

# Reseau de neurones

Projet de ML

Cherchour Lièce  
Thauvin Dao



Faculty of Science and Engineering  
Sorbonne université

# Contents

<b>1</b>	<b>Travail effectué</b>	<b>2</b>
<b>2</b>	<b>Réseau linéaire</b>	<b>2</b>
<b>3</b>	<b>Réseau non-linéaire</b>	<b>3</b>
3.1	Résultat pour 2 neurones avec un batch_size = len(datax)	3
3.2	Résultat pour 3 neurones avec un batch_size = len(datax)	3
3.3	Résultat pour 5 neurones avec un batch_size = len(datax)	4
3.4	Résultat pour 3 neurones avec un batch_size = 100	4
3.5	Observations	4
<b>4</b>	<b>Mise en place d'une pipeline</b>	<b>4</b>
<b>5</b>	<b>Mise en place du multi-classe</b>	<b>4</b>
<b>6</b>	<b>Auto-encodeur</b>	<b>5</b>
6.1	Visualisation	5
6.1.1	Compression de taille 10	5
6.1.2	Compression de taille 3	6
6.1.3	Conclusion	7
6.2	T-SNE et 2 dimensions	7
6.2.1	Observations	7
6.2.2	Conclusion	7
6.3	Clustering	8
<b>7</b>	<b>Réseau convolutif</b>	<b>9</b>

## 1 Travail effectué

Nous avons implementé la partie obligatoire du sujet et les différents modules basique nécessaire à un réseau convolutifs. Nous avons de plus défini une fonction d'activation LeakyReLU, rajouté la possibilité d'avoir un biais (ou non) pour le module linéaire. Nous avons effectué plusieurs types de tests pour être sur que nos différents modules marche bien. Pour les modules linéaire et convolutif nous avons créé des fichiers permettant de vérifier les différentes dimensions à plusieurs étapes. Pour le reste (cela prend en compte les fonctions d'activations), nous avons crée des réseaux de neurones pour observer les résultats que l'on obtenait en utilisant nos différents modules.

## 2 Réseau linéaire

Pour tester notre réseau linéaire, nous avons décider de générer des données aléatoirement à 2 attributs et utilisé une fonction linéaire bruité (sous la forme  $a * x_1 + b * x_2 + c$ ) pour calculer les  $y$  de nos données. Le but de notre réseau est donc de retrouver la fonction linéaire associé, il s'agit donc d'un problème de régression.

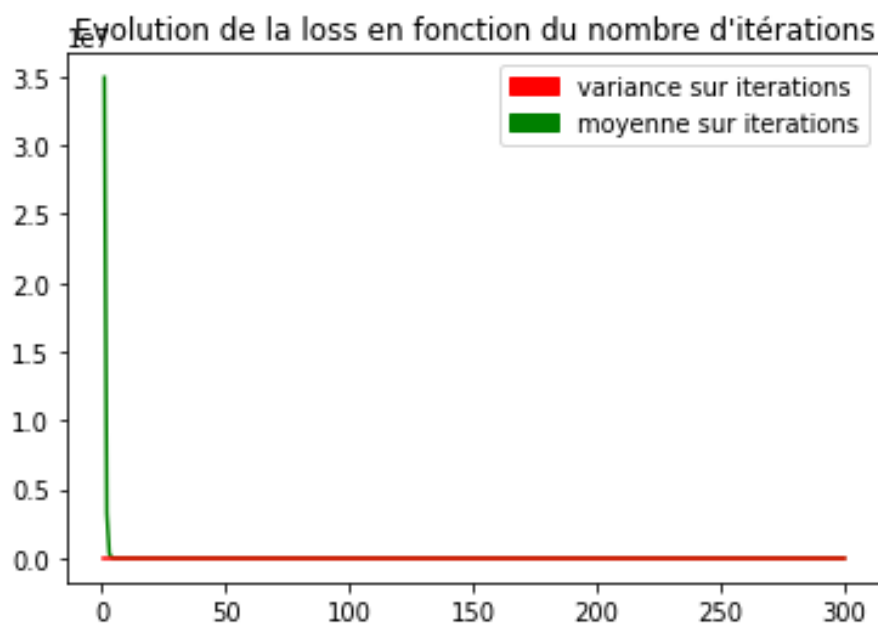
Nous utilisons seulement un seul neurone pour notre réseau prenant  $x_1$  et  $x_2$  en entrée et renvoyant  $y$ . En premier, nous utilisons 100 données en entrée avec un gradient step de  $1e - 4$  et 300 itérations.

