Rapport de Projet

Machine Learning

Dans le cadre du
MASTER de Sorbonne Université

Spécialité
Informatique

Parcours
DAC

ZUO Nicolas *3702107* Étudiant
ZHUANG Pascal *3701123* Étudiant
BASKIOTIS Nicolas Encadrant

Table des Matières

1 Introduction

L'objectif de ce projet est d'implémenter un réseau de neurones. L'implémentation est inspirée des anciennes versions de pytorch (en Lua, avant l'autograd que vous verrez l'année prochaine) et des implémentations analogues qui permettent d'avoir des réseaux génériques très modulaires. Chaque couche du réseau est vu comme un module et un réseau est constitué ainsi d'un ensemble de modules. En particulier, les fonctions d'activation sont aussi considérées comme des modules (cf Figure 1).

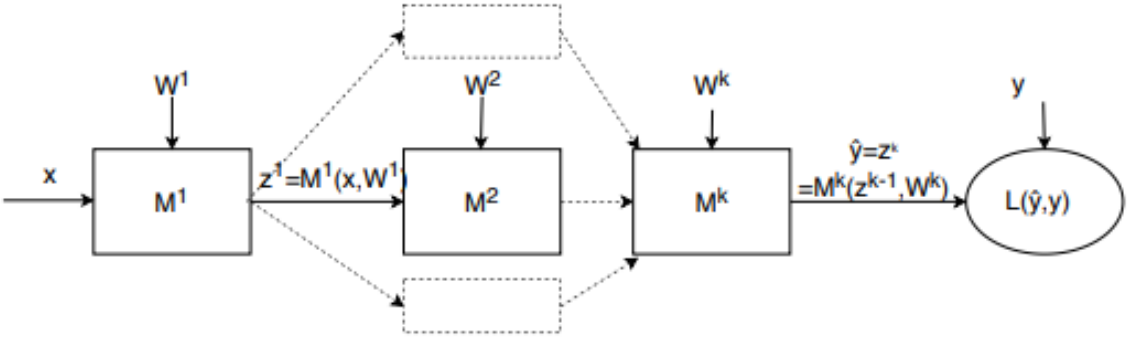


FIGURE 1 – Architecture module d'un réseau

Nous allons dans ce projet implémenter des réseaux de neurones de A à Z, avec différents modules et couches, qui nous permettront de mieux comprendre le processus de ces derniers. Nous allons commencer à travailler sur des cas simples, avec peu d'expérimentations afin d'assimiler la base des différentes notions, puis nous allons ensuite tester avec des données plus réelles, comme la base de données MNIST.

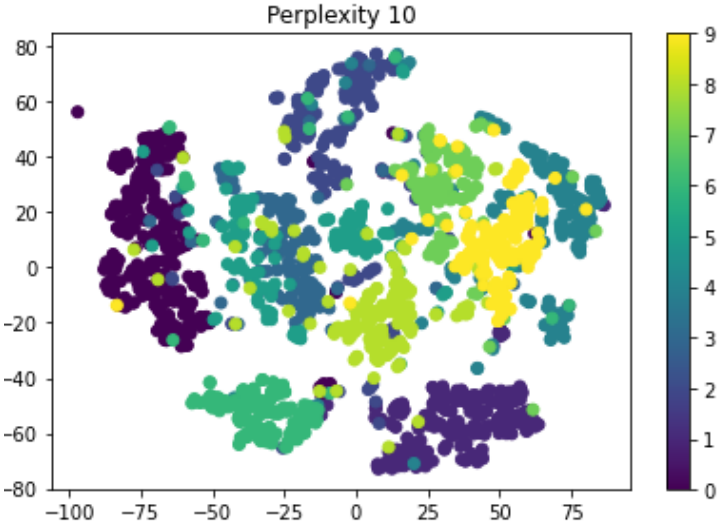


FIGURE 2 – Projection 2D du dataset MNIST avec t-SNE

2 Mon premier est . . . linéaire !

Pour notre premier réseau de neurones, nous avons réalisé une simple descente de gradient, où les données ont été générées de manière aléatoire avec un bruit sigma d'une fonction linéaire.

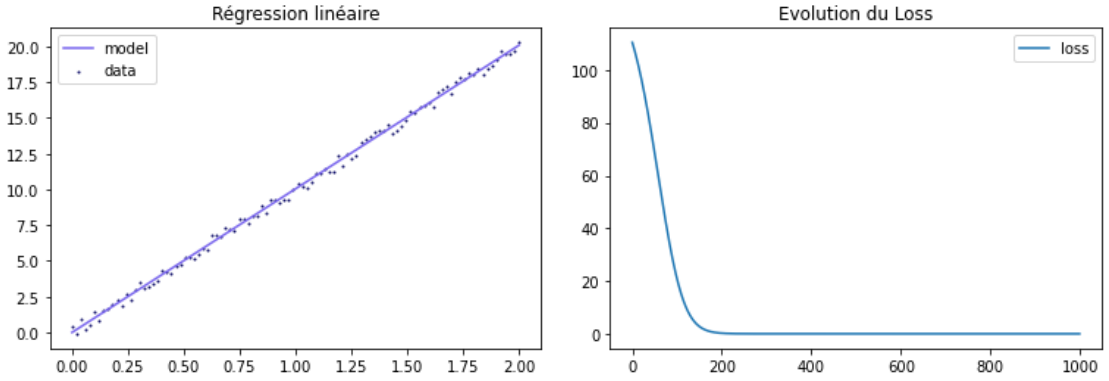


FIGURE 3 – Régression linéaire

Rien de spécial à signaler ici, nous avons utilisé le coût des moindres carrés qui marche très bien dans ce cas simple d'une régression linéaire.

