Université de Mons

Faculté des Sciences

Utilisation de « Self Supervised Learning » dans le contexte de l'imagerie médicale

Présenté par **Guns Antoine**

Sciences informatiques

Dupont Stéphane

Ben Taieb Souhaib

Année Académique 2024-2025

L’auteur, Guns Antoine, atteste avoir respecté les règles éthiques en vigueur, y compris la charte de l’université relative à l’utilisation de l’intelligence Artificielle.

Table des matières

Introduction

Depuis plusieurs années maintenant, l'intelligence artificielle se développe dans différents domaines. On peut citer la création d'images, les voitures autonomes, la détection d'objets dans une image et même les grands modèles de langage qui ont, avec chat GPT, bouleversé la vie de milliers, voire de millions de personnes dans le monde. Ce développement rapide et fulgurant de l'intelligence artificielle ainsi que l'apparition de celle-ci dans notre vie inaugurent une avancée importante de nos capacités technologiques.

Mais qu'est-ce que l'intelligence artificielle ?

On pourrait prendre pour réponse à cette question l'avis de John McCarthy, pionnier dans le domaine de l'intelligence artificielle.

« *Il s'agit de la science et de l'ingénierie de la fabrication de machines intelligentes, en particulier des programmes informatiques intelligents »[39]*

Seulement voilà, certaines innovations sont davantage utilisées que d'autres, comme c'est le cas des outils/logiciels tels que chat GPT ou DALL.E2. Même si ceux-ci sont très impressionnants et relevaient encore de la fiction il y a une dizaine d'années, qu'en est-il de leur utilité réelle pour notre société ? Cette réponse va varier d'individu en individu mais si je devais donner un avis, il s'orienterait actuellement vers l'idée qu'ils n'apportent pas suffisamment à l'humanité comparé à l'effort investi. De plus, pour pouvoir entrainer ces mastodontes, il faut qu'ils consomment toujours plus de données mais ces données ne sont pas infinies et demandent beaucoup de temps pour être utilisables, que ce soit en recherche ou en obtention de résultats.

Néanmoins, il existe plusieurs branches de L'IA qui permettent en partie de résoudre ce problème, comme l'apprentissage auto-supervisé ou SSL (Self-Supervised Learning) qui est justement le sujet de ce travail de mémoire. Celui-ci est une méthode assez innovante, qui grâce à des idées nouvelles, parvient à apprendre sans l'utilisation de labels grâce à diverses techniques utilisant l'image en elle-même comme label. Même si le développement de l'IA crée des applications non indispensables pour l'instant, il ne faut pas oublier que l'IA peut et a déjà apporté dans certains domaines des avancées significatives. On pourrait citer notamment les voitures autonomes qui, même si imparfaites aujourd'hui, pourraient donner de bons résultats dans un futur proche.

Un autre domaine qui nous est vital est celui de la santé, il existe déjà des applications de l'IA dans plusieurs spécialités notamment celles des diagnostics. Cependant, la plupart des modèles sont en apprentissage supervisé, ils demandent donc des données qui ont nécessité un traitement et ce traitement nommé « l'étiquetage » prend du temps et coûte cher.[8][30]

Par conséquent, on se retrouve avec une belle quantité de données non labellisées que nous pouvons considérer comme inutiles pour un modèle supervisé. Mais on peut utiliser des technologies comme le SSL pour traiter ces montagnes de données et créer des modèles utilisables pour certaines tâches, comme les diagnostics de maladies ou la détection de tumeurs.

On peut alors se poser une question : *l'utilisation de divers modèles SSL dans un contexte médical peut-il fonctionner ?*

Il s'agit de la question que ce travail tente d'éclaircir en réalisant une analyse et des tests des différents modèles avec des « datasets » d'imagerie médicale. Il s'agit aussi de savoir si le SSL peut être utile dans ce contexte.

1 Réseaux de neurones : concepts et fonctionnement

Cette courte section introduira les réseaux de neurones et la section suivante sera consacrée à une explication détaillée des réseaux de neurones convolutifs. Les équations utilisées pour l'exemple seront celles du livre MIT Deep Learning Book[41], ainsi que celles implémentées par Pytorch. Cette partie parle des réseaux de neurones dans un contexte d'entrainement supervisé et donc avec des données labellisées.

1.1 Réseaux de neurones, fonctionnement

Cette a pour source : [41], [47]

Avant d'introduire les réseaux de neurones convolutifs, il est essentiel de rappeler d'abord le fonctionnement des réseaux de neurones traditionnels puisque les réseaux convolutifs en sont une extension. Leur fonctionnement tente de « répliquer » ceux des neurones qu'on peut trouver dans le monde animal. Le réseau est constitué de plusieurs couches contenant chacune plusieurs neurones, ceux-ci ont une sortie qui est reliée aux couches supérieures ainsi que plusieurs entrées qui sont les sorties des neurones de la couche inférieure.

La structure d'un neurone est détaillée comme suit.

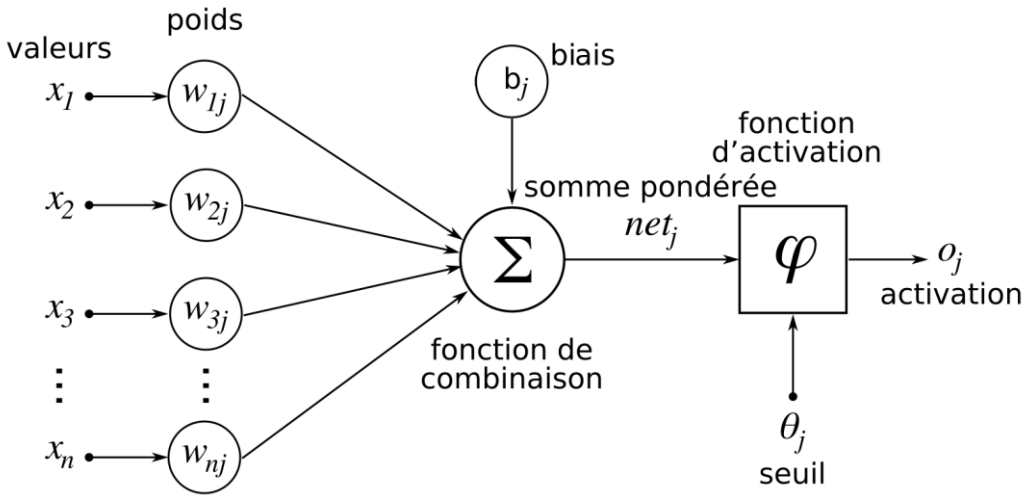


Figure 1.1 : fonctionnement d'un neurone.

- Dans ce cas général, les données $x_1 \dots x_n$ représentent des valeurs d'entrée.

- Chaque valeur d'entrée possède un poids ($w_{1j}, \dots, w_{nj}$), celui-ci pourra être modifié pour ajuster la sortie du neurone. Ces valeurs peuvent être initialisées de différentes manières, par exemple de manière aléatoire.
- Le neurone possède une fonction de combinaison, elle permet de transformer toutes les valeurs des entrées en une valeur de sortie (net_j) grâce à une opération (ici une somme).
- Le neurone possède également un paramètre de biais, celui-ci est utilisé pour changer le résultat de la fonction d'activation : $net_j = w_{1j}x_1 + w_{2j}x_2 + \cdots + w_{nj}x_n + b_j$
- Le neurone possède finalement une fonction d'activation(ϕ), il en existe plusieurs, leur rôle est de donner une valeur de sortie en fonction bien souvent d'un seuil d'activation (ϑ_j) : ce seuil est lié à la fonction utilisée (certaines fonctions ne possèdent pas de seuil d'activation).
- Elle permet également d'introduire de la non-linéarité sinon le réseau de neurones pourrait être simplifié à une combinaison linéaire. (Ce qui limiterait grandement ses capacités).

Un exemple de fonction avec une seuil d'activation est la fonction sigmoïde : $f(x) = \frac{1}{1+e^{-x}}$.

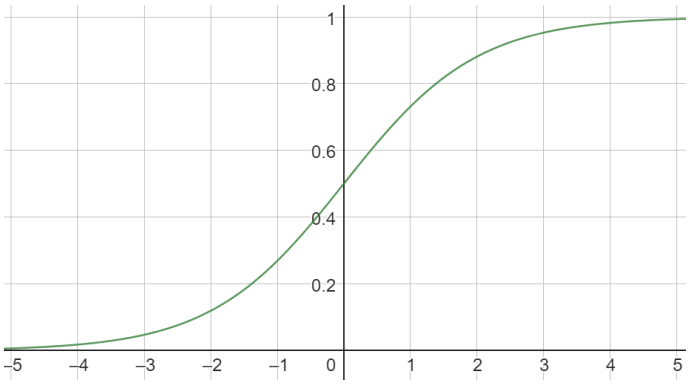


Figure 1.2 : représentation de la fonction sigmoïde où le seuil d'activation est de 0.

Ce seuil d'activation va définir le comportement de la fonction et donc décider de la sortie. Par exemple pour sigmoïde si l'entrée est inférieure au seuil d'activation (0) alors le résultat ne pourra pas être supérieur à 0.5 et à l'inverse, si l'entrée est supérieure à 0, le résultat ne pourra jamais être en-dessous de 0.5. Toutes les fonctions ne possèdent pas forcément de seuil d'activation.

La structure d'un réseau de neurones est composée de 3 parties, un vecteur d'entrée (parfois appelé couche d'entrée) servant à introduire les données brutes, une ou plusieurs couche(s) cachée(s), ce sont elles qui permettent aux réseaux de neurones d'apprendre et enfin une couche de neurones de sortie, donnant le résultat du réseau.

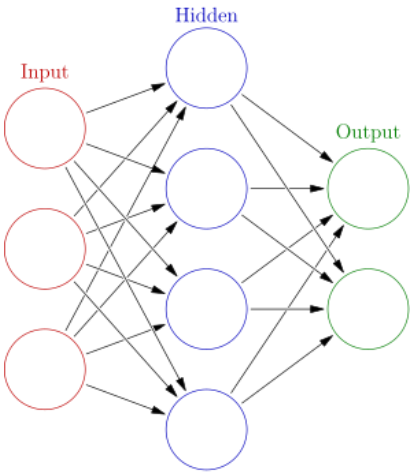


Figure 1.3 : structure d'un réseau de neurones avec une couche cachée.

Cette structure ne permet cependant pas d'apprendre quelque chose, du moins pas encore. Il manque quelques éléments importants : propagation avant, fonction de perte et propagation arrière.

D'abord, la propagation avant ou « forward propagation » est la plus simple. Il s'agit de la propagation des données depuis les entrées du réseau jusqu'à la ou les sortie(s) en passant par la ou les couche(s) cachée(s). Les données d'entrée vont se propager couche par couche au neurone via la sortie des neurones de la couche ultérieure. Une fois la dernière couche atteinte, le réseau va alors donner un résultat. On va ensuite utiliser la sortie donnée et la comparer à la valeur attendue via une fonction : « loss fonction » ou fonction de perte. Celle-ci représente à quel point la prédiction est proche de la vraie valeur. Il existe plusieurs fonctions de perte et chacune donnera un résultat différent. De plus, certaines sont cantonnées à des domaines précis comme la classification ou la régression.

Un exemple couramment utilisé pour la régression est la MSE ou l'erreur quadratique moyenne : $MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$

Pour n prédiction avec y_i la vraie valeur de la prédiction i et $\hat{y}_i$ la prédiction de i, on peut calculer le résultat de la fonction de perte sur n prédictions.

Plus la prédiction est proche de la vraie valeur, plus la valeur calculée sera proche de 0 dans le cas de la MSE (attention car le comportement de la fonction de perte varie et il est possible que 0 ne soit pas le résultat de celle-ci quand la prédiction et la vraie valeur sont égales) .

Dans le reste de cette partie, nous utiliserons une fonction de perte où le résultat d'une prédiction correcte est égal à 0. On en déduit assez simplement que l'objectif du réseau de neurones est simplement de minimiser cette fonction. Car la « loss fonction » représente l'écart entre la « vraie » valeur de notre exemple et celle de la valeur prédite et si cet écart est égal à 0 alors notre modèle va correctement prédire la valeur de sortie à partir de celle d'entrée. Il est donc logique qu'on utilise ce résultat pour améliorer le réseau de neurones (et donc les futures prédictions). Pour que ça fonctionne, il faut donc avoir une « vraie » valeur pour pouvoir la comparer à la valeur prédite, cette valeur varie selon le type de données utilisées : la vraie valeur peut être la donnée en elle-même ou alors, pour prendre l'exemple de la classification, cette valeur sera la classe de l'exemple. C'est ici qu'arrivent les données labellisées, le label représente la « vraie » valeur de la donnée, c'est comme ça que fonctionne l'apprentissage supervisé.

Par conséquent, il n'est pas possible de faire de l'apprentissage supervisé sans label car on ne sait pas comparer notre prédiction.

Une fois le résultat de la fonction de perte obtenu, celui-ci sera transféré au reste du réseau de neurones via la propagation arrière ou « backward propagation». Le but de cette opération est de pouvoir modifier les différents paramètres du réseau de neurones pour que celui-ci soit plus précis pour les prédictions futures (et pour ça, chaque neurone a besoin du résultat de la fonction de perte). On va donc utiliser un algorithme d'optimisation (ou «optimizer»), son objectif étant de minimiser la fonction de perte. Comme algorithme d'optimisation, nous pouvons prendre comme exemple celui de la descente de gradient ou « Gradient Descent ». Il va calculer le gradient de la fonction objectif (la fonction de perte) par rapport aux différents paramètres. Le gradient représente un vecteur dont la direction indique la variation la plus forte, c'est-à-dire quelle direction je dois prendre pour augmenter le plus vite la fonction. Par conséquent, nous allons prendre la direction inverse au gradient (vu que l'objectif est de diminuer la fonction de perte).

Ce processus va nous rapprocher légèrement de l'objectif et faire baisser le résultat de la fonction de perte. Mais cette méthode d'apprentissage ne permet pas d'arriver à des bons résultats avec seulement quelques données, ce processus est lent et est donc utilisé de manière itérative sur un ensemble de données (un « dataset »).

On arrêtera l'apprentissage dans plusieurs cas : soit car on a atteint le minimum global, ou un minimum local, soit car le modèle a atteint un certain nombre d'itérations.

On peut donner à « l'optimizer » [38] certains paramètres pour influencer l'apprentissage du modèle, l'un de ces paramètres importants est le « learning rate » ou le taux d'apprentissage [37]. Celui-ci représente la taille d'un « pas » que le modèle va faire pour se rapprocher de la solution (à chaque itération). Plus le « learning rate » est grand, plus le modèle va converger rapidement vers une solution potentielle. Il risque cependant de dépasser la solution mais aussi de ne pas pouvoir atteindre de solution précise. Plus le « learning rate » est faible plus on va pouvoir arriver à une solution précise, mais le modèle va mettre beaucoup plus de temps pour converger et risque de ne pas atteindre la solution. Le choix du « learning rate » est crucial pour que le modèle donne de bons résultats et il faut trouver un juste milieu entre précision et vitesse. Étant donné que celui-ci peut être modifié pendant l'entrainement du modèle, on peut utiliser des stratégies beaucoup plus complexes en le faisant notamment varier pendant l'entrainement.

On peut, par exemple, modifier le « Momentum » ou l'élan [37]. On peut le voir comme l'élan que va prendre le modèle durant l'entrainement quand celui-ci avance dans une même direction pendant un certain temps, il va alors prendre de la vitesse qui va lui permettre de converger plus vite vers un minimum et également d'éviter les minimas locaux. Il faut bien le choisir, si le « Momentum » est trop grand alors le modèle risque de ne pas s'arrêter dans un minima intéressant et s'il est trop faible le modèle ne va pas accélérer et faire comme s'il n'y avait pas de « Momentum ».

Pour entrainer et tester un réseau de neurones, il faut faire au minimum 2 sets de données du « dataset », c'est-à-dire qu'on va diviser le « dataset » en 2 parties, chacune possède ses propres données. Pour ce qui est de la taille des sets, le set d'entrainement sera plus important que le set de tests. Le set d'entrainement sert pour l'entrainement du réseau, le set de tests sert pour tester le réseau après son entrainement. On peut créer un autre set qu'on va appeler set de validation, celui-ci sera utilisé pour « surveiller » l'entrainement du modèle, c'est-à-dire ajuster les paramètres du modèle en fonction de son comportement.

Par exemple, si on remarque que le réseau apprend très lentement, on pourrait augmenter le « learning rate » ou si « l'accuracy » sur le set de validation se dégrade, on va alors arrêter l'apprentissage. Aussi, ce set peut servir à récolter des données sur l'entrainement du modèle.

L'entrainement du réseau de neurones consiste à lui fournir des données en entrée. En général, on va fournir au réseau de neurones, des données regroupées sous forme de batch. Le batch va représenter un sous ensemble des données d'entrainement d'une certaine taille définie par le « batch size » (il représente le nombre d'éléments dans le batch). On utilise cette méthode car entrainer le réseau image par image est beaucoup trop lent et ça conduit à des estimations du gradient qui sont très bruitées (la direction à suivre pour que le réseau s'améliore ne sera pas représentative d'assez de données et le modèle risque d'aller dans toute les directions). Également, entrainer le modèle avec l'intégralité du « dataset » d'entrainement n'est pas une bonne idée car celui-ci va aussi prendre énormément de temps. De plus, les paramètres ne seront que trop peu mis à jour et le modèle sera trop lent pour converger vers une solution. Le batch permet de mettre plus souvent à jour les paramètres du modèle sans être trop lent.

Un « Epoch » signifie que le modèle a été entrainé sur un passage complet du « dataset » d'entrainement. Un « Epoch » est divisé en b « itération », à chaque fin de celle-ci, les paramètres du modèle seront modifiés.

Pour obtenir le nombre d'itérations, on peut utiliser la formule suivante : $b = \frac{nb\ de\ donnée}{batch\ size}$)

Evidemment, il ne faut pas qu'une seule itération pour que le modèle puisse apprendre des données et paterne , il en faut plusieurs, en général, des dizaines voire plus (tout dépend aussi du nombre de données présentes pour l'entrainement).

Pour résumer, à chaque « Epoch » le modèle aura b itération durant lesquelles les données fournies en entrée seront propagées au reste du réseau via la propagation avant (la « forward propagation »). Une fois que le réseau a donné un résultat, il va utiliser la fonction de perte pour comparer sa prédiction avec la valeur attendue (la vraie valeur de l'élément). La propagation arrière (« backward propagation ») va alors propager ce résultat aux différentes couches du réseau pour que chaque neurone modifie ses paramètres via « l'optimizer ».

L'entrainement du réseau de neurones consiste à lui fournir beaucoup d'exemples pour qu'au fur et à mesure, le réseau puisse changer ses paramètres (poids, biais …) pour qu'après un certain temps, le réseau soit capable, par exemple, de prédire un certain résultat à partir d'exemples. Pour que cette méthode fonctionne, il faut que le réseau puisse avoir beaucoup de données pour son entrainement et le temps de s'entrainer (avec un nombre « d'epochs » raisonnable). Les résultats d'un réseau de neurones dépendent de plusieurs facteurs, en voici quelques exemples.

- Son architecture et son nombre de couches.
- Le nombre d'itérations.
- Le nombre et la qualité des données fournies.

Cependant, le modèle peut entrer en état « d'overfitting », c'est-à-dire qu'il va durant l'entrainement trop « coller » aux données d'entrainement et, par conséquent, se spécialiser sur peu de données. Le résultat de « l'overfitting » est que « l'accuracy » durant l'entrainement, sur les données d'entrainement, sera très haute, mais très basse sur les données de validation, rendant alors le modèle inutilisable sur des nouvelles données. Cela peut se produire si le nombre d'itérations est trop élevé.

Il n'existe pas de méthode pour connaitre exactement le nombre d'itérations à effectuer pour un modèle donné, on ne peut qu'estimer, mais on peut utiliser la méthode de « l'early stopping ».

Cette méthode va simplement arrêter l'entrainement du modèle si quelque chose se produit qui indique que le modèle entre en « overfitting », par exemple si « l'accuracy » de l'ensemble de validation s'empire.

1.2 Réseaux de neurones convolutifs, fonctionnement

Le fonctionnement des réseaux de neurones convolutifs reste globalement le même que ceux des réseaux de neurones. Les plus grandes différences se trouvent dans les fonctions utilisées (perte, activation…) mais également dans le fonctionnement des différentes couches ainsi que dans le traitement des entrées.

Cette section va introduire les réseaux de neurones convolutifs et développer l'explication des différentes couches utilisées. Les réseaux de neurones convolutifs ou CNN en anglais sont des réseaux de neurones spécialisés pour traiter l'analyse d'images en 2D. Ceux-ci peuvent avoir divers rôles, de la classification d'images en différentes classes ou en reconnaissance et différenciation d'objets sur une image, en passant par la reconnaissance faciale.

Avant de passer à la suite, il est important de comprendre comment une image peut être lue ou analysée par un réseau de neurones. D'abord, on peut se poser la question de qu'est-ce qu'une image ? Une image est une matrice de 2 dimensions contenant des pixels qui contiennent une couleur et celle-ci peut être divisée assez simplement en 3 composantes fondamentales : le rouge, le vert, le bleu (on peut afficher n'importe quelle couleur avec ces 3 couleurs fondamentales dites couleurs primaires). En réalité, notre image va donner 3 « sous images » qui sont l'image de base mais uniquement avec une couleur primaire (par exemple, une image uniquement avec la couleur rouge).

L'objectif principal de notre CNN est d'extraire des paternes de l'image pour pouvoir en tirer certaines représentations/propriétés uniques appartenant à certaines classes/objets et ainsi, réussir à les différencier. On pourra faire ça grâce aux 3 sous images, ces 3 canaux vont pouvoir donner différentes informations au CNN et révéler des informations utiles pour en déduire des paternes (par exemple, faire ressortir les zones plus sombres, l'arrière-plan ou des détails).

1.2.1 Couche de convolution

Cette partie est inspirée et a pour source : [1]

Pour ça, on va utiliser ce qu'on appelle une couche convolutive, celle-ci va parcourir l'image à l'aide d'un filtre (aussi appelé « kernel ») et extraire certaines représentations pour créer une carte de caractéristiques de l'image (aussi appelée « feature map »).

Voici la formule du calcul de couche de convolution en 2D [32].

$$g(x,y) = \sum_{i=-a}^{a} \sum_{i=-b}^{b} w(i,j) f(c-i, u-j) + b$$

Avec :

- f, l'image originale, qui va subir l'opération de convolution.
- g, l'image de sortie, qui a subi l'opération de convolution.
- $f(x,y)$, la valeur du pixel aux coordonnées (x, y) de l'image originale.
- $g(x,y)$, la valeur du pixel aux coordonnées (x, y) de l'image de sortie.
- w, le noyau du filtre (« kernel »).
- a, b, respectivement chaque élément du filtre tel que :$-a \leq i \leq a \; et \; -b \leq j \leq b$.
- c, u , respectivement les coordonnées x et y de l'image.
- b représente le biais.

Le « kernel » est une matrice beaucoup plus petite que l'image qui peut, en fonction de sa construction, faire ressortir certaines caractéristiques comme les éléments verticaux ou horizontaux.

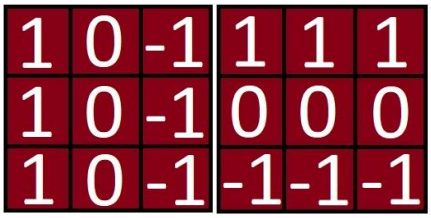


Figure 1.4 : ces 2 kernels de taille 3x3 sont respectivement un « kernel » pour mettre en évidence les lignes verticales et les lignes horizontales.

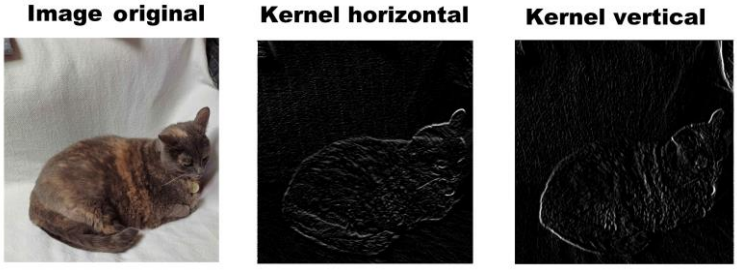


Figure 1.5 : voici l'illustration du résultat une fois appliqué à une image, on peut même observer que la transformation permet de distinguer des éléments qui n'étaient pas visibles avant, comme le motif sur la couverture.

Le « kernel » va donc parcourir l'image et créer une « feature map » ou carte de caractéristiques, via le kernel, celui-ci va parcourir et transformer chaque partie de l'image (faisant la taille du « kernel ») en un nombre qui fera partie de la « feature map ».

La carte de caractéristiques sera en général plus petite que l'image d'entrée car on ne peut pas appliquer le « kernel » à chaque colonne ou ligne de l'image d'origine et mettra en évidence certaines « features « de l'image.

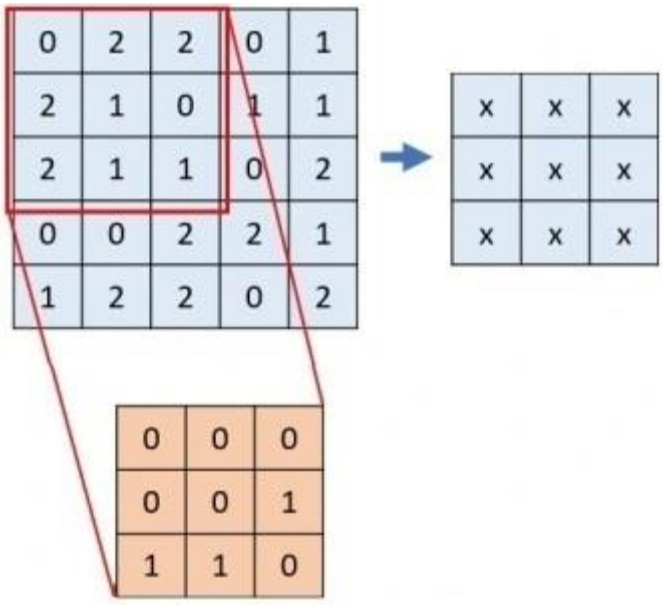


Figure 1.6 : dans cet exemple, l'image a une taille 5x5 et le « kernel » a une taille 3 x 3, le résultat est une image 3x3, on a perdu 16 pixels de l'image avec une couche de convolution.

