

UNIVERSITÉ DE MONTRÉAL

# Rapport : Devoir3

Pierre Gérard  
Mathieu Bouchard

**IFT3395-6390 Fondements de l'apprentissage machine**  
Pascal Vincent, Alexandre de Brébisson et César Laurent

Année académique 2015 - 2016

# 1 Partie pratique : Implémentation du réseau de neurones

## 1.1 Utilisation

Pour tester notre programme, exécutez simplement la commande dans le dossier *src* :

```
python main.py
```

Les exercices 5 et 9/10 s'exécutent en dernier car ils prennent le plus de temps. Il faut compter quelques secondes pour les premiers exercices, 10 min pour le 5 et quelques dizaines de minutes pour le 9 et 10.

Un dossier */result* a été créé, il contient les résultats pour le 5 et 9-10 car leur exécution prend trop de temps et il n'est pas intéressant de rajouter trop de graphique à ce rapport.

## 1.2 Exercices 1 et 2

Nous avons implémenté la vérification de gradient par différence finie. Pour chaque scalaire "perturbé", on conserve le ratio dans une liste. Une boucle vérifie ensuite que les ratios sont bien entre 0.99 et 1.01. Les résultats semblent indiquer que notre implémentation est correcte.

Pour un réseau  $2 \times 2 \times 2$ , on obtient :

```
>>EXERCICE 1 et 2
Liste des ratio W1, b1, W2, b2
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0.99997500062302613, 1.0000250006249738]
>Tout les ratio sont bien entre 0.99 et 1.01
```

Note : Lorsque des valeurs de  $w_1$  ou  $w_2$  seraient générés aléatoirement à 0, le ratio peut ne pas être dans l'intervalle. Cela ne devrait pas se produire lors de l'exécution du programme car un seed a été passé au générateur de nombre aléatoire.

## 1.3 Exercices 3 et 4

On procède comme à l'exercice 1 et 2 sauf qu'on somme l'ensemble des gradients pour  $K$  exemples avant de calculer le ratio. Les résultats semblent eux-aussi indiquer que notre implémentation est correcte.

Pour un réseau  $2 \times 5 \times 2$ , on obtient :

```
>>EXERCICE 3 et 4
Liste des ratio W1, b1, W2, b2
[1, 1, 1.000002072572213, 1.000008244958942, 1, 1, 0.99999469791365814, 0.99997890803949541,
>Tout les ratio sont bien entre 0.99 et 1.01
```

