UNIVERSITÉ DU QUÉBEC

MÉMOIRE PRÉSENTÉ À
L'UNIVERSITÉ DU QUÉBEC À TROIS-RIVIERES

COMME EXIGENCE PARTIELLE
DE LA MAÎTRISE EN MATHEMATIQUES ET INFORMATIQUE APPLIQUEES

PAR
JEAN NOEL DIBOCOR DIOUF

CLASSIFICATION, APPRENTISSAGE PROFOND ET RESEAUX DE NEURONES : APPLICATION EN SCIENCE DES DONNEES

NOVEMBRE 2020

Université du Québec à Trois-Rivières

Service de la bibliothèque

Avertissement

L'auteur de ce mémoire ou de cette thèse a autorisé l'Université du Québec à Trois-Rivières à diffuser, à des fins non lucratives, une copie de son mémoire ou de sa thèse.

Cette diffusion n'entraîne pas une renonciation de la part de l'auteur à ses droits de propriété intellectuelle, incluant le droit d'auteur, sur ce mémoire ou cette thèse. Notamment, la reproduction ou la publication de la totalité ou d'une partie importante de ce mémoire ou de cette thèse requiert son autorisation.

Liste des tableaux

1.1 USArrests - Extrait du jeu de données 14
1.2 USArrests - Partition en 3 classes obtenue par la CAH 17
1.3 USArrests - Moyenne des variables par classes 18

2.1 Exemple de Résultat attendu 32

4.1 ACC et NMI de différents modèles existants appliqués sur MNIST . . . 51
4.2 hyperparamètres utilisés et résultats obtenus 52

Table des figures

1.1 USArrests - Dendogramme de l'ensemble des villes 15

1.2 USArrests - Choix du nombre de classe obtenu par NbClust 16

2.1 Organisation du perceptron multicouches 27

2.2 Fonction seuil 28

2.3 Fonction sigmoïde 29

2.4 Fonction tangente hyperbolique 29

2.5 Fonction RelU 30

2.6 Echantillon de la base MNIST 32

2.7 Courbes d'erreur d'entrainement et de test du perceptron multicouches 33

2.8 Exemple d'opération de convolution 35

2.9 Exemple d'opérations de « pooling max » et de « pooling average» . . 36

2.10 Architecture d'un réseau de neurones à convolution 37

2.11 Courbes d'erreur d'entrainement et de test du réseau de neurones à convolution 38

3.1 Architecture d'un autoencodeur simple 40

3.2 Architecture d'un autoencodeur convolutionnel 41

4.1 Représentation des différents chiffres par classes 53

Table des matières

Table des matières **2**

Résumé **5**

Abstract **6**

Avant-propos **7**

Introduction **8**

1 Méthodes de classification classiques **10**

1.1 Méthodes de classification hiérarchique 13

1.2 Méthodes de classification non hiérarchique 18

1.2.1 Méthode des centres mobiles 18

1.2.2 Méthode des nuées dynamiques 19

1.3 Modèle de mélange 20

1.3.1 Définition 20

1.3.2 Cas Gaussien 21

2 Méthodes neuronales **22**

2.1 Processus d'apprentissage 23

2.1.1 Apprentissage automatique 23

2.1.1.1 Types d'apprentissage 23

2.1.1.2 Règles d'apprentissage 24

2.1.2 Apprentissage profond 25

2.2 Perceptron Multicouches 26
2.2.1 Fonctionnement 27
2.2.2 Fonctions d'activation 27
2.2.3 Rétropropagation des erreurs 30
2.2.3.1 Définition 30
2.2.3.2 Algorithme de descente du gradient 30
2.2.4 Application du perceptron multicouches sur MNIST 31
2.2.5 Accuracy (ACC) 32
2.3 Réseau de neurones à convolution 33
2.3.1 Architecture 33
2.3.1.1 La couche de convolution 34
2.3.1.2 La fonction d'activation RelU 35
2.3.1.3 La couche « pooling » 35
2.3.1.4 La couche entièrement connectée 36
2.3.2 Application du réseau de neurones à convolution sur MNIST . . 37

3 Classification profonde **39**
3.1 Autoencodeur 39
3.1.1 Définition et architecture d'un autoencodeur 39
3.1.2 Types d'autoencoders 41
3.2 Classification profonde 42
3.2.1 Alternating Directed Method of Multipliers (ADMM) 42
3.2.2 Formulation 42
3.2.3 Architecture 43
3.3 Intégration des méthodes classiques dans les autoencodeurs profonds . . 47
3.3.1 Intégration de la méthode des K-means 47
3.3.2 Intégration du modèle de mélange gaussien 47

4 Application sur la base MNIST **49**
4.1 Définitions de quelques indicateurs de mesure de performances de modèle 49
4.1.1 Accuracy (ACC) 49

4.1.2 Normalized Mutual Information (NMI) 49
4.2 Application de la classification profonde sur MNIST 50
4.2.1 Comparaison des différents modèles de classification profonde . 50
4.2.2 Expérimentations et résultats 52

Conclusion et perspectives **54**

Bibliographie **57**

A Application du perceptron multicouches avec python **58**

B Application du réseau de neurones à convolution avec python **63**

C Application de la classification profonde avec python **68**

Résumé

Les méthodes de classification ont pour but de regrouper en classes assez homogènes un ensemble d'individus. Il existe plusieurs méthodes classiques dont notamment la Classifictaion Ascendante Hiérarchique, la méthode des centres mobiles dont les K-means, les modèles de mélange gaussien etc. Elles sont cependant limitées dans le traitement de données de grande dimension. Ce mémoire s'intéresse à la clasification profonde pour contourner les limitations des méthodes classiques. Elle est basée sur l'apprentissage profond et chapeaute plusieurs méthodes neuronales dont notamment le perceptron multicouches et les réseaux de neurones à convolution. Dans ce mémoire, nous exploiterons plus particulièrement les autoencodeurs convolutionnels. Ces derniers font appel à des méthodes de classification classiques au niveau de la couche latente faisant suite à une réduction de dimensionnalité des données. Ces deux méthodes sont toutes appliquées sur la base de données MNIST pour des fins de prédiction. Le perceptron nous donne de meilleurs résultats.
Les architectures et les résultats de plusieurs méthodes de classification profonde appliqués sur MNIST sont présentés dans un tableau récapitulatif. Ils sont comparés sur la base de leurs ACC et leurs NMI. En faisant des simulations sur certains hyperparamètres d'un modèle développé, nous sommes parvenus à améliorer les résultats.

Abstract

The aim of classification methods is to group together a set of individuals into fairly homogeneous classes. There are several classical methods including in particular the Ascending Hierarchical Classification, the method of mobile centers including K-means, Gaussian mixture models etc. They are however limited in the processing of large data. This thesis focuses on deep classification to get around the limitations of classical methods. It is based on deep learning and covers several neural methods including in particular the multilayer perceptron and convolutional neural networks. In this thesis, we more particularly use convolutional autoencoders. The latter use classical classification methods at the level of the latent layer following a reduction in dimensionality of the data. These two methods are all applied on the MNIST database for prediction purposes. The perceptron gives us better results.

The architectures and the results of several deep classification methods applied on MNIST are established in a summary table. They are compared on the basis of their ACCs and NMIs. By doing simulations on certain hyperparameters of a developed model, we managed to improve the results.

Avant-propos

Ce document s'intéresse aux méthodes de classification, en particulier la classification profonde. Les architectures inspirées des modèles d'apprentissage profond permettront de bien distinguer leurs performances. Elles sont appliquées sur la base de données MNIST.

Je tiens à adresser mes sincères remerciements à ma directrice de recherche Mme Nadia Ghazzali pour avoir accepté de diriger mon projet de recherche. Je souhaiterais également la remercier pour sa disponibilité et son appui tant du côté financier que pédagogique. Merci infiniment pour les efforts consentis dans la lecture et des corrections de fond et de forme apportées, sans lesquelles ce document ne pourrait atteindre ce stade de clarté et de précision.

Mes remerciements vont également à l'endroit de mes parents qui m'ont toujours orienté, appuyé et motivé. Un grand merci aussi à mes frères et soeurs, mes collègues, mes amis et à tous ceux qui ont contribué de près ou de loin à la rédaction de ce mémoire.

Introduction

La classification est une méthode très importante en analyse de données et peut être appliquée dans plusieurs domaines. Elle inclut des méthodes classiques dont le but est de former des classes selon des critères de distance ou de similarité. Cependant, elles sont limitées compte tenu de leur temps d'exécution et de la capacité de fonctionnement des machines. Ainsi, avec l'arrivée de l'intelligence artificielle, différentes méthodes de classification ont été développées et parviennent à contourner certaines limitations des méthodes classiques. En 1943, McCulloch et Pits ont conçu le premier modèle de réseau de neurones inspiré du fonctionnement des cellules nerveuses biologiques [21]. C'est dans ce cadre qu'est né le perceptron multicouches pour traiter des situations plus complexes en terme de taille de données. Par la suite, d'autres algorithmes ont été proposés suivant la nature des donnés. On note par exemple les réseaux de neurones à convolution pour le traitement de donnés d'images.
Des travaux de recherches récents ont proposé un cadre général de classification profonde (DeepCluster) pour intégrer certaines méthodes de classification classiques, notamment la méthode K-means et les modèles de mélange gaussien dans les modèles d'apprentissage profond. Plusieurs études et algorithmes ont été menés dans ce cadre. Les autoencoders sont des méthodes neuronales apprenant à reconstruire les entrées originales à la sortie le plus fidèlement possible. Selon la nature des données et la méthode de classification intégrée, un type d'autoencoder est généralement utilisé comme branche principale pour l'extraction des caractéristiques avant la formation des classes. D'autres modèles utilisent le perceptron multicouches.
Ce mémoire a pour objectif principal d'étudier l'architecture des méthodes de classification profonde et de comparer différentes méthodes existantes et appliquées sur la

base de données MNIST (http ://yann.lecun.com/exdb/mnist/) [17]. Elles sont évaluées suivant des indicateurs de performance. Nous avons par la suite effectué plusieurs simulations des hyperparamètres dans le but d'étudier le comportement d'un modèle. Le mémoire sera divisé en quatre chapitres. Dans le premier, nous étudierons les méthodes de classification classiques et nous les appliquerons sur des exemples simples. Le deuxième aborde les méthodes neuronales notamment le perceptron multicouches et les réseaux de neurones à convolution. Le chapitre trois concerne la classification profonde où nous décrirons son architecture et ses différentes composantes. Finalement, nous nous intéresserons aux méthodes de classification profonde appliquées sur la base MNIST dans le chapitre quatre avant de conclure et d'esquisser quelques perspectives.

Chapitre 1

Méthodes de classification classiques

L'objectif des méthodes de classification est de regrouper en classes assez homogènes un ensemble d'individus décrits par p variables. De ce fait, afin d'affecter chaque individu dans la classe qui le représente le mieux, il faudra commencer par se donner :

— Soit une mesure de distance : On appelle d une mesure de distance entre les individus, si pour tout x, y et $z \in \{1, ..., n\}$, on a :

  1. $d(x, y) \geq 0$
  2. $d(x, y) = d(y, x)$
  3. $d(x, z) \leq d(x, y) + d(y, z)$

  Lorsque les deux premières conditions sont vérifiées, on parle de dissimilarité.

— Soit une mesure de similarité : On appelle s un indice de similarité entre les individus, si pour tout x et $y \in \{1, ..., n\}$, on a :

  1. $s(x, y) \geq 0$
  2. $s(x, y) = s(y, x)$
  3. $s(x, y)$ augmente lorsque x et y se ressemblent davantage

Généralement, $s(x, y) \leq 1$ pour tout $1 \leq x, y \leq n$

Exemple 1. *La distance euclidienne est le type de distance le plus couramment utilisé. Elle se calcule comme suit :*
Si on considère deux individus x et y, décrits par p variables quantitatives,

$$d(x,y) = (\sum_{k\in p} |x_k - y_k|^2)^{1/2}$$

Pour former les classes, nous devons choisir un critère d'agrégation qui peut être-basé sur une distance, une dissimilarité ou une similarité. Ce critère nous permettra de déterminer si des classes sont suffisamment similaires pour n'en former qu'une seule. Il existe plusieurs méthodes d'agrégation que nous décrirons ici en termes de distance :

- **Le saut minimum** : $d(A,B) = \min d(x,y)$ avec $x \in A$ et $y \in B$
  C'est la plus petite distance entre deux membres des groupes A et B.

- **Le saut maximum** : $d(A,B) = \max d(x,y)$ avec $x \in A$ et $y \in B$
  C'est la plus grande distance entre deux membres des groupes A et B.
  Ces deux méthodes sont sensibles aux données aberrantes

Exemple 2. *Soit 5 individus a, b, c, d et e dont la matrice de distance est la suivante :*

$$A_1 = \begin{pmatrix} & a & b & c & d & e \\ a & 0 & 35 & 6 & 19 & 20 \\ b & 35 & 0 & 27 & 18 & 9 \\ c & 6 & 27 & 0 & 23 & 16 \\ d & 19 & 18 & 23 & 0 & 20 \\ e & 20 & 9 & 16 & 20 & 0 \end{pmatrix}$$

On applique le critère du saut maximum. A la première étape, les individus les plus proches sont a et c, avec une distance de 6. Ils sont donc regroupés et il importe de déterminer la distance entre la nouvelle classe (a,c) et chaque autre individu. On obtient alors la nouvelle matrice suivante :

$$A_2 = \begin{pmatrix} & (a,c) & b & d & e \\ (a,c) & 0 & 35 & 23 & 20 \\ b & 35 & 0 & 18 & 9 \\ d & 23 & 18 & 0 & 20 \\ e & 20 & 9 & 20 & 0 \end{pmatrix}$$

Ici, à la 2ème étape, la distance minimale est 9, regroupant b et e. On obtient alors la matrice suivante :

$$A_3 = \begin{pmatrix} & (a,c) & (b,e) & d \\ (a,c) & 0 & 35 & 23 \\ (b,e) & 35 & 0 & 20 \\ d & 23 & 20 & 0 \end{pmatrix}$$

A la 3ème étape, l'agrégation suivante est celle de (b,e) et d avec une distance de 20. Il ne reste plus que 2 éléments (a,c) et (b,e,d).

$$A_4 = \begin{pmatrix} & (a,c) & (b,e,d) \\ (a,c) & 0 & 35 \\ (b,e,d) & 35 & 0 \end{pmatrix}$$

Tous les individus sont finalement regroupés dans une seule classe avec une distance de 35.

- **Le saut moyen** : $d(A,B) = \frac{1}{n_A n_B} \sum_{i \in A} \sum_{j \in B} d(x_i, x_j)$
  avec n_A et n_B le nombre d'individus respectivement de deux classes A et B. C'est la moyenne des distances deux à deux entre les individus de A et B.

- **Méthode du centroïde** : $d(A,B) = d(\bar{x}_A, \bar{x}_B)$
  avec $\bar{x}_A = \frac{1}{n_A} \sum_{i \in A} x_i$; $\bar{x}_B = \frac{1}{n_B} \sum_{j \in B} x_j$ et $\bar{x}_{AB} = \frac{\bar{x}_A n_A + \bar{x}_B n_B}{n_A + n_B}$, les centres de gravité respectivement de A, de B et de la réunion de A et B.

— **Méthode de la médiane** : $d(A,B) = d(\bar{x}_A, \bar{x}_B)$
avec $\bar{x}_{AB} = \frac{\bar{x}_A + \bar{x}_B}{2}$, la moyenne des centres de gravité de A et B.

