Sorbonne Université

Rapport de projet

ML

Zhihao ZHU
Master 1 Semestre 2

Tuteur : M. sylvain LAMPRIER

Février 2021 — Mai 2021

Index

Table des matières

1 Première partie : lossFonction.py **3**

2 Deuxième partie : Module.py **3**

2.1 Classe Module . . . 3

2.2 Partie Linéaire . . . 3

2.3 Partie CNN (Convolutional neural network) . . . 4

2.3.1 Partie Conv1D . . . 4

2.3.2 Partie MaxPool1D et avgPool1D . . . 4

2.3.3 Partie Flatten . . . 4

3 Troisième partie : Sequentiel.py **4**

4 Quatrième partie : Optim.py **5**

5 Cinquième partie : Kmeans.py **5**

6 Partie test **6**

6.1 test pour petit jeu de données pour réseau non linéaire . . . 6

6.1.1 TanH et sortie taille 1 . . . 6

6.1.2 Sigmoide et sortie taille 2 . . . 6

6.1.3 Softmax et multiclass . . . 7

6.1.4 BCE . . . 8

6.2 test pour les traitements linéaires . . . 9

6.2.1 Générateur de données(Linéaire) . . . 9

6.2.2 Une couche Lineaire . . . 9

6.2.3 Deux couches Lineaires . . . 9

6.2.4 Plusieurs couches Lineaires . . . 10

6.3 Analyse de data MNIST (classification) . . . 12

6.3.1 Transfert datay de taille *(nb_train,1)* à *(nb_train,nb_class)* (onehot) . . . 12

6.3.2 Classification des MNIST . . . 12

6.4 Encodage et Décodage (on utilise MNIST) 15
6.4.1 Si on normalise datax 15
6.4.2 Si on transfert datax ver type 0-1 17
6.4.3 Les performances en débruitage 18
6.4.4 Le pré-traitement par l'auto-encodeur et faire la classification 19
6.5 Convolution 20

1 Première partie : lossFonction.py

Tout d'abord, on prend la partie des Fonction Loss et on les écrit dans la partie `LossFonction.py`. On met ici Les Classes `MSEloss`, `CE` et `BCE` qui contient les fonctions de *forward* et *backward* pour régler le problème de loss.

Par exemple, on affiche ici la classe de `MSEloss` :

```
class MSELoss (Loss) :
    def forward(self,yhat,y) :
        """
        in :
            y : (batch,d)
            yhat : (batch,d)
        out : (batch,1)

        """
        return np.linalg.norm (y - yhat,axis=1)**2

    def backward (self, yhat, y):
        """
        in :
            y : (batch,d)
            yhat : (batch,d)
        out : (batch,d)

        """
        return 2*(yhat - y)
```

2 Deuxième partie : Module.py

2.1 Classe Module

La deuxième partie qu'on réalise est la classe `Module`, c'est une partie principale qui contient les fonctions principales de projet comme

- la fonction *update_parameters* qui sert à calculer la mise a jour des parametres selon le gradient calcule et le pas de *gradient_step*
- la fonction *backward_update_gradient* sert à mettre à jour la valeur du gradient
- la fonction *backward_delta* sert à calculer la derivee de l'erreur

2.2 Partie Linéaire

On développe dans cette partie l'ensemble des classes de fonction activation comme `Linear`, `Sigmoide`, `TanH`, `Softmax` et `ReLU`. Pour chaque modèle, on change la fonction g et $g\_gradient$ (mais pour `Softmax`, il faut change *forward*). On implémente les modèles `Sigmoide`, `TanH`,`Softmax` et `ReLU` à partir de la classe `Linear` :

```
class Linear (Module)

class Sigmoide (Linear)

class TanH (Linear)

class Softmax (Linear)

class ReLU (Linear)
```

2.3 Partie CNN (Convolutional neural network)

2.3.1 Partie Conv1D

On développe dans cette partie l'ensemble des fonctions principales pour Convolutional neural network. Il faut redéfinir tous les fonction de la classe `Module`.

```
class Conv1D (Module)
```

2.3.2 Partie MaxPool1D et avgPool1D

On développe dans cette partie l'ensemble des fonctions principales pour la couche de `MAX Pooling` et `AVG Pooling` de dimension 1.Les fonctions sont presque de la forme de partie CNN. **On rappel que *MaxPooling* et *AvgPooling* n'ont pas de paramètre.**

```
class MaxPool1D(Module)

class AvgPool1D(Module)
```

2.3.3 Partie Flatten

On développe dans cette partie l'ensemble des fonctions principales qui permet de transformer la forme de l'entrée. Il suffit d'utiliser `np.reshape` et **on rappel que *Flatten* n'a pas de paramètre**.

```
class Flatten(Module)
```

3 Troisième partie : Sequentiel.py

La troisième partie qu'on réalise est la classe `Sequentiel`,qui correspondant à permet d'ajouter des modules en série.

