Rapport de projet
Différentes approches du Machine Learning sur la base MNIST

Gabin Durteste, Tristan Lorriaux

23 janvier 2021

Résumé

Le rapport ci-dessous décrit notre évolution dans l'apprentissage en Machine Learning à l'école des Mines de Saint-Etienne, cursus ISMIN, en deuxième année.
Le but de ce rapport de travaux pratiques est de mettre en exergue les connaissances vues en cours. Pour cela nous travaillerons tout au long de la réalisation de ce rapport sur la base de données "Mixed National Institute of Standards and Technology", MNIST, qui est composée de milliers de chiffres écrits à la main. C'est un jeu de données très utilisé en Machine Learning notamment.
Le rapport suivra le même chemin logique que le sujet du TP.

Table des matières

Table des figures

1 Chargement et préprocessing sur la base de données MNIST

Pour les algorithmes, la reconnaissance de dessins manuscrits, et donc de toutes formes d'écriture, est un bon exercice. La base de données MNIST comporte 70000 images de taille 28 par 28, normalisées, centrées et labelisées, en noir et blanc. Nous allons, au cours de ce rapport, appliquer divers algorithmes sur cette base de données : supervisés, non supervisés, Deep Learning, etc. Mais avant d'appliquer ces algorithmes, il nous faut traiter les données en entrée : le produit de ce traitement de données, ce "préprocessing", sera injecté en entrée de tous les algorithmes par la suite (excepté pour le DL où la taille des tenseurs d'entrée sera changée). On peut d'abord afficher quelques images de la base de données de manière brute sans reshaping.

```
X = np.load('../MNIST_X_28x28.npy')
Y = np.load('../MNIST_y.npy')
for i in range (100) :
    nb_sample = i
    plt.imshow(X[nb_sample])
    img_title = 'Classe' + str(Y[nb_sample])
    plt.title(img_title)
    plt.show()
    plt.clf
```

On peut observer la variance et la moyenne sur les vérités terrains pour observer la base MNIST, et on obtient avec le code suivant

```
moy = np.average(Y)
var = np.std(Y)
print("Moyenne : ", moy)
print("Variance : ", var)
```

un écart-type de

$$\sigma = 2.89 \tag{1}$$

et une moyenne de

$$\mathrm{M} = 4.45 \tag{2}$$

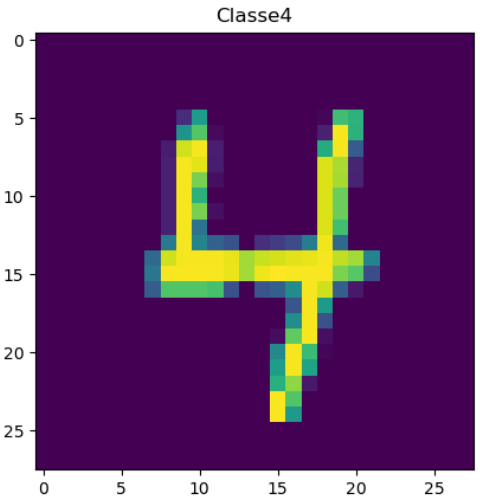

FIGURE 1 – Un nombre manuscrit issu de la base de données MNIST

Dans le cadre des divers problèmes d'apprentissage que nous allons imposer à cette base de donnée, nous allons devoir la séparer entre un jeu de données d'entraînement ("train") et un jeu de données de validation ("test"). Nous utilisons un module de sklearn pour réaliser cette opération.

```
import numpy as np
from sklearn.model_selection import train_test_split
X = np.load('../MNIST_X_28x28.npy')
Y = np.load('../MNIST_y.npy')
X_train, X_test, Y_train, Y_test =
train_test_split(X, Y, test_size=0.2, random_state=42)
```

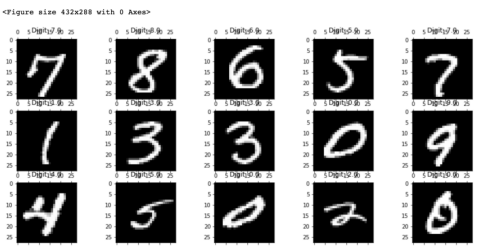

FIGURE 2 – Extrait de la base MNIST (issu d'un autre de nos codes)

2 Apprentissage non supervisé

2.1 Réduction de dimension

Pour la PCA ("Principal Component Analysis"), nous redimensionnerons les données, de 28x28 à seulement 7000x784 au lieu de 70000x28x28, puis on applique le "split" classique entre le jeu d'entraînement et le jeu de validation.

```
# PreProcessing des données
X = np.load('../MNIST_X_28x28.npy')
Y = np.load('../MNIST_y.npy')
Xr= X.reshape(70000,784)/255.0 #On reshape les données
x_train,x_test,y_train,y_test=
train_test_split(Xr,Y,train_size=0.1) #On split
```

Le but de cette section est de réduire la dimension de nos données d'entrée en appliquant une méthode dite d'analyse par composantes principales. Le but est donc de réduire le nombre de dimensions des données d'entrées, vers par exemple un espace de dimension 2 pour faciliter la visualisation de ces dernières.
Par exemple, en notant x une donnée d'entrée, elle appartient à l'espace :

$$\mathrm{x} \in \mathbb{R}^{784} \tag{3}$$

On obtient après la projection sur les variances principales :

$$\mathrm{x} \in \mathbb{R}^{2} \tag{4}$$

On choisit donc tout logiquement de projeter sur un espace plan le jeu de données de "test", puis on utilise le module SeaBorn pour afficher la PCA.

```
n_components = 2 #pour la PCA
pca = PCA(n_components=n_components)
reduced_data = pca.fit_transform(x_train) #Résultat de la PCA
```