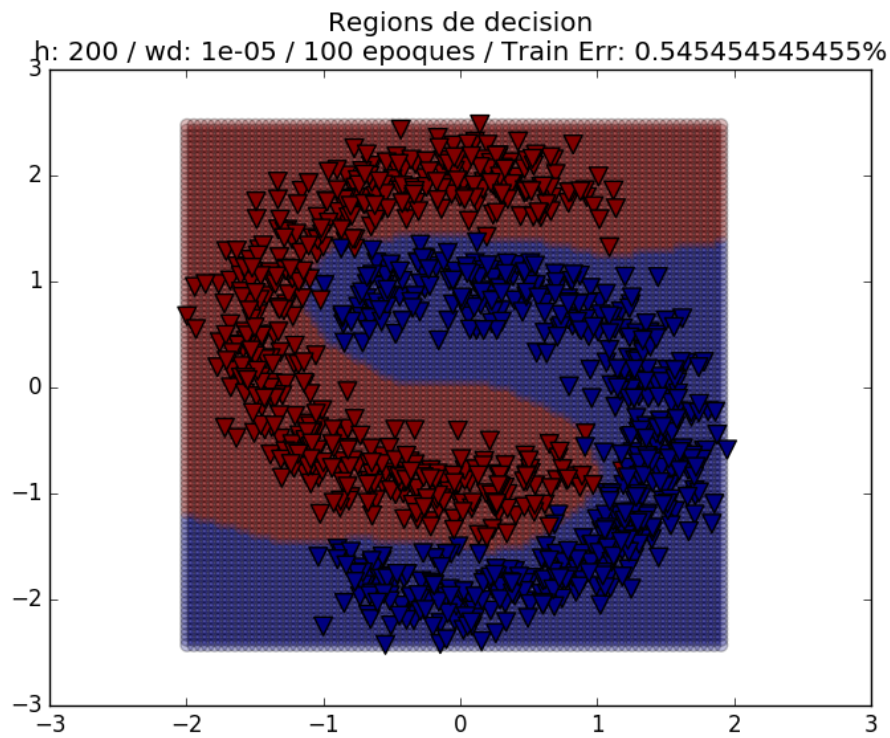
## 1.4 Exercices 5

On a ensuite entraîné notre réseau par descente de gradient mini-batch. L'entraînement semble un succès. Les résultats semblent montrer que plus on augmente le nombre d'époques et de neurones cachés, plus l'erreur sera petite.

### 1.4.1 Exemple de résultat

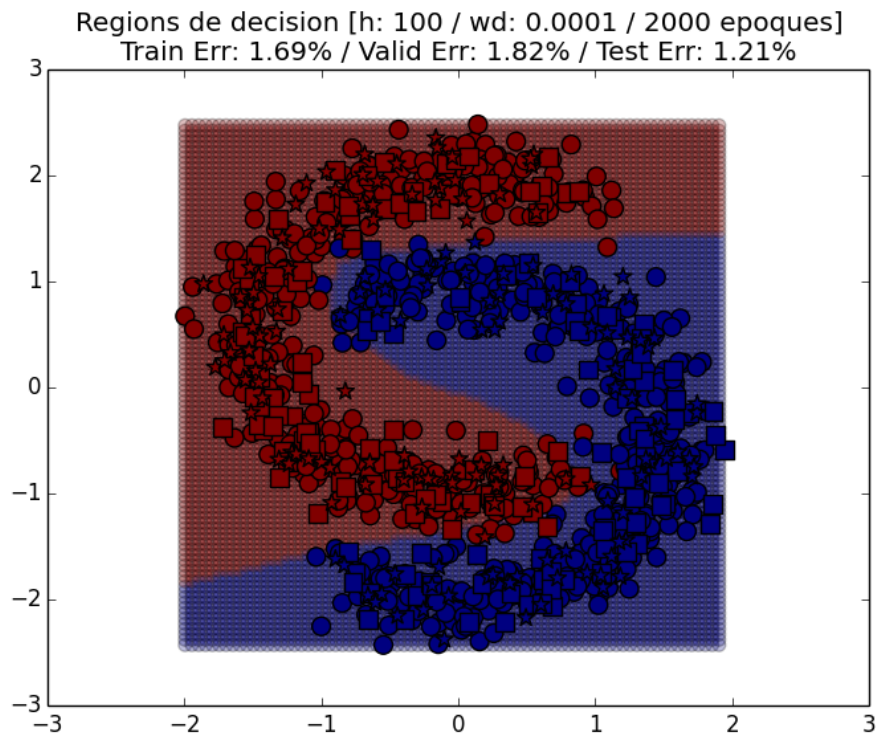
Voici notre meilleur résultat sur l'ensemble d'entraînement :

Ici nous n'avons considéré que un ensemble d'entraînement contenant tous les points.