3 Mon second et troisième sont . . . non-linéaires !

Pour cette partie, le but est de réussir à faire une classification des données qui ne sont pas linéairement séparables. Nous avons donc réalisé un réseau de neurones à deux couches linéaires avec comme fonctions d'activation, une tangente hyperbolique entre les deux couches et une sigmoide à la sortie, pour avoir un résultat dans la plage [0,1].
Des données de type XOR ont été générées afin de tester le réseau.

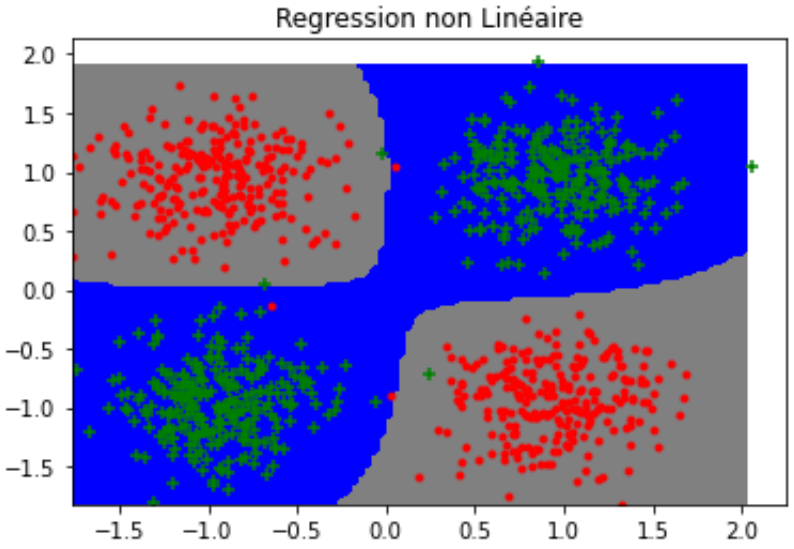


FIGURE 4 – Classification sur des données type XOR

Plusieurs tests ont été réalisés ici afin d'observer le comportement du réseau en fonction des hyper-paramètres, tels que le nombre de neurones du réseau, le pas d'apprentissage du gradient ou encore le nombre d'itérations. Nous avons pu remarquer que plus le nombre de neurones augmente et plus il nous faudra d'itérations pour que notre réseau apprenne correctement. Ceci peut être expliqué par le fait qu'il faut tout simplement plus d'itérations pour pouvoir apprendre tous les neurones du réseau. Néanmoins, le mode batch arrive tout de même à bien classifier malgré un faible nombre d'itérations, car comparé au mode stochastique, le mode batch permet d'apprendre sur l'ensemble des données en même temps. Une itération correspond à une époque en batch mais seulement 1/TailleDuBatch époque en stochastique.

Par la suite, nous avons automatisé le processus de chaînage entre les modules, les opérations de forward et backward, et les résultats obtenus par les deux modèles sont similaires, le processus a seulement été automatisé. Nous avons ainsi utilisé le mode automatisé pour faire nos tests avec le mode stochastique, et celui non automatisé avec le mode batch pour pouvoir ainsi les comparés.

Nous remarquons ici que le mode stochastique n'a pas bien réussi à classifier les données, il faudrait plus d'époques pour le mode stochastique, soit plus d'itérations.

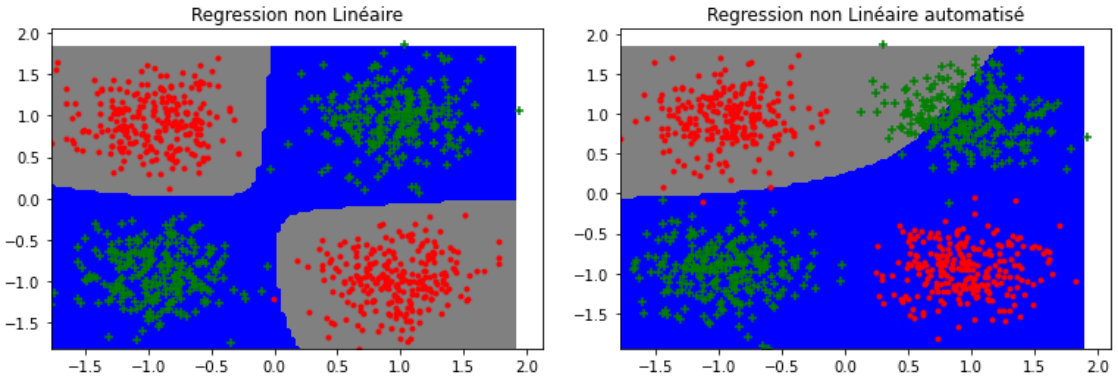


FIGURE 5 – 100 neurones & 100 itérations - batch (à gauche) et stochastique (à droite)

Nous avons ensuite augmenté le nombre de neurones, pour essayer de voir si le mode batch pouvait continuer a bien classifié malgré le faible nombre itérations. Les résultats nous montrent que le mode batch atteint aussi ses limites et n'arrive pas à apprendre sur tous les neurones présents dans le réseau.