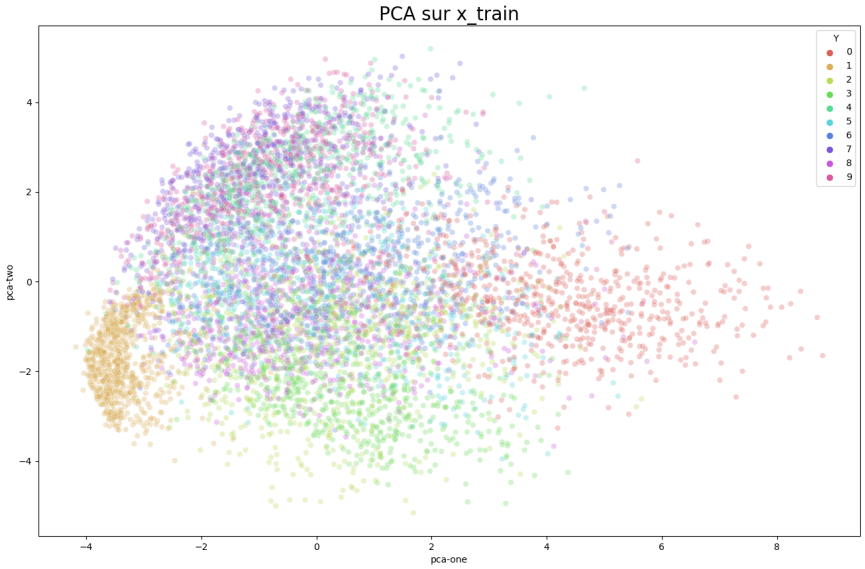

FIGURE 3 – Résultat d'une PCA vers un espace de deux dimensions

On observe, lorsqu'on regarde les métriques de cette PCA, que la variance explicite par composante principale est [0.09616645, 0.0706938] : c'est le pourcentage de la variance exprimée par chacune des composantes sélectionnées. Avec 2 composantes, la variance explicite est donc d'environ 0.167. Ceci étant dit, la corrélation des données semble être encore importante. Au moment de choisir le nombre de composantes principales (k), il faut bien sûr choisir k le plus petit possible (intérêt de la réduction dimensionnelle) pour qu'un maximum, en l'occurrence ici dix-sept pour cent de la variance cumulée, soit conservé.

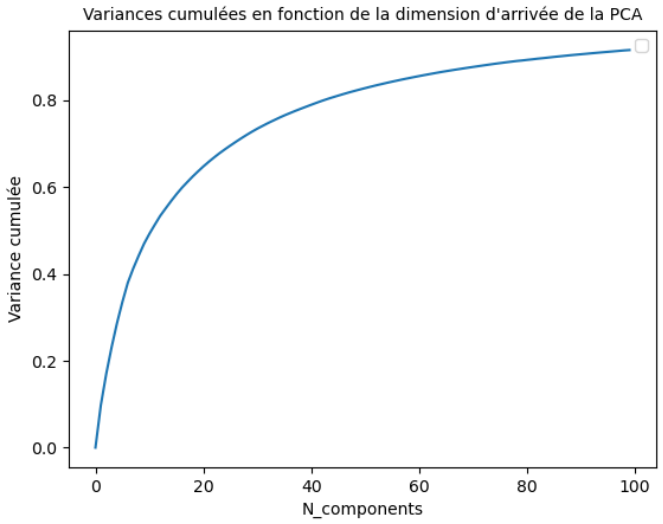

FIGURE 4 – Issu d'un autre code, disponnible sur le Git

Après plusieurs tests peu ou pas plus concluants, notamment une projection graphique en dimension 3, nous pouvons conclure que nous resterons sur une PCA vers un espace de dimension 2, qui nous offre une dé-corrélation correcte des données sans être, soulignons le, exceptionnelle.

2.2 Partitionnement sur le jeu de données

Maintenant qu'on a réalisé cette réduction dimensionnelle du jeu de données, on peu espérer réaliser un partitionnement sur ledit jeu de données, le "clusteriser", via notamment deux méthodes : un K-Mean ainsi qu'un mélange Gaussien.

2.2.1 La méthode du K-Mean

Après le classique pré-traitement des données, mentionné précédemment, nous allons donc d'abord appliquer la méthode dite du K-Mean, en utilisant les données du jeu de "test", projetées en dimension 2.

Il s'agit en premier lieu de choisir un K, le nombre de clusters, optimal. Pour cela, nous utiliserons une méthode graphique populaire pour déterminer le nombre de clusters : la méthode du coude. Nous allons donc opérer une boucle de *MiniBatchKMeans*(un K-Mean plus léger pour nos processeurs à tous !), et un

```
kmeans.inertia_
```

pour afficher la distorsion en fonction du nombre de clusters. Le coude du graphe nous indiquera dès lors le nombre de clusters optimal. Notons que la fonction de distorsion représente la somme des erreurs quadratiques (SEQ), c'est à dire basiquement la somme des distances pour chaque point par rapport à son centre :

$$\sum (donneX_1 - Centrode)^2 + (donneX_2 - Centrode)^2 ... \tag{5}$$

La méthode se résume donc ainsi : le point du coude est celui du nombre de clusters à partir duquel la variance ne se réduit plus significativement.
Donc si on applique cette méthode à notre jeu de données :

```
disto = list()
for k in range(1, 15):
    kmeans = MiniBatchKMeans(n_clusters=k)
    kmeans = kmeans.fit(x_train)
    disto.append(kmeans.inertia_)

plt.figure(figsize=(15, 6))
plt.plot(range(1, 15), disto, marker="x")
plt.show()
```

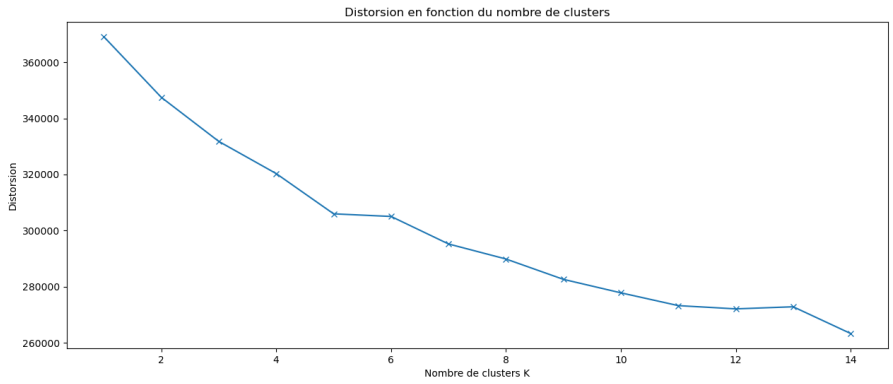

FIGURE 5 – Méthode graphique du "coude"

Nous prendrons donc un nombre de clusters de K=10 (lecture graphique). Nous pouvons appliquer un K-Means, puis tracer les marges, les centroïdes etc. et nous obtenons finalement un clustering correct mais imparfait.

```
#PCA
pca = PCA(n_components=n_components)
pca_result = pca.fit_transform(x_train)#Résultat de la PCA

#KMeans
kmeans = KMeans(init ="k-means++",n_clusters = 10)
kmeans = kmeans.fit(pca_result)
```