- les procédures de *forward* et *backward* sont automatisés pour quel que soit le nombre de modèles mis à la suite selon la demande de sujet de projet.
- *zero_grad* va annuler gradient.

4 Quatrième partie : Optim.py

La quatrième partie qu'on réalise est la classe `Optim`,qui est pour condenser une itération de gradient
– la fonction *step* pour calculer une fois de backward de tous les modèles.
– la fonction *backward* pour faire plusieurs étapes et apprendre le Réseau.

5 Cinquième partie : Kmeans.py

La cinquième partie qu'on réalise est la classe `Kmeans`,qui sert à étudier le clustering induit dans l'espace latent donc on prend l'exemple de `Kmeans` pour traiter ce problème. Elle contient

– $dist(p1, p2)$ pour retourner la distance euclidenne entre deux points
– $cal\_centroids(clusters, precision)$ pour retourner $centroids$ (nouveux centres) pour un input de liste de clusters
– $compare\_lists(list\_1, list\_2)$ pour retourner `True` quand les deux listes sont meme sinon returns `False` comme on sait que les deux listes ne sont pas en ordre
– $cost\_fonction$ pour calculer le coût
– fit pour exécuter le clustering

6 Partie test

6.1 test pour petit jeu de données pour réseau non linéaire

6.1.1 TanH et sortie taille 1

On affiche le résultat de yhat et le graphe de ce test pour MSE coût pour la fonction `TanH`

Réseau : `Linear(4,4)` ⇒ `TanH(4,1)`

```
yhat :
    array([[ 0.98997776],
    [ 0.99892385],
    [-0.9999996 ],
    [-0.99697107],
    [-0.99999839]])

y :
    array([[ 1],
           [ 1],
           [-1],
           [-1],
           [-1]])
```

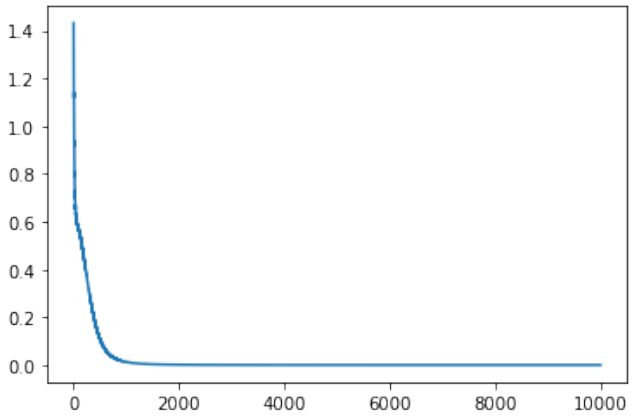

FIGURE 1 – coût de réseau pour TanH test

6.1.2 Sigmoide et sortie taille 2

De même, on affiche le résultat de yhat et le graphe de ce test pour MSE coût pour la fonction `Sigmoide`.

Réseau : `Linear(4,4)` ⇒ `Sigmoide(4,2)`

```
yhat :
    array([[9.82458307e-01, 9.82312952e-01],
```

```
            [9.97114103e-01, 9.97102596e-01],
            [3.60625499e-06, 3.52962312e-06],
            [5.19614326e-03, 5.17631664e-03],
            [1.02136928e-05, 9.89739964e-06]])

y :
    array([[1, 1],
           [1, 1],
           [0, 0],
           [0, 0],
           [0, 0]])
```

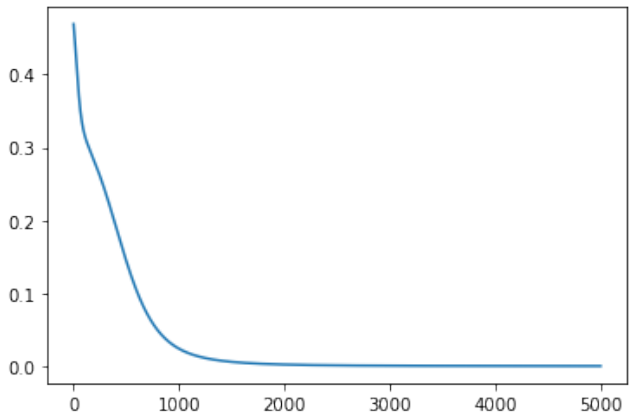

FIGURE 2 – coût de réseau pour Sigmoide test

6.1.3 Softmax et multiclass

Encore une fois, on affiche le résultat de yhat et le graphe de ce test pour MSE coût pour la fonction `Softmax` mais cette fois on calcule la précision comparé avec l'entraînement et on obtient le coefficient **1.0** (tous correct).