Par conséquent, si la taille réduit à chaque fois, on ne peut alors appliquer qu'un nombre limité d'opérations de convolution. (On ne peut appliquer que 6 couches à une image 28X28 avant d'avoir une taille inférieure à celle du « kernel » (de taille 5)). Mais on peut remédier à ce problème avec l'utilisation de remplissage ou « padding ». Un remplissage des bords de l'image permet alors de ne pas réduire la taille de l'image. Le remplissage est constitué de valeur 0, pour ne pas interférer avec le kernel et la « feature map » produite.

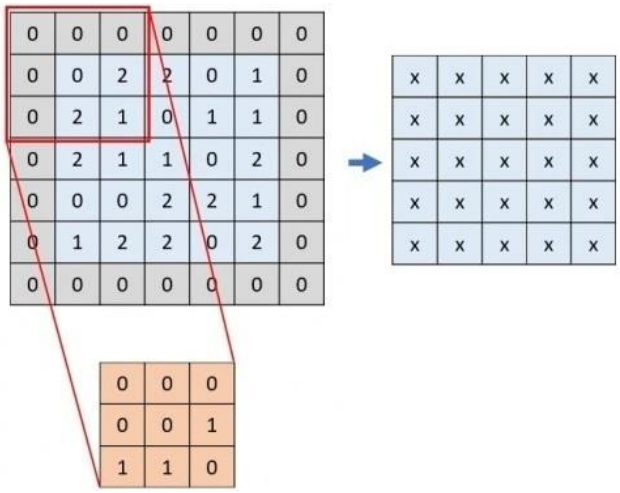


Figure 1.7 : dans cet exemple, on a rajouté 1 de « padding » et le résultat sera une image de la même taille que l'originale (5x5).

Il faut pouvoir calculer la taille de sortie des couches de convolution pour la création du modèle, heureusement on peut utiliser une formule simplifiée que voici : $\left(\frac{(D_i - F + 2*P)}{S}\right) + 1 = D_h[1]$

- D_i: la taille de l'entrée.
- D_h: la taille de la sortie.
- F: la taille du « kernel ».
- P: la taille du « padding » rajouté à l'image.
- S: stride : c'est le nombre de pixels par lesquels le « kernel » se déplace sur l'image, en général celui-ci est mis à 1.

Cette formule est utilisée pour obtenir à partir des sorties d'une couche k_i, les sorties de la couche k_{i+1}. Le calcul est important car si les paramètres d'entrée des couches n'ont pas les bonnes dimensions, le réseau de neurones ne va tout simplement pas fonctionner.

1.2.2 Couche de pooling

Cette partie est inspirée et a pour source : [2]

La couche « pooling » permet ainsi de transformer une image en une autre de taille bien inférieure, on va faire glisser une fenêtre d'une taille spécifiée, ainsi qu'un « stride » de taille spécifiée (souvent la même valeur que la taille de la fenêtre).

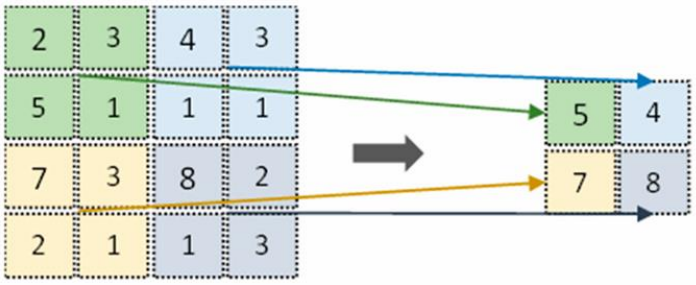


Figure 1.8 : illustration de la couche de « pooling », au départ on se trouve avec une image 4x4 et avec une opération de « pooling » ayant une taille de fenêtre de 2 et un « stride » également de 2, on obtient une image de taille 2x2, la taille de l'image a été divisée par 2.
L'opération ici est « MaxPooling ».

L'opération pour transformer les valeurs de la fenêtre en une seule valeur varie en fonction du type de « pooling » utilisé, voici 2 exemples fortement utilisés.

Les paramètres des 2 fonctions ci-dessous sont :

- h, w : respectivement la hauteur et largeur de l'image en entrée, h_{out}, w_{out} en sortie.
- N_i, C_j : respectivement le nombre d'exemples dans le « batch » et le nombre de canaux d'entrée/de sortie.
- Un dernier paramètre est la taille du « kernel » : (kH, kW)

« MaxPooling « prend la valeur maximum de la fenêtre, pour ça, il utilise cette formule :

$$\text{out}(N_i, C_j, h_{out}, w_{out}) = \max_{m=0,\ldots,kH-1} \max_{n=0,\ldots,kW-1} [\, input(N_i, C_j, stride[0] \times h + m, stride[1] \times w + n)]$$

« Average pooling » prend la moyenne de toutes les valeurs qui composent la fenêtre, pour ça, il utilise cette formule :

$$\text{out}(N_i, C_j, h_{out}, w_{out}) = \frac{1}{kH * kW} \sum_{m=0}^{kH-1} \sum_{n=0}^{kW-1} input(N_i, C_j, stride[0] \times h + m, stride[1] \times w + n)$$

Les 2 opérations donneront des résultats distincts et seront utiles dans des domaines différents.

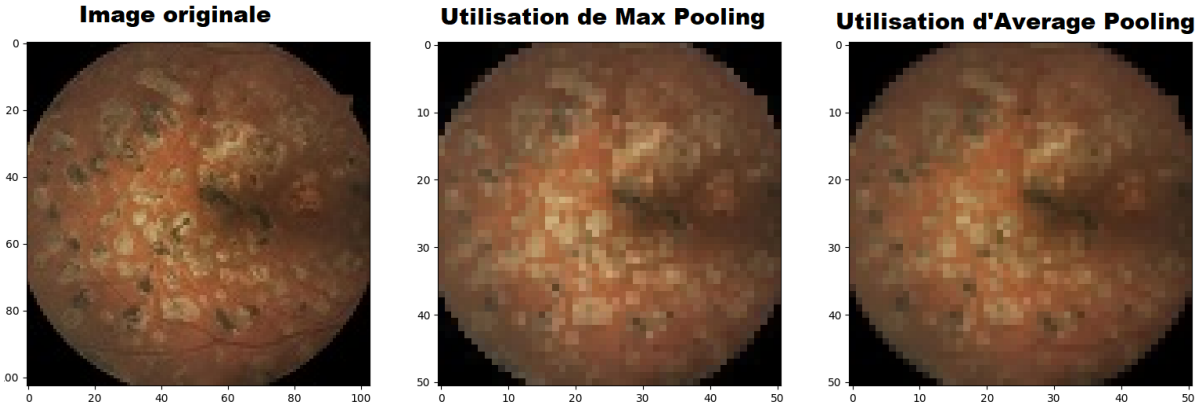


Figure 1.9 : rapide illustration de la différence entre « Maxpooling » au centre et « Average pooling » à droite.

Une remarque à faire sur la couche de « pooling » est qu'elle ne possède pas de paramètres, ni poids, ni biais, comparée à certaines autres couches comme la couche de convolution par exemple.

La couche de « pooling », en plus de réduire la taille de l'image ou de la « feature map », va également être utilisée pour rendre le modèle plus invariant aux translations ; c'est-à-dire que le modèle peut toujours reconnaitre un objet, même si celui-ci a été déplacé dans l'image.[40]. C'est très important pour la reconnaissance d'objets car parfois, l'objet n'est pas le sujet principal de l'image ou est partiellement caché par autre chose.

1.2.3 Dropout

Cette partie est inspirée et a pour source : [22][4]

Le « dropout » n'est pas une couche mais un système qui va permettre bien des choses. Le « dropout » va « désactiver » des neurones (ainsi que leur connexion) dans le réseau de neurones durant la phase d'entrainement. Il s'agit d'une méthode pour combattre « l'overfitting » et permettre d'améliorer les performances des réseaux de neurones. Les éléments « désactivés » sont choisis indépendamment suivant une distribution de Bernoulli. Par désactivé, on veut dire que la sortie du neurone est mise à 0 et qu'il ne participe à aucune des propagations. Les neurones seront cependant utilisés dans la phase de tests mais les neurones normalement désactivés effectueront une fonction identité.

L'implémentation du système dans Pytorch est la suivante.

La couche de « dropout » (considérée par Pytorch comme une couche) va « désactiver » avec une certaine probabilité p (par défaut 0.5), des neurones du tenseur d'entrée (la couche inférieure à la couche de « dropout »). Comme pour la couche de « pooling », la couche « dropout » ne possède pas de paramètre.

1.2.4 Couche Dense

Cette partie est inspirée et a pour source : [4][25]

Elle sert de transition entre les couches convolutives et la fin du modèle pour obtenir une classification, cette partie est entrainée comme les autres couches et a la même structure que celle d'un réseau MLP (« multi-layer perception »). Pour pouvoir obtenir du réseau des résultats (par exemple quelle classe a été déterminée), celui-ci va utiliser la fonction « softmax ».

Cette fonction va transformer un vecteur de valeurs en des scores pour chaque classe (précisément un vecteur de probabilité entre 0 et 1), la classe prédite aura la probabilité la plus haute.

La fonction « softmax » est la suivante pour tout élément j $\in\{1, \ldots,\mathrm{K}\}$:

$$\sigma(Z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$$

Avec :

- K, nombre d'éléments du vecteur d'entrée.
- Z, vecteur d'entrée possédant les éléments $(z_1, \ldots, z_K)$.
- $\sigma(Z)$ vecteur de sortie de K éléments (compris entre 0 et 1).

L'implémentation par Pytorch est exactement la même.

1.2.5 Couches linéaires

Cette partie a pour source [43].

Parmi les couches les plus simples, appliquer juste une transformation linéaire à l'entrée via la formule suivante : $\mathrm{y} = \mathrm{W} \cdot \mathrm{x} + \mathrm{b}$

Avec :

- y, la valeur de sortie de la couche.
- W, le poids de la couche.
- x, la valeur d'entrée de la couche.
- b, le biais.

Les couches linéaires sont très bonnes pour résoudre des problèmes linéaires mais elles peuvent avoir des soucis pour résoudre des problèmes non linéaires. C'est d'ailleurs pour pallier ces problèmes que d'autres couches non linéaires existent.

1.2.6 Fonction d'activation

Cette partie est inspirée et a pour source : [5][26]

Il y a plusieurs choix pour la fonction d'activation. Celle choisie pour ce travail est la fonction ReLU. Elle est présente dans le modèle ResNet18, utilisé en tant que modèle étalon et qui est beaucoup simple que la fonction sigmoïde :

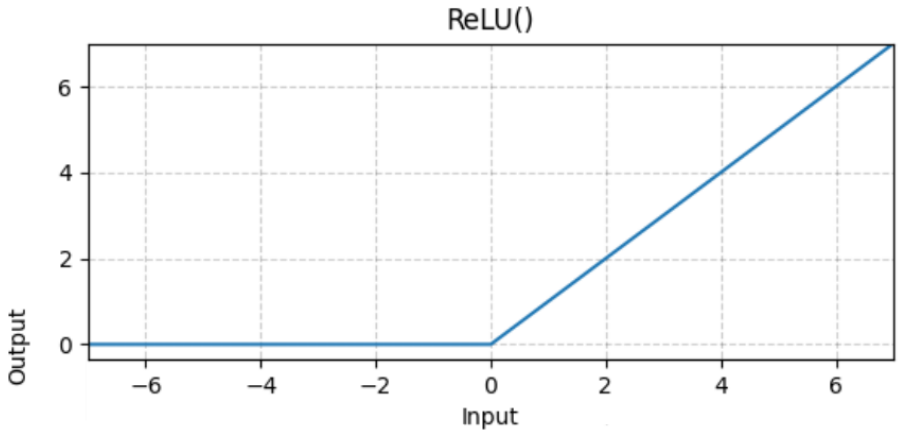


Figure 1.10 : fonction ReLu.

La fonction ReLu : $ReLU(x) = (x)^+ = \max(0, x)$

Elle a l'avantage d'être beaucoup plus simple à calculer que d'autres fonctions[26] et elle donne de très bons résultats en CNN et est, par conséquent, assez utilisée. Elle possède quelques désavantages mais ceux-ci ne seront pas cités dans ce travail.

1.2.7 Fonction de perte

Cette partie est inspirée et a pour source : [6]

Il y a plusieurs choix pour la fonction de perte. Dans ce travail, les différents modèles auront bien souvent une fonction de perte spécifique. Pour l'exemple, on pourrait choisir la fonction « Cross-Entropy Loss ». Le principe de cette fonction est assez simple, on va utiliser la confiance que le modèle a en sa prédiction (la confiance du modèle en son choix pour une classe) et comparer la probabilité de la vraie classe à la prédiction du modèle. Plus cette prédiction est haute, moins on pénalisera le modèle.

L'utilisation de la probabilité va permettre plusieurs choses, la fonction va « punir »(c'est-à-dire donner une erreur plus grande) le modèle qui donne une faible confiance pour la vraie classe, mais également un résultat vrai où la confiance est très faible. Au mieux, le modèle devrait avoir une haute confiance en ses bonnes prédictions.

L'équation générale de « Cross-Entropie » pour une application discrète est la suivante :

$$H(p, q) = -\sum_{k \in K} p(k)\ log\ q(k)$$

Avec :

- K, l'ensemble de classes possibles.
- k, une valeur possible de K.
- $p(k)$, distributions attendues pour la valeur K.
- $q(k)$, distribution prédite par le modèle pour la valeur K.

$p(k)$ et $q(k)$, représentent les probabilités des classes pour un élément k, c'est-à-dire dans la distribution, chaque classe aura une valeur associée. Avec $p(k)$, la distribution attendue est celle où une valeur est égale à 1 (celle de la vraie classe) et les autres à 0. Et $q(k)$, la distribution prédite, qui aura chaque valeur comprise entre 0 et 1. On va alors comparer les 2 distributions et voir à quel point la distribution $q(k)$ est inexacte comparée à $p(k)$. Plus cette inexactitude est faible plus le modèle se rapproche de la vérité et inversement.

A noter que certaines fonctions de perte ne sont utilisées que pour certaines applications. « Cross-Entropy Loss » est utilisée pour les problèmes de classification par exemple ou sa variante « Binary Cross-Entropy » pour la classification binaire. Certaines fonctions de perte conviennent mieux à certaines tâches ou objectifs que d'autres.

1.3 Autres aspects

1.3.1 Performance du modèle

Pour connaitre les performances d'un modèle, on peut calculer « l 'accuracy » qui représente le nombre de bonnes prédictions du modèle sur le nombre total de prédictions :

$$accuracy = \frac{Bonnes\ prédictions}{Toutes\ les\ prédictions}$$

On va tester le modèle sur un ensemble de tests, celui-ci a une petite partie du « dataset » de départ et est composé de données que le modèle n'a jamais vues, ce qui est parfait pour évaluer sa performance. En général, on calcule « l'accuracy » sur chaque classe et « l'accuracy » totale.

Mais on peut avoir un problème avec « l'accuracy » : imaginons un modèle entrainé sur 5 classes avec la première classe très minoritaire, on pourrait très bien avoir une « accuracy » assez bonne (par exemple 80%) mais avoir une précision de 4% pour la première classe et 99% pour les 4 autres, rendant le modèle moins pertinent qu'au premier abord. On peut alors utiliser la précision globale qui n'est qu'une simple moyenne des « accuracy « de chaque classe.

1.3.2 Déséquilibre de classes

Un problème récurrent en CNN et en « machine learning » est le déséquilibre des classes. Il est dû au nombre d'exemples présents dans chacune des classes. Le mieux est d'avoir des classes avec le même nombre d'exemples, mais ce n'est pas toujours possible et il n'est pas rare de se retrouver avec des classes déséquilibrées dans un « dataset ». S'il y a un déséquilibre de classes, la ou les classes minoritaires seront celles possédant le moins d'exemples ou un nombre d'exemples bien inférieur aux autres classes. Ça peut être assez faible (quelques centaines d'exemples), à très important.

Par exemple, dans le « dataset Diabetic Retinopathy » [16] possédant 5 classes, la classe n°1 possède 25 807 exemples alors que la classe n°5 n'en possède que 708, soit seulement 2,7%, donnant un déséquilibre de classes énorme. Les modèles, lors de leur entrainement, deviennent biaisés pour les classes majoritaires, par conséquent les classes minoritaires seront plus souvent mal classées et confondues avec la ou les classes majoritaires[17]. (Ce qui ne donne pas de bons modèles). En fonction de comment les expériences sont effectuées (par exemple, un ensemble de tests déséquilibrés), on peut se retrouver avec un faux sentiment de précision et des résultats qui induisent en erreur.

1.3.3 Augmentation des données

L'augmentation de données est une méthode qu'on peut utiliser pour augmenter le nombre de données disponibles, cette méthode est assez simple et moins coûteuse que de labelliser des données ou de s'en procurer d'autres. L'idée est d'effectuer des transformations aux données déjà existantes, celles-ci peuvent être diverses et variées, on peut réaliser des transformations telles que des rotations sur l'image, des zooms, des changements de tailles. On peut également jouer sur la couleur, changer les teintes, la luminosité, rajouter du bruit, etc...

On part du principe qu'effectuer une transformation sur une image ne fait pas perdre la sémantique de celle-ci. Il existe encore une dernière méthode qui est la création de données synthétiques, autrement dit, créer de toutes pièces des données artificielles. Les méthodes varient en fonction de la nature des données, par exemple, on pourrait utiliser des modèles 3D et un logiciel de rendu pour créer des images de voitures de façon assez simple. On peut également utiliser GANs « generative adversarial networks » [33] qui, sans rentrer dans les détails, permet de générer des données synthétiques.

Figure 2.7 : quelques exemples d'images générées par GANs, aucune de ces personnes n'existe.

1.4 ResNet 18

Cette partie est inspirée et a pour source : [21]

Cette partie est dédiée au réseau ResNet 18, celui-ci est un réseau de neurones convolutif, utilisé tout au long de ce travail.

ResNet ou « DeepResidual Learning for Image Recognition » est une architecture de CNN un peu particulière, elle permet d'avoir des réseaux très profonds et très performants en réglant le problème de dégradation. Le problème de dégradation est qu'à une certaine profondeur du réseau, la précision du modèle va saturer, puis celle-ci va fortement diminuer.

Pour résoudre le problème de dégradation, ResNet utilise une architecture un peu particulière, tout d'abord le réseau est composé en « blocs », chacun de ses blocs est constitué de plusieurs couches de convolution utilisant une fonction d'activation ReLu.

Ensuite, l'architecture possède des « connexions raccourcis » ou « shortcut connection », des connexions qui esquivent les blocs de convolution et vont directement à la sortie de ceux-ci. (Dans ce cas présent, ces connexions raccourcis sont de simples « identity mapping », la sortie est ajoutée à la sortie du bloc). Ces « raccourcis » n'ajoutent pas de charge de travail pour entrainer le modèle et ils sont toujours ouverts.

Une telle architecture permet 2 choses.

- Durant la « forward propagation », elle va permettre à l'information qui rentre dans la cellule d'être ajoutée directement à la sortie de celui-ci, ça permet de mieux conserver l'information d'une couche à l'autre.
- Durant la « backward propagation », ces raccourcis peuvent également être utilisés pour le gradiant.

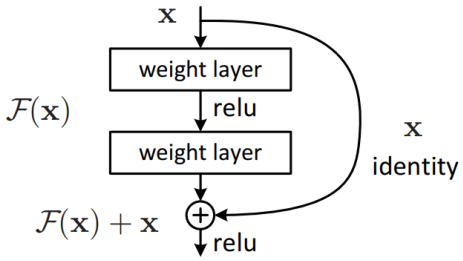


Figure 1.11 : illustration de l'architecture par bloc de ResNet ainsi que de la couche identité.

- x représente l'entrée fournie au bloc.
- $F(x)$ représente la sortie du bloc avec l'entrée x.
- $x\ identity$ représente le « raccourci » pour ajouter les données d'entrée aux données de sortie.
- $F(x) + x$ représente la sortie du bloc , «l'identity mapping » du raccourci.

Sans l'utilisation du raccourci, le problème de dégradation est présent.

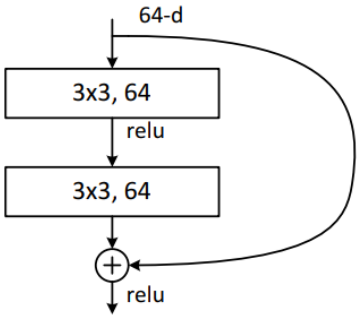


Figure 1.12 : illustration du premier bloc de ResNet18, le 3x3 représente la taille du filtre, et 64 représente le nombre de filtres.

ResNet 18, comme son nom l'indique, possède 18 couches, 4 blocs de 4 couches de convolution, 1 couche de convolution initiale et une couche de fin pour la classification, les paramètres de celles-ci doublent entre chaque bloc. Les architectures de ResNet 18 et 34 sont reprises dans le tableau suivant en guise de comparaison.

layer name	output size	18-layer	34-layer
conv1	112×112	7×7, 64, stride 2	
conv2_x	56×56	3×3 max pool, stride 2	
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$
	1×1		

Figure 1.13 : architecture de ResNet 18 et ResNet 34, on peut voir que leur construction est très similaire et que le seul changement entre les 2 versions est le nombre et les paramètres des blocs.

1.5 Vit (Vision transformer)

Cette partie est inspirée et a pour source : [35]

Le ViT ou « Vision Transformer » est une architecture de réseaux de neurones alternatifs aux réseaux de neurones convolutifs, ceux-ci se basent sur les mêmes concepts que ceux utilisés en LLMs. Ils utilisent un mécanisme « d'attention » pour pouvoir plus facilement capturer des relations dans les images. Le fonctionnement des ViT est assez complexe, toutefois voici un bref résumé de leur fonctionnement avec un exemple portant sur une tâche de classification.

D'abord, une image va être découpée en différents patchs , c'est-à-dire en différents morceaux de même taille, on va également sauvegarder la position de chaque patch. Ces patchs sont alors toujours en 2D mais nous avons besoin de les transformer en 1D. Pour ça, nous allons aplatir ces patchs et les transformer en vecteurs via la formule suivante: $\mathrm{Sp} = P^2.C$

Avec :

- P, la taille de chaque patch.
- C, le nombre de canaux de l'image d'origine.
- Sp, la taille finale du vecteur.

Pour plus de simplicité, ces différents vecteurs seront appelés $v_1, v_2 \dots v_x$,x étant le nombre de patchs dans l'image.

Pour conserver la position des différents patchs , nous allons ajouter aux différents vecteurs

v_i , i ∈ (1, x) un vecteur de position. Celui-ci est de taille identique au vecteur v_i et contient la position du patch i ,appelée « position embedding ». De plus, nous allons utiliser la position 0, qui est assez spéciale car elle n'est pas reliée à un patch de l'image. Elle contiendra un vecteur de taille égale à celle du nombre de patchs, c'est ce vecteur qui sera donné à la « MLP Head » pour pouvoir prédire la classe d'une image.

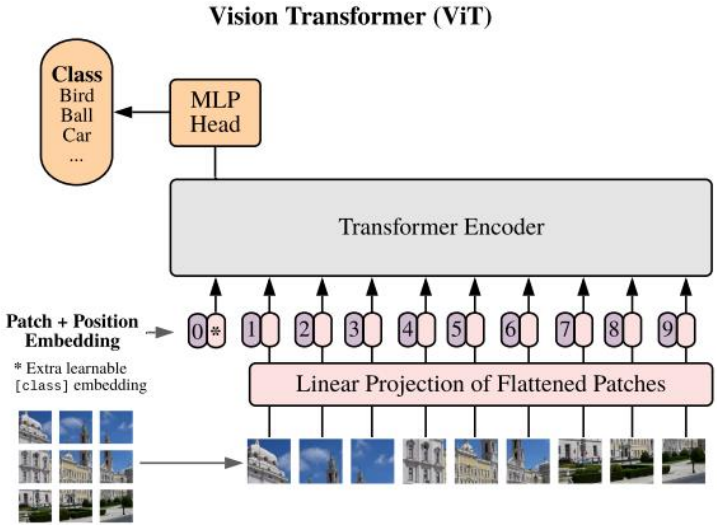
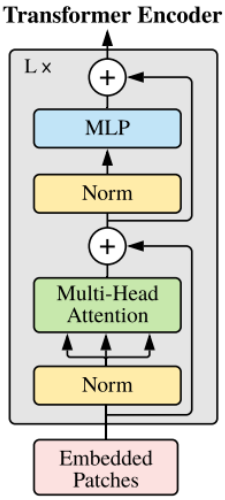


Figure 1.14 : fonctionnement d'un ViT (vue globale).



Maintenant que cette courte partie sur la préparation des données est terminée, nous pouvons maintenant passer à l'explication succincte du « Transformer Encoder ». Celui-ci est la partie centrale du VIT et comme pour ResNet, c'est un bloc qui est répété L fois.

Il contient plusieurs blocs.

- Bloc de normalisation utilisé pour stabiliser l'apprentissage.
- Connexions résiduelles, utilisation similaire à celles utilisées sur ResNet.
- MLP, « Multi-layer Perceptron », permet au modèle d'apprendre des représentations utiles sur un patch (sans prendre en compte les autres).
- MHA, « Multi-Head ». Le sous-bloc le plus important est la « Multi-Head ». Attention, celle-ci a pour but de former des relations entre les différents vecteurs v et leur position dans l'image. Cela permet au modèle d'apprendre des représentations en prenant en compte les autres patchs dans l'image.

Les ViT sont une alternative au CNN, il en existe plusieurs types différents, pour ce travail deux modèles les utilisents.

Figure 1.15 : fonctionnement d'un ViT (vue du transformer).

2 Application de l'IA en imagerie médicale

Cette section porte sur les différentes utilisations possibles de CNN et de l'IA dans l'imagerie médicale. Dans le domaine médical, les réseaux de neurones convolutifs sont en général utilisés pour réaliser des diagnostics à partir d'images provenant d'un patient, via un simple classificateur ou une détection d'objet. Quelques exemples d'utilisations sont la détection de tumeurs au niveau du cerveau ou la détection de cancers du sein.[19] Cela peut être utilisé avant traitement ou après traitement, dans le premier cas pour la présence ou non d'une maladie et dans le second, pour savoir si le traitement a été efficace ou s'il y a une rechute. Cette automatisation fait gagner un temps précieux aux différentes personnes du domaine et permet d'alléger la charge de travail du personnel. Le facteur temps peut jouer un rôle crucial pour la santé du patient dans le traitement de certains types de maladies, il y a aussi le facteur de précision, les erreurs de traitement peuvent conduire à une inaction, faisant encore une fois perdre un temps précieux. Avec l'avancement de l'IA ces dernières années, son utilisation devient de plus en plus courante. Très récemment une équipe de recherche a créé une variante du modèle Mirai (un modèle qui permet la prédiction de cancers du sein), nommé « AsyMirai », plus simple et plus compréhensif que le précédent. Celui-ci est capable de détecter le cancer du sein chez une femme jusqu'à 5 ans à l'avance.[15] Un autre exemple est la détection de cancers de la peau, avec un niveau comparable à celui de dermatologues [27]. Il y a également beaucoup d'utilités pour « le machine learning » en général mais ce point ne sera pas discuté lors de ce travail.