— **Méthode de Ward** : $d(A,B) = \frac{n_A n_B}{n_A + n_B} d^2(\bar{x}_A, \bar{x}_B)$
La méthode de Ward correspond à la perte d'inertie résultant de l'agrégation de la classe A et de celle de B. Elle tient également compte de la taille des classes contrairement à celle de la médiane.
Ces trois dernières méthodes s'appliquent uniquement sur des données quantitatives.
Il s'agira dans ce chapitre de décrire la méthodologie des méthodes de classification hiérarchiques et non hiérarchiques (ou méthodes de partitionnement) en faisant ressortir leurs différents éléments et d'exhiber leurs forces et leurs limites en s'appuyant sur un exemple classique. Ensuite, nous passerons en revue l'aspect probabiliste de la classification.

1.1 Méthodes de classification hiérarchique

Il existe deux familles de classification hiérarchique : la classification ascendante hiérarchique (CAH) et la classification descendante hiérarchique. On s'intéresse à la première famille dont les méthodes sont constituées par une suite de partitions emboitées. Les distances définies précédemment permettent itérativement de construire un arbre de classification dit dendogramme qui va aider sur le choix du nombre de classes à retenir. Ainsi, en découpant cet arbre à une certaine hauteur choisie, on produira la partition désirée.

La procédure de la CAH consiste à rassembler successivement les individus suivant le plus grand nombre de degré de leurs ressemblances, c'est-à-dire en des partitions de moins en moins fines.

L'algorithme de la CAH le plus simple est le suivant :

Etape 1 : Chaque individu forme sa propre classe;

Etape 2 : On regroupe les deux individus (ou groupes d'individus) les plus proches,

c'est-à-dire ceux dont la ressemblance est maximale ;

Etape k : Ainsi de suite, on regroupe les classes les plus proches jusqu'à ce que tous les individus soient dans la même classe ;

De façon générale, le problème majeur des méthodes de classification demeure la détermination du nombre de classes. Ainsi, nous avons utilisé le package Nbclust [4], implémenté dans le logiciel statistique R permettant de comparer une trentaine de critères d'arrêt et propose le nombre optimal de classes qu'il faudra retenir.

Exemple 3. *Nous appliquons la CAH sur la base USArrests [10] disponible directement dans le logiciel R (Hartigan 1975). L'ensemble de données contient des statistiques sur les arrestations pour 100 000 habitants concernant les variables agression, meurtre et viol dans chacun des 50 états américains. Nous disposons aussi du pourcentage de la population vivant en zone urbaine.*

Le tableau 1.1 nous renseigne sur les 6 premiers individus de la base.

	Murder	**Assault**	**UrbanPop**	**Rape**
Alabama	13,2	236	58	21,2
Alaska	10	263	48	44,5
Arizona	8,1	294	80	31
Arkansas	8,8	190	50	19,5
California	9	276	91	40,6
Colorado	7,9	204	78	38,7

TABLE 1.1 – USArrests - Extrait du jeu de données

Analyse avec R

```
USArrests
print(head(USArrests))
hc <- hclust (dist(USArrests))
plot (hc, cex=0.6, hang=-1)
rect.hclust (hc, k=3, border = 2:8)
```

La figure 1.1 donne la partition en 3 classes obtenue de la CAH.

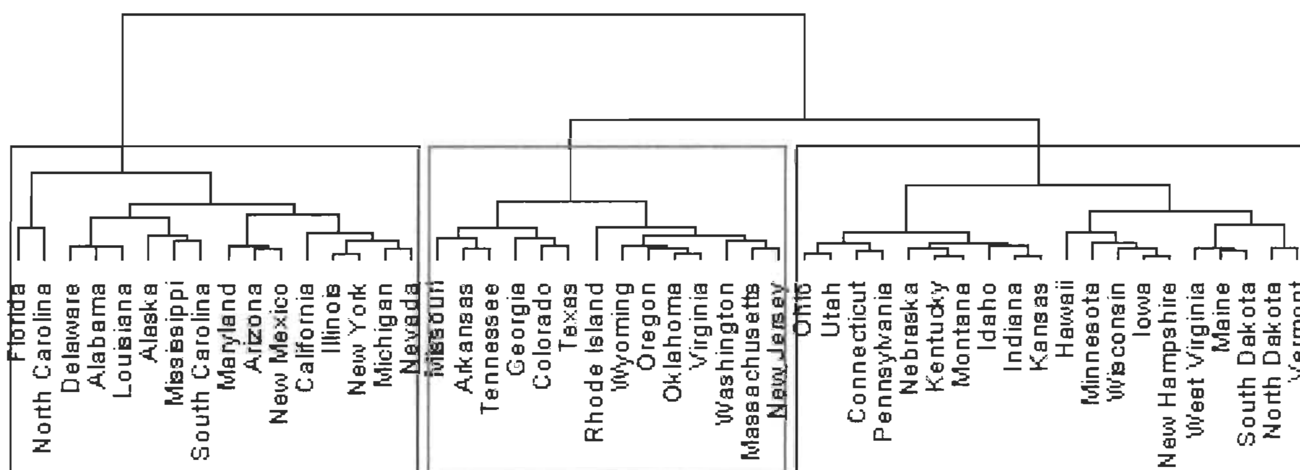

FIGURE 1.1 – USArrests - Dendogramme de l'ensemble des villes

```
#Partition optimale avec Nbclust
library(NbClust)
NbClust(USArrests, distance = "euclidean", min.nc = 2, max.nc = 8,
method =  "complete")

* Among all indices:
* 7 proposed 2 as the best number of clusters
* 13 proposed 3 as the best number of clusters
* 2 proposed 5 as the best number of clusters
```

```
* 1 proposed 6 as the best number of clusters
* 1 proposed 8 as the best number of clusters

***** Conclusion *****

* According to the majority rule, the best number of clusters is  3
```

Le nombre optimal de classes donné par NbClust est 3. Graphiquement, il correspond au nombre où nous avons une chute brutale de la courbe. Elle peut être visualisée sur la figure 1.2.

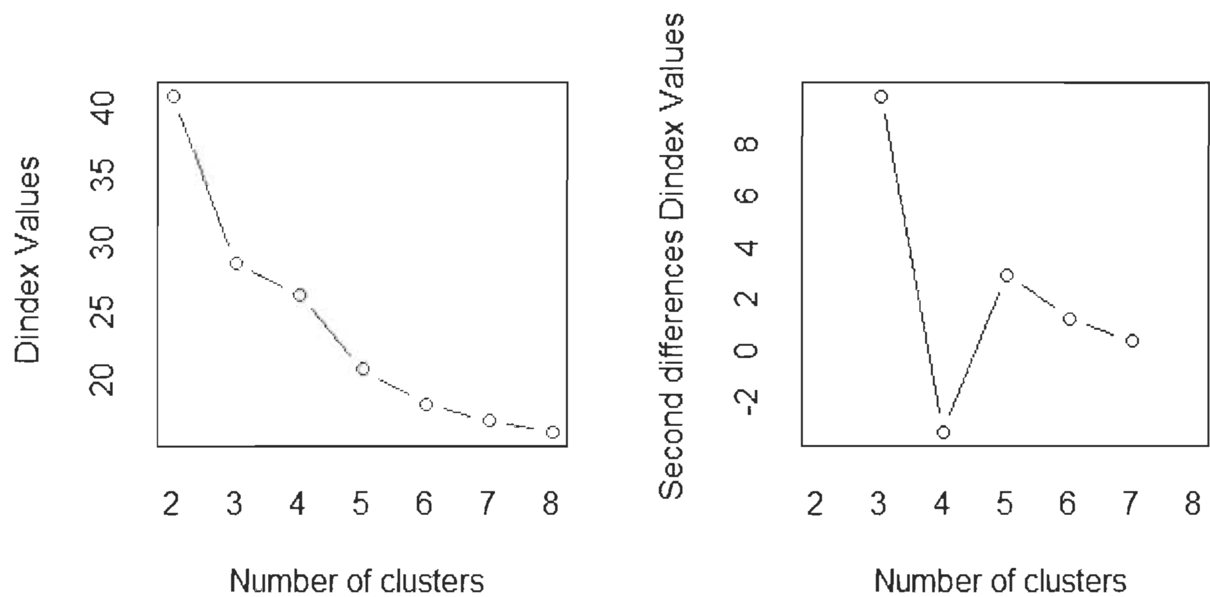

FIGURE 1.2 – USArrests - Choix du nombre de classe obtenu par NbClust

La composition des classes obtenue par la CAH est détaillée dans le tableau 1.2

Classe 1	Classe 2	Classe 3
Connecticut	Arkansas	Alabama
Alaska	Colorado	Alaska
Idaho	Georgia	Arizona
Indiana	Massachusetts	California
Iowa	Missouri	Delaware
Kansas	New Jersey	Florida
Kentucky	Oklahoma	Illinois
Maine	Oregon	Louisiana
Minnesota	Rhode Island	Maryland
Montana	Tennessee	Michigan
Nebraska	Texas	Mississippi
New Hampshire	Virginia	Nevada
North Dakota	Washington	New Mexico
Ohio	Wyoming	New York
Pennsylvania		North Carolina
South Dakota		South Carolina
Utah		
Vermont		
West Virginia		
Wisconsin		

TABLE 1.2 – USArrests - Partition en 3 classes obtenue par la CAH

Le tableau 1.3 nous donne la moyenne des variables pour les 3 classes. Nous remarquons que les classes sont formées selon les taux d'agression de meurtre et de viol. Les états de la classe 1 ont des taux plus faibles. Cependant, l'insécurité est plus observée au niveau des états de la classe 3.

Classes	Agression	Meurtre	Population zone Urbaine	Viol
1	4,27	87,55	59,75	14,39
2	8,21	173,29	70,64	22,84
3	11,81	272,56	68,31	28,36

TABLE 1.3 – USArrests - Moyenne des variables par classes

1.2 Méthodes de classification non hiérarchique

Les méthodes de classification non hiérarchique dites de partitionnement ont pour objectif de fixer dès le début le nombre de classes à former. Le problème auquel elles se proposent de répondre peut s'énoncer de la façon suivante :

— Etant donné un ensemble de n individus, on cherche à les classer en K classes $(2 \leq k \leq n-1)$;

— Etant donné un critère W permettant de mesurer la qualité d'une partition sur l'ensemble de toutes les partitions possibles en K classes sur les n individus, on détermine la partition P' qui optimise W.

Cependant, à cause de l'explosion combinatoire (lorsque n dépasse quelques dizaines), il est impossible d'énumérer toutes les partitions possibles.

Il existe plusieurs méthodes de partitionnement. Mais on analysera plus particulièrement la méthode des centres mobiles dont les K-means et une généralisation de cette dernière appelée méthode des nuées dynamiques.

1.2.1 Méthode des centres mobiles

L'idée générale de la méthode des centres mobiles consiste à fixer K classes et de déplacer les individus d'une classe à l'autre jusqu'à l'obtention d'une meilleure partition. La procédure générale de la méthode des centres mobiles est la suivante :

— Choisir une configuration initiale

— Affecter les individus aux classes les plus proches

— Calculer les nouveaux centres de ces classes

— Répéter l'algorithme jusqu'à ce qu'il y ait une stabilité de toutes les classes ou un nombre d'itérations maximal atteint sinon on recommence en réaffectant

les individus aux classes les plus proches.

Les nombreux algorithmes proposés diffèrent par le choix des K classes d'individus fixés a priori et par la définition qu'ils donnent à l'expression « meilleure partition ». S'agissant de la méthode des K-means, développée par McQueen en 1967, on choisit dans un premier temps K individus au hasard, considérés comme centres de classes [20]. Par la suite, on calcule la distance entre chaque individu et ces K centres de classes et on affecte les individus aux centres les plus proches. Puis, on calcule les centres de gravité de ces classes et on réaffecte les individus aux nouveaux centres les plus proches. Ainsi de suite, on reprend le même procédé jusqu'à ce que l'algorithme soit convergent.

L'avantage de la méthode des centres mobiles est qu'elle n'utilise que les distances des individus autour des centres, et par conséquent elle peut classer de grands ensembles de données.

L'inconvénient majeur est le fait de fixer a priori le nombre de classes à former. En plus, la partition finale à retenir dépend fortement du nombre de classes initialement fixé et donc de la partition initiale.

Exemple 4. *Nous allons appliquer la méthode des K-means sur les mêmes données que celles utilisées dans la CAH en se fixant K=3.*

```
> kmeans = kmeans(USArrests, centers = 3)
> kmeans = kmeans(USArrests, centers = 3, nstart = 10)
> kmeans
K-means clustering with 3 clusters of sizes 20, 14, 16
```

Nous obtenons les mêmes classes que celles de la CAH.

1.2.2 Méthode des nuées dynamiques

C'est une extension de la méthode des K-means [7]. Ici, au lieu de choisir un seul individu comme centre de gravité, on considère un ensemble d'individus représentatifs

de la classe. Cet ensemble est appelé noyau de la classe.

L'algorithme de la méthode des nuées dynamiques est le suivant :

- On sélectionne au hasard k sous ensembles notés N_j^0 de q objets parmi les n objets de l'ensemble E. Les N_j^0 sont les noyaux.
- On affecte ensuite les objets i de E aux classes j dont la distance $D(i, N_j^0)$ est minimale. On obtient alors une partition $P^0 = (P_1^0, ..., P_k^0)$ de E.
- On détermine dans chaque classe P_j^0 les q objets, notés N_j^1 qui minimisent $D(N_j^1, P_j^0)$.
- On réitère les deux dernières étapes jusqu'à l'obtention d'une partition stable.

Puisque le choix initial des noyaux se fait au hasard, il faudra alors répéter la procédure plusieurs fois de suite, en partant à chaque fois d'une famille différente de noyaux. On appelle « forme forte », un ensemble d'individus qui auront été classés ensemble dans toutes les partitions. Il est intéressant de déterminer ces individus car il faut une certaine homogénéité pour être dans la même classe malgré le choix de différentes familles de noyaux.

Enfin, la méthode des nuées dynamiques peut s'appliquer même sur des données dont la notion de centre de gravité n'a pas de sens.

1.3 Modèle de mélange

1.3.1 Définition

C'est l'aspect probabiliste de la classification [22]. La ressemblance entre les éléments d'un même groupe peut se traduire par le fait qu'ils proviennent tous d'une même loi de probabilité. La combinaison linéaire des lois associées à chaque classe conduit à un mélange de lois donné par :

$$P(X) = \sum_{k=1}^{K} \pi_k p_k(X)$$

avec $X = (X_1, ..., X_n)$, l'ensemble des individus ;

$\pi_k = P(Z_{ik} = 1)$, la probabilité a priori que l'individu X_i provienne du groupe G_K ;

$Z = (Z_1, ..., Z_i, ..., Z_n)$ leur partition en K classes $G_1, ...G_K$ où :

$Z_i = (Z_{i1}, ..., Z_{ik})$ tel que $Z_{ik} = 1$ si $i \in G_K$ et $Z_{ik} = 0$ sinon ;

$p_k(X) = P(X = X_i/Z_{ik} = 1)$, la probabilité conditionnelle qu'un individu X_i de X appartienne au groupe G_K.

Ainsi, on appelle modèle de mélange de lois m, le couple constitué par la loi paramétrique $p(X;\theta)$ et un espace Θ contenant l'ensemble des paramètres du modèle, dans lequel évolue $\theta = (\pi_k; \alpha_k)$

avec α_k le paramètre de la loi qui peut éventuellement être vectoriel.

1.3.2 Cas Gaussien

Dans le cadre des modèles de mélange gaussien, les données X_i $(i = 1, ..., n)$ sont des variables continues de $\mathbb{R}^d$ et la densité conditionnelle des composantes s'écrit :

$$p_k(x_i) = f(xi; \alpha_k) = \frac{1}{(2\pi)^{d/2}|\Sigma_k|^{1/2}} \exp\left\{-\frac{1}{2}(x_i - \mu_k)^T \Sigma_k^{-1}(x_i - \mu_k)\right\}$$

avec $\alpha_k = (\mu_k, \Sigma_k)$, $\mu_k \in \mathbb{R}^d$, la moyenne de la composante et $\Sigma_k \in \mathbb{R}^{d \times d}$ sa matrice de variance-covariance.

