Rapport : Machine Learning

Jules Soria - Arnaud Cotten

21 Janvier 2022

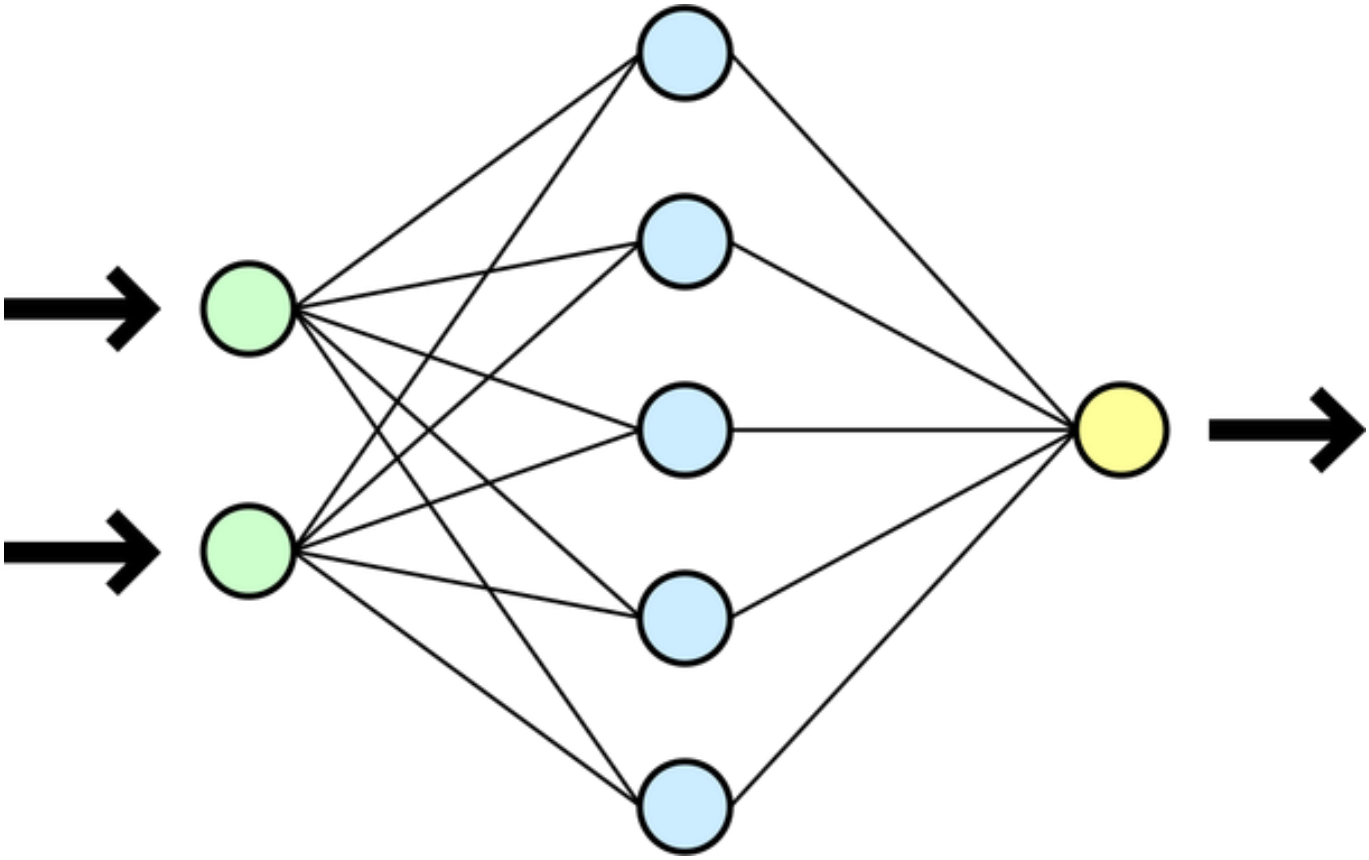

Table des matières

Table des figures

Introduction

Au cours de notre enseignement aux Mines Saint-Etienne, nous avons pu explorer le monde de l'apprentissage automatique (*Machine Learning*) et de l'apprentissage profond (*Deep Learning*) à travers divers exemples de modèles, problèmes, et résolutions de ces problèmes.

Dans ce rapport, nous allons résumer toutes nos pistes de réflexion lors de l'implémentation de modèles standards vus en cours. Le rapport sera rédigé en français mais nous avons fait le choix de garder les titres originaux en anglais, et nous nous permettrons des anglicismes lorsque le langage technique s'y prête. Nous avons tenté d'expliquer comment chaque modèle fonctionne du mieux que nous pouvions.

Durant tout ce travail pratique, nous avons utilisé le langage informatique Python, et nous avons manipulé les bibliothèques suivantes : Scikit-Learn et TensorFlow pour l'apprentissage automatique, Keras pour l'apprentissage profond.

L'objectif de ce compte-rendu est de mettre en évidence les diverses méthodes que l'on peut utiliser pour résoudre le problème de la reconnaissance des chiffres écrits à la main.

Le code que nous avons produit est un fichier Python disponible en annexe, avec les tableaux NumPy contenant les données $\mathbf{X}$ et $\mathbf{y}$. Comme le sujet nous paraissait passionnant, nous avons inclus dans le rapport des approches de résolution non demandées, qui sont fournies dans des fichiers Python différents.

1 The datasets : MNIST

Pour apprendre à nos modèles à reconnaître les chiffres écrits à la main, de styles et de complexités différents, nous utiliserons la base de données MNIST [1] conçue par Yann Le Cun. Les données sont disponibles sur son site [2] `http://yann.lecun.com/exdb/mnist/` et mais également dans la bibliothèque Keras. Le jeu de données contient initialement 60 000 images dans le training set et 10 000 images dans le test set, cependant, notre méthode d'importation mixant les deux sets, nous allons devoir recréer un test set nous-mêmes.

Le jeu de données est de taille 70000x28x28x1 : 70 000 images de taille 28 pixels de hauteur x 28 pixels de largeur, et les images étant en noir et blanc on se permet de noter immédiatement le nombre de canaux égal à 1 (si les images étaient colorées nous aurions dû préciser 3 canaux, un pour chaque couleur).

Nous importons les données avec le script Python suivant :

```
# load the data
X = np.load("MNIST_X_28x28.npy")
y = np.load("MNIST_y.npy")
```

Voici quelques exemples d'images que nos modèles vont traiter :

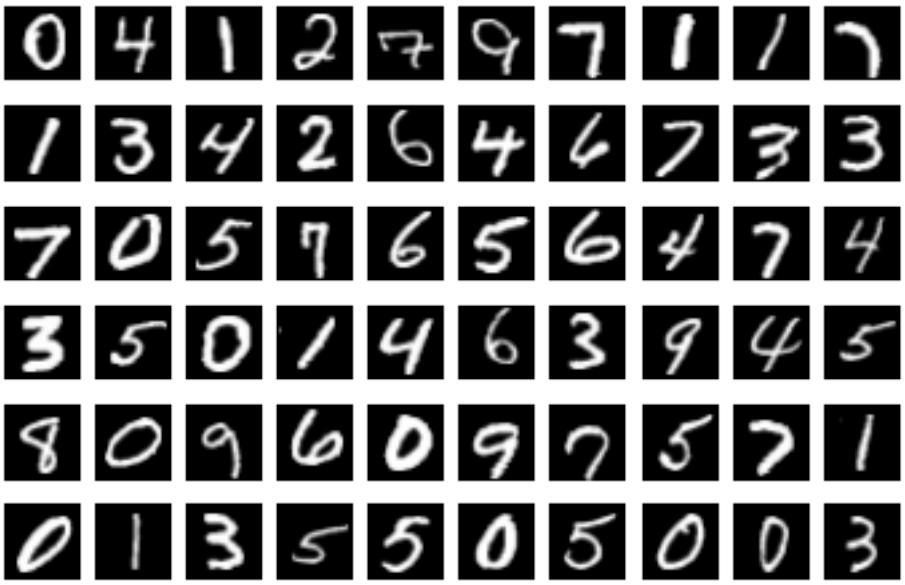

Figure 1 – Exemples d'images du MNIST

1. Mixed National Institute of Standards and Technology
2. On trouve aussi sur ce site la liste des meilleurs résultats pour chaque modèle traitant la reconnaissance de chiffres manuscrits.

Dans notre cas, tout le reste du TP va se concentrer sur un cas de Classification de données en apprentissage automatique, c'est-à-dire que les modèles vont devoir apprendre, de manière supervisée ou non-supervisée, à prédire la catégorie à laquelle chaque image appartient (un chiffre entre 0 et 9). Il est donc primordial de s'assurer que les données sont réparties de manière équilibrée, et qu'il y ait (presque) autant d'images de chaque chiffre, afin de ne pas délaisser ou sur-apprendre certains chiffres. On s'attend donc à voir environ 7000 images d'exemple par chiffre.