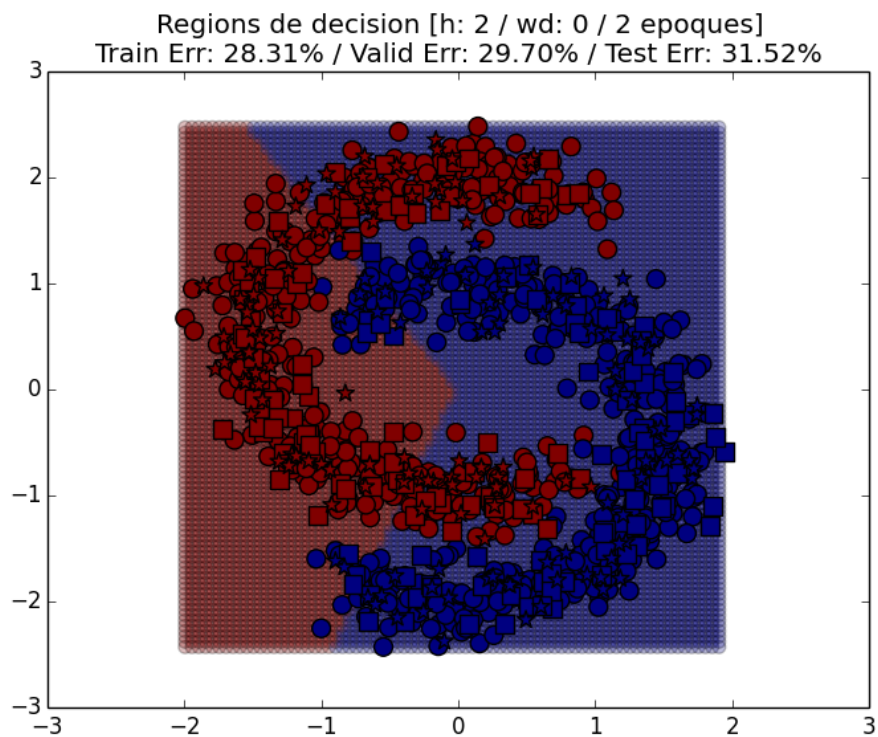


En divisant l'ensemble en 3 ensembles : entraînement, validation et test et pour d'autres valeurs d'hyper-paramètre, on obtient :

Ici nous avons mis beaucoup d'itération de manière à obtenir un bon résultat. Les batch ont une taille de 50.



Un dernier exemple pour de mauvaises valeurs. Ici on a délibérément choisit de mauvais hyper-paramètre : seulement 2 neurones cachés et deux itérations. Les erreurs sont élevées.



### 1.4.2 Généralisation

D'après les résultats obtenus en faisant varier les résultats des hyper-paramètres, voici leurs influences :

- Les hyper-paramètres *theta* et *weightDecay* font varier la vitesse l'algorithme converge. Pour des résultats consistant entre plusieurs exécutions et si possible d'un point de vue computationnel, il vaut mieux les garder petit et augmenter le nombre d'itération
- Il semblerait que augmenter l'hyper-paramètre K fait converger l'algorithme vers un bon résultat plus rapidement.
- Le nombre de neurone caché semble avoir une influence sur les résultats, il vaut mieux en avoir un grand nombre mais à partir d'un moment, augmenter le nombre ne diminue plus le taux d'erreur.
- Le nombre d'itération semble être le paramètre le plus important. Plus il augmente, plus on a de chance de se retrouver avec un bon résultat et des erreurs très petite.

## 1.5 Exercices 6 et 7

### 1.6 Input

La fonction *fprop* et *bprop* vont recevoir de nouveaux inputs. Il recevait avant les exemple un par un et ils avaient la forme suivante :

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix}$$

A la place ils ont maintenant la forme suivante pour un batch de taille K.

$$\begin{bmatrix} x_{11} & x_{12} & x_{13} & \dots & x_{1K} \\ x_{21} & x_{22} & x_{23} & \dots & x_{2K} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_{d1} & x_{d2} & x_{d3} & \dots & x_{dK} \end{bmatrix}$$

Cela de même pour y.

#### 1.6.1 fprop

On cherche a obtenir une matrice pour la sortie du même format que l'input. C'est a dire une matrice ou la colonne i correspond aux sorties pour la colonne i de la matrice X.

Il a été nécessaire de modifier la fonction relu et softmax pour qu'elles acceptent des matrices. Ensuite, il n'a pas fallu modifier w1, w2, b1 ou bien b2.

En effet lorsqu'on multiplie la transposé de X par w1, au lieu d'obtenir un vecteur colonne comme avant, on obtient une matrice de K vecteur colonne. Il suffit de rajouter b1 a chaque colonne ensuite.