Les résultats obtenus sont :

	$a$	$b$	$c$
originaux	1002	13	4
obtenus	1002.01	13.00	3.85

Les différentes valeurs sont obtenues en récupérant les poids de notre neurone, on observe bien une correspondance avec les poids de notre fonction linéaire ce qui montre bien le fonctionnement de notre réseau.

Pour en être sur, regardons l'évolution de la loss au cours des itérations:



On observe que la loss diminue très peu après quelques itérations, essayons de faire seulement 50 itérations.

Résultats obtenus :

	$a$	$b$	$c$
originaux	1002	13	4
obtenus	1001.81	13.89	-4.74

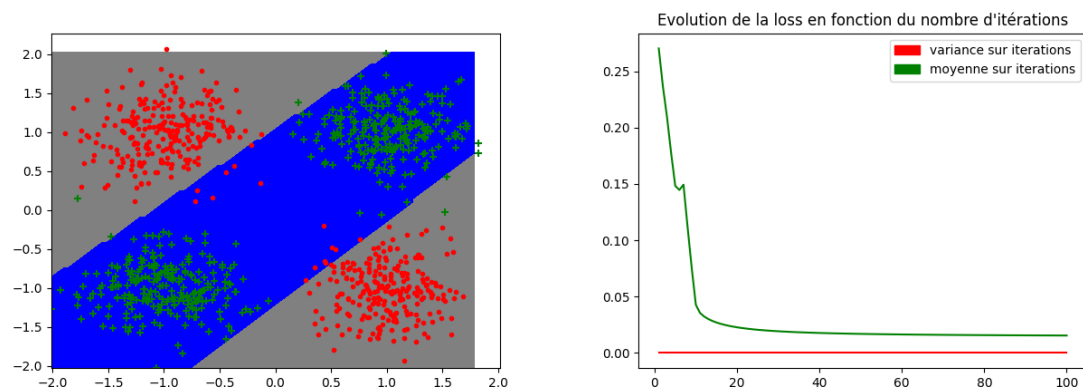
Les résultats obtenues sont plutôt bon mais moins bon que lors de notre dernière itération surtout au niveau du biais.

### 3 Réseau non-linéaire

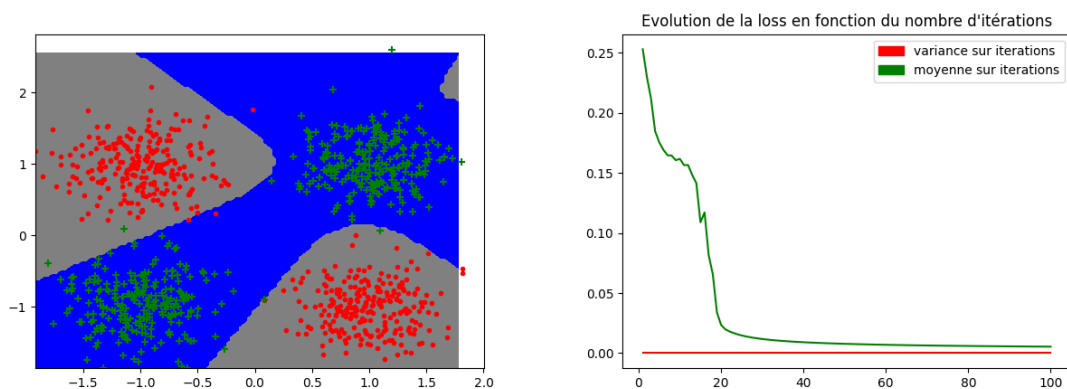
Nous avons implémenté notre réseau linéaire et avons utilisé les données de `gen_arti`, `data_type=1` défini pour les TPs précédent. Le but de notre réseau est de faire une classification des données `gen_arti`, `data_type=1`, par la suite on va afficher la frontière de décision obtenues en 2D. Nous utilisons entre 2 et 5 neurones pour la partie cachée de notre modèle durant les tests, et nous comparons les solutions obtenues. En premier temps, nous avons un réseau avec 2 entrées ( $x_1$  et  $x_2$  d'une donnée) une partie cachée avec 3 neurones et notre sortie qui n'en possède qu'un seul.

