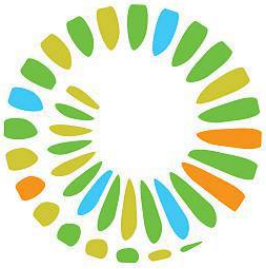
École nationale supérieure d'électrotechnique, d'électronique, d'informatique, d'hydraulique et des télécommunications

Projet long

Rapport technique

2020-2021

Deep clustering pour le diagnostic médical des cancers: transfert et adaptation de caractéristiques générales sans supervision dans les images de microscopie.



INSTITUT UNIVERSITAIRE
DU CANCER DE TOULOUSE
Oncopole

Sommaire

1 Introduction

Dans ce projet axé sur le transfert des connaissances générales obtenues pour l'analyse d'un domaine non-médical vers le domaine de l'histopathologie, le défi initiale est le manque de bases de données suffisamment grandes et bien annotées. Pour résoudre ce probléme, il y'a eu plusieurs approches d'apprentissage non supervisé, notamment des approches de deepclustering, pour implementer ce dernier; plusieurs choix de conceptions sont possibles soit au niveau du modèle en question, l'architecture de ce modèle et le code d'implémentation. Dans ce rapport technique, on justifie nos choix d'implémentations et on montre nos résultats obtenus.

2 Choix d'implémentations.

2.1 Deep Cluster et pourquoi pas un autre algorithme?

2.1.1 Deep Clustering

Le Deep Clustering est une approche d'apprentissage profond (Deep learning) complètement non supervisée pour regrouper des données de grande dimension. Dans le clustering, nous essayons de regrouper un ensemble de points de données en plusieurs clusters en fonction de leurs similitudes inhérentes. Contrairement à la classification, où on donne un label à chaque point de données et où on forme le modèle en fonction des labels, on n'a pas besoin d'informations sur les labels dans le clustering. Il existe de nombreuses méthodes de clustering comme le clustering avec K-means, le clustering hiérarchique et le clustering basé sur la densité, qui sont fiables lorsque nous travaillons avec des données de faible dimension. Cependant, ils ne sont pas adaptés pour des données de haute dimension comme les images.

Ainsi, lorsque nous traitons des données de grande dimension, nous pouvons effectuer une réduction de dimension en utilisant l'analyse de composant principal ou l'auto encodeur pour extraire les caractéristiques importantes. Ensuite, nous pouvons utiliser K-means pour regrouper les caractéristiques de faible dimension. Une autre solution consiste à utiliser un pipeline de bout en bout qui peut simultanément extraire des représentations de caractéristiques et attribuer des étiquettes.

Les frameworks de Deep Clustering combinent l'extraction de fonctionnalités, la réduction dimensionnelle et le clustering dans un modèle de bout en bout, permettant aux réseaux neuronaux profonds d'apprendre des représentations appropriées pour s'adapter aux hypothèses et critères du module de clustering qui est utilisé dans le modèle. Cela réduit la nécessité d'effectuer un apprentissage multiple ou une réduction de la dimensionnalité sur de grands ensembles de données séparément, au lieu de cela, on l'intègre dans la formation sur le modèle.

Plus que nos connaissances dans la pathologie sont presque inexistantes, l'utilisation d'un modèle qui n'utilise pas des connaissances préalables semble un choix raisonnable et puisque la base de données "colorectal histology" est une base relativement petite, Deep Cluster semble le choix parfait.

2.1.2 Architectures utilisées : VGG16 , ResNet50

VGG Neural Networks VGG16 est un modèle de réseau neuronal convolutif proposé par K.Simonian et A. Zisserman de l'université d'Oxford dans l'article " Very Deep Convolutional Networks for Large-Scale Image Recognition".

C'est une amélioration par rapport à AlexNet, en remplaçant les grands filtres de la taille d'un noyau par plusieurs filtres 3 x 3 de la taille d'un noyau l'un après l'autre.

Alors que les dérivés précédents d'AlexNet se concentrent sur des tailles et des foulées plus petites dans la première couche convolutionnelle, VGG aborde un autre aspect très important des CNN qu'est la profondeur. Passons en revue l'architecture de VGG.

Input: VGG prend une image RVB de 224x224 pixels. On peut découper le patch central 224x224 dans chaque image pour garder la taille d'image d'entrée cohérente si la taille initiale n'est pas adaptée.

Convolutional Layers: Les couches convolutionnelles de VGG utilisent un champ réceptif très petit (3x3, la plus petite taille possible qui capture encore gauche/droite et haut/bas). Il existe également des filtres de convolution 1x1 qui agissent comme une transformation linéaire de l'entrée, qui est suivie par une unité ReLU. La stride de convolution est fixée à 1 pixel afin que la résolution spatiale soit préservée après la convolution.

Fully-Connected Layers: VGG a trois couches entièrement connectées : les deux premières ont 4096 canaux chacune et la troisième a 1000 canaux, 1 pour chaque classe.

Hidden Layers: Toutes les couches cachées de VGG utilisent ReLU (Rectified Linear Units). L'avantage de ReLU réside dans le temps d'entraînement ; un CNN utilisant ReLU peut atteindre une erreur de 25 % sur l'ensemble de données CIFAR-10 six fois plus vite qu'un CNN utilisant tanh.