L'histogramme suivant illustre la répartition du jeu de données importé :

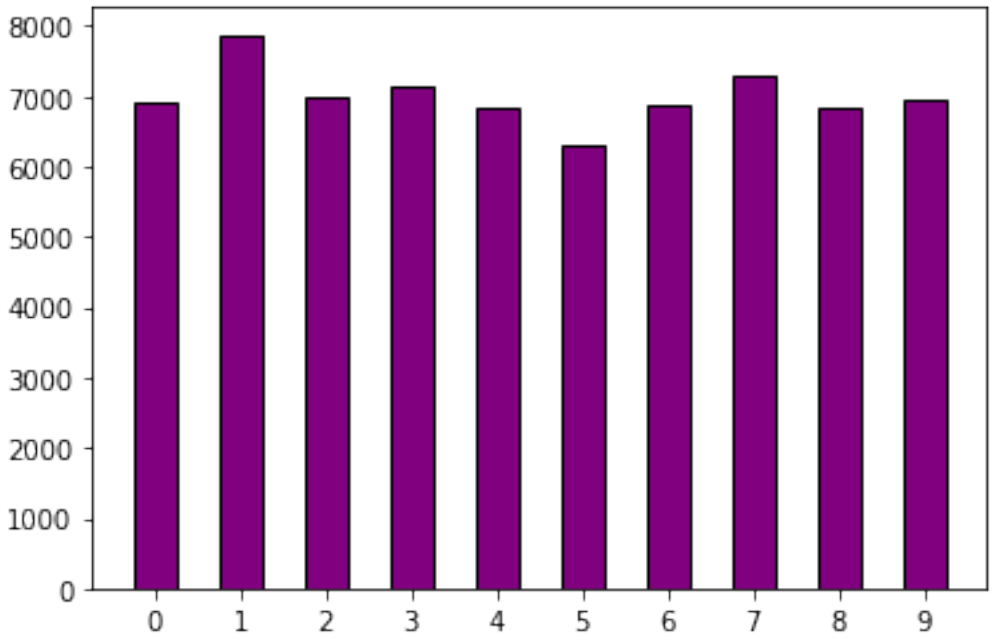

Figure 2 – Répartition des chiffres dans le dataset

Comme nous pouvons le constater, la répartition est équilibrée, et nous allons maintenant pouvoir séparer les données en deux sous-ensembles : l'ensemble de données d'entraînement, et l'ensemble de données de test. Tant que cela n'est pas précisé, nous n'utiliserons pas d'ensemble de données de validation. Nous allons donc entraîner chaque modèle sur l'ensemble des données d'entraînement puis évaluer sa performance sur l'ensemble des données de test. Nous avons décidé de séparer l'ensemble des données en attribuant 80% des images à l'ensemble d'entraînement, et 20% des images à l'ensemble de test. Nous devons, cependant d'abord normaliser les données afin que chaque pixel (codé sur $[\![0, 255]\!]$) ait une valeur entre 0 et 1.

Le code Python qui nous permet d'effectuer notre étape de pré-traitement de données est :

```
# data normalization
X /= 255.0
```

Le code Python qui nous permet de diviser de manière aléatoire les 70,000 images en deux ensembles est le suivant :

```
# creation des jeux de donnes training set et testing set
X_train, X_test, y_train, y_test = train_test_split(X,y,
                                                    test_size=0.2,
                                                    random_state=42)
```

Pour que certains de nos modèles puissent correctement comprendre les données, nous devons vectoriser chaque image durant l'étape de pre-processing : chaque matrice 28x28 doit changer de forme et devenir un vecteur 784. Nous le faisons de la manière suivante :

```
# vectorisation des jeux de donnees
X_train = X_train.reshape(len(X_train), -1)
X_test = X_test.reshape(len(X_test), -1)
```

Les deux jeux de données de train et de test sont désormais des tables de longueurs respectives 56000 et 14000, et de largeur 784.

2 Unsupervised Machine Learning

Dimensionality reduction

Les données sont désormais dans $\mathbb{R}^{784}$, ce qui implique que les modèles vont demander beaucoup de puissance de calcul pour traiter les images vectorisées. On souhaiterai donc réduire la dimensionalité des données pour gagner en temps de calcul. On peut le faire grâce à une *Principal Component Analysis*[3] (PCA) qui va, de manière non-supervisée, évaluer la corrélation entre chaque dimension (à comprendre dans notre cas, entre chaque pixel), et va nous permettre de transformer notre jeu de données en le plaçant dans $\mathbb{R}^n$. On aimerait que n soit considérablement plus petit que 784, afin de gagner en temps de traitement de données durant la phase d'entraînement des modèles.

Le script Python qui nous permet d'évaluer l'intérêt de chaque composante grâce à la PCA est le suivant :

```
# Principal Component Analysis
pca = PCA(n_components = 784)
pca.fit(X_train)
```

En utilisant différentes valeurs de **n_components**, on obtient une analyse plus ou moins réductrice, avec cependant une perte d'informations proportionnelle au gain dimensions. Dans notre premier cas, nous avons fait l'analyse avec 784 composantes pour évaluer l'importance relative de chaque dimension par rapport à la quantité d'informations comprises dans une image. Voyons maintenant à quoi ressemble les données après une transformée par PCA en choisissant 196 composantes (on remet les données en format 14x14 pour la visualisation plutôt que de garder un long vecteur de taille 196)

3. Analyse en Composantes Principales

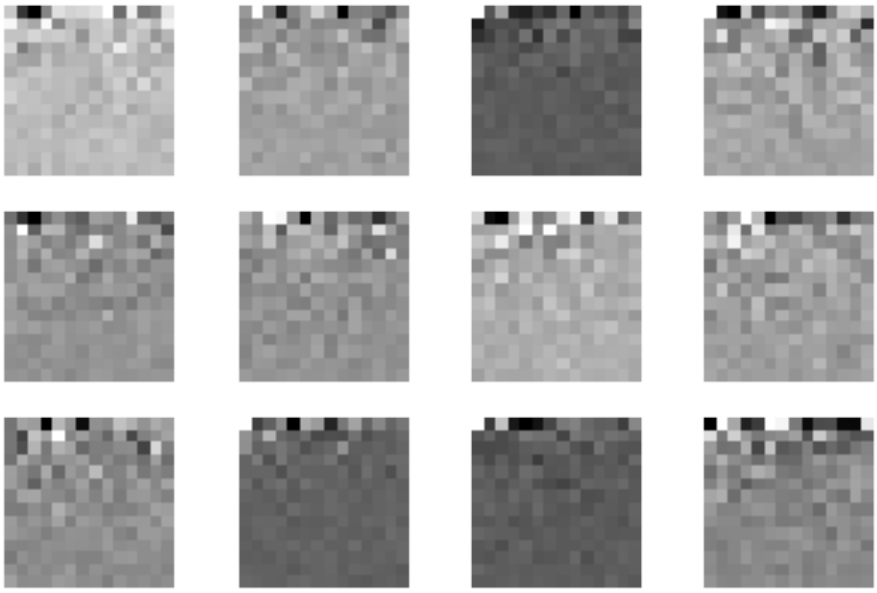

Figure 3 – Représentation des chiffres après une PCA

On observe bien que les images n'ont alors plus aucun sens pour nous, car nous ne serions pas capables de deviner quel chiffre se cache derrière chaque image transformée. On peut toutefois remarquer plusieurs chose. Les pixels de la première ligne sont les plus différents les uns des autres et sont ceux qui varient le plus d'une image à l'autre. ce sont en effet, les composantes les mieux classées par la PCA. A l'inverse, les pixels des dernières lignes varient peu les uns des autres et semblent même dépendants les uns des autres. On peut donc imaginer qu'une PCA plus réductrice serait envisageable.

Si ces images restent inexploitables pour nous, elles ont un sens pour les modèles d'apprentissage automatique, qui vont pouvoir apprendre sur ses données transformées et vont surtout passer moins de temps à le faire car elles sont plus compactes : on a ici divisé par deux la largeur et la hauteur, et donc appliqué une racine carrée sur le nombre de features à traiter par les modèles. L'analyse en composantes principales est un outil important que nous allons réutiliser plus tard pour voir si nos modèles apprennent tout aussi bien en ayant en entrée le training dataset avec une PCA ou sans PCA.