Nous allons donc faire varier le nombre de neurones et la tailles des batchs pour voir les différentes frontière de décision et évolution de la fonction de coût associé

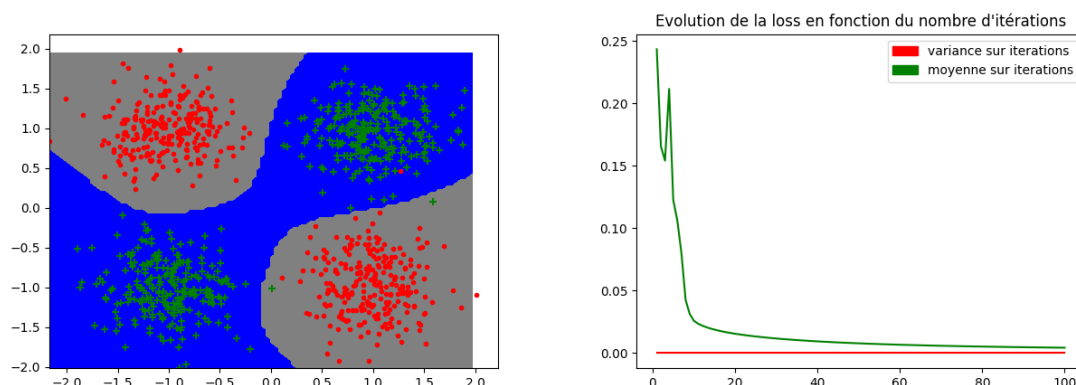
#### 3.1 Résultat pour 2 neurones avec un `batch_size = len(datax)`



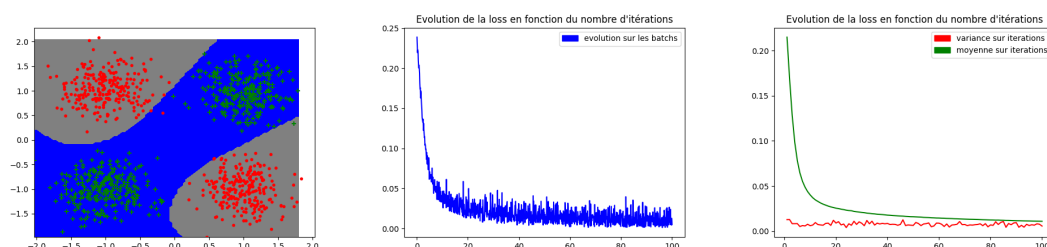
#### 3.2 Résultat pour 3 neurones avec un `batch_size = len(datax)`



### 3.3 Résultat pour 5 neurones avec un `batch_size = len(datax)`



### 3.4 Résultat pour 3 neurones avec un `batch_size = 100`



### 3.5 Observations

On peut observer que plus on ajoute de neurones cachées plus la frontière de décision obtenues est complexe, c'est assez logique car elle ne vient pas d'une combinaisons linéaire, grâce à nos fonctions d'activations entres les différents modules. On observe également que pour 3 neurones, en changeant seulement la taille du batch, on obtient une frontière de décision légèrement plus lisse, mais une convergence bien plus rapide (environ 2 fois moins d'itérations).

## 4 Mise en place d'une pipeline

Notre pipeline permettant d'emboîter différents module indépendants entre eux est complètement opérationnelle et correspond à celui de l'énoncé. Nous avons seulement ajouté un paramètre verbose à la fonction SGD, une verbose de 1 permet l'affichage de l'évolution du cout durant l'exécution et une verbose de 2 permet en plus d'ajouter l'affichage d'une courbe correspondante en fin d'itérations (avec une autre courbe qui fait de même mais en affichant les couts des différents batch au lieu d'en faire une moyenne).

