Projet - Réseau de neurones : DIY

Auteur
Youva Addad

Client
Nicolas BASKIOTIS

Résumé

Le but de ce projet est d'implémenter un réseau de neurones version modulaire, plus particulièrement les modules les plus utilisés tel que le module linaire qui permet d'applique une transformation linéaire aux inputs $y = xA^T + b$.
Des activations pour pouvoir appliqué une non-linéarité tel que le tanh qui applique la fonction élément par élément $tanh(x) = \frac{exp(x)+exp(-x)}{exp(x)-exp(-x)}$, ainsi que la sigmoid qui applique la fonction élément par élément $Sigmoid(x) = \frac{1}{1+exp(-x)}$.
Ensuite la réalisation d'un module séquentiel, les modules y seront ajoutés dans l'ordre dans lequel ils sont passés dans le constructeur. Alternativement, un orderedDict de modules peut également être transmis.

$model = Sequential(Linear(10, 5), TanH(), Linear(5, 1)Sigmoid())$

On a de plus rajouté un SoftMax pour pouvoir faire du multi-class. Il applique la fonction Softmax à un module d'entrée à n dimensions en les remettant à l'échelle de sorte que les éléments de sortie à n dimensions se trouvent dans la plage [0,1] et la somme à 1.

$SoftMax(x_i) = \frac{exp(x_i)}{\sum_j^n exp(x_j}$
Pour pouvoir faire de la compression d'image (information) on a réalisé un autoEncoder, l'autoEncoder réduit les dimensions des entrées.
Un exemple d'architecture d'autoEcoder :
$Encodage$: $Linear(256, 100) \rightarrow TanH() \rightarrow Linear(100, 10) \rightarrow TanH()$
$Dcodage$: $Linear(10, 100) \rightarrow TanH() \rightarrow Linear(100, 256) \rightarrow Sigmoide()$

Pour finir nous avons réalisé le Conv1d et le Conv2d avec les pooling (maxPool et avgPool)

Table des matières

1 Module Réalisé

Note : tous les modules implémentés ont été testé avec les modules de PyTorch.

1.1 Module Linéaire/Activation

1. Linear
2. Tanh
3. Sigmoid
4. Sequantial
5. SoftMax
6. LogSoftMax
7. Threshold
8. ReLU (héritage de Threshold)

1.2 AutoEncoder

1. AutoEncoder

1.3 Module Convolutionnel

1. Conv1d
2. Conv2d
3. MaxPool1d/AvgPool1d
4. MaxPool2d

1.4 Loss Function

1. MSE
2. MAE
3. Binary Cross Entropy
4. Cross Entropy
5. LogSoftMax Cross Entropy(Implémentation brute pas de class explicite NLLLoss+logSoftMax)
6. NLLLoss
7. Hinge
8. CrossEntropyCriterion (version plus stable de CELoss utilisant NLLLoss+LogSoftMax)

2 Linéaire

2.1 Expérimentation

Dans cette section nous avons testé notre implémentation du module linéaire avec LinearRegression de scikit-learn avec des données générés avec la méthode make_regression de scikit-learn, 100 exemples et 1 features et un bruit de 30 ce qui nous donne comme frontiére (w*X+b). Pour notre module linear $model = Linear(1, 1)$ une entrée une sortie.
Nous avons utilisé la MSELoss pour l'optimisation (correspond au test Test_LinearVSLinREg)

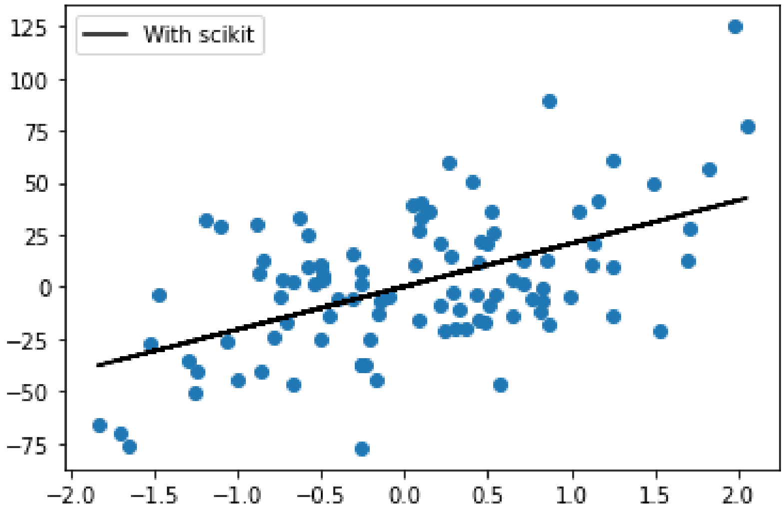

FIGURE 1 – La droite de décision avec Scikit Learn

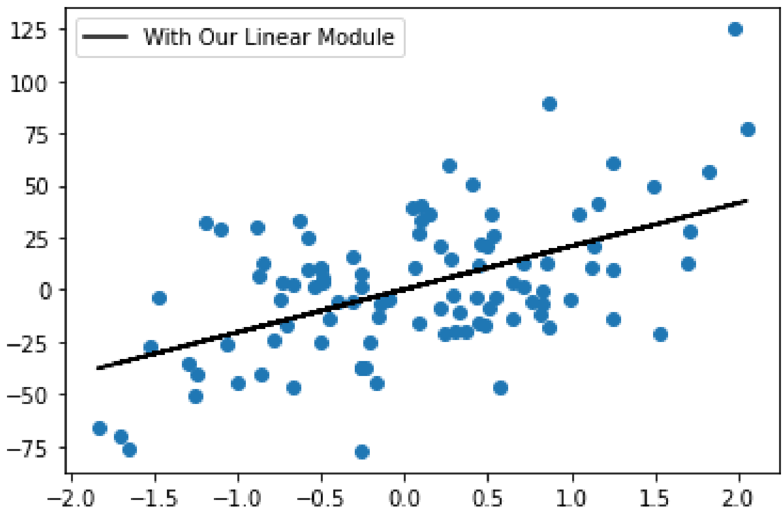

FIGURE 2 – La droite de décision avec Le module Linéaire

On remarque que les deux droite ce colle parfaitement et la convergence est atteinte rapidement.

3 Non-Linear

3.1 Dérivés

$$tan(x) = \frac{exp(x) + exp(-x)}{exp(x) - exp(-x)} => tan'(x) = \frac{(exp(x) - exp(-x))^2 - (exp(x) + exp(-x))^2}{(exp(x) - exp(-x))^2} \tag{1}$$

$$tan'(x) = 1 - tan(x)^2 \tag{2}$$

$$sigma(x) = \frac{1}{1 + exp(-x)} => sigma'(x) = \frac{exp(-x)}{(1 + exp(-x))^2} = \frac{1}{1 + exp(-x)} * \frac{exp(-x)}{1 + exp(-x)} \tag{3}$$

$$sigma'(x) = sigma(x) * (1 - sigma(x)) \tag{4}$$

3.2 Expérimentation

3.2.1 Lineairement séparable

Tout d'abord ici pour la génération des datas nous avons utilisé gen_arti de mltools des précédents TME avec un sigma=0.4 et le modéle suivant :
$linear1 = (Linear(2, 2), TanH(), Linear(2, 1), Sigmoide())$

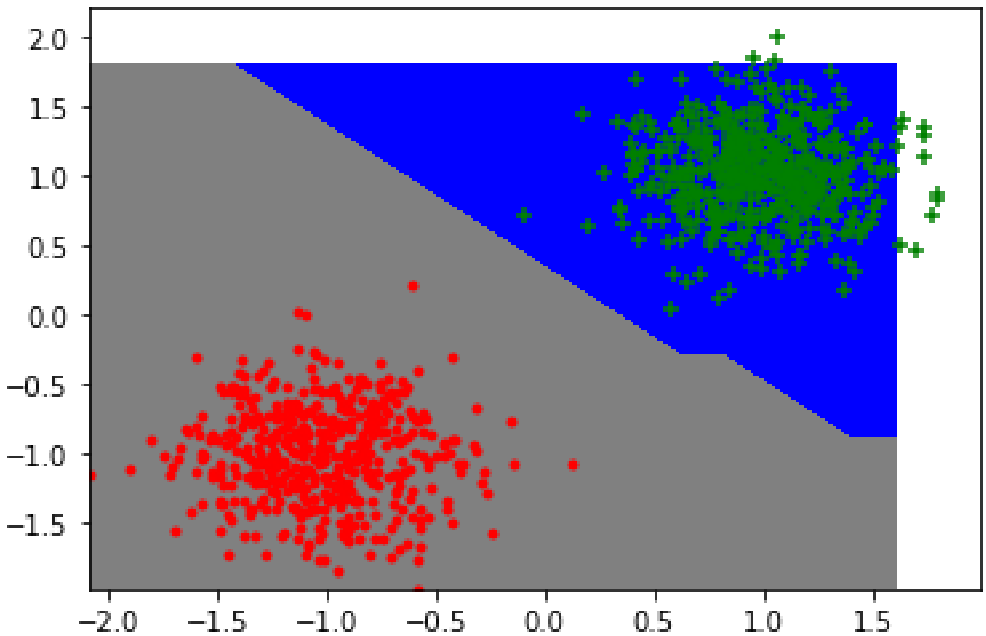

Figure 3 – Classification de données linearement séparable

On remarque que la décision est parfaitement bien séparé avec des données linearement séparable.