Comment choisir le nombre de **n_components** à fournir en paramètre de la PCA ? Nous allons étudier les résultats de notre analyse sur 784 composantes pour décider si toutes ces dimensions sont importantes pour nos modèles. En utilisant la bibliothèque *matplotlib.pyplot* pour afficher la somme des variances des composantes classées par importance croissante, on conclut que seules 153 composantes sont nécessaires pour avoir au moins 95% de l'information comprises dans les données. Ainsi, il existe des correlations linéaires entre les valeurs des pixels, et cela se comprend : pour les images des chiffres, les zones noires ne sont pas intéressantes, et surtout nous ne voulons pas que les modèles les utilisent. On souhaite qu'ils apprennent à remarquer les pixels blancs et les pixels noirs qui forment le contours des chiffres. Le résultat visuel des variances des composantes est le suivant :

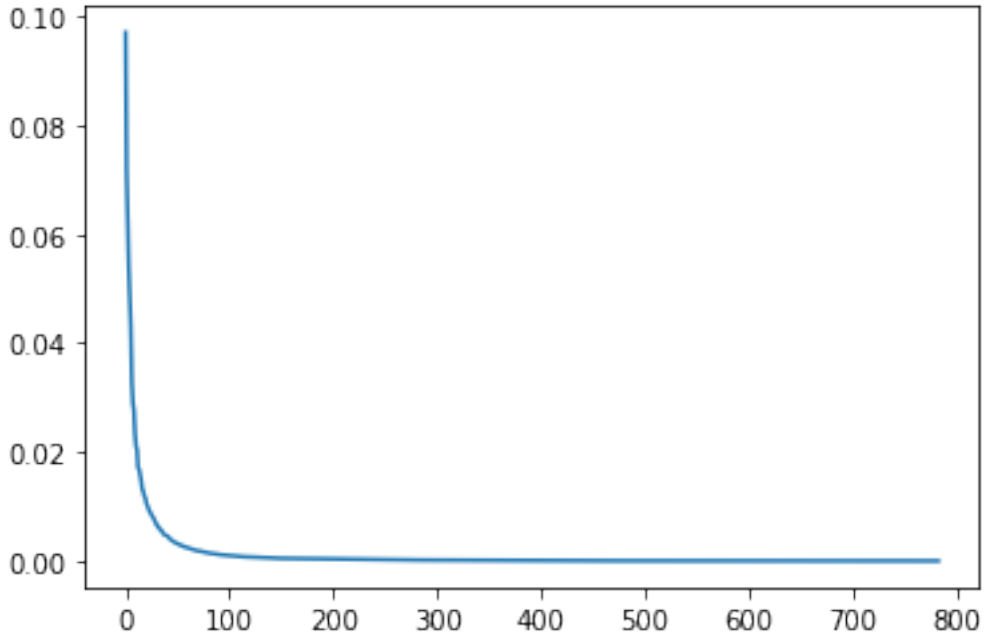

Figure 4 – Représentation des variances de chaque composante

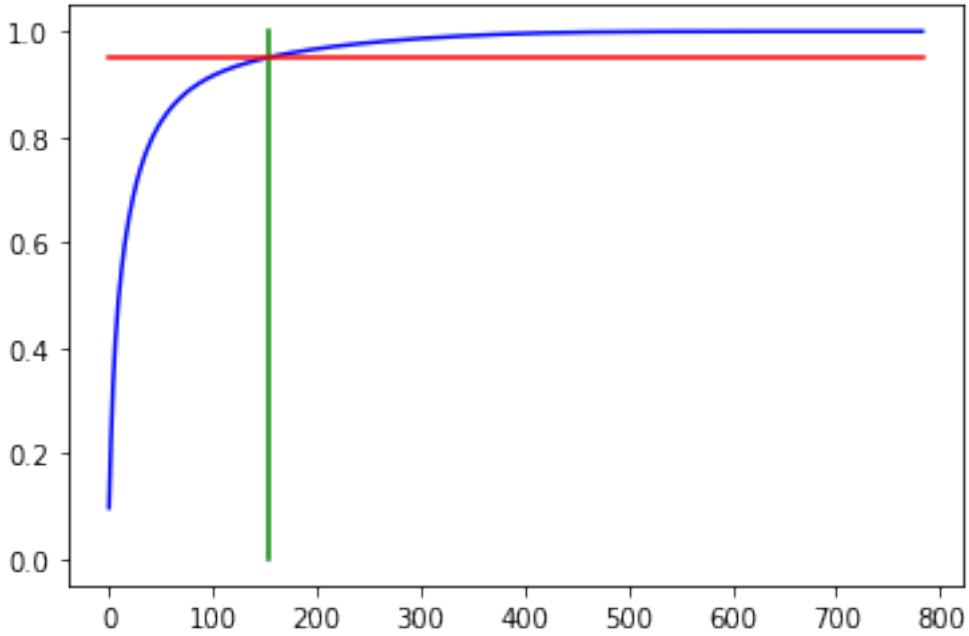

Figure 5 – Représentation de la somme des variances

Data clustering

K-Means

A ce stade, le jeu de données a été normalisé, vectorisé, pré-traité, et séparé en deux ensembles de données : celles d'entraînement et celles de test. Nous allons maintenant entraîner des modèles d'apprentissage automatique non-supervisés en observant l'impact qu'a la PCA sur les résultats qu'ils fournissent.

Nous utilisons d'abord les fonctions de la bibliothèque SkLearn pour créer un modèle d'apprentissage suivant la méthode des K-Means. Cette méthode consiste à trouver des centroïdes en se basant sur le training dataset : les centroïdes sont initialisés de manière aléatoire, puis sont déplacés à chaque itération au centre du nuage formé par les points dont ils sont le plus proche centroïde, et ce jusqu'à une condition d'arrêt.

Ensuite, chaque point du testing dataset est attribué au noyau (*cluster*) dont le centroïde est le plus proche du point. Dans notre étude, nous avons 10 classes différentes, une pour chaque chiffre. Le modèle sera donc optimal lorsque nous lui demanderons de chercher 10 centroïdes (et donc 10 noyaux).

Le code Python utilisé pour générer ce modèle est le suivant :

```
# K-Means Clustering
kmeans = KMeans(n_clusters=10, init='k-means++',n_init=10)
kmeans.fit(X_train)
y_kmeans = kmeans.predict(X_test)
```

Parmi les différents paramètres, il est important de noter que le choix d'initialisation à **k-means++** est le plus efficace car il place préemptivement les centroïdes de manière espacée et donc accélère grandement la tâche séquentielle de rapprochement des centroïdes vers leur position optimale.

Il nous est important de parler des méthodes de *scoring* des modèles d'apprentissages automatiques : ici, comparer naïvement les clusters seraient contre-productif car rien ne nous assure que le cluster contenant les images labellisées par le chiffre "2" soit lui-même numéroté "2". C'est pour cette raison que nous allons assigner à chaque cluster son étiquette pour pouvoir mieux évaluer nos modèles. De surcroît, plus particulièrement pour les algorithmes de clustering, on définit trois notions : l'inertie (*inertia*), la complétude (*completeness*) et l'homogénéité (*homogeneity*). L'inertie représente la distance moyenne entre les points des clusters et leur centroïde associé. L'homogénéité note la capacité du modèle à construire des clusters contenant uniquement des éléments de même label (ici, les éléments d'un cluster doivent avoir le même chiffre pour étiquette). Enfin, la complétude note la capacité du modèle à ranger les éléments de même label dans le même cluster. On note qu'il y a une relation entre l'homogénéité et la complétude : si on inverse les rôles des étiquettes vraies et des étiquettes prédites, le calcul de la complétude donne l'homogénéité en retour.

En pratique, on apprécie un algorithme de clustering qui réunit ces trois qualités. Nous prendrons aussi en compte un score qui les réunis tous : le V_measure_score. Nous allons évaluer notre modèle de KMeans pour plusieurs valeurs de K avec toutes ces notes. Voici les résultats :