On cherche à obtenir une meilleure estimation de θ, donc des probabilités conditionnelles. Celeux et Govaert ont proposé en 1995 une décomposition spectrale des matrices de variance pour une interprétation géométrique plus simple [3]. Ainsi, chaque matrice est décomposée sous la forme $\Sigma_k = \lambda_k D_k A_k D_k^T$, avec :

$\lambda_k = |\Sigma_k|^{1/2}$, représentant le volume de la classe k ;

D_k, la matrice orthogonale des vecteurs propres de Σ_k et

A_k, la matrice diagonale des valeurs propres normalisées, de déterminant un et s'interprétant comme la forme de cette classe.

En permettant à certains paramètres de varier dans les classes, ils sont parvenus à obtenir quatorze modèles différents qu'ils regroupent en trois familles (la famille sphérique, la famille diagonale et la famille générale). En plus, compte tenu du modèle considéré, on peut retrouver certains critères classiques construits sur des notions de distance.

Chapitre 2

Méthodes neuronales

Un réseau de neurones se définit comme un ensemble d'algorithmes dont la conception est à l'origine très schématiquement inspirée du fonctionnement des cellules nerveuses biologiques. En effet, l'expérimentation menée sur les réseaux de neurones formels datent de 1943 [21], suite aux travaux de McCulloch et Pitts. Ensuite, différents algorithmes ont été proposés selon le domaine d'étude et les résultats attendus. Les méthodes neuronales sont susceptibles de contourner des problèmes complexes que les ordinateurs classiques ne peuvent pas en assurer la gestion. En effet, si les bases à traiter sont volumineuses, les tâches peuvent se révéler fastidieuses et coûteuses en ressources et, dans ce contexte, l'intelligence artificielle devient incontournable.
De nos jours, les réseaux de neurones peuvent être appliqués dans de nombreux secteurs dont notamment :

- Le traitement d'images : reconnaissance de caractères, de signatures, ou de formes, compression d'images, cryptage, classification, etc.
- Le traitement du signal : filtrage, classification, identification de sources, traitement de la parole, etc.
- Contrôle : commande de processus, diagnostic de pannes, contrôle de qualité, etc.
- Optimisation : planification, gestion, finance, etc.
- Simulation : prévisions météorologiques, recopies de modèles, etc.

Nous abordons dans cette partie, les différentes règles et types d'apprentissage d'un

réseau de neurones, puis, une application avec la méthode du perceptron multicouches.

2.1 Processus d'apprentissage

Cette section s'appuie sur les travaux de Parizeau [23].

2.1.1 Apprentissage automatique

L'apprentissage automatique (machine learning) est une branche de l'intelligence artificielle permettant au réseau d'apprendre de ses erreurs et d'améliorer sa performance à partir de stimuli qu'il reçoit de son environnement. Il permet d'améliorer un réseau de neurones couche après couche jusqu'à la sortie des résultats. Selon le nombre de neurones et de couches cachées du réseau, le modèle peut ne pas prévoir de manière fiable les observations futures. Ainsi, s'il n'a pas pu apprendre toute l'information contenue dans l'ensemble de données, on parle de sous-apprentissage. Le sur-apprentissage par contre survient lorsque le modèle se souvient d'un grand nombre d'exemples sans parvenir à remarquer des fonctionnalités. Il existe plusieurs types et règles d'apprentissage.

2.1.1.1 Types d'apprentissage

Compte tenu des informations disponibles et du type de réseau, nous pouvons avoir notamment :

- **L'apprentissage supervisé**

  L'apprentissage est dit supervisé lorsque les données qui entrent dans le processus sont déjà catégorisées et que les algorithmes doivent s'en servir pour prédire un résultat. Il sera ainsi possible de connaitre les sorties en se basant sur les données d'entrainement. On peut parexemple donner au système une liste de profils clients contenant des habitudes d'achat, et expliquer à l'algorithme lesquels sont des clients habituels et lesquels sont des clients occasionnels. Une fois l'apprentissage terminé, l'algorithme devra pouvoir déterminer tout seul à partir d'un profil client à quelle catégorie celui-ci appartient.

- **L'apprentissage non supervisé**

  En apprentissage non supervisé, notre réseau doit apprendre sans intervention externe, c'est à dire sans professeur, et donc on n'a aucune idée sur les sorties. C'est à la fin de l'apprentissage que le réseau formera des classes comportant des individus similaires.

2.1.1.2 Règles d'apprentissage

Il y a différentes règles d'apprentissage pour l'adaptation des poids des neurones. La méthode à appliquer dépend de ce qu'on veut obtenir du réseau en question [23]. On en distingue notamment :

- **L'apprentissage compétitif**

  Généralement, tous les neurones doivent apprendre simultanément, et de la même manière d'une couche à l'autre jusqu'à la sortie des données. En apprentissage compétitif, un seul neurone (ou éventuellement ses voisins) appelé «vainqueur» bénéficie d'une adaptation de son poids. Il s'agit donc de faire compétitionner les neurones afin de déterminer celui qui sera actif à un moment donné.

  Dans sa forme la plus simple, il n y a pas de couches de neurones intermédiaires entre les entrées et les sorties.

- **L'apprentissage par correction des erreurs**

  Il est principalement fondé sur la correction de l'erreur observée en sortie. En effet, Soit $a_i(t)$ la sortie que l'on obtient pour le neurone i au temps t et $d_i(t)$ la sortie que l'on désire obtenir pour ce même neurone au même temps.

  L'objectif de cette méthode d'apprentissage est de calculer l'erreur $e_i(t) = d_i(t) - a_i(t)$ et de chercher à la réduire autant que possible.

  L'apprentissage par correction des erreurs consiste à minimiser un indice de performance F basé sur les signaux d'erreur $e_i(t)$, dans le but de faire converger les sorties du réseau avec ce qu'on voudrait qu'elles soient. Un critère très populaire est la somme des erreurs quadratiques :

$$F(e(t)) = \sum_{i=1}^{S}(e_i(t))^2$$

avec $e(t) = [e_1(t)...e_i(t)...e_S(t)]^T$ et S le nombre de neurones de sortie du réseau.

— **L'apprentissage par la règle de Hebb**

Cette règle d'apprentissage est basée sur la règle du neurophysiologiste Donald Hebb : Si deux neurones interconnectés sont simultanément activés, alors le poids de la connexion qui les relie doit augmenté, sinon il est affaibli ou carrément éliminé [11].

2.1.2 Apprentissage profond

Depuis les années 2000, l'apprentissage profond (deep learning) est apparu comme une nouvelle zone de l'intelligence artificielle. C'est une classe de techniques d'apprentissage automatique, modélisant les données avec un haut niveau d'abstraction grâce à de multiples architectures. Les modèles d'apprentissage profond ont considérablement amélioré l'état de l'art en matière de reconnaissance vocale, reconnaissance visuelle d'objets, détection d'objets et de nombreux autres domaines tels que la découverte de médicaments et la génomique [16]. Plus le nombre de couches cachées est élevé, plus le réseau est profond. Au sein du cerveau humain, chaque neurone en activité peut produire un effet excitant ou inhibiteur sur ceux auxquels il est connecté. Le principe est similaire dans le cadre d'un réseau de neurones artificiel où un certain poids est assigné à différents neurones. Un neurone qui reçoit plus de charge exercera plus d'effet sur les neurones adjacents. La couche finale du réseau émet une réponse à ces signaux. Prenons un exemple concret dans le cadre de la reconnaissance d'images en utilisant un réseau de neurones pour reconnaitre les images qui comportent au moins un chat. Pour pouvoir identifier les chats, l'algorithme doit pouvoir distinguer les différents types de chats et reconnaitre un chat de manière précise quelque soit sa position. Pour cela, le réseau de neurones doit être bien entrainé en compilant plusieurs images de chats différents, mélangées avec d'autres types d'objets. Toutes ces images seront ensuite converties en données et transférées sur le réseau. C'est enfin la dernière couche du réseau qui va déduire s'il s'agit ou non d'un chat. Ce processus est

ensuite répété plusieurs fois (c'est le nombre d'époques) jusqu'à ce que le réseau soit en mesure de reconnaitre un chat sur n'importe quelle image. Il garde cette information en mémoire et s'en sert plus tard. Il s'agit donc d'un apprentissage supervisé. Il existe plusieurs architectures selon le champ d'application :

- Perceptron multicouches
- Réseau de neurones à convolution / Convolutional Neural Networks (CNN)
- Autoencodeur
- Restricted Boltzmann machine (RBM)
- Recurrent neural network (RNN)
- Long Short Term Memory networks (LSTM)
- etc

Dans ce mémoire, nous nous intéressons plus particulièrement aux 3 premiers réseaux pour décrire l'architecture de la classification profonde. Les autoencodeurs seront présentés au chapitre suivant.

2.2 Perceptron Multicouches

Le perceptron multicouches est un type de réseau de neurones, introduit en 1958 par Franck Rosenblatt [27]. De nos jours, c'est l'un des réseaux les plus utilisés pour résoudre des problèmes d'approximation, de classification ou de prédiction. Il est généralement appliqué aux problèmes d'apprentissage supervisé. L'apprentissage par correction des erreurs (rétropropagation du gradient) est utilisé pour effectuer les ajustements de pondération et biais entre les différents neurones. Nous étudierons le fonctionnement du perceptron multicouches ainsi que ses différentes composantes.

Dans sa forme la plus simple, le perceptron comprend deux couches : une couche d'entrée et une couche de sortie directement connectées. Dans ce cas, les fonctions d'activation sont de type seuil (0 ou 1) (Voir section 2.2.2). Il est ainsi limité dans ses applications car il faut en plus que les variables soient linéairement séparables.

Les auteurs Rumelhart et al. ont proposé une généralisation en incluant une ou plusieurs couches couchées [28] et utilisé d'autres types de fonctions d'activation qui

seront présentés à la section 2.2.2.

2.2.1 Fonctionnement

Le fonctionnement du perceptron multicouches est organisé en trois parties (voir figure 2.1) :

- **La couche d'entrée** : Elle est constituée d'un ensemble de neurones qui portent le signal d'entrée et reçoivent les données sources que l'on veut utiliser pour l'analyse.
- **Les couches cachées** : Elles constituent le véritable moteur de calcul du perceptron. C'est la phase de l'apprentissage. Ici, le réseau étudiera les relations entre les différentes variables et effectue ensuite les pondérations. Le choix du nombre de couches est arbitraire mais il se fait généralement par essai erreur, et les sorties d'une couche sont les entrées de la suivante.
- **La couche de sortie** : Elle donne enfin le résultat final obtenu par notre réseau.

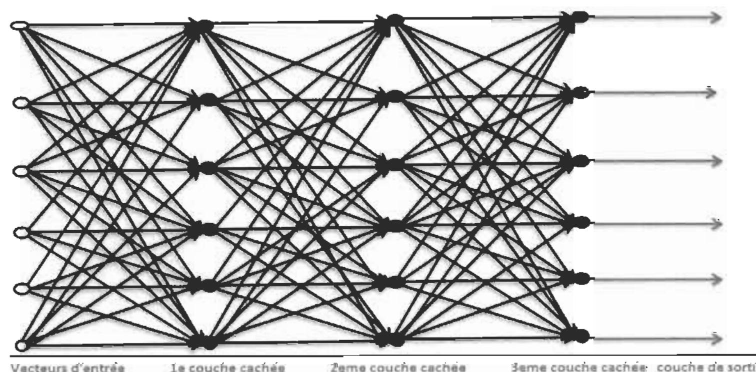

FIGURE 2.1 – Organisation du perceptron multicouches

2.2.2 Fonctions d'activation

Une fonction d'activation est utilisée pour entrainer le réseau en transmettant des signaux aux autres noeuds entre les différentes couches. Les poids sont ensuite ajustés au fur et à mesure. Les fonctions sont nombreuses mais les plus utilisées sont :

— **La fonction seuil** : Également appelée modèle tout ou rien, elle applique une valeur seuil sur son entrée et permet de prendre des décisions binaires (voir figure 2.2).

$$f(x) = \begin{cases} 0 & \text{si } x < 0 \\ 1 & \text{si } x \geq 0 \end{cases}$$

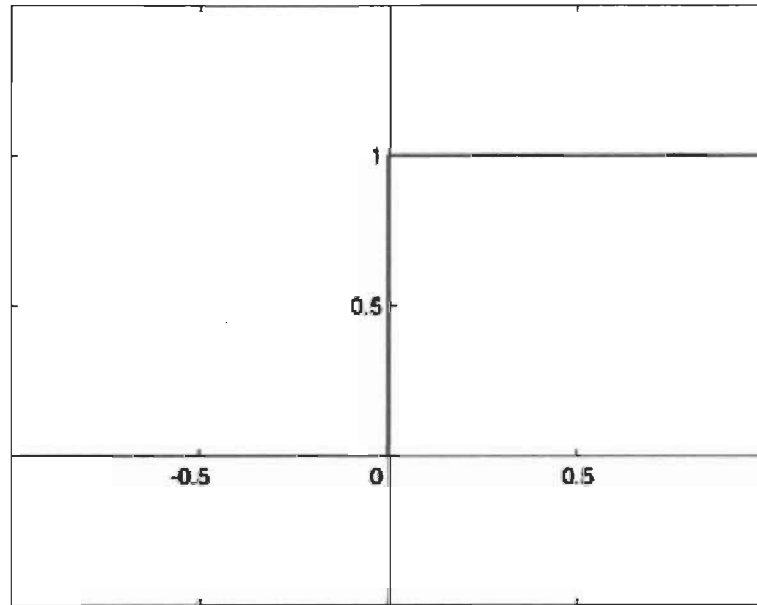

FIGURE 2.2 - Fonction seuil

— **La fonction sigmoïde** : Son équation est donnée par :

$$f(x) = \frac{1}{1 + e^{-x}}$$

Elle est similaire à la fonction seuil, mais elle est continue, dérivable en tout point et sa sortie est toujours comprise dans l'intervalle $[0, 1]$ (voir figure 2.3). Elle est généralement utilisée dans la dernière couche puisqu'elle permet de sortir une probabilité.

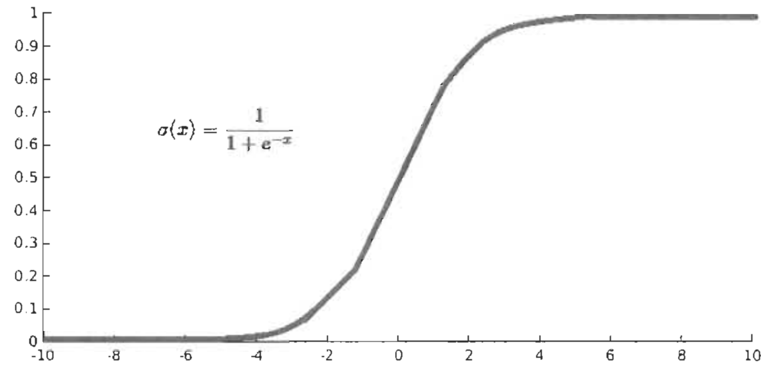

FIGURE 2.3 – Fonction sigmoïde

— **La fonction tangente hyperbolique** :

$$f(x) = \tanh(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}}$$

Elle ressemble à la fonction sigmoïde. La différence est qu'elle produit un résultat compris entre -1 et 1 et est centrée sur 0 (voir figure 2.4). Elle est utilisée dans des situations où on a pas des probabilités à ressortir.