3.2.2 XOR

: Deuxiéme teste avec un gen_arti(data_type=1), avec le même module que précédemment.

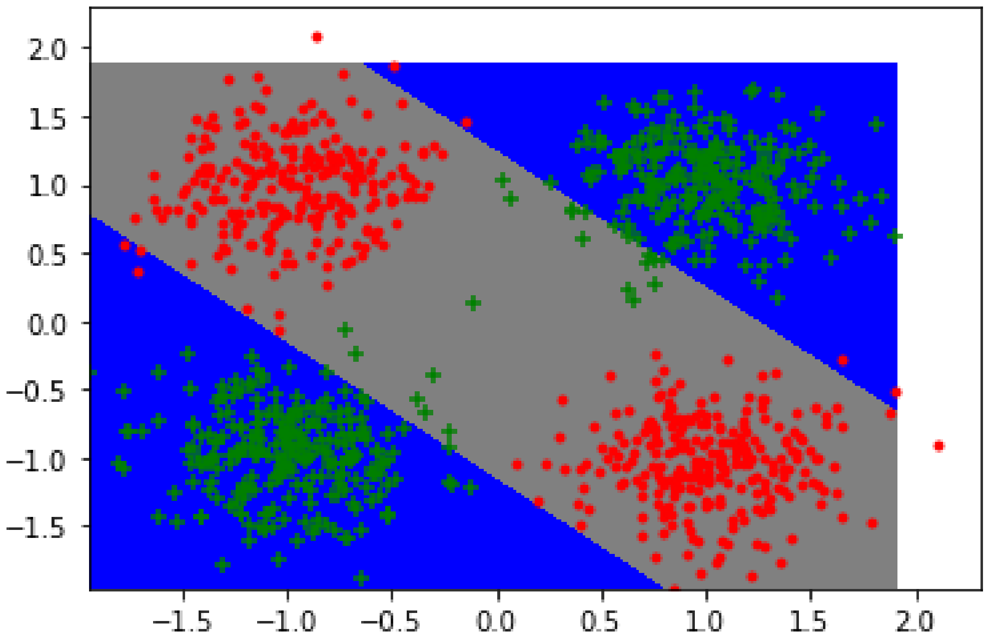

FIGURE 4 – Classification Probléme du XOR

On remarque ici aussi il arrive a bien séparé les données, noté tout de même qu'on utilise toute le taille des données pour l'apprentissage du modéle.

4 Sequential

Comme annoncé précédemment le constructeur de la classe Sequential est :

$model = Sequential(Linear(10, 5), TanH(), Linear(5, 1)Sigmoid())$

mais peut aussi prendre un dictionnaire ordonné, de plus la classe Optimizer est une classe abstraite, SGD hérite de cette class, ce qui permet d'implimenté d'autre Optimizer autre que le SGD.

4.1 Exprémentation

Avec maxIter=100 et batch_size et le modéle $Sequential(Linear(2, 4, bias = True), TanH(), Linear(4, 1, bias = True), Sigmoide())$

4.1.1 Linéarement séparable

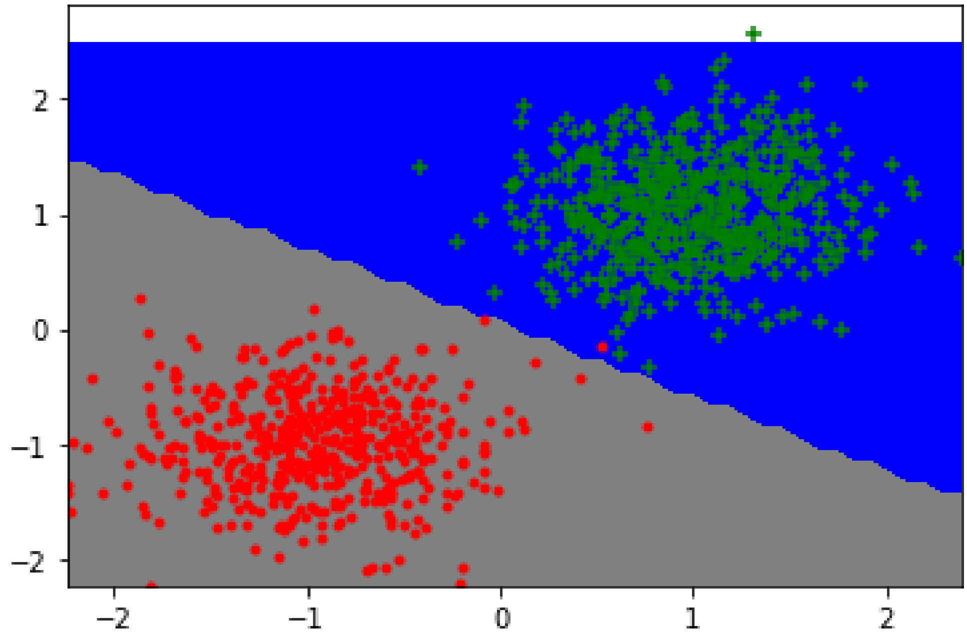

FIGURE 5 – Sequential Classification boundary

Comme précédemment il arrive bien a séparer les données, mais ici comme il travaille sur un batch de données cella permet une convergence plus rapide en peu d'iteration.

4.1.2 XOR

On augmente maxIter=500 le probléme étant plus defficile qu'un classification normale avec le même modéle

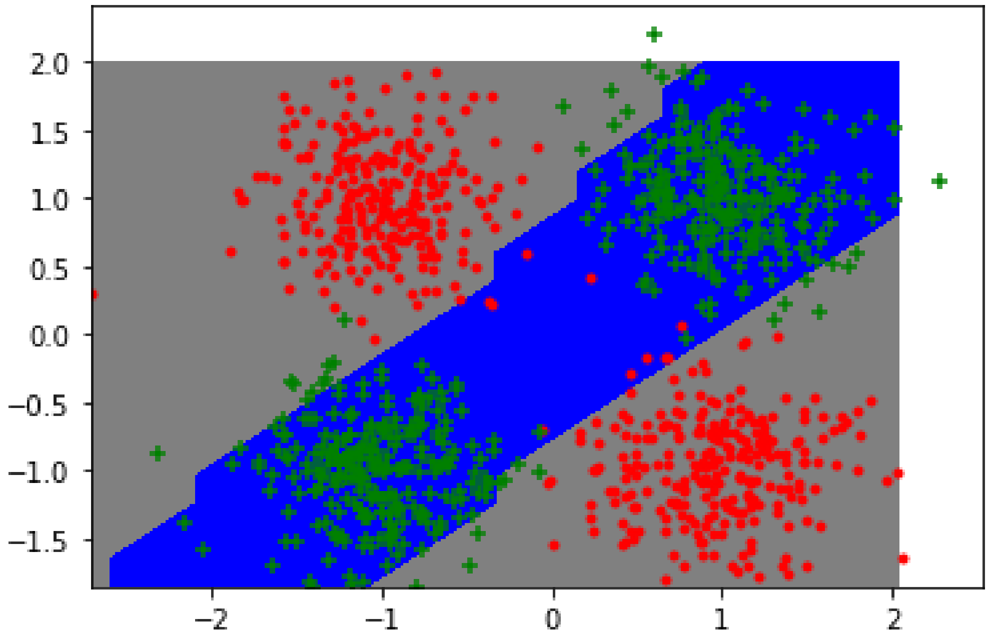

FIGURE 6 – Sequential XOR max$_{i}ter = 500 avec 2 modules lineare$

4.1.3 XOR avec un réseau profond

Ici on va réaliser un réseau avec plus de 2 couches pour voir les performances d'apprentissage, toute en baissant le maxIteration $Sequential(Linear(2, 4, bias = True), TanH(), Linear(4, 4, bias = True), Sigmoide(), Linear(4, 1, bias$ $True), TanH())$