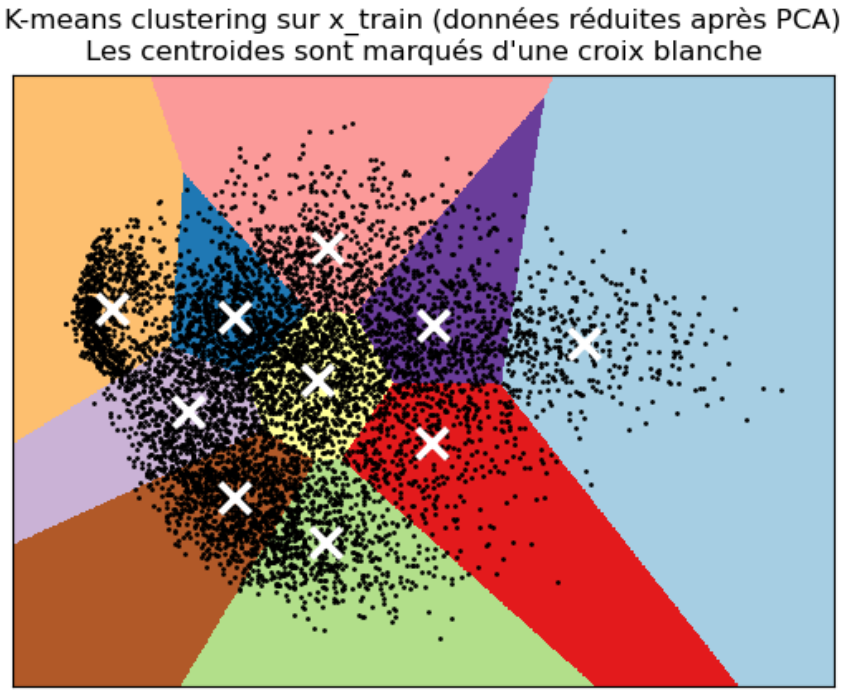

FIGURE 6 – Résultat du partitionnement par K-Means avec K = 10 centroïdes

Notons qu'on initialise avec *k-means++* plutôt qu'aléatoirement pour de meilleurs résultats.

On observe un score d'homogénéité entre la prédiction et la vérité terrain de environ 0.36 : on est bien loin d'un score de 1 malgré tous nos efforts (en changeant les clusters, cela bouge peu). Notons que ce score monte à environ 0.45 sans PCA : logique, puisqu'on perd des informations avec la réduction dimensionnelle, mais le calcul est grandement facilité. Le score a été obtenu avec

```
sklearn.metrics.homogeneity_score()
```

On va donc observer le contenu de nos clusters pour comprendre mieux ce score ! L'objectif de cette partie est de modéliser, via des diagrammes barres, les contenus de nos dix clusters. Pour cela, on utilise un code qui basiquement consiste en un indexage et un comptage des labels dans les clusters, puis en l'affichage des résultats. Nous invitons le lecteur à le consulter sur le Git.

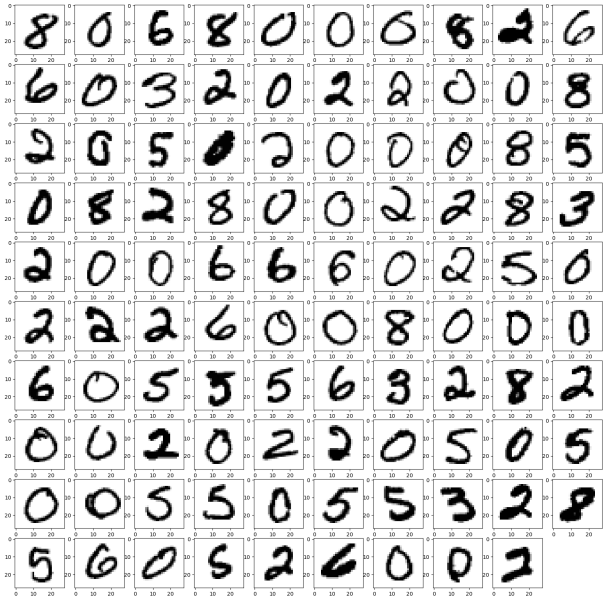

FIGURE 7 – "Echantillon" d'un cluster

Les résultats sont divers : nos clusters sont parfois riches de beaucoup de nombres, même si certains semble prouver que le K-Means est fructueux sur quelques chiffres. En abscisse, les différents labels (les 9 chiffres), en ordonnée, leur occurrence par cluster.

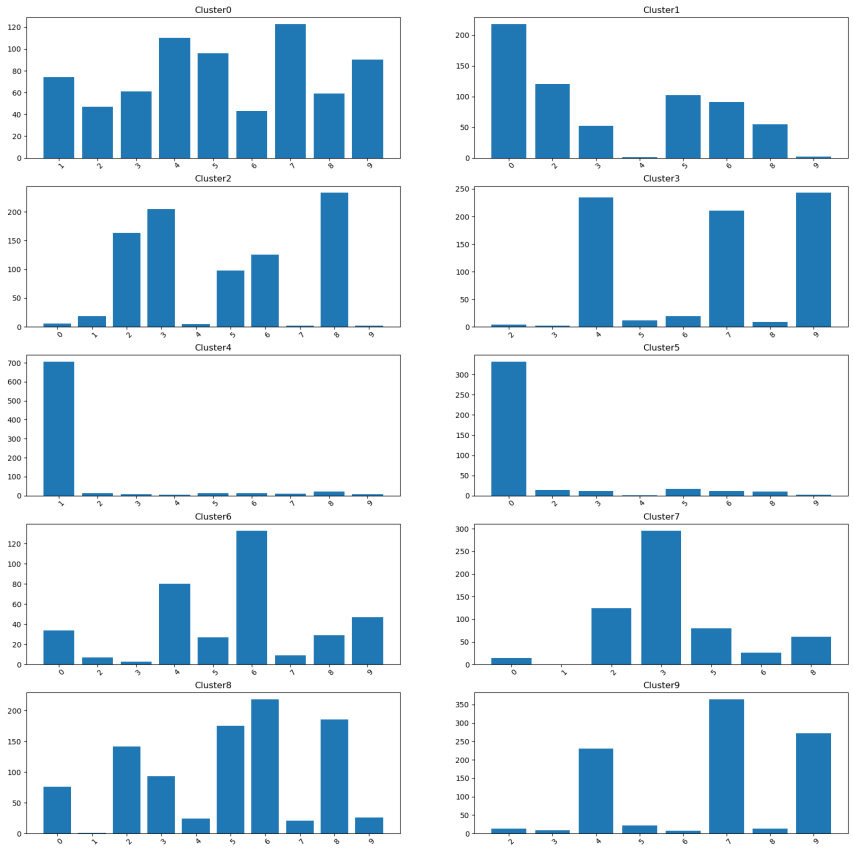

FIGURE 8 – Contenu des clusters après un K-Mean avec K = 10. Les noms des clusters sont au dessus des graphes

On peut donc tester notre partitionnement sur notre jeu de test et on obtient donc finalement, avec les centroïdes obtenu de la construction du modèle sur le jeu d'entraînement :