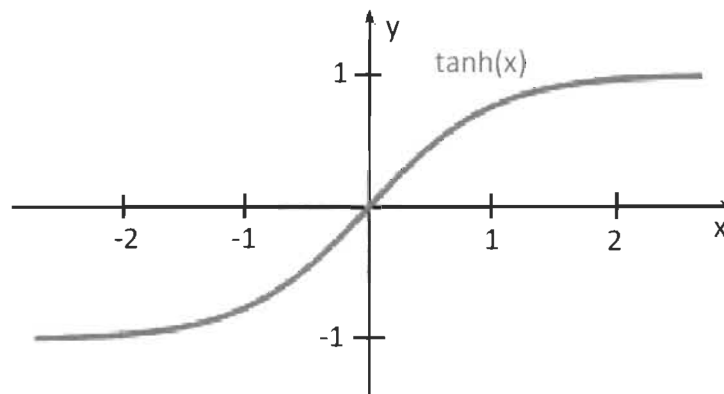

FIGURE 2.4 – Fonction tangente hyperbolique

— **La fonction ReLu (Unité de Rectification Linéaire)** : Elle est donnée par la formule :

$$f(x) = max(0, x)$$

Elle peut être considérée comme une généralisation de la fonction seuil (voir

figure 2.5). Si l'entrée x est négative, la sortie est 0, sinon la sortie est x. Elle est plus adaptée aux modèles d'apprentissage profond.

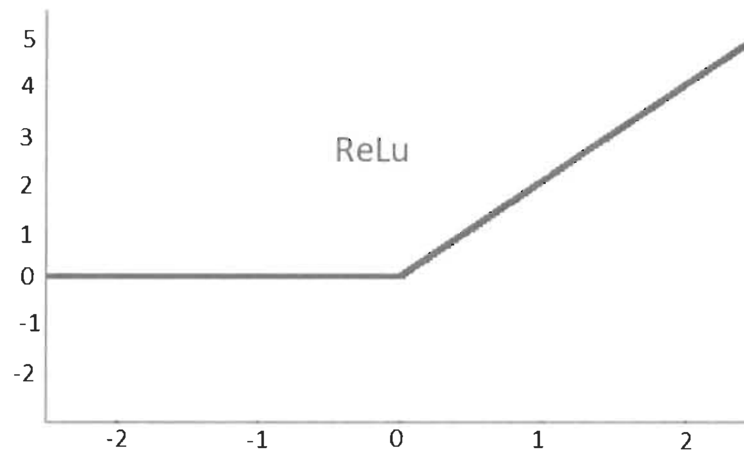

FIGURE 2.5 – Fonction RelU

2.2.3 Rétropropagation des erreurs

2.2.3.1 Définition

Il s'agit de calculer l'erreur commise par le réseau à la sortie et d'essayer de la minimiser [28]. Etant donné que l'apprentissage est de type supervisé, on propage cette erreur de la sortie vers les couches cachées jusqu'à l'entrée. L'algorithme utilise l'erreur quadratique moyenne comme indice de performance et est optimisé par la méthode de descente du gradient .

2.2.3.2 Algorithme de descente du gradient

Il est appliqué lorsqu'on cherche à trouver le minimum d'une fonction dont on connait l'expression analytique et dont le calcul direct de ce minimum s'avère difficile. Ce minimum doit être global pour tout le modèle. Il faut ainsi initialiser les poids à de petites valeurs aléatoires pour éviter d'obtenir un minimum local. C'est ce qui rend possible l'apprentissage et facilite la mise à jour des poids afin d'adapter le modèle aux données. La méthode de la descente du gradient consiste alors à construire une

suite de valeurs x_i de manière itérative :

$$x_{i+1} = x_i - \eta f'(x)$$

ou encore

$$\Delta x = -\eta f'(x)$$

avec η une constante appelée vitesse d'apprentissage et $f'(x)$ la dérivée de la fonction d'activation $f(x)$.

Δx représente la valeur que l'on ajoute à x à chaque itération.

De manière générale, en dimension n, si on note le vecteur d'entrée $X = (x_1, x_2, ..., x_n)$, on cherche à minimiser la fonction E de n variables $E(X) = E(x_1, x_2, ..., x_n)$. La formule de mise à jour est :

$$\Delta X = -\eta \nabla E(X)$$

$\nabla E(X)$ désigne la fonction « gradient » et est un vecteur avec n coordonnées. Dans la formule de la descente du gradient, la *jième* coordonnée de X, x_j, est mise à jour avec :

$$\Delta x_j = -\eta \frac{\partial E}{\partial x_j(X)} = -\eta \frac{\partial E}{\partial x_j(x_1, x_2, ..., x_n)}$$

2.2.4 Application du perceptron multicouches sur MNIST

La bibliothèque tensorflow (Voir le script en annexe A) de python permet l'entrainement du perceptron multicouches. Ce dernier est appliqué sur la base MNIST (http ://yann.lecun.com/exdb/mnist/), constituée de 70 000 chiffres manuscrits (7000 images par chiffre de 0 à 9), dont 60 000 pour l'entrainement des données et 10 000 pour le test. Chaque image a 28×28 pixels et est traitée par les algorithmes d'apprentissage comme un vecteur de $784 = 28 \times 28$ variables.

La figure 2.6 est un échantillon de 64 images tirées au hasard sur la base MNIST.

L'objectif est de reconnaitre les chiffres de la base test suite à l'entrainement des données.

Par exemple, s'il s'agit d'un 5 sur la base test, le résultat doit apparaitre sous forme

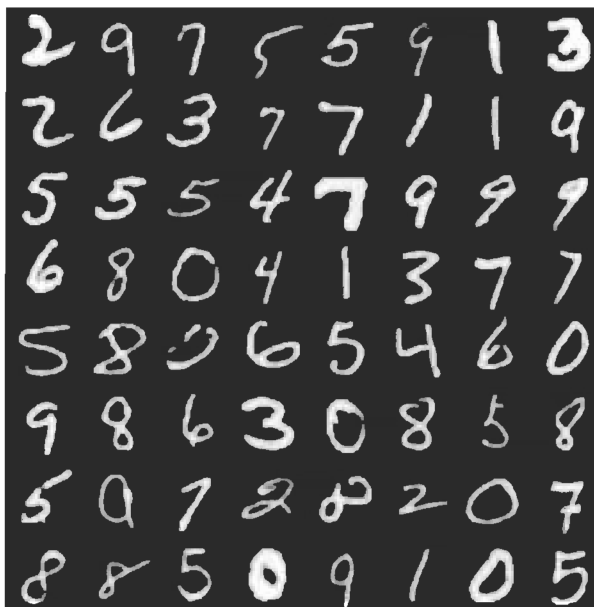

FIGURE 2.6 – Echantillon de la base MNIST

vectorielle comme l'indique le tableau 2.1 :

0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	**1**	0	0	0	0

TABLE 2.1 – Exemple de Résultat attendu

La précision du modèle est évaluée à l'aide de la métrique accuracy (ACC) compte tenu de la classe où il met l'image comparée à la vraie classe attendue de l'image.

2.2.5 Accuracy (ACC)

C'est le ratio entre le nombre d'observations bien prédites et le nombre total d'observations. Elle est donnée par la formule suivante :

$$ACC = max \frac{\sum_{i=1}^{N} 1(r_i = m(c_i))}{N}$$

avec 1(.), la fonction indicatrice,

r_i, le vrai label de l'observation i,

c_i, la classe contenant i

et $m(.)$ représentant l'ensemble des possibilités d'affectation des observations dans les

classes.

En considérant 150 neurones dans la couche intermédiaire avec 300 époques pour l'entrainement des données, nous aboutissons à une ACC = 0,93. La figure 2.7 donne les courbes d'erreur pour les jeux de données d'entrainement et de test en fonction du nombre d'époques.

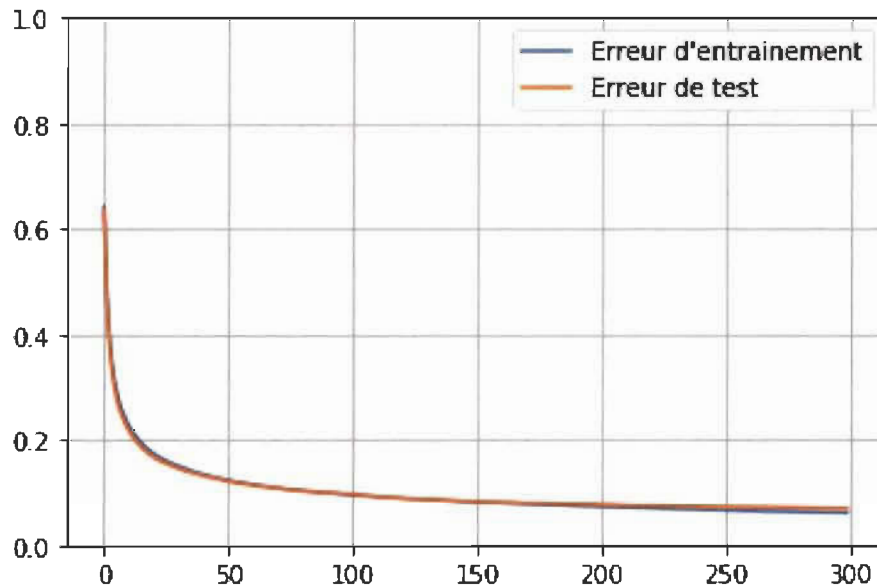

FIGURE 2.7 – Courbes d'erreur d'entrainement et de test du perceptron multicouches

2.3 Réseau de neurones à convolution

Les réseaux de neurones à convolution, inspirés du cortex visuel des animaux, sont très utiles pour effectuer la classification d'images. Grâce à leur architecture, ils sont capables d'exploiter certaines caractéristiques propres comprises dans l'image à travers les couches de convolution [24].

2.3.1 Architecture

L'architecture des réseaux de neurones à convolution est généralement constituée de trois types de couches : la couche de convolution, la couche « pooling » et la

couche entièrement connectée. Habituellement, une couche de convolution est suivie de la fonction d'activation RelU puis d'une couche de « pooling », cette séquence peut être répétée plusieurs fois jusqu'à la couche entièrement connectée pour former un réseau convolutif.

2.3.1.1 La couche de convolution

Il s'agit du premier traitement fait sur les images d'entrée. Celles ci sont dans un premier temps transformées en tableau de pixels. L'opération de convolution consiste à construire plusieurs cartes de caratéristiques ou « future maps » à travers des matrices appelées noyaux ou filtres. Ces noyaux sont générés par le réseau compte tenu des caractéristiques (formes, couleurs, textures, etc) qu'il souhaite extraire des images. On choisit le « stride », le pas auquel on déplace horizontalement et verticalement le noyau à travers l'entrant.
La phase de convolution est ainsi formulée comme suit :

$$s(t) = (f * g)(t) = \int_{-\infty}^{+\infty} f(\tau)g(t-\tau)d\tau$$

avec f représentant la fonction des entrées t, et g la fonction du noyau.
La convolution est ainsi décrite comme une moyenne pondérée de f, avec g représentant la pondération associée à f.

Exemple 5. *La figure 2.8 donne l'exemple d'une opération de convolution avec un noyau de dimension 3*3 appliqué sur l'image d'entrée. Les valeurs de la carte de caractéristiques sont obtenues de la façon suivante :*

- *Par exemple, le chiffre 0 qui se trouve sur la carte de caractéristique en haut à gauche s'obtient en faisant la somme des produits par correspondance des valeurs entre le filtre et la matrice d'ordre 3 se situant à l'extrémité gauche, en haut de l'image d'entrée ;*
- *Nous avons choisi un « stride » égal à 1 ;*
- *Ainsi de suite, on fait passer le filtre à travers l'image d'entrée en effectuant les mêmes opérations jusqu'à ce qu'elle soit totalement parcourue.*

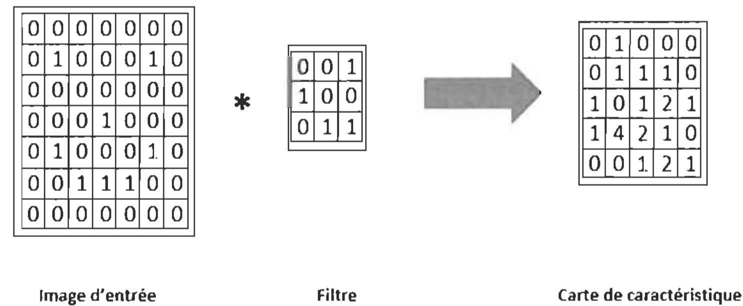

FIGURE 2.8 – Exemple d'opération de convolution

2.3.1.2 La fonction d'activation RelU

Il s'agit de la même fonction d'activation décrite dans la section 2.2.2. Elle est appliquée juste après la phase de convolution afin de rendre le modèle non linéaire. Les valeurs positives sont maintenues et les valeurs négatives sont ramenées à 0. La taille des cartes de caractéristiques reste toutefois inchangée.

2.3.1.3 La couche « pooling »

D'autres cartes de caractéristiques sont construites à partir des couches de convolution. L'objectif des couches de pooling est d'obtenir une invariance spatiale en réduisant la résolution des cartes de caractéristiques [30]. De manière similaire à la phase de convolution, on choisit le « stride » pour l'opération de « pooling ». Cette opération réduit également le risque de sur-apprentissage.
Deux types d'opérations de « pooling » sont généralement utilisés :

— Le « pooling max » : Conserve uniquement le maximum entre les valeurs sélectionnées dans la carte de caractéristique ;
— Le « pooling average » : Fait la moyenne de l'ensemble des valeurs sélectionnées dans la carte de caractéristique.

Exemple 6. *La figure 2.9 nous donne les résultats obtenus en appliquant les deux méthodes de « pooling » sur la carte de caractéristique obtenue précédemment.*

— *Par exemple, pour l'opération « Pooling max », le 1 en haut à gauche est égal à* $max(0, 1, 0, 1)$. *Ces valeurs sont les éléments de la matrice d'ordre 2 se situant*

à l'extrémité gauche, en haut de la carte de caractéristique ;

Nous avons choisi un « stride » égal à 2 ;

Ainsi de suite, on fait passer le filtre à travers l'image d'entrée en effectuant les mêmes opérations jusqu'à ce qu'elle soit totalement parcourue.

— *En ce qui concerne le « Pooling average », la démarche est similaire et les valeurs sont obtenues en faisant la moyenne des éléments de la matrice. Par exemple, le* 0,5 *en haut à gauche est obtenu en faisant :*

$$(0 + 1 + 0 + 1)/4 = 0,5$$

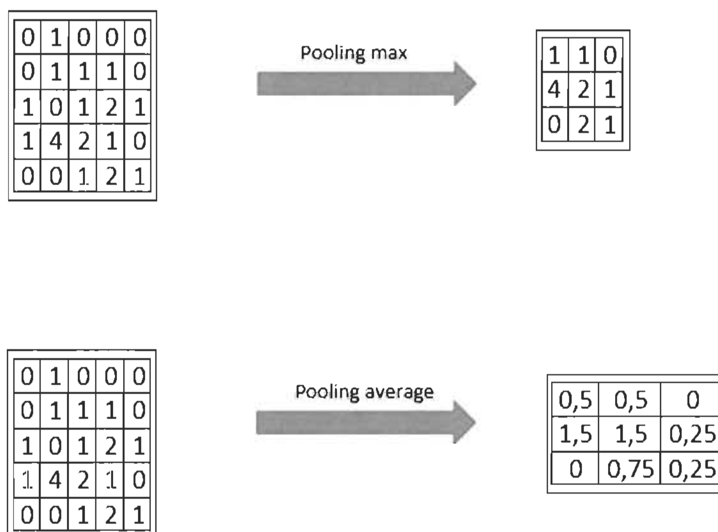

FIGURE 2.9 – Exemple d'opérations de « pooling max » et de « pooling average»

Dans leur article, D. Scherer et al. ont prouvé, grâce aux expérimentations qu'ils ont effectuées, que l'opération « pooling max » donne des résultats plus performants [30].