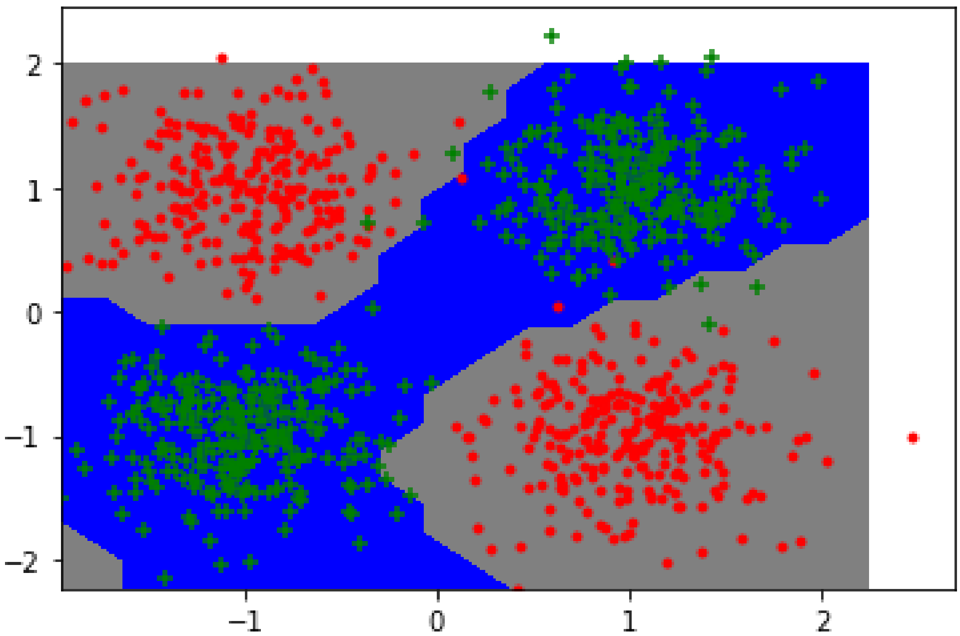

FIGURE 7 – Sequantial XOR avec le maxIter=200

On remarque ici que on augmentant le nombre de hidden layer il arrive parfaitement a apprendre les données avec maxIter=200

$Sequential(Linear(2, 50, bias = True), TanH(), Linear(50, 50, bias = True), Sigmoide(), Linear(50, 1, bias = True), TanH())$

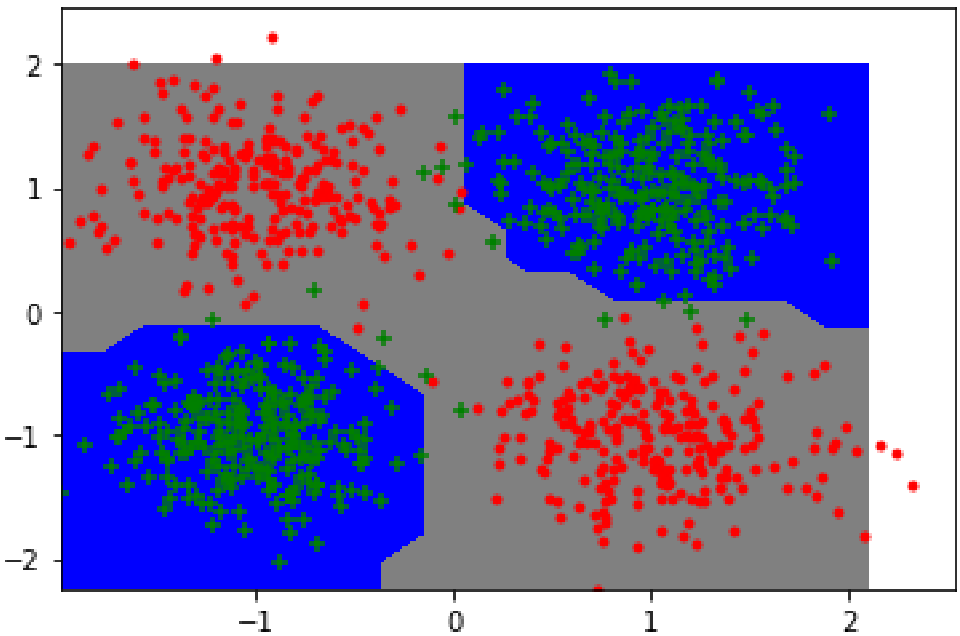

FIGURE 8 – 3 hidden layer avec 50 neurones pour la sortie de la premiere couche en 20 iterations

4.1.4 Echéquier

Dans cette partie nous avons essaie de résoudre le probléme échequier avec le model

$Sequential(Linear(2, 128, bias = True), TanH(), Linear(128, 64, bias = True), Sigmoide(), Linear(64, 1, bias = True), TanH())$

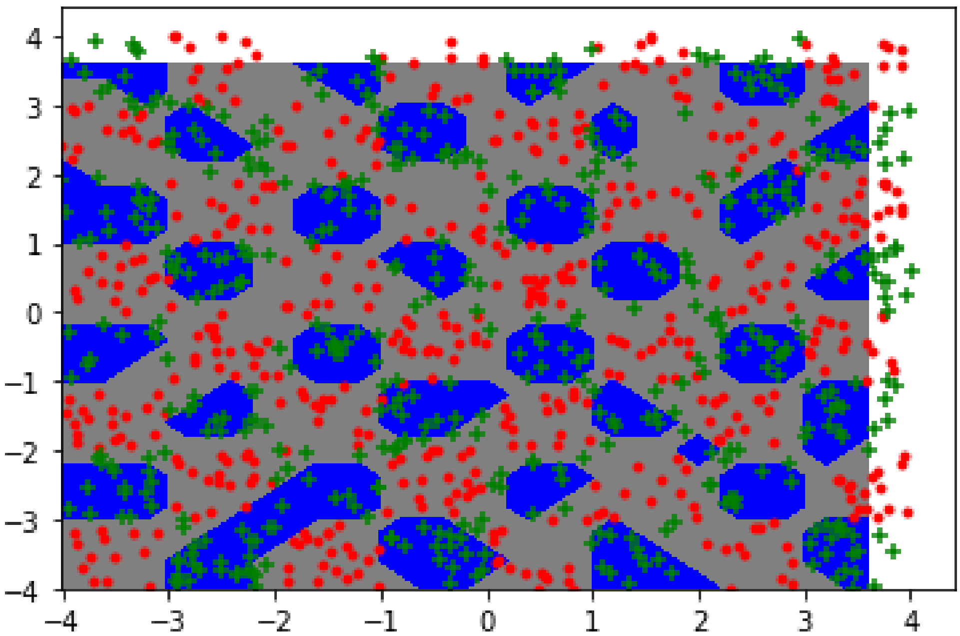

FIGURE 9 – Echequier avec maxIter=500 et 3 couches cachée

On arrive raisonnablement a apprendre l'échéquier, on remarque que avec maxIter=500 on arrive a discriminer les classes. Donc en rajoutant des couches et neuronne supplémentaire nous permet de résoudre des problémes plus complexes, mais plus le probléme est complexe plus faudra augmenter le maxIter.

5 Multi-Classe

Ici pour l'implémentation du SoftMax nous avons soustrait le max de chaque ligne pour des problémes computationnel, l'exponentiel augmente trés rapidement, il peux potentiellement donné des résultats faussé sans la normalisation.

5.1 Dérivée

$$SoftMax(x_i) = \frac{exp(x_i)}{\sum_j^k exp(x_j)} => SoftMax'(x_i) = \begin{cases} SoftMax(x_i) * (1 - SoftMax(x_j)) & \text{si } i = j \\ SoftMax(x_i) * (-SoftMax(x_j)) & \text{sinon.} \end{cases} \quad (5)$$

mais comme il prend en vecteur de supervision un vecteur one hot encoding alors la dérivée des $i \neq j$ va être annuler par delta.

Pour les Losses on peut aussi bien utilisé SoftMax + CrossEntropyLoss ou directement une LogSoftMax-CrossEntropy.