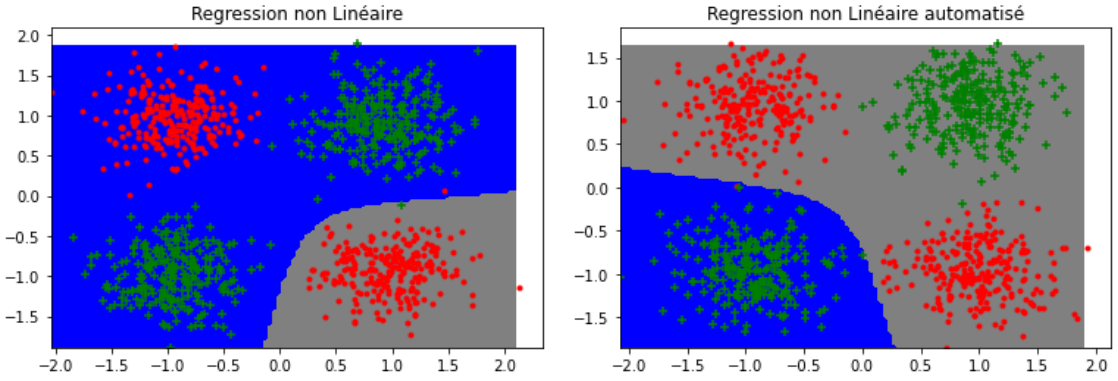


FIGURE 6 – 150 neurones & 100 itérations - batch (à gauche) et stochastique (à droite)

Enfin, avec plus d'itérations, le réseau arrive à bien classifier.

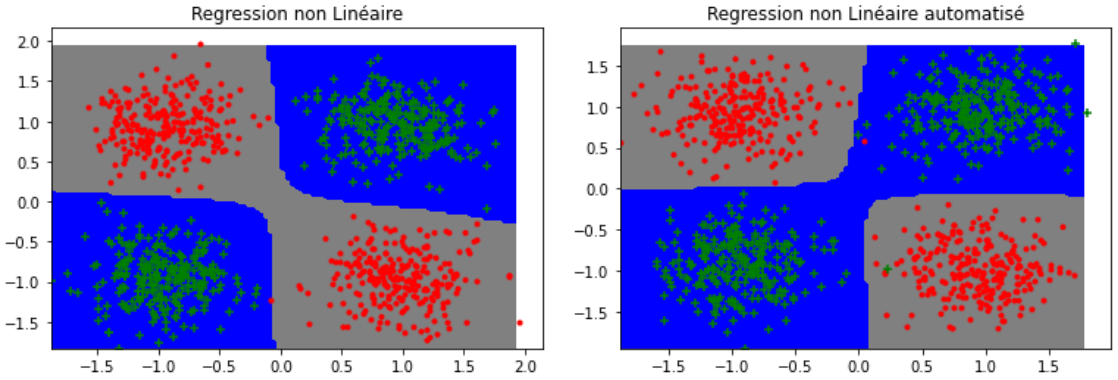


FIGURE 7 – 150 neurones & 1000 itérations - batch (à gauche) et stochastique (à droite)

En conclusion de cette partie, nous aurions aussi bien pu tester avec le mode mini-batch, mais c'est en réalité l'entre deux de ces deux méthodes, et la préférence entre ces trois méthodes dépend vraiment du type de la courbe d'erreur attendue. Le mode batch semble plus efficace dans notre cas car il nécessite moins d'itérations, mais il ne serait pas applicable sur des bases de données très grandes, car cela néccessiterait un immense espace de mémoire pour pouvoir stocker toute la matrice des données. C'est pourquoi le mode mini-batch est préféré dans la plupart des cas.

4 Mon quatrième est multi-classe

Dans cette partie, nous allons faire notre premier réseau multi-classe. Il nous ait dit que dans la théorie, nous devons utiliser un loss de type Softmax Cross-Entropy, car le Mean Squared Error a tendance à trop moyenner les résultats. Il est donc mieux d'utiliser des coûts adaptés aux distributions de probabilités comme la cross-entropie combinée avec le softmax afin d'éviter des instabilités numériques.
Cependant, nous n'observons pas de grandes différences sur les résultats obtenus avec les deux fonctions de coût.

Les résultats obtenus sont les suivants :

Loss	Neurone	Itération	Nombre de Classe	Score	Temps
MSE	100	100	10	0.87	8.09
CESM	100	100	10	0.87	8.78
MSE	50	100	10	0.86	4.67
CESM	50	100	10	0.88	5.01
MSE	150	100	10	0.88	11.44
CESM	150	100	10	0.88	11.93
MSE	100	50	10	0.88	3.96
CESM	100	50	10	0.85	4.23
MSE	100	200	10	0.87	15.98
CESM	100	200	10	0.88	17.19
MSE	100	100	5	0.94	4.69
CESM	100	100	5	0.93	4.87

Table 1 – Comparaison des résultats de CESM & de MSE sans bruit

Loss	Neurone	Itération	Nombre de Classe	Score	Temps
MSE	100	100	10	0.84	8.17
CESM	100	100	10	0.83	8.5

Table 2 – Comparaison des résultats de CESM & de MSE avec bruit

Nous observons que quels que soient les hyper-parametres, entre la MSE et la CESM, il n'y a pas de différences significatives. La faible différence entre les résultats de ces deux fonctions de coût est dûe au fait que la base de données utilisée est très facile à apprendre (données MNIST). Nous pourrons vérifier si ceci restera la même dans la suite de nos expériences où nous allons faire un auto-encodeur afin de voir s'il y a une réelle différence entre ces deux loss.