2.1 Le domaine de l'imagerie médicale

Cette partie a pour source :[48]

Cette partie traite des différents aspects et complexités de travail sur des données provenant du monde médical. Le domaine de l'imagerie médicale est très vaste, complexe et ce pour plusieurs raisons. D'une part, il utilise une grande diversité de technologies pour la production d'images tout en tenant compte du fait que chacune d'entre elles produira des images différentes en fonction de ce qu'on souhaite trouver ou observer. En effet, ce n'est pas l'image en elle-même qui est intéressante mais plutôt les détails qu'elle met en évidence. En général, ce qu'on recherche sur une image est la présence ou l'absence d'un élément. Par exemple, dans le cas de la radiographie d'un bras pour observer s'il y a ou non une fracture, l'image du bras en tant que telle ne nous intéresse pas, ce qui est important c'est uniquement le détail qui atteste de la présence de la fracture.

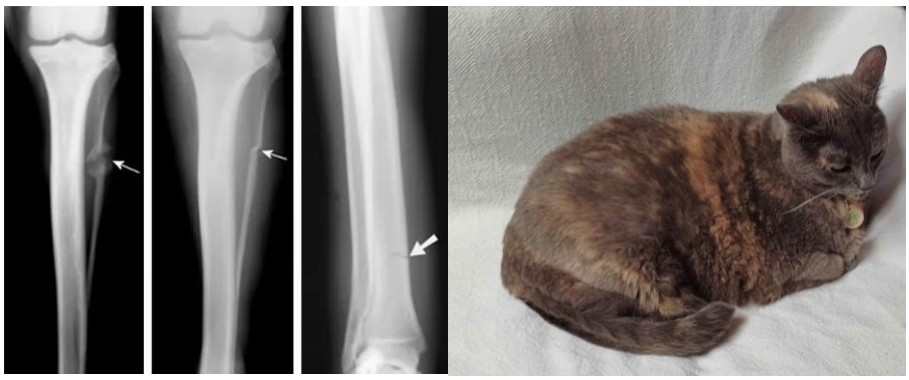


Figure 2.1 : on le voit bien sur l'exemple ci-dessus, le vrai « sujet » de l'image n'est pas ce qui se voit au premier coup d'œil, il faut savoir où poser son regard. Par comparaison à un autre type d'image où le sujet est central et son interprétation sans ambiguïté.

Le domaine de l'imagerie médicale est découpé en plusieurs types d'imageries qui viennent donc des nombreuses technologies actuelles : rayons X, radioactivité, ultrasons, radiographie, résonnance magnétique…

Chaque technologie a ses avantages et inconvénients et a bien souvent un domaine de prédilection dans lequel elle est alors largement utilisée. La lecture de ces images nécessite un grand nombre de compétences et connaissances conduisant à une analyse pointue, précise et pertinente. Une autre conséquence de ce changement d'examen est que les images obtenues seront différentes en fonction de la méthode utilisée, ceci même si elles ciblent la même partie du corps ou la même personne.

Cette différence vient du processus de création de l'image en elle-même. Par exemple, une radiographie utilise des rayons X, ce qui permet de voir très facilement les parties plus dures comme les os, mais donnera un effet de superposition s'il y a plusieurs os dans le chemin du rayon. Une IRM emploie la résonnance magnétique pour fonctionner et permet de voir les tissus mous sans superposition, donnant une image plus compréhensible, plus nette comme si on avait coupé le sujet en deux.


Figures 2.2 et 2.3 : à gauche, une radiographie, à droite une IRM, même si ces 2 photos sont issues de la même région du corps humain, le résultat est complètement différent et sera utilisé pour traiter des problèmes distincts.

2.2 Complications liées aux données labellisées dans l'imagerie médicale

L'entrainement de modèles CNN supervisés demande plusieurs choses. La plus importante est les données utilisées pour l'entrainement. Il en faut beaucoup et que celles-ci soient de bonne qualité. Mais il faut également que ces données soient « labellisées »(ou étiquetées). Ces labels (en général des mots) vont définir la ou les classes à laquelle/auxquelles vont appartenir les images. Sans ces labels, les données ne sont pas exploitables en CNN supervisé. Et pour ce faire, il n'y a pas de méthode automatique donnant 100% de labels corrects, il faut le faire « à la main », ça doit donc être réalisé par un être humain et non par une machine pour éviter d'avoir des exemples incorrectement classés. L'utilisation humaine apporte plusieurs problèmes pour labelliser les données de « datasets ». Tout d'abord, il y a le problème du temps, labelliser des données prend beaucoup de temps. Chaque image doit être analysée puis classée. On ne peut le faire qu'une image à la fois et s'il y a plusieurs labels par image, ce sera encore plus long. En plus, les « datasets » utilisés comportent un très grand nombre d'images, par exemple ImageNet contient 14 197 122 images annotées[20], et ça multiplie encore le temps utilisé pour labelliser les données. Par conséquent, c'est un processus très coûteux car il faut l'intervention de l'homme pour obtenir des labels de qualité et donc payer de la main d'œuvre pendant longtemps, il y a aussi l'argent utilisé pour la récolte ou l'achat de données.[8][30]

2.2.1 Erreurs dans les « datasets »

Même si les humains sont très efficaces pour différencier des objets, « l'erreur est humaine» et il n'est pas rare de retrouver des erreurs dans les « datasets ».

On peut avoir plusieurs types d'erreurs. La plus courante étant des données mal classées, le label qui a été assigné est incorrect. Cifar10[34] est un « dataset » qui reprend des images d'animaux et de véhicules, il contient des erreurs, comme l'image ci-dessous qui n'a pas été bien classée.

Figure 2.4 : une image de grenouille mais celle-ci a été classée en tant que chat[49] dans l'ensemble de tests de Cifar10[34] (erreur au niveau du dataset).

Un autre type d'erreur est d'avoir des données qui ont été dédoublées dans les ensembles de validation et d'entrainement, ça s'appelle la contamination des jeux de données. Comme ci-dessous, cette image venant du « dataset BloodMNIST » est dupliquée.

Figure 2.5 : une image de globule blanc qui est donc dupliquée entre le set d'entrainement et celui de validation.

Ces différentes erreurs peuvent causer des problèmes pour la qualité des modèles si elles sont en grande quantité mais en général elles sont minoritaires. Elles sont souvent dues à des fautes d'inattention et peuvent être simples à détecter s'il n'y a que peu de données. En revanche, sur de grands « datasets », elles deviennent plus complexes à repérer car il faut manuellement parcourir le « dataset » pour les trouver (ou utiliser des modèles déjà entrainés pour les voir rapidement).

On peut également avoir le problème de l'interprétation, deux personnes peuvent interpréter différemment une image, ça cause plus de soucis quand une image possède plusieurs labels ou quand le problème n'est pas de labelliser une image mais davantage de délimiter une zone dans une image. [24]

Ça touche d'autres domaines, par exemple la classification de discours de haine où plusieurs facteurs (âge, personnalité…) influencent la décision de la personne labellisant le texte[23].

2.2.2 Connaissances d'experts

Bien souvent, pour labelliser les données, on peut faire appel à des personnes n'ayant pas de connaissances spécifiques sur le sujet. Cependant, en fonction du type de données qu'on labellise, on pourrait devoir faire appel à des personnes qualifiées ayant des connaissances spécialisées ,appelées « experts knowledge », à des experts donc dans un domaine précis et dont le travail est de différencier/classer les images.Dans le domaine de l'imagerie médicale, il faut souvent faire appel à ces experts pour labelliser les données et ceci pour plusieurs raisons.

- Différences entre classes très faibles.
- Détection de la présence ou non d'objets particuliers dans l'image.
- Besoin de comprendre l'anatomie humaine.
- Différents types d'images, chacune demandant des compétences précises.

Également, l'utilisation d'experts réduit la chance d'erreur, ce qui est nécessaire car les modèles sont créés pour être utilisés lors de diagnostics de maladies d'où l'importance d'avoir de bonnes données pour avoir des résultats exacts. Il y a aussi un autre problème, il n'y a pas assez de personnes ayant les qualifications nécessaires pour pouvoir classer les images, sachant que chaque domaine demande des qualifications spécifiques et qu'il y en a beaucoup de différents.

On se retrouve alors face à une situation un peu spéciale dans laquelle on a une multitude de données (même si un nombre assez réduit comparé à d'autres domaines d'étude et d'analyse) mais il n'y a pas assez de personnes compétentes pour pourvoir les traiter et également, dans certains domaines, trop de nouvelles données arrivent chaque jour pour que la personne disponible puisse les labelliser. En radiologie par exemple, il faudrait que le radiologue interprète et labellise une image toutes les 3-4 secondes pour suivre le flux de nouvelles données, ce qui n'est pas possible.(Selon les auteurs de l'article «Self-supervised learning for medical image classification: a systematic review and implementation guidelines » [8]).Et dans une application supervisée, les données non labellisées ne valent rien et ne sont tout simplement pas utilisables.

Le besoin d'experts pour labelliser des images ou autres données touche bien d'autres domaines que celui du médical et est présent à chaque fois qu'une connaissance spécifique sur un sujet est requise. On peut notamment citer le domaine de la géologie, de l'astronomie, du monde animal et de l'histoire comme le montre l'exemple ci-dessous.

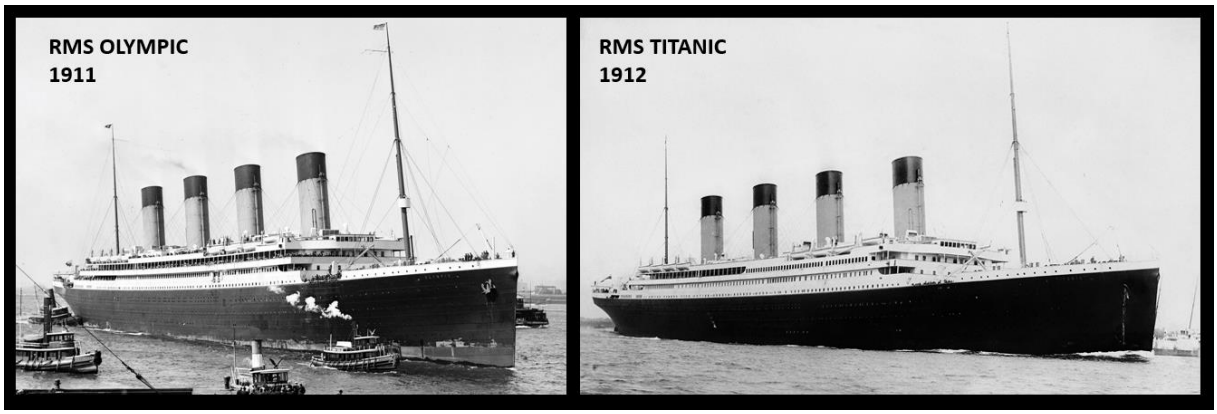


Figure 2.6 : voici un exemple concernant des bateaux de croisière, à gauche l'Olympic, et à droite le Titanic, ces 2 bateaux sont des « Sister Ship », ils sont quasiment identiques car construits sur les mêmes plans, sans être un expert dans le domaine, il est compliqué de les différencier.

2.2.3 Déséquilibre de classes dans l'imagerie médicale

Le problème du déséquilibre de classes est très présent dans le domaine de l'imagerie médicale, notamment à cause d'une raison assez simple, on possède plus de données de personnes en bonne santé que de personnes malades. Le problème est que, par exemple, pour la détection de maladies, les exemples de personnes malades sont de plus grande importance que ceux de personnes saines.[17] On peut encore une fois utiliser le « dataset Diabetic Retinopathy » pour le voir, la classe n°0, la plus importante (en nombre d'exemples) est celle provenant de personnes saines et les 4 autres de personnes malades. Sur un total de 35 128 images, seulement 26,5% proviennent de personnes malades (réparties entre différentes sévérités de Rétinopathie diabétique) et plus la maladie est à un stade grave, plus le nombre d'exemples est faible. [16]

Un autre problème peut apparaitre, le nombre de personnes atteintes d'une certaine maladie n'est pas égal, la distribution des maladies n'est pas uniforme et dépend de plusieurs facteurs (sexe, âge, autres maladies, lieu de vie, etc...). De plus, pour pouvoir obtenir des données, il faut de l'équipement bien précis, qui n'est pas disponible partout et qui dépend de facteurs économiques et géographiques importants.

On peut également avoir le cas d'une maladie ou d'un problème médical ayant une chance plus élevée d'arriver à une partie du corps bien spécifique. On peut prendre le cas des fractures qui peuvent se produire plus facilement à certains endroits. Par exemple, des fractures liées aux côtes sont presque 4 à 5 fois plus élevées que celles du pelvis.[18] Provoquant des biais dans la détection dû à un déséquilibre de classes.

2.3 Disponibilité et confidentialité des données

Les données médicales sont souvent davantage protégées que d'autres types de données, rendant leurs accès et utilisations plus compliqués dû à des implications éthiques et légales. Il s'agit de données personnelles permettant d'obtenir beaucoup d'informations « confidentielles » sur une personne. En Europe par exemple, en plus d'être vues comme données à caractère personnel, celles-ci sont également classées par le RPGD (voir RGPD art 9,(vii)) comme des données sensibles. Cette protection vient du fait qu'il est possible de pouvoir identifier les personnes propriétaires des images grâce à des données biométriques. Cette couche de protection supplémentaire rend le stockage et la collecte de ces données plus compliqués et plus longs.

2.4 Augmentation et amélioration de données dans l'imagerie médicale

Comme cité plus haut, on peut utiliser certaines méthodes pour pouvoir créer des données (données synthétiques) [28] mais il est également possible d'améliorer les données déjà obtenues. Par exemple, il n'est pas rare d'avoir, lors de radios, du bruit sur l'image, il est possible avec certaines méthodes notamment des variantes de GANs de le retirer le plus possible[28]. Cela permet de créer plus de données et d'améliorer celles déjà existantes, ce qui devrait rendre meilleure la qualité des modèles créés avec ces données.

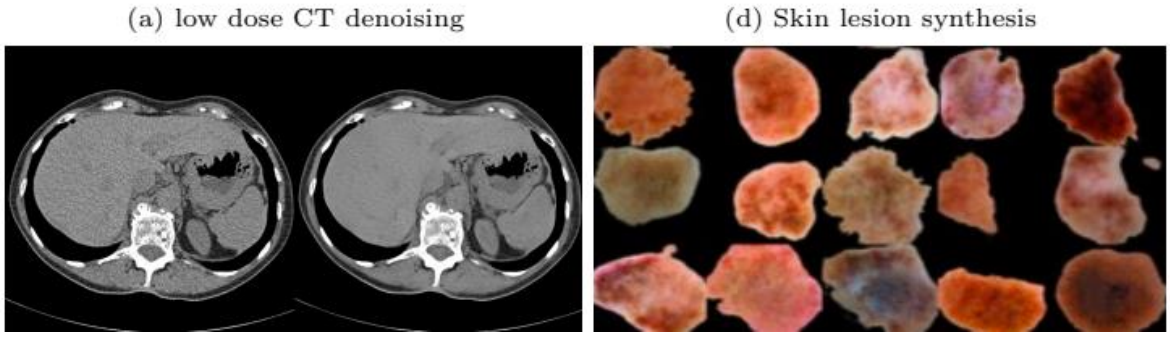


Figure 2.8, (a) un exemple d'image où le bruit a été retiré, (b) création de données synthétiques de lésions de la peau (malignes et bénignes).

3 Fonctionnement de l'apprentissage auto-supervisé

Cette partie est inspirée et a pour source : [7][8]

L'idée principale derrière l'apprentissage auto-supervisé est assez simple : pouvoir utiliser une grande quantité de données non labellisées pour que le modèle apprenne des représentations utiles et/ou des paternes à partir des données fournies et d'une tâche appelée tâche prétexte et qui ne sert qu'à cet apprentissage. Ensuite, ce que le modèle a appris peut être utilisé pour résoudre une vraie tâche (comme une tâche de classification), avec l'aide de données labellisées mais en plus faible quantité. Comme vu précédemment, avoir des données labellisées est coûteux en temps et en argent et nécessite forcément une intervention humaine. Utiliser principalement des données non labellisées permet en grande partie d'éliminer ces facteurs.

Normalement, en apprentissage supervisé, le modèle a besoin d'une base de vérité pour son apprentissage, c'est-à-dire pouvoir comparer les classes auxquelles appartiennent les données avec la classe prédite et ainsi savoir si la prédiction était correcte ou pas. Sans classe pour comparer, la tâche peut paraitre impossible. Mais ce qu'on va utiliser comme base de vérité sera l'image en elle-même. Etant donné qu'on n'a que l'image, on peut imaginer des tâches qui permettront au modèle de s'entrainer, d'apprendre la structure interne de l'image pour pouvoir par la suite réussir à les différencier. On pourrait imaginer diverses tâches qui permettent d'apprendre la structure interne.

Par exemple, effectuer une rotation de l'image et forcer le modèle à déterminer l'angle de rotation ou alors diviser l'image en plusieurs morceaux ou même cacher certaines parties de l'image et entrainer le modèle à la reconstituer.

En « Self-Supervised Learning » ou SSL, de telles tâches sont appelées tâches prétextes (« pretext task ») et comme leur nom l'indique, les tâches prétextes ne sont pas utilisées pour la tâche choisie (classification par exemple), elles ne servent qu'à entrainer le modèle sur des données non labellisées. En effet, un modèle SSL est découpé en 2 parties, la partie « backbone » (ou encodeur noté $f(.)$), celle-ci est située avant la partie décodeur et c'est l'encodeur pour lequel les différents paramètres seront entrainés. Puis après la « backbone », on trouve le décodeur (ce qui va résoudre la tâche prétexte, noté $g(.)$), qui, comme cité avant, ne sert qu'à l'entrainement.

Après ça, il est retiré et remplacé pour un décodeur pour la « downstream task » , c'est-à-dire la vraie tâche qui est l'objectif du modèle (par exemple un classificateur). Cette partie sera entrainée via un ensemble plus restreint de données supervisées : elle est appelée le « fine tuning ».

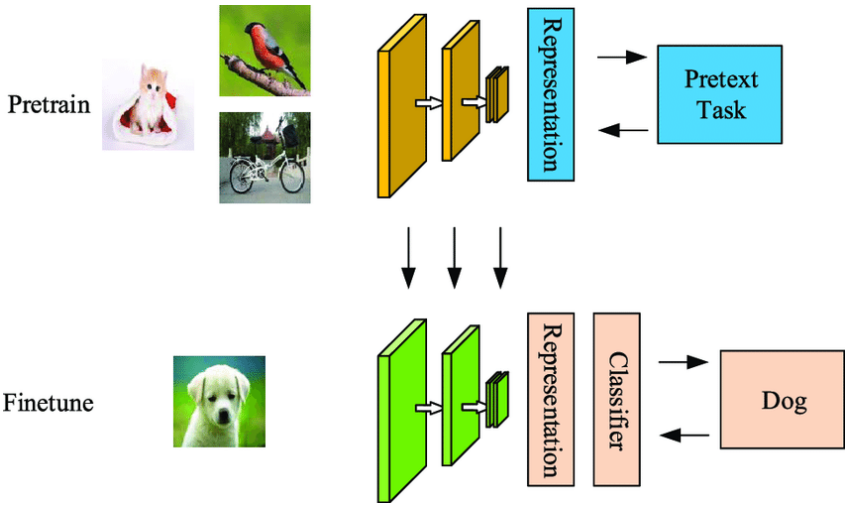


Figure 3.1 : cette image illustre le fonctionnement du SSL, on commence avec un entrainement sur une tâche prétexte puis, on remplace le décodeur utilisé pour résoudre la tâche prétexte, par un autre décodeur pour la vraie tâche, ici un classificateur.

Inconvénients des tâches prétextes

Les tâches prétextes ne sont pas parfaites, parfois le modèle peut trouver des « raccourcis » lors de l'entrainement sur celles-ci, c'est-à-dire que les caractéristiques que le modèle va trouver dans les images ne seront d'aucune utilité pour la « downstream task »[8].

Aussi, certaines tâches prétextes, dues à leur nature, peuvent ne pas être utiles dans certains domaines, certaines peuvent cacher une partie de l'image et si cette partie contient de l'information cruciale, par exemple la fracture sur une radio, cette information est détruite[8].

Un autre inconvénient est qu'il faut créer ces tâches (l'architecture, la fonction de perte ...) ce qui prend du temps et des ressources et ensuite, il faut les tester pour voir leur résultat, il est possible que ce qui a été appris pendant la tâche prétexte ne soit pas utile pour la « downstream task », rendant ainsi la tâche prétexte inutile.

3.1 Catégories de tâches prétextes

Les tâches prétextes sont divisées en plusieurs « catégories », celles-ci ne sont pas universelles. De plus, les différentes limites entre catégories sont assez floues. Il n'y a pas vraiment de standard. Les différentes catégories listées ci-dessous sont basées sur 2 papiers de recherche : [8] et [7], les illustrations sont des modifications de celles utilisées dans le papier de recherche [8], mais celles-ci ont été légèrement changées pour plus de clarté et pouvoir être plus facilement utilisée pour ce travail. Le SSL est encore une discipline jeune et en plein essor, il est très probable que cette partie ne reflète pas la réalité future par rapport à son évolution.

Avant de passer à la présentation des différentes catégories de tâches prétextes, nous allons présenter une légende pour comprendre les différents schémas utilisés.

Figure 3.6 : représente l'encodeur (la « backbone »).

Figure 3.7 : représente le décodeur (ce qui va résoudre la tâche prétexte), est parfois appelé la « projection Head ».

Figure 3.8 : représente « l'embedding » de l'image (une représentation de l'image sous forme de d'un vecteur de taille fixe).

3.1.1 Self-prediction SSL

Cette partie est inspirée et a pour source : [8]

Le principe du « self-prediction SSL » est assez simple : altérer une portion de l'image et utiliser le reste de l'image non altérée pour la reconstituer. L'idée vient des NLP où l'utilisation de textes à trous lors de l'entrainement de modèles a donné de très bons résultats et ce principe est réutilisé mais pour des images. [8] Normalement, le reste de l'image contient assez d'informations pour pouvoir reconstituer la partie manquante, cependant ce n'est pas toujours le cas. Il y a beaucoup d'altérations possibles, voici quelques exemples.

- L'altération peut être de masquer une petite partie de l'image, la tâche prétexte devra reconstituer la partie manquante de l'image à partir du reste de celle-ci.
- Elle peut également être de découper une partie de l'image en différents patchs d'image, la tâche prétexte devra remettre aux bonnes coordonnées les patchs manquants de l'image.

Cette catégorie est assez proche de « generative SSL » (qui altère également l'image), la différence majeure est qu'en self-prédiction SSL, la transformation n'est effectuée que pour une partie de l'image, alors qu'en « generative SSL », la transformation s'applique à toute l'image. Le modèle est composé de 2 parties. L'encodeur : celui-ci va, à partir des morceaux non altérés de l'image, apprendre des représentations utiles. La partie décodeur : son rôle sera de reconstituer l'image d'origine, via ce que l'encodeur a appris. La fonction de perte utilisée pour optimiser le modèle devra déterminer à quel point les 2 images sont proches.

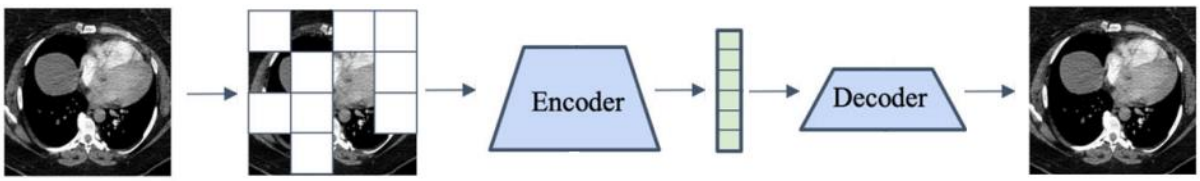


Figure 3.2 : voici une illustration de la méthode self-prédiction où la tâche prétexte est de déterminer la position de différents patchs de l'image.

3.1.2 SSL Innate relationship

Cette partie est inspirée et a pour source : [8]

Le principe du «SSL innate relationship » est assez simple. Créer des tâches prétextes assez spécifiques pour que le modèle arrive à apprendre des caractéristiques visuelles pour « comprendre » la structure interne des images. Les tâches prétextes consisteront bien souvent en une légère transformation des données de base.

Voici quelques exemples de transformations possibles.

- Effectuer une rotation de l'image, qu'elle soit horizontale ou verticale, une inversion sur un axe est aussi possible, la tâche prétexte sera un classificateur pour déterminer la transformation effectuée.
  Exemple : savoir quel est le degré de rotation de l'image parmi 4 rotations possibles.
- Découper l'image en plusieurs parties et la tâche prétexte sera de trouver l'ordre de la partie dans l'image via un classificateur.

Un problème qu'on peut rencontrer avec ces méthodes est que le modèle apprenne des caractéristiques visuelles seulement utiles pour la tâche prétexte et inutiles pour la « downstreak task », bien souvent en prenant des raccourcis lors de l'apprentissage initial.[9]

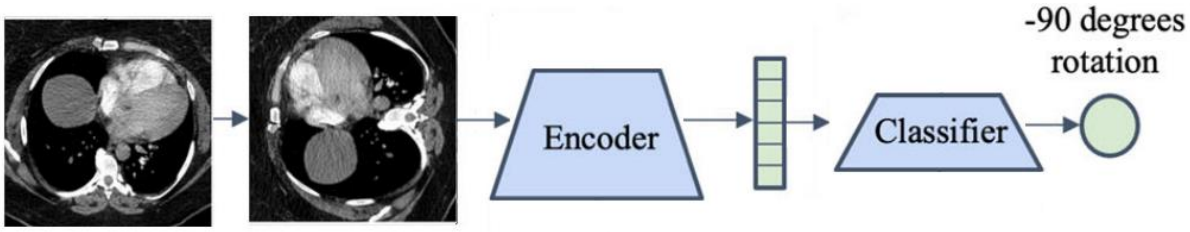


Figure 3.3 : voici une illustration de la méthode relation innée où la tâche prétexte est de déterminer la rotation de l'image originale.