5.2 Expérimentation

Avec maxIter=200 et batch_size=100 avec le modéle suivant : avec input la taille d'un image et output le nombre de classe. $Sequential(Linear(input, 128), TanH(), Linear(128, 64), TanH(), Linear(64, output), SoftMax())$ Ainsi que une loss CrossEntropique $loss = CrossEntropyLoss()$

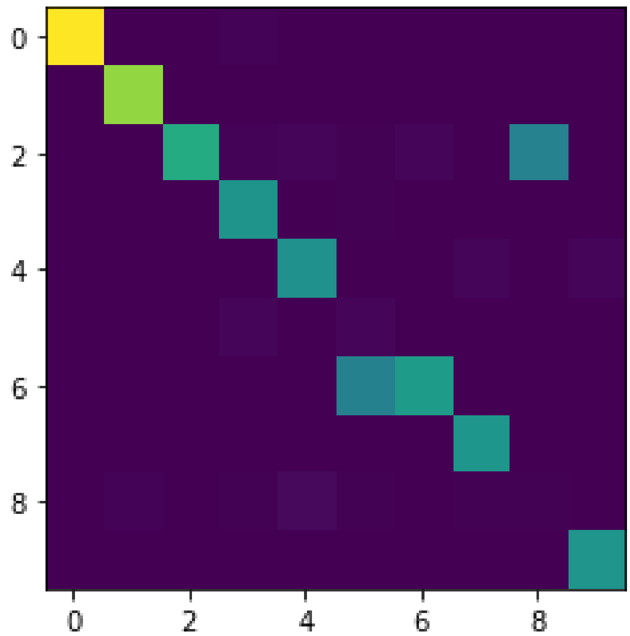

FIGURE 10 – MultiClass Confusion

Nous obtenons un score en accuracy de 0.82 ce qui est raisonnable, mais tout de meme il faudra signalé que sensible a l'initialisation une autre relance donne d'autre résultat.

6 Auto-Encodeur

Note1 : nous avons reformuler le backward de la BCELoss
Note2 : l'apprentissage tarde un peu du fait du choix du batch_size et maxIter, donc faudra baisser pour etre plus rapide, il s'agit ici de converger vers la meilleur solution et non pas d'etre rapide.

Reformulation du backward :$- > -(y-1)/1 - yhat - y/yhat- > (1-y)/1 - yhat - y/yhat- > (1-y) * yhat - y(1-yhat)/(1-yhat)yhat- > yhat - y * yhat - y + y * yhat/(1-yhat) * yhat- > yhat - y/(1-yhat) * yhat$

6.1 Visualiser les images reconstruites après une forte compression

Dans cette partie nous avons utilisé l'architecture mentionner dans le projet pour pouvoir fortement compresser les données, donc de 256 jusqu'a 10.
Nous avons testé les differents hyperparametre ainsi que une initialisation uniforme du module Linear de la sorte :
$bound = 1/np.sqrt(input)$
$np.random.uniform(-bound, bound, (input, output))$.
On essaie de reconstruire les données USPS,un echantillon :

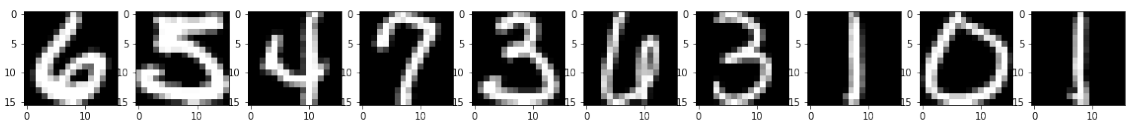

FIGURE 11 – Echantillon

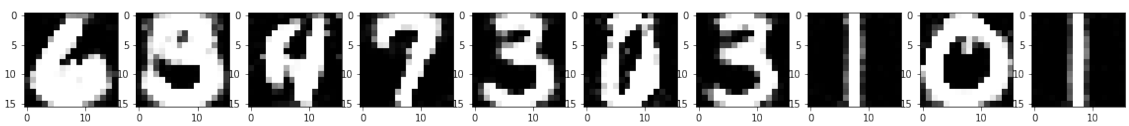

FIGURE 12 – Initialisation 2 * (np.random.rand(input, output) - 0.5), maxIter=120, batch=20

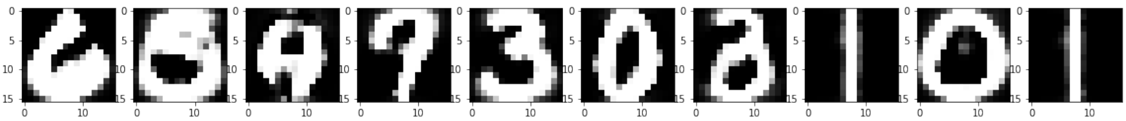

FIGURE 13 – Uniform initialisation maxIter=120, batch=20

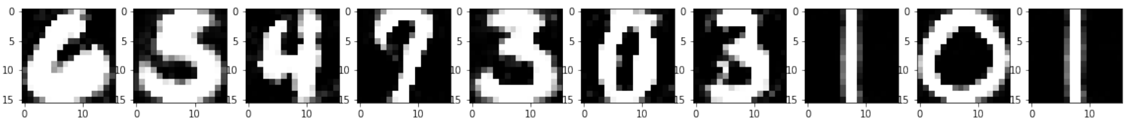

FIGURE 14 – Initialisation 2 * (np.random.rand(input, output) - 0.5), maxIter=120, batch=10

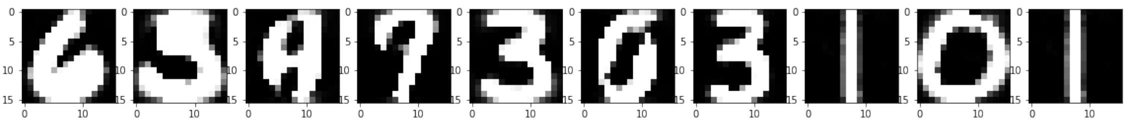

FIGURE 15 – Uniform maxIter=120, batch=10

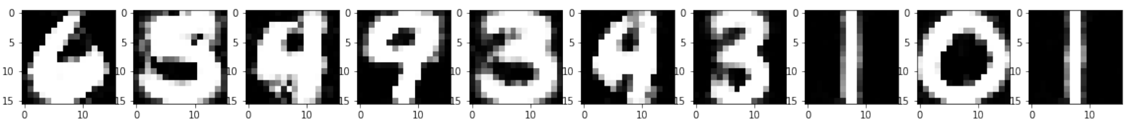

FIGURE 16 – Random maxIter=150, batch=10

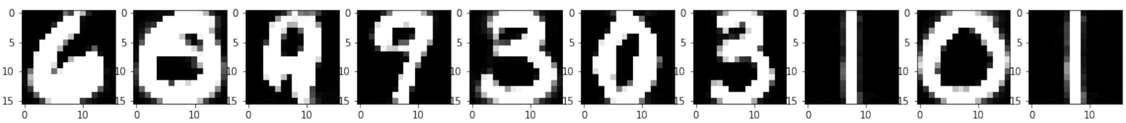

FIGURE 17 – Uniform maxIter=150, batch=10

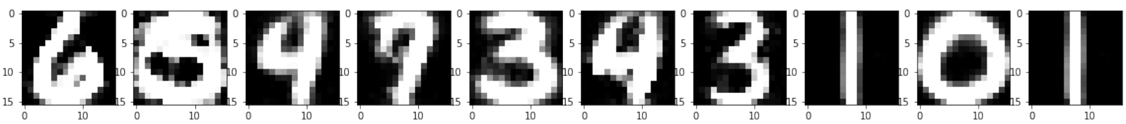

FIGURE 18 – Initialisation 2 * (np.random.rand(input, output) - 0.5), maxIter=200, batch=10

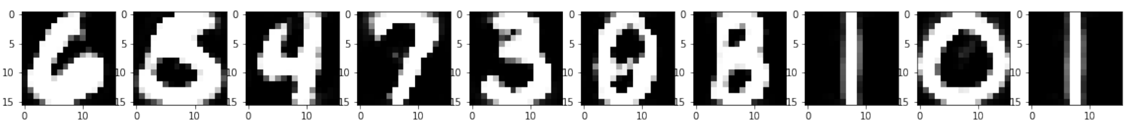

FIGURE 19 – Uniform maxIter=200, batch=10

On remarque tout de meme une petite sensibilité a l'initialisation, mais sur ces scenario nous avons décidé de laissé l'intialisation 2 * (np.random.rand(input, output) - 0.5), qui permet de donné des résultats mieux.