Néanmoins nous voulons testé différent fonction d'activation, on va par la suite comparer notre réseau avec une activation avec une fonction tangente hyperbolique et une fonction sigmoide.

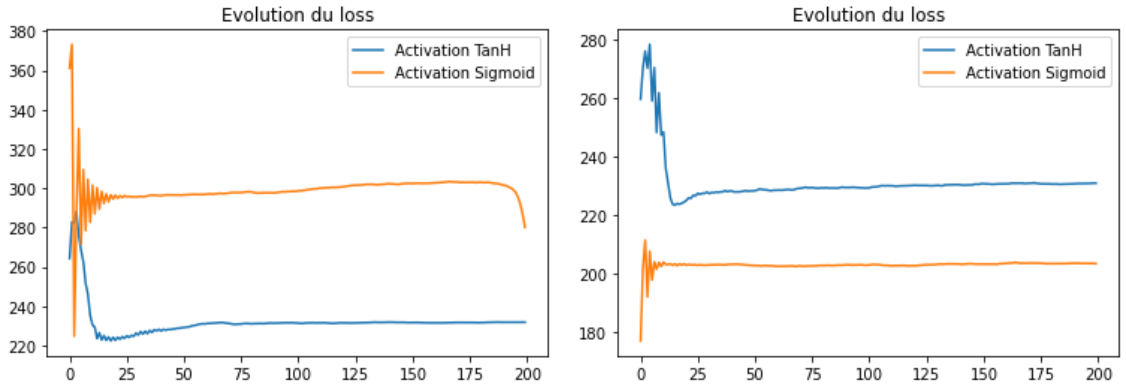


FIGURE 8 – 100 neurones - 100 itérations - grad. 0.001

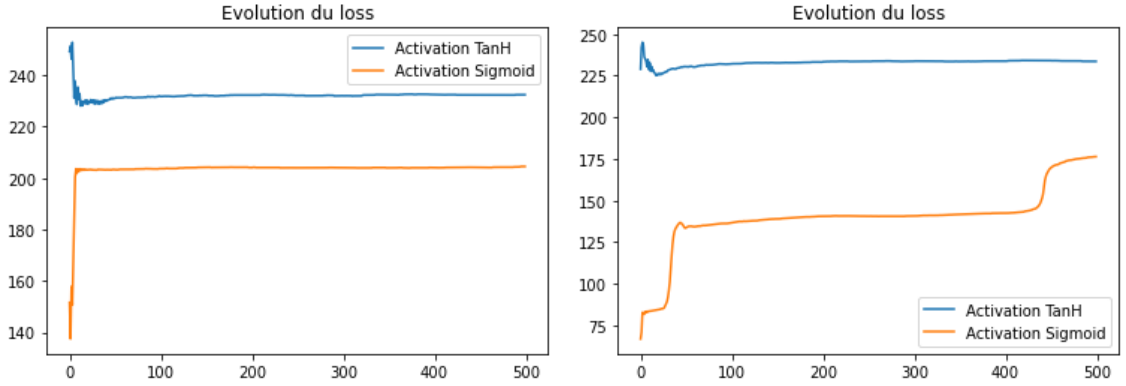


FIGURE 9 – 100 neurones - 500 itérations - grad. 0.001

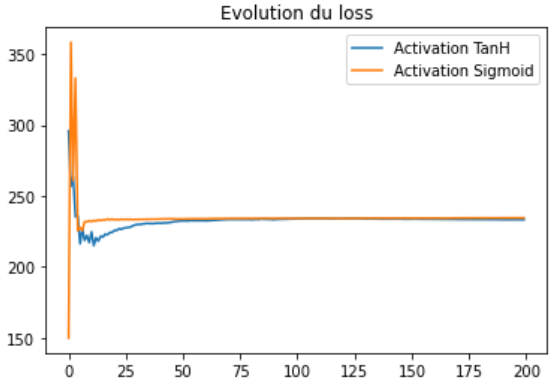


FIGURE 10 – 100 neurones - 200 itérations - grad. 0.001

En résumé, nous apercevons qu'avec une fonction d'activation Sigmoide, notre réseau à du mal à converger vers l'optimum, contrairement à la fonction tangente hyperbolique, ce qui est dû au **Vanishing Gradient Problem**.