Dans ce cas de figure, le rôle de l'encodeur sera d'apprendre, à partir de l'image, des représentations utiles, puis le but du décodeur (ici un classificateur, à ne pas confondre avec le classificateur de la « downstream task ») est de déterminer quelle transformation l'image a subie, par exemple, quelle est la rotation qui a été appliquée à l'image ? La fonction de perte sera une simple fonction de perte pour une tâche de classification. Si on reprend l'exemple d'une rotation appliquée à l'image, alors celle-ci devra déterminer si la prédiction du modèle correspond bien à la vraie rotation appliquée (chaque rotation différente est une classe).

3.1.3 SSL Generative

Cette partie est inspirée et a pour source : [8]

Le principe de la méthode « SSL generative » est assez simple, on va appliquer une distorsion sur une image et on va entrainer le modèle pour qu'il arrive au plus proche de l'image originale. On peut également utiliser cette méthode pour créer des données synthétiques (de nouvelles données) à partir de données existantes. Il existe plusieurs sortes de distorsions possibles.

- La distorsion peut être de rajouter du bruit à l'image d'origine et la tâche prétexte devra retirer le bruit tout en rendant l'image la plus proche possible de l'originale.
- Diviser l'image en plusieurs parties, mélanger ces différentes parties et voir si la tâche prétexte peut reconstruire l'image en mélangeant les différents patchs.

Le modèle est composé de 2 parties.

L'encodeur va, à partir de l'image altérée, apprendre des représentations utiles. La partie décodeur, son rôle sera de reconstituer l'image d'origine, via ce que l'encodeur a appris. La fonction de perte utilisée pour optimiser le modèle devra déterminer à quel point les 2 images sont proches. En méthode « generative », la transformation est appliquée à toute l'image et non pas à une partie de celle-ci (principale différence entre les méthodes « generative » et « contrastive »).

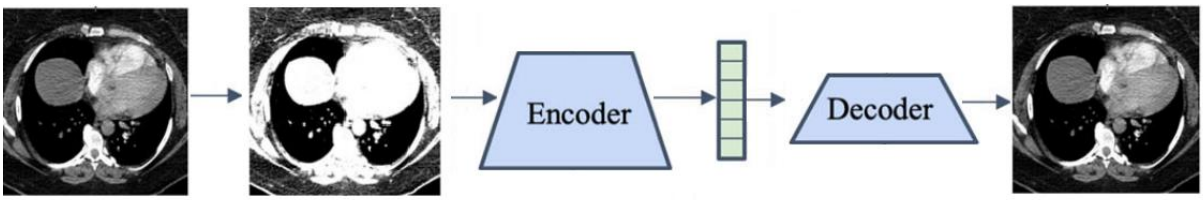


Figure 3.4 : voici une illustration de la méthode « generative » où la tâche prétexte est de reconstruire l'image pour arriver à l'image originale, après que celle-ci a subi un changement de couleur.

3.1.4 SSL Contrastive

Cette partie est inspirée et a pour source : [8]

La méthode « contrastive SSL » se base sur le principe suivant : une image a une « signification » et si on applique une ou plusieurs altération(s) sur cette image, la signification est conservée. Cela fonctionne tant que les altérations sont légères (recadrage, changement de couleur, etc...). La méthode « contrastive » va utiliser cette base en créant 2 altérations différentes de la même image. Etant donné que les images altérées viennent de la même image ; elles ont une « signification » similaire et vont former une paire positive. Si on effectue à nouveau ce procédé avec une autre image alors les 2 images altérées produites auront une « signification » différente de notre paire positive, ces images seront alors la paire négative. Le but de la tâche prétexte va être assez simple, elle va minimiser ou maximiser une « distance » (calculée par la fonction de perte) entre 2 images (la distance représente à quel point les images sont proches au niveau de la « signification »). Si on prend les 2 images de la paire positive ou les 2 de la paire négative alors la distance devrait être la plus proche possible de 0. Cependant, si on prend une image de chaque paire, alors le modèle devra maximiser la distance étant donné que celles-ci n'ont pas la même signification. Le modèle sera donc entrainé à maximiser ou minimiser la distance en fonction des images fournies.

Il y a beaucoup d'augmentations possibles, voici quelques exemples.

- Faire des rotations de l'image.
- Zoomer sur des parties de l'image.
- Changer la colorimétrie de l'image.

La tâche prétexte ne changera pas et sera toujours de minimiser ou de maximiser les distances des différentes paires. La méthode contrastive est l'une des plus populaires dans les recherches sur le SSL.

Cette méthode ne possède pas de décodeur, elle utilise directement le résultat des encodeurs pour la fonction de perte. Le rôle de celle-ci est de minimiser la distance des paires positives et de maximiser la distance des paires négatives.

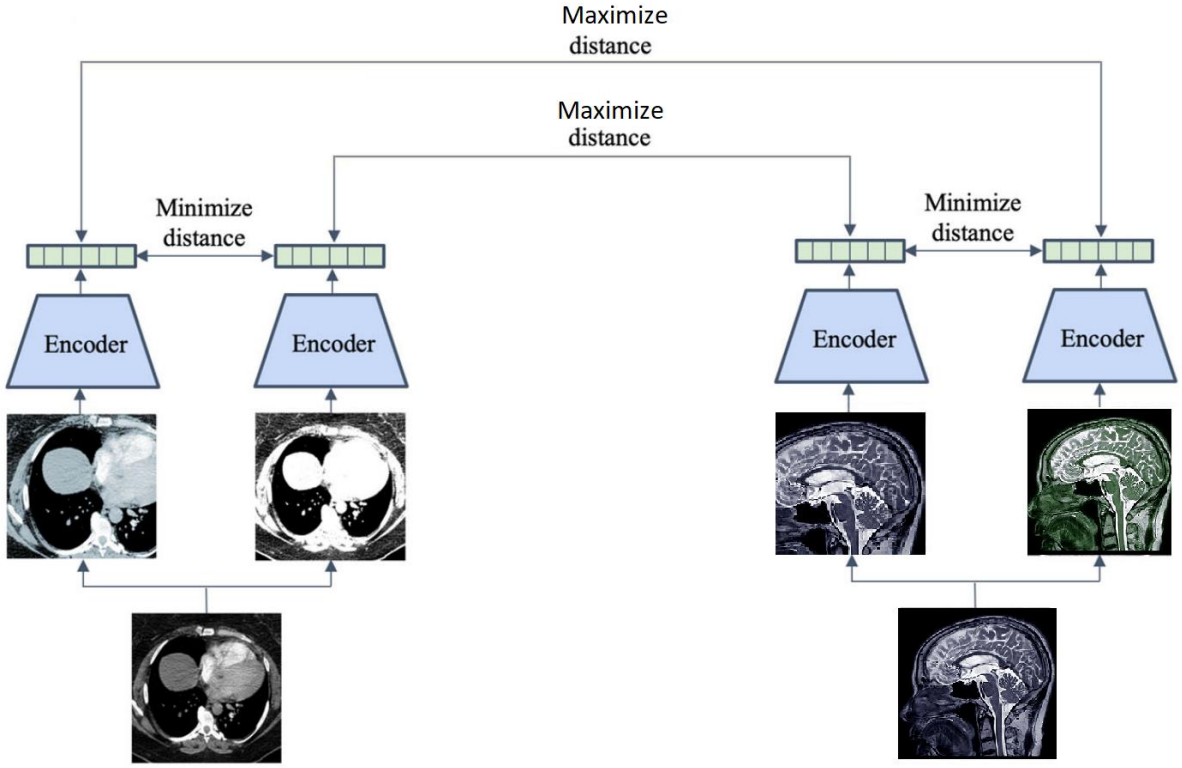


Figure 3.5 : cette image illustre parfaitement le fonctionnement de la méthode « contrastive », pour les paires positives, le modèle devra minimiser la distance, mais pour les paires négatives, on va maximiser la distance.

Cependant, on peut, avec les méthodes « contrastive », rencontrer un problème. Ce problème est lié aux principes de base sur lesquels ces méthodes fonctionnent. En effet, pour la grande majorité des images, l'application d'une altération ne change pas la signification. En revanche, dans le domaine de l'imagerie médicale, ce n'est parfois pas le cas car le sujet de l'image peut ne pas être important et seuls certains détails comptent. Certaines transformations peuvent alors changer la signification de l'image [14]. En résumé, l'altération nous fait perdre la ou les information(s) nécessaire(s) pour pouvoir différencier la classe de l'image. On peut le voir avec l'exemple ci-dessous, si nous avons une tâche de classification pour déterminer si un os a une fracture ou pas, on pourrait se retrouver dans le cas suivant où la fracture est dans la zone masquée.

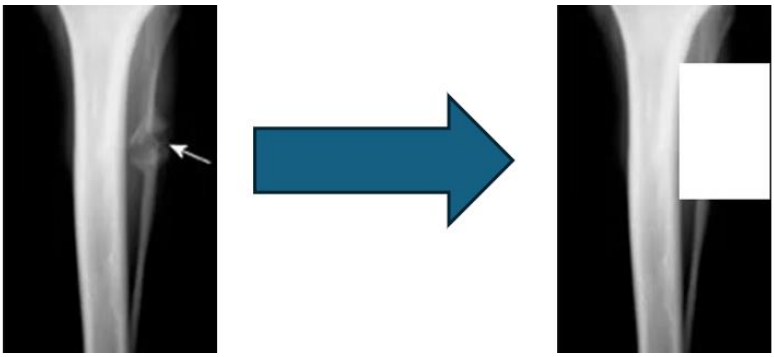


Figure 3.9 : la première image est modifiée de telle sorte que l'information est perdue et l'image maintenant représente celle d'une autre classe.

Ce problème n'est pas présent quand le sujet de l'image est important car la transformation n'est pas assez forte pour masquer entièrement l'image, laissant alors assez d'informations pour que le sujet puisse être identifié.

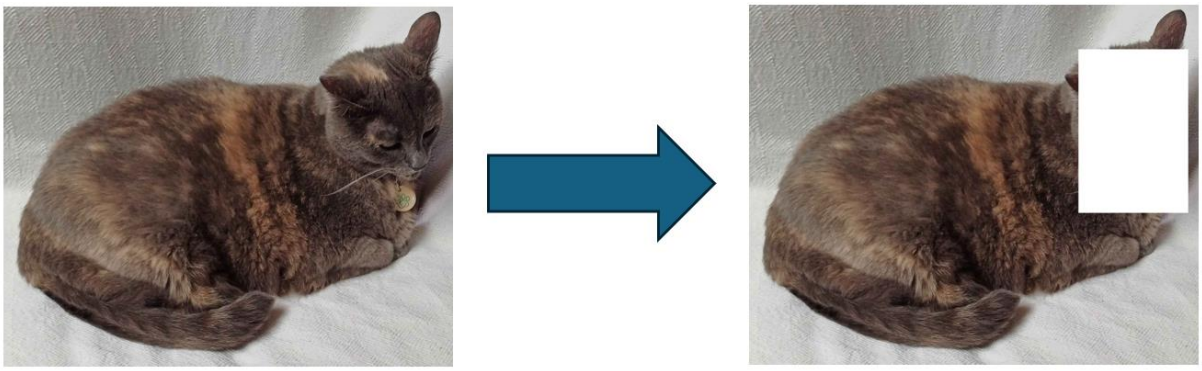


Figure 3.10 : dans le cas où le sujet est l'élément principal de l'image, même après une transformation, le sujet reste suffisamment important pour ne pas perdre la classe à laquelle il appartient.

En conclusion, il faut être attentif aux altérations utilisées par les méthodes contrastives dans certains cas notamment celui de l'imagerie médicale et si les informations nécessaires pour différencier 2 ou plusieurs classes sont liées à des détails alors il faudra faire attention si une méthode contrastive est utilisée.

3.2 Fine-tuning

L'entrainement du réseau de neurones sur la tâche prétexte a pour but que ce dernier arrive à apprendre des paternes qui seront utiles pour la « downstream task », c'est-à-dire la vraie tâche que le modèle devra effectuer après son entrainement. Pour ça, on va retirer du modèle la tâche prétexte, il va alors nous rester la « backbone » (qui est un réseau de neurones entrainé) et on va venir y ajouter un réseau de neurones supplémentaire (qui lui n'est pas entrainé). Celui-ci aura pour objectif d'accomplir la vraie tâche, par exemple un classificateur. Ce dernier sera entrainé avec le « fine-tuning », l'entrainement « final » du modèle sur un ensemble de données labellisées mais qui sera bien moins important que les données disponibles utilisées pour entrainer le modèle SSL. Il existe 2 stratégies pour entrainer le modèle.

- *End to end fine-tuning [8]*
    - On va entrainer le nouveau modèle et tous les poids peuvent être modifiés, autant ceux de la « backbone » que du nouveau classificateur. En général, cette méthode utilisera plus de ressources dû au nombre de paramètres à modifier.

- *Extract feature for classifier [8]*
    - On va entrainer le nouveau modèle mais seuls les poids du classificateur pourront être modifiés, ceux de la « backbone » sont « gelés » (ceux-ci ne sont pas modifiés durant l'entrainement) et sont laissés tels quels.
      Cette méthode est en général plus rapide car le plus grand nombre de paramètres sont gelés (et donc non modifiés), le classificateur ne comporte qu'un nombre limité de paramètres.

Le « fine-tuning » est crucial pour que le modèle puisse donner de bons résultats. Si on utilisait uniquement la « backbone », le modèle ne saurait pas réaliser la « downstream task », c'est pour ça que l'ajout d'un modèle au-dessus de celle-ci est important. Ce dernier permettra d'utiliser les données apprises par la « backbone » pour la vraie tâche ; vu qu'elle a été entrainée avec la tâche prétexte, il ne reste plus qu'à faire de légers ajustements pour arriver à un bon résultat. Lors de l'entrainement de la vraie tâche, on utilisera des données labellisées mais en moindre quantité par rapport à un modèle supervisé classique. Au cours de ce travail, les 2 méthodes de « fine tuning » seront testées et comparées, il va être intéressant de voir comment l'environnement et certains paramètres affectent les résultats de ces 2 méthodes.

3.3 Comparaison entre l'apprentissage auto-supervisé et les approches supervisées

Cette brève partie va se focaliser sur une comparaison entre l'apprentissage supervisé et l'apprentissage auto-supervisé. En premier lieu, il est bon de rappeler que chacune des méthodes a des avantages et inconvénients, ainsi que certaines situations où il est préférable d'en choisir une plutôt que l'autre. Également, certains problèmes touchent les 2 types d'apprentissage, comme « l'overfitting ». Un autre problème est que les modes entrainés via les 2 méthodes sont des boites noires, les modèles en eux même sont impossibles à comprendre pour un humain et ne sont interprétables que par leurs résultats.

Les modèles supervisés possèdent beaucoup de qualités. On pourrait dire que la précision générale de ces modèles est très bonne, ils sont moins complexes dans leur approche que les modèles SSL et par conséquent, un peu plus simples à comprendre. Un autre avantage est que la quantité de recherches disponibles pour ce genre de modèles est très importante (car ceux-ci existent depuis plus longtemps que les modèles SSL), donnant ainsi davantage de ressources disponibles pour leur création, tests, etc...Mais leur principal désavantage est l'utilisation de données labellisées en grande quantité et que des données non labellisées ne sont pas utilisables.

D'un autre côté, les modèles SSL ont comme principal avantage d'utiliser moins de données labellisées et que les données non labellisées sont utilisables. Cela crée des gains de temps et d'argent très importants. Mais ceux-ci possèdent également plusieurs défauts. D'une part, ils sont beaucoup plus récents et ont moins de ressources disponibles à disposition (étude, documentation, etc...). On peut voir ça dans le nombre de papiers de recherche scientifique

publiés : en 2021, il y a eu pour le supervisé 18 119 papiers publiés et pour le SSL seulement 302[8]. De plus, la méthode SSL complexifie les modèles, ceci est dû à la tâche prétexte qui, en plus de rajouter du temps d'entrainement (car on entraine le modèle 2 fois, une fois pour la tâche prétexte et une fois pour la vraie tâche), rajoute une difficulté supplémentaire quant à son choix ou sa création.

Quant aux résultats, ceux-ci sont en général égaux ou moins bons que leur équivalent supervisé et également un point non négligeable, ils demandent beaucoup plus de temps et de ressources d'entrainement pour donner des résultats pertinents.

Cela peut s'expliquer par plusieurs points.

- Il faut entrainer un modèle et un classificateur contre un modèle en apprentissage supervisé.
- Certains décodeurs (pour réaliser la tâche prétexte) demandent beaucoup de paramètres et donc on « entraine » certains neurones pour rien car ceux-ci seront retirés quand on entrainera la vraie tâche.
- En général, la qualité s'améliore avec davantage de paramètres. (Voir SimCLR pour plus d'explications).

4 Comparaison des différents types de tâches prétextes

Cette partie portera sur une comparaison et une analyse des différents modèles SSL utilisés pour ce travail. Ici, on va se focaliser davantage sur comment le modèle fonctionne de manière simplifiée (seule la tâche prétexte a une explication en détails), plutôt que des explications détaillées pour chaque partie des différents modèles SSL.

4.1 SimCLR

Cette partie est inspirée et a pour source : [9]

Le premier modèle en apprentissage auto-supervisé que nous allons comparer est le SimCLR, une méthode contrastive qui selon les auteurs, à la date de publication en 2020, est aussi bonne que de l'apprentissage supervisé ; cependant, sous certaines conditions : que le modèle soit peaufiné « fine tuning » et que la « backbone » soit un modèle ResNet 50 (variante plus profonde et plus grande que ResNet18, mais basée sur le même principe).

Les auteurs font également plusieurs affirmations quant aux résultats et performances des modèles SSL contrastifs (surtout pour SimCLR).

- « Pour les méthodes contrastives, le choix de composition d'augmentation de données (les transformations effectuées sur les images) est crucial pour avoir des résultats fiables. » [Citation 9]
- « Les méthodes contrastives donnent de meilleurs résultats et bénéficient davantage d'avoir une taille de « batchs » et un nombre « d'epochs » plus important que les méthodes supervisées. » [Citation 9]

La méthode utilisée par SimCLR est légèrement différente que celle présentée dans la partie sur les tâches prétextes de ce travail mais le principe reste tout de même relativement proche. On prend 2 images issues d'une image parente et au lieu de minimiser la distance entre elles, on va plutôt maximiser un « accord ». L'accord représente à quel point les 2 images sont proches, ce qui est le même principe que de minimiser une distance entre 2 images.

Par exemple, si on prend 2 images, I_1 et I_2 formant une paire positive provenant d'une image I, alors, on s'attend à ce que le modèle maximise l'accord entre les images: $Maximise(Accord(I_1,I_2))$

Une autre différence se passe dans le choix des paires positives et négatives ; la paire positive est composée de 2 images toutes deux résultantes d'une augmentation d'une image parente, cette augmentation est différente suivant l'image. Pour ce qui est des paires négatives, c'est assez simple, tout le reste du « batch ». Avec un « batch » de taille N, nous aurons un total de 2N images (la moitié d'entre elles, des augmentations), dont 2 images qui constitueront la seule paire positive et les 2(N-1) images restantes, seront les paires négatives. Une autre différence présente dans l'augmentation des données est que les images augmentées subissent 3 transformations : un découpage aléatoire et redimensionné à la taille de l'image d'origine (avec parfois un « flip » aléatoire), une distorsion de couleur aléatoire et pour finir l'ajout d'un flou gaussien aléatoire. Un flou gaussien est un type de flou utilisant une fonction gaussienne que voici [29].

$$G(x,y) = \frac{1}{2\pi\,\sigma^2} e^{-\frac{x^2+y^2}{2\,\sigma^2}}$$

Avec :

- $x, y,$ la distance respectivement la distance depuis l'origine (0,0) en abscisse et ordonnée.
- σ, l'écart type de la distribution gaussienne (plus cette valeur est élevée plus le flou sera fort).

Sans cette combinaison, le modèle n'aura pas de bonnes performances car une seule augmentation ne suffit pas.

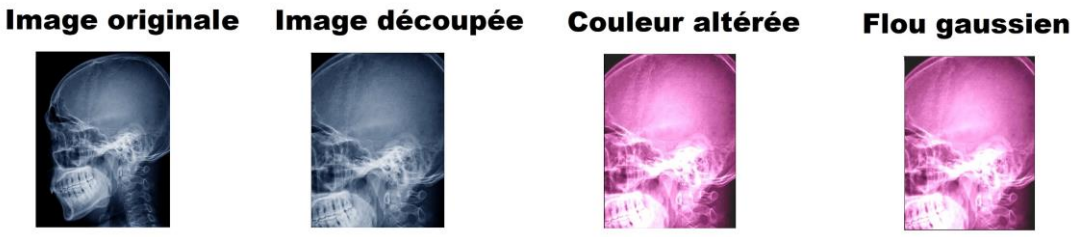


Figure 4.1 : exemple des 3 transformations appliquées sur une image de crâne, on voit que les transformations sont cumulatives.

Nous allons maintenant passer à une explication plus en détails de SimCLR.

Augmentation de données.

A partir d'une image originale notée x, nous allons appliquer 2 transformations différentes : t , t' (provenant toutes deux d'une même famille de transformation T) donc telle que : $t \sim T$ et $t' \sim T$, elles seront toutes les deux une succession de 3 transformations simples (découpage aléatoire,

distorsion de couleur aléatoire, ajout d'un flou gaussien). Ces 2 transformations donneront donc deux images notées: $\tilde{x}_i$, $\tilde{x}_j$ qui formeront la paire positive.

L'encodeur.

L'encodeur noté $f(.)$ est formé en utilisant un réseau ResNet18, il transformera les images $\tilde{x}_i$, $\tilde{x}_j$ en représentation vectorielle h_i et h_j : $h_i = f(\tilde{x}_i) = ResNet(\tilde{x}_i)$. La dernière couche de l'encodeur avant la « projection Head » est une couche de « pooling » moyenne.

La projection Head (ou décodeur).

La « projection Head » ou le décodeur, sera notée $g(.)$, celle-ci sera très petite comparée à l'encodeur, son objectif est d'effectuer une projection ou « mapping » en anglais, les représentations h_i et h_j en une autre représentation : z_i et z_j pour y appliquer la fonction de perte, la transformation effectuée est la suivante : $z_j = g(h_j) = W^{(2)}\sigma\,(W^{(1)}\,h_j)$, avec σ une fonction d'activation ReLu non linéaire. Cette partie résout la tâche prétexte et sera enlevée une fois le modèle entrainé.

Fonction de perte contrastive.

La fonction de perte contrastive, sera appliquée au résultat du décodeur.

Elle utilisera les valeurs z_i et z_j pour calculer le résultat de la fonction de perte via la formule suivante :

$$l_{i,j} = -log\frac{\exp\,(sim(z_i, z_j)/\tau)}{\sum_{k=1}^{2N} \mathbb{1}_{[k\neq i]}\exp\,(sim(z_i, z_j)/\tau)}$$

Avec :

- $\mathbb{1}_{[k\neq i]} \in \{0,1\}$, une expression logique permettant d'évaluer si $k \neq i$, cela permet de ne pas inclure z_i si $= i$ dans la somme (car on cherche à comparer z_i au reste des éléments, et non à lui-même).
- $sim(u, v)$, représente cette fonction :$\frac{u^\tau v}{||u||||v||}$ (appelée similarité cosinus), représente la similarité entre 2 vecteurs (entre -1 et 1).
- τ, un paramètre de température ajustable, sa valeur optimale change en fonction du batch size.
- $\exp(x)$, fonction exponentielle, équivalente à e^x
- Le but de cette fonction de perte est de comparer les 2 images et de mesurer leur niveau de similarité (via notamment sim), le but étant de se retrouver avec le plus de similarités possibles.Celle-ci a déjà été utilisée dans d'autres travaux et est appelée par les auteurs NT-Xent, pour « the normalized temperature-scaled cross entropy loss »
- Le résultat : la fonction de perte sera utilisée pour pouvoir entrainer les parties $f(.)$ et $g(.)$ du modèle. Une fois le modèle entrainé et la partie $g(.)$, les représentations h_i et h_j seront utilisées par le nouveau réseau pour la vraie tâche.

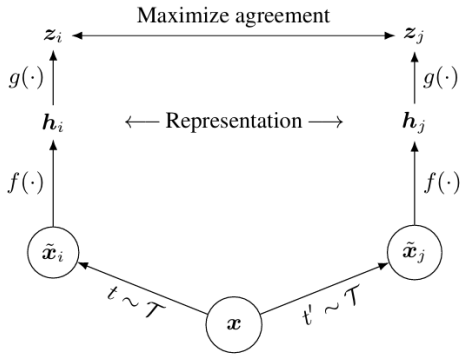


Figure 4.2 : schéma récapitulatif du fonctionnement de SimCLR.

Le schéma ci-dessus récapitule le fonctionnement de SimCLR. On démarre avec une image x qui est utilisée pour créer 2 sous images ($\tilde{x}_i$, $\tilde{x}_j$) à partir de transformations différentes (t, t'). Le modèle créera, à partir des images, 2 représentations (h_i,h_j) qui serviront de base pour que le décodeur puisse donner un résultat (l'accord entre z_i,z_j). Il sera ensuite comparé au résultat attendu avec la fonction de perte NT-Xent. Ce processus va permettre au modèle d'apprendre des représentations utiles pour la « downstream task ».

4.2 MoCo

Cette partie est inspirée et a pour source : [10][11]

MoCo ou « Mometum Contrast » est une autre méthode contrastive pour l'apprentissage auto-supervisé, elle est assez similaire à SimCLR, dont la fonction est de maximiser la similarité entre les paires positives et la minimiser entre les paires négatives (rôle de la fonction de perte). Cependant, même si le principe est très proche, le fonctionnement de MoCo est différent. La version utilisée pour ce travail est la version MoCoV2, une amélioration de la première version, mais le principe de fonctionnement reste quasiment le même.

Principes de fonctionnement de MoCo

Nous prenons une image et à partir de celle-ci, on crée plusieurs augmentations, nous allons en prendre 2 et elles formeront la paire positive. Le modèle MoCo est « divisé » en 2 parties, tout d'abord une partie encodeur qui va recevoir une image x (une des images de la paire positive), elle sera appelée x^{query}, elle va transformer l'image en une $query\, q$ (un vecteur de caractéristique), via l'encodeur f_q. On a un second encodeur, « Momentum encodé », noté f_k, à partir de l'autre image de la paire positive, il va créer des « clés » (x_0^{key}, x_1^{key} ….) ; une représentation de l'image. Si on compare la « query q » et une clé, par exemple x_0^{key}, elles proviennent toutes les 2 d'augmentations d'une même image et sont, en toute logique, assez similaires (car elles proviennent de la même paire positive). Le « Momentum encodé » est une copie de l'encodeur mais la mise à jour de ses paramètres est très différente (voire dans la partie suivante pour de meilleures explications).