6.2 Débruitage d'image

Dans cette partie nous avons bruité les images en utilisant la normale de tensorflow multiplie par un facteur de noise=0.2.

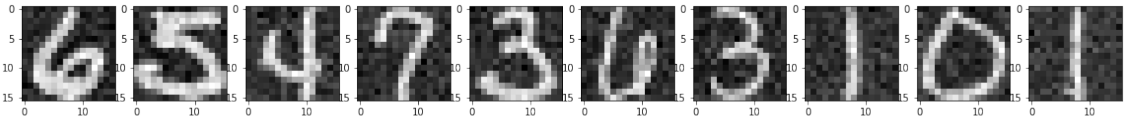

Figure 20 – Bruitage Image noise=0.2

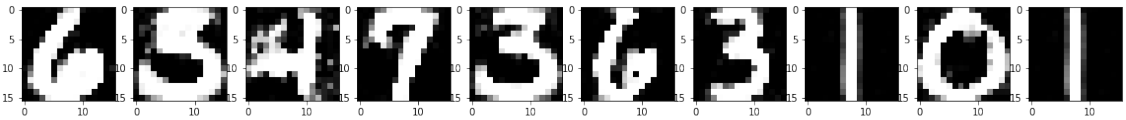

Figure 21 – Debruitage

Apprentissage avec maxIter=200 et batch_size= 10 nous remarquons qu'on arrive bien a débruité les images et a reconnaitre parfaitement les 10 premiers chiffres.

On a augmenté le facteur de noise ici qui est egale a 0.5

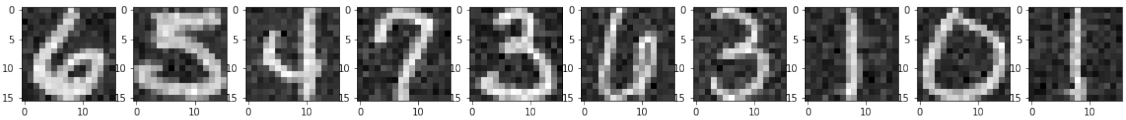

Figure 22 – Bruitage Image avec noise 0.5

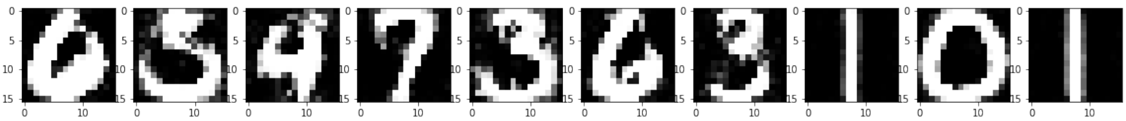

Figure 23 – Debruitage

Avec 0.5 de noise aussi on arrive parfaitement a distigué les chiffres.

Pour un facteur de noise=1

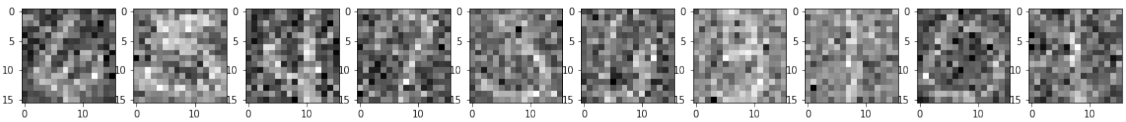

Figure 24 – Bruitage Image avec noise 1

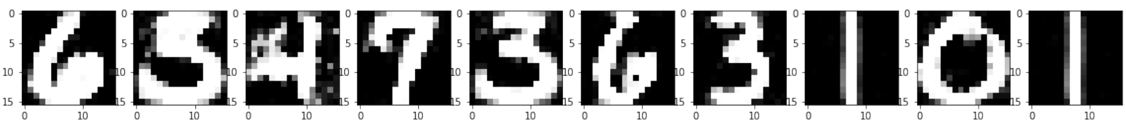

Figure 25 – Debruitage

On arrive raisonnablement a reconnaitre les chiffres avec un fort bruitage et une forte compression, difficilement visible a l'oeil nu.

6.3 Détection d'anomalies

Dans cette partie on a entraîner un auto-encodeur pour détecter les anomalies sur l'ensemble de données ECG5000.
On utilisera ici une version simplifié où chaque exemple a été étiqueté soit 0 (correspondant à un rythme anormal), soit 1 (correspondant à un rythme normal). On souhaite donc identifier les rythmes anormaux.

Pour cella nous avons utilisé une MAELoss(absolute loss ici), ainsi que le modéle suivant :

$encoder = Sequential(Linear(140, 32), TanH(), Linear(32, 16), TanH(), Linear(16, 8), TanH())$
$decoder = Sequential(Linear(8, 16), TanH(), l5 = Linear(16, 32), TanH(), Linear(32, 140), Sigmoide())$

Le modéle précédent a été entrainer sur les rythmes normaux

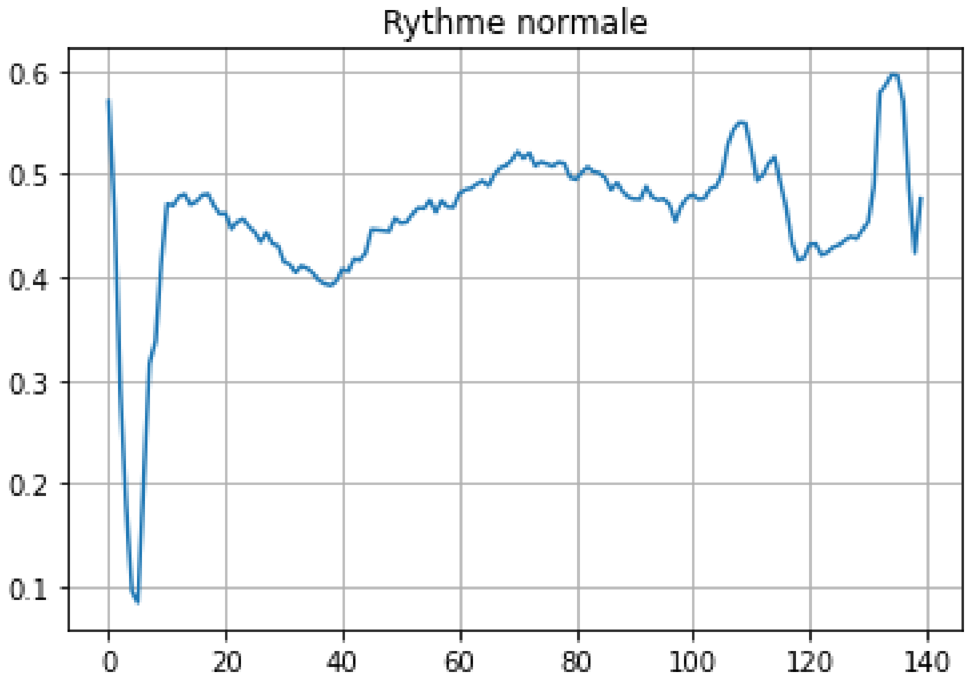

Figure 26 – Rythme normale

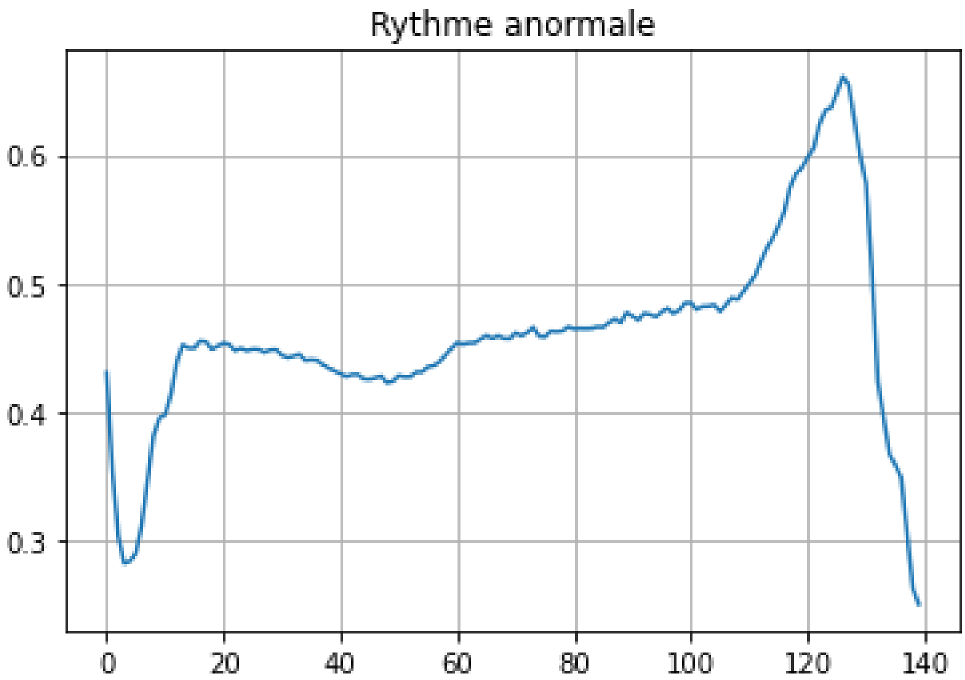

Figure 27 – Rythme anormale

Donc la reconstruction sur les données de test on donnée les figures suivante :