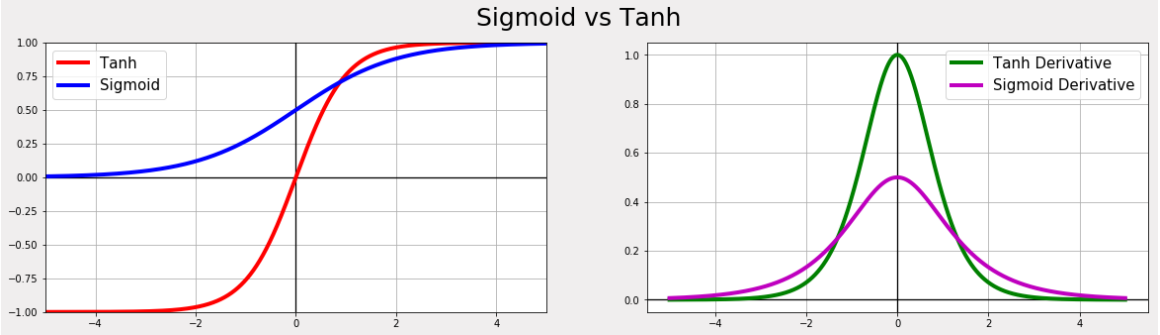


FIGURE 11 – Comparaison des deux fonctions d'activation et de leur dérivée

La fonction tangente hyperbolique est plus souvent préféré à la sigmoïde car la sigmoïde a tendance à se rapprocher du 0, ce qui rend le réseau difficile à apprendre. Avec les courbes précédemment affichées, nous observons qu'avec la sigmoïde, le réseau a du mal à atteindre l'optimum. Ceci est traduit par le fait que la dérivée du tangente hyperbolique est plus *expressive*. Toute fois, si par chance la sigmoïde arrive à se rapprocher assez de l'optimum, alors la convergence est un peu plus rapide qu'avec la fonction tangente hyperbolique.
Donc la meilleure solution ici serait d'utiliser des fonctions d'activation de type TanH et en sortie une Sigmoïde pour avoir une sortie compris entre [0 ;1], qui sera plus pratique à manipuler que TanH qui est lui compris entre [-1 ;1].

5 Mon cinquième se compresse

Nous allons maintenant réaliser un auto-encodeur, qui est un réseau de neurones dont l'objectif est d'apprendre un encodage des données dans le but généralement de réduire les dimensions. Il s'agit d'apprentissage non-supervisé.

L'architecture de notre réseau sera :

— Encodage : Linear(256,100) → TanH() → Linear(100,10) → TanH()
— Décodage : Linear(10,100) → TanH() → Linear(100,256) → Sigmoide()

Par la suite, plusieurs expérimentations sur différents hyper-paramètres seront faites, tels que le nombre de neurones ou les différents fonctions de coût utilisés.

5.1 Images reconstruites après une forte compression

Nous pouvons voir que la reconstruction après une forte compression avec un loss de type Cross Entropy - Softmax, nous donne des résultats très moyens. Nous observons que la CE-Softmax a tendance à être très expressive.

Première ligne : données initiales
Deuxième ligne : données reconstruites

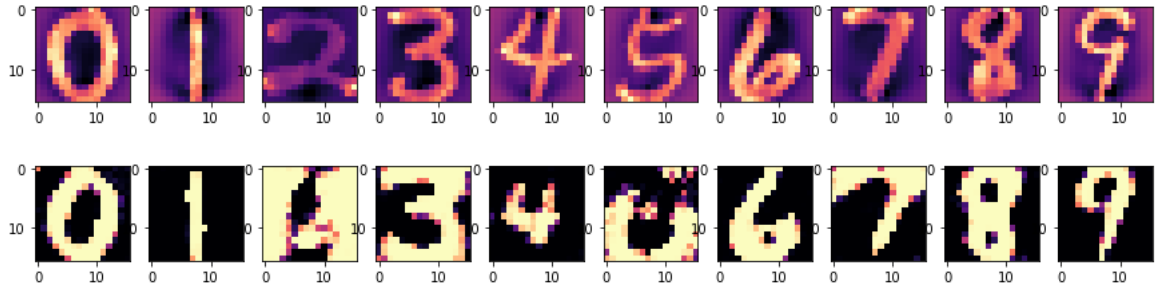


FIGURE 12 – CE-Softmax - 100 neurones - 500 itérations

A l'inverse du CE-Softmax, la fonction de coût MSE a tendance à trop moyenner, ce qui implique une perte d'information à la fin dans le rendu.

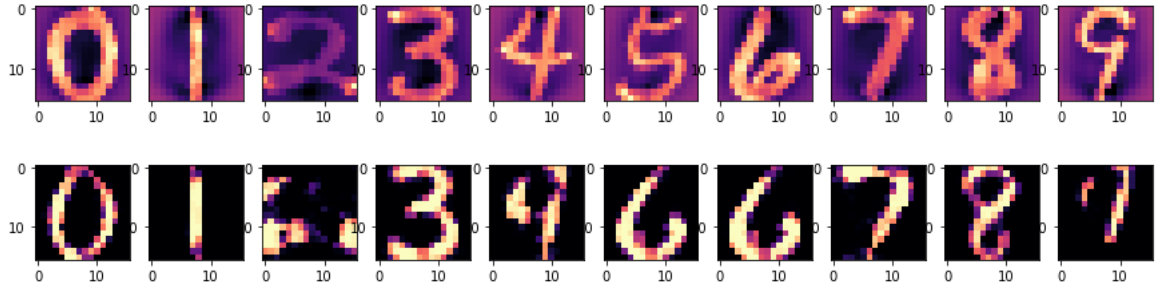


FIGURE 13 – MSE - 100 neurones - 500 itérations

Enfin, nous observons dans notre dernier test avec la Binary Cross-Entropy, un entre deux des résultats précédents.