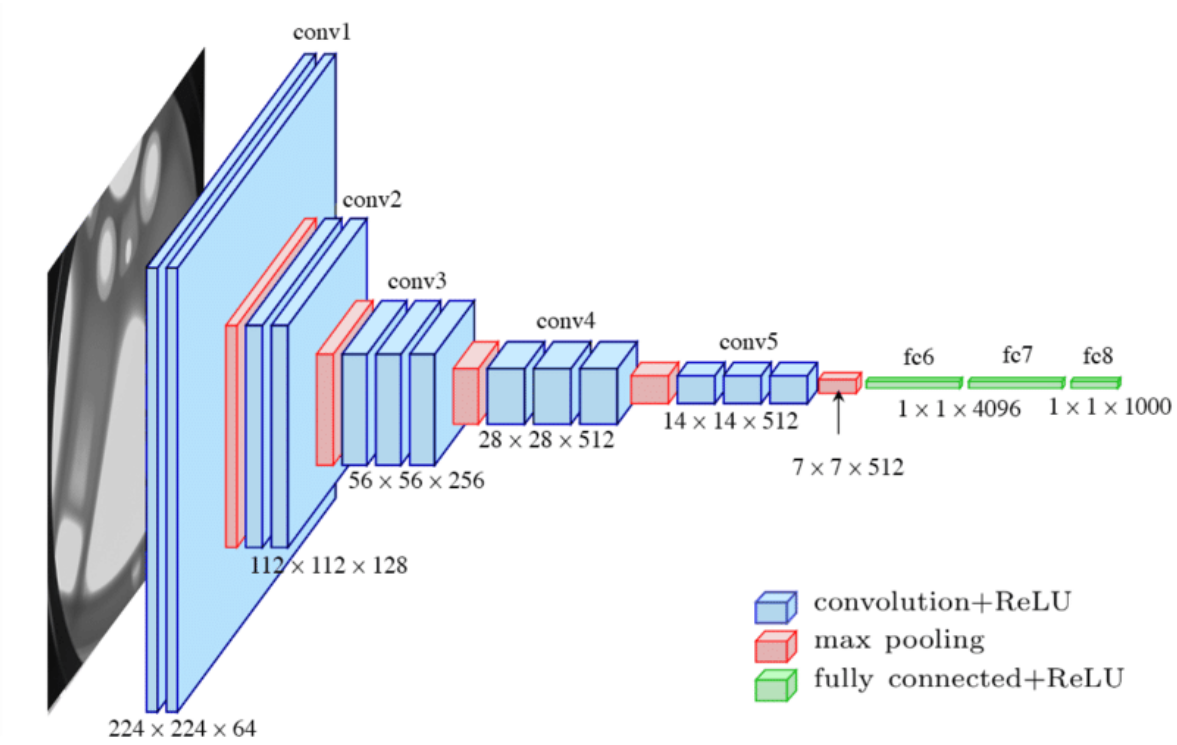


Figure 1 – Architecture du VGG16.

ResNet 50 est un réseau neuronal convolutionnel qui a 50 couches de profondeur. C'est une solution à un problème de dégradation qui se produit lorsque les réseaux plus profonds commencent à converger. Cela conduit à la saturation de la précision et à sa dégradation rapide. Les couches additionnelles des réseaux les plus profonds ont une meilleure approximation du mapping et réduisent énormément l'erreur.

Lorsque des réseaux profonds commencent à converger, un problème de dégradation se produit. La profondeur du réseau augmente, la précision est saturée et se dégrade rapidement

Dans les réseaux profond, les couches supplémentaires se rapprochent mieux du mapping que de sa contrepartie moins profonde et réduisent l'erreur d'une marge significative.

L'une des forces de ResNet est sa capacité à sauter quelques connexions. Les avantages de ce concept sont de nous permettre d'avoir des raccourcis pour que le gradient circule et d'atténuer ainsi le problème de la disparition du gradient. Il permet également au modèle d'apprendre une fonction d'identité qui garantit que la couche supérieure fonctionnera au moins aussi bien que la couche inférieure.

2.2 Conception du code

2.2.1 Importation de la dataset

Nous avons commencé par importer le dataset "colorectal histology" de tensorflow datasets, en précisant le chemin des données , que c'est un dataset d'entraînement via l'argument split="train" , en activant le paramètre shuffle permettant de mélanger la dataset téléchargée , ainsi que le paramètre info permettant de récupérer les métadonnées du dataset.

2.2.2 La Dataset

La datasat contient des images (150 x 150 x 3) appartenant à 8 classes différentes. Cette dataset comporte 5000 images divisées d'une manière équitable sur ces classes (625 images par classe).

Ces classes sont :
0 - tumor
1- stroma
2- complex
3- lympho
4- debris
5- mucosa
6- adipose
7- empty

2.2.3 Importation des modules nécessaires

Après avoir charger le dataset dans le chemin voulu , On importe les modules nécessaires pour travailler. Ces modules sont: tensorflow.keras.applications.resnet50
tensorflow.keras.preprocessing
tensorflow.keras.applications.resnet50
preprocess..input
decode..predictions
numpy
matplotlib.pyplot
matplotlib.image
keras
keras.models
ImageDataGenerator
optimizers
sklearn.preprocessing
sklearn.cluster
sklearn.decomposition

keras.layers
sklearn.manifold
pandas
seaborn
google.colab
tensorflow.keras.initializers
random
keras.utils.np..utils
klearn.metrics

2.2.4 Visualisation du dataset

La fonction **show_examples** provenant du module tensorflow_datasets permet de visualiser quelques données .