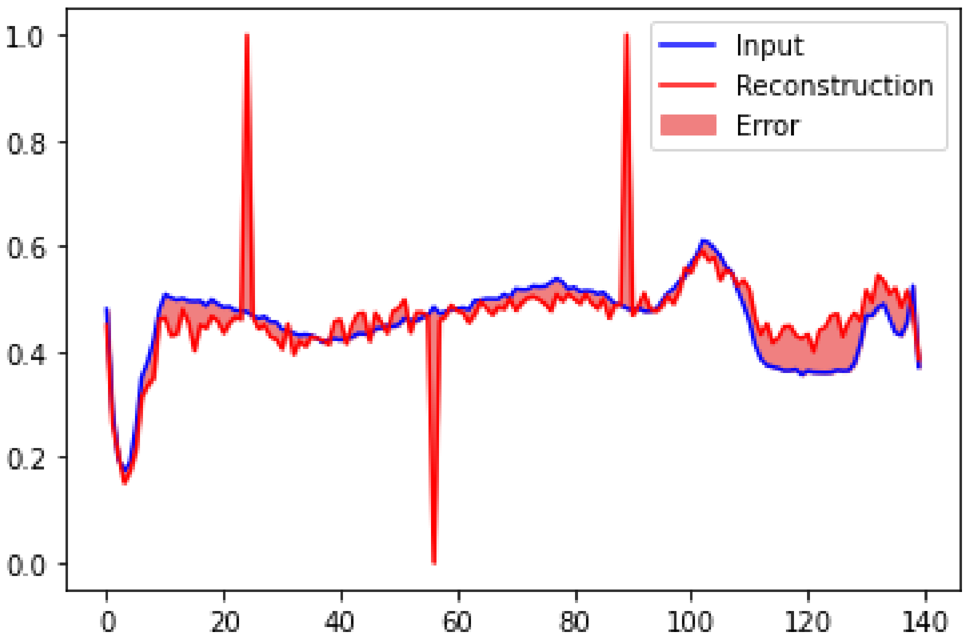

Figure 28 – Test Data rythme normale

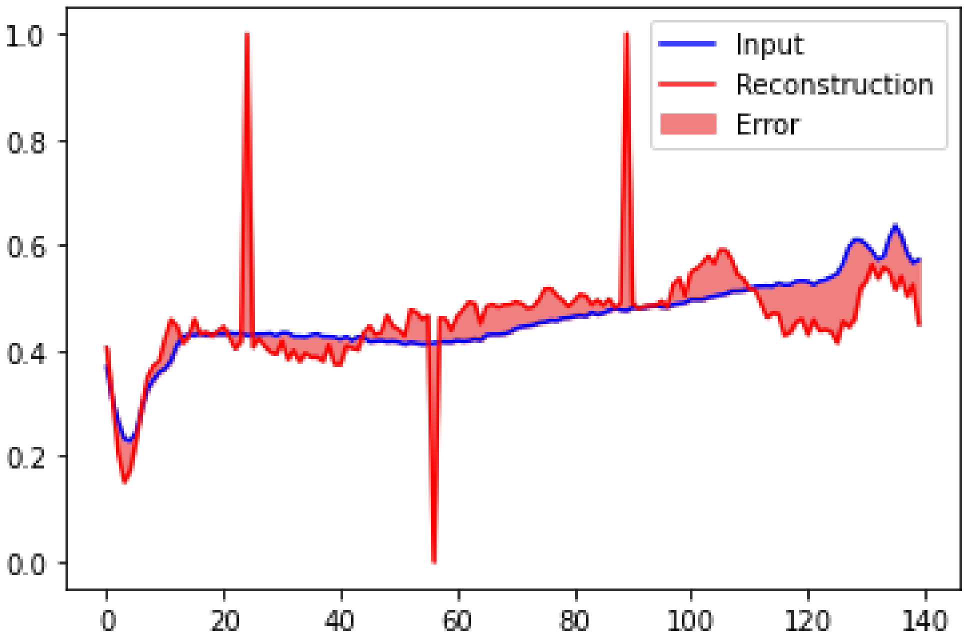

FIGURE 29 – Test Data rythme anormale

Nous avons donc utilisé un seuil pour affecter les rythmes cardiaque, nous obtenons les performances suivante en test :

Accuracy = 0.943
Precision = 0.9941060903732809
Recall = 0.9035714285714286

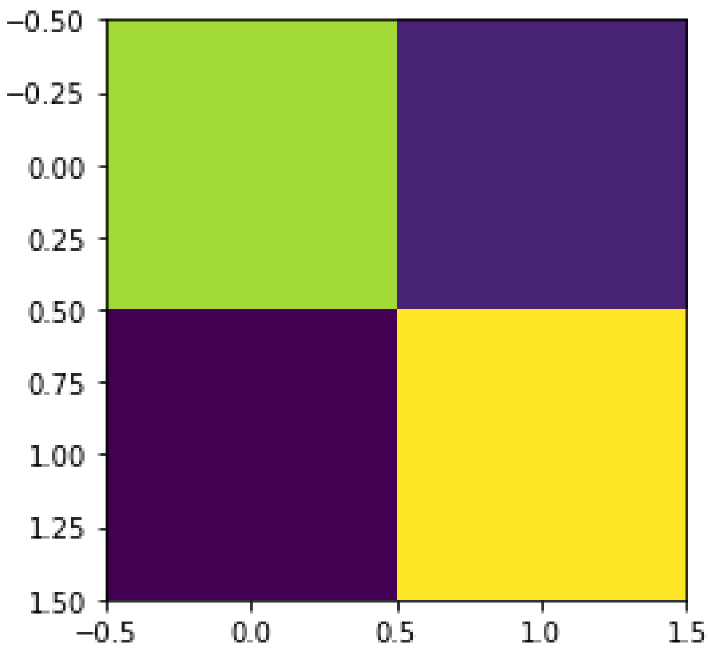

FIGURE 30 – Matrice de confusion de bonne classification

Nous arrivons parfaitement bien a detecté les rythmes anormaux avec cette architecture modulaire.
mais nous remarquons juste un petit pic anormale dans la reconstruction, cella dis le modéle marche bien et arrive bien a détecter les anormaux.