Cela de même pour oa, au lieu d'obtenir un vecteur colonne, on obtient la matrice des K vecteurs colonnes.

Il suffit ensuite d'appliquer le softmax modifier pour accepter la matrice.

### 1.6.2 bprop

On cherche a obtenir les 4 gradients w1, w2, b1 b2.

Il a été nécessaire de modifier la fonction onehot pour qu'elle accepte un vecteur de classe de sortie et renvoie la matrice correspondante.

Le gradient gradb2 sera une matrice dont les colonnes seront les gradient b2 pour chaque exemple car il est égale a gradoa qui lui même est la soustraction des résultats de fprop par onehot.

Le gradient w2 quant a lui aura la même dimension que avant. Cependant ce sera la somme des gradients individuels. Avant on avait un produit matriciel qui faisait un element de oa pour un element de hs, maintenant on a la somme des produits d'une ligne de oa par une colonne de hs et cela correspond a la somme des gradients individuels.

C'est ensuite le même principe pour les autres matrices.

### 1.6.3 Verification des modifications

Pour cette question, on a écrit une méthode qui pour deux réseaux de neurones vérifie que les gradients soient équivalent. C'est a dire qui va regardé pour deux classe *NeuralNetwork* que les quatre attributs de classes (gradw1, gradw2, gradb1, gradb2) sont égales. Il faut bien sur initialiser les matrices w1 et w2 à des mêmes valeurs.

Pour vérifier que les gradients soient équivalents, on commence par instancier les deux classes correspondant aux deux implémentations *NeuralNetwork* et *NeuralNetworkEfficient*. On initie ensuite W1 et W2 des deux classes aux mêmes valeurs. Ensuite on execute l'entrainement pour K=1. On utilise ensuite la méthode qui pour deux réseaux de neurones vérifie que les gradients soient équivalent. On recommence le procédé pour K=10.

Les tests de comparaison nous donne (ok signifie que les deux gradients sont égaux) :

```
--- K=1 ---
Test Ok : gradient b2
Test Ok : gradient w2
Test Ok : gradient b1
Test Ok : gradient w1
--- K=10 ---
Test Ok : gradient b2
Test Ok : gradient w2
Test Ok : gradient b1
Test Ok : gradient w1
```

## 1.7 Exercices 8

Les résultats des mesures de temps pour une exécution sont les suivant :

```
>>EXERCICE 8 MNIST
--- Reseau de depart ---
Cela a mis : 0.419163 secondes
--- Reseau optimise ---
Cela a mis : 0.123445 secondes
```

Sur cette exécution, on remarque que le temps d'exécution sur le réseau avec matrice est entre 3 et 4 fois plus rapide que sur le réseau avec boucles. Le gain de performance n'est donc pas négligeable. Si on essaye d'autres exécutions, on remarque des résultats similaires.

## 1.8 Exercices 9 et 10

Pour alléger la charge de calcul et ainsi augmenter la vitesse d'exécution sur les données MNIST, nous avons décidé de calculer les valeurs d'erreur et de coût moyen à toutes les 10 époques.

Pendant l'exécution, les informations sont affichées comme ceci :

```
>>EXERCICE 9-10
Train Err;Valid Err;Test Err;Avg Cost Train;Avg Cost Valid;Avg Cost Test
89.598;88.860;90.020;2.300;2.299;2.300 (0)
77.486;75.380;76.970;2.225;2.222;2.224 (10)
64.096;62.000;64.990;2.156;2.150;2.154 (20)
57.392;55.690;58.460;2.058;2.050;2.054 (30)
49.040;46.590;48.940;1.932;1.920;1.924 (40)
45.362;43.740;45.610;1.804;1.788;1.795 (50)
37.700;35.720;37.270;1.669;1.649;1.656 (60)
32.322;29.390;31.920;1.530;1.507;1.516 (70)
24.604;22.070;23.910;1.391;1.364;1.372 (80)
25.958;23.320;24.950;1.254;1.221;1.235 (90)
23.490;20.810;22.470;1.143;1.107;1.121 (100)
23.094;20.320;21.760;1.053;1.015;1.031 (110)
21.656;18.420;20.360;0.970;0.930;0.946 (120)
19.366;16.830;18.000;0.898;0.854;0.872 (130)
18.676;16.130;17.380;0.839;0.794;0.814 (140)
18.384;15.880;17.270;0.803;0.757;0.778 (150)
18.462;16.030;17.370;0.758;0.713;0.732 (160)
17.796;15.560;16.500;0.720;0.675;0.695 (170)
18.480;16.080;17.460;0.707;0.662;0.684 (180)
16.594;14.350;15.560;0.669;0.622;0.645 (190)
```