2.3.1.4 La couche entièrement connectée

La succession de couches de convolutions se termine par un réseau neuronal «fully connected». Celui-ci est composé de deux parties, la première partie est constituée de couches que l'on appelle des couches «fully connected». La particularité de cette

couche est que tous ses neurones sont connectés avec tous les neurones de la couche précédente, mais aussi avec tous les neurones de la couche suivante. La deuxième partie normalise les résultats, on connecte la dernière couche «fully connected» à un softmax. Cette fonction est utilisée pour produire une distribution de probabilité entre les différentes classes (chaque classe aura une valeur réelle comprise dans l'intervalle [0, 1]) [24].

La figure 2.10 donne l'architecture d'un réseau de neurones à convolution avec 2 étapes de convolution et 2 étapes de pooling. Elle est extraite de [25]

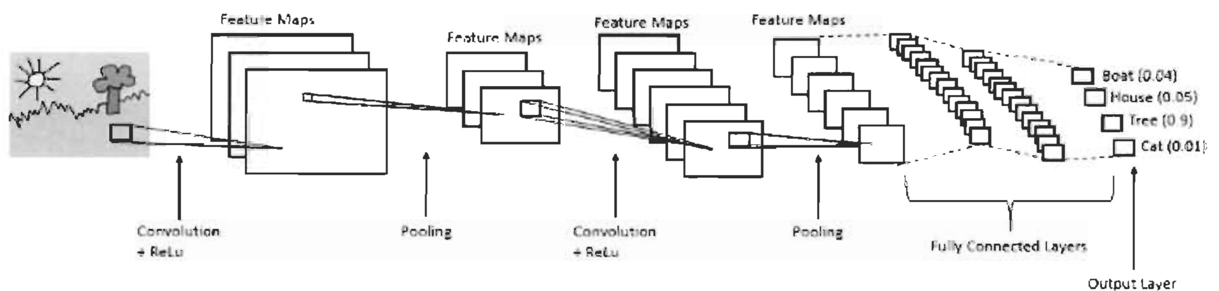

FIGURE 2.10 – Architecture d'un réseau de neurones à convolution [25]

2.3.2 Application du réseau de neurones à convolution sur MNIST

Nous avons utilisé la même bibliothèque, tensorflow (Voir le script en annexe B), pour l'entrainement du réseau de neurones à convolution et la même mesure ACC pour l'évaluation de la précision du modèle.

En considérant 150 neurones dans la couche intermédiaire avec 300 étapes pour l'entrainement des données, nous aboutissons à une ACC = 0,77. La figure 2.11 donne les courbes d'erreur pour les jeux de données d'entrainement et de test en fonction du nombre d'époques.

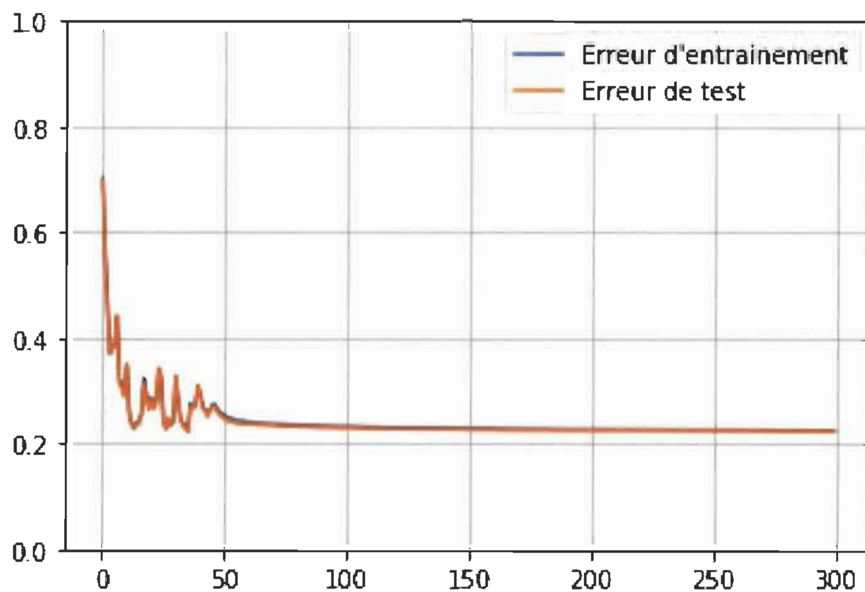

FIGURE 2.11 – Courbes d'erreur d'entrainement et de test du réseau de neurones à convolution

Chapitre 3

Classification profonde

Des travaux de recherche récents ont proposé un cadre général de classification profonde pour intégrer les méthodes de classification classiques (K-means et modèle de mélange gaussien) dans les modèles d'apprentissage profond. La classification profonde utilise un autoencodeur profond pour reconstruire les données et associer les fonctionnalités profondes aux méthodes de classification. Dans ce chapitre, nous décrivons l'architecture d'un autoencodeur dans un premier temps, ensuite nous parlerons des autoencodeurs convolutionnels, et enfin la classification profonde.

3.1 Autoencodeur

3.1.1 Définition et architecture d'un autoencodeur

Un autoencodeur est un réseau de neurones artificiel qui apprend à reconstruire son entrée originale à la sortie en minimisant l'erreur. Dans sa forme la plus simple, il est constitué d'une couche d'entrée, une couche cachée et une couche de sortie. La couche d'entrée a le même nombre de neurones que la couche de sortie. Afin de réduire la dimensionnalité, la couche cachée doit être plus petite. La figure 3.1 illustre l'architecture d'un autoencodeur simple. La transition de la couche d'entrée à la couche cachée est appelée étape de codage et la transition de la couche cachée à la couche de sortie est appelée étape de décodage. On peut définir ces deux transitions de la façon suivante :

$\phi : \mathcal{X} \longmapsto \mathcal{Z}$ tel que $x \longmapsto \phi(x) = \sigma(Wx + b) = z$

$\psi : \mathcal{Z} \longmapsto \mathcal{X}$ tel que $z \longmapsto \psi(z) = \sigma(\hat{W}z + \hat{b}) = \hat{x}$

avec $\mathcal{X}$ et $\mathcal{Z}$ respectivement les ensembles d'entrée et de sortie de l'encodeur, σ, une fonction d'activation, W et $\hat{W}$ des matrices de poids du réseau de neurones et enfin, b et $\hat{b}$ des vecteurs de biais.

Le vecteur d'entrée x est d'abord projeté sur la couche code z, appelée couche latente. La représentation cachée z est ensuite projetée sur la sortie $\hat{x}$ en utilisant le réseau de décodage. L'entrainement consiste donc à minimiser la fonction de perte :

$$\mathcal{L}(x, \hat{x}) = \|x - \hat{x}\|^2$$

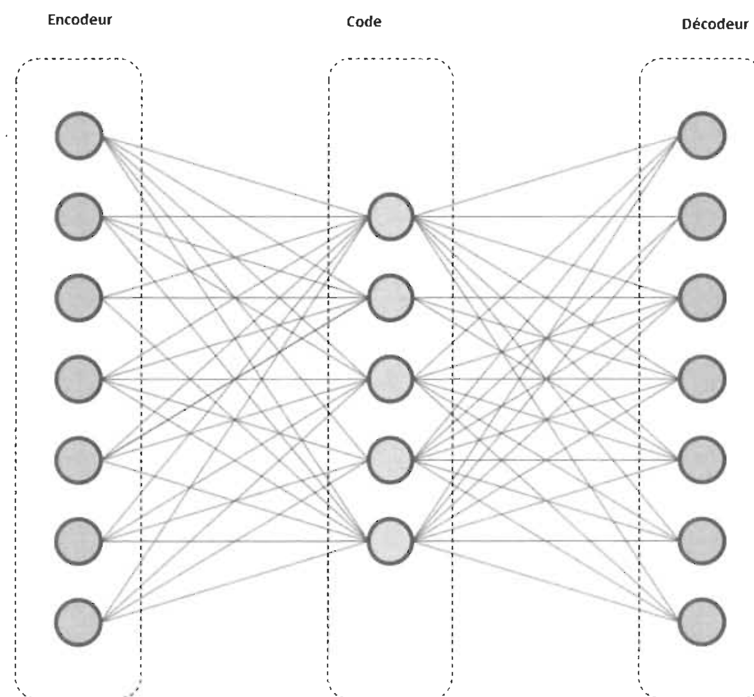

FIGURE 3.1 – Architecture d'un autoencodeur simple

3.1.2 Types d'autoencoders

Compte tenu de leurs architectures, Il existe plusieurs types d'autoencodeurs [26] notamment :

- Les autoencodeurs convolutionnels
- Les autoencodeurs variationnels
- Les autoencodeurs contractifs
- Les autoencodeurs undercomplete
- etc

Dans cette section, on s'intéresse aux autoencodeurs convolutionnels, généralement utilisés pour l'extraction de caractéristiques avant la classification.

Ils ont la même configuration que l'autoencodeur simple décrit précédemment. Les phases d'encodage et de décodage des données se font à l'aide de réseau de neurones convolutionnels. Les structures des deux réseaux sont symétriques par rapport à la couche de représentation cachée. La première couche de l'encodeur prend les données brutes en entrée. La figure 3.2 tirée de l'article [9] illustre l'architecture d'un autoencodeur convolutionnel.

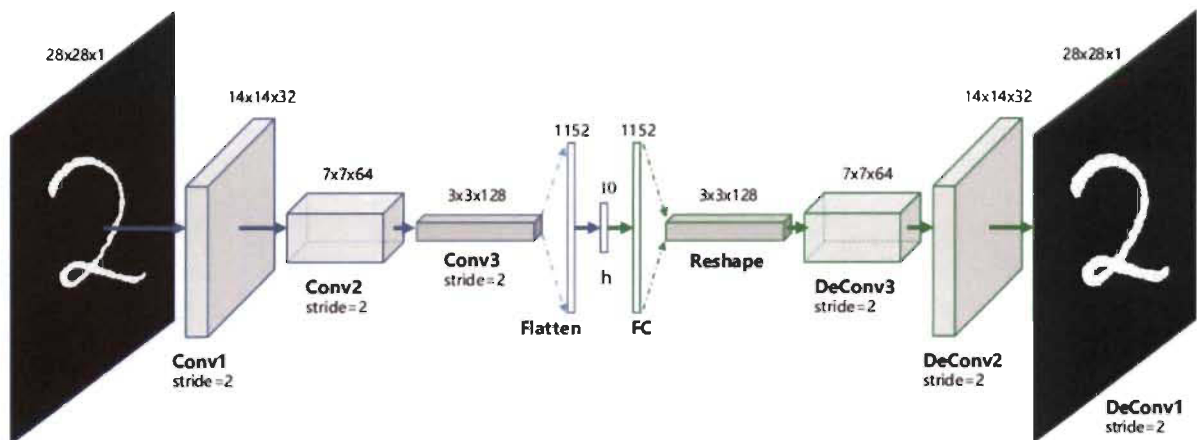

FIGURE 3.2 – Architecture d'un autoencodeur convolutionnel
[9]

3.2 Classification profonde

3.2.1 Alternating Directed Method of Multipliers (ADMM)

C'est un algorithme qui sert à résoudre des problèmes d'optimisation de fonctions convexes [31].
Une fonction est dite convexe lorsque pour tous éléments A et B, le segment [AB] se trouve au dessus de sa courbe de représentation.

Les auteurs K. Tian et al. ont considéré dans l'article [31] le problème général d'optimisation suivant :

$$min_{x \in \mathbb{R}^n} h(x) + o(Dx)$$

avec D une matrice à m lignes et n colonnes ;
h et o, deux fonctions convexes respectivement de $\mathbb{R}^n$ et $\mathbb{R}^m$;
Considérons une variable $z = Dx \in \mathbb{R}^m$, on a :

$$min\ h(x) + o(z)$$

ADMM utilise le lagrangien augmenté pour faire la décomposition suivante en y intégrant un paramètre $\rho > 0$:

$$\begin{cases} x^{k+1} \in arg\ min_{x \in \mathbb{R}^n} \{h(x) + o(z^k) + \langle \lambda^k, Dx - z^k \rangle + \frac{\rho}{2} \|Dx - z^k\|^2\} \\ z^{k+1} \in arg\ min_{z \in \mathbb{R}^m} \{h(x^{k+1}) + o(z) + \langle \lambda^k, Dx^{k+1} - z \rangle + \frac{\rho}{2} \|Dx^{k+1} - z\|^2\} \end{cases}$$

Finalement, ils ont obtenu l'équation suivante, indépendante des fonctions h et o :

$$\lambda^{k+1} = \lambda^k + \rho(Dx^{k+1} - z^{k+1})$$

3.2.2 Formulation

Elle utilise un type d'autoencodeur spécifique [31] et se fait en intégrant une variable muette Y au niveau de la couche de représentation cachée où l'affectation des individus dans leurs classes est faite. La variable muette permet de mettre à jour les paramètres du réseau et les centres de classe.

La classification profonde peut être formulée comme suit :

$$min : \|X - \hat{X}\|_F^2 + \lambda \times \mathcal{G}_w(Y)$$

avec X, les données brutes à classer et $\hat{X}$ la reconstruction de X ;
$\mathcal{G}_w(Y)$, une fonction de classification spécifique dépendant de la méthode classique à intégrer ;
λ, une constante permettant une transaction entre le réseau de neurones et la classification.
Son lagrangien augmenté est donnée par :

$$\mathcal{L}_\rho(\theta, Y, U, w) = \|X - \hat{X}\|_F^2 + \lambda \times \mathcal{G}_w(Y) + \frac{\rho}{2}\|Y - \phi_{\theta 1}(X) + U\|_F^2$$

avec U la réciproque de λ dans la solution donnée par la méthode ADMM
$\theta = \{\theta_1, \theta_2\}$ l'ensemble des paramètres de l'autoencodeur profond, avec θ_1 les paramètres de l'encodeur et θ_2 ceux du décodeur.

3.2.3 Architecture

Cette section est tirée des travaux de Aljalbout et al. publiés dans [1].
La classification profonde utilise généralement un type d'autoencodeur avec une procédure d'entrainement en deux temps :

- l'autoencodeur est entrainé à la perte de reconstruction d'erreur quadratique moyenne standard.
- l'autoencodeur est réglé avec une fonction de perte combinée.

L'architecture se présente comme suit :

1. **Branche principale du réseau de neurones**
   - Architecture de la branche principale où dans la plupart le réseau de neurones est utilisé pour transformer les entrées en une solution latente.
   - Ensemble de fonctionnalités avancées qui peuvent être prélevées sur une ou plusieurs couches. Généralement la sortie de la dernière couche est utilisée et c'est bénéfique en raison de la faible dimensionnalité sinon on fait une

combinaison des sorties de plusieurs couches.

2. **Perte du réseau de neurones**

— Perte non liée au regroupement : Elle est indépendante de l'algorithme et impose une contrainte sur le modèle appris. On peut avoir :

- Pas de perte liée à la classification : Aucune fonction supplémentaire de perte liée à la classification n'est utilisée ;
- Perte de reconstruction de l'autoencodeur : La partie décodeur n'est pas utilisée une fois la formation faite. En général, il s'agit d'une mesure de distance $d_{AE}(x_i; f(x_i))$ entre l'entrée x_i et sa reconstruction correspondante $f(x_i)$. L'erreur quadratique moyenne des deux variables est souvent utilisée :

$$L = d_{AE}(x_i; f(x_i)) = \sum_i \|x_i - f(x_i)\|^2$$

- Perte d'autoaugmentation : Elle propose un terme de perte qui rapproche la représentation de l'échantillon original et de ses augmentations.

$$L = -\frac{1}{N} \sum_N s(f(x); f(T(x)))$$

T est la fonction d'augmentation et s est une mesure de similarité ;
N représente le nombre d'individus.