## 5 Mise en place du multi-classe

Pour le multi-classe, nous avons implémenté un réseau de la forme :

`linear(input_size,hidden_size) -> Sigmoid() -> linear(hidden_size,10) -> Softmax`

En utilisant un coût d'entropie croisée, une `batch_size` de 100 et un pas de gradient de  $1e-3$ . Nous avons fait varier la `hidden_size` du réseau et nous avons pu observé différentes choses.

Le but de ce réseau est de classé les nombres de la base USPS, bien sur nous faisons une séparation test/train.

Le nombre de neurones intermédiaire est un paramètres assez critique, effectivement entre choisir 128

neurones cachées ou 10, on obtient un temps d'exécution assez différents et des résultats meilleur plus le nombre de neurones est important. Mais on remarque également qu'il y à un moment où cela ne sert plus à rien d'ajouter des neurones intermédiaire.

Effectivement si l'on passe de 5 à 10 neurones cachées on observe une amélioration sur la précision de 4%, mais si l'on passe de 10 à 100 neurones, on observe une amélioration de à peine 1% et on pourrait se demander si le temps de calcul supplémentaire vau vraiment le coup dans ce cas la.

Nous avons aussi testé une taille de couche cachée absurde de 1000 mais notre précision diminue assez peu (1%), sûrement dû à la simplicité du problème.

Nos résultats :

Taille couche cachée	Précision
1000	0.90
128	0.91
100	0.90
50	0.89
10	0.88
5	0.84

## 6 Auto-encodeur

Ici nous réalisons les tests avec 100 itérations, un batch size de 50, un gradient step de  $1e-3$  des couches cachés de taille 100 et une compression de taille 10 quand cela n'est pas précisé.

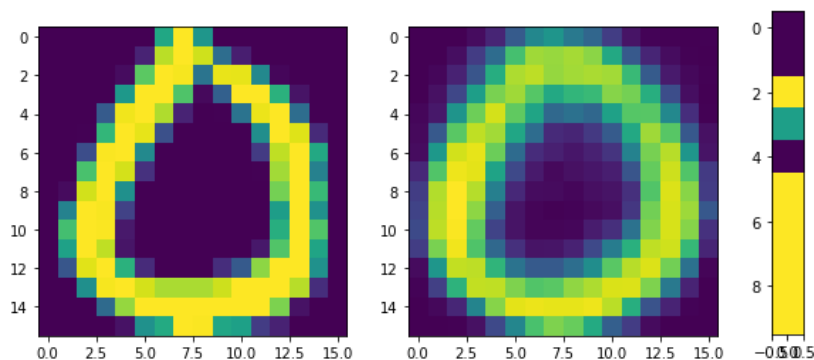
Nous avons testé en général nos données sur deux jeux de données : USPS et MNIST. Mais nous avons décidé de présenter nos résultats sur le jeu de données USPS, le jeu de données étant plus petit, il est possible que les résultats soient meilleurs avec MNIST mais les calculs seraient plus lourds.