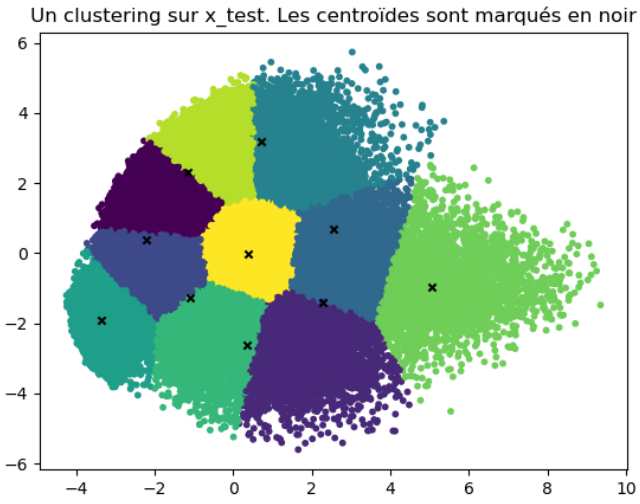

FIGURE 9 – Partitionnement sur le jeu de test

2.2.2 La méthode du mélange Gaussien

Passons maintenant à une autre méthode : celle du mélange Gaussien, et voyons si le partitionnement est meilleur. Nous utiliserons toujours notre espace de dimension 2 comme entrée ici, ainsi qu'un nombre K de 10 clusters. Nous ne reviendrons pas sur le fonctionnement de la mixture gaussienne, parceque assez complexe. Il faut seulement souligner qu'on va utiliser une algorithme itératif dit "EM", comprenant deux phases : une étape d'évaluation de l'espérance (E), où l'on calcule l'espérance de la (log)vraisemblance en tenant compte des dernières variables observées, une étape de maximisation (M), où l'on estime le maximum de vraisemblance des paramètres en maximisant la vraisemblance trouvée à l'étape E.
L'algorithme se résume en python par un simple module de sklearn :

```
    #PCA
    pca = PCA(n_components=n_components)
    pca_result = pca.fit_transform(x_train)#Résultat de la PCA
    pca_test = pca.fit_transform(x_test)
    #Algo Expectation-Maximization
    GM = GaussianMixture(n_components=k,covariance_type='full')
    GM.fit(pca_result)
    #Métriques
    print(GM.means_)
```

```
print(GM.covariances_)
print("La probabilité de mélange gaussien est de {}"
.format(math.exp(GM.score(pca_test))))
```

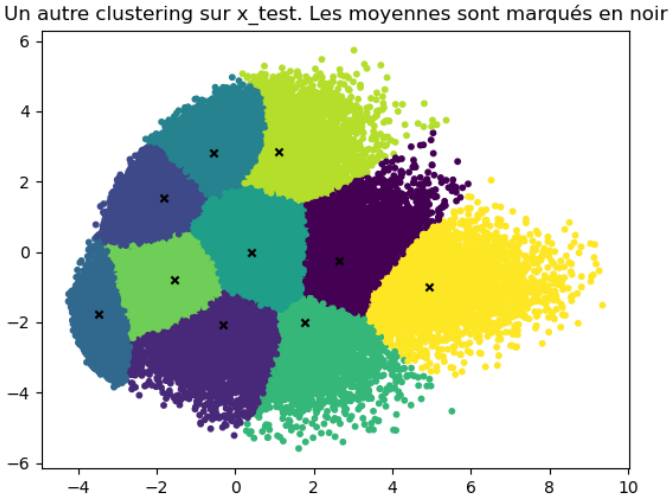

FIGURE 10 – Partitionnement sur le jeu de test en utilisant la méthode de la mixture gaussienne

La probabilité de mélange Gaussien est d'environ $1.7 * 10^{-2}$. Sans réduction dimensionnelle, la probabilité de mélange Gaussien devient dès lors $1.44 * 10^{-105}$. Nous ne nous attarderons pas non plus sur ces derniers chiffres, qui sont des métriques plus complexes que la précision. En effet, cela ne représente en rien la véritable répartition de nos données : le pourcentage de probabilité calculé de nos échantillons n'est pas distribué uniformément. Mais, excepté la matrice de confusion, cette métrique reste la meilleure que l'on a pu observer.
Passons donc maintenant aux matrices de confusion

2.2.3 Matrices de confusion

Comme les métriques l'avaient dit auparavant : les résultats ne sont pas incroyablement probants. Nous pouvons considérer dans une moindre mesure qu'ils sont satisfaisants. Notons qu'on utilise pour ces matrices deux modules :

```
from sklearn.metrics import confusion_matrix
import seaborn as sns
```

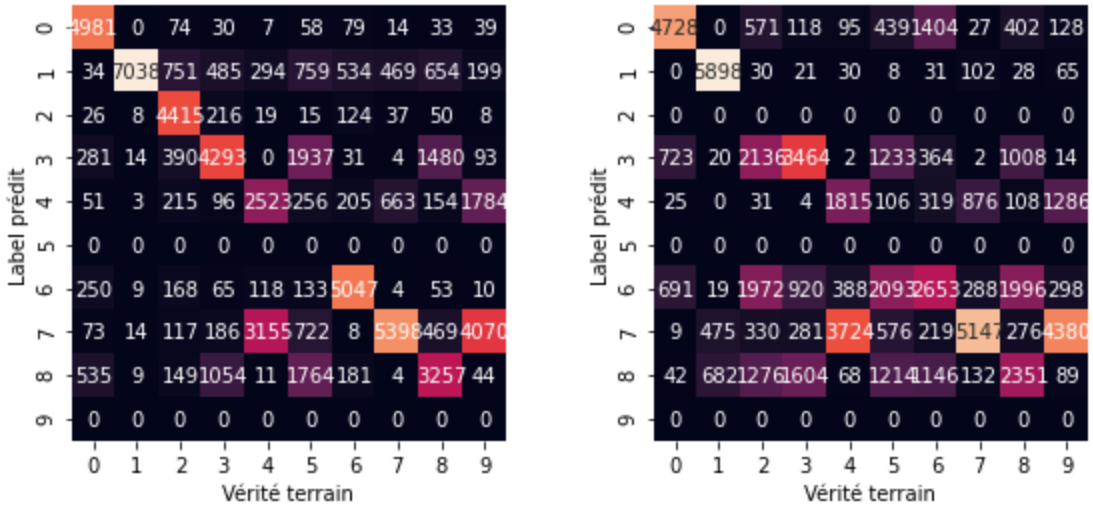

FIGURE 11 – Matrice de confusion. A gauche, celle du K-Means, à droite, celle du mélange Gaussien

Ici une difficulté a été de faire correspondre les labels au jeu de données. Pour cela, on postule que si l'on compte le nombre de prédictions label par label, le label le plus souvent prédit au sein d'un cluster correspond bien au label exact de celui-ci. Pour modéliser cela, on utilise la fonction mode, qui renvoie comme son nom l'indique le mode d'une série de données (le mode étant en mathématiques la valeur statistiquement la plus représentée dans une population donnée) :

```
from scipy.stats import mode
```

C'est cette méthode qui fait ressortir des rangées de zéro au sein des deux matrices de confusions. La seule hypothèse que nous pouvons émettre dès lors est que notre postulat est faux pour certains clusters dans les deux méthodes, ce qui impliquerait donc une qualité médiocre des prédictions.