— Perte par regroupement : Elle donne généralement de meilleurs résultats. On distingue notamment :

- Perte par K-means : Elle assure que la nouvelle représentation respecte les K-means (les individus sont répartis de manière égale autour des centres de classes). Pour obtenir une telle distribution, le réseau de neurones est formé avec la fonction de perte suivante :

$$L(\theta) = \sum_{i=1}^{N} \sum_{k=1}^{K} s_{ik} \|z_i - \mu_k\|^2$$

N et K respectivement le nombre d'individus et de classes ;

z_i est un point de donnée intégré ;

μ_k est un centre de classe ;

s_{ik}, une variable booléenne pour assigner z_i avec k.

Minimiser cette perte par rapport aux paramètres du réseau assure que la distance entre chaque individu et son centre de classe attribué est petite. On obtiendrait ainsi une meilleure qualité de classification.

- Durcissement des tâches : Nécessite l'utilisation d'attributions souples des individus aux centres de classes. Par exemple, la distribution de Student peut être utilisée comme noyau pour mesurer la similarité entre les individus et les centres. La distribution est formulée comme suit :

$$q_{ij} = \frac{(1 + \frac{\|z_i - \mu_j\|^2}{\nu})^{-\frac{\nu+1}{2}}}{\sum_{j'}(1 + \frac{\|z_i - \mu_{j'}\|^2}{\nu})^{-\frac{\nu+1}{2}}}$$

avec z_i un individu intégré, μ_j est le centroïde de la j^e classe et ν une constante généralement égale à 1.

La distribution suivante peut être utilisée afin de normaliser l'affectation des individus aux centres :

$$p_{ij} = \frac{\frac{q_{ij}^2}{\sum_i q_{ij}}}{\sum_{j'}(\frac{q_{ij'}^2}{\sum_i q_{ij'}})}$$

On cherche à minimiser la divergence entre les deux distributions, notées respectivement P et Q, par la formule suivante appelée divergence de Kullback-Leibler [15] :

$$L = KL(P\|Q) = \sum_i \sum_j p_{ij} log \frac{p_{ij}}{q_{ij}}$$

- Perte d'affectation équilibrée : Elle est utilisée pour que les affectations soient équilibrées. Elle est similaire à la formule de Kullback-Leibler et est définie par :

$$L = KL(G\|U)$$

avec U suivant la loi uniforme et G, la probabilité d'affectation des individus dans les classes :

$$g_k = P(y = k) = \frac{1}{N} \sum_i q_{ik}$$

— Combinaison des deux pertes (phase d'entrainement) Dans le cas où une fonction de perte non liée au regroupement et une fonction par regroupement sont utilisées, elles sont combinées de la façon suivante :

$$L(\theta) = \alpha L_c(\theta) + (1 - \alpha) L_n(\theta)$$

avec L_c, la fonction de perte par regroupement et L_n la fonction de perte non liée au regroupement ;
α est une constante $\in [0; 1]$

3. **Classification**

Durant l'entrainement de l'autoencodeur, les classes sont mises à jour selon l'un des critères suivants :

— Conjointement avec le modèle de réseau : Dans ce cas, les affectations sont formulées sous forme de probabilités et peuvent donc être incluses comme paramètres du réseau et optimisées par la méthode de rétropropagation des erreurs.

— De manière asynchrone avec le modèle de réseau : Dans ce cas, il faut fixer un critère d'arrêt, notamment :

- Un nombre d'itérations : L'algorithme s'exécute jusqu'à ce qu'un certain nombre d'éléments fixé changent les affectations entre deux itérations consécutives.
- Une fréquence des mises à jour : On note une étape de mise à jour pour chaque étape de mise à jour du modèle.

3.3 Intégration des méthodes classiques dans les autoencodeurs profonds

Dans cette partie, nous allons décrire l'intégration de la méthode des K-means et du modèle de mélange gaussien dans les modèles de classification profonde. Tel qu'illustré dans l'article [31], nous avons la fonction spécifique de la classification ainsi que son lagrangien augmenté pour les 2 méthodes.

3.3.1 Intégration de la méthode des K-means

Elle est illustrée de la façon suivante :

$$min : \frac{1}{N}\sum_{i=1}^{N}\|x_i - \hat{x}_i\|^2 + \frac{\lambda}{2} \times \|y_i - c_i{}^*\|^2$$

avec $y_i = f_{\theta_1}(x_i), i = 1, ..., N$ et $c_i{}^* = argmin_{c_j}\|y_i - c_j\|^2$ $,j = 1, ..., K$, le plus proche centroïde de y_i.

K et N , respectivement le nombre de classes et d'individus.

Son langrangien augmenté est donné par la formule suivante [31] :

$$\mathcal{L}_\rho(\theta, Y, U, C) = \frac{1}{N}\sum_{i=1}^{N}\|x_i - \hat{x}_i\|^2 + \frac{\lambda}{2} \times \|y_i - c_i{}^*\|^2 + \frac{\rho}{2}\|y_i - f_{\theta_1}(x_i) + u_i\|^2$$

3.3.2 Intégration du modèle de mélange gaussien

Dans le cadre des modèles de mélange, la fonction spécifique est la suivante :

$$min : \frac{1}{N}\sum_{i=1}^{N}\left\{\|x_i - \hat{x}_i\|^2 - \lambda \times \ln\left[\sum_{k=1}^{K}\pi_k\mathcal{N}(y_i|\mu_k, \Sigma_k)\right]\right\}$$

avec $\mathcal{N}(y_i|\mu_k, \Sigma_k)$ une distribution gaussienne multivariée de paramètres μ_k et Σ_k et $y_i = f_{\theta_1}(x_i), i = 1, ..., N$

Son langrangien augmenté est donné par [31] :

$$\mathcal{L}_\rho(\theta, Y, U, \mu, \Sigma, \pi) = \frac{1}{N}\sum_{i=1}^{N}\left\{\|x_i - \hat{x}_i\|_2^2 - \lambda \times \ln\left[\sum_{k=1}^{K}\pi_k p(y_i|\mu_k, \Sigma_k)\right] + \frac{\rho}{2}\|y_i - f_{\theta_1}(x_i) + u_i\|^2\right\}$$

Chapitre 4

Application sur la base MNIST

4.1 Définitions de quelques indicateurs de mesure de performances de modèle

4.1.1 Accuracy (ACC)

Nous rappelons la définition de l'accuracy qui est le ratio entre le nombre d'observations bien prédites et le nombre total d'observations. Elle est donnée par la formule suivante :

$$ACC = max \frac{\sum_{i=1}^{N} 1(r_i = m(c_i))}{N}$$

avec $1(.)$, la fonction indicatrice,
r_i, le vrai label de l'observation i,
c_i, la classe contenant i
et $m(.)$ représentant l'ensemble des possibilités d'affectation des observations dans les classes.

4.1.2 Normalized Mutual Information (NMI)

C'est une métrique très utile du fait qu'elle compare des méthodes de classification même si elles n'ont pas le même nombre de classes [2]

Elle est donnée par la formule suivante :

$$NMI = \frac{I(r, \hat{c})}{(H(r) + H(\hat{c}))/2}$$

avec r_i, le vrai label de l'observation i,

$\hat{c}$, la classe de l'observation i,

$I(.)$ définit par la métrique suivante :

Pour X et Y deux variables aléatoires,

$$I(X, Y) = \sum_{x \in X} \sum_{y \in Y} p_{x,y}(x, y) log \frac{p_{x,y}(x, y)}{p_x(x) p_y(y)}$$

et $H(.)$ représentant l'entropie

4.2 Application de la classification profonde sur MNIST

4.2.1 Comparaison des différents modèles de classification profonde

Les résultats présentés dans le tableau 4.1 permettent de comparer les performances des modèles de classification profonde existantes suivant leurs ACC et leurs NMI. Il est tiré de l'article [1]. On s'intéresse plus particulièrement à la méthode et l'architecture utilisées. Sur la base du NMI, la méthode la plus performante est celle proposée dans l'article [1] avec une NMI = 0,923. Ils ont utilisé un autoencoder convolutionnel au niveau de l'architecture et appliqué la méthode des centroïdes pour la classification. Les travaux de Hu et al. publiés dans [13] donnent les meilleurs résultats en terme d'accuracy avec ACC = 0,984. Le perceptron multicouches est la branche principale et ils ont effectué des prédictions pour l'affection dans les 10 classes.

Method	Arch	Features for clutering	Non-clustering loss	Clustering loss	Clustering algorithm	NMI	ACC
JULE [35]	CNN	CNN output	-	Agglomerative loss	Agglomerative clustering	0.915	-
CCNN [12]	CNN	Internal CNN layer	-	Cluster classification loss	k-Means	0.876	-
SCCNN [19]	CNN	CNN output	Classification loss	-	Standard Clustering Method	-	-
DEC [33]	MLP	Encoder output	Autoencoder reconstruction loss	Cluster assignment hardening	estimates centroids (as trainable parameters) ; soft assignments done based on distance to centroids	0.800	0.843
DBC [18]	CNN	Encoder output	Autoencoder reconstruction loss	Cluster assignment hardening	K-Means	0.917	0.964
DEPICT [8]	CNN	Encoder output	Autoencoder reconstruction loss	Balanced-assignment	Network estimates centroids (as trainable parameters) and predicts assignments	0.916	0.965
DCN [34]	MLP	Encoder output	Autoencoder reconstruction loss	K-Means loss	Network estimates centroids ; soft assignments based on distance	0.810	0.830
DEN [14]	MLP	Encoder output	Autoencoder reconstruction loss	Locality preserving + Group sparsity	k-Means	-	-
Neural Clustering [29]	MLP	Concat. of encoder layers output	Autoencoder reconstruction loss	-	k-Nearest Neighbors	-	0.966
NMMC [6]	DBN	Deepest DBN layer	Usual DBN loss	Maximum margin for mixture components	-	-	-
UMMC [5]	DBN	Deepest DBN layer	Usual DBN loss	Localitypreserving loss + Cluster assignment hardening	K-Means	0.864	-
VaDE [36]	VAE	Encoder output	VAE loss with mixture model for clusters		Network estimates centroids	-	0.944
TAGnet [32]	Sparse-coding iterations time-unfolded (reformulated) into a deep network		Sparse coding (implicit in architecture)	Similarity graph regularization (implicit) - Cluster classification or maximum margin	Network predicts soft-assignments to clusters	0.651	0.692
IMSAT [13]	MLP	MLP output	Information maximization + Self-Augmentation loss	-	Network predicts soft-assignments to clusters	-	**0.984**
CDL [1]	CNN	Encoder output	Autoencoder reconstruction loss	Cluster assignment hardening	Network estimates centroids ; soft assignments based on distance	**0.923**	0.961

TABLE 4.1 – ACC et NMI de différents modèles existants appliqués sur MNIST [1]

4.2.2 Expérimentations et résultats

Dans cette partie, on teste quelques hyperparamètres afin de maximiser les indicateurs de performance du modèle. A titre d'exemple, un hyperparamètre peut être le nombre de couches cachées, le nombre d'époques, le « batch size » etc.
On effectue différentes simulations des hyperparamètres du réseau afin de déterminer la meilleure combinaison, celle qui maximise les indicateurs. Finalement, nous avons ajusté le coefficient d'apprentissage ainsi que le « batch size », tout en conservant le même nombre d'époques. Le coefficient d'apprentissage détermine la taille du pas à chaque itération tout en se déplaçant vers un minimum de la fonction de perte. Le « batch size » représente le nombre d'observations entrainé à chaque itération. (Voir le script en annexe C).
Suivant l'architecture que nous avons développée tout au long du chapitre 3, nous utilisons celle présentée dans l'article [9] et nommée DCEC. Elle utilise un autoencodeur convolutionnel comme branche principale, constitué de 3 couches de convolutions et de 3 couches de « pooling » avec respectivement 32, 64 et 128 cartes de caractéristiques. Les pas vertical et horizontal pour le déplacement des noyaux est égal à 2. La première couche entièrement connectée est constituée de 1152 neurones et la deuxième de 10 neurones de sortie pour la classification de 0 à 9. L'encodage et le décodage ont les mêmes configurations. La classification se fait avec la méthode des K-means au niveau de la couche code. La distribution de Student est utilisée comme fonction de perte pour mesurer la similarité entre les observations et les centres de classes. En adoptant une procédure d'entrainement de 200 époques par lots de 300 (« batch size ») et un coefficient d'apprentissage égal à 0,01, nous sommes parvenus à améliorer leurs résultats. Le tableau 4.2 fournit les hyperparamères utilisés par les auteurs [9] et ceux que nous avons utilisés ainsi que les indicateurs de performance ACC et NMI.

	Hyperparamètres			ACC (%)	NMI (%)
	« batch size »	Coefficient d'apprentissage	Nombre d'époques		
DCEC [9]	256	0,001	200	88,97	88,49
JND	300	0,01	200	90,25	89,47

TABLE 4.2 – hyperparamètres utilisés et résultats obtenus

Les classes sont d'autant plus homogènes (séparées en 10 blocs représentant les chiffres de 0 à 9) que les indicateurs de performance sont proches de 1.
Avec nos résultats (JND voir https ://github.com/Jean-Noel-Diouf843/Deep-Cluster), nous avons représenté les dix classes obtenues par des nuages de points, tel qu'illustrés à la figure 4.1. Chaque classe correspond à un chiffre. On note que certains nuages sont plus denses que d'autres.

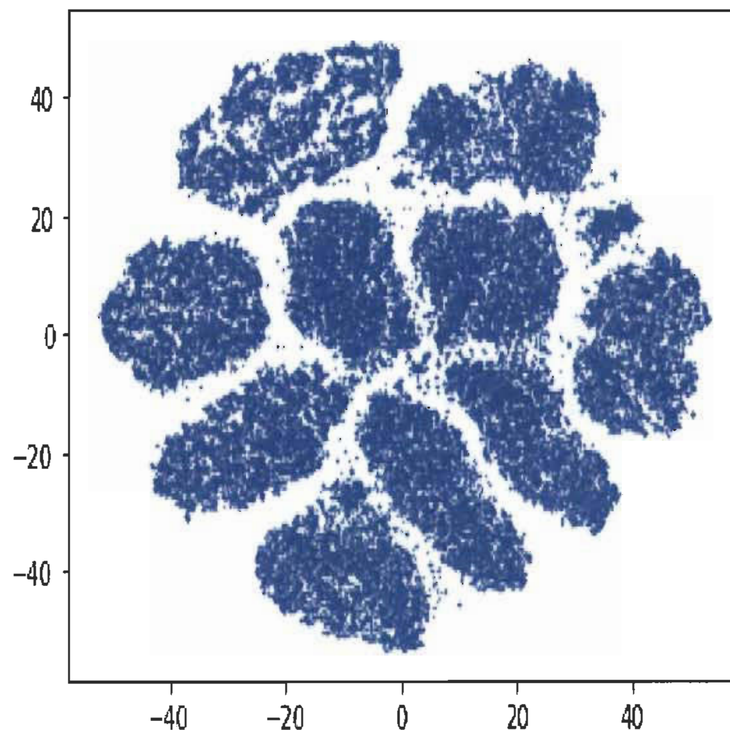

FIGURE 4.1 – Représentation des différents chiffres par classes

Le modèle a été développé à l'aide de la bibliothèque tensorflow de python et est disponible sur github à l'adresse https ://github.com/Jean-Noel-Diouf843/Deep-Cluster. Le modèle a déjà été entrainé et il est donc possible de vérifier l'exactitude des résultats.