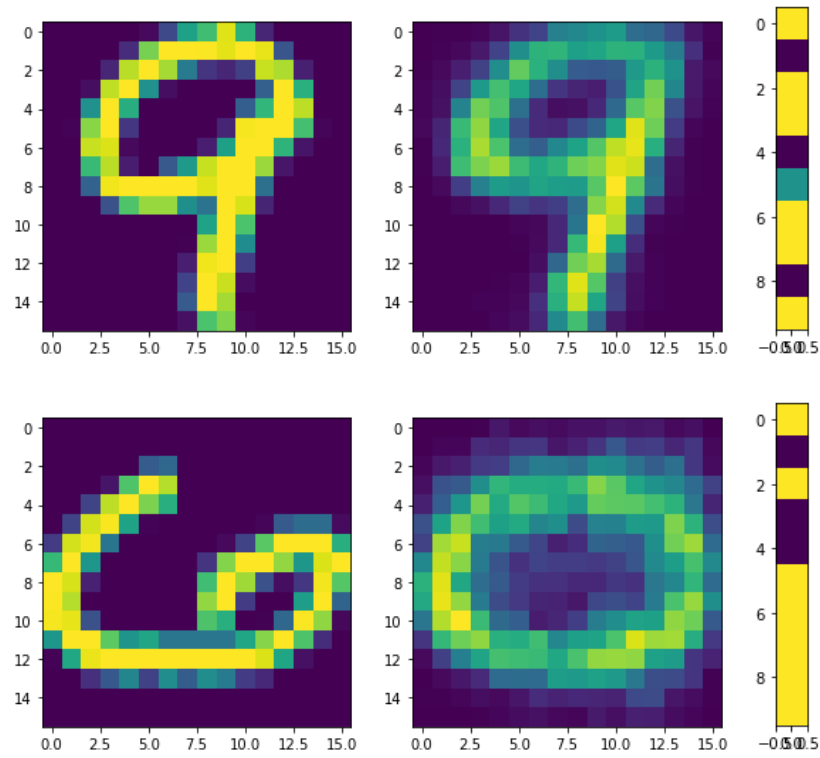
Bien sur nous faisons une séparation test/train pour les tests comme dans les parties précédentes.

### 6.1 Visualisation

Ici nous visualiserons les représentations obtenues après compréhension (en dernier) et après reconstruction (en second) en fonction du facteur de compression.