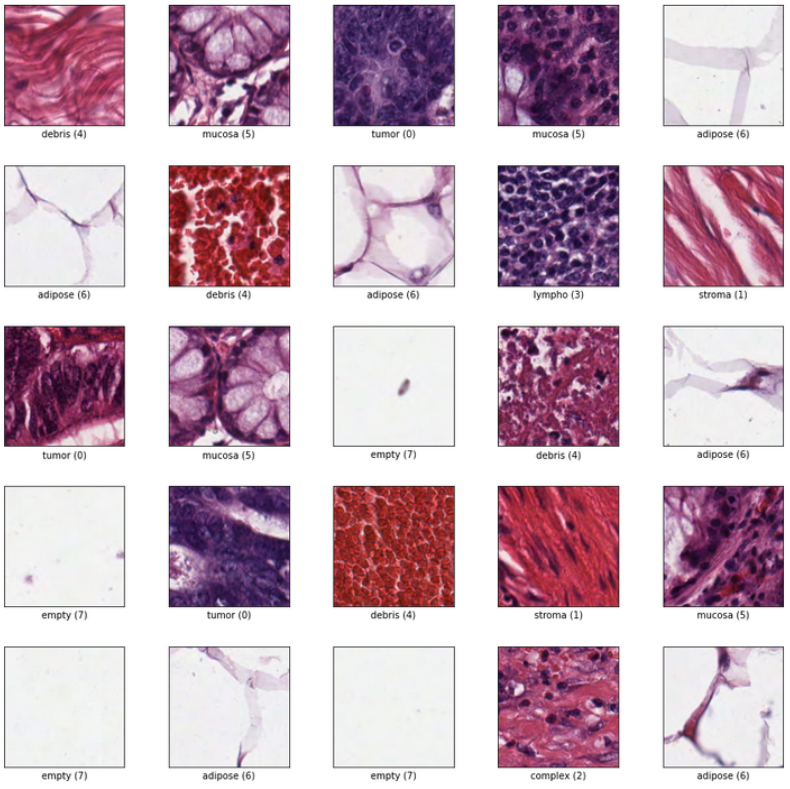


Figure 2 – Un exemple de 25 échantillons aléatoires pris du dataset (5 lignes , 5 colonnes) avec leurs labels respectifs.

2.2.5 Prétraitement du dataset

L'idée est de normaliser nos images. Ce traitement est fait grâce à la fonction train_datagen.flow_from_directory ,qui prend en paramètre le chemin de notre data , la taille des images voulues , le batch size , et finalement le paramètre class mode="categorical" .

2.2.6 Création du réseau

Comme nous avons dit avant , notre point de commencement sera un réseau déjà pré entraîné , qui sera dans notre cas un réseau qui se constitue des layers soit de VGG 16 , soit de ResNet 50 , à qui nous avons supprimé la dernière couche , et qui a été initialisé par les poids de imagenet , et à qui nous avons ajouté une layer de average pooling ,qui permet de redimensionner la sortie du réseau.

2.2.7 Algorithme de Clustering

Dans cette étape , on commence par extraire les features. L'idée est de boucler sur la data obtenue après le prétraitement , pour récupérer les images , les labels et enfin les features, qui sont les prédictions de notre réseau des images du dataset.

La prochaine étape est de donc réduire la dimension des features en utilisant l'ACP.

Comme on peut le voir sur le notebook , la taille des données avant L'ACP est (5000,2048).

L'ACP réduit donc effectivement la taille de la data , puisqu'elle la fait passer à (5000,412).

Pour L'ACP nous avons pris: variance = 0.98

Il est maintenant le temps d'utiliser un algorithme de Clustering.

Dans notre cas , nous avons décidé d'utiliser Kmeans et Birch , qui sont tous les deux disponibles en Sklearn , puis comparer l'efficacité de chacun d'eux.

Pour KMeans , les paramètres choisis sont les suivants: init = "k-means++ n_clusters = 8 n_init =2

Pour BIRCH , les paramètres choisis sont les suivants: n_clusters=8

Première itération:

Nous commençons par diviser notre dataset en deux parties : Entraînement et test.

La partie d'entraînement contient 4000 images. La partie du test contient 1000 images.

On commence par créer un modèle à partir du réseau précédent , en lui ajoutant à la fin une couche Dense qui prend 8 comme hyper paramètre (Nombre de clusters), et softmax comme fonction d'activation.

Pour cette première itération, on prend le modèle tel qu'il est sans réentrainement. Ce modèle sera sauvegardé pour être utilisé dans la prochaine itération.

2.2.8 Les itérations du Deep Cluster

On commence par charger le modèle de l'itération précédente ainsi que ses labels, puis on crée le nouveau modèle via l'ajout d'une layer average pooling au modèle précédent.