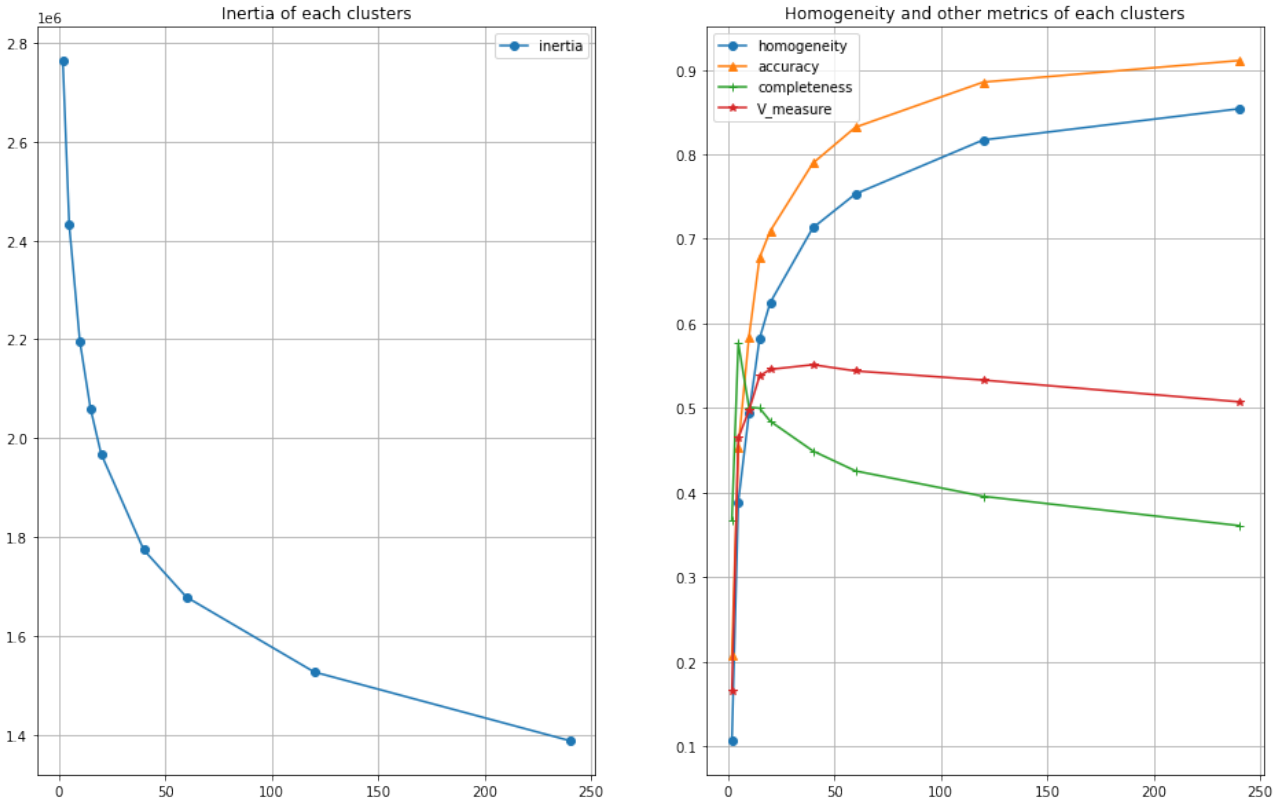

Figure 6 – Evaluation sur plusieurs métrique

Ces résultats sont cohérents avec notre compréhension des méthodes de clustering : quand on augmente le nombre de centroïdes (et de noyaux), on augmente la capacité de ces noyaux à contenir des images qui se ressemblent beaucoup. L'homogénéité et l'inertie sont améliorées (plus l'inertie est faible, mieux l'algorithme fonctionne), la complétude décroit (la V_measure aussi). Le score global augmente donc. Le choix du bon K (quand nous n'avons pas de connaissances a priori sur les données) reste une tâche compliquée car on ne peut pas se fier aveuglément au score de l'algorithme.

Pour le reste du TP, nous noterons les performances de nos modèles de K-Means (avec toujours K=10) en utilisant les fonctions **rand_score** et **adjusted_rand_score** de **sklearn.metrics** car elles nous permettent de réaliser le taux de réussite du modèle ainsi que son efficacité par rapport à un algorithme qui attribuerait de manière aléatoire les points qu'on lui donne à un chiffre. L'intérêt des méthodes rand_score et adjusted_rand_score que nous utilisons est qu'elles comparent le taux de similitude entre les deux ensembles (les **true** labels et le **predicted** labels). Enfin, elles ont la propriété d'être symétriques, on peut aisément inverser les deux ensembles d'étiquettes. Le score net est compris dans [0 ;1] et le score ajusté est inclus dans [-1 ;1].

Pour un apprentissage sur 10 centroïdes, avec une initialisation **k-means++**, et sans aucune transformée PCA sur le jeu de données, on obtient les résultats suivants :

```
training_score:          0.8812447849834053
adjusted_training_score:0.36242353339880357
testing_score:           0.881983641688692
adjusted_testing_score: 0.36490393108616875
```

Nous constatons qu'il n'y a pas d'overfitting (sur-apprentissage), mais nous aimerions que nos prochains modèles soient plus performants, car 88% de précision est un score qui peut être amélioré.

Essayons désormais de voir si une PCA permet de garder un score proche de celui obtenu ci-dessus, malgré la perte d'information. On choisit comme toujours K = 10 pour le modèle de K-Means et on souhaite garder uniquement deux composantes (on passe de 784 dimensions à 2 dimensions). Voici le résultat :

```
training_score:            0.8579192951149638
adjusted_training_score:   0.2392671457963771
testing_score:             0.8570000306144316
adjusted_testing_score:    0.23471566629754195
```

Nous pouvions nous en douter, en gardant deux dimensions seulement, nous perdons des informations, ce qui se reflète sur le score du modèle entraîné. Néanmoins, nous sommes surpris que le modèle reste plutôt performant malgré la diminution de features. Nous ferons attention à cela lors des prochaines transformées PCA. Nous allons désormais essayer une autre technique, celle des Expectation-Maximization (EM) Gaussian Mixtures.

EM Gaussian Mixtures

Ces deux algorithmes ont le même objectif mais ne l'atteignent pas de la même manière. L'algorithme de K-Means cherche à placer K centroïdes où chaque point appartient au cluster correspondant au centroïde le plus proche. A chaque itération, les centroïdes sont déplacés de façon à ce qu'ils soient la moyenne arithmétique de tous les points dans son cluster. Puis les données d'entraînement sont réutilisées et redistribuées dans le cluster le plus proche. Ces étapes sont répétées jusqu'à une condition d'arrêt. La Mixture de Gaussiennes consiste à placer plusieurs fonctions Gaussiennes multidimensionnelles pondérées. Cet algorithme a un avantage : les noyaux n'ont pas besoin d'être circulaires. A chaque itération, les points des données d'entraînement sont ajoutés à un cluster selon les poids, puis chaque cluster met à jour ses paramètres selon la position des données (ceux dans le cluster et ceux en dehors). Le code Python permettant d'implémenter ce modèle et de l'entraîner est :

```
# EM Gaussian Mixture
EMGauss = GaussianMixture(n_components=10, random_state=42)
EMGauss.fit(X_train)
y_EM_train = EMGauss.predict(X_train)
y_EM_test = EMGauss.predict(X_test)
```

Une fois créé, on obtient ces résultats sur les jeux de données d'entraînement et de test :

```
training_score:  0.8442799731117647
adjusted_training_score:  0.2297375672396504
testing_score:  0.844204881981366
adjusted_testing_score:  0.23178228979845533
```

On remarque que la Mixture de Gaussiennes obtient des résultats moins efficaces et précis avec un temps de calcul beaucoup plus important. Elle reste en théorie plus efficace mais pas dans notre cas.

Deep Learning : Autoencoders (Extra)

Nous abordons brièvement l'apprentissage profond dans cette partie non-supervisée pour parler des autoencodeurs. La particularité de ces modèles est que la sortie est l'entrée : on construit notre réseau de neurones de manière symétrique, une partie encodeur et une partie décodeur. Le bloc d'encodage est un réseau dont les couches diminuent en taille, prenant en entrée l'image (dans notre cas) jusqu'à atteindre la *Latent Size*. Le bloc de décodage prend en entrée le vecteur de taille réduite et donne en sortie l'image reconstruite ; ses couches sont les mêmes que celle de l'encodeur mais dans l'ordre inverse. Ce modèle permet de faire à la fois de la réduction de dimension, et aussi de crypter les données : le vecteur réduit est l'information chiffrée, les deux blocs sont les clefs. Le modèle suit cette structure :

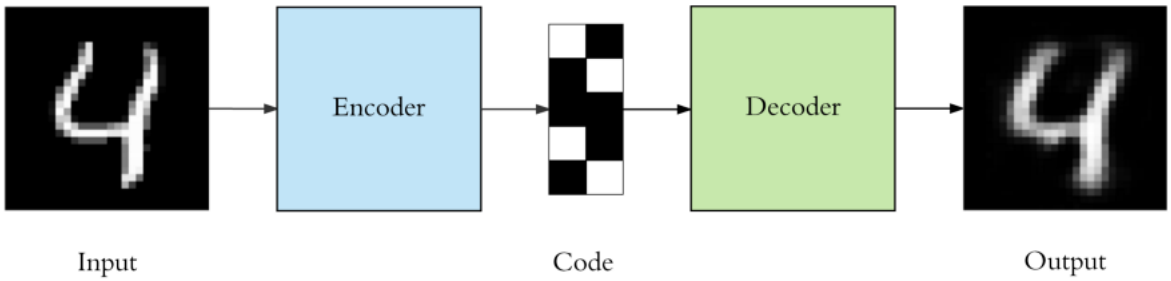

Figure 7 – Exemple d'Autoencodeur

Après avoir construit notre autoencodeur (de taille réduite de 32 dimensions) et l'avoir entraînée sur 20 epochs, nous obtenons cette reconstruction d'image :

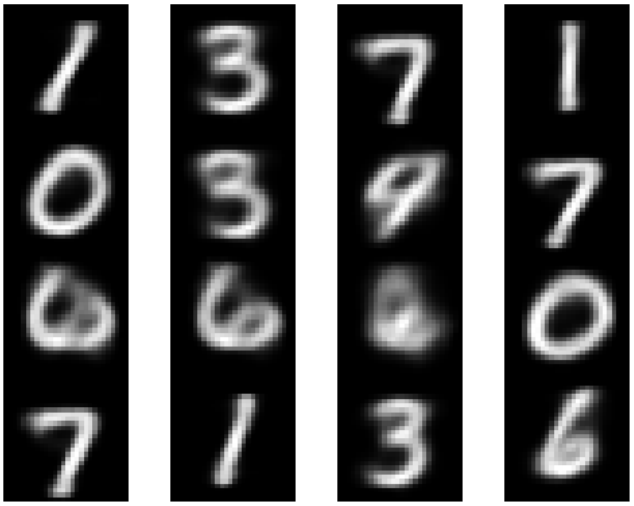

Figure 8 – Exemple de reconstructions d'images

Compte tenu de la qualité des images en entrée nous pouvons être satisfaits par les résultats de notre autoencodeur. Nous pourrions obtenir des résultats plus efficaces en construisant un Variational AutoEncoder et en entraînant le modèle sur plus d'epochs, mais nous ne traiterons pas cela ici. Voyons désormais des modèles d'apprentissages automatiques supervisés.