Réseau : `Linear(4,4)` ⇒ `Softmax(4,2)`

```
yhat :
    array([[9.93707028e-01, 6.29297159e-03],
           [9.99354631e-01, 6.45369387e-04],
           [1.73170845e-07, 9.99999827e-01],
           [1.59887541e-03, 9.98401125e-01],
           [5.87007047e-07, 9.99999413e-01]])

classification predict :
    array([0, 0, 1, 1, 1], dtype=int64)

y :
    np.array([[1,0],[1,0],[0,1],[0,1],[0,1]])
```

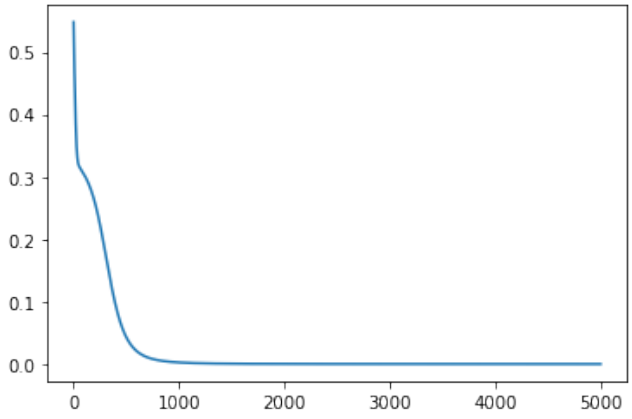

FIGURE 3 – coût de réseau pour Softmax test

6.1.4 BCE

Enfin, on affiche le résultat de yhat et le graphe de ce test pour BCE coût pour la fonction `Softmax`. On calcule la précision comparé avec l'entraînement et on obtient le coefficient **0.8** (mais on peut dire que `BCE` n'est pas bien adapté pour ce réseau).

Réseau : `Linear(4,4)` ⇒ `Softmax(4,2)`

```
yhat :
      array([[1.0298656e-01, 8.9701344e-01],
       [1.0000000e+00, 8.7355095e-31],
       [0.0000000e+00, 1.0000000e+00],
       [0.0000000e+00, 1.0000000e+00],
       [0.0000000e+00, 1.0000000e+00]])

y :
    np.array([[1,0],[1,0],[0,1],[0,1],[0,1]])
```

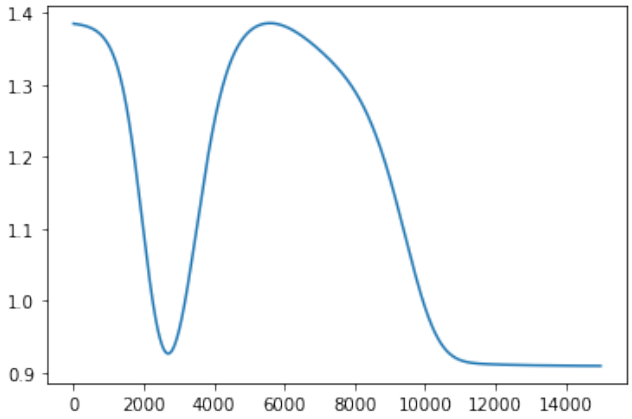

FIGURE 4 – coût de réseau pour Softmax et BCE test

6.2 test pour les traitements linéaires

6.2.1 Génerateur de données(Linéaire)

on définit le jeu de donnée :
- entrée : n'importe quel dimension de données
- sortie : dimension 1
- si on peut ajouter le bruit

6.2.2 Une couche Lineaire

On établit une couche lineaire et on étudie la relation entre le coût et la taille de input.