6.4 Clustering avec K_means & Visualisation en 2D

Dans cette partie nous travaillons sur les données usps de taille (16*16), nous avons normaliser les données, nous obtenons tout d'abord la visualisation suivante :

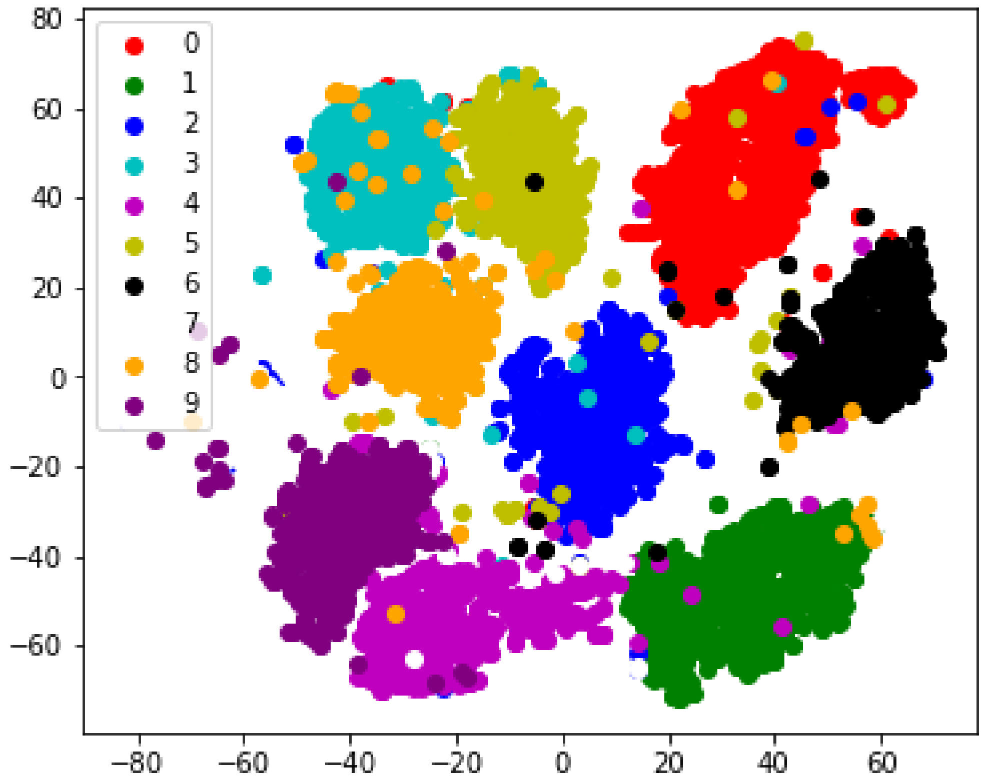

FIGURE 31 – Visualisation TSNE

On applique un k_means de nb_component 10 nous aurons donc :

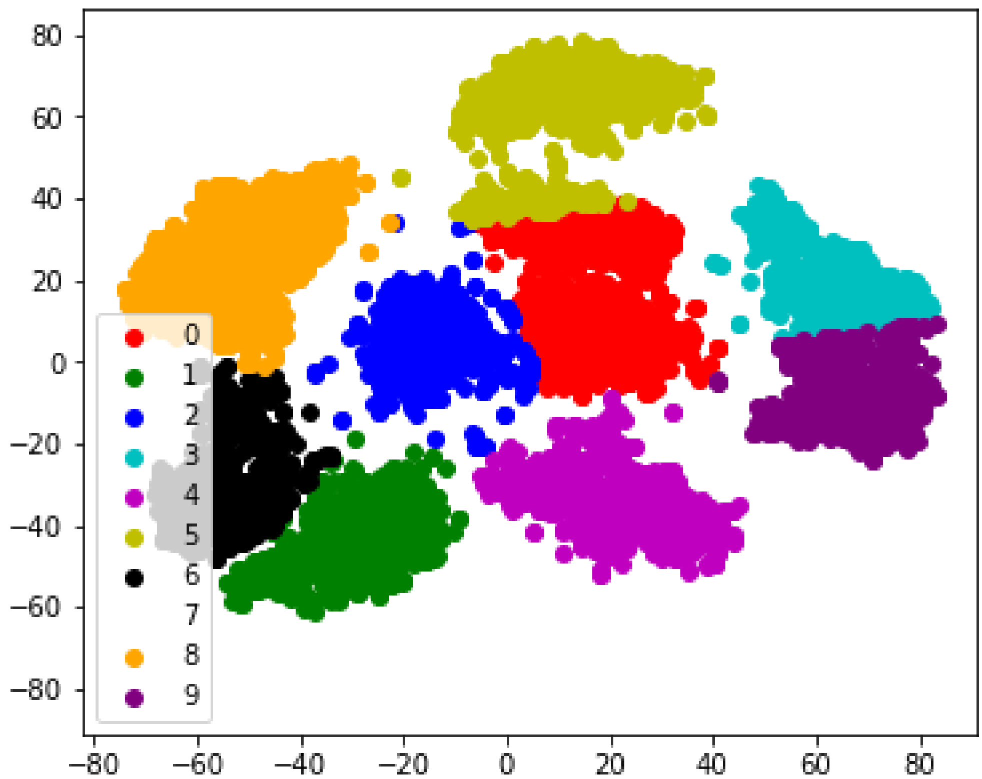

FIGURE 32 – KMEANS Visualisation TSNE

Avec le modéle suivant :
$encoder = Sequential(Linear(256, 32), TanH(), Linear(32, 16), TanH(), Linear(16, 10), TanH())$
$decoder = Sequential(Linear(10, 16), TanH(), Linear(16, 32), TanH(), Linear(32, 256), Sigmoide())$
Avec une MSELoss nous obtenons donc cette representation

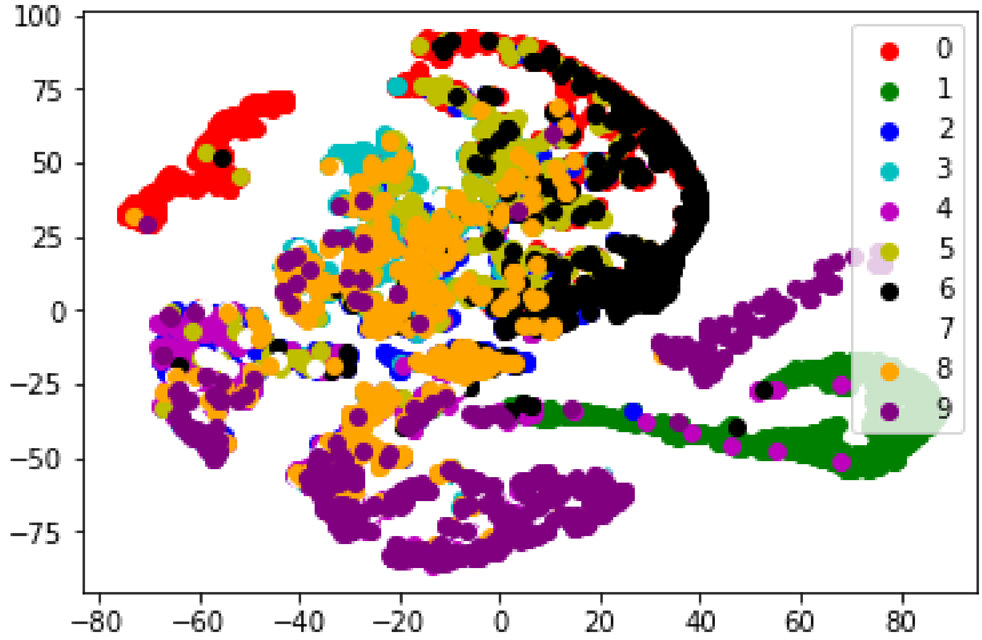

FIGURE 33 – AutoEncoder Clustering avec visualisation TSNE

7 Réseau neuronal convolutif

7.1 Conv1d

Tout d'abord nous avons comparer notre résultat au résultat de l'implémentation de pytorch ce qui donne :

```
]: import torch
   from torch import nn
```

```
]: #Le module Conv1d de Pytorch
   k_size=3
   chan_in=16
   chan_out=33
   m = nn.Conv1d(16, 33, 3, stride=2,bias=False)
   param=m.weight.detach().numpy().transpose(2,1,0)
   input = torch.randn(20, 16, 50)
   m(input).detach().numpy().transpose(0,2,1)[0].max()

   1.6579295
```

```
]: #Le ConvlD que nous avons realisée
   lay=Conv1D(3,16, 33,  stride=2,bias=False)
   lay._parameters=param
   tar=input.detach().numpy().transpose(0,2,1)
   lay.forward(tar)[0].max()

   1.6579295
```

FIGURE 34 – Comparaison de notre modele et celui de pytorch-Conv1d

le conv1d utilise des produits scalaire entre fenetre, ce qui génére enormement de temps de process, il mets enormement de temps a s'excuter, c'est pour cela qu'on a besoin d'un GPU pour l'execution, et une programmation au niveau machine comme c++ pour reduire le temps de latence.