3 Supervised Machine Learning

Decision Tree

Le premier modèle d'apprentissage supervisé que nous avons testé est l'arbre de décision. Cette solution nous paraissait trop brute pour travailler sur un problème de classification d'image, et nous nous attendions donc à des résultats médiocres.

Comme première approche, nous avons gardé les paramètres par défaut de l'arbre. Nous avons obtenu les résultats suivants :

```
depth: 42
n_leafs: 3786
accuracy on training set: 1.0
accuracy on testing set: 0.8693571428571428
```

On obtient un overfitting manifeste inhérent à l'entrainement d'un arbre de décision sans aucune contrainte de taille. Le modèle a simplement appris chaque image par coeur dans ses branches. En revanche, les resultats sur le testing set ne sont pas mauvais pour autant et même plutôt surprenants au regard de l'incohérence entre le modèle et la nature du problème.

Pour autant, nous avons décidé d'améliorer les performances de l'arbre de décision en limitant son nombre de feuilles (et donc sa complexité) afin qu'il cesse d'overfitter à l'apprentissage. En prenant un maximum de 1000 feuilles [4], nous obtenons les résultats suivants :

```
depth: 17
n_leafs: 1000
accuracy on training set: 0.9344642857142857
accuracy on testing set: 0.8802142857142857
```

Le modèle overfit moins et on a gagné en performances sur l'apprentissage. Cela est compréhensible car l'arbre et moins complexe et a donc été forcé de généraliser. Une dernière piste que nous avons exploré est l'usage de la PCA : puisque l'arbre de décision choisit des features et place des critères dessus pour classifier les images, nous allons "l'aider" à choisir les features importants. Nous avons cependant obtenu des résultats médiocres avec les différents nombres de composantes que nous avons essayés. Ici pour 153 composantes :

```
depth: 21
n_leafs: 1000
accuracy on training set: 0.9154107142857143
accuracy on testing set: 0.8433571428571428
```

Notre analyse est que le modèle de l'arbre de décision est si peu adapté au problème que réduire la taille de ses entrées ne lui permet pas de gagner en performance. L'arbre de décision parvenait à tirer une information utile de presque tous les pixels de l'image.

SVM

Les Support Vector Machines (SVM) ont longtemps été les modèles les plus utilisés car les réseaux de neurones n'existaient qu'à l'état théorique, et les SVM sont en général les modèles d'apprentissage automatique supervisé les plus performants. Elles fonctionnent en séparant les données de manière linéaire (même si des SVM plus avancées sont non-linéaires) selon un hyperplan. En travaillant dans $\mathbb{R}^{784}$, le modèle va trouver dix équations (on suppose une stratégie d'apprentissage "one-vs-all") caractéristiques dans $\mathbb{R}^{783}$ pour séparer les données en dix catégories, en se basant sur des Support Vectors, c'est-à-dire des points des données qui modélisent ensemble un hyperplan auquel ils sont tous équidistants, et duquel les autres données sont le plus éloigné. Plus les données sont éloignées du plan de séparation, mieux l'algorithme fonctionne. Son seul bémol est qu'il est très sensible aux "outliers" (les données qui sortent de la norme). Pour pallier ce problème, nous distinguerons par la suite les SVM à *hard-margin* et à *soft-margin*. Le second modèle est plus permissif, et autorise à un certain degré, des données à se trouver entre l'hyperplan de séparation et la marge. Cela permet d'avoir des modèles plus souples et plus représentatifs de la réalité. Cette distinction se fera dans le langage Python par le paramètre C, qui va régler la souplesse de l'hyperplan par rapport aux points du modèle.

Voici les résultats obtenus pour C = 1 (Hard-Margin) et C = 0.05 (Soft-Margin) :

```
# Linear Hard-Margin
Training Dataset: 0.9723214285714286
Testing Dataset: 0.9352142857142857
```

4. Ne pas confondre avec la pâtisserie

```
# Linear Soft-Margin
Training Dataset: 0.9555535714285714
Testing Dataset: 0.9424285714285714
```

Nous constatons que le modèle Hard-Margin fait plus d'overfitting : l'absence de tolérance ne lui permet pas de classifier de manière généralisée les images selon leur chiffre, ce que le modèle Soft-Margin fait mieux car il apprend de ses données avec plus de souplesse, de régularisation. Nous pouvons aussi modéliser la réussite d'un modèle de classification par une matrice de confusion, afin de visualiser les performances des clusters créés. La Confusion Matrix est constructible car disposons à la fois de la vraie étiquette de chaque image, ainsi que de son label prédit :

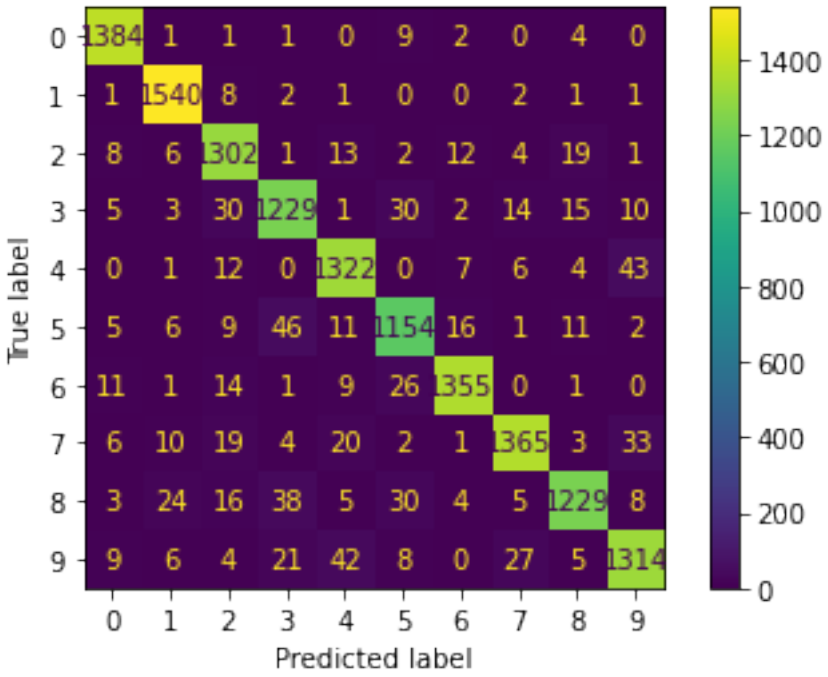

Figure 9 – Matrice de Confusion pour la SVM Soft-Margin

Les résultats sont très satisfaisants car dans la plupart des cas, le modèle prédit avec succès le label des données de test. La matrice de confusion met en évidence le taux de succès élevé du modèle Soft-Margin.

Nous nous intéressons enfin à un autre modèle de SVM, le Radial Basis Function (RBF) qui utilise les distances euclidiennes aux Support Vectors, qui servent alors de centres. Ce modèle permet une classification non-linéaire. Voici les résultats obtenus :

```
# SVM Radial Basis Function
Training Dataset: 0.9897857142857143
Testing Dataset: 0.9785
```

Ce modèle est plus performant que les deux précédents : il a une meilleure précision sur le test dataset mais il fait un peu d'overfitting. Globalement, les SVM sont très puissantes et on comprend pourquoi celles en mode 'RBF' ont été les plus populaires avant l'avènement de l'apprentissage profond et des réseaux de neurones.

Logistic Regression

La regression logistique se modélise par la formule mathématique suivante [5] :
$\hat{y} = sigmoid(W^t * x + b)$

La fonction sigmoïde rapporte sur [0 ;1] le résultat obtenu de la somme des produits des poids avec la feature de x correspondante, et un biais. Il est à noter que cette formule est pour la classification binaire, et que nous nous intéressons à une régression logistique multiclasse : la formule doit être adaptée. Pour la régression logistique, nous avons obtenu des résultats très bons avec peu de temps de calcul en gardant les paramètres par défaut.

```
accuracy on training set: 0.9338214285714286
accuracy on testing set: 0.9284285714285714
```

Nous avons ensuite paramétré le modèle comme conseillé sur le site de sklearn. Nous avons donc choisi le solver 'sag' optimisé pour les grands datasets avec de nombreuses features. Nous avons fixé un max_iter à 200 dans l'espoir de voir converger les coefficients, ce qui ne s'est malheureusement pas produit. Nous avons tout de même obtenu des résultats très satisfaisants mais en dessous de ceux obtenus avec les paramètres par défaut :

```
accuracy on training set: 0.9407142857142857
accuracy on testing set: 0.9190714285714285
```

Naive Bayes Classifier

Le Classificateur Naive Bayes se base mathématiquement sur la loi de probabilité de Bayes :

$$P(\mathbf{Y}|\mathbf{X}) = P(\mathbf{Y})\frac{P(\mathbf{X}|\mathbf{Y})}{P(\mathbf{X})}$$

On cherche, comme dans tout modèle d'apprentissage automatique supervisé, à apprendre P(Y|X), mais cette fois ci on passe par une approximation de P(X|Y). De la manière la plus simple, on estime que la distribution est Gaussienne avec aucune covariance pour chaque dimension.

```
# Naive Bayes Classifier - Gaussian
Training Dataset: 0.5718571428571428
Testing Dataset: 0.5717857142857142
```

On remarque que ce modèle fait de l'underfitting sur la reconnaissance de chiffres manuscrits, il est le modèle le plus mauvais rencontré jusque là.