Réseau : `Linear(inputsize,1)`

```
sans bruit :

parametre optimise :
    array([[100.],
           [  2.],
           [ 15.],
           [ 16.],
           [ 19.]])

parametre reel :
    [[100], [2], [15], [16], [19]]
```

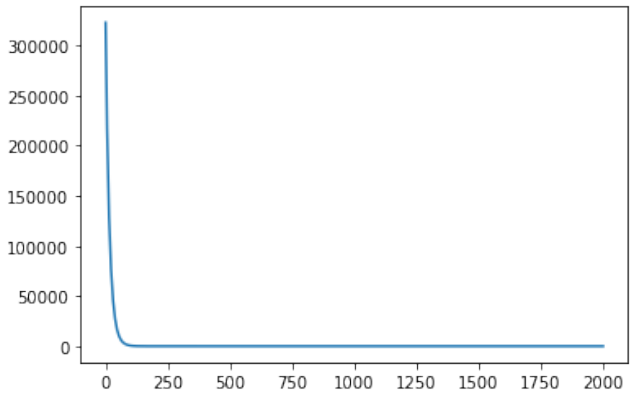

FIGURE 5 – Une couche Lineaire

6.2.3 Deux couches Lineaires

De même,on établit deux couches lineaires et on étudie la relation entre le coût et la taille de input.

Réseau : `Linear(inputsize1,inputsize2)` → `Linear(inputsize2,ouputsize2)`

```
avec bruit :

parametre optimise :
    array([[-98.99319498],
           [194.97790237]])

parametre reel :
    [[1,10],[-5,-20]]

parametre reel combine :
    array([[-99],
           [195]])
```

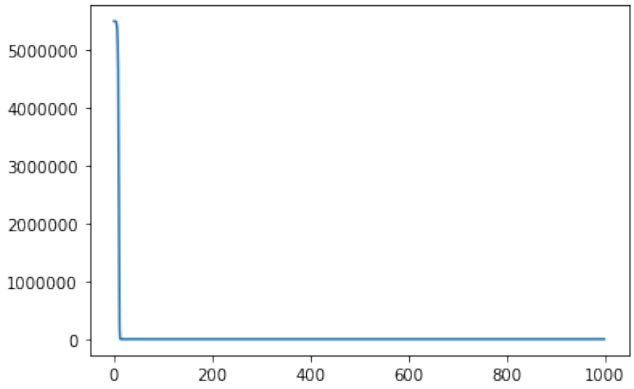

FIGURE 6 – Deux couches Lineaires

6.2.4 Plusieurs couches Lineaires

Pour comparaison,on établit Plusieur couches Lineaire et on étudie la relation entre le coût et la taille de input.

Réseau : on définit par le `list_parametres` qu'on entre.

```
Lineaire_test (list_parametres, nb_data=nb_data, bruit=False, maxite=5000,
   eps=1e-5, batch_size=100, Nan_eviter=1e-1)
```

```
list_parametres = [
    [[1,2,10],[-5,2,-20]],
    [[1,-10],[2,5],[3,0]],
    [[8],[-8]]
]

parametres reel :
 [[ 280.]
 [-968.]]

parametres estime :
 [[ 280.]
```

```
13 [-968.]]
```

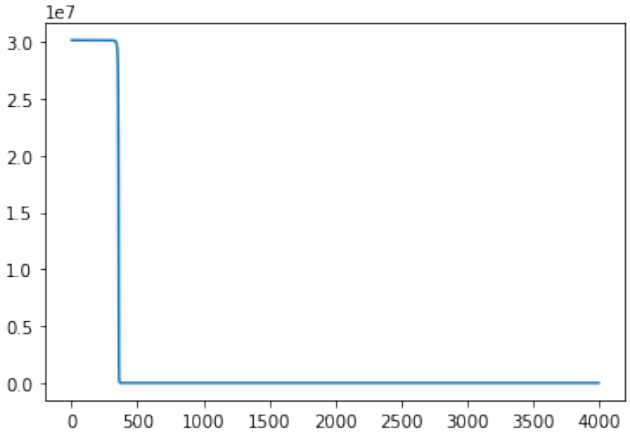

FIGURE 7 – Plusieurs couches Lineaires

6.3 Analyse de data MNIST (classification)

Cette partie on traite la classification pour vérifier que notre Réseau marche bien et au desous c'est quelque chiffres sur notre dataset.

```
nombre classe :  10
train :  (5000, 256)
test :  (100, 256)
valeur de image : de  0.0  a  2.0
```

6.3.1 Transfert datay de taille *(nb_ train,1)* à *(nb_ train,nb_ class)* (onehot)

On prend deux méthodes

Méthode 1 :

```
datay = []
classe_vecteur = np.identity(nb_class)
for classe in datay_singleton :
    datay.append (classe_vecteur[classe])
datay = np.array(datay)
```

Méthode 2 :

```
datay = np.zeros((datay_singleton.size,nb_class))
datay[np.arange(datay_singleton.size),datay_singleton]=1

dataytest = np.zeros((datay_singletontest.size,nb_class))
dataytest[np.arange(datay_singletontest.size),datay_singletontest]=1
```

on préfère la méthode 2

6.3.2 Classification des MNIST

Méthode supervisé

On teste la précision pour la méthode supervisé et on a une précision de 96% pour l'entraînement et une précision de 95% pour le test et D'ailleurs le graphe pour MSE.

