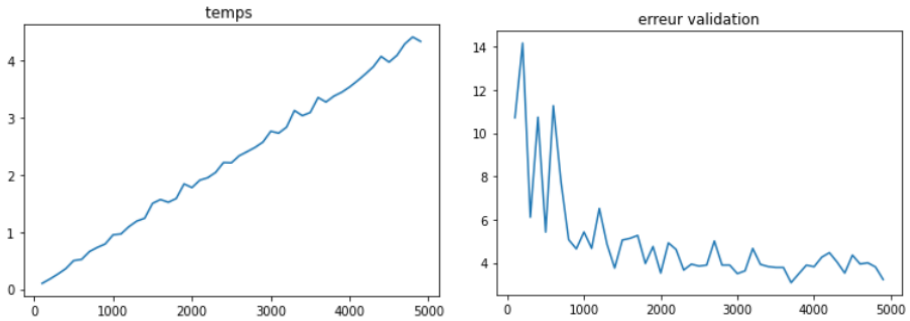
Add a training set, a validation set and a test set, and study overfitting.
Pour répondre a cette question , on a implémente le code suivant :

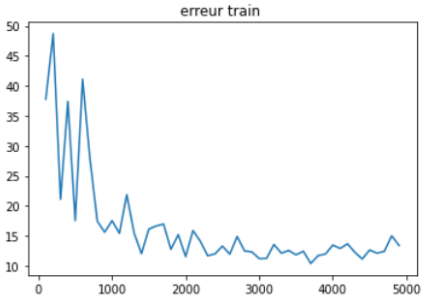
```
from ssl import PROTOCOL_TLS_CLIENT
from random import randrange

ratio_train = 0.8
ratio_test = 0.2
ratio_validation = 0.2          # 20% du train

# Creation du dataSet

batch = Batch()
size = 500
size_train = int(size * ratio_train)
size_test = int(size * ratio_test)
size_validation = int(size_train * ratio_validation)

batch.make_nonlinear_batch_data(size)

X_train = batch.x_data[:(size_train - size_validation)]
X_validation = batch.x_data[(size_train - size_validation):(size_train)]
X_test = batch.x_data[size_train:]
# print(len(X_train))
# print(len(X_test))
# print(len(X_validation))



Y_train = batch.y_data[:(size_train - size_validation)]
Y_validation = batch.y_data[(size_train - size_validation):(size_train)]
Y_test = batch.y_data[size_train:]



model = NeuralNetwork(1, 30, 50, 50, 1, learning_rate=0.015)
start = time.process_time()

cpt=0
for i in range(1000):
  index = randrange((size_train-size_validation))
  x = X_train[index]
  y = Y_train[index]
  xt = np.array([x])
```

```
  yt = th.from_numpy(np.array([y])).float()
  output = model.f(xt)
  loss = func.mse_loss(output, yt)
  model.update(loss)
  cpt+=loss

print("cpt",cpt)
# Evaluation du train

error_train = 0
for i in range(len(X_train)):
  x = X_train[i]
  y = Y_train[i]
  xt = np.array([x])
  yt = th.from_numpy(np.array([y])).float()
  output = model.f(xt)
  error_train += func.mse_loss(output, yt)

print("error train division",(error_train/len(X_train)))

error_validation = 0

##############################
#VALIDATION

for i in range(len(X_validation)):
  x = X_validation[i]
  y = Y_validation[i]
  xt = np.array([x])
  yt = th.from_numpy(np.array([y])).float()
  output = model.f(xt)
  error_validation += func.mse_loss(output, yt)

print("error validation division",(error_validation/len(X_validation)))



# Etude du parametre
list_iteration = np.arange(100,5000,100)
liste_temps = []
liste_error_validation= []
liste_erreur_train = []

for iteration in list_iteration:
  model = NeuralNetwork(1, 30, 50, 50, 1, learning_rate=0.015)
  start = time.process_time()

  cpt=0

  for i in range(iteration):
    index = randrange((size_train-size_validation))
    x = X_train[index]
    y = Y_train[index]
    xt = np.array([x])
    yt = th.from_numpy(np.array([y])).float()
    output = model.f(xt)
    loss = func.mse_loss(output, yt)
    model.update(loss)
    cpt+=loss
  fin = time.process_time()

  liste_temps.append(fin - start)

  # print("cpt",cpt)

  # Evaluation du train

  error_train = 0
  for i in range(len(X_train)):
```

```
    x = X_train[i]
    y = Y_train[i]
    xt = np.array([x])
    yt = th.from_numpy(np.array([y])).float()
    output = model.f(xt)
    error_train += func.mse_loss(output, yt)

  # print("error train",(error_train/len(X_train)))
  # print("error train",(error_train))
  liste_erreur_train.append(error_train)

  error_validation = 0

  ##############################
  #VALIDATION

  for i in range(len(X_validation)):
    x = X_validation[i]
    y = Y_validation[i]
    xt = np.array([x])
    yt = th.from_numpy(np.array([y])).float()
    output = model.f(xt)
    error_validation += func.mse_loss(output, yt)

  # print("error validation",(error_validation/len(X_validation)))
  # print("error validation",(error_validation))
  liste_error_validation.append(error_validation)

print(liste_error_validation)
print(liste_erreur_train)
print(liste_temps)

plt.figure()
plt.title("temps ")
plt.plot(list_iteration,liste_temps)
plt.show()

plt.figure()
plt.title("erreur validation")
plt.plot(list_iteration,liste_error_validation)
plt.show()

plt.figure()
plt.title("erreur train ")
plt.plot(list_iteration,liste_erreur_train)
plt.show()
```

Après avoir plot les résultats on obtient :





Nous avons diviser notre ensemble de données tel que : 80% Train set, 20% test set et 20% du train set concerne le validation set. Nous allons donc faire varier l'hyperparametre du nombre d'itération de l'apprentissage de notre modèle avec l'ensemble de train puis tester l'erreur sur l'ensemble de validation. On peut voir sur ces figures que plus le nombre d'itération augmente, plus le temps augmente.
Nous avons effectuer les tests sur le modele Batch, et nous constatons d'après nos résultats que le nombre optimal ditération est au alentour de 1000. Au dela notre modele risque d'overfitté.