Ces clés vont être encodées dans une queue, il faut voir la queue comme une liste LiFO, c'est-à-dire que si la queue est pleine, le dernier élément va être retiré. Cette queue est appelée dictionnaire. Le but de MoCo va être d'entrainer le réseau de neurones pour qu'il arrive à maximiser la similarité entre certaines clés et la « query q » : on associe les clés et la « query » provenant d'images de paires positives, les autres clés présentes dans le dictionnaire sont vues

comme clés négatives et donc pour elles, il faudra minimiser la similarité (la fonction de perte permettant cela est expliquée plus tard). Détail très important, seul l'encodeur bénéficie de l'apprentissage (f_q), le Momentum encodé n'en bénéficie pas et par conséquent, ça veut dire que plus on avance dans l'entrainement, plus les premiers éléments ajoutés au dictionnaire sont obsolètes (car ils ne reflètent plus l'état de l'encodeur). C'est pour cette raison que la taille du dictionnaire est limitée. De plus, pour ne pas avoir un dictionnaire « instable » il est mis lentement à jour via la formule suivante :

$$\theta_k \leftarrow m\theta_k + (1-m)\theta_q$$

Avec :

- θ_k, les paramètres de f_k.
- θ_q, les paramètres de f_q.
- m, hyperparamètre appelé le coefficient de « Momentum ».

Cela permet à θ_k et donc f_k, une évolution de manière plus douce à θ_q et donc f_q.

Tout ceci pour éviter que les clés encodées ne soient créées avec des paramètres trop différents entre elles. Également, le dictionnaire n'est pas limité par le « batch size » (mais par un hyperparamètre), et ça permet d'avoir beaucoup de clés négatives pendant l'entrainement du modèle. (Comparé à SimCLR où là, le nombre de paires négatives utilisables est limité par le « batch size »). Nous allons maintenant passer à une explication plus en détails de MoCo.

Augmentation de données.

L'augmentation de données ne joue pas un rôle crucial dans le modèle MoCo V1, elle correspond à une suite d'augmentations aléatoires : d'abord un recadrage aléatoire de l'image, suivi d'une modification aléatoire de la couleur, d'un retournement aléatoire et pour finir une modification aléatoire des niveaux de gris de l'image. Mais pour le modèle MoCoV2, elle est plus forte et comparable à celle utilisée dans SimCLR. Cela se traduit par l'ajout de flous aux augmentations. (Mais comparé à SimCLR, pas de changement de couleur car ça réduit la précision).

Les encodeurs.

L'encodeur noté f_q aura pour tâche de transformer des données, bien souvent une image qui sera notée x^q en requête (elle représente les « feature » de l'image d'origine(une représentation)) via la transformation suivante: $q = f_q(x^q)$. Il sera de même pour le Momentum encodé, celui-ci noté f_k, sera responsable de transformer des données x^k en clé k via la transformation suivante: $k = f_k(x^k)$. Les encodeurs sont créés à partir du modèle ResNet18, f_q et f_k sont, au départ, identiques. Comme dit plus haut, seul f_q est mis à jour avec la propagation arrière et f_k est mis à jour à partir des paramètres de f_q via la formule citée au-dessus. Les données utilisées pour $x^q et\ x^k$ peuvent être de plusieurs types différents : image, patch, ensemble d'images…

La projection Head ou décodeur.

Elle correspond à un petit réseau de neurones MLP à 2 couches (couches cachées de 2048-d, avec fonction d'activation ReLu).

Fonction de perte contrastive.

MoCo , plus précisément la version 2, utilise une fonction de perte contrastive nommée InfoNCE (à ne pas confondre avec InfoNCE de MoCo V1), elle est adaptée à la recherche de requêtes dans un dictionnaire :

$$\mathcal{L}_{q,k^+,\{k^-\}} = -log\frac{\exp\left(\frac{q.k^+}{\tau}\right)}{\exp\left(\frac{q.k^+}{\tau}\right) + \sum_{k^-}\exp\left(q\frac{q.k^-}{\tau}\right)}$$

Avec :

- q : la requête , la « query ».
- k^+, k^- : respectivement les échantillons de clés similaires (positives) et de clés différentes (négatives).
- τ : un paramètre de température ajustable.
- exp : la fonction exponentielle

Le but de cette fonction de perte est de maximiser la similarité entre la requête q et une clé k^+ et à minimiser la similarité avec une clé k^- . Plus la requête q et une clé k sont similaires, plus la valeur de retour sera proche de 0.

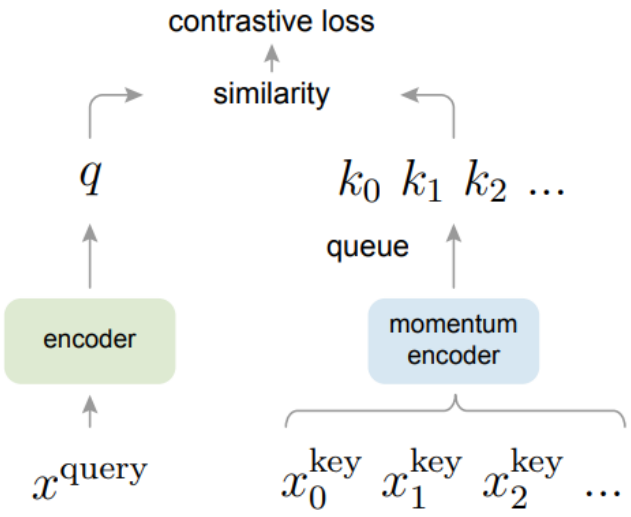


Figure 4.3 : image représentant le fonctionnement de MoCo de manière simplifiée.

Cette image résume bien le fonctionnement de MoCo, à gauche la partie « query », à droite la partie dictionnaire sous forme de files. La tâche prétexte est comme dans SimCLR la partie pour maximiser les similarités, mais dans ce cas-ci, le fonctionnement est bien différent, mais l'objectif d'entrainer le modèle en « créant » une méthode qui n'utilise pas de label reste la même.

4.3 AIM

Cette partie est inspirée et a pour source : [12]

AIM ou « Autoregressive Image Models » est un modèle génératif dont le fonctionnement est très fortement inspiré des LLMs ou« Large language models ». Celui-ci, en plus d'avoir une bonne « accuarcy » sur imageNet1K (84%)[12], est, d'après ses auteurs, potentiellement une nouvelle frontière pour les « large scale vision models », car selon leurs observations, la performance ne plafonne pas.

Le modèle a 2 avantages assez importants.

- La performance augmente avec le nombre de données utilisées et la taille du modèle.
- Les performances du modèle en aval sont corrélées aux performances de la fonction objectif en amont (la tâche prétexte).

Le modèle est autorégressif : il va prédire chaque unité de sortie en fonction de l'unité précédemment générée. Il s'agit du principe utilisé par ChatGPT ainsi que beaucoup de LLMs. Quand on demande à ChatGPT par exemple d'écrire une phrase, celui-ci, en plus d'autres méthodes va écrire un mot en se basant sur les mots précédents. Si je lui demande de compléter la phrase suivante : « J'ai mangé une ..... au pomme », grâce au mot « manger » il va pouvoir compléter la phrase avec un mot tel que tarte ou un synonyme. Et cette méthode donne de très bons résultats. Cependant il est assez simple de voir comment un LLMs fonctionne avec du texte, mais comment utiliser le même principe avec des images ?

On va utiliser le même principe de prédiction mais au lieu de prédire du texte, nous allons prédire une partie d'une image. Nous allons prendre une image, la découper en différents morceaux et le modèle devra prédire un ordre d'apparition des différentes parties de l'image (pour la reconstituer). Ceci va permettre au modèle d'apprendre la structure du sujet de l'image. On se base sur le principe que si on prend le morceau d'une image il est possible, grâce au contexte présent dans ce morceau, de retrouver la suite parmi un ensemble de morceaux d'image.

Découpage des patchs.

Une image x est divisée en une grille de K patchs (ceux-ci ne se chevauchent pas) suivant un ordre spécifique, ici un balayage de l'image de gauche à droite et de bas en haut (l'ordre du balayage est fixé et identique pour toutes les images). Nous obtenons un ensemble de patchs x_k , ils seront appelés les « tokens ». Le modèle utilise ViT pour le découpage de l'image en différents patchs et pour la « backbone ». [35]

En prenant en compte l'ordre imposé, la probabilité d'une image (avec l'ordre correct des patchs) peut être factorisée avec la formule suivante :

$$P(x) = \prod_{k=1}^{K} P(x_k | x_{<k}),$$

- $x_{<k}$ représente le premier ensemble k-1 patch, il sera utilisé pour prédire les k^{th} patchs .

Fonction de perte d'entrainement.

La fonction de perte d'entrainement négative « log-likelihood » ou (NLL) sera utilisée durant l'entrainement du modèle, elle est calculée sur un ensemble de X images :

$$\sum_{x \in X} \sum_{k=1}^{K} -\log \, \mathrm{P}(x_k | x_{<k})$$

Mais pour avoir la fonction de perte utilisée pour comparer les images entre elles, il faut utiliser celle-ci. Cette fonction de perte sert à évaluer la qualité de l'image reconstituée (on entend par là l'image reconstruite suivant l'ordre des patchs donnés par le modèle) contre l'image originale.

$$\min_{\theta} \frac{1}{K} \sum_{k=1}^{K} ||\hat{x}_k(\theta) - x_k||_2^2$$

Avec :

- $\hat{x}_k(\theta)$, la prédiction du k^{-ieme} d'un réseau paramétré avec θ.
- x_k, la base de vérité de la k^{-ieme} prédiction.

L'objectif est de minimiser le carré de la distance entre la prédiction et la base de vérité.

Encodeur

Premièrement, grâce à la « backbone ViT », on découpe l'image en patchs dans un certain ordre et ceux-ci sont transformés en vecteurs 1D. Ils seront donnés à l'encodeur, mais vu que la tâche prétexte est de pouvoir déterminer l'ordre des patchs dans l'image, on doit « masquer » le « Multi-Head », sinon retrouver l'ordre des patchs serait trop simple vu que pour un patch, il peut accéder à ceux autour. Par « masquer » on entend que le « MHA » ne peut pas voir les patchs qui sont devant le patch qu'il traite. (Les patchs suivants). Cependant, le reste du ViT reste inchangé et permet au modèle d'apprendre des représentations utiles pour les différentes images.

Décodeur

Il est le « MLP Head » du modèle ViT. Le but de la tâche prétexte sera de prédire à partir d'un patch k, le patch suivant, on ne donne au modèle que le premier patch. Une fois que tous les patchs ont été prédits, on utilise la fonction de perte pour voir si le modèle a prédit les patchs dans le bon ordre. Le résultat de la fonction de perte sera calculé en fonction de l'ordre des images, c'est-à-dire à quel point le modèle est proche de l'image d'origine. Une fois l'entrainement terminé, le MLP Head sera retiré et remplacé par la « downstream task ».

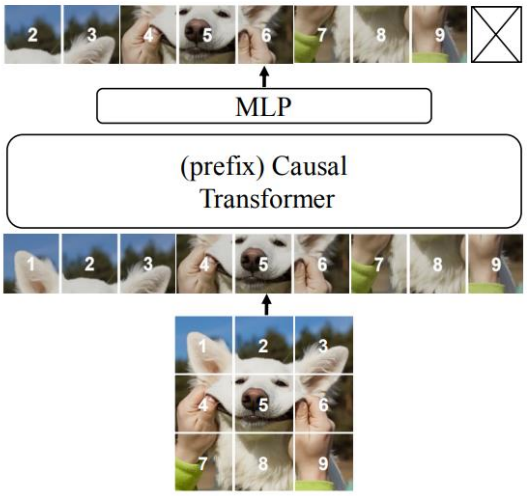


Figure 4.4 : cette image résume bien le fonctionnement de AIM, du découpage de l'image, jusqu'à la prédiction faite par le modèle sur l'ordre des patchs.

4.3.1 Remarque sur le gain de précision de AIM

Quand les auteurs de l'étude disent qu'un plafond n'a pas été observé pour AIM, il est important de replacer le contexte, on peut utiliser ce graphique ci-dessous pour voir qu'effectivement l'augmentation du nombre d'images permet d'augmenter la précision, celle-ci n'a qu'un gain très faible, seulement pour une augmentation de 1,67 %, pour une augmentation de 199 900% du nombre d'images d'entrainement. Donc ce modèle aura sûrement peu d'utilisations pour les données médicales dû au manque de données.

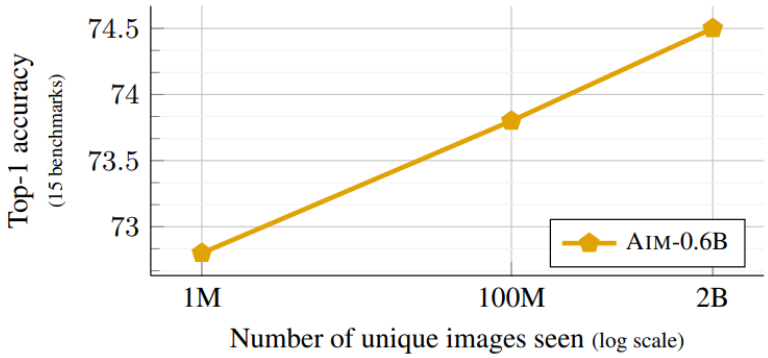


Figure 4.5 : illustration de l'augmentation de la précision par rapport à l'augmentation du nombre d'images vues par le modèle.

4.4 MAE

Cette partie est inspirée et a pour source : [13]

MAE ou « maksed auto encoder » est une méthode « self-prediction » SSL avec une approche assez particulière mais très efficace. Lors de l'entrainement, les images utilisées vont être altérées, certaines parties de l'image seront masquées et le modèle sera entrainé pour pouvoir

reconstituer l'image d'origine à partir du reste des données disponibles. MAE utilise un ViT comme « backbone ».

La partie suivante porte sur la description en détails du fonctionnement de MAE.

Augmentation de données.

Comme expliqué précédemment, le fonctionnement de MAE n'est pas très complexe. On va modifier les images en masquant certaines parties de celles-ci. Les parties masquées sont des « patchs », c'est-à-dire des carrés d'une certaine taille. Ils sont sélectionnés de manière aléatoire dans l'image puis masqués (on entend par là supprimés). L'image sera donc divisée en 2 parties, une partie visible qui sera minoritaire (15% des patchs de l'image) et une partie masquée qui sera majoritaire (75% des patchs de l'image). Cette partie est la seule augmentation de données appliquée à MAE.

Encodeur et décodeur.

Contrairement à d'autres systèmes, MAE fonctionne avec un système d'encodeur et de décodeur asymétriques. L'encodeur ne reçoit que les patchs visibles de l'image, ce qui permet 2 choses. Premièrement , de pouvoir utiliser moins de temps à calculer (vu que la partie masquée représente 75% de l'image, ne pas les traiter est un gain de temps). Deuxièmement, le modèle MAE performe beaucoup moins bien si les patchs masqués sont donnés à l'encodeur (14% moins bien sur ImageNet1K).

Le but de l'encodeur est d'extraire des représentations utiles des patchs restants. Ceux-ci seront utilisés par le décodeur pour accomplir la tâche prétexte.

Le décodeur quant à lui, est « léger » et son but est, à partir des représentations extraites de l'encodeur et des patchs masqués, de pouvoir au mieux reconstituer l'image d'origine en recréant les patchs masqués. Par recréer, on entend donner un vecteur de pixel .

Dans leur travail, les auteurs soulignent que le décodeur de MAE est significativement plus petit que ceux d'autres architectures similaires, c'est pour cette raison qu'il est appelé un décodeur léger.

Cette méthode se base sur le principe qu'une image en partie masquée peut être reconstituée (presque à l'identique) à partir des zones visibles, mais une image possède beaucoup de zones « redondantes ». Exemple : le ciel d'une image ; pour éviter que le modèle ne trouve des « raccourcis » le nombre de patchs visibles est très bas pour forcer le modèle à trouver des représentations utiles et complexes.

Si la proportion de zones masquées était moindre, alors ça ne serait pas une bonne tâche prétexte. Il ne faut cependant pas non plus cacher trop d'images car alors le modèle ne sera pas en mesure de reconstituer l'image ou perdra trop d'informations dans le processus.

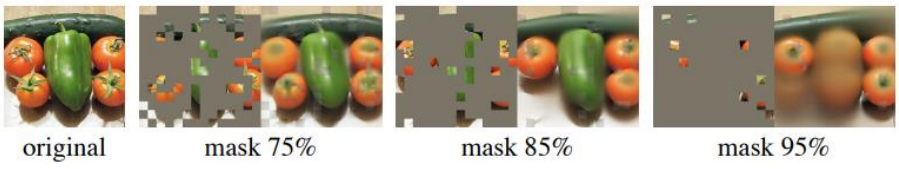


Figure 4.6 : on peut voir que si le modèle n'a pas assez d'informations, alors il va extrapoler les informations disponibles, sur la dernière image, il ne sait pas qu'il y a un poivron et va donc combler le vide avec des tomates.

Fonction de perte.

La fonction de perte sera calculée en fonction de la proximité entre l'image d'origine et l'image fournie par le décodeur. Plus précisément, en utilisant une MSE entre l'image reconstruite et l'image d'origine en utilisant la valeur des pixels. Le but étant d'arriver à une image reconstruite la plus proche possible de l'image d'origine.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

Pour n prédiction avec y_i la vraie valeur de la prédiction i et $\hat{y}_i$ la prédiction i , on peut calculer le résultat de la fonction de perte sur n prédiction.

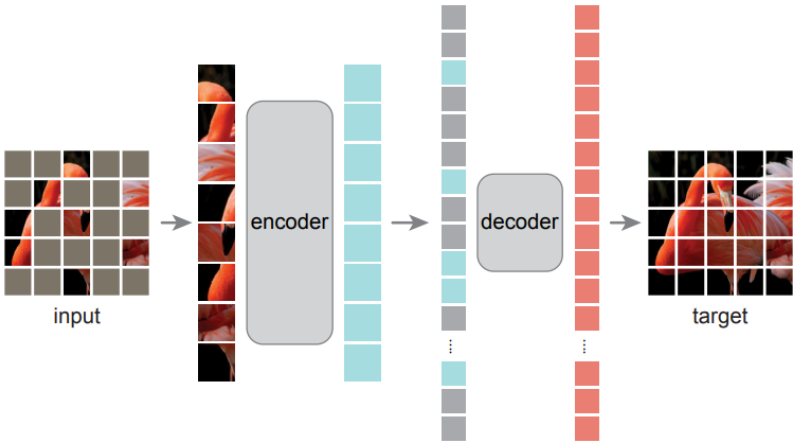


Figure 4.7 : cette image résume bien le fonctionnement de MAE, du masquage de certains patchs, jusqu'au modèle reconstituant l'image d'origine avec les données disponibles.

Figure 4.8 : exemple du processus de reconstruction de l'image à partir de morceaux de l'image originale (à gauche, l'image partiellement masquée, au centre, l'image reconstruite et à droite, l'image d'origine).

On peut observer que même avec peu de données, la tâche prétexte de MAE arrive à un résultat proche de l'image d'origine. Cependant, on observe également que la reconstruction est loin d'être parfaite. *Comme cité dans la partie de ce travail traitant des méthodes « generative », on peut se retrouver à avoir une perte d'informations en fonction des patchs masqués comme le montre l'exemple ci-dessus. Le t-shirt du surfer n'est pas visible et n'aura donc pas la bonne couleur.*

5 Méthodologie

Cette section parlera de la méthodologie utilisée lors de ce travail, que ce soit le choix des outils, des bibliothèques, en passant par le choix du « dataset » et des paramètres des modèles. L'objectif principal est de tester/de comparer différents modèles de SSL entre eux, dans un contexte d'imagerie médicale et avec un modèle étalon qui est un modèle supervisé. Le but est bien évidemment de pouvoir comparer les diverses performances des modèles (entre elles et avec le modèle étalon) pour pouvoir en tirer des conclusions sur l'utilisation de modèles SSL dans un contexte d'imagerie médicale.

5.1 Choix du « dataset »

Nous allons comparer différents modèles de SSL mais également les comparer à un modèle CNN classique utilisant des données labellisées pour avoir une base étalon.

Pour réaliser ces comparaisons, il faut choisir un « dataset » approprié, c'est-à-dire un « dataset » du domaine médical. Le dataset choisi est BloodMNIST[5], faisant partie d'un ensemble de « datasets » appelé MedMNIST[50] qui regroupe 18 « datasets » médicaux avec des domaines différents et variés (fracture, rétine, dermatologie, etc ...). Il y en a 12 en 2D et 6 en 3D, dans le cadre de ce travail ; seuls ceux en 2D seront exploités.

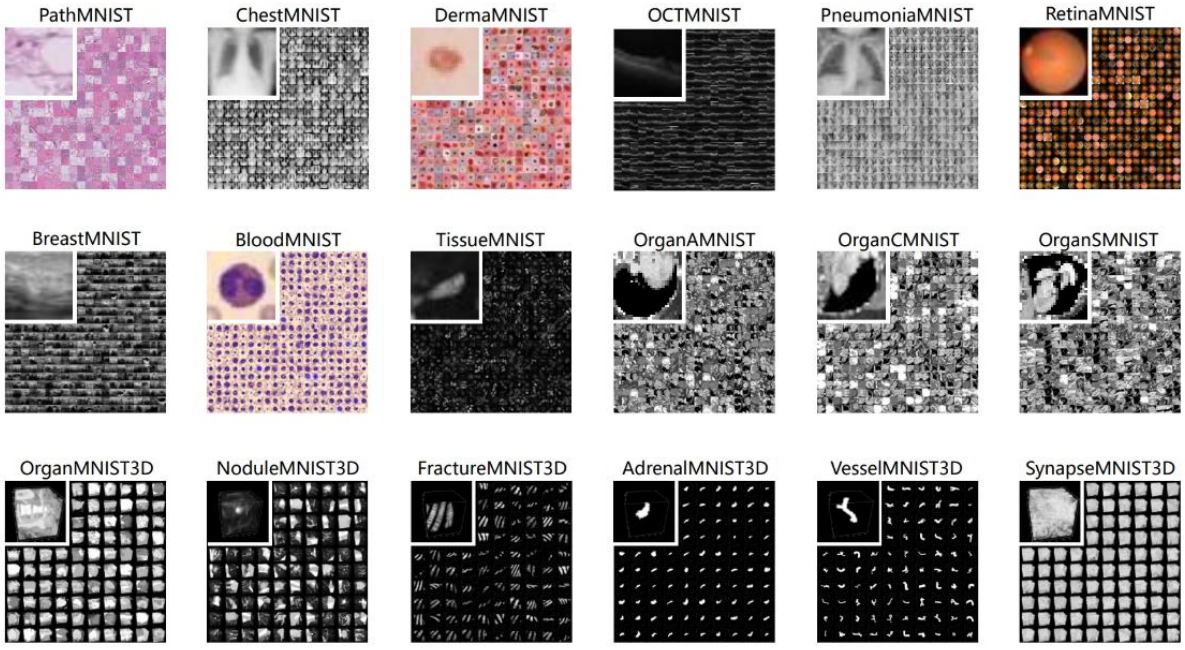


Figure 5.0 : exemple de chaque « datasets » présent dans MedMNIST

Le choix de MedMNIST comporte différents avantages.

- Taille d'images standardisée entre tous les « datasets ».
- Beaucoup de diversités entre les différents « datasets ».
- Accès relativement simple.
- Taille faible (28x28) permettant des calculs rapides.

La diversité de MedMNIST provient de multiples facteurs dont voici une liste non-exhaustive.

- La taille des « datasets » varie: certains contiennent un grand nombre d'images, tandis que d'autres n'en contiennent que très peu.
- Les caractéristiques des images diffèrent: certains « dataset » sont constitués uniquement d'images en couleur alors que d'autres ne contiennent que des images en noir et blanc.
- Le type de tâches à accomplir varie d'un « dataset » à l'autre: certains « dataset » concernent des classifications multi classes, d'autres sont dédiés à la classification binaire et certains sont même multi lables.

La taille standard entre tous les « dataset » permet de pouvoir tous les tester de manière relativement simple, à noter que tous les « datasets » ne sont pas égaux en termes de «difficulté ».

Plus un « dataset » est « difficile », plus il sera compliqué pour le modèle d'obtenir de bonnes performances. On peut utiliser le tableau suivant pour voir quels sont les « datasets » les plus simples et les plus complexes, celui-ci est fourni par les auteurs de MedMNIST[5]. (Valable pour les images de dimensions 28 px).

	PathMNIST	ChestMNIST	DermaMNIST	OCTMNIST	PneumoniaMNIST	RetinaMNIST
ACC ResNet 18 (28)	0.907	0.947	0.735	0.743	0.854	0.524
ACC ResNet 50 (28)	0.911	0.947	0.735	0.762	0.854	0.528

	BreastMNIST	BloodMNIST	TissueMNIST	OrganAMNIST	OrganCMNIST	OrganSMNIST
ACC ResNet 18(28)	0.863	0.958	0.676	0.935	0.900	0.782
ACC ResNet 50(28)	0.812	0.956	0.680	0.935	0.905	0.770

On peut voir que BloodMNIST est celui qui donne de meilleurs résultats et que RetinaMNIST donne les moins bons résultats.

BloodMNIST a été choisi pour plusieurs raisons.

- Nombre de données correctes.
- Multi classes (8 classes).
- La balance entre les classes est meilleure par rapport aux autres « datasets ».
- L'un des plus simples à apprendre.

Un autre avantage de MedMNSIT est la présence d'un ensemble de validations. Celui-ci peut être utilisé pendant l'entrainement du modèle pour voir s'il ne rentre pas en « overfitting » ou pour obtenir d'autres données (si on n'utilise pas de set de validation).

Le « dataset » BloodMNIST regroupe différentes cellules présentes dans le sang et possède 8 classes.

- Basophiles (globules blancs).

- Granulocytes éosinophiles (globules blancs).
- Erythroblastes (jeunes globules rouges).
- Immatures granulocytes (globules blancs immatures).
- Lymphocytes (globules blancs).
- Monocytes (globules blancs).
- Neutrophiles (globules blancs).
- Thrombocytes (plaquettes).

Le ratio de distribution des données est de 7:1:2 pour respectivement l'ensemble d'entrainement, l'ensemble de validation et l'ensemble de tests pour tous les « datasets ». Cela a été réalisé par les créateurs de ceux-ci, aucun split n'est fait de mon côté. Pour plus de simplicité tout au long de ce travail, les classes seront numérotées de 0 à 7. Concernant le déséquilibre des classes de Bloodmnist, celui-ci est présent, mais sans être très important, la classe la plus haute possède 2330 exemples et la plus basse 849, c'est-à-dire 2,744 fois moins de données.