La distribution d'échantillions pour chaque classe i :

`[846, 677, 480, 390, 439, 351, 483, 450, 385, 499]`

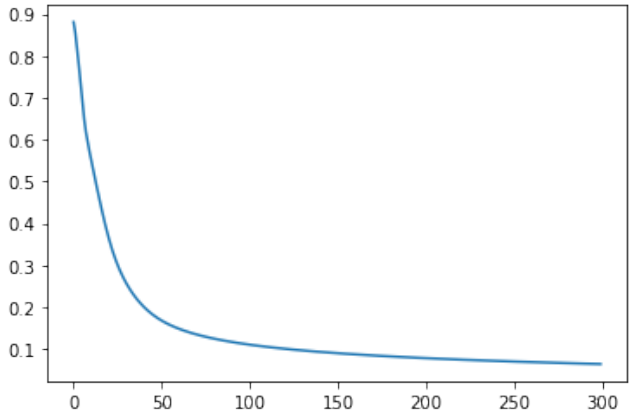

FIGURE 8 – méthode supervisé

Méthode non supervisé

On essaie d'estimer par la méthode non supervisé. On ajoute une colonne pour vérifier les classes de chaque echantillion et c'est ne change pas beaucoup parceque la dimension et 256 (257 ne change pas beaucoup)

K-means

Premièrement, c'est la méthode de `K-means`.on ne sait pas c'est quelle classe répresenté par chaque cluster, mais la distribution est proches par rapport la distribution de données réelles (pureté et implémentation : `np.bincount(np.argmax(opti.output,axis=1))`) d'après la colonne on a ajouté avant

Par exemple, ici on peut dire que
– le premier cluster est de classe 7 de 477 échantillons et réellement classe 7 de 450 échantillons
– le deuxième cluster est de classe 1 de 697 échantillons et réellement classe 1 de 677 échantillons

```
nombre d'echantillion dans chaque cluster :  [477, 697, 809, 484, 226, 549,
   273, 678, 359, 448]
la classe estime pour chaque cluster :  [7, 1, 9, 6, 0, 4, 0, 3, 0, 2]
```

Et ainsi le graphe de coût.

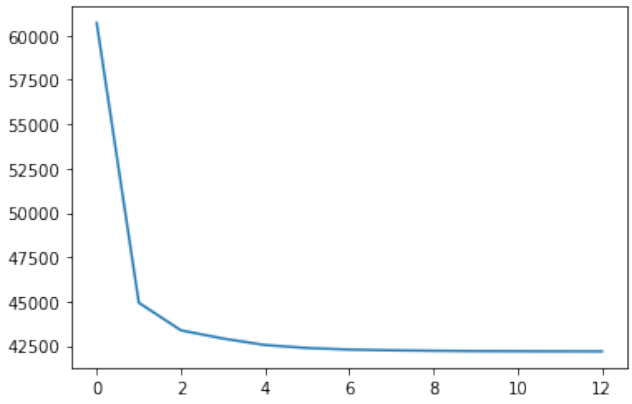

FIGURE 9 – Kmeans

t-SNE

D'ailleurs, on essaie une autre méthode de `t-SNE` et On obtient le graphe de la distribution des points.