On retrouve alors dans l'ordre les 3 valeurs d'erreur de classification (train, validation, test) suivies des 3 valeurs de coût moyen (train, validation, test). Finalement, la valeur entre parenthèses représente le numéro de l'époque pour laquelle ces valeurs ont été calculées.

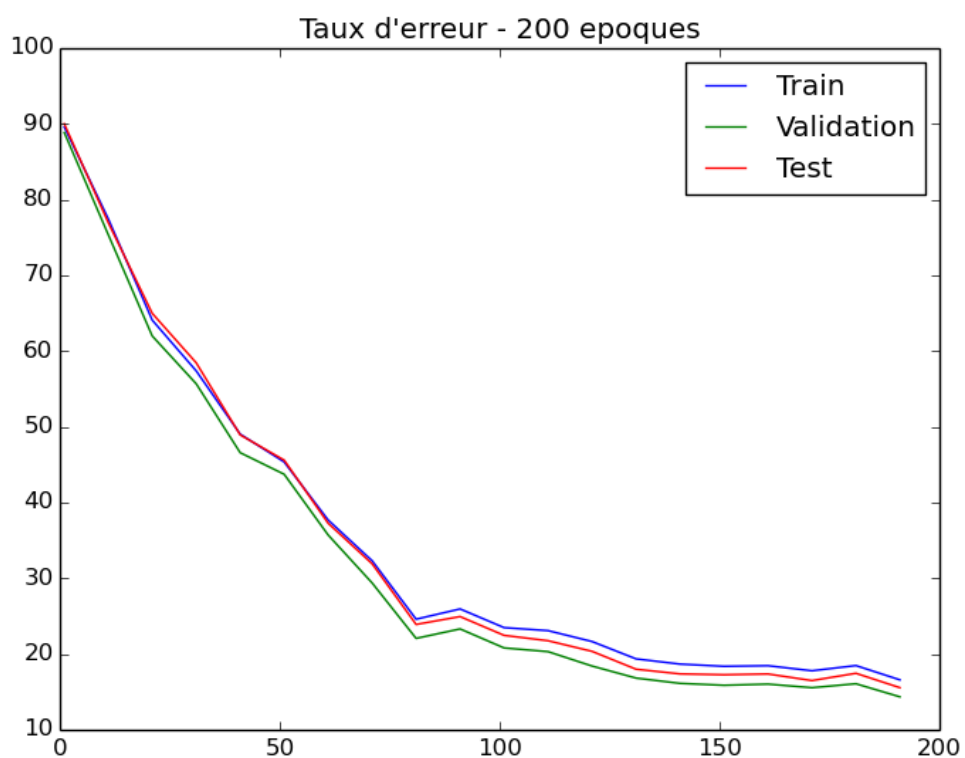
À la toute fin de l'exécution, ces valeurs sont également enregistrées dans un fichier texte nommé 9.txt se trouvant à la racine du dossier src.