Les nombres de données sont détaillés ci-dessous :

- Classe 0 : 852
- Classe 1 : 2181
- Classe 2 : 1085
- Classe 3 : 2026
- Classe 4 : 849
- Classe 5 : 993
- Classe 6 : 2330
- Classe 7 : 1643

Soit un total de 11 959 images pour le set d'entrainement.

Pour ne pas utiliser qu' un seul « dataset », un autre a été choisi, DermaMNIST, celui-ci est similaire à BloodMNIST sur beaucoup de points, il s'agit également d'une tâche de classification, mais avec 7 classes, les images sont de même taille et sont également en couleur. Mais la méthode utilisée pour prendre les images n'est pas la même. La plus grande différence est le nombre de données disponibles (17K pour Blood contre seulement 10K pour Derma), de plus le déséquilibre des classes est beaucoup plus important dans DermaMNIST, on peut le voir avec le tableau ci-dessous :

- Classe 0 : 228
- Classe 1 : 359
- Classe 2 : 769
- Classe 3 : 80
- Classe 4 : 779
- Classe 5 : 4693
- Classe 6 : 99

On peut se rendre compte que le déséquilibre des classes est donc beaucoup plus important, il va être intéressant de voir comment vont réagir les différents modèles. Pour ce qui est du ratio de distribution des données, celui-ci est exactement le même que celui de BloodMNIST, les 7 classes sont les suivantes (mais seront référées aux classes de 0 ➔ 6 durant ce travail pour plus de clarté).

- Kératose actinique (lésion cutanée).
- Carcinome basocellulaire (cancer de la peau).
- Kératose séborrhéique bénigne (verrue).
- Dermatofibrome (accumulation de collagène).
- Melanocytic nevi (grain de beauté).
- Lésions vasculaires.
- Mélanome (cancer de la peau).

5.2 Données pour comparaison

Pour les différentes comparaisons, nous allons utiliser des variations de « datasets » (BloodMNIST et DermaMNIST). La partie variable est le nombre d'exemples labellisés utilisés pour le classificateur, c'est-à-dire le nombre d'exemples du set d'entrainement. Le nombre d'exemples du set de tests ne sera pas modifié.

Ces variations seront appelées « dossier x » durant ce travail. Si par exemple il est noté que le modèle a été entrainé avec le dossier Quarter, alors celui-ci n'a été entrainé qu'avec un quart des données disponibles. Le dossier Quarter est le dossier Full mais avec, de manière aléatoire, $^3/_4$ des données supprimées.

Pour ce qui est de l'entrainement des modèles SSL, nous allons utiliser la méthode suivante : le modèle sera entrainé avec toutes les données du set d'entrainement, quel que soit le dossier utilisé (mais celles-ci ne sont pas labellisées). En revanche, pour la phase de fine tuning, le classificateur sera entrainé sur les données d'entrainement du dossier utilisé. C'est fait dans le but de simuler une situation où on possède les données mais celles-ci n'ont pas encore pu être labellisées.

Voici une liste des différentes variations des « datasets ».

- Full : « dataset » original sans modification.
- Quarter : on n'a qu'un quart des données à disposition.

Concernant l'utilisation ou non d'augmentation de données dans les différents modèles (modèles étalon, SSL, etc...), celle-ci ne sera pas utilisée durant l'entrainement des modèles étalons, ni pour les entrainements des classificateurs. En revanche ; elle est utilisée durant l'entrainement des modèles SSL. Ce choix de ne pas en utiliser a été effectué pour plusieurs raison.

- D'abord, ça permet d'avoir moins de paramètres à prendre en compte et d'avoir des données plus stables qui ne dépendent pas des augmentations (on pourrait avoir le cas où certaines augmentations ne fonctionnent que très peu avec certains « datasets »).
- Il n'existe pas ou très peu de données sur les paramètres/types d'augmentation de données effectuées et leurs résultats, en général c'est induit par l'expérience et varie en fonction du type de « dataset » et de la tâche effectuée. [37]
- Si les augmentations de données avaient dû être utilisées, elles n'auraient pas été utilisées pour l'entrainement des modèles SSL, car ceux-ci possèdent déjà des augmentations prédéfinies (voir ci-dessous).

Pour ce qui est de l'utilisation de l'augmentation de données pour les données d'entrainement des modèles SSL, celle-ci n'est pas utilisée dans le but d'augmenter le nombre de données disponibles mais pour le bon fonctionnement des tâches prétextes.

5.3 Critique et limitation du « dataset » choisi

Le « dataset » choisi pour ce travail peut également être critiqué sur certains points dont notamment la taille des images choisies. Plus la qualité d'une image est grande, plus il sera possible d'en extraire des informations et par conséquent, plus celle-ci est petite, moins on pourra en extraire et donc les chances d'obtenir des résultats fiables diminuent.

De plus, les images originales (les images brutes) étaient très certainement de meilleure qualité, et cette qualité était sûrement nettement supérieure à celle présente dans les « datasets ». On peut le voir avec RetinaMNIST, dont l'image source est de 1736px X 1824px bien supérieure à la qualité maximale disponible qui n'est que de 224 px. Bien évidemment, quand on réduit la taille d'une image on perd forcément de l'information et cette information ne pourra pas être apprise par le modèle.

Le choix de « dataset » avec une taille de 28x28 px, peut sembler une mauvaise idée dû à la faible taille des images. Mais ce choix a principalement été fait pour une question de limitation de matériel, plus une image a une taille importante plus elle prendra de ressources et celles-ci sont limitées.

Pour le « dataset » Bloodmnist, on n'a que peu de pertes d'informations car ce qui nous intéresse est le sujet central, dans ce cas, quelle est cette cellule sanguine ? Mais pour d'autres comme Retinamnsit, on perd beaucoup d'éléments comme on peut le voir ci-dessous.

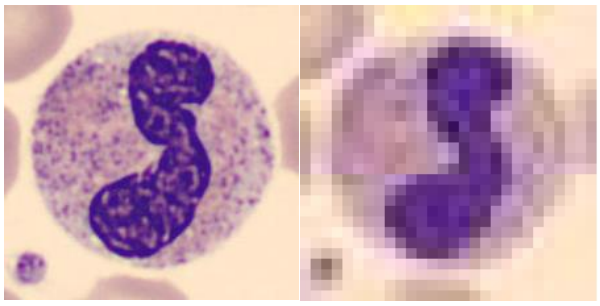


Figure 5.1 : transition de l'image de 224 à 28 pixels on peut voir que peu d'informations sont perdues car la forme est importante mais moins les détails de l'image.

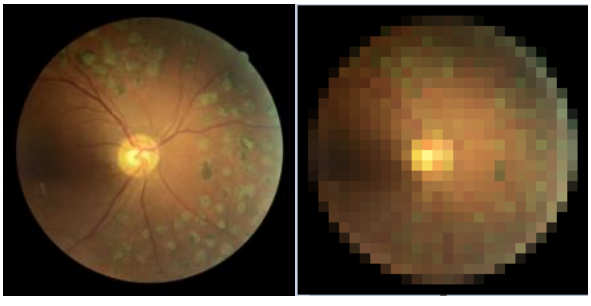


Figure 5.2 : en revanche, pour cet exemple, transition de l'image de 224 à 28 pixels, on peut voir que davantage d'informations sont perdues (beaucoup de détails importants et ici la forme n'a que peu de valeur) et qu'il sera difficile de classer l'image.

Une autre critique peut être émise sur le choix de DermaMNIST, il ne contient que très peu de données et le déséquilibre des classes est très important, mais il a été choisi justement pour tester les différents modèles SSL dans le cas où le « dataset » ne serait pas très qualitatif sur la distribution et le nombre de données présentes.

5.4 Choix de la bibliothèque et limitation du matériel

Avant de se lancer dans l'explication du modèle étalon, il faut d'abord parler des bibliothèques utilisées ainsi que des limitations techniques du matériel.

Cette section discute de la construction du réseau de neurones utilisé comme base pour les comparaisons. Pour commencer, il a fallu choisir une bibliothèque pour pouvoir créer des réseaux de neurones. Le choix s'est orienté vers la bibliothèque Pytorch.

Ce choix a été effectué pour plusieurs raisons. D'une part, la librairie pour le SSL (Lightly SSL), qui n'est compatible qu'avec Pytorch et d'autre part, Pytorch est de plus en plus utilisé dans le monde scientifique, davantage que son concurrent Tensorflow.

A noter que d'autres bibliothèques seront utilisées tout au long du modèle pour faire diverses tâches telles que la lecture d'images ou l'affichage de graphiques. Tous les scripts ayant servi à la création des modèles seront disponibles en annexes(les scripts des modèles).

Également, tous les modèles seront exécutés en local, via jupyter notebook, sur une carte graphique NVIDIA GeForce RTX 3080, moins puissante que des serveurs destinés à ce genre de tâches mais elle permet quand même des résultats assez rapides. Le processeur n'est pas utilisé car il souffre de gros problèmes de refroidissement et chauffe donc très vite.

Certains paramètres et limitations sont de mise ; par conséquent, il ne serait pas raisonnable de laisser un modèle s'entrainer pendant des heures, donc le temps d'entrainement ainsi que d'autres paramètres seront limités (comme la taille des modèles).

Une alternative qui aurait pu être utilisée est Google Colab, il n'a pas été choisi pour plusieurs raisons.

- Il utilise des GPU assez puissants, mais ceux-ci ont une disponibilité très limitée sans l'utilisation d'abonnement payant.
- Un autre très gros problème de Google Colab est celui du « timeout ». En effet, après un certain temps, la page va « timeout » (dû à une inactivité) et stopper tout processus en cours, ce qui n'est pas l'idéal surtout quand entrainer des modèles peut prendre plus de temps que le temps minimum du « timeout ».

Il est, par conséquent, tout à fait possible que les résultats changent totalement en utilisant plus de ressources, que ce soit en temps, en taille de « datasets » et surtout en puissance de calculs disponibles.

5.5 Modèle étalon pour comparaison

Cette section porte sur des points importants de la mise en place d'un modèle « Etalon » pour faire différentes comparaisons, que ce soit concernant le choix de la bibliothèque ou celui de la structure utilisée pour le modèle en lui-même.

5.5.1 Structure du réseau de neurones

Cette partie porte sur le modèle classique CNN utilisé comme valeur étalon tout au long de ce travail (le modèle pourra être trouvé en annexes). Etant donné les déséquilibres de classes, on va affecter un poids différent à chaque classe (pour ne pas avoir de problème de biais). Il y a plusieurs manières de gérer le déséquilibre des classes en CNN et en « machine learning » en général.

- La manière la plus simple est tout simplement de réduire le nombre d'éléments des classes pour arriver au nombre d'éléments des plus petites classes. Cette méthode est la moins bonne consécutivement à la perte de données et qui dit moins de données dit de moins bons résultats.
- Une autre méthode consiste à faire de l'augmentation de données, donc, à créer de nouvelles données à partir de celles existantes. On pourrait, par exemple, retourner certaines images, effectuer des zooms ou même rajouter du bruit.
- On peut également créer des données synthétiques (de nouvelles données qui ne sont pas basées sur d'autres données) pour les classes avec moins de données.

La méthode qui a été choisie pour ce travail est de jouer sur le résultat de la fonction de perte avec des poids.

La fonction de perte utilisée est « CrossEntropyLoss » et elle possède un paramètre poids (chaque classe possède un poids, en fonction du nombre d'éléments qu'elle contient). On réalise le calcul d'un poids pour chaque classe via la formule suivante :$w_i = \frac{(t-e_i)}{t}$

Avec w_i ,e_i respectivement pour le poids et le nombre d'éléments dans une classe i et t pour le nombre d'éléments total de toutes les classes confondues.

Cette méthode permet d'améliorer grandement les résultats pour les classes minoritaires et par conséquent, améliore la précision globale du modèle. Cette méthode sera la même pour l'entrainement des classificateurs en SSL et du modèle étalon. L'idée est de garder un maximum de paramètres identiques tout au long de ce travail pour avoir un environnement plus propice à la comparaison des différents modèles entre eux.

5.5.2 Construction du réseau de neurones

Au départ, le modèle étalon était un modèle assez simple, composé de 7 couches et d'un total de 3,1,520 paramètres. Celui-ci donnait de bons résultats avec en général une précision globale proche de 91% sur les prédictions de l'ensemble des tests. Il était inspiré du principe de ResNet (pour l'ordre des couches) mais était beaucoup plus simple.

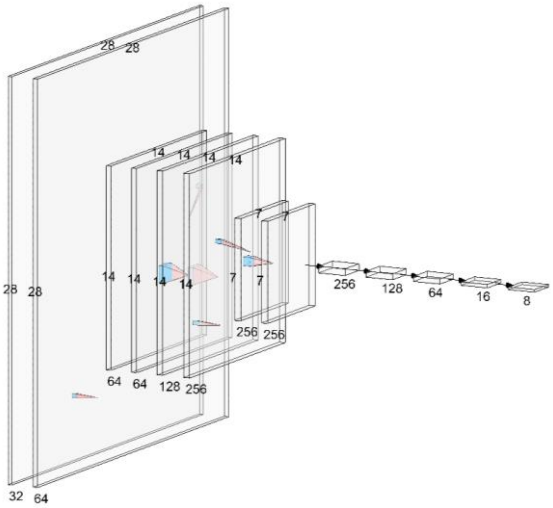


Figure 5.3 : cette illustration est une représentation visuelle du modèle dans son ensemble. Les couches de « dropout » ne sont pas représentées pour plus de clarté. La dernière couche (ici d'une taille de 8) est variable en fonction du nombre de classes dans le modèle.

Mais celui-ci a été abandonné pour un autre modèle, un modèle ResNet18 non préentraîné et ceci pour plusieurs raisons.

- ResNet 18 est plus complexe (comporte plus de paramètres) et peut donc en théorie mieux apprendre.
- Le modèle de base n'améliorait pas ses résultats si celui-ci était approfondi ou qu'on augmentait certains de ses paramètres.
- ResNet 18 donne de bien meilleurs résultats que le modèle de base.

Et la plus grande raison du choix de ResNet 18 est que celui-ci est utilisé en tant que « backbone » d'une bonne partie des modèles en SSL et donc c'est tout simplement beaucoup plus logique de comparer des modèles identiques que des modèles différents. A noter que l'utilisation d'un modèle plus simple peut également avoir de la signification, notamment pour voir si un modèle moins complexe peut donner de meilleurs résultats. Le choix d'utiliser particulièrement ResNet 18 et pas un modèle ResNet avec plus de couches (qui, si par exemple on prend ResNet 50, donne de meilleurs résultats) vient d'une limitation de matériel.

5.6 Choix des modèles

Une partie importante de ce travail a été le choix des modèles SSL utilisés, ainsi que le choix du modèle étalon. Pour ce dernier, l'utilisation de ResNet était une évidence, car celui-ci est utilisé en tant que « backbone » par certains modèles[10] et donne de meilleurs résultats qu'un réseau CNN classique. De plus, les réseaux SSL sont souvent comparés dans la littérature à des réseaux ResNet. Pour ce qui est des choix des modèles, l'objectif initial était d'avoir un modèle de chaque type de SSL et, au moins, 2 modèles contrastifs (étant le type le plus utilisé et avec le plus de ressources scientifiques disponibles aujourd'hui). Pour les modèles contrastifs, MoCoV2 et SimCLR ont été retenus. Ils ont été choisis pour plusieurs raisons mais la principale est que ce sont des modèles purement contrastifs.

Le premier modèle choisi a été MoCoV2 (version supérieure à MoCo), celui-ci est censé être meilleur que SimCLR donc ce dernier a été pris pour vérifier cette affirmation.

De plus, ces 2 modèles existent depuis quelques années et il y a plus de ressources disponibles à leur sujet. Pour le cas de AIM et de MAE, ceux-ci ont été utilisés car ils sont respectivement de type Generatif et « self-predictif ». Également, il fallait absolument que ces modèles soient utilisables par la bibliothèque choisie (Lightly SSL). De plus, la difficulté pour le choix des différents modèles est que les catégories utilisées dans ce travail ne sont pas unanimes et les auteurs ne précisent que très rarement le type de modèles SSL dans leurs études (sauf pour le contrastif). Donc, dans un souci de gain de temps, dès qu'un modèle tombait dans une catégorie, il était choisi.

Un modèle « Innate-relationship » a été envisagé mais finalement n'a pas été retenu car il n'y en avait pas dans la bibliothèque choisie .

De plus, avec 4 modèles et tous leurs dérivés, nous avons décidé que ça représentait suffisamment de données et nous avons fait le choix de ne pas en choisir un pour se focaliser sur les différents modèles que nous avions déjà.

Au travers de ce travail, il n'est malheureusement pas possible, par manque de temps et de moyens techniques, de tester tous les modèles disponibles de la bibliothèque Lightly SSL, mais il n'y a pas de doute sur l'utilité d'une telle tâche. (L'auteur de la bibliothèque les a déjà comparés mais pas sur des données médicales).

5.7 Méthodologie de tests.

Cette partie sera consacrée à la méthodologie concernant l'analyse de performances. Pour commencer, nous allons décrire la méthodologie pour l'analyse des modèles utilisés pour le point 6 de ce travail. L'idée principale est de comparer chaque modèle, de manière individuelle, au modèle étalon (sur le même « dataset » et le même dossier) mais également de comparer chaque variante des modèles SSL. Par variante, on parle du «fine tuning» (End to end fine tuning et Extract feature for classifier). Pour plus de clarté, « End to end fine tuning» sera appelé BNF et «extract feature for classifier» sera appelé BF ou non BNF.

Ces appellations font référence à l'état de la « backbone », si celle-ci est gelée ou non. Il sera intéressant de voir si les modèles donnent une meilleure précision que le modèle étalon et également quelle est la meilleure méthode de « fine tuning ».

Une fois toutes ces analyses faites, nous ferons une analyse globale pour, cette fois, comparer les différents modèles SSL entre eux. Cela permettra de voir quel modèle SSL est le meilleur en fonction des « datasets » et dossiers.

Pour l'analyse des modèles, chaque modèle sera testé sur les 2 « datasets » (BloodMNIST et DermaMNIST), et sur les 2 dossiers (Full et Quarter). La principale donnée comparée sera « l'accuracy générale » des modèles sur l'ensemble de test.

6 Implémentation des modèles

6.1 Implémentation du modèle étalon

Concernant l'implémentation du modèle étalon, celle-ci est très simple car Pytorch fournit déjà des architectures ResNet ainsi que des modèles déjà pré-entraînés sur imagenet.

Pour cette implémentation, un modèle ResNet 18 a été choisi, sans pré-entrainement, mais il a été légèrement modifié. Celui-ci était utilisé à la base pour ImageNet qui possède 1000 classes. Pour l'adapter à nos problèmes de classification de K classe, il faut tout simplement modifier la dernière couche de ResNet18. Ce n'est pas très complexe et ne prend qu'une seule ligne. ResNet50 n'a pas été choisi, car beaucoup trop complexe et long à entrainer par rapport au matériel disponible.

6.2 Implémentation des différents modèles SSL

Pour ce qui est de l'implémentation des différents modèles SSL, ils viennent directement de la bibliothèque Lightly SSL qui les fournit. Ceux-ci n'ont subi que de très légères modifications. (Si des modifications sont réalisées, elles seront bien entendu indiquées).

Les paramètres ont été choisis par la bibliothèque et ne suivent pas les recommandations des différents papiers sur les modèles SSL. Car ces derniers sont faits pour de la recherche
(beaucoup de données, de ressources, plusieurs GPU, etc...)

La bibliothèque est conçue pour des modèles tournant sur une configuration plus modeste (1 seul GPU) et donc convient très bien pour ce travail. Cependant, parfois, certains paramètres seront choisis pour réduire certaines tailles de modèles. Dans un objectif de gain de temps et de performance, le classificateur ainsi que le système d'entrainement et de chargement des données, sont repris (avec quelques modifications) du tutoriel de Lightly SSL sur MoCo[31]. L'ensemble du code se trouvera dans les annexes, avec les sources que cette partie utilise (que ce soit de la documentation ou autres).

6.3 Autres aspects techniques de l'implémentation

Cette courte section va porter principalement sur les hyperparamètres des modèles. Pour les paramètres d'entrainement, le nombre « d'epochs » sera de 100 et le « batch size » de 128 (pour des questions de matériel).

Au même titre, pour le choix du « learning rate » concernant les modèles SSL, pour ceux des modèles, ils suivent les « learning rate » fournis pour les exemples d'implémentation.

Et pour ceux des classificateurs, ils ont une valeur de 0.01 (trouvée après des tests de différentes valeurs). Pour le modèle étalon, il commence à 0.05 et est divisé par 2 tous les 20 « epochs » mais ne peut pas dépasser 0.0005. Ces valeurs ont été trouvées après différents tests pour avoir la meilleure précision. Car si on prend un « learning rate » statique de l'ordre de 0.001 ou 0.0001 alors le modèle n'arrive pas à dépasser une précision de 90%. Concernant le « Momentum », celui-ci est à 0.9 (exactement le même que dans le papier source) [21]. Pour les hyperparamètres des modèles SSL, ils sont directement repris de la bibliothèque utilisée, ils sont adaptés à une

utilisation avec un seul GPU. Aussi pour assurer la reproductibilité des modèles et des expériences, une graine aléatoire a été définie pour rendre l'aléatoire déterministe ; la valeur de cette graine est de 621.

6.4 Implémentation générale

Cette partie portera sur une brève explication du code utilisé pour l'implémentation des modèles, cette partie ne sera pas exhaustive et ne portera que sur les respects et choix importants de l'implémentation.

Pytorch fonctionne via un système de « notebooks », celui-ci est constitué de plusieurs cellules, chaque cellule peut être exécutée de manière indépendante. Les variables globales d'une cellule sont utilisables par toutes les autres cellules. Ce système permet par exemple de pouvoir séparer la zone d'exécution du modèle et la zone de traitement des données du modèle (par exemple, la zone pour afficher les graphiques relatifs aux données collectées durant l'entrainement). Cela permet aussi pour les modèles SSL de séparer la zone d'entrainement du modèle et du classificateur. Ce qui est pratique pour pouvoir uniquement réentraîner le classificateur par exemple. Le rôle de chaque cellule sera expliqué ci-dessous, pour plus de clarté, chaque cellule sera numérotée.

6.4.1 Modèle étalon

Nous allons commencer par une explication du modèle étalon.

La cellule n°1 sert pour importer toutes les bibliothèques nécessaires pour le bon fonctionnement du modèle.

La cellule n°2 est une fonction appelée « stop random ». Elle sert, comme son nom l'indique, à avoir de l'aléatoire déterministe (basé sur une « seed »), on fait ça pour pouvoir reproduire les résultats du modèle car sans cette fonction (ou une fonction similaire) un même modèle pourrait donner des résultats différents, ce qui dans ce cadre ne nous intéresse pas. Cette fonction sera également mise au début de chaque bloc du modèle. (Car elle ne marche pas de manière globale et ne fonctionne que de manière locale).

La cellule n°3 est un bloc de configuration où on choisit les paramètres suivants pour le modèle : le « dataset », son dossier, le nombre « d'epochs » et le « batch size ». Celui-ci configure également des variables pour l'enregistrement du modèle et les chemins d'accès aux données des « datasets ».

Le bloc n°4 va créer les « dataloader » à partir des « datasets, » comme dit précédemment, il n'y a pas d'augmentation de données. Les 2 « dataloader » créés sont pour l'entrainement et la validation.

La cellule n°5 s'occupe de la création du modèle, il est très petit car on ne fait que récupérer un modèle ResNet 18, celui-ci n'est pas pré-entrainé et on modifie sa dernière couche pour coïncider avec le nombre de classes de notre « dataset ».

La cellule n°6 s'occupe de gérer le déséquilibre des classes, on va calculer pour chaque classe un poids, celui-ci est compris entre [0,1], plus une classe a un faible nombre d'exemples plus son poids tendra vers 1 et inversement.

Les poids calculés sont donnés en argument à la fonction d'optimisation SGD. Ils affichent également le déséquilibre des classes via un graphique.

La cellule n°7 est la plus importante car il s'agit de l'entrainement du modèle ainsi que la collecte de données pour réaliser des graphiques (respectivement les résultats des fonctions de « loss » pour l'entrainement et la validation, ainsi que « l'accuracy » pour l'entrainement et la validation). Cette partie contient également le calcul du « learning rate », celui-ci commence à 0.1 et est divisé par 2 tous les 20 « epochs », cette valeur a été obtenue après plusieurs tests et donne de bons résultats.

Le fonctionnement de l'entrainement est assez simple. Durant chaque « epoch », on va améliorer le modèle à chaque itération (constituée d'un nombre d'exemples égal au « batch size »). L'entrainement du modèle peut être résumé en 5 lignes importantes.

- Optimizer.nograd() : on reset les valeurs de « l'opimizer » (sinon elles sont cumulatives et le modèle ne va pas arriver à s'entrainer).
- Outputs = net(input) : le modèle va prédire une valeur.
- Loss = criterion (outputs,lables) : on calcule la « loss » avec la fonction de perte à partir de la prédiction du modèle et de la prédiction attendue.
- Loss.backward() : on transfère le résultat de la fonction de perte via la propagation arrière à tout le modèle.
- Optimizer.step() : on optimise le modèle via la valeur de la fonction de « loss ».

Pendant l'entrainement du modèle, celui-ci est mis en mode entrainement (via Net.train()) et pendant la phase de validation, on met celui-ci en mode évaluation (via Net.eval()) car on ne souhaite pas l'entrainer mais simplement l'évaluer.

La cellule n°8 sert à afficher et sauvegarder les résultats des fonctions de perte et de « l'accuracy » des modèles tout au long de l'entrainement.

Les cellules n°9 et 10 servent à la sauvegarde et au chargement du modèle. Pour ne pas devoir à le réentraîner tout le temps.

Et pour finir, la cellule n°10 sert à tester le modèle sur un set de données non utilisées pour l'entrainement et la validation, le set de tests. Celui-ci n'utilise pas de « dataloader » mais lit directement les données d'un dossier. Les tests sont effectués sur beaucoup d'images pour avoir une « accuracy » la plus précise possible et sont représentés sous forme de matrice de confusion une fois les tests terminés.

6.4.2 Modèles SSL

Les prochaines parties porteront sur les différents modèles SSL, générales pour l'ensemble des modèles SSL (à part le modèle qui change, le reste des cellules sont quasiment identiques).