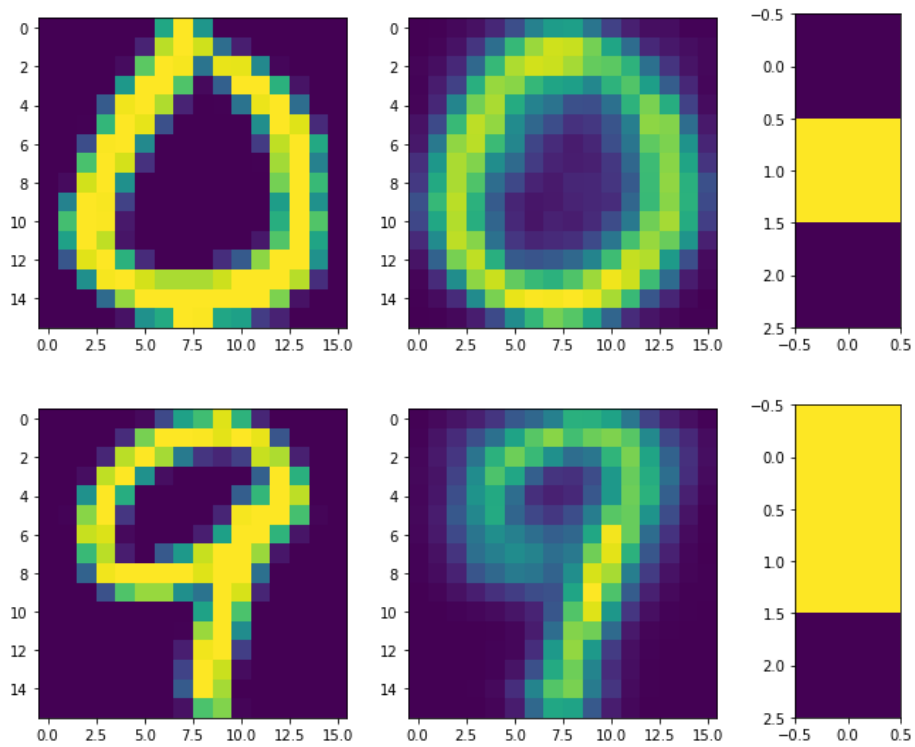
#### 6.1.1 Compression de taille 10

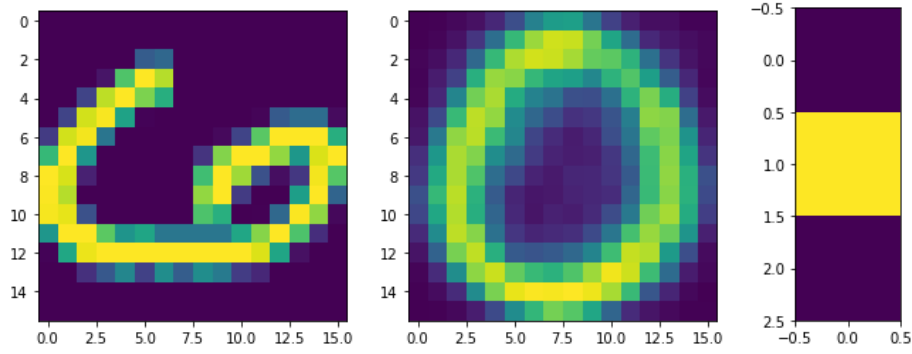




Ici on affiche un 0, un 9 et un 6. Il est déjà facile de voir que le 6 n'est pas reconstruit correctement, ça reconstruction correspond plus à un 0. On observe d'ailleurs que notre compression du 6 est très proche de la compression du 0 mais est plus éloigné du 9, ce qui correspond à notre observation.

### 6.1.2 Compression de taille 3





On observe le même comportement que dans la compression de taille 10 mais cette fois-ci la reconstruction et la compression de notre 6 et 0 sont complètement identiques, alors qu'on observait une différence majeure dans les reconstructions (et les compressions) précédentes.

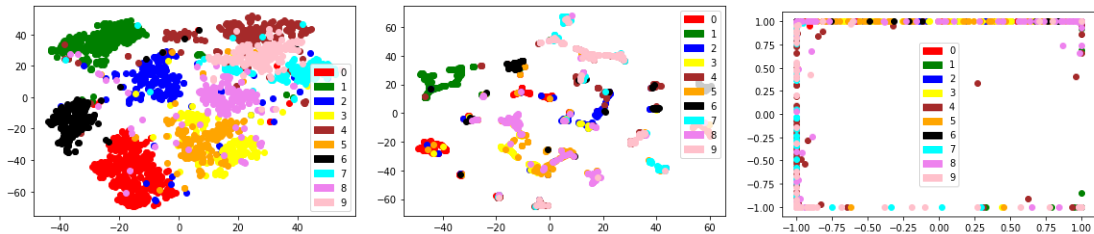
### 6.1.3 Conclusion

On observe ici que plus notre facteur de compression est petit (donc la compression est forte), plus notre reconstruction est générale, ce qui est induit par l'information gardé par la compression, dans le cas de nos 6 et 0, leur différentiation est perdu pendant la compression de taille 3 mais pas pour la taille 10. De plus on observe bien une compression proche quand les nombres se ressemblent (même si le 6 et le 0 ne sont pas les mêmes nombres, il se ressemble assez ici), ce qui peut permettre du clustering en réduisant fortement les dimensions, on en parlera dans une partie suivante.

## 6.2 T-SNE et 2 dimensions

Ici on va comparer le résultat obtenue en réalisant une compression à 2 valeurs et le T-SNE obtenue sur les mêmes données.

### 6.2.1 Observations



Ici, nous affichons 3 schémas, le premier est obtenue par T-SNE sur les données de test de base, le second est obtenue par T-SNE sur les données compressé en taille 10 et le dernier est obtenue par compression de taille 2.

On observe dans un premier temps que T-SNE arrive très bien à représenter les données en 2D, on observe facilement des nuages de couleurs différentes.

C'est toujours le cas dans notre deuxième schéma mais les nuages sont beaucoup plus concentrés mais cette fois-ci dans différentes zones pour une même classe, sûrement du à la compression de notre auto-encodeur.

Dans notre dernier schéma, les données semblent complètement jeté au hasard.

### 6.2.2 Conclusion

Les résultats obtenues après compression sont peu convainquant pour représenter les classes des données, l'auto-encodeur n'étant pas fait pour de la réduction de dimension à la différence de T-SNE.

De plus l'utilisation d'un auto-encodeur en pré-traitement pour la réduction de données ne permet pas d'améliorer les performances du T-SNE.