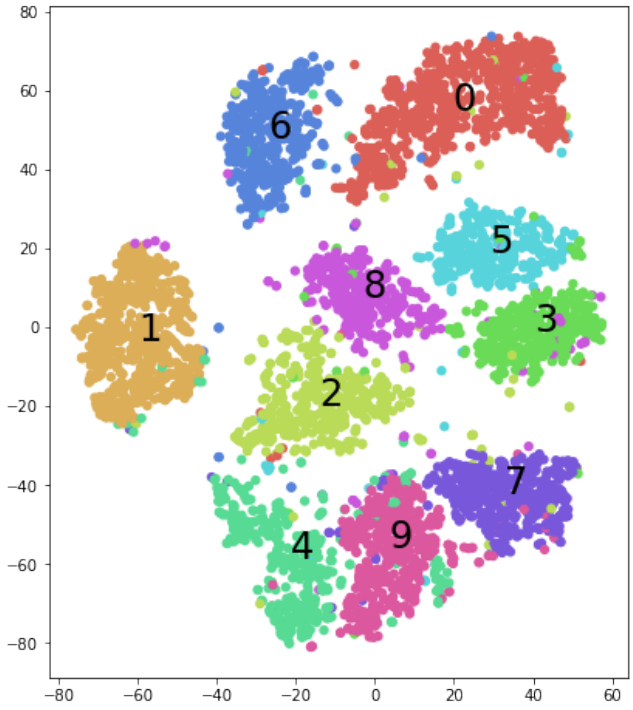

FIGURE 10 – t-SNE

6.4 Encodage et Décodage (on utilise MNIST)

c'est quelque chiffres sur notre dataset c'est le même qu'avant.

```
nombre classe :  10
train :  (5000, 256)
test :  (100, 256)
valeur de image : de  0.0  a  2.0
```

on fait $maxite = 100, batch\_size = nb\_train = 5000$ juste pour tester notre code marche bien. Le resultat va devenir mauvais

6.4.1 Si on normalise datax

On fait la normalisation de datax et puis on obtient les précisions et Loss de chaque network.

Net de exemple de sujet

on ne peut pas calcule le précision, on peut just observer les images

Réseau : `TanH(inputsize,100)` → `TanH(100,nb_class)` → `TanH(nb_class,100)` → `Sigmoide(100,inputsize)`

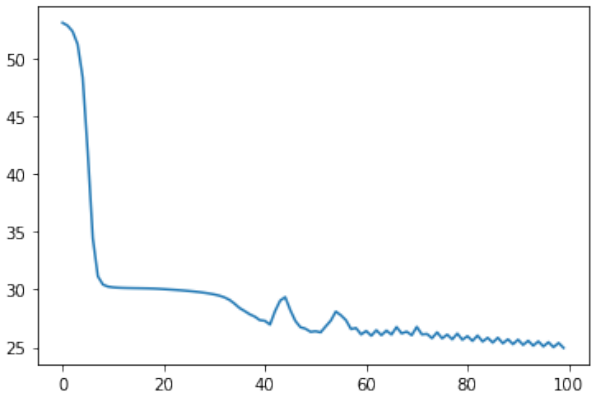

FIGURE 11 – Coût

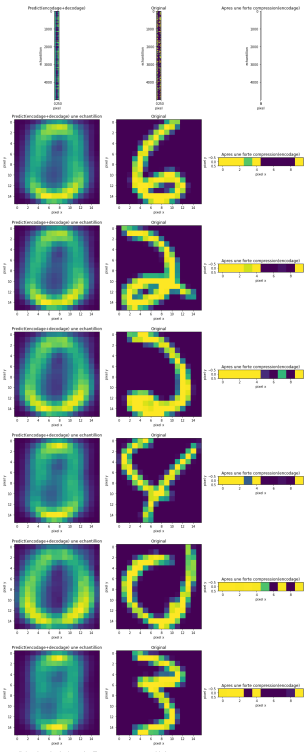

FIGURE 12 – Précision

Net on créé

Réseau : `Linear(inputsize,50)` → `Sigmoide(50,nb_class)` → `Linear(nb_class,50)` → `Sigmoide(50,inputsize)`