5. Il se trouve que c'est la formule du perceptron, mais on lui laissera plus de choix dans la fonction d'activation.

K-Nearest Neighbors (Extra)

On va tester un autre modèle de Supervised Learning qui n'a pas été présenté mais qui peut-être intéressant : le modèle des K-Nearest Neighbors. Le modèle va simplement apprendre la position de chaque point dans le jeu de données d'entraînement. Pour la prédiction de nouvelles images dans le jeu de données de test, il va assigner l'étiquette qui appartient aux K voisins plus proches de chaque image. Voici des résultats pour des valeurs différentes de K :

```
# K = 2
Training Dataset: 0.9850357142857142
Testing Dataset: 0.9669285714285715
# K = 5
Training Dataset: 0.9810178571428572
Testing Dataset: 0.9707142857142858
# K = 8
Training Dataset: 0.9759464285714285
Testing Dataset: 0.9678571428571429
# K = 16
Training Dataset: 0.9694107142857142
Testing Dataset: 0.962642857142857
```

Ces résultats montrent que le K-Nearest Neighbors n'est pas à sous-estimer, mais il est difficile de choisir le meilleur K. Nous voyons ici que si K est trop petit, on rencontre de l'overfitting, mais que si K est trop grand, le modèle perd en précision. Ainsi, on comprend l'importance de réessayer et de vérifier les performances pour plusieurs K différents, en utilisant par exemple la *cross-validation* pour trouver le meilleur modèle. Il est désormais temps de s'intéresser aux modèles d'apprentissages profonds, qui ont énormément de potentiel.

Deep Learning

Dataset

Nous avons utilisé le même dataset que précédemment pour entraîner nos modèles. Nous aurions pu réduire sa taille pour gagner en calcul, mais soucieux d'obtenir les meilleurs résultats possibles, et ayant à dispositions des machines relativement performantes, nous ne l'avons pas fait. Nous avons donc entraîné nos modèles sur 56000 images (80% du set), et utilisé les 14000 restantes pour tester l'efficacité finale du modèle.

Objective and tools

Pour la construction de réseaux de neurones, la bibliothèque Keras fournie par TensorFlow a été notre principal outil. Elle propose deux API, la Sequential API et la Functional API. La première nous permet de créer un modèle puis d'y rajouter des couches petit à petit. La seconde nous permet de définir chaque couche en précisant laquelle lui précède : cette méthode est plus flexible car elle permet de définir les couches indépendemment les unes des autres, et l'on peut "branch out" des couches pour les réutiliser dans un autre réseau de neurones. Comme ceci ne nous intéresse pas ici, nous avons décidé d'utiliser la Sequential API [6] pendant ce TP.

6. Malgré l'indication du sujet qui préfère le Functional API

MultiLayer Perceptron (MLP)

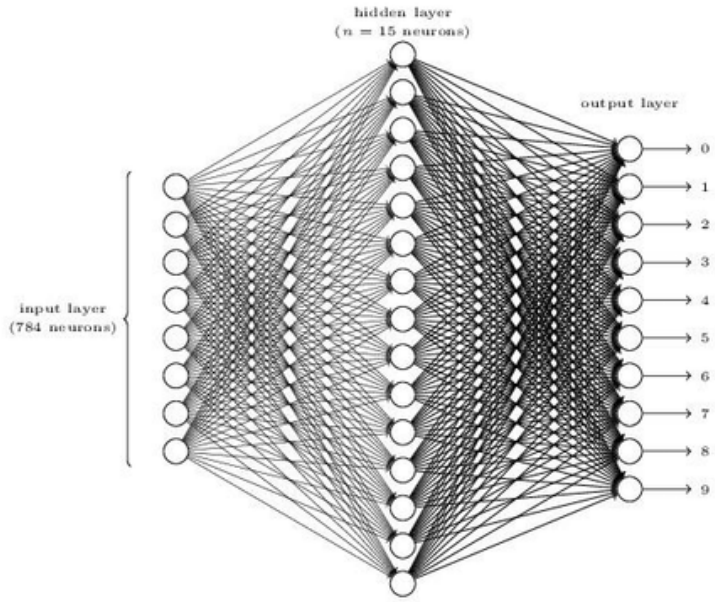

Figure 10 – Exemple de Réseau de Neurones pour le MNIST

Un réseau de neurones peut être décomposé en trois parties : la couche d'entrée (*input layer*), la couche cachée (*hidden layer*) et la couche de sortie (*output layer*). La couche d'entrée est composée d'autant de neurones que de features utilisés (ici il y en a 784, un pour chaque pixel). La couche de sortie est composée de 10 neurones : un neurone par chiffre. La couche cachée est là où nous pouvons effectuer nos manipulations et nos choix : on y met autant de couches que souhaité, chacune avec un nombre de neurones et fonction d'activation propre. Une fois le modèle compilé, avec sa fonction de loss et sa fonction d'optimisation, il peut être entraîné. La première étape est la *forward-pass* où le modèle va classer chaque image selon ses poids et enfin calculer son erreur. Ensuite vient l'étape de *backpropagation* où les poids sont mis à jour selon les erreurs. L'art du Deep Learning est de trouver les bons nombre de neurones et de couches, ainsi que les fonctions adaptées. Trouver l'architecture la plus efficace pour notre MLP a été une tâche ardue à cause du nombre de possibilités d'implémentation (une infinité). Nous avons commencé par un MLP avec un unique *hidden layer*. En choisissant arbitrairement 512 neurones pour le premier essai, nous avons obtenu des chiffres meilleurs que tous ceux obtenus jusqu'ici.

```
Train loss: 0.0224596094340086
Train accuracy: 0.9966250061988831
Test loss: 0.11025997996330261
Test accuracy: 0.9818571209907532
```

Plein d'espoir, nous avons pensé atteindre les 99% de *Test accuracy* en modifiant l'architecture pour diminuer l'overfitting manifeste du modèle, visible sur le graphique ci-dessous.

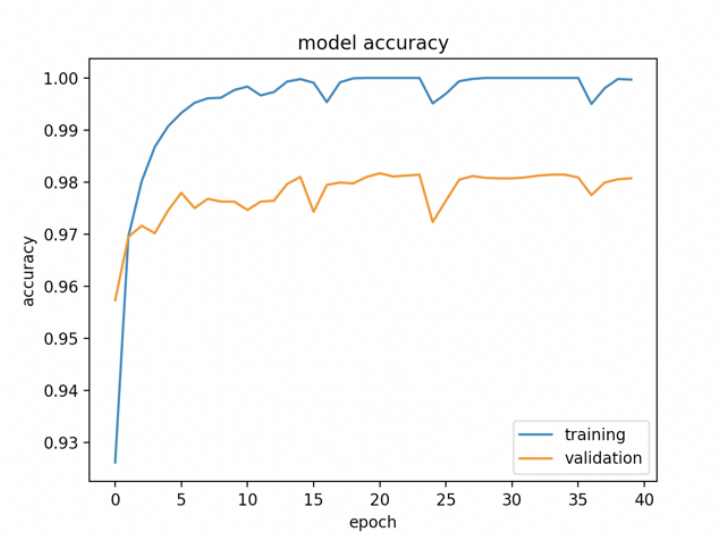

Figure 11 – Illustration de l'overfitting à l'apprentissage

On peut en effet s'apercevoir que le modèle a une précision (presque) égale à 1 sur les données d'apprentissage, et des résultats bien plus médiocres sur les données de validation. Les résultats généraux restent tout de même très bon, même pour un MLP.

Afin de diminuer l'overfitting, nous avons choisi d'implémenter un *dropout layer* avant et après le *hidden layer*. Nous avons choisi les valeurs en nous basant sur les recommandations de divers sites spécialisés, et avons donc selectionné 0.25 comme probabilité de dropout sur l'input et 0.50 sur le *hidden layer*.