3 Apprentissage supervisé

Nous allons dans cette partie réaliser un apprentissage supervisé sur notre jeu de données, en utilisant plusieurs méthodes connues (Support Vector Machine, Naive Bayes Classifier, Decision Tree etc.). Notons que, si auparavant le *test train split* n'était pas essentiel, il devient obligatoire ici !

3.1 Support Vector Machine

3.1.1 Sans réduction dimensionnelle

En premier lieu, nous avons décidé de tester cette méthode. Il convient de mentionner que nous testons la méthode sur des données en premier lieu dimensionnellement non réduites, c'est à dire avec le pré traitement classique sans PCA : nous la testerons ensuite sur des données réduites. Notre premier constat : le temps d'exécution du code est d'environ 15 minutes sans réduction dimensionnelle. Ceci dit le système semble converger dès le premier essai : nul besoin de changer le kernel ici. Il est par défaut non linéaire, en "radial basis function". Notons qu'en testant avec des kernels linéaires, ou la fonction :

```
svm.LinearSVC
```

le système ne semble pas converger. Nous resterons donc par défaut sur l'implémentation non linéaire RBF. Le système convergeant (au bout d'un temps qui nous semble infini), on peut donc étudier ses métriques. On atteint des taux de succès avec cette solution proche de 94%.

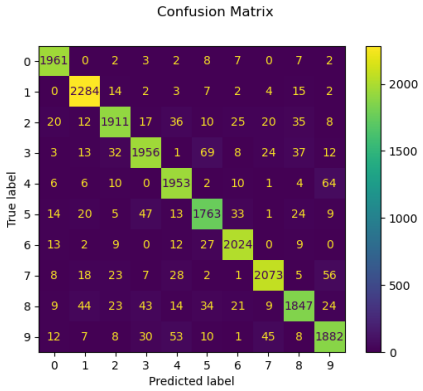

FIGURE 12 – Matrice de confusion de la méthode SVM sans réduction dimensionnelle

3.1.2 Avec une réduction dimensionnelle

On opère donc une PCA sur le jeu de données de test durant son pré traitement. On observe une accélération considérable du temps de compilation du code, mais une légère baisse des métriques : le taux de succès de la solution descend à 90% pour une réduction dimensionnelle à 18 dimensions, et chute à environ 40% pour une réduction dimensionnelle d'ordre 2. Il faut donc établir un compromis entre vitesse d'exécution et résultats de la méthode lorsque l'on travaille avec la réduction dimensionnelle

```
    pca = PCA(n_components=n_components) #PCA
    reduced_data = pca.fit_transform(Xr)
    x_train,x_test,y_train,y_test=
    train_test_split(reduced_data,Y,train_size=0.7,shuffle=False)
    print("Préprocessing terminé")
    clf = svm.SVC(gamma=0.001) #SVM
    clf.fit(x_train, y_train) #Apprentissage
    print("Apprentissage terminé")
    predicted = clf.predict(x_test) #Prédiction
    print("Prédiction terminée")
```

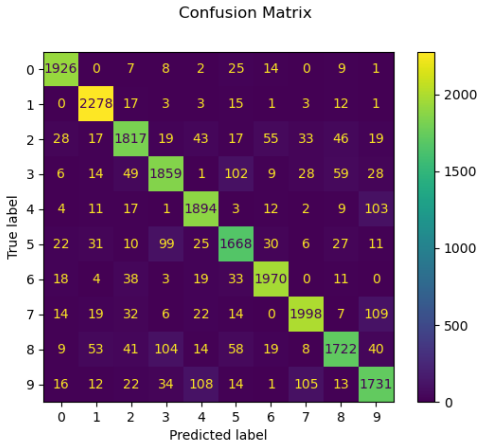

FIGURE 13 – Matrice de confusion de la méthode SVM avec réduction dimensionnelle vers un espace de dimension 18

Nous pouvons donc conclure sur le fait que la méthode dite du "Support Vector Machine" est très efficace pour notre situation, notamment avec un noyau type RBF et si tant est qu'on réalise une réduction dimensionnelle raisonnable.

3.2 Naïve Bayes Classifier

3.2.1 Sans réduction dimensionnelle

Maintenant testons la méthode Naïve Bayesienne. On part d'un postulat fort sur notre jeu de données en entrée, à savoir que les variables d'entrée sont toutes indépendantes. Nous ne pouvons que souligner la haute corrélation des données, que nous avons montrée auparavant : à première vue, peu de chance que cette méthode soit réellement efficace. On l'implémente, comme d'habitude, via sklearn :

```
from sklearn.naive_bayes import GaussianNB
clf = GaussianNB() #NBC
clf.fit(x_train, y_train) # Apprentissage
print("Apprentissage terminé")
predicted = clf.predict(x_test) # Prédiction
print("Prédiction terminée")
```

La méthode fournit, comme nous l'attendions, un résultat peu concluant. En imprimant ses métriques et sa matrice de confusion, on se rend compte que la haute corrélation des données a impacté le résultat. Le score final est de 58% : peu concluant donc.

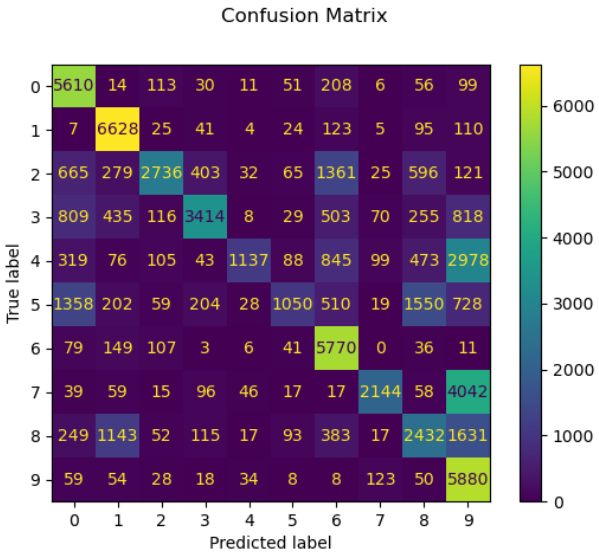

FIGURE 14 – Matrice de confusion de la méthode NBC sans réduction dimensionnelle

3.2.2 Avec réduction dimensionnelle

Avec une réduction dimensionnelle à 0.9 composantes, on réduit grandement la corrélation des données : dès lors, la méthode NBC a plus de sens et on observe une nette amélioration des métriques. Le score final augmente grandement et les temps d'exécution aussi : environ 87%. On notera que pour des PCA à maximum 2-3 composantes, les résultats sont probants. Ceci étant dit, une NBC sur un jeu de données moins réduit (nous avons testé avec 18 composantes) reste très peu concluant, voire aggrave le cas de la NBC.