Nous avons de plus tester l'architecture sur les données USPS pour la classification de degit

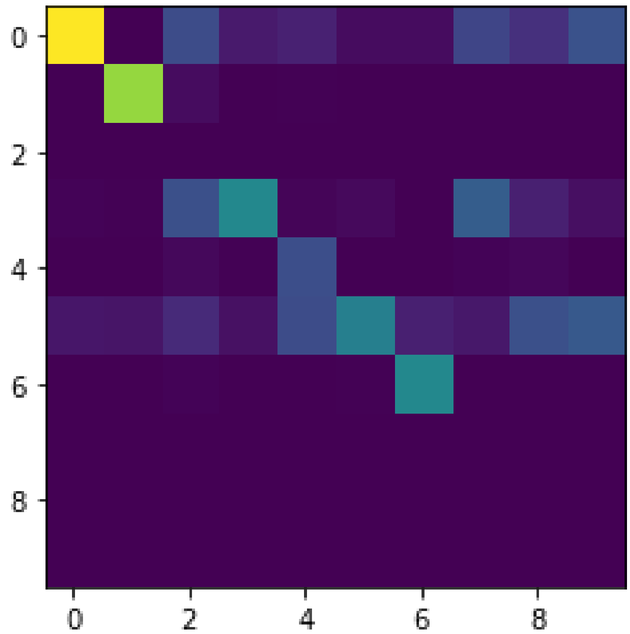

FIGURE 35 – Convolution degit matrix confusion

Nous avons testé aussi sur la classification d'iris nous obtenant un score de : 0.98 avec le modéle suivant : $Sequential(Conv1D(2, 1, 16), MaxPool1D(2, 2), Flatten(), Linear(16, 3), SoftMax())$ avec un maxIter=200 et batch_size=20

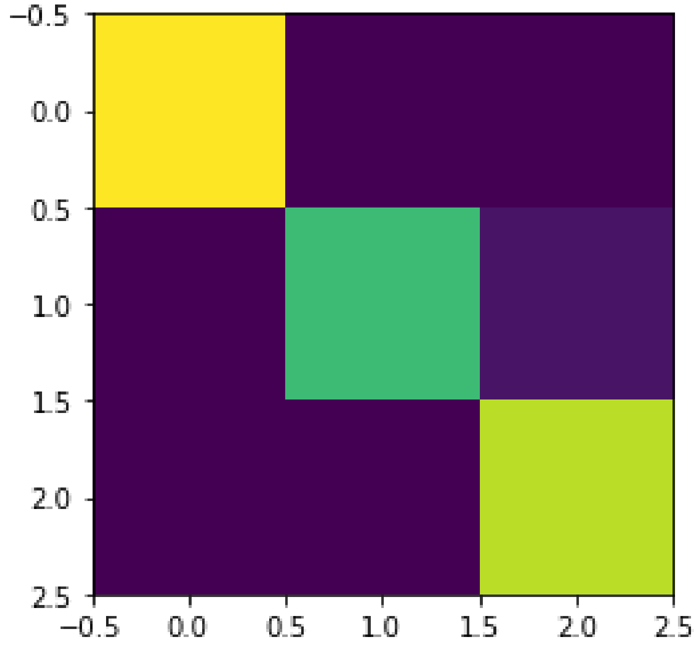

FIGURE 36 – Matrice de confusion de classification d'iris

On pourra dire donc que sa permet de bien classifier avec le CNN, et que ici sa marche relativement vite.

7.2 Conv2d

De meme ici nous avons comparer le forward de pytorch et le notre :

```
N=20
Cin=2
H=5
W=10
k_size=3
chan_out=2
input = torch.randn(N, Cin, H, W)
layer=nn.Conv2d(Cin, chan_out, k_size, stride=1)
param=layer.weight.detach().numpy().transpose(2,3,1,0)
bias=layer.bias.detach().numpy()
layer(input).detach().numpy().transpose(0,2,3,1)[2][1].min()
```

```
-0.8825046
```

```
layer2=Conv2D(k_size,Cin,chan_out)
layer2._parameters=param
layer2._bias=bias
output=input.detach().numpy().transpose(0,2,3,1)
layer2.forward(output)[2][1].min()
```

```
-0.8825046271085739
```

Figure 37 – Comparaison de notre modele et celui de pytorch-Conv2d

Dans cette derniere partie nous avons fait un reshape des données USPS pour quel soit en taille (16,16), le modéle utilisé est le suivant :

$$Sequantial(Conv2D(3,1,32), MaxPool2D(2,2), Flatten(), Linear(1568,100), ReLU(), Linear(100,10), SoftMax())$$

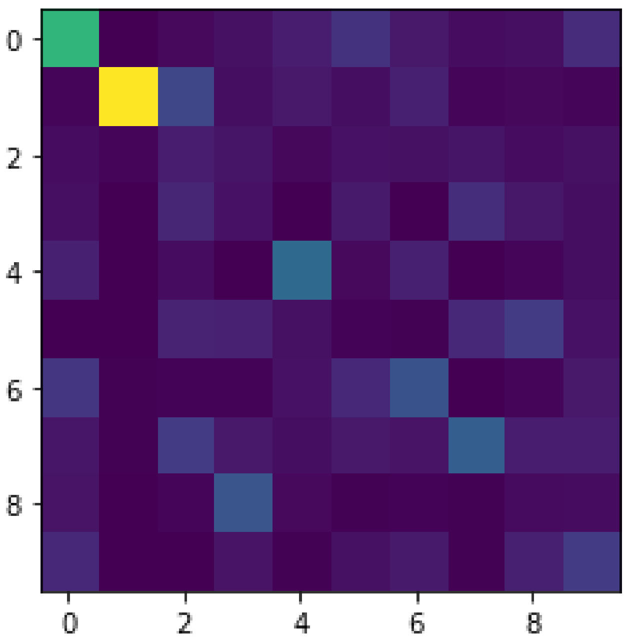

Figure 38 – Matrice de confusion classification d'image avec une MSELoss

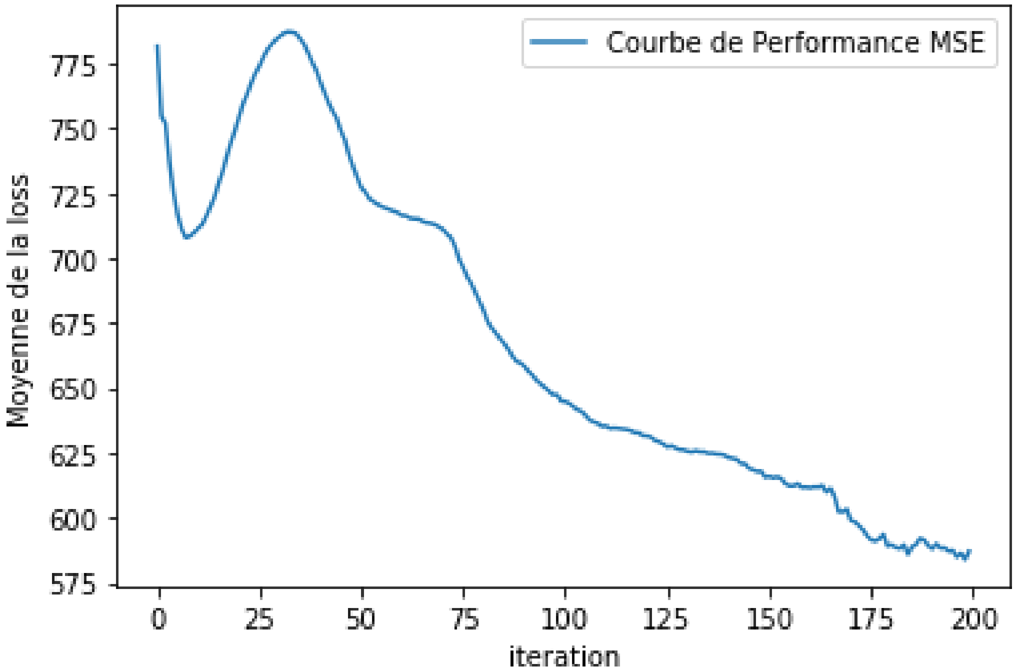

Figure 39 – Courbe de performance (moyenne de la loss) en fonctions des itérations
A noté que nous avons fait la somme pour un step, et moyenne sur une epoche.

Sur le probleme de classification d'image, on remarque que la loss déminue au fur et a mesure des itérations, donc si on avait augmenté le nombre d'itération sa convergerai vers une meilleur solution.

8 Conclusion

Nous avons réussie a implémenté tout les modules demandés, et nous avons teste plusieur type d'initilisation. Dans le Linear nous avons decidé de garde l'initialisation 2*rand - 0.5 mais dans les modules convolutionnel nous avons choisis une initialisation uniforme avec $np.random.uniform(-bound, bound, (k_size, chan_in, chan_out))$ et $bound = 1/np.sqrt(chan_in * k_size)$, on effet nous avons remarqué une sensibilité a l'intialisation.