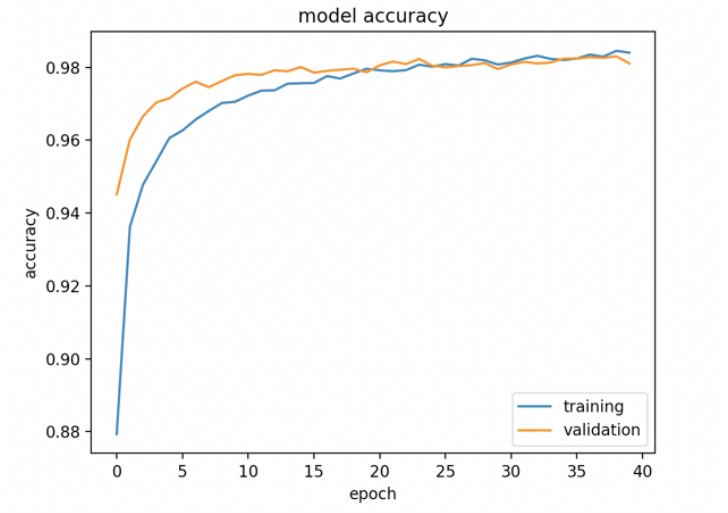

Figure 12 – Évolution de la précision du modèle sur les données d'entrainement et de validation au cours de l'apprentissage

Comme le montre la figure, nous avons réduit l'overfitting à l'apprentissage. Cela s'en ressent sur les résultats, puisque nous obtenons une amélioration de 18% sur l'error_rate obtenue avec les images de test.

```
Train loss: 0.01946776546537876
Train accuracy: 0.9949643015861511
Test loss: 0.064360611140728
Test accuracy: 0.9850714206695557
```

Enfin, nous avons tenté de modifier les coefficients de Dropout, et le nombre d'epochs pour améliorer le modèle en diminuant l'overfitting sur les données de test.
Notre MLP final comporte 2 *hidden layers* de 512 neurones. Nous avons appliqué un dropout de 20% au layer d'input et 50% aux autres. Après l'avoir entrainé sur 200 epochs, nous obtenons un modèle qui overfit légèrement avec cependant les meilleurs résultats obtenus jusqu'ici.

```
Train loss: 0.020253123715519905
Train accuracy: 0.9969106912612915
Test loss: 0.10803595185279846
Test accuracy: 0.9859285950660706
```

Nous avons aussi rajouté de la régularisation, c'est-à-dire que les poids vont désormais participer activement à la fonction de loss, pour continuer à diminuer l'overfitting, en utilisant la formule mathématique de la régularisation L2 :

$$Loss = Error(Y - \widehat{Y}) + \lambda \sum_1^n w_i^2$$

Cependant, les résultats n'ont pas été à la hauteur de nos espérances puisque le modèle a nettement baissé en performances.

Convolutional Neural Network (CNN)

Nous allons désormais nous intéresser à un modèle encore plus puissant que le MLP, un modèle qui prend en compte la notion de pixels voisins dans une analysique graphique. Nous parlons des réseaux convolutionnels, qui plutôt que d'être composés de couches de neurones, sont composés de couches de filtres, qui vont parcourir chacun l'image en entrée, afin d'obtenir en sortie la convolution de l'image par l'ensemble d'entre eux. Voici un exemple de modèle :

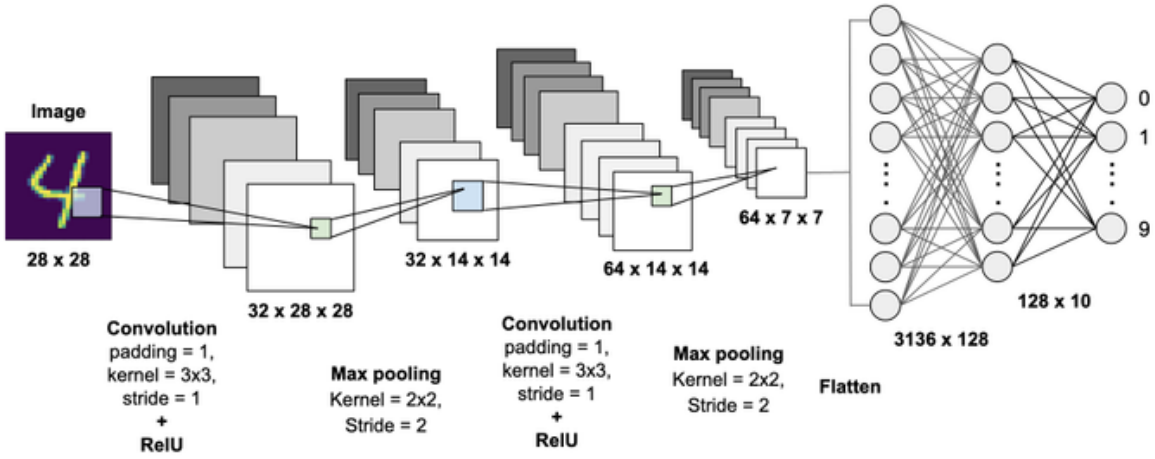

Figure 13 – Représentation d'un exemple de Réseau Convolutionnel pour le MNIST

Quels sont les paramètres grâce auxquels nous pouvons influencer notre modèle ?
Premièrement, nous décidons de combien de couches nous intercalons entre la couche d'entrée et la couche de sortie [7]. Nous pouvons changer la taille des filtres, le nombre de filtres par couches, le padding et le MaxPooling. Le padding indique comment les filtres parcourent l'image : un padding valant 'same' renvoie donc une image de même taille, et les pixels manquant pour les filtres au niveau des bordures sont générés automatiquement (par des zéros par défaut) ; un padding valant 'valid' renvoie une image légèrement plus petite, car les filtres commencent et terminent uniquement quand cela est possible (aucun pixel n'est créé). L'étape de MaxPooling entre chaque couche réduit la taille de l'image, selon la taille de son noyau (*kernel*) et la valeur de son pas (*stride*). Le pas décide combien de pixels séparent chaque regroupement de pixels. Pour un kernel = 2x2 et stride = 2, on réduit par deux la taille de l'image. Le MaxPooling permet de rendre le modèle robuste face aux translations (il saura reconnaître un chiffre même s'il est plus à gauche ou à droite). Il est nécessaire de ne pas oublier l'étape d'aplanissement (*Flatten*) qui était primordiale au début du TP pour les modèles précédents d'apprentissage automatique.

Nous allons implémenter d'abord une version très simple d'un modèle CNN. Nous avons utilisé 16 filtres de taille 2x2, un padding 'valid' car c'est pour nous le choix le plus judicieux : on évite ainsi de changer les informations aux extrémités, qui n'ont pas beaucoup d'intérêt dans notre cas de reconnaissance de chiffres. On lance le modèle sur 12 epochs et on obtient les courbes d'apprentissage suivantes :

7. Notons par ailleurs que la couche de sortie emploie la fonction d'activation 'softmax'

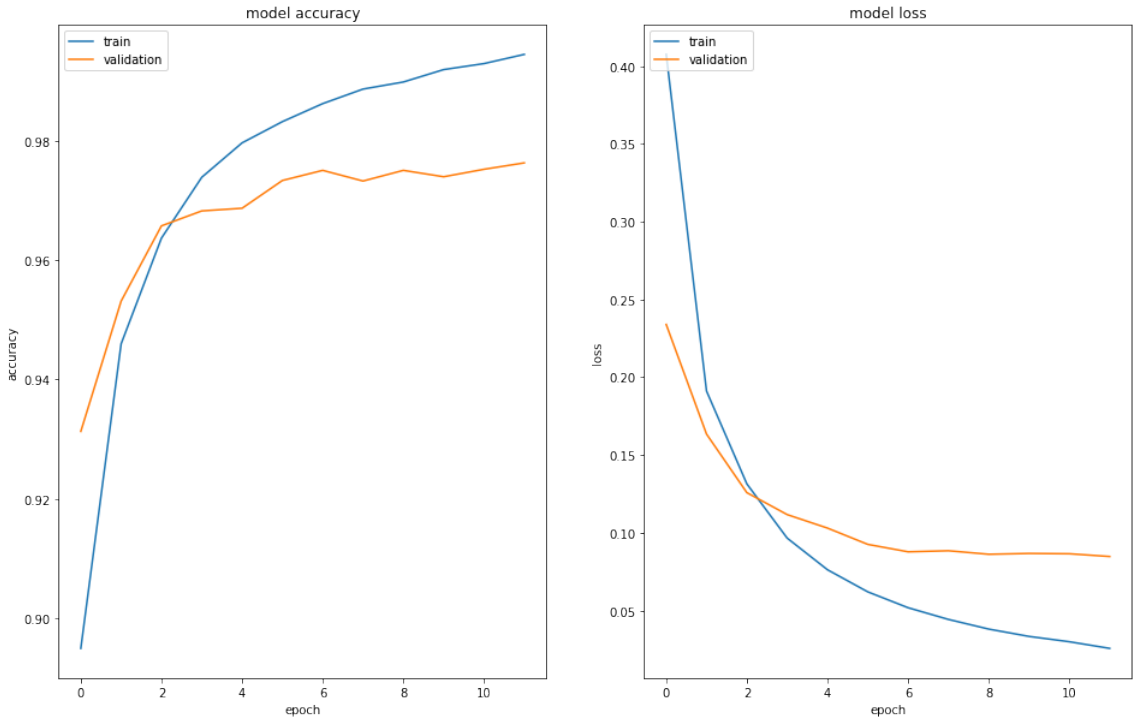

Figure 14 – Evolution de la loss pour un CNN sim

```
# Test Prediction Results
loss: 0.0856 - accuracy: 0.9764
```