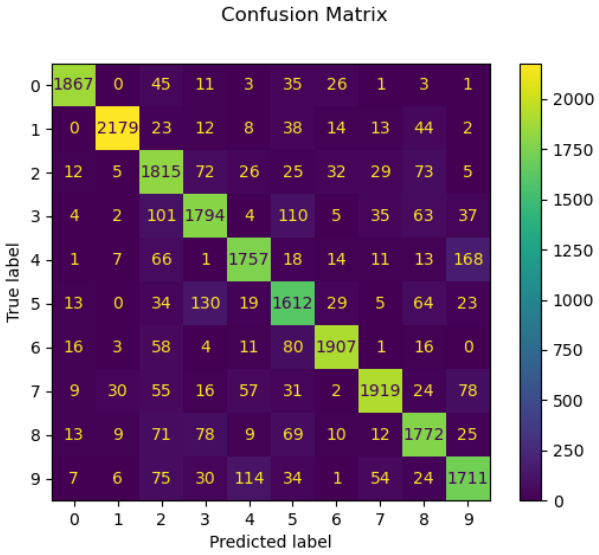

FIGURE 15 – Matrice de confusion de la méthode NBC avec réduction dimensionnelle. Ici K = 0.9

3.3 Decision Tree

On décide finalement de tester une méthode basée sur un arbre de décision.

```
from sklearn.tree import DecisionTreeClassifier
...
clf = DecisionTreeClassifier()
clf.fit(x_train, y_train)# Apprentissage
print("Apprentissage terminé")
predicted = clf.predict(x_test)# Prédiction
print("Prédiction terminée")
```

Observons rapidement ses métriques : nous obtenons un court temps de calcul et un score de 79% sur la jeu de données de test. Le temps de calcul est le meilleur des trois méthodes : ceci dit les taux d'erreurs sont plus élevés sur ce type de problème.

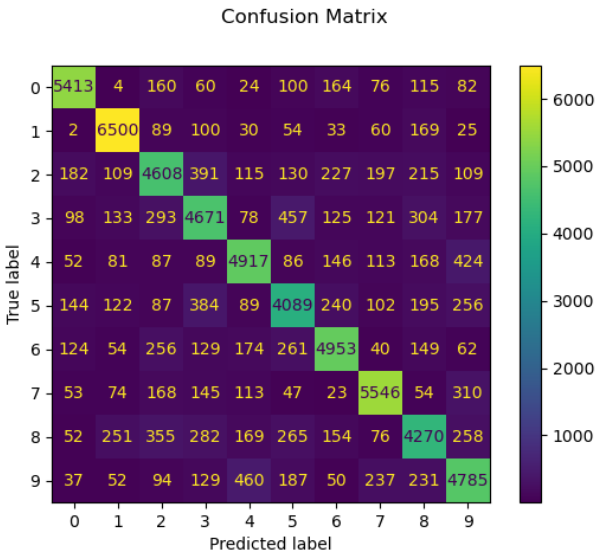

FIGURE 16 – Matrice de confusion de la méthode NBC avec réduction dimensionnelle. Ici K = 0.9

4 Deep Learning

Dans cette partie, nous allons tester et modifier les paramètres d'un apprentissage qui sera basé sur deux types de réseaux de neurones. Nous nous appuyons toujours sur les jeux de donnés issus de la base MNIST, ainsi qu'au prétraitement classique de ces données.

4.1 Multi-Layer Perceptron

En premier lieu, nous allons tester un premier modèle de réseau neuronal : le perceptron multicouche. C'est un type de réseau neuronal artificiel organisé en plusieurs couches. Ainsi dans ces couches, l'information circule de l'entrée, à la sortie, chaque couche étant constituée d'un nombre variable de neurones. On choisira 10 sorties pour 10 chiffres ! On a donc une taille du vecteur d'entrée est de 784, et une sortie de taille 10. On utilise un nombre d'epochs raisonnable (15) pour détecter un éventuel overfitting, et limiter le temps d'exécution (ce sera d'autant plus vrai par la suite pour les réseaux de neurones convolutionnels). Notons que l'epoch représente le nombre d'entraînements successifs de notre modèle sur le jeu de données d'entraînement. On fixe un batch à 128 : il s'agit de la taille de l'échantillon des entrées à traiter à chaque étape de l'entraînement.

```
# Paramètres
number_features=784
batch_size = 128
hidden_units = 256
dropout = 0.45
epochs = 20
```

On notera qu'on active à chaque étape avec ReLu, le tout combiné avec un étage de dropout (ce qui nous permet d'éviter un overfitting au coût d'une précision légèrement amoindrie). Le modèle détaillé du MLP est disponible en annexe.

On obtient donc le modèle suivant avec 269 322 paramètres entraînables :

Layer (type)	**Output Shape**	**Parameters**
dense (Dense)	(None, 256)	200960
activation (ReLu)	(None, 256)	0
dropout (Dropout)	(None, 256)	0
dense1 (Dense)	(None, 256)	65792
activation1 (Relu)	(None, 256)	0
dropout1 (Dropout)	(None, 256)	0
dense2 (Dense)	(None, 10)	2570
activation2 (Softmax)	(None, 10)	0

Observons désormais les métriques de notre modèle de MLP. Le programme a mis environ une quinzaine de secondes à tourner. Pour ce qui est de la précision, on atteint un excellent score de environ 97% de précision. On obtient donc ces deux graphes que nous allons commenter.

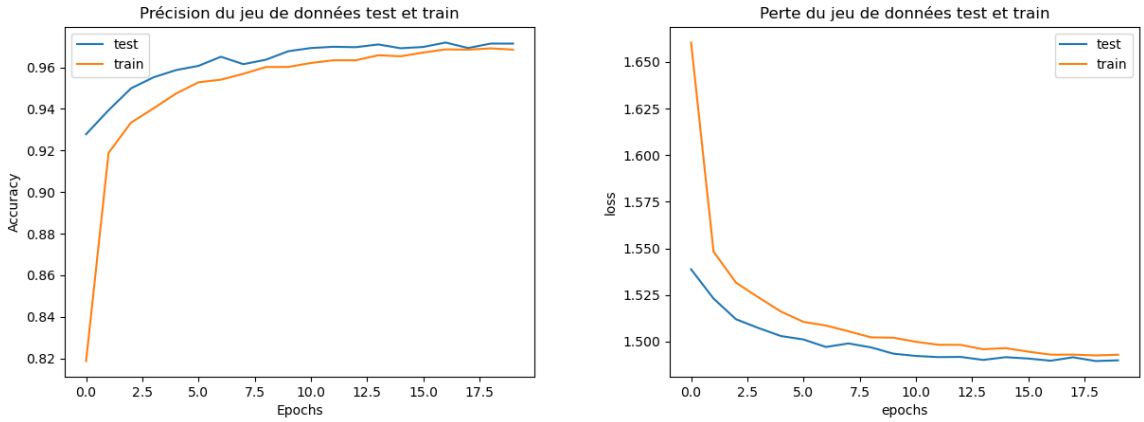

FIGURE 17 – Évolution de la précision et de la perte sur les jeu de données d'entraînement et de test

Grâce au "dropout", on n'observe pas de phénomène d'"overfitting" qui se traduirait par un palier infranchissable et la précision semble finalement peu affectée. Aussi, on n'observe à la fin du procédé aucun palier significatif entre le jeu de données d'entraînement et celui de test. On peut conclure en affirmant que ce réseau MLP semble être excellent en tout point, tant en terme de performance qu'en terme de temps sur notre système.