Conclusion et perspectives

La problématique de ce mémoire s'inscrit dans le cadre général des méthodes de classification qui consistent à donner le meilleur regroupement en classes de l'ensemble à étudier. Les méthodes de classification hiérarchiques et non hiérarchiques sont décrites dont notamment la classification ascendante hiérarchique (CAH) et la méthode des centres mobiles. Les méthodes d'apprentissage automatiques sont aussi décrites. Toutes ces méthodes peuvent être utilisées à des fins de classification ou de prédiction. Cependant, leurs algorithmes différent et elles n'ont pas les mêmes performances. Le perceptron multicouches, bien étant moins adapté pour la classification de données d'images, a fourni de meilleurs résultats que les réseaux de neurones à convolution. Toutefois, ces méthodes exploitent directement les données fournies en entrée et peuvent ainsi subir la haute dimensionnalité. L'autoencodeur, qui est une méthode neuronale permet de réduire la dimension des données et tente de les reconstruire le plus fidèlement possible. Les méthodes développées basées sur l'apprentissage profond (Deep learning) l'utilisent et effectuent une méthode de classification classique (généralement K-means et modèle de mélange gaussien) au niveau de sa couche latente. Les nombreux algorithmes de classification profonde fournissent ainsi des résultats plus performants en terme d'ACC, de NMI.
Comme perspectives, il serait intéressant d'analyser le comportement des modèles de classification profonde si on intègre une autre méthode classique (analyse discriminante ou nuées dynamiques par exemple) à l'intérieur de la couche latente. De ce fait, quelque soit la nature et la dimension des données, ils pourraient être plus adaptés et plus utiles. Il serait également important d'élargir notre analyse en comparant les méthodes sur d'autres jeux de données et avec d'autres indicateurs de performance.

Bibliographie

[1] E. Aljalbout, V. Golkov, Y. Siddiqui, M. Strobel, and D. Cremers. Clustering with deep learning : Taxonomy and new methods. 2018.

[2] A. Amelio and C. Pizzuti. Is normalized mutual information a fair measure for comparing community detection methods. 2015.

[3] G. Celeux and G. Goavert. Gaussian parsimonious clustering models. Mai 1995.

[4] M. Charrad, N. Ghazzali, V. Boiteau, and A. Niknafs. Nbclust : An r package for determining the relevant number of clusters in a data set. 2014.

[5] D. Chen, J. Lv, and Z. Yi. Unsupervised multi-manifold clustering by learning deep representation. 2017.

[6] G. Chen. Deep learning with nonparametric clustering. 2015.

[7] E. Diday. Une nouvelle méthode en classification automatique et reconnaissance des formes : la méthode des nuées dynamiques. 19(2) :19–33, 1971.

[8] K. G. Dizaji, A. Herandi, C. Deng, W. Cai, and H. Huang. Deep clustering via joint convolutional autoencoder embedding and relative entropy minimization. 2017.

[9] X. Guo, X. Liu, E. Zhu, and J. Yin. Deep clustering with convolutional autoencoders. Octobre 2017.

[10] J. A. Hartigan. Clustering algorithms : Wiley and Sons. 1975.

[11] Donald O. Hebb. *The organization of behavior :*. Wiley, New York.

[12] C.-C. Hsu and C.-W. Lin. Cnn-based joint clustering and representation learning with feature drift compensation for large-scale image data. 2017.

[13] W. Hu, T. Miyato, S. Tokui, E. Matsumoto, and M. Sugiyama. Learning discrete representations via information maximizing self augmented training. 2017.

[14] P. Huang, Y. Huang, W. Wang, and L. Wang. Deep embedding network for clustering. 2014.

[15] S. Kullback and R. A. leibler. On information and sufficiency. the annals of mathematical statistics. pages 79–86, 1951.

[16] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521(14539) :436–444, 2015.

[17] Y. Lecun, B. E. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. E. Hubbard, and L. D. Jackel. Handwritten digit recognition with a back-propagation network. pages 396–404, 1990.

[18] F. Li, H. Qiao, B. Zhang, and X. Xi. Discriminatively boosted image clustering with fully convolutional auto-encoders. 2017.

[19] Y. Lukic, C. Vogt, O. Düurr, and T. Stadelmann. Speaker identification and clustering using convolutional neural networks. 2016.

[20] J. B. MacQueen. Some methods for classification and analysis of multivariate observations. 1967.

[21] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4) :115–133, Dec 1943.

[22] G. J. McLachlan. Discriminant analysis and statistical pattern recognition : Wiley and Sons. 1992.

[23] M. Parizeau. Réseaux de neurones, université laval. 2009.

[24] L. Pibre, M. Chaumont, D. Ienco, and J. Pasquet. étude des réseaux de neurones pour la stéganalyse. Mai 2016.

[25] Prabhu. Understanding of convolutional neural network (cnn) - deep learning. Mars 2018.

[26] A. Prakash. Different types of autoencoders. 2014.

[27] F. Rosenblatt. The perceptron : A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65 :386–408, 1958.

[28] D.E. Rumelhart, G.E. Hinton, and R.J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088) :533–536, 1986.

[29] S. Saito and R. T. Tan. Neural clustering : Concatenating layers for better projections. 2017.

[30] D. Scherer, A. Müller, and S. Behnke. Evaluation of pooling operations in convolutional architectures for object recognition. Septembre 2010.

[31] K. Tian, S. Zhou, and J. Guan. Deepcluster : A general clustering framework based on deep learning. 2017.

[32] Z. Wang, S. Chang, J. Zhou, M. Wang, and T.S. Huang. Learning a task-specific deep architecture for clustering. 2016.

[33] J. Xie, R. Girshick, and A. Farhadi. Unsupervised deep embedding for clustering analysis. 2016.

[34] B. Yang, X. Fu, N. D. Sidiropoulos, and M. Hong. Towards k-means-friendly spaces : Simultaneous deep learning and clustering. 2016.

[35] J. Yang, D. Parikh, and D. Batra. Joint unsupervised learning of deep representations and image clusters. 2016.

[36] Y. Zheng, H. Tan, B. Tang, and H. Zhou. Variational deep embedding : A generative approach to clustering. 2016.

Annexe A

```
https://github.com/L42Project/Tutoriels/tree/master
/Tensorflow/tutoriel1

#Installation et importation des bibliothèques

pip install tensorflow==2.0
pip install opencv-python

import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()

import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plot
import cv2


#Initialisation des paramètres

nbr_ni=100
learning_rate=0.0001
taille_batch=150
```

```
nbr_entrainement=300


#Importation de la base MNIST

mnist_train_images=np.fromfile("C:/Users/user/Desktop
/Application_MNIST_Python
/train-images.idx3-ubyte", dtype=np.uint8)[16:].
reshape(-1, 784)/255
mnist_train_labels=np.eye(10)[np.fromfile("C:/Users/user
/Desktop/Application_MNIST_Python/train-labels.idx1-ubyte",
 dtype=np.uint8)[8:]]
mnist_test_images=np.fromfile("C:/Users/user/Desktop/
Application_MNIST_Python/
t10k-images.idx3-ubyte", dtype=np.uint8)[16:].
reshape(-1, 784)/255
mnist_test_labels=np.eye(10)[np.fromfile("C:/Users/user/Desktop
/Application_MNIST_Python/t10k-labels.idx1-ubyte",
dtype=np.uint8)[8:]]


#Création des "placeholder"
(Interface pour effectuer les calcules du réseau)

ph_images=tf.placeholder(shape=(None, 784), dtype=tf.float32)
ph_labels=tf.placeholder(shape=(None, 10), dtype=tf.float32)

#Définition des poids, biais et fonctions d'activation en vue de
l'entrainement du réseau
```

```
wci=tf.Variable(tf.truncated_normal(shape=(784, nbr_ni)),
 dtype=tf.float32)
bci=tf.Variable(np.zeros(shape=(nbr_ni)), dtype=tf.float32)
sci=tf.matmul(ph_images, wci)+bci
sci=tf.nn.sigmoid(sci)

wcs=tf.Variable(tf.truncated_normal(shape=(nbr_ni, 10)),
dtype=tf.float32)
bcs=tf.Variable(np.zeros(shape=(10)), dtype=tf.float32)
scs=tf.matmul(sci, wcs)+bcs
scso=tf.nn.softmax(scs)

#Définition de la fonction de perte et de la mesure du critère de
performance

loss=tf.nn.softmax_cross_entropy_with_logits_v2(labels=ph_labels,
 logits=scs)
train=tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)
accuracy=tf.reduce_mean(tf.cast(tf.equal(tf.argmax(scso, 1),
 tf.argmax(ph_labels, 1)), dtype=tf.float32))

#Entrainement du réseau, calcul de la mesure du critère de
performance, affichage des courbes d'erreur et des résultats

with tf.Session() as s:
s.run(tf.global_variables_initializer())

tab_acc_train=[]
tab_acc_test=[]
for id_entrainement in range(nbr_entrainement):
```

```
print("ID entrainement", id_entrainement)
for batch in range(0, len(mnist_train_images), taille_batch):
s.run(train, feed_dict={
ph_images: mnist_train_images[batch:batch+taille_batch],
ph_labels: mnist_train_labels[batch:batch+taille_batch]
})

tab_acc=[]
for batch in range(0, len(mnist_train_images), taille_batch):
acc=s.run(accuracy, feed_dict={
ph_images: mnist_train_images[batch:batch+taille_batch],
ph_labels: mnist_train_labels[batch:batch+taille_batch]
})
tab_acc.append(acc)
print("accuracy train:", np.mean(tab_acc))
tab_acc_train.append(1-np.mean(tab_acc))

tab_acc=[]
for batch in range(0, len(mnist_test_images), taille_batch):
acc=s.run(accuracy, feed_dict={
ph_images: mnist_test_images[batch:batch+taille_batch],
ph_labels: mnist_test_labels[batch:batch+taille_batch]
})
tab_acc.append(acc)
print("accuracy test :", np.mean(tab_acc))
tab_acc_test.append(1-np.mean(tab_acc))

plot.ylim(0, 1)
plot.grid()
plot.plot(tab_acc_train, label="Erreur d'entrainement")
```

```
plot.plot(tab_acc_test, label="Erreur de test")
plot.legend(loc="upper right")
plot.show()

resulat=s.run(scso, feed_dict={ph_images:
mnist_test_images[0:taille_batch]})
np.set_printoptions(formatter={'float': '{:0.3f}'.format})
for image in range(taille_batch):
print("image", image)
print("sortie du réseau:", resulat[image], np.argmax(resulat[image]))
print("sortie attendue :", mnist_test_labels[image],
np.argmax(mnist_test_labels[image]))
cv2.imshow('image', mnist_test_images[image].reshape(28, 28))
if cv2.waitKey()&0xFF==ord('q'):
break
```

Annexe B

```
#https://github.com/L42Project/Tutoriels/tree/master
/Tensorflow/tutoriel2


#Importation des bibliothèques

import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plot
import cv2

#Définition des couches, des poids et des biais en vue de
l'entrainement des données

def convolution(couche_prec, taille_noyau, nbr_noyau):
w=tf.Variable(tf.random.truncated_normal(shape=(taille_noyau,
taille_noyau, int(couche_prec.get_shape()[-1]), nbr_noyau)))
b=np.zeros(nbr_noyau)
result=tf.nn.conv2d(couche_prec, w, strides=[1, 1, 1, 1],
 padding='SAME')+b
return result
```

```
def fc(couche_prec, nbr_neurone):
w=tf.Variable(tf.random.truncated_normal
(shape=(int(couche_prec.get_shape()[-1]), nbr_neurone),
dtype=tf.float32))
b=tf.Variable(np.zeros(shape=(nbr_neurone)), dtype=tf.float32)
result=tf.matmul(couche_prec, w)+b
return result

#Définition des paramètres

taille_batch=150
nbr_entrainement=300
learning_rate=0.001

#Importation de la base MNIST

mnist_train_images=np.fromfile("C:/Users/user/Desktop
/Application_MNIST_Python/train-images.idx3-ubyte",
dtype=np.uint8)[16:].reshape(-1, 28, 28, 1)/255
mnist_train_labels=np.eye(10)[np.fromfile("C:/Users/user/Desktop
/Application_MNIST_Python/train-labels.idx1-ubyte",
dtype=np.uint8)[8:]]
mnist_test_images=np.fromfile("C:/Users/user/Desktop
/Application_MNIST_Python/t10k-images.idx3-ubyte",
 dtype=np.uint8)[16:].reshape(-1, 28, 28, 1)/255
mnist_test_labels=np.eye(10)[np.fromfile("C:/Users/user/Desktop
/Application_MNIST_Python/t10k-labels.idx1-ubyte",
dtype=np.uint8)[8:]]

#Création des "placeholder"
```

```
(Interface pour effectuer les calcules du réseau)

ph_images=tf.placeholder(shape=(None, 28, 28, 1),
dtype=tf.float32)
ph_labels=tf.placeholder(shape=(None, 10),
dtype=tf.float32)


#Définitions des valeurs des paramètres des couches

result=convolution(ph_images, 5, 32)
result=convolution(result, 5, 32)
result=tf.nn.max_pool(result, ksize=[1, 2, 2, 1],
strides=[1, 2, 2, 1],
padding='SAME')

result=convolution(result, 5, 128)
result=convolution(result, 5, 128)
result=tf.nn.max_pool(result, ksize=[1, 2, 2, 1],
strides=[1, 2, 2, 1],
padding='SAME')

result=tf.contrib.layers.flatten(result)

result=fc(result, 512)
result=tf.nn.sigmoid(result)
result=fc(result, 10)
scso=tf.nn.softmax(result)

#Définition de la fonction de perte et de la mesure du critère
```

```
de performance

loss=tf.nn.softmax_cross_entropy_with_logits_v2(labels=ph_labels,
 logits=result)
train=tf.train.AdamOptimizer(learning_rate).minimize(loss)
accuracy=tf.reduce_mean(tf.cast(tf.equal(tf.argmax(scso, 1),
 tf.argmax(ph_labels, 1)), tf.float32))

#Entrainement du réseau, calcul de la mesure du critère de
 performance, affichage des courbes d'erreur et des résultats

with tf.Session() as s:
s.run(tf.global_variables_initializer())
tab_train=[]
tab_test=[]
for id_entrainement in np.arange(nbr_entrainement):
tab_accuracy_train=[]
tab_accuracy_test=[]
for batch in np.arange(0, len(mnist_train_images), taille_batch):
s.run(train, feed_dict={
ph_images: mnist_train_images[batch:batch+taille_batch],
ph_labels: mnist_train_labels[batch:batch+taille_batch]
})
for batch in np.arange(0, len(mnist_train_images), taille_batch):
precision=s.run(accuracy, feed_dict={
ph_images: mnist_train_images[batch:batch+taille_batch],
ph_labels: mnist_train_labels[batch:batch+taille_batch]
})
tab_accuracy_train.append(precision)
for batch in np.arange(0, len(mnist_test_images), taille_batch):
```

```
precision=s.run(accuracy, feed_dict={
ph_images: mnist_test_images[batch:batch+taille_batch],
ph_labels: mnist_test_labels[batch:batch+taille_batch]
})
tab_accuracy_test.append(precision)
print("> Entrainement", id_entrainement)
print("   train:", np.mean(tab_accuracy_train))
tab_train.append(1-np.mean(tab_accuracy_train))
print("   test :", np.mean(tab_accuracy_test))
tab_test.append(1-np.mean(tab_accuracy_test))

plot.ylim(0, 1)
plot.grid()
plot.plot(tab_train, label="Erreur d'entrainement")
plot.plot(tab_test, label="Erreur de test")
plot.legend(loc="upper right")
plot.show()

resulat=s.run(scso, feed_dict={ph_images:
 mnist_test_images[0:taille_batch]})
np.set_printoptions(formatter={'float': '{:0.3f}'.format})
for image in range(taille_batch):
print("image", image)
print("sortie du réseau:", resulat[image], np.argmax(resulat[image]))
print("sortie attendue :", mnist_test_labels[image],
np.argmax(mnist_test_labels[image]))
cv2.imshow('image', mnist_test_images[image])
if cv2.waitKey()&0xFF==ord('q'):
break
```