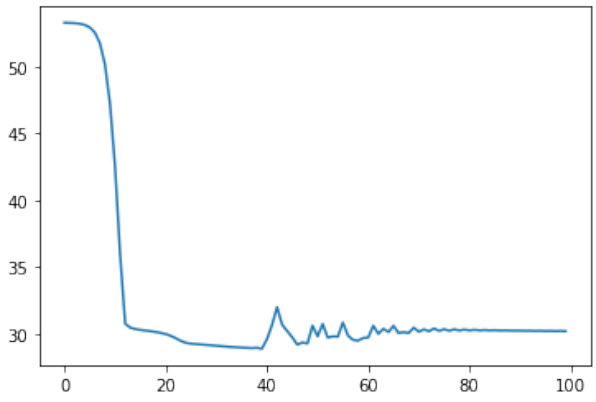

FIGURE 13 – Coût

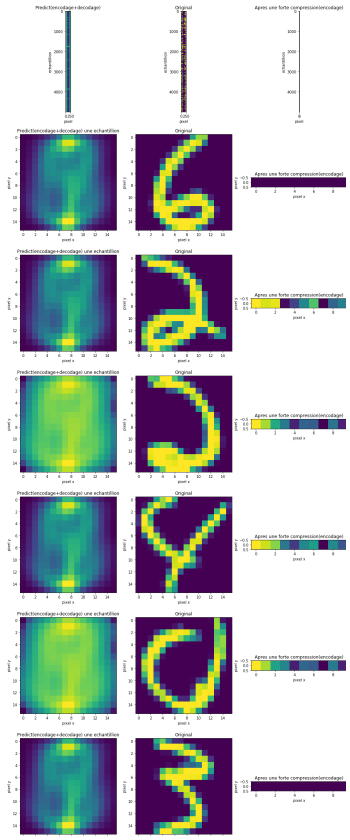

FIGURE 14 – Précision

6.4.2 Si on transfert datax ver type 0-1

on peut evaluer notre auto-encodeur par cette méthode.
On a une précision de 0.78044140625 pour l'entraînement et 0.7601171875 pour le test. On peut dire que notre auto-encodeur conserve 75% de données

Réseau : `TanH(inputsize,100)` → `TanH(100,nb_class)` → `TanH(nb_class,100)` → `Sigmoide(100,inputsize)`

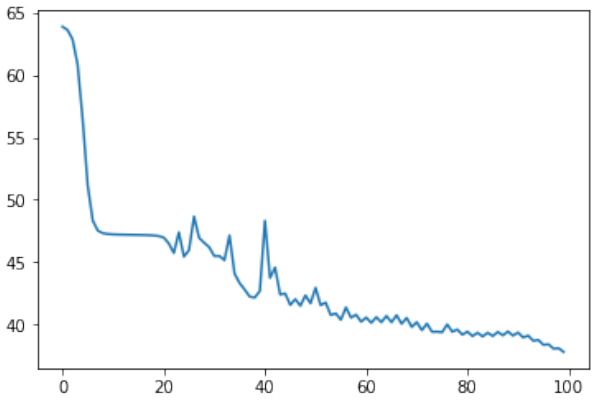

FIGURE 15 – Coût

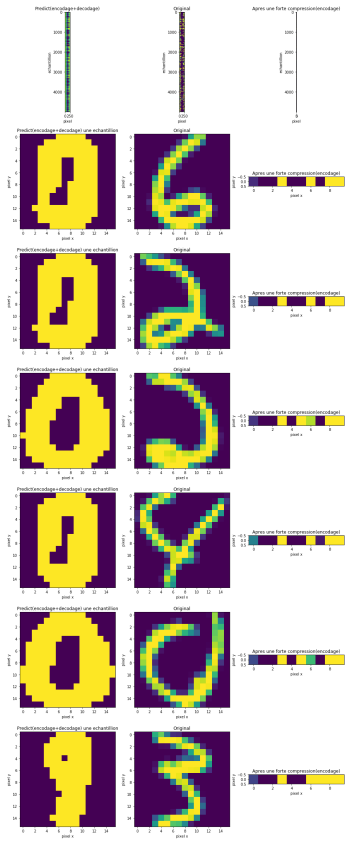

FIGURE 16 – Précision

6.4.3 Les performances en débruitage

Il n'y pas de sens si on bruite les données de type 0-1, donc on traite les données normalisées. La fonction *normalize* va retourner les resultats proches pour chaque echantillion.

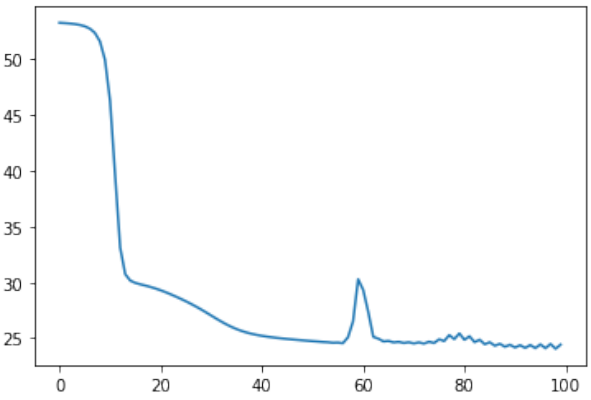

FIGURE 17 – Coût

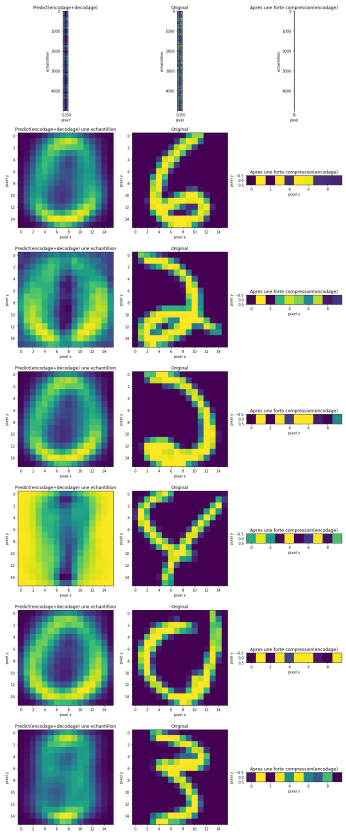

FIGURE 18 – Précision

6.4.4 Le pré-traitement par l'auto-encodeur et faire la classification

Après la normalization de donnée,
– le cas non bruité on a un graphe de loss avec la précision de 0.4124 et 0.37.
– le cas bruité on a un graphe de loss avec la précision de 0.4788 et 0.47.