4.2 Convulotional Neural Network

Nous tenons à souligner de prime abord que dans le monde du deep learning, les réseaux de neurones convolutionnels sont particulièrement réputés dans des applications comme le traitement vidéo, et donc aussi le traitement d'images. Le principe d'un CNN repose sur un grand nombre de filtres, repérant les différentes structures d'une image. Ces filtres, de deux dimensions, permettent de prendre en compte les pixels de l'image en fonction de leur position les uns par rapport aux autres : l'entrée du modèle sera donc cette fois ci un tenseur de taille 28x28x1.

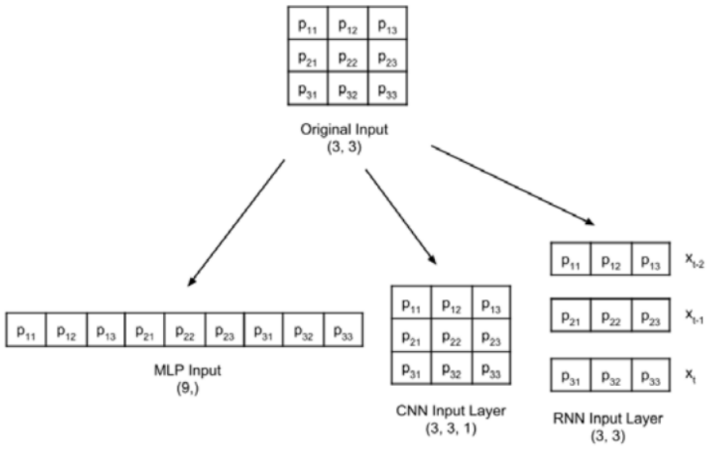

FIGURE 18 – A chaque réseau de neurones son entrée

On retrouve des dropout encore une fois pour éviter l' "overfitting". On active toujours via ReLu et Softmax en sortie. Ici, on réduit le nombre d'epochs à 10 pour deux raisons : la longueur du temps du procédé global, qui possède beaucoup plus de paramètres qu'auparavant (causé par les filtres), et pour éviter un "overfitting" (15 epochs ont tendance à causer un léger "overfitting" sur notre modèle). Notons qu'on aurait pu ici utiliser des couches dite de MaxPooling pour accélérer l'entraînement de notre modèle : en réalisant une sorte de "sous échantillonnage", trouvant des occurrences communes dans la matrice, et réduisant la taille donc le nombre de paramètres à calculer de cette matrice, mais nous ne souhaitions pas perdre de donnée pour atteindre des scores de précision maximaux.

En revanche, pour accélérer l'entraînement de notre modèle, on utilise des couches de "Batch Normalization", visant à résoudre plusieurs problèmes : éviter les gradients instables, réduire les effets de l'initialisation du réseau sur la convergence et donc permettre des taux d'apprentissage plus rapides conduisant à une convergence plus rapide.

On obtient donc le modèle suivant avec 327 242, dont 326 410 paramètres entraînables. On notera que la couche "dense" est la fonction d'activation finale : SoftMax. Sinon toutes les couches sont activées par ReLu :

Layer (type)	**Output Shape**	**Parameters**
conv2d (Conv2D)	(None, 26, 26, 32)	320
batchnormalization (BatchNorm)	(None, 26, 26, 32)	128
conv2d1 (Conv2D)	(None, 24, 24, 32)	9248
batchnormalization1 (BatchNorm)	(None, 24, 24, 32)	128
conv2d2 (Conv2D)	(None, 12, 12, 32)	25632
batchnormalization2 (BatchNorm)	(None, 12, 12, 32)	128
dropout (Dropout)	(None, 12, 12, 32)	0
conv2d3 (Conv2D)	(None, 10, 10, 64)	18496
batchnormalization3 (BatchNorm)	(None, 10, 10, 64)	256
conv2d4 (Conv2D)	(None, 8, 8, 64)	36928
batchnormalization4 (BatchNorm)	(None, 8, 8, 64)	256
conv2d5 (Conv2D)	(None, 4, 4, 64)	102464
batchnormalization5 (BatchNorm)	(None, 4, 4, 64)	256
dropout1 (Dropout)	(None, 4, 4, 64)	0
flatten (Flatten)	(None, 1024)	0
batchnormalization6 (BatchNorm)	(None, 128)	512
dropout2 (Dropout)	(None, 128)	0
dense1 (Dense)	(None, 10)	1290

Observons, pour finir, les métriques de ce modèle. On atteint un score impressionnant (mais prévisible de par la nature même des CNN issu du biomimétisme de la reconnaissance d'images des animaux) : 99.2%. Le temps d'exécution du programme est évalué à 15 minutes environ, mais les résultats sont probants.

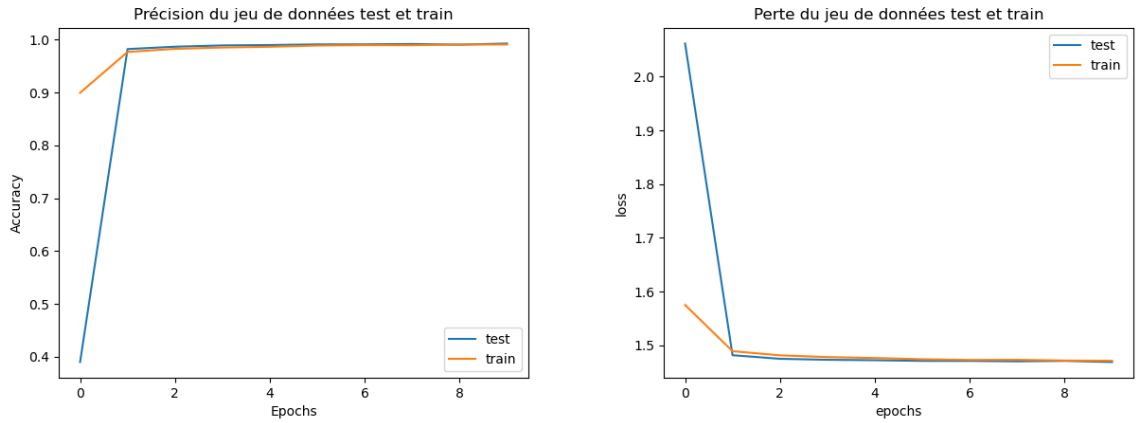

FIGURE 19 – Évolution de la précision et de la perte sur les jeu de données d'entraînement et de test

On voit que dès les premiers epochs, le modèle apprend vite. La perte devient minime, et l'"overfitting" est clairement absent. On peut finalement conclure que ce modèle semble parfaitement adapté à notre système.