## 1.9 Nombre d'itération petit

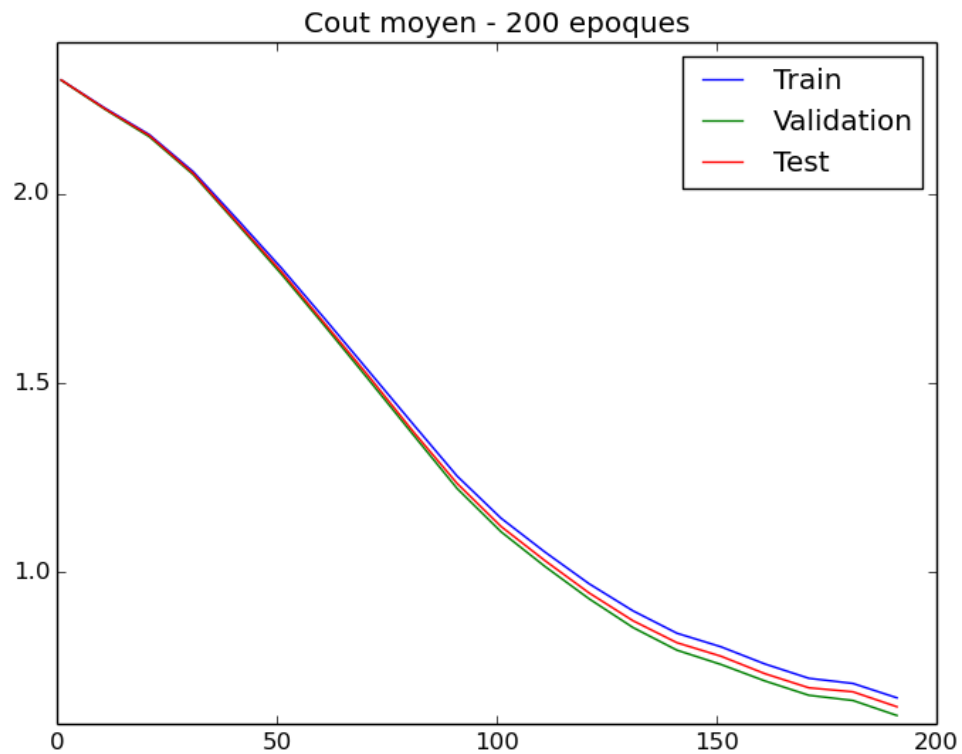
Nous avons obtenus les résultats suivant lorsque nos hyper-paramètres avaient les valeurs suivantes :

`h = 30`  
`wd = 0.0001`  
`maxIter = 200`  
`K = 50`

On obtient alors les résultats suivants :