De plus, les cellules 1,2, 3,4,7 et 14 ne seront pas discutées car très proches de celles présentes.

La cellule n°6 est différente pour chaque modèle car il s'agit du modèle SSL. Chacun de ces modèles possède des hyperparamètres qui sont en général liés à la tâche prétexte. Pour SimCLR et MoCo, on a un hyperparamètre pour la taille de celle-ci « projection Head », MoCo possède également un paramètre pour la taille de la « memory bank size ».

Pour les modèles MAE et AIM, ils possèdent plus de paramètres pour définir la tâche prétexte ainsi que des paramètres pour le ViT (celui-ci est utilisé pour la « backbone » des 2 modèles).

Pour le modèle AIM, une partie des paramètres sert à définir le ViT qui est utilisé pour la « backbone » du modèle (les paramètres sont sur la taille du modèle, la profondeur et la taille des images qui seront utilisées) ainsi que des paramètres pour la tâche prétexte.

Pour le cas de MAE, il utilise un ViT préconfiguré et la plupart des paramètres sont, pour la taille du réseau utilisé, pour la tâche prétexte.

Les paramètres de tous les modèles ont été adaptés pour être utilisés sur un matériel assez faible et pour des images d'une taille de 28 px (sauf MAE, pour plus d'explications, voire la partie remarques sur l'intégration de MAE).

La cellule n°8 sert à définir le classificateur pour les modèles SSL. Il sera utilisé après leur entrainement. Le classificateur est presque identique pour chaque modèle, la seule différence notable est la taille du niveau qui va venir s'attacher sur le modèle SSL. Le classificateur a ainsi été construit pour avoir 2 couches et 1.1 millions de paramètres (pour pouvoir être entrainé si on utilise le « fine tuning extract feature from classifier »). La dernière couche est adaptée en fonction du nombre de classes dans le « dataset ».

De plus, le classificateur permet de récupérer, pendant l'entrainement pour chaque « epoch », les valeurs suivantes : « training » et « validation loss » ainsi que « training et validation accuracy ». Pour davantage de simplicité, ceux-ci sont stockés localement sur un fichier.

La cellule n°10 va entrainer le modèle sur le « dataset » train_model_ssl (données d'entrainement mais sans label), de plus, une fois l'entrainement terminé, on va sauvegarder le modèle et afficher la « loss » pour chaque « epoch », pour pouvoir la sauvegarder également.

La cellule n°12 va entrainer le classificateur. Pour ça, on doit d'abord récupérer le modèle SSL entrainé, puis, en fonction du fine tuning, soit le mettre en mode entrainement ou évaluation (pour le comportement de la « backbone » durant l'entrainement du classificateur). Une fois que c'est fait, il suffit de « coller » la « backbone » du modèle et le classificateur et d'entrainer le nouveau modèle. Lors de l'entrainement, le modèle affiche sa progression et divers paramètres, tout ceci est écrit dans un fichier texte car l'afficher directement sur jupyter pourrait faire « crasher » celui-ci rendant alors l'utilisation du modèle entrainé impossible. Une fois l'entrainement du modèle terminé, il est sauvegardé localement.

6.4.3 Modèle MoCo 34, MoCo Plus et Modèle fusion

Concernant les modèles MoCo 34 et MoCo_Plus, ils n'ont pas subi de modifications, sauf pour les dossiers d'entrainement utilisés pour MoCo_Plus et la taille de la « backbone » pour MoCo 34. Concernant le modèle fusion, il est légèrement différent dans sa construction car on n'a pas besoin de l'entrainer, seulement de récupérer les résultats des différents modèles. La présence dans le code du modèle des différents modèles SSL ainsi que de leur classificateur ne sert que pour pouvoir charger ceux-ci.

6.5 Implémentation du classificateur

Cette partie porte sur l'implémentation du classificateur, celui-ci n'a pas une très grande importance car la partie la plus importante du réseau se trouve dans la « backbone ». Comme dit

précédemment, le classificateur va venir prendre la place du décodeur des modèles SSL, celui-ci a pour but, avec un petit peu d'entrainement, de pouvoir utiliser ce que le modèle a appris avec la tâche prétexte pour réaliser une tâche de classification. Pour son architecture, elle diffère pour chacun des modèles SSL, mais est bien moins complexe et beaucoup moins profonde que les modèles SSL. Elles suivent toutes un modèle linéaire, celui-ci n'est pas très compliqué et est constitué de seulement 2 couches linéaires avec comme fonction d'activation ReLu et la dernière couche aura un nombre de sorties égal au nombre de classes dans le « dataset ». Concernant la première couche du classificateur, elle dépend de la dernière couche de la « backbone ». Le classificateur fait 1.6 millions de paramètres en tout.

6.6 Remarque sur l'intégration de SimCLR

Pour cette intégration lors de la transformation des données, le flou gaussien n'est pas effectué alors que celui-ci est normalement une partie très importante du fonctionnement de SIMCLR. Il a été retiré car il faisait diminuer la précision du modèle. On peut émettre l'hypothèse que les données sont trop petites pour que ce type de transformation puisse avoir un apport positif.

6.7 Remarque sur l'intégration de AIM

Avant de passer à la partie tests, il est important de donner quelques informations supplémentaires quant à l'intégration de AIM.

Certaines concessions ont dû être réalisées pour pouvoir faire des tests avec AIM car ce modèle est basé sur une dimension d'images de 224 (celle-ci a été réduite à 28), mais surtout il a été conçu pour être utilisé sur de très larges « datasets » (plusieurs centaines de millions d'images) et par conséquent plusieurs problèmes surviennent lors de l'utilisation d'un « dataset » plus petit.

L'un d'entre eux est le nombre de paramètres du modèle notamment pour sa tâche prétexte, qui, avec les paramètres les plus bas possibles, est bloqué à 34.2 millions de paramètres ; en comparaison, MoCo est à seulement 1.6 millions de paramètres pour sa tâche prétexte.

Un autre problème survient lors de la connexion du classificateur à la « backbone » (après avoir retiré l'encodeur). En effet, pour les autres modèles SSL, la taille de la sortie de la « backbone » est de 512, ou un nombre proche. Pour AIM, la taille est de 49 000 ce qui, si utilisé tel quel, va obliger le classificateur à avoir 11,5 millions de paramètres. Ceci est beaucoup trop et rendrait le modèle difficile à comparer aux autres modèles SSL. De plus, ça donne des résultats très médiocres (le modèle ne va reconnaitre qu'une seule classe). La solution qui a été trouvée est de rajouter une couche avec des poids gelés et de faire que cette couche n'ait que 512 sorties, pour après y connecter une dernière couche (même que celle présente dans les autres modèles). Cette solution n'est pas optimale, mais permet d'avoir de très bons résultats.

Voici les principaux paramètres qui ont été modifiés.

- img_size, taille d'image mise à 28.
- patch_size, taille des patchs de l'image, mis à 4 pour avoir 7 patchs.
- embed_Dim, dimension de l'espace d'encodage, c'est-à-dire la « backbone », mise à une valeur de 280 pour avoir +- 11M de paramètres pour correspondre aux autres modèles.
- Depth, nombre de couches du transformer (profondeur du modèle), mis à 12.
- num_heads, nombre de têtes pour la tâche prétexte, mis au minimal : 1

Même si les paramètres ont été pour la plupart fortement réduits pour mieux adapter le modèle à l'utilisation de ce travail, son entrainement fut le plus long de tous dû au grand nombre de

paramètres. Nous allons, encore une fois, commencer par une comparaison entre les différents modèles AIM et le modèle étalon sur le « datasets BloodMNIST», débutons avec le dossier Full.

6.8 Remarque sur l'intégration de MAE

Pour fonctionner correctement, MAE utilise comme « backbone » un ViT qui a été spécialement conçu pour la tâche prétexte de celui-ci (la gestion des patchs de l'image). Cependant, il a été prévu pour des images de taille 224 et il n'existe pas de ViT pour des images de taille inférieure.

Par conséquent, les images sont redimensionnées de 28 px à une taille de 224 px avant d'être utilisées par le modèle. Également, le changement de taille des images provoque un changement dans le nombre de paramètres du modèle et de la tâche prétexte, respectivement 88.2 et 5.1 millions de paramètres. Ce qui donne au modèle MAE un avantage comparé aux autres modèles SSL. Cependant, son entrainement ainsi que celui du classificateur ont été réduits à 50 « epochs », dû au temps que celui-ci prend pour être entrainé.

7 Analyse des résultats des modèles

Cette partie du travail portera sur les différents tests réalisés, ainsi qu'une analyse et interprétation de leurs résultats. Cette analyse se fera d'abord de manière séparée pour chaque modèle puis de manière globale avec une analyse plus générale. Le premier modèle testé sera bien évidemment le modèle étalon étant donné qu'il sert de modèle de référence. Tous les modèles seront testés sur les 2 « datasets » mais nous commencerons toujours par BloodMNIST avec une analyse poussée, puis DermaMNIST avec une analyse un peu plus simple.

Nous faisons ce choix car DermaMNIST est très déséquilibré et il est plus sage de ne pas l'utiliser comme « dataset » principal pour l'analyse. La métrique principalement utilisée pour l'analyse sera la moyenne de « l'accuracy » de toutes les classes, c'est dû encore une fois au déséquilibre de classes dans le set de tests. La fin de cette section portera sur une comparaison globale des différentes données pour déterminer quel est le « meilleur » modèle à notre disposition, mais également pour voir quel modèle est à utiliser en fonction des circonstances.

7.1 Remarque sur les graphiques

Pour analyser les différents modèles, nous allons utiliser des graphiques, ceci pour avoir une représentation visuelle plus agréable qu'un simple tableau de données. L'axe Y va représenter la précision en %, donc le pourcentage d'éléments de mon set de tests qui ont été bien classés par le modèle. L'axe X représentera les différentes classes testées ainsi que la précision globale du modèle et son « accuracy ». (Ces 2 derniers ne sont pas des classes mais seront utilisés pour les analyses). De plus, l'axe Y ne commencera pas à 0 mais à une valeur plus grande pour pouvoir mieux afficher les différences entre les modèles. (Si on a 2 modèles avec, par exemple, 89% et 90% de précision, il n'est pas utile d'afficher la précision de 0 à 80 car elle est identique partout).

7.2 Analyse des résultats du modèle étalon

Cette section est dédiée à l'analyse des différents résultats des modèles étalons avec les différents « datasets » et différents dossiers. Cette partie est donc cruciale car l'analyse de tous les modèles SSL repose sur les données de cette analyse.

Avant de commencer l'analyse de modèles, il convient de signaler une petite précision. Dans les analyses suivantes, nous comparerons le modèle étalon à divers modèles SSL BNF, c'est-à-dire des modèles dont la « backbone » n'est pas gelée (les paramètres peuvent être entrainés durant l'entrainement du classificateur).

Ces modèles seront confrontés aux modèles étalons. Comme les classificateurs sont tous entraînés sur un nombre similaire « d'epochs » comme lors de l'entrainement des modèles SSL, il faudrait donc, pour rendre l'analyse la plus précise possible, entrainer le modèle étalon sur 200 « epochs ». Mais la performance n'augmente pas, elle diminue (de l'ordre de 0.5%) donc pour cette raison, le modèle étalon n'aura qu'un entrainement de 100 « epochs » (car meilleures performances). De plus, analyser les performances des modèles SSL dans des conditions similaires au modèle étalon (c'est-à-dire avec des données labellisées avec le dossier Full) peut sembler inutile, mais l'idée derrière cela est de comparer les résultats dans des conditions similaires contre un modèle supervisé. Le dossier Quarter, lui, est là pour simuler une situation où, en théorie, un modèle SSL serait plus efficace qu'un modèle supervisé, c'est-à-dire une situation où on a beaucoup de données non labellisées mais peu de données labellisées.

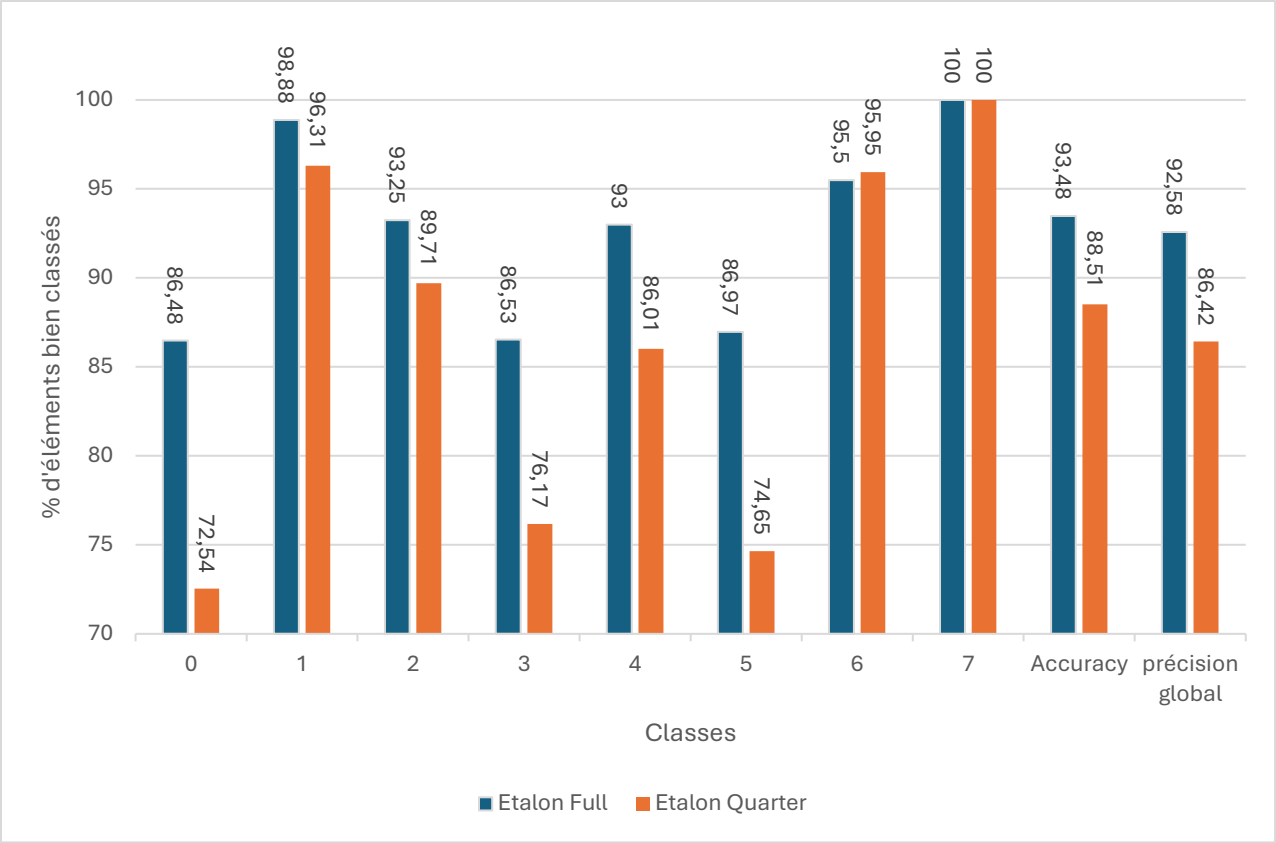
7.2.1 Analyse Dataset BloodMNIST

Nous allons commencer par une comparaison entre le dossier Full et le dossier Quarter de BloodMINST. Pour les résultats du dossier Full, on peut faire plusieurs observations.

D'abord, le modèle n'atteint pas la précision obtenue sur un même réseau de neurones (ResNet18) que celle obtenue dans le papier de recherches sur MedMNIST. On peut expliquer ça par plusieurs facteurs.

- Le nombre « d'epochs » et « batch size » inférieur.
- Il n'y a pas d'utilisation d'augmentation de données.
- La méthode de calcul du « learning rate » est différente.

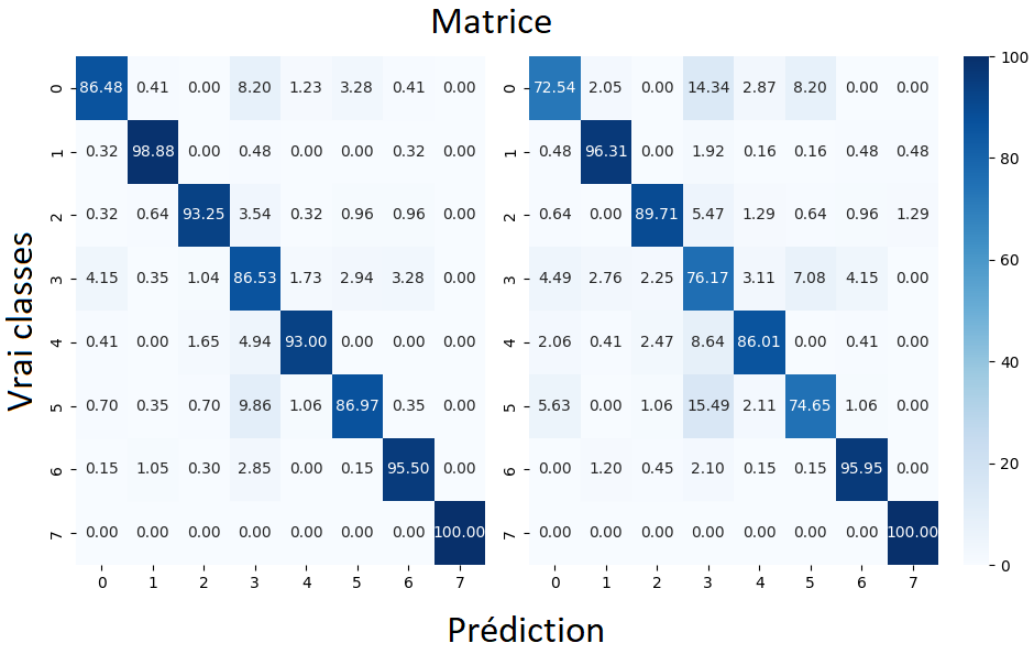
Toutefois on arrive quand même à un niveau de précision très satisfaisant avec une « accuracy » moyenne de 92,58% (contre 95.8%) pour le dossier Full. De même pour le dossier Quarter, les résultats sont impressionnants et on arrive à 86,42% de précision, soit une baisse de 6,18%.



Graphique comparant le modèle étalon entre les dossiers Full et Quarter

Observations.

On peut réaliser plusieurs observations. D'abord, le dossier Quarter donne de moins bons résultats, c'était attendu car nous n'avons que ¼ des données à disposition. De plus, on peut également observer que les classes diminuent légèrement sauf pour les classes 0, 3, 5 qui ont une diminution très importante (surtout les classes 0 et 5). On peut utiliser la matrice de confusion des modèles pour comprendre ce qui se passe pour les classes n°0, 3 et 5. Une matrice de confusion est une matrice qui permet de connaitre et de représenter les erreurs et bonnes prédictions d'un modèle. On peut également vérifier si le modèle s'est trompé dans sa prédiction et identifier la classe avec laquelle il l'a confondue.

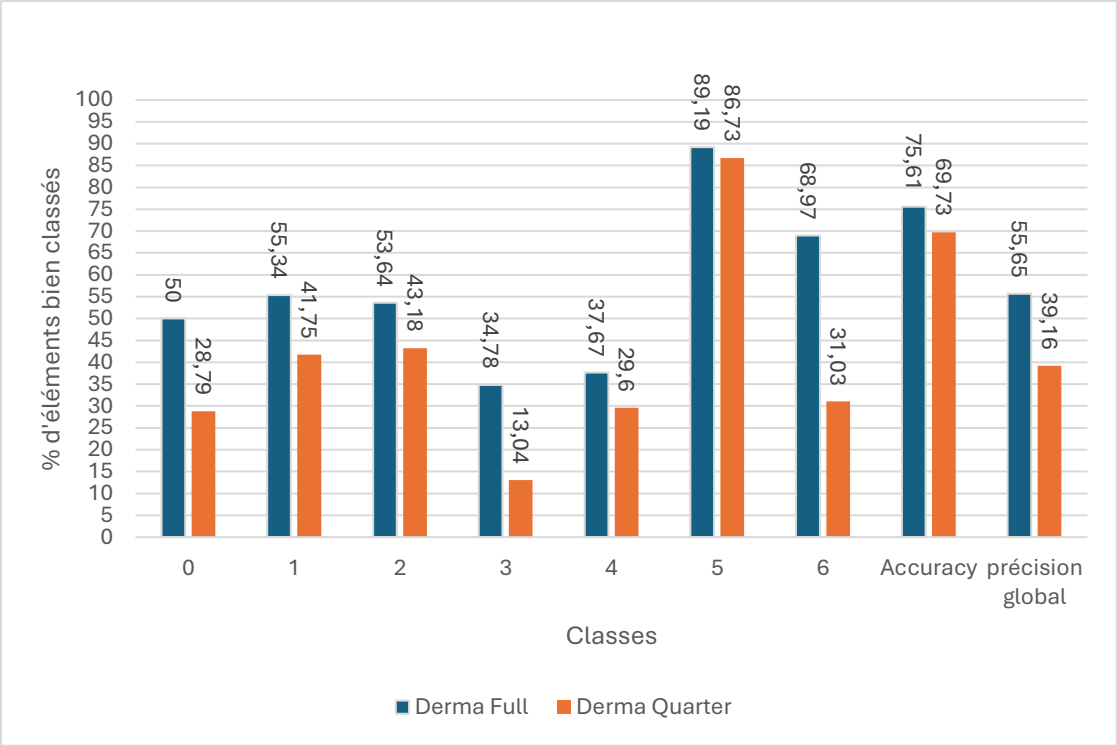


Matrice de confusion du modèle étalon Full (à gauche) et Quarter (à droite) pour le « dataset BloodMNIST » .

On peut observer que les classes n°0, n°3, n°5 sont souvent confondues entre elles par le modèle, les classes n°0 et n°5 sont confondues avec la classe n°3 qui, quant à elle, est confondue avec toutes les autres classes sauf la classe n°7. Cela peut s'expliquer par le fait que ces 3 classes en question sont des globules blancs et sont à vue d'œil assez similaires. Ceci peut expliquer la confusion du modèle entre les différentes classes car il est probable le modèle n'arrive pas à les différencier correctement. On peut également observer que pour le modèle entrainé sur le dossier Quarter, c'est bien plus important. Il va être très intéressant de comparer leurs résultats à ceux obtenus avec les modèles SSL et de voir s'ils subissent le même type de problèmes que le modèle étalon pour ce « dataset ».

7.2.2 Analyse Dataset DermaMNIST

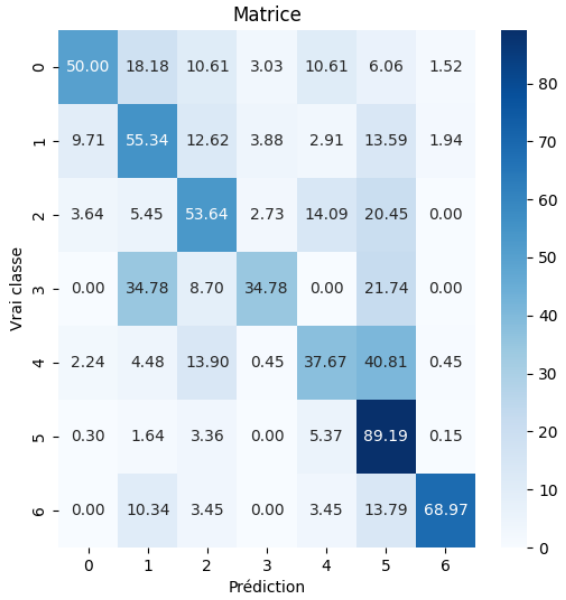
Nous allons maintenant passer à une comparaison entre les dossiers Full et Quarter pour DermaMNIST. Avec des données dans un nombre limité et un déséquilibre de classes important, on observe de très mauvaises performances du modèle. De plus, le modèle avec le dossier Quarter donne des résultats très médiocres avec une « accuracy » générale en dessous de 50%.



Graphique comparant le modèle étalon et les dossiers Full et Quarter du « dataset DermaMNIST ».

Observations

La première observation est que le modèle (dans sa configuration actuelle) ne semble pas adapté pour un « dataset » si petit, avec autant de déséquilibre de classes et cela devient encore pire avec le dossier Quarter. La classe n°5 semble être assez précise mais c'est uniquement dû au fait que le modèle confond la grande majorité des classes avec elle comme on peut le voir sur la matrice de confusion ci-dessous. Ce phénomène est une nouvelle fois encore pire quand on passe au dossier Quarter.



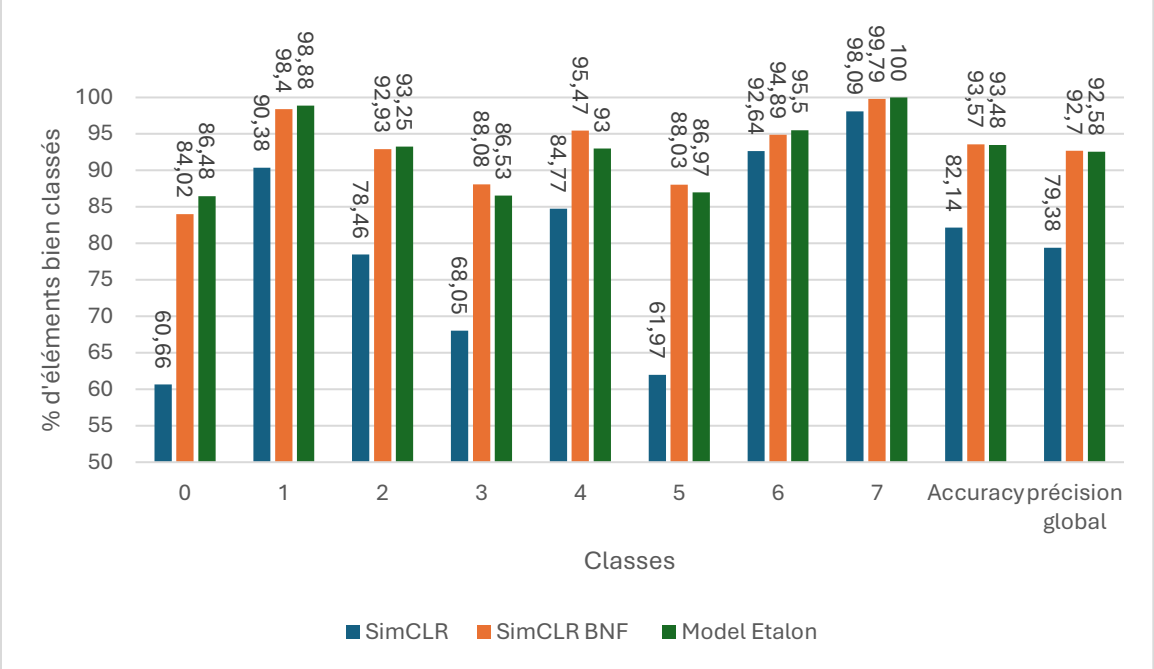
Matrice de confusion du modèle étalon Full pour le « dataset » DermaMNIST .