### 6.3 Clustering

Nous avons effectué un K-Means avec 10 points de départ pour arriver à trouver des clusters pour chaque nombre.

Pour tester nos clusters, nous avons utilisé la formule suivante pour calculer un score de pureté pour chaque cluster,

$$\text{Purity} = \frac{1}{N} \sum_{i=1}^k \max_j |c_i \cap t_j| \quad (1)$$

avec  $t_j$  la classe majoritaire d'un cluster.

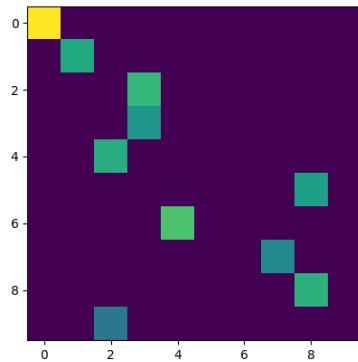
Nous avons également calculé un randscore avec la fonction [adjusted rand score de sklearn](#). Voici les résultats obtenues sur le dataset USPS après compression par l'auto-encodeur :

Calcul du score de pureté sur les données compressés	
Clusters	Score
0	0.55
1	0.93
2	0.59
3	0.59
4	0.44
5	0.61
6	0.46
7	0.79
8	0.59
9	0.59
Pureté moyenne = 0.62	
Adjusted Rand score (SKlearn)	0.44

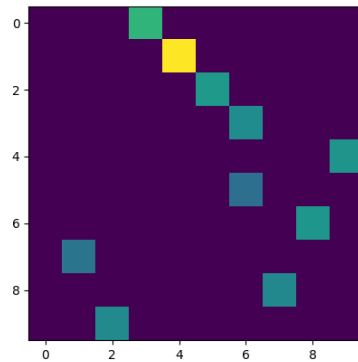
Si on essaye d'effectuer un clustering sur les données non-compressées, on observe les résultats suivants :

Calcul du score de pureté sur les données non-compressées	
Clusters	Score
0	0.46
1	0.96
2	0.70
3	0.75
4	0.66
5	0.58
6	0.78
7	0.67
8	0.72
9	0.68
Pureté moyenne = 0.69	
Adjusted Rand score (SKlearn)	0.49

Voici les classes majoritaires de ces 2 différents clustering :



(a) Argmax sur les données compressées



(b) Argmax sur les données originales

On observe bien des classes majoritaires différentes pour chaque cluster, mais 2 clusters ont la même classe majoritaire dans nos clusters après compression, ce qui indique sûrement une perte d'information (ou une mauvaise initialisation mais cela ne semble pas être le cas car arrive souvent).

Une pureté moyenne de 0.62 est nettement supérieure à une pureté issue de l'aléatoire, le rand score obtenue nous permet de confirmer cette idée, en effet le rand score est égal à 0 quand le clustering obtenues est complètement aléatoire et 1 si les clusters sont identiques à ceux attendus. On observe une perte faible de précision entre les données dans leur dimension d'origine et les données compressées, mais par contre il est beaucoup moins coûteux à garder en mémoire.

## 7 Réseau convolutif

Nous avons implémenté le réseau décrit dans le projet, nous avons effectué des batch de taille 50 avec un pas de gradient de 1e-3 pour éviter de diverger. Les résultats obtenues sont les suivants :

Comparaison précision convolutif/linéaire		
nb iterations	precision conv	precision lineaire
10	0.884	0.856
100	0.897	0.901
1000	0.91	0.923

Le réseau comparé ici est le réseau présenté dans la partie Mise en place du multi-classe, nous utilisons une couche cachée de taille 128 ici.

La précision de notre réseau convolutionnel semble un peu moins bon que notre réseau multi-classe. Pour nous, 2 raisons sont possibles, notre réseau convolutionnel contient beaucoup de paramètres (407 496) par rapport à l'autre réseau (34 048), ce qui ralentit fortement la convergence et donc nécessite beaucoup plus de données ou d'itérations. L'autre possibilité est l'absence de convolution 2D ou vertical.