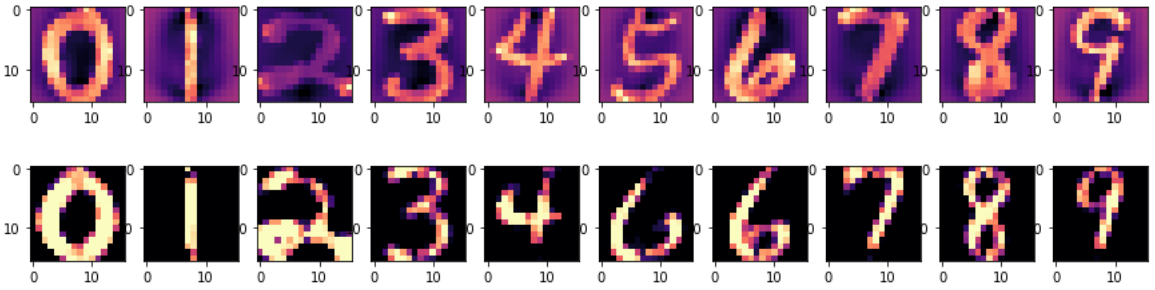


FIGURE 14 – BCE - 100 neurones - 500 itérations

Nous avons continué à faire des tests, en modifiant par exemple, le nombre de neurones, d'itérations et le pas d'apprentissage du gradient. Nous observons que avec un nombre de neurones dépassant les 150, cela n'apporte plus autant d'améliorations, et que la convergence se fait aux alentours de 500 itérations. Avec environ 100 neurones et un pas de gradient à 10^{-4}, un nombre trop faible de neurones fait que les chiffres manuscrits sont difficiles à différencier entre eux. Nous pouvons noter que les chiffres 4 se confondent avec les chiffres 9, et les chiffres 5 avec les 6. Pour vérifier cette hypothèse, Nous allons faire une projection en 2D de nos données grâce à l'algorithme t-sne pour essayer de trouver les classes qui sont plus ou moins proches. Il faut aussi noter que nous avons testé avec du bruit sur les données initiales mais le résultat ne change pas énormément.

5.2 Projection en 2D de nos données

Nous avons utilisé la t-sne, pour réduire la dimension de nos données afin d'obtenir une projection en 2D et de pouvoir ensuite les visualiser.

Les figures ci-dessous représentent nos données de tests après t-SNE avec des perplexités différentes. Les résultats concordent avec nos anciennes expérimentations, où les chiffres 9 sont vraiment très proches des chiffres 4 sur la projection, ou encore avec les chiffres 5 et 6 qui sont presque mélangés entre eux. Ce qui se traduit par la forte difficulté à reconstruire les chiffres 5 après une forte compression.

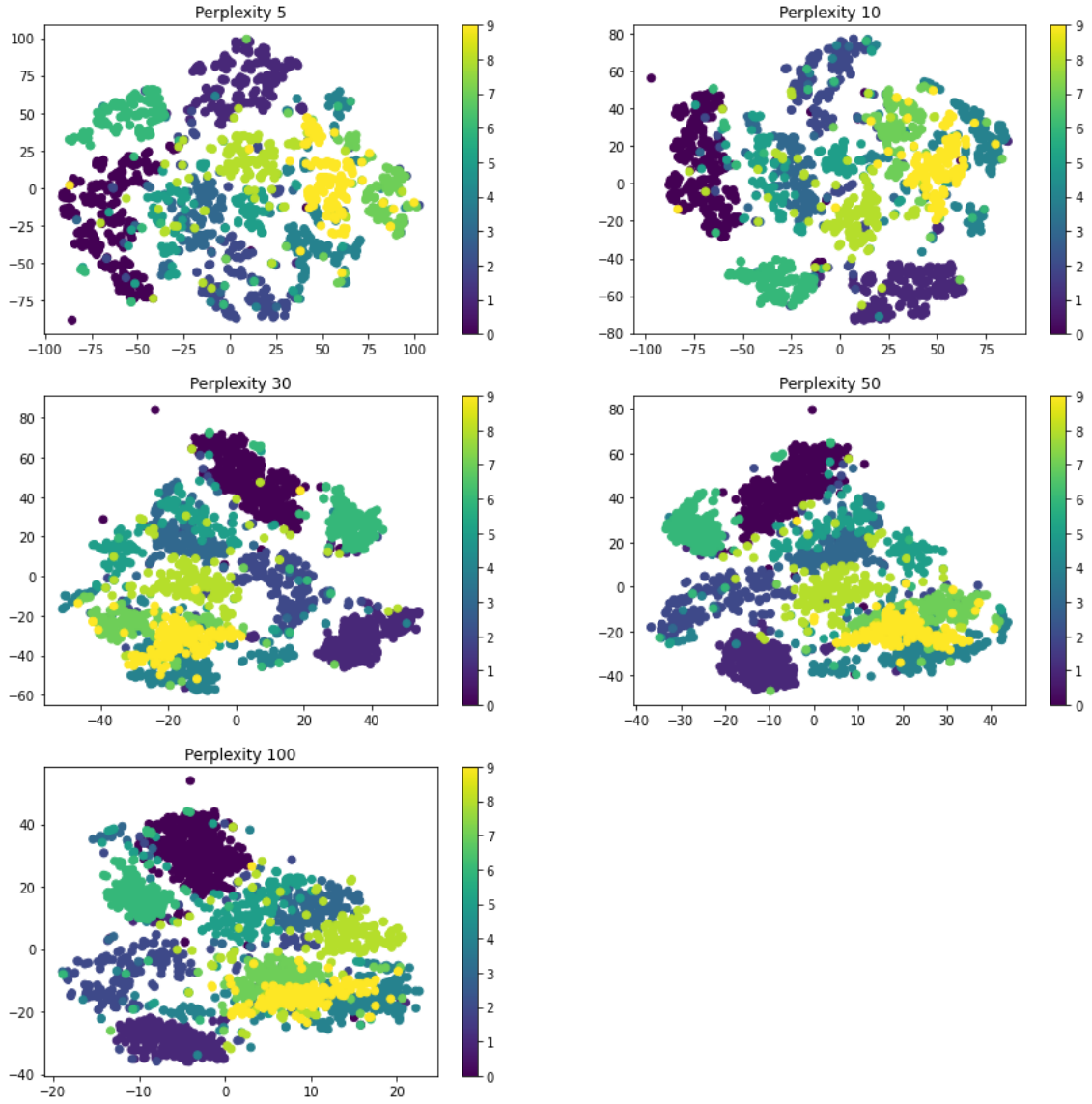


FIGURE 15 – t-SNE, sur les données initales

Nous pouvons voir ici qu'après l'encodage puis le décodage, nos données ont été fortement impactées et que la distinction entre les groupes ne sont plus aussi évidente que sur les données initales, ce qui est vraisembablement dû à la perte de données après la forte compression.