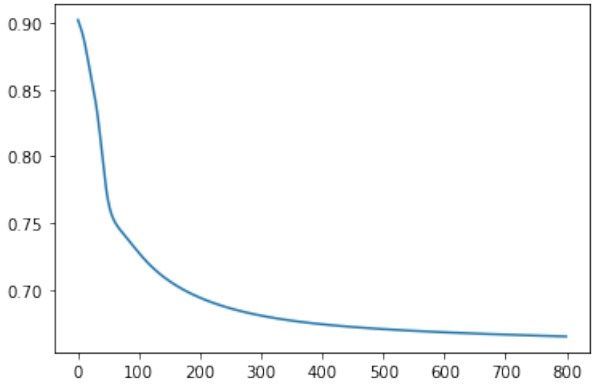

FIGURE 19 – bruite

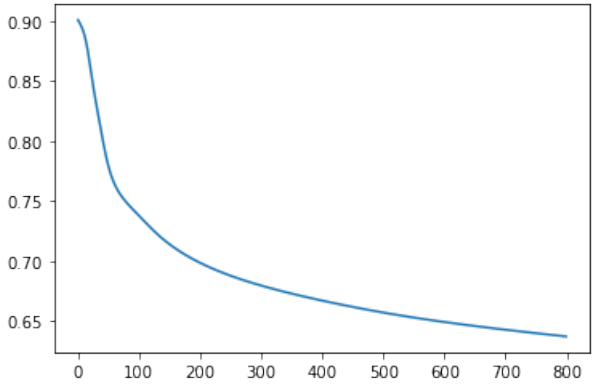

FIGURE 20 – non bruite

Conclusion :
1. le bruit va améliorer un peu mais pour la classification c'est pas nécessaire de trouver l'auto-encodeur
2. la performance de Réseau non convolutionnal est mauvaise
3. la performance de Réseau non convolutionnal va améliorer, si on juste entraîn pas grand nombre d'echantillions (25 par exemple)
4. on peut faire plus d'itérations et plus d'echantillions pour l'apprentissage pour améliorer la performance

Pour la fonction de coût :
1. CE est inutile pour tous les deux cas, parce que la dernière couche n'est pas Softmax, et Sigmoide minimise BCE si les résultat sont tous 1
2. MSE fonctionne mieux que BCE

6.5 Convolution

c'est quelque chiffres sur notre dataset.

```
nombre classe :  10
train :  (500, 256)
test :  (100, 256)
valeur de image : de  0.0  a  2.0
```

Après, on normalise les données (shape : (500, 256)) et ajouter une channel (shape : (500, 256, 1)). On teste la précision pour les cas où on utilise la couche de `MAX Pooling` et `AVG Pooling`.

Réseau : `Conv1D(chan_in=1,chan_out=32,k_size=3)` → `MaxPool1D(k_size=2,stride=2)` → `Flatten()` → `ReLU(input=4064,output=100)` → `Softmax(input=100,output=10)`

On fait `maxite = 100,batch_size = nb_train` juste pour tester si ça marche bien.

Max Pooling La précision est de 0.884 et 0.82.

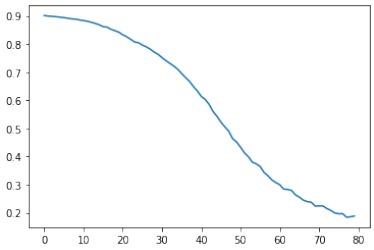

FIGURE 21 – Max Pooling

Avg Pooling La précision est de 0.87 et 0.83.

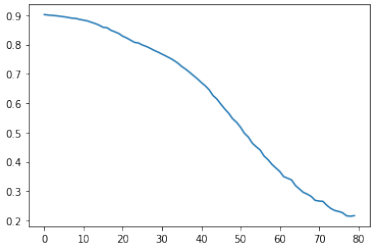

FIGURE 22 – AVG Pooling