### 1.10 En dessous de 5 pourcent : grand nombre d'itération

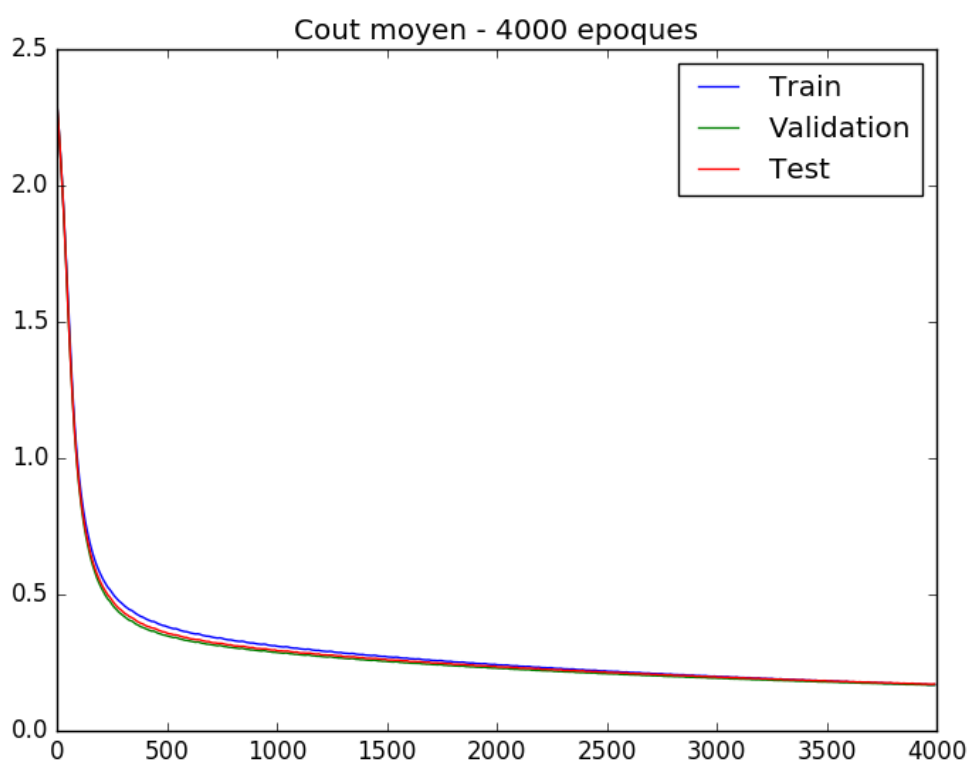
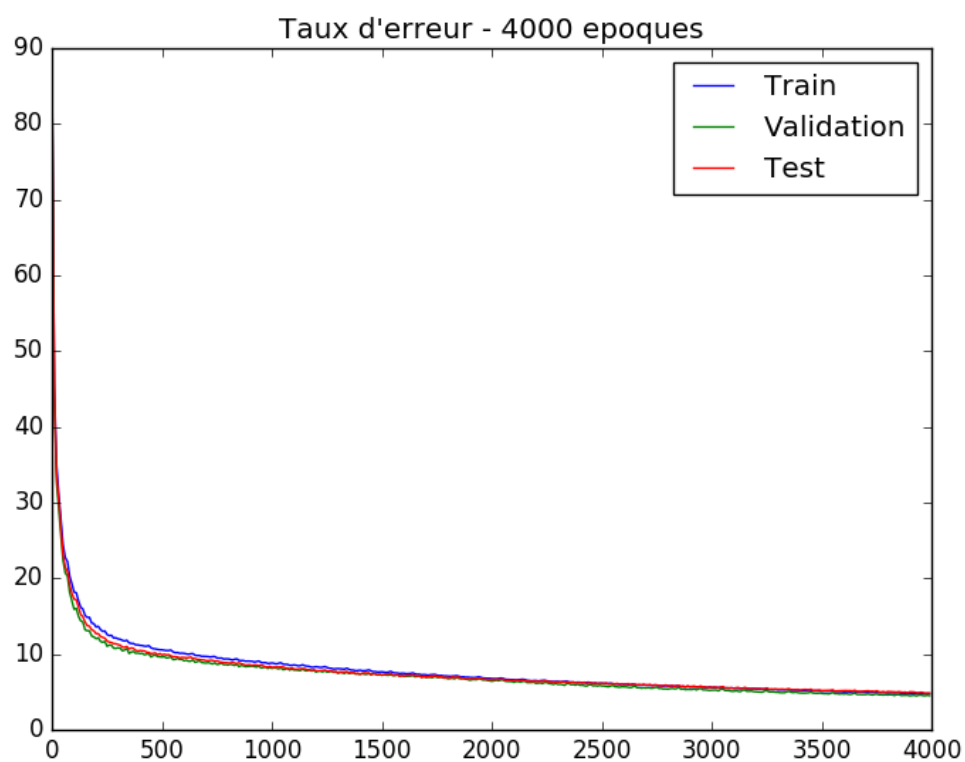
Nous avons obtenus les résultat suivant avec un grand nombre d'itération

- Erreur d'entraînement : 4.762%
- Erreur de validation : 4.510%
- Erreur de test : 4.850%

Les hyper-paramètres avaient les valeurs suivantes :

```
h = 400
wd = 0.0001
maxIter = 4000
K = 50
```

Et les graphiques sont les suivants :



Ces résultats en dessous de 5% auraient pu être établis avec moins d'itération en modifiant les hyper-paramètres qui définissent la vitesse de convergence.