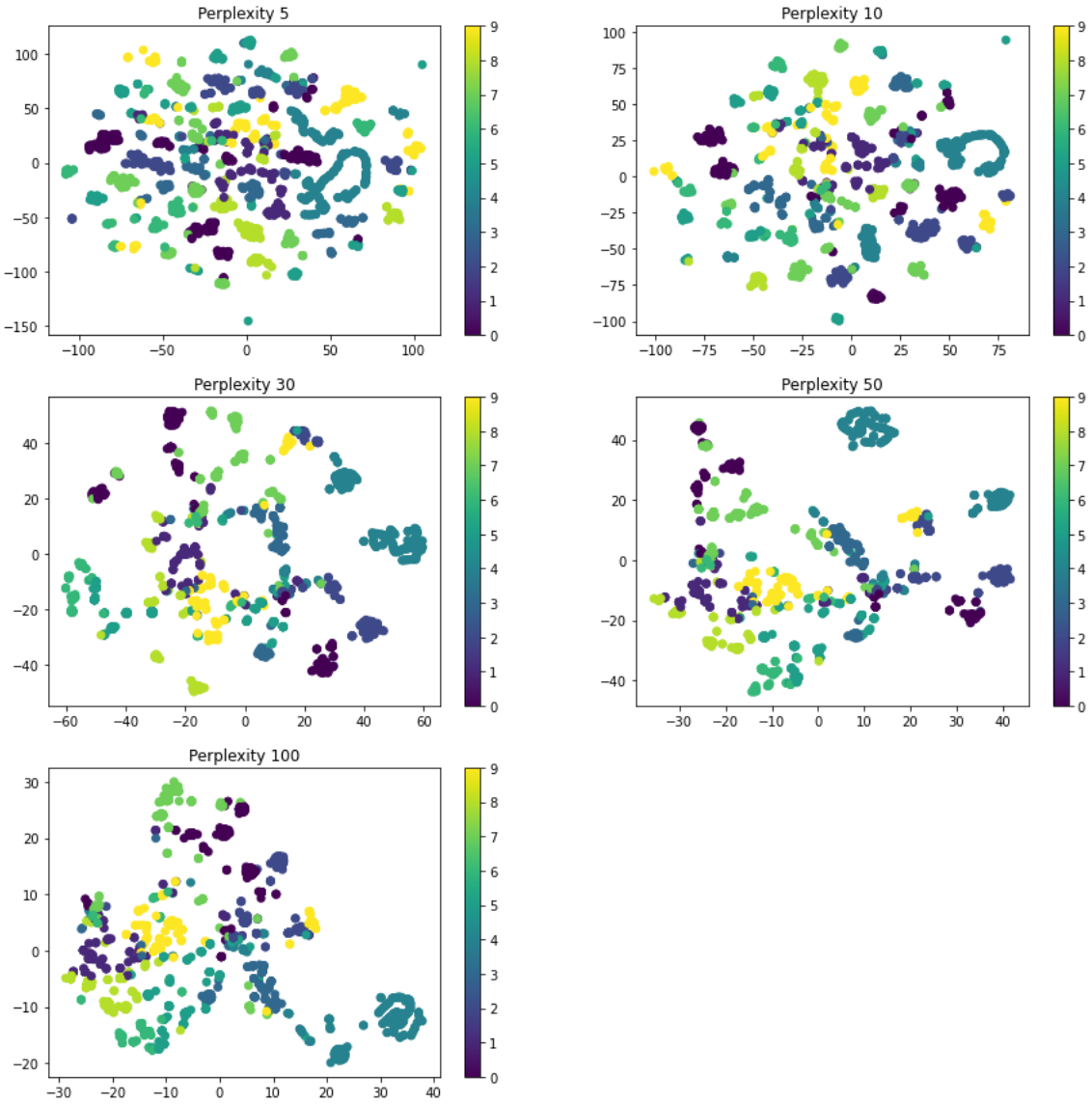


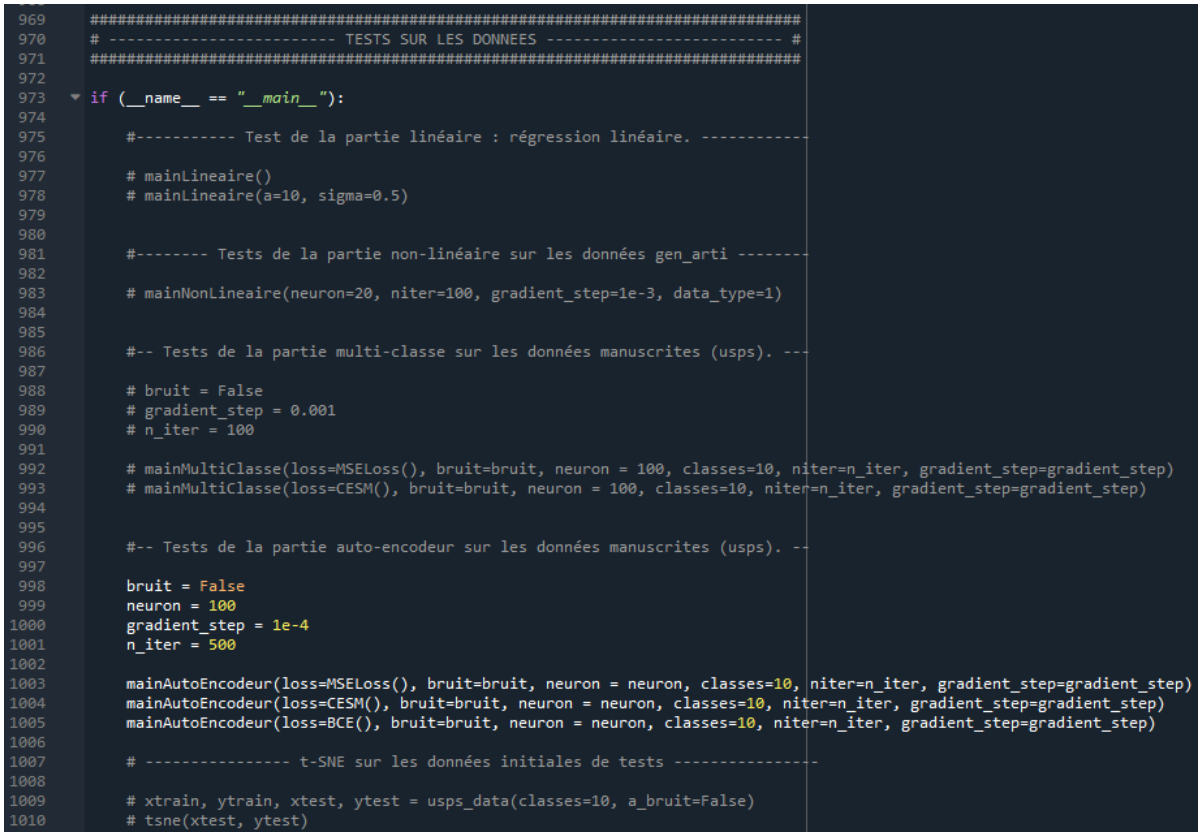
FIGURE 16 – t-SNE, sur les données reconstruites

6 Conclusion

En conclusion, nous avons beaucoup appris les détails du fonctionnement des réseaux de neurones et des multiples solutions possibles pour tester notre algorithme d'apprentissage. Dans notre cas, dans la partie auto-encodeur, la fonction de coût Binary Cross-Entropy est un très bon compromis entre la Mean Squared Error et la Softmax Cross-Entropy. Néanmoins, nous avons tout de même des difficultés pour la classification de certaines classes très similaires, comme les chiffres 5 et 6. Il aurait fallu faire plus de tests avec des réseaux différents, mais dû à une limitation de nos machines personnelles et une incapacité d'accéder aux machines proposées par l'université à cause du confinnement, nous n'avons pas pu faire des tests plus approfondis. Chaque apprentissage prenait un temps assez conséquent, malgré un certain niveau d'optimisation du code et l'utilisation des opérations matricielles.

7 Manuel d'utilisation du code

Vous trouverez avec ce rapport un fichier python nommé **ml.py**, qui contient le code avec tous les tests et observations. Vous trouverez à la fin du fichier, différents parties pré-écrites qui nécessite juste de décommenter le bout de code que l'on veut tester.



```
969  ##########################################################################
970  # ------------------------ TESTS SUR LES DONNEES ------------------------ #
971  ##########################################################################
972