La prochaine tâche est l'extraction des features du nouveau modèle , identique à celle déjà expliquée dans les parties ci-dessus.

Puis , on applique l'ACP a nos features , ainsi qu'un algorithme de clustering (Birch ou Kmeans) aux nouvelles features.

Afin de s'assurer du bon comportement de notre algorithme , on envisage toujours la visualisation des clusters issus de ces nouveaux labels en utilisant la fonction TSNE plot , de telle façon que les clusters doivent être bien séparés et uniformément distribués.

On calcul après le score nmi qui compare les labels de l'itération d'avant avec les labels actuels et qui doit être normalement une fonction croissante.

Fine tuning:

Dans cette partie, on charge le modèle issu de la première itération,On lui ajoute une layer Dense d'hyper-paramètre nombre de clusters (8 dans notre cas)

On commence maintenant la phase d'entraînement, en prenant sgd ou RMSEPropcomme optimizer. On calcule la matrice de confusion, ainsi que le score de performance. Puis on finit par sauvegarder le modèle.

3 Contraintes rencontrées

La phase d'implémentation a connu le succès après avoir résolu plusieurs contraintes, qui risquent la mise en place du produit final.

Les problèmes rencontrés sont de nature technique, concernant le choix de paramètres et de techniques pour implémenter et optimiser le travail, la consommation de la mémoire vive et la complexité temporelle.

D'abord, on avait un problème fatal à résoudre qui risque de tout planter et cesser l'exécution de notre composition, concerne la consommation inarrêtable et terrible de la mémoire vive (environ 12 GB dans les environnements d'exécution hébergés par Google).

Ce problème intervient instantanément lors de la phase d'extraction des caractéristiques [Feature extraction], qui est une étape fondamentale du processus de reconnaissance, préalable à la classification, surtout pour lors de l'utilisation du modèle pré-entraîné ResNet-50, qui a pour sortie des éléments de taille (x, 5, 5, 2048) qui est largement grand et massive.

Ce problème peut se propager aussi lors d'utilisation du modèle VGG-16 durant la phase d'adaptation du domaine et surtout lors de l'extraction des descripteurs à passer pour adapter notre réseau conventionnel. Pour en résoudre, on a décidé d'ajouter à la sortie du réseau convolutif une couche de pooling qui réduit la taille spatiale d'une image intermédiaire, réduisant ainsi la quantité de paramètres et de calcul dans le réseau, et crée aussi une forme d'invariance par translation. Plus précisément, on utilise en sortie une couche de type "average pooling" qui donne en sortie la moyenne des valeurs du patch d'entrée, ce qui donne des sorties vectoriels de taille considérablement moyenne de l'ordre (x,2048) pour ResNet-50, au lieu de (x, 5, 5, 2048).

De surcroît, on a introduit les moyens d'analyse en composante principale (ACP), qui permet de réduire la dimensionnalité de notre problème suivant la direction des composants principaux; Cela rend les grands ensemble de données (datasets) plus simples, faciles à explorer et à visualiser. En outre, on réduit la complexité de calcul du modèle, ce qui accélère l'exécution des algorithmes d'apprentissage. Dans ce contexte, l'application de ces techniques sur les caractéristiques extraits des images de notre base de donnée, sur le cas d'usage de l'architecture ResNet-50 permet de réduire d'une manière efficace la dimensionnalité du problème, comme le montre ces détails :

Nombre de composants avant l'ACP	2048 (Sortie du réseau ResNet-50 avg = "pooling")
Nombre de composants avant l'ACP avec une variance de 98%	412

En résultat, ces mesures prises ont permis de bien réduire la dimensionnalité du problème tout en gardant les caractéristiques des données, et réduire aussi le temps d'exécution de notre modèle d'apprentissage.

Par ailleurs, on a rencontré d'autres contraintes techniques, qui concernent le choix des paramètres, outils et méthodes qui améliorent l'apprentissage par classification. Par exemple, lors de la phase d'entraînement du réseau de neurone, le choix important des optimiseurs. Ces derniers sont des algorithmes ou des méthodes utilisés pour modifier les attributs de votre réseau de neurones tels que les poids et le taux d'apprentissage afin de réduire les pertes.

Ces algorithmes jouent un rôle très important concernant le comportement des modèles et des méthodes d'apprentissage et de classification. Dans ce contexte, l'un des actions qui risquent la tâche d'apprentissage est le fait de tomber sur le cas d'un extremum local, qui bloque et boucle notre procédure d'apprentissage. Par conséquent, on risque d'avoir des précisions [accuracies] faibles, et autres problèmes, genre overfitting qui reflètent l'ignorance considérable d'un réseau.

Pour en faire, on a décidé lors de la phase d'implémentation d'appliquer l'un des optimiseurs suivants dans nos architectures :

Stochastic Gradient Descent (SGD): qui est une variante de l'algorithme du gradient descendant, et qui permet de mettre à jour les paramètres d'un modèle après le calcul de la perte lors de chaque exemple d'entraînement. Ce choix nous permet de garantir la mise à jour fréquente et permanente des paramètres du modèle, obtenir de nouveaux extremum à chaque exécution, conserver la mémoire grâce à la non-conservation de la fonction de perte, et converger rapidement.