Annexe C

https://github.com/XifengGuo/DCEC

```
******************Datasets****************************
#Importation de la library numpy
(Pour effectuer des analyses de données avec python)

import numpy as np

#Définition des formats de la base de données MNIST



def load_mnist():
# the data, shuffled and split between train and test sets
from keras.datasets import mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()

x = np.concatenate((x_train, x_test))
y = np.concatenate((y_train, y_test))
x = x.reshape(-1, 28, 28, 1).astype('float32')
x = x/255.
print('MNIST:', x.shape)
return x, y
```

```
*****************Metrics***************************

#Importation de la library numpy
(Pour effectuer des analyses de données avec python)

import numpy as np

from sklearn.metrics import normalized_mutual_info_score,
adjusted_rand_score

#Définition des critères de performance

nmi = normalized_mutual_info_score
ari = adjusted_rand_score


def acc(y_true, y_pred):
"""
Calculate clustering accuracy. Require scikit-learn installed

# Arguments

y: true labels, numpy.array with shape `(n_samples,)`
y_pred: predicted labels, numpy.array with shape `(n_samples,)`


accuracy, in [0,1]
```

```
"""
y_true = y_true.astype(np.int64)
assert y_pred.size == y_true.size
D = max(y_pred.max(), y_true.max()) + 1
w = np.zeros((D, D), dtype=np.int64)
for i in range(y_pred.size):
w[y_pred[i], y_true[i]] += 1
from sklearn.utils.linear_assignment_ import linear_assignment
ind = linear_assignment(w.max() - w)
return sum([w[i, j] for i, j in ind]) * 1.0 / y_pred.size

****************ConvAE**************************

#Définition du répertoire de travail

import os
os.chdir("C:/Users/user/Downloads/DCEC-master/DCEC-master")

#Importation des fonctions en vue de la définition des couches et
de l'entrainement du modèle

from keras.layers import Conv2D, Conv2DTranspose, Dense, Flatten,
 Reshape
from keras.models import Sequential, Model
from keras.utils.vis_utils import plot_model
import numpy as np

#Définition des couches

def CAE(input_shape=(28, 28, 1), filters=[32, 64, 128, 10]):
```

```
model = Sequential()
if input_shape[0] % 8 == 0:
pad3 = 'same'
else:
pad3 = 'valid'
model.add(Conv2D(filters[0], 5, strides=2, padding='same',
activation='relu', name='conv1', input_shape=input_shape))

model.add(Conv2D(filters[1], 5, strides=2, padding='same',
activation='relu', name='conv2'))

model.add(Conv2D(filters[2], 3, strides=2, padding=pad3,
activation='relu', name='conv3'))

model.add(Flatten())
model.add(Dense(units=filters[3], name='embedding'))
model.add(Dense(units=filters[2]*int(input_shape[0]/8)*
int(input_shape[0]/8), activation='relu'))

model.add(Reshape((int(input_shape[0]/8), int(input_shape[0]/8),
filters[2])))
model.add(Conv2DTranspose(filters[1], 3, strides=2, padding=pad3,
activation='relu', name='deconv3'))

model.add(Conv2DTranspose(filters[0], 5, strides=2, padding='same',
activation='relu', name='deconv2'))

model.add(Conv2DTranspose(input_shape[2], 5, strides=2,
padding='same',
 name='deconv1'))
```

```
model.summary()
return model

if __name__ == "__main__":
from time import time

#Définition des hyperparamètres

import argparse
parser = argparse.ArgumentParser(description='train')
parser.add_argument('--dataset', default='mnist')
parser.add_argument('--n_clusters', default=10, type=int)
parser.add_argument('--batch_size', default=300, type=int)
parser.add_argument('--epochs', default=200, type=int)
parser.add_argument('--save_dir', default='results/temp', type=str)
args = parser.parse_args()
print(args)

import os
if not os.path.exists(args.save_dir):
os.makedirs(args.save_dir)

#Importation de la base MNIST

from datasets import load_mnist
if args.dataset == 'mnist':
x, y = load_mnist()

#Définition du modèle
```

```
model = CAE(input_shape=x.shape[1:], filters=[32, 64, 128, 10])
plot_model(model, to_file=args.save_dir + '/%s-pretrain-model.png'
 % args.dataset, show_shapes=True)
model.summary()

#Compilation du modèle

optimizer = 'adam'
model.compile(optimizer=optimizer, loss='mse')
from keras.callbacks import CSVLogger
csv_logger = CSVLogger(args.save_dir + '/%s-pretrain-log.csv' %
args.dataset)

#Entrainement du réseau

t0 = time()
model.fit(x, x, batch_size=args.batch_size, epochs=args.epochs,
callbacks=[csv_logger])
print('Training time: ', time() - t0)
model.save(args.save_dir + '/%s-pretrain-model-%d.h5' %
 (args.dataset, args.epochs))

#Extractions des caractéristiques

feature_model = Model(inputs=model.input,
outputs=model.get_layer(name='embedding').output)
features = feature_model.predict(x)
print('feature shape=', features.shape)

#Classification par la méthode K-means
```

```
from sklearn.cluster import KMeans
km = KMeans(n_clusters=args.n_clusters)

#Prédiction et affichage des résultas des critères de performance

features = np.reshape(features, newshape=(features.shape[0], -1))
pred = km.fit_predict(features)
import metrics
print('acc=', metrics.acc(y, pred), 'nmi=', metrics.nmi(y, pred),
'ari=', metrics.ari(y, pred))



*****************DCEC**********************

#Importation des fonctions en vue de la définition des couches et
de l'entrainement du modèle

from time import time
import numpy as np
import keras.backend as K
from keras.engine.topology import Layer, InputSpec
from keras.models import Model
from keras.utils.vis_utils import plot_model
from sklearn.cluster import KMeans
import metrics
from ConvAE import CAE
```

```
class ClusteringLayer(Layer):
"""
Clustering layer converts input sample (feature) to soft label,
 i.e. a vector that represents the probability of the
sample belonging to each cluster. The probability is calculated
with student's t-distribution.

# Example
```
model.add(ClusteringLayer(n_clusters=10))
```
# Arguments
n_clusters: number of clusters.
weights: list of Numpy array with shape `(n_clusters, n_features)`
witch represents the initial cluster centers.
alpha: parameter in Student's t-distribution. Default to 1.0.
# Input shape
2D tensor with shape: `(n_samples, n_features)`.
# Output shape
2D tensor with shape: `(n_samples, n_clusters)`.
"""

def __init__(self, n_clusters, weights=None, alpha=1.0, **kwargs):
if 'input_shape' not in kwargs and 'input_dim' in kwargs:
kwargs['input_shape'] = (kwargs.pop('input_dim'),)
super(ClusteringLayer, self).__init__(**kwargs)
self.n_clusters = n_clusters
self.alpha = alpha
self.initial_weights = weights
self.input_spec = InputSpec(ndim=2)
```

```
def build(self, input_shape):
assert len(input_shape) == 2
input_dim = input_shape[1]
self.input_spec = InputSpec(dtype=K.floatx(), shape=(None, input_dim))
self.clusters = self.add_weight((self.n_clusters, input_dim),
initializer='glorot_uniform', name='clusters')
if self.initial_weights is not None:
self.set_weights(self.initial_weights)
del self.initial_weights
self.built = True

def call(self, inputs, **kwargs):
""" student t-distribution, as same as used in t-SNE algorithm.
q_ij = 1/(1+dist(x_i, u_j)^2), then normalize it.
Arguments:
inputs: the variable containing data, shape=(n_samples, n_features)
Return:
q: student's t-distribution, or soft labels for each sample.
 shape=(n_samples, n_clusters)
"""
q = 1.0 / (1.0 + (K.sum(K.square(K.expand_dims(inputs, axis=1)
 - self.clusters),
 axis=2) / self.alpha))
q **= (self.alpha + 1.0) / 2.0
q = K.transpose(K.transpose(q) / K.sum(q, axis=1))
return q

def compute_output_shape(self, input_shape):
assert input_shape and len(input_shape) == 2
```

```
return input_shape[0], self.n_clusters

def get_config(self):
config = {'n_clusters': self.n_clusters}
base_config = super(ClusteringLayer, self).get_config()
return dict(list(base_config.items()) + list(config.items()))


class DCEC(object):
def __init__(self,
input_shape,
filters=[32, 64, 128, 10],
n_clusters=10,
alpha=1.0):

super(DCEC, self).__init__()

self.n_clusters = n_clusters
self.input_shape = input_shape
self.alpha = alpha
self.pretrained = False
self.y_pred = []

self.cae = CAE(input_shape, filters)
hidden = self.cae.get_layer(name='embedding').output
self.encoder = Model(inputs=self.cae.input, outputs=hidden)

# Definition du modèle DCEC
clustering_layer = ClusteringLayer(self.n_clusters,
 name='clustering')(hidden)
```

```
self.model = Model(inputs=self.cae.input,
outputs=[clustering_layer, self.cae.output])

def pretrain(self, x, batch_size=300, epochs=200,
optimizer='adam', save_dir='results/temp'):
print('...Pretraining...')
self.cae.compile(optimizer=optimizer, loss='mse')
from keras.callbacks import CSVLogger
csv_logger = CSVLogger(args.save_dir + '/pretrain_log.csv')

# Enrainement du modèle
t0 = time()
self.cae.fit(x, x, batch_size=batch_size, epochs=epochs,
 callbacks=[csv_logger])
print('Pretraining time: ', time() - t0)
self.cae.save(save_dir + '/pretrain_cae_model.h5')
print('Pretrained weights are saved to
%s/pretrain_cae_model.h5' % save_dir)
self.pretrained = True

def load_weights(self, weights_path):
self.model.load_weights(weights_path)

def extract_feature(self, x):  # extract features
 from before clustering layer
return self.encoder.predict(x)

def predict(self, x):
q, _ = self.model.predict(x, verbose=0)
return q.argmax(1)
```

```
@staticmethod
def target_distribution(q):
weight = q ** 2 / q.sum(0)
return (weight.T / weight.sum(1)).T

def compile(self, loss=['kld', 'mse'],
 loss_weights=[1, 1], optimizer='adam'):
self.model.compile(loss=loss,
 loss_weights=loss_weights, optimizer=optimizer)

def fit(self, x, y=None, batch_size=300, maxiter=2e4, tol=1e-2,
update_interval=140, cae_weights=None, save_dir='./results/temp'):

print('Update interval', update_interval)
save_interval = x.shape[0] / batch_size * 5
print('Save interval', save_interval)

# ETAPE 1: Préentrainement si nécessaire
t0 = time()
if not self.pretrained and cae_weights is None:
print('...pretraining CAE using default hyper-parameters:')
print('   optimizer=\'adam\';   epochs=200')
self.pretrain(x, batch_size, save_dir=save_dir)
self.pretrained = True
elif cae_weights is not None:
self.cae.load_weights(cae_weights)
print('cae_weights is loaded successfully.')

# ETAPE 2: initialisation des classes avec la méthodes K-means
```

```
t1 = time()
print('Initializing cluster centers with k-means.')
kmeans = KMeans(n_clusters=self.n_clusters, n_init=20)
self.y_pred = kmeans.fit_predict(self.encoder.predict(x))
y_pred_last = np.copy(self.y_pred)
self.model.get_layer(name='clustering').set_weights
([kmeans.cluster_centers_])

# ETAPE 3: Classification profonde
# logging file
import csv, os
if not os.path.exists(save_dir):
os.makedirs(save_dir)
logfile = open(save_dir + '/dcec_log.csv', 'w')
logwriter = csv.DictWriter(logfile,
fieldnames=['iter', 'acc', 'nmi', 'ari', 'L', 'Lc', 'Lr'])
logwriter.writeheader()

t2 = time()
loss = [0, 0, 0]
index = 0
for ite in range(int(maxiter)):
if ite % update_interval == 0:
q, _ = self.model.predict(x, verbose=0)
p = self.target_distribution(q)  # update the auxiliary
target distribution p

# Critères de performance
self.y_pred = q.argmax(1)
if y is not None:
```

```
acc = np.round(metrics.acc(y, self.y_pred), 5)
nmi = np.round(metrics.nmi(y, self.y_pred), 5)
ari = np.round(metrics.ari(y, self.y_pred), 5)
loss = np.round(loss, 5)
logdict = dict(iter=ite, acc=acc, nmi=nmi, ari=ari,
 L=loss[0], Lc=loss[1], Lr=loss[2])
logwriter.writerow(logdict)
print('Iter', ite, ': Acc', acc, ', nmi', nmi, ', ari',
ari, '; loss=', loss)


delta_label = np.sum(self.y_pred !=
 y_pred_last).astype(np.float32) / self.y_pred.shape[0]
y_pred_last = np.copy(self.y_pred)
if ite > 0 and delta_label < tol:
print('delta_label ', delta_label, '< tol ', tol)
print('Reached tolerance threshold. Stopping training.')
logfile.close()
break


if (index + 1) * batch_size > x.shape[0]:
loss = self.model.train_on_batch(x=x[index * batch_size::],
y=[p[index * batch_size::], x[index * batch_size::]])
index = 0
else:
loss = self.model.train_on_batch(x=x[index * batch_size:
(index + 1) * batch_size],
y=[p[index * batch_size:(index + 1) * batch_size],
x[index * batch_size:(index + 1) * batch_size]])
```

```
index += 1

# Enregistrement du modèle dans le répertoire de travail

if ite % save_interval == 0:
# save DCEC model checkpoints
print('saving model to:', save_dir + '/dcec_model_' + str(ite) + '.h5')
self.model.save_weights(save_dir + '/dcec_model_' + str(ite) + '.h5')

ite += 1



logfile.close()
print('saving model to:', save_dir + '/dcec_model_final.h5')
self.model.save_weights(save_dir + '/dcec_model_final.h5')
t3 = time()
print('Pretrain time:   ', t1 - t0)
print('Clustering time:', t3 - t1)
print('Total time:      ', t3 - t0)



if __name__ == "__main__":
# Définition des hyperparamètres
import argparse
parser = argparse.ArgumentParser(description='train')
parser.add_argument('--n_clusters', default=10, type=int)
parser.add_argument('--batch_size', default=300, type=int)
parser.add_argument('--maxiter', default=2e4, type=int)
parser.add_argument('--gamma', default=0.1, type=float,
help='coefficient of clustering loss')
```

```
parser.add_argument('--update_interval', default=140, type=int)
parser.add_argument('--tol', default=0.01, type=float)
parser.add_argument('--cae_weights', default=None,
help='This argument must be given')
parser.add_argument('--save_dir', default='results/temp')
args = parser.parse_args()
print(args)

import os
if not os.path.exists(args.save_dir):
os.makedirs(args.save_dir)

# Importation de la base MNIST

from datasets import load_mnist
if args.dataset == 'mnist':
x, y = load_mnist()

# Préparation du modèle DCEC

dcec = DCEC(input_shape=x.shape[1:], filters=[32, 64, 128, 10],
 n_clusters=args.n_clusters)
plot_model(dcec.model, to_file=args.save_dir + '/dcec_model.png',
 show_shapes=True)
dcec.model.summary()

# Classification

optimizer = 'adam'
dcec.compile(loss=['kld', 'mse'], loss_weights=[args.gamma, 1],
```

```
 optimizer=optimizer)
dcec.fit(x, y=y, tol=args.tol, maxiter=args.maxiter,
update_interval=args.update_interval,
save_dir=args.save_dir,
cae_weights=args.cae_weights)
y_pred = dcec.y_pred
print('acc = %.4f, nmi = %.4f, ari = %.4f' % (metrics.acc(y, y_pred),
metrics.nmi(y, y_pred), metrics.ari(y, y_pred)))


#Affichage du nuage de points dans un espace de dimension 2

import matplotlib.pyplot as plt

from sklearn.manifold import TSNE
tSNE = TSNE(n_components=2)
tSNE_result=tSNE.fit_transform(features)



x=tSNE_result[:,0]
y=tSNE_result[:,1]

#colors = 'b' ,'r', 'g', 'c', 'm', 'y', 'k', 'w', 'o', 'p'
marker_size=0.1

plt.scatter(tSNE_result[:,0], tSNE_result[:,1], colors, marker_size)
plt.show()
```