Avec ces paramètres, les résultats ne sont pas à la hauteur de nos espérances, et nous subissons de l'overfitting. En combinant tous les éléments que nous avons découverts, et en rajoutant un petit MLP entre la sortie de notre réseau convolutionnel et la *output layer*, on obtient les résultats suivants :

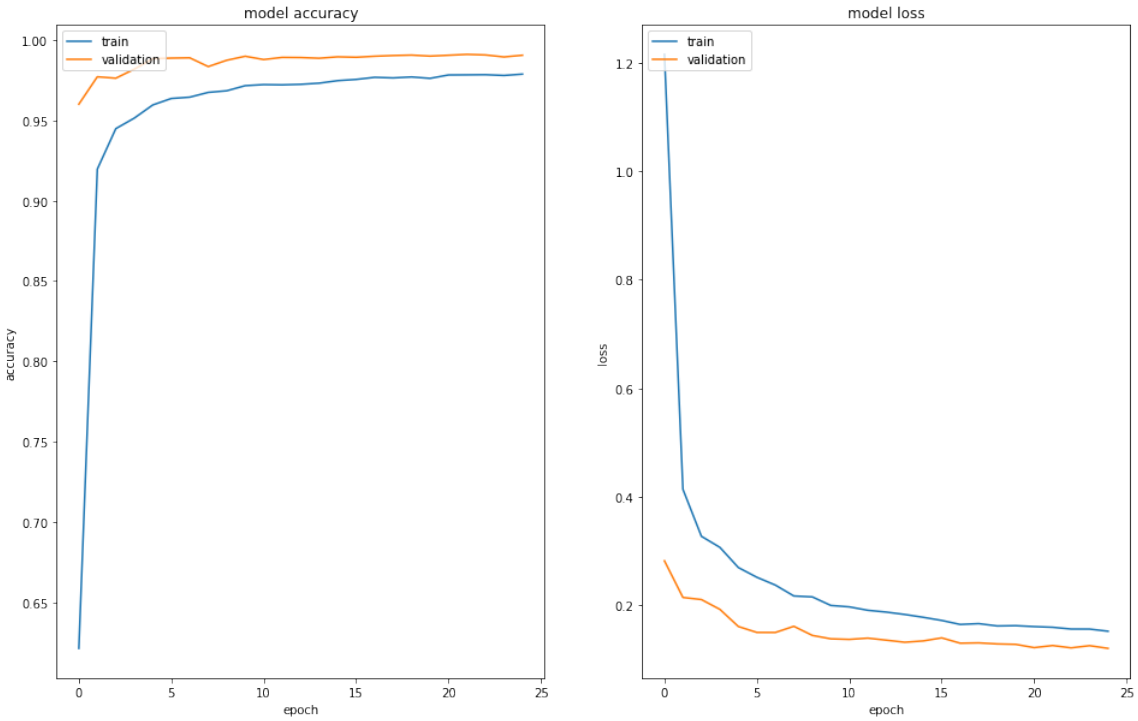

Figure 15 – Evolution de la loss pour un CNN complexe

```
# Test Prediction Results
loss: 0.1207 - accuracy: 0.9910
```

Nous sommes très fiers d'avoir obtenus 99,1% de précision sur le test dataset avec notre dernier modèle. Nous sommes, de plus, satisfaits que le Dropout et la régularisation soient si efficaces que tout le long de l'apprentissage, le modèle obtienne un score plus élevé sur les données de validation : il n'y a aucun overfitting. Nous avons essayé pleins de variations différentes, de rajouter ou de retirer des couches, d'augmenter ou baisser le Dropout, d'augmenter ou baisser la regularisation L2, de changer de fonction d'activation et d'optimisation. Nous ne sommes pas en mesure de trouver mieux avec nos capacités actuelles, mais nous avons un modèle très performant à la fin.

4 Machine Learning Theory (Extra)

Comment déterminer quel modèle utiliser en pratique ? Nous venons de voir une pléthore de modèles, mais comment en pratique rapidement choisir le meilleur modèle, celui le plus adapté au problème, et celui dont le tuning des hyperparamètres va donner les meilleurs résultats ? Pour répondre à cette question nous nous sommes intéressés à une bibliothèque python, qui semblait très pertinente : **LazyPredict** [8]. Elle permet de tester de nombreux [9] modèles de prédiction pour deux types de problèmes classiques : la classification (la fonction *LazyClassifier*) et la régression

8. la documentation est disponible sur https://lazypredict.readthedocs.io/en/latest/
9. Tous ceux fournis par les bibliothèques dont elle dépend. Cela ne concerne pas les réseaux de neurones à l'exception de l'unique perceptron.

(la fonction *LazyRegressor*). Elle utilise essentiellement les modèles fournis par **Scikit-Learn** mais aussi par d'autres bibliothèques moins connues telles que **XGBoost** et **LightGBM** (attention à ces dernières, il faut les installer soi-même). Une fois le calcul effectué, la fonction renvoie un tableau contenant une trentaine de modèles et leurs performances selon plusieurs métriques. Nous avons essayé de l'utiliser pour le dataset MNIST. Nous déconseillons cependant au lecteur de le faire car la biblothèque est peu adaptée à la version actuelle de SkLearn et l'opération demande beaucoup de temps de calcul. Néanmoins, voici les résultats renvoyés :

Out[3]:

Model	Accuracy	Balanced Accuracy	ROC AUC	F1 Score	Time Taken
XGBClassifier	0.98	0.98	None	0.98	512.57
ExtraTreesClassifier	0.97	0.97	None	0.97	50.24
LGBMClassifier	0.97	0.97	None	0.97	194.71
RandomForestClassifier	0.97	0.97	None	0.97	56.84
SVC	0.96	0.96	None	0.96	946.55
KNeighborsClassifier	0.94	0.94	None	0.94	1110.82
BaggingClassifier	0.93	0.93	None	0.93	509.27
LogisticRegression	0.92	0.92	None	0.92	19.11
CalibratedClassifierCV	0.91	0.91	None	0.91	2053.54
LinearSVC	0.90	0.90	None	0.90	506.84
SGDClassifier	0.90	0.90	None	0.90	368.22
Perceptron	0.88	0.87	None	0.87	11.60
DecisionTreeClassifier	0.87	0.87	None	0.87	25.87
PassiveAggressiveClassifier	0.87	0.86	None	0.87	20.56
LinearDiscriminantAnalysis	0.87	0.86	None	0.87	17.23
NuSVC	0.86	0.86	None	0.87	7307.46
RidgeClassifier	0.85	0.85	None	0.85	4.77
RidgeClassifierCV	0.85	0.85	None	0.85	16.22
BernoulliNB	0.83	0.82	None	0.83	6.42
ExtraTreeClassifier	0.82	0.81	None	0.82	3.28
NearestCentroid	0.80	0.79	None	0.80	3.49
AdaBoostClassifier	0.71	0.70	None	0.70	124.75
GaussianNB	0.54	0.53	None	0.50	4.76
QuadraticDiscriminantAnalysis	0.52	0.51	None	0.47	15.71
DummyClassifier	0.10	0.10	None	0.10	2.64

Figure 16 – Analyse complète des modèles

Ce classement place sur le podium des fonctions issues des bibliothèques qui nous sont inconnues. On note cependant que les fonctions à Forêts sont efficaces pour un temps de calcul faible, et il serait intéressant d'étudier leur fonctionnement pour nos futurs projets. On remarque que les modèles de Support Vector Classifier ont un très bon score et il serait intéressant de tenter de modifier les hyperparamètres afin d'obtenir un modèle encore plus précis, si cela est possible. Nous pourrions aussi chercher à optimiser en parallèle les réseaux de neurones MLP et CNN, en particulier le second car il est très performant pour la reconnaissance d'images. En effet, il possède plusieurs avantages : moins de poids, la possibilité de déceler des propriétés sur les groupes de pixels proches, et obtention d'une invariance aux translations. Il était intéressant nonobstant de voir un résumé des performances des techniques d'apprentissages automatiques pour savoir ce qu'il faut chercher à battre avec nos modèles d'apprentissage profond.

Conclusion

Nous avons parcourus beaucoup de chemin : nous avons vu plusieurs modèles standards de Machine Learning, de complexité croissante, et nous les avons implémentés en faisant varier les hyperparamètres, dans quête déterminée et inarrêtable du meilleur algorithme pour la reconnaissance du dataset du MNIST. Après avoir passé beaucoup d'efforts sur la reconnaissance de chiffres manuscrits, nous pourrions tenter d'appliquer nos modèles sur la classification de vêtements grâce à la Fashion MNIST - images 28x28 fournies par Zalando sur une variété de leurs produits. Les images sont plus compliquées et contiennent plus d'informations, mais nous avons désormais les outils pour résoudre ce problème.

Nous aimerions remercier le professeur Pierre-Alain Moëllic qui a pris le temps de nous instruire sur ce sujet, et de lire notre rapport.
Nous aimerions aussi remercier son doctorant qui était présent lors des séances de TP, et qui a su nous éclairer et nous donner des pistes de réflexion.

Nous souhaitions que ce rapport résume de manière satisfaisante nos connaissances acquises sur le Machine Learning.