RMSprop: qui est une méthode qui permet d'adapter le taux d'apprentissage selon l'exécution. Cette technique est très importante, car elle permet de bien contrôler la convergence de notre apprentissage, en termes du temps, permet de maintenir une moyenne mobile (actualisée) du carré des dégradés, et permet d'éviter des cas particuliers et triviaux, voire le cas où la condition initiale correspond à un extremum

local. Ce choix à comme avantage majeure la rapidité de convergence tout en assurant d'éviter des cas fatales (comme mentionné au-dessus).

Enfin, un autre problème qui porte sur l'orientation de notre implémentation, concerne d'une manière majeure le choix des couches à entraîner et à stopper pour évoluer l'apprentissage du modèle. En effet, stopper des couches cachées [hidden layer] permet de bien optimiser et accélérer l'apprentissage du réseau de neurone. Dans notre cas, où on fait un apprentissage par transfert, la première couche du réseau est figée généralement tout en laissant les couches d'extrémité ouvertes à la modification. Notre choix d'implémentation dans ce cas était de choisir entre deux cas qui étaient bien répéter dans la littérature. Soit dans un premier lieu, on freeze les layers du modèle construit jusqu'à le bloc 4 ou 5 de la base conventionnelle et entraîner le reste avec une procédure spécifique.

En détails, on a essayé d'entraîner les layers dense (en-tête) du réseau au début, et après tout entraîner comme prévu. Ceci permet de bien contrôler l'optimisation, la convergence et la variation des poids de telle façon d'éviter les cas de divergence et d'extrema local. D'autres côté, une autre technique est de laisser tous les layers s'entraîner, sachant qu'on va utiliser un initialiseur [initialize] des couches qui permet de contrôler l'initiation et la variation de ses poids, plus précisément, on utilise keras.initializers.RandomNormal comme initializer, avec qui varie les paramètres et les poids de notre couche suivant une loi normale centrée, cette méthode est aussi efficace, car elle permet de bien optimiser et contrôler l'apprentissage du réseau. Pour conclure, ces deux méthodes sont efficaces pour accélérer et optimiser notre travail en ce qui concerne le fine tuning du réseau.

4 Resultats

Après avoir exécuté la boucle d'apprentissage et amélioré notre réseau de neurones, nous pouvons visualiser et analyser la classification de nos clusters et la distribution des clusters des architectures VGG16 et ResNet.

Pour cette fin, on utilise dans un premier temps l'algorithme T-SNE afin de réduire la dimension des features et pouvoir visualiser les dimensions. Dans un deuxième temps on utilise la matrice de confusion pour analyser les erreurs relatives à chaque classe et calculer la précision totale et celle de chaque classe.

4.1 TSNE PLOTS:

Dans un premier temps, nous traçons le graphe des deux dimensions données par l'algorithme TSNE pour la première itération, ensuite, une autre figure après plusieurs itération dans le but de comparer et de savoir si le réseau neurone entrainé se

développe.

VGG16 : Dans la première itération, nous obtenons des clusters aléatoires et chaotiques.

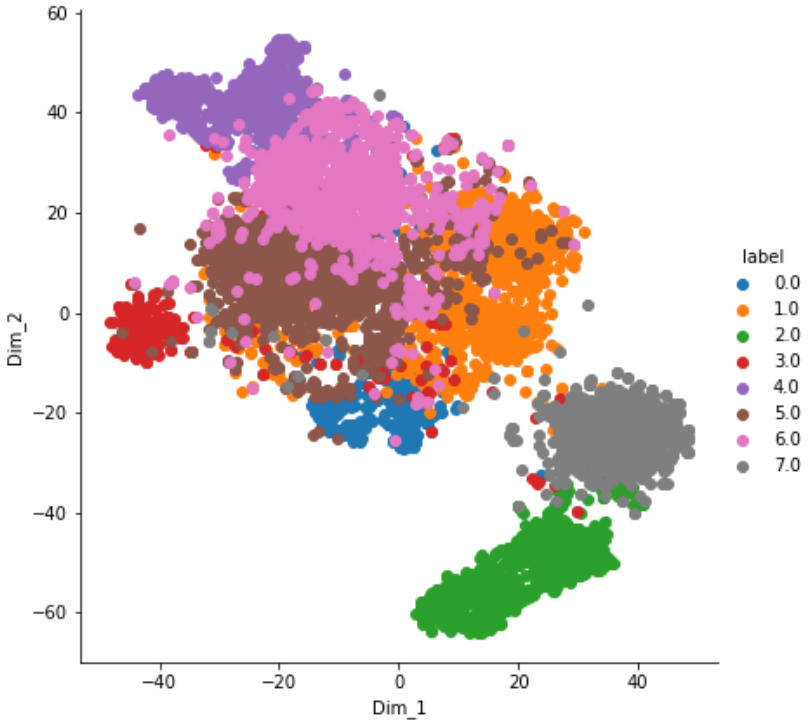


Figure 3 – Premiére itération du VGG16 suivi par le Kmeans.

nous pouvons conclure que notre réseau de neurones n'a pas appris suffisamment de paramètres sur les données et doit être ré-entraîné.
Pour la quinzieme itération:

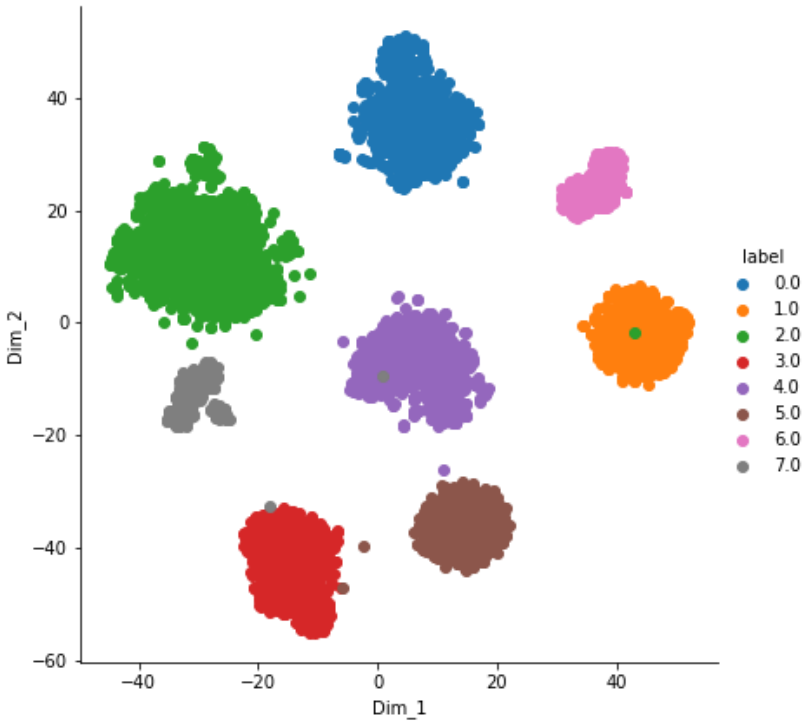


Figure 4 – 15éme itération du VGG16 suivi par le Kmeans.

Après 15 itérations, nous pouvons voir que notre réseau parvient à bien séparer les différentes classes de la base de données "colorectal histology"

conclusion: Le Transfer learning est possible, il suffit de bien choisir ses hyper-paramètres afin d'obtenir de meilleurs résultats.

ResNet50:
Dans la première itération, nous avons obtenu des clusters aléatoires et chaotiques.

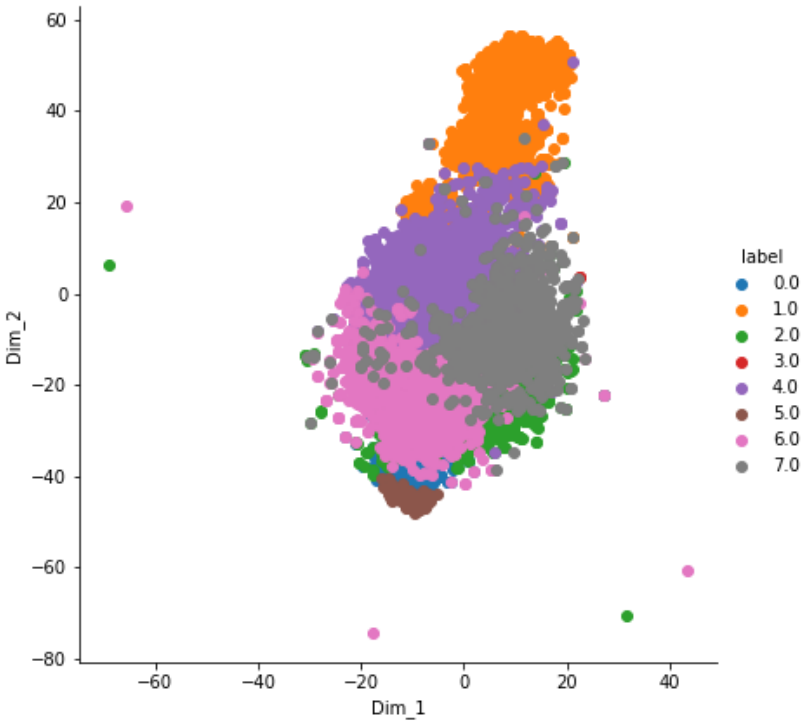


Figure 5 – La premiére itération du ResNet50 suivi par le Kmeans.

ils ne sont même pas séparés, cela montre que la classification de notre réseau est aléatoire.

Pour la 10émé itération:

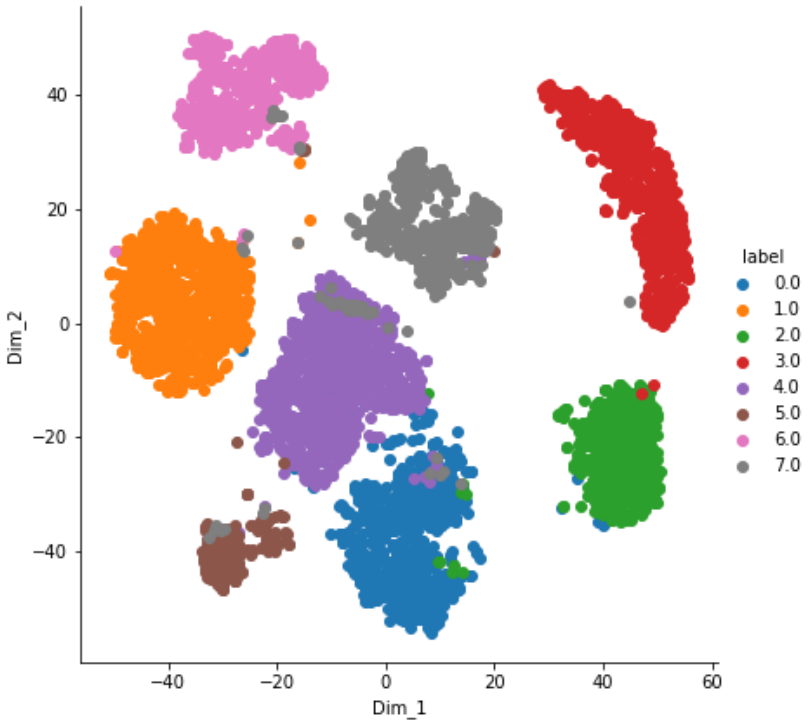


Figure 6 – La 10éme itération du ResNet50 suivi par le Kmeans.

nous pouvons voir que notre réseau parvient à bien séparer les différentes classes de la base de données "colorectal histology".
Conclusion:

Le Transfer learning est possible, il suffit de bien choisir ses hyperparamètres afin d'obtenir de meilleurs résultats.

VGG16 vs ResNest (KMeans):

En outre, nous avons exécuté l'algorithme du clustering avec différents nombres d'itérations comme l'exemple pris pour les figures précédentes 10 pour resnet et 15 pour vgg16. Ceci est en fait directement lié à l'exécution du temps. En fait, l'architecture ResNet50 nous prend beaucoup de temps à mettre à jour les poids des neutrons par rapport à VGG16. Ceci est dû au fait que l'architecture du ResNet 50 est plus profonde que celle du VGG16.

4.2 Matrice de confusion

Nous mesurons également notre résultat en calculant une matrice de confusion et en calculant la précision totale et celle de chaque classe.

En général, une matrice de confusion est un résumé des résultats de prédiction sur des problèmes de classification, elle montre le nombre de prédictions correctes et erronées pour chaque cluster. Les vraies prédictions sont placées dans la diagonale de cette matrice et les mauvaises prédictions sont dans les autres cas. Cette matrice est généralement dense.

Les figures suivantes sont les résultats de nos prédictions sur un ensemble de tests de 1000 images.
VGG16

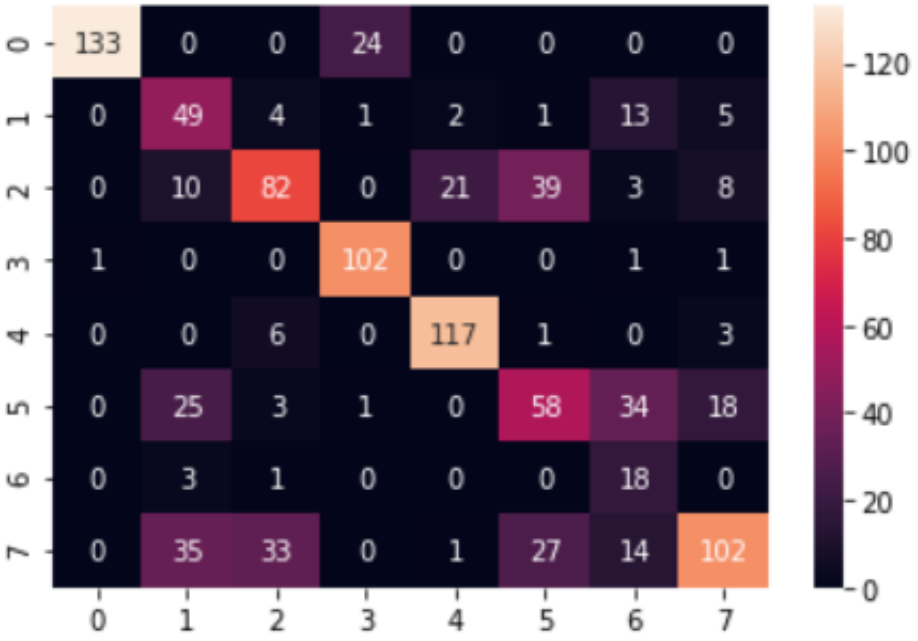


Figure 7 – Matrice de confusion pour l'architecture VGG16

Ensuite, On calcule le taux de réussite de reconnaissance pour chaque cluster, nous obtenons:

Cluster	0_	1_	2_	3_	4_	5_	6_	7_
Accuracy	84,71%	65,33%	50,30%	97,14%	92,12%	41,72%	81,81%	48,11%

Nous pouvons voir qu'il y a des résultats presque parfaits tels que 92,12% pour le cluster 4 et des résultats insatisfaisants, tels que 41,72% pour cluster 5. **ResNet50**

Figure 8 – Matrice de confusion pour l'architecture ResNet50

De même pour ResNet50, on calcule le taux de réussite de reconnaissance pour chaque cluster, nous obtenons:

Cluster	0_	1_	2_	3_	4_	5_	6_	7_
Accuracy	93,68%	86,15%	40,01%	57,29%	51,74%	77,45%	69,76%	35,21%

VGG16 vs Resnet : Nous obtenons pour l'architecture VGG16 une précision totale de 66,10%. Et pour ResNet50 une précision totale de 61,70%
Pour pouvoir bien comparer les résultats il faut recourir à des études statistiques, pour cela on calcule la précision totale, cette fonction "accuracy score" est déjà prédéfinie et trouvée dans la bibilothéque sklearn.metrics. En résumé, nous pouvons voir que la précision globale pour les deux architectures est supérieure à 60%, ce qui est prometteur pour d'autres travaux axés sur le transfert des connaissances générales obtenues pour l'analyse d'un domaine non-médical vers le domaine de l'histopathologie Cependant, on peut dire que VGG16 est efficace par rapport à resnet-50, en termes de consommation mémoire et de rapidité.

4.3 Comparaison entre les algorithmes de clustering Kmeans et BIRCH pour une architecture VGG16.

Pour l'algorithme de classification BIRCH, le model DeepCluster avec une architecture VGG16, on obtient pour la visualisation des clusters aprés 15 itérations la figure suivante:

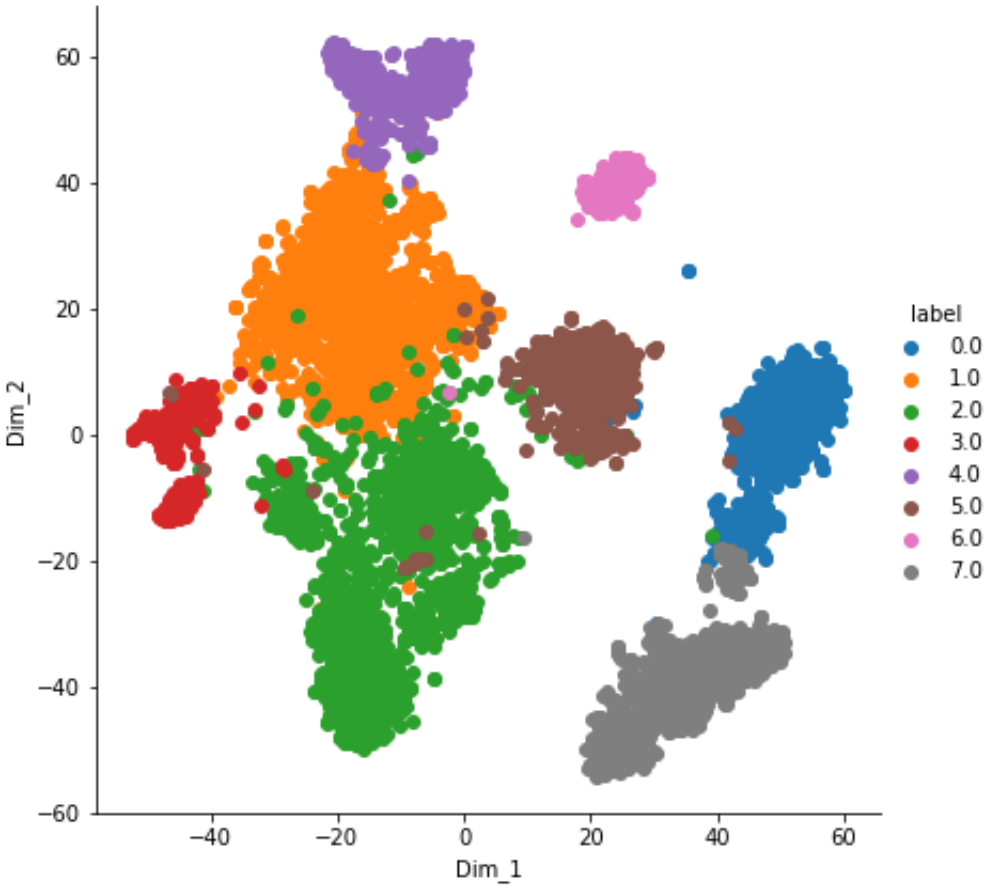


Figure 9 – 15éme itération du VGG16 suivi par le BIRCH

Et pour la précision totale, on obtient 60,0%.

Conclusion:

Pour l'algorithme de classification Kmeans, on avait obtenu une précision totale de 66.10% comme mentionné précédemment, alors que pour l'algorithme Birch on obtient que 60,0%, De plus, on remarque que l'algorithme Kmeans était plus efficace en termes de rapidité. Ainsi Kmeans reste plus efficace que l'algorithme de Birch dans notre cas d'étude.

5 Conclusion

Ce projet nous a permis d'explorer le monde du Deep learning, de voir concrètement le fonctionnement des réseaux neuronaux et de pouvoir appliquer nos connaissances sur un projet de pathologie digitale. On a eu la chance de se pencher sur des variantes du Deepclustering, en particulier le DeepCluster du papier de Caron et al., de découvrir plusieurs algorithmes de clustering et d'améliorer notre appréhension du sujet. On a pu atteindre plusieurs de nos objectifs à travers ce projet, et on espère faire face à d'autres problèmes de ce genre.