973  if (__name__ == "__main__"):
974
975      #---------- Test de la partie linéaire : régression linéaire. -----------
976
977      # mainLineaire()
978      # mainLineaire(a=10, sigma=0.5)
979
980
981      #-------- Tests de la partie non-linéaire sur les données gen_arti --------
982
983      # mainNonLineaire(neuron=20, niter=100, gradient_step=1e-3, data_type=1)
984
985
986      #-- Tests de la partie multi-classe sur les données manuscrites (usps). ---
987
988      # bruit = False
989      # gradient_step = 0.001
990      # n_iter = 100
991
992      # mainMultiClasse(loss=MSELoss(), bruit=bruit, neuron = 100, classes=10, niter=n_iter, gradient_step=gradient_step)
993      # mainMultiClasse(loss=CESM(), bruit=bruit, neuron = 100, classes=10, niter=n_iter, gradient_step=gradient_step)
994
995
996      #-- Tests de la partie auto-encodeur sur les données manuscrites (usps). --
997
998      bruit = False
999      neuron = 100
1000     gradient_step = 1e-4
1001     n_iter = 500
1002
1003     mainAutoEncodeur(loss=MSELoss(), bruit=bruit, neuron = neuron, classes=10, niter=n_iter, gradient_step=gradient_step)
1004     mainAutoEncodeur(loss=CESM(), bruit=bruit, neuron = neuron, classes=10, niter=n_iter, gradient_step=gradient_step)
1005     mainAutoEncodeur(loss=BCE(), bruit=bruit, neuron = neuron, classes=10, niter=n_iter, gradient_step=gradient_step)
1006
1007     # ---------------- t-SNE sur les données initiales de tests ----------------
1008
1009     # xtrain, ytrain, xtest, ytest = usps_data(classes=10, a_bruit=False)
1010     # tsne(xtest, ytest)
```

FIGURE 17 – Code source dans ml.py