5 Conclusion

Nous avons pu, tout au long de ces travaux, mettre à l'épreuve nos connaissances apprises durant nos cours de Machine Learning. Nous avons certainement apprécié le projet, qui permettait de balayer le panel de possibilités que nous offrait la matière, du clustering à l'apprentissage automatique.

Nous tenons évidemment à remercier M. P-A Moellic ainsi que M. R Bernhard pour leurs cours, leur bienveillance et leur détermination à trouver un format d'adaptation aux TP habituellement enseignés en présentiel, en ces temps troubles. Nous ne pouvions que conclure ainsi : longue vie au Machine Learning !

A Modèles de Deep Learning

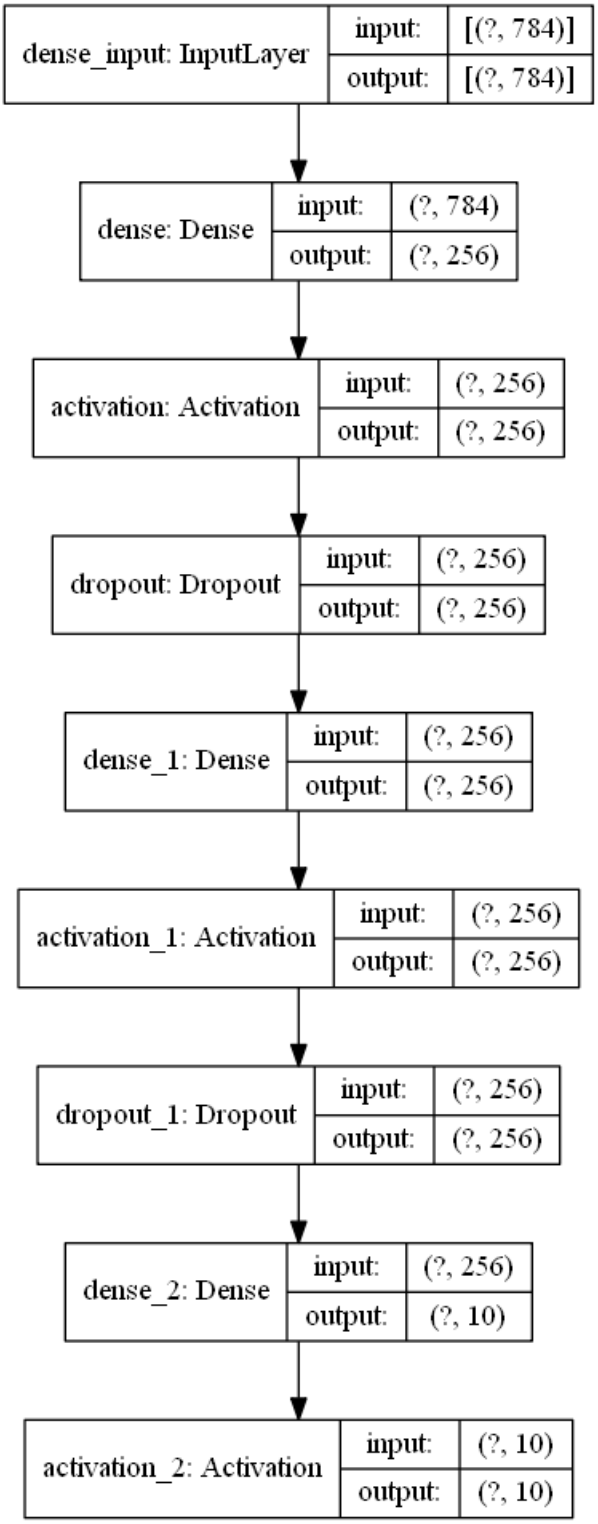

FIGURE 20 – Modèle de MLP

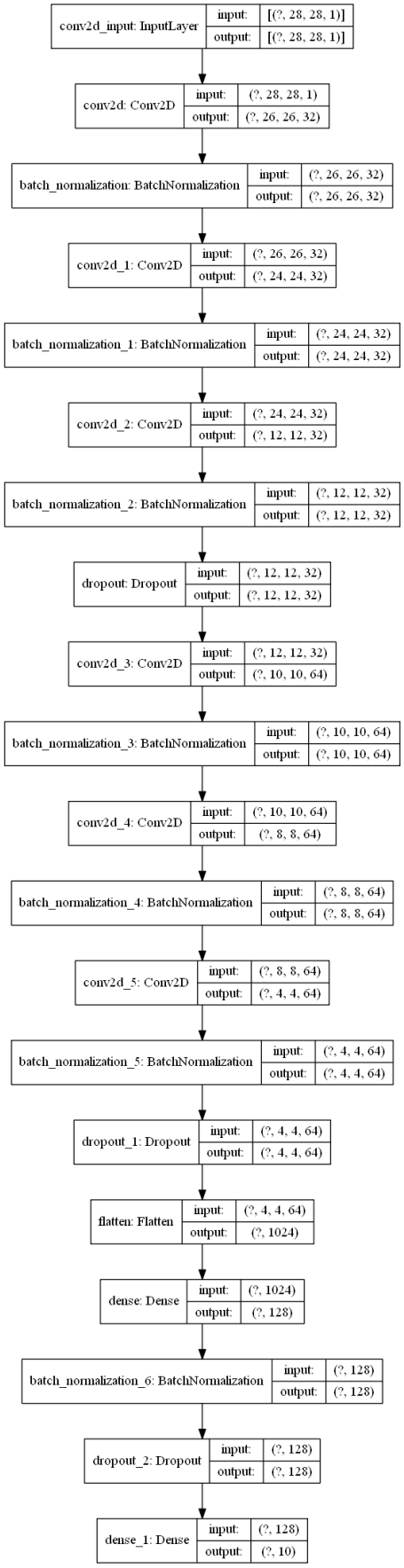

FIGURE 21 – Modèle de CNN

B Informations annexes

Vous pouvez retrouver le code utile à l'écriture de ce rapport ici.
Notez que nous avons utilisé plusieurs bibliothèques indispensables pour ce projet.

```
# Bibliothèques
import numpy as np
import matplotlib.pyplot as plt
import math
import sklearn
import tensorflow as tf
import seaborn as sns
import scipy
```

Ce document a été rédigé sous LaTeX, et compilé avec pdfLaTex.