Observations générales.

Après avoir fait ces tests, nous pouvons tirer plusieurs conclusions et observations. On observe de façon assez simple que plus le nombre de données est élevé, plus la précision est haute et à l'inverse, plus le nombre est bas, plus la précision sera restreinte. Cette observation est tout à fait normale et classique en CNN et était prévisible. Également, le manque de données intensifie le phénomène durant lequel le modèle confond une ou plusieurs classe(s). Les prochaines analyses porteront sur les modèles SSL et il va être intéressant de voir si ceux-ci ont les mêmes réactions que les modèles supervisés plus classiques.

7.3 Analyse des résultats du modèle SimCLR

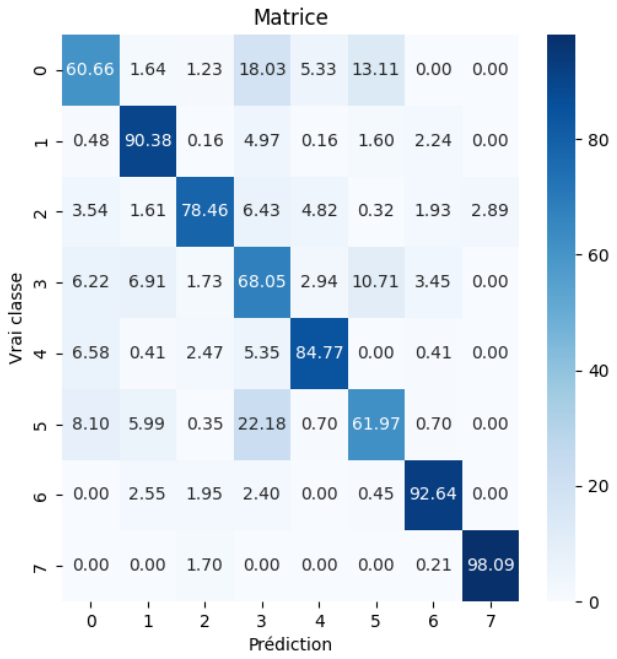
Nous allons commencer par une analyse des résultats de SimCLR. En premier lieu, nous allons comparer les modèles sur le « dataset BloodMNIST » pour le dossier Full.



Graphique des résultats, la précision est le nombre d'éléments correctement classés par classe et la précision globale est le nombre d'éléments correctement classés sur l'ensemble du set de validation.

Observations.

Nous pouvons réaliser plusieurs observations, la première est que le modèle SimCLR BNF a une meilleure prédiction générale que le modèle étalon, mais cette différence est assez faible (0.6%). En revanche, le modèle SimCLR , quant à lui, ne donne que des performances très moyennes et largement en-dessous du modèle BNF. On peut expliquer cette différence assez facilement : le modèle SimCLR n'a que peu de paramètres entrainables durant l'entrainement du classificateur comparé au modèle BNF (1 million contre12 millions). Cette différence de précision est générale mais les classes n° 0, 3 et 5 sont particulièrement imprécises. On peut regarder la matrice de confusion pour se rendre compte que le problème présent dans le modèle étalon est toujours là mais beaucoup plus important.



Matrice de confusion du modèle SimCLR sur le dossier Full du « dataset » BloodMnist.

Il est fortement possible que, dû à la ressemblance visuelle de ces 3 classes, durant la phase d'entrainement du modèle, celui-ci n'ait pas réussi à les différencier convenablement. Surtout quand on sait que même le modèle supervisé rencontrait des difficultés et que le modèle SSL n'a pas de labels à sa disposition. Il n'est pas impossible de conclure que le modèle s'est mal entrainé sur ces classes. Également, dû au faible nombre de paramètres du classificateur et à la « backbone » gelée, il n'ait pas non plus réussi à contrer ce mauvais apprentissage. Ce qui expliquerait le fait que le modèle donne largement de meilleurs résultats quand la « backbone » n'est pas gelée (car ça permet de modifier les erreurs d'entrainement du réseau de neurones).

On pourrait sûrement améliorer les résultats de SimCLR en augmentant fortement le nombre de paramètres, comme le montre la figure ci-dessous. Mais l'augmentation de la précision du modèle n'est pas linéairement proportionnelle à l'augmentation du nombre de paramètres.

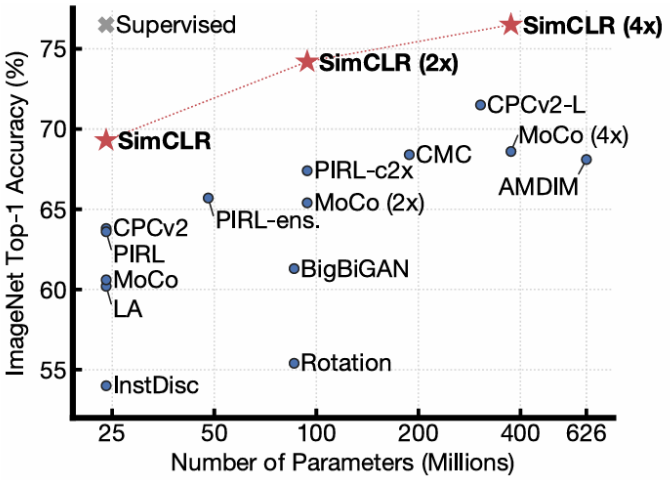
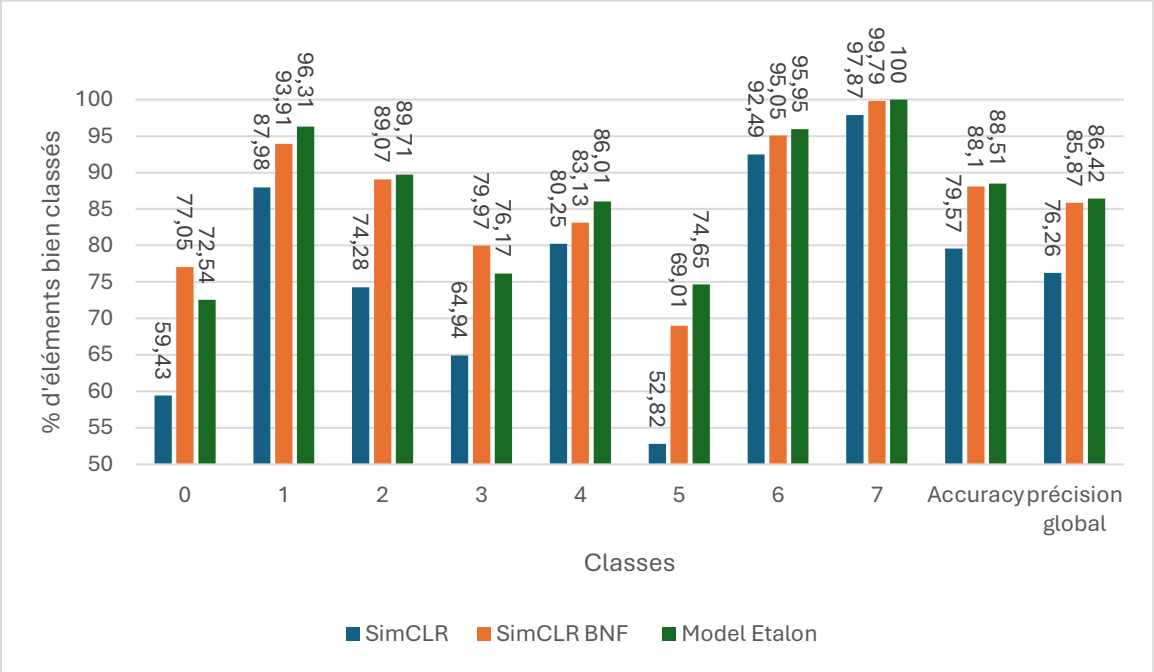


Figure 7.1: résultats de SimCLR comparés à un modèle supervisé et à d'autres modèles SSL sur ImageNet.

Nous allons maintenant passer à une courte analyse des résultats, toujours pour le même « dataset » mais avec le dossier Quarter.

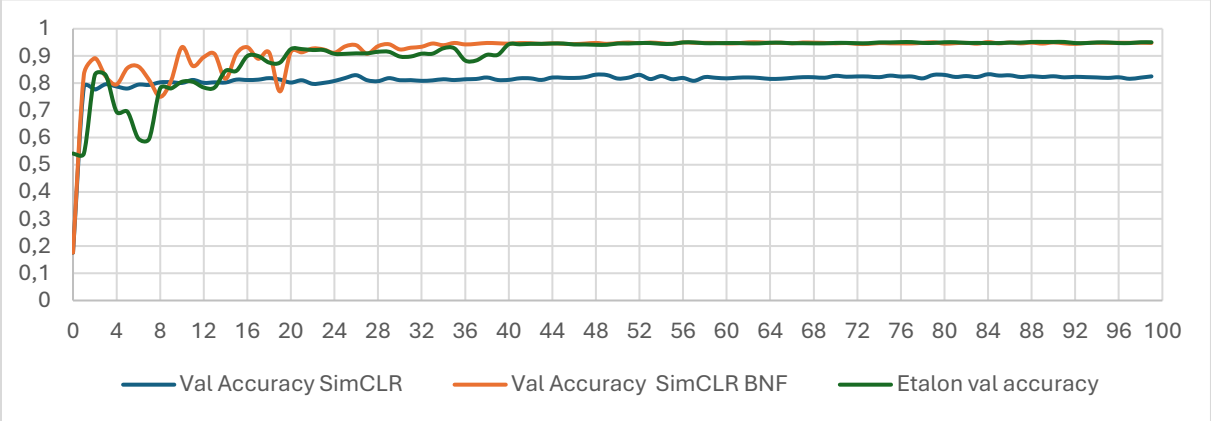


Graphique de comparaison générale entre les différents modèles SimCLR et le modèle étalon sur le dossier Quarter.

Observations.

On se retrouve avec les mêmes observations que pour le dossier Full, à la différence que pour le dossier Quarter, le modèle étalon reste meilleur avec une très faible marge de 0.55%. Encore une fois, le modèle BNF est plus performant, même si l'écart entres les 2 modèles SimCLR est plus faible. Dans de telles conditions SimCLR n'est pas à privilégier.

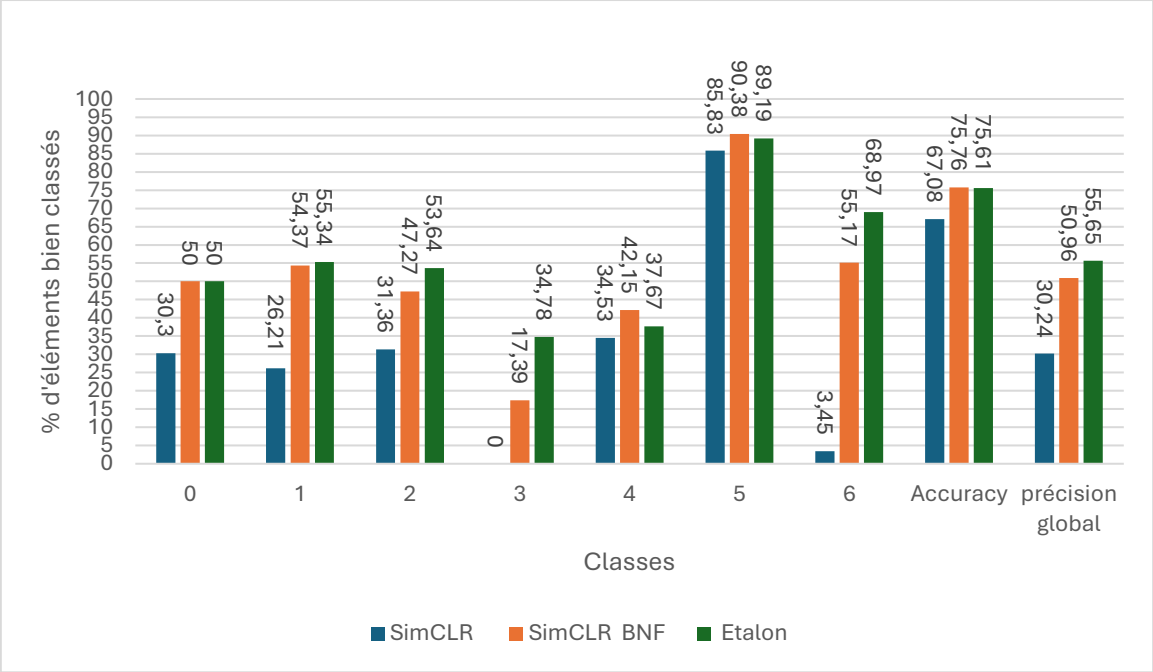
Avant de passer à l'analyse des résultats du « dataset DermaMNIST », il est intéressant d'analyser les courbes « d'accuracy » de l'ensemble de validation durant l'entrainement des divers modèles. Celle-ci nous donnera une idée de l'apprentissage des différents modèles.



Graphique comparatif entre « l'accuracy » sur l'ensemble de validation des différents modèles SimCLR et celle et du modèle étalon pour le dossier Full.

On peut faire plusieurs observations. La première est que les modèles SSL, lors de leur premier « epoch », ont une « accuracy » sur le set de validation très mauvaise, c'est potentiellement dû au fait que les paramètres du classificateur n'ont pas encore réussi à bien se paramétrer et que le résultat est alors proche de l'aléatoire. On peut également voir que le modèle SimCLR n'arrive pas à déplacer 83% de précision, ce qui est raccord avec sa précision finale. De plus, le modèle SimCLR BNF oscille beaucoup durant les 40 premiers « epochs », probablement une période durant laquelle les paramètres de la « backbone » changent et sont affinés par le modèle, il en résulte des précisions très variables.

Maintenant, nous allons passer à l'analyse du « dataset DermaMNSIT », nous allons commencer par l'analyse des résultats du dossier Full.

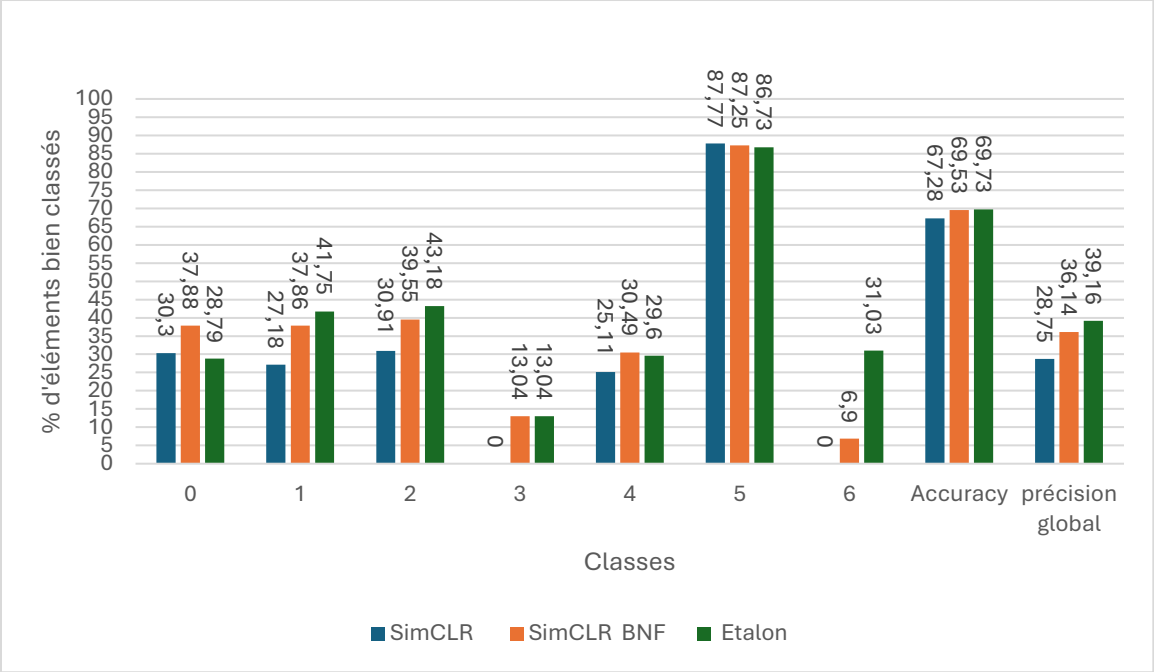


Graphique comparatif entre SimCLR et le modèle étalon pour le dossier Full.

Observations.

On peut très vite se rendre compte que le modèle SimCLR n'arrive pas à reconnaitre certaines classes, (les classes n°3 et n°6), et a une précision globale très mauvaise. D'un autre côté, le modèle SimCLR BNF donne de meilleurs résultats mais ceux-ci sont inférieurs aux résultats du modèle étalon. Il est très probable que beaucoup de classes soient confondues avec la classe n°5, qui est surreprésentée. Le modèle étalon semble rester le meilleur choix. Pour les observations du dossier Quarter, celles-ci sont très similaires, la seule différence est que le modèle SimCLR n'arrive pas à reconnaitre les classes n°3 et 6, donnant une précision de 0, et par conséquent une très mauvaise « accuracy » globale.

Nous pouvons maintenant terminer sur une analyse des résultats du dossier Quarter :



Graphique comparatif entre SimCLR et le modèle étalon pour le dossier Quarter.

Observations.

Nous pouvons réaliser quelques rapides observations.

D'abord, aucun modèle n'arrive à une précision similaire à celle du modèle étalon. De plus, l'utilisation du « end to end fine-tuning » permet à certaines classes qui avaient une précision de 0, d'avoir une meilleure précision, même si celle-ci reste très faible (notamment pour la classe n°6). SimCLR , quel que soit son modèle, a beaucoup de mal pour identifier cette classe. Et les modèles SimCLR restent en dessous du modèle étalon.

Observations générales sur SimCLR.

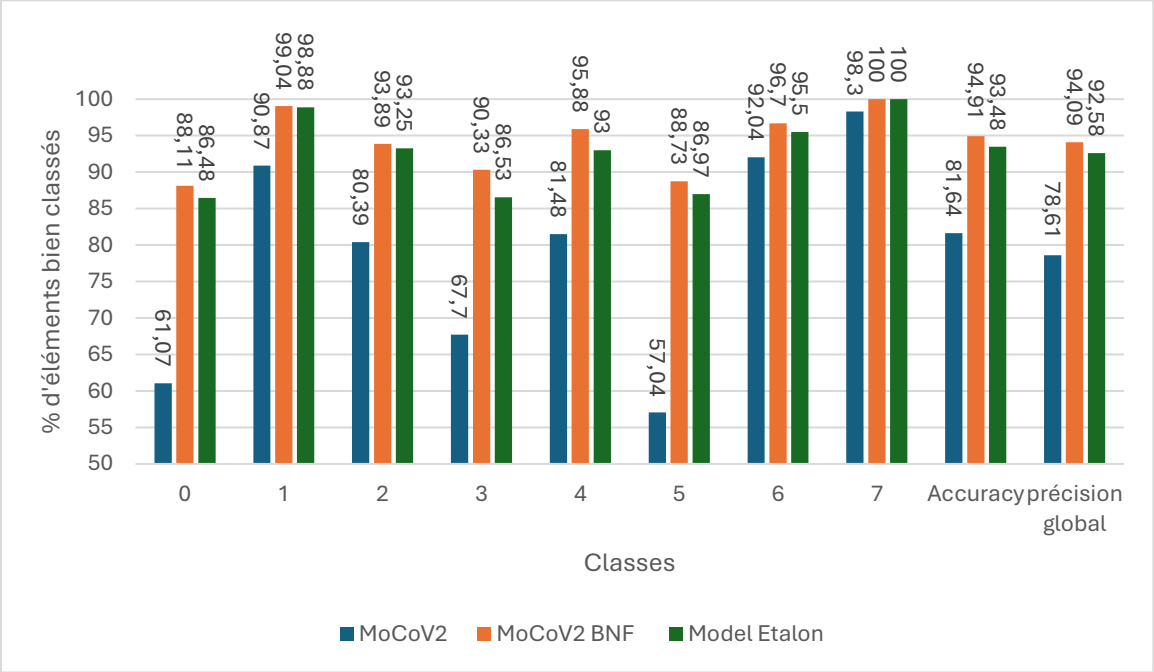
Voici quelques observations qu'on peut faire sur SimCLR.

- Si toutes les données sont labellisées et en suffisance, SimCLR fera des résultats très légèrement supérieurs ou égaux à un modèle supervisé.
- Augmenter le nombre de paramètres semble donner une meilleure amélioration des résultats[9] .
- S'il y a un déséquilibre de classes très important, alors SimCLR ne donnera pas de bons résultats.

Ce dernier point est important car il met en lumière un problème potentiel des modèles SSL durant l'entrainement du modèle. En effet, étant donné que les données ne sont pas labellisées, il est impossible de déterminer le nombre d'éléments de chaque classe et par conséquent, on ne peut pas appliquer de technique pour empêcher le modèle de « sur-apprendre » la classe majoritaire. On ne peut lutter contre le déséquilibre de classes uniquement lors de l'entrainement de la « downstream task » (ici un classificateur).

7.4 Analyse des résultats du modèle MoCo.

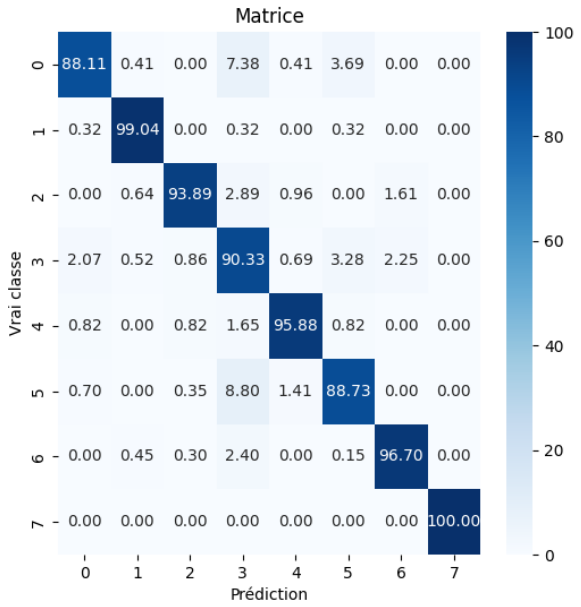
Nous allons maintenant effectuer les mêmes tests réalisés pour SimCLR mais avec MoCo V2, bien évidemment, nous allons commencer par une comparaison entre les différents modèles pour le « dataset BloodMNIST » et le dossier Full.



Graphique de comparaison générale entre les différents modèles MoCoV2 et le modèle étalon sur le dossier Full.

Observations.

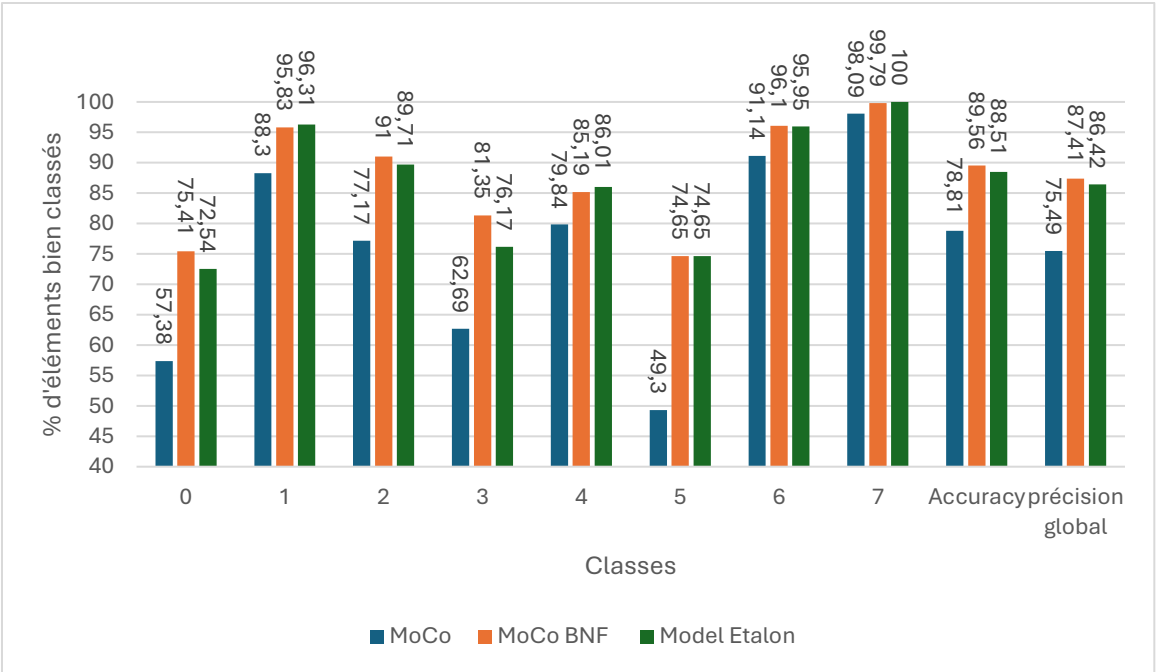
On se retrouve avec des observations assez similaires comparées à SimCLR. MoCoV2 donne des résultats en dessous du modèle étalon, avec une précision assez mauvaise pour encore une fois les classes n°0, 3 et 5. Mais MoCoV2 BNF est meilleur que le modèle étalon et d'une marge plus importante qu'avec SimCLR, de 1,51 %. L'explication derrière cette augmentation est que MoCoV2 a une meilleure précision dans chacune des classes. On peut utiliser la matrice de confusion ci-dessous pour voir que même si le modèle se trompe toujours pour les classes n°0, 3, 5, le modèle se trompe moins que le modèle étalon. Il est fort probable qu'avec plus d'entrainement ou de données, le modèle réussisse à limiter les erreurs entre ces 3 classes.



Matrice de confusion du modèle MoCoV2 BNF sur le dossier Full du « dataset BloodMnist ».

Également, MoCo existe en plusieurs versions (de 1 à 3), mais dans ce travail, uniquement la version 2 est testée. De plus, il est plus que certain qu'en testant la version n°1 de MoCo, il aurait eu des résultats en-dessous de MoCoV2, mais il n'est pas possible de donner une estimation pour la différence de précision. (L'article [11], ne donne malheureusement pas un comparatif entre MoCoV1 et MoCoV2 dans les mêmes conditions). De même, la version 3 est probablement meilleure.

Nous allons maintenant passer à une courte analyse des résultats, toujours pour le même « dataset » mais avec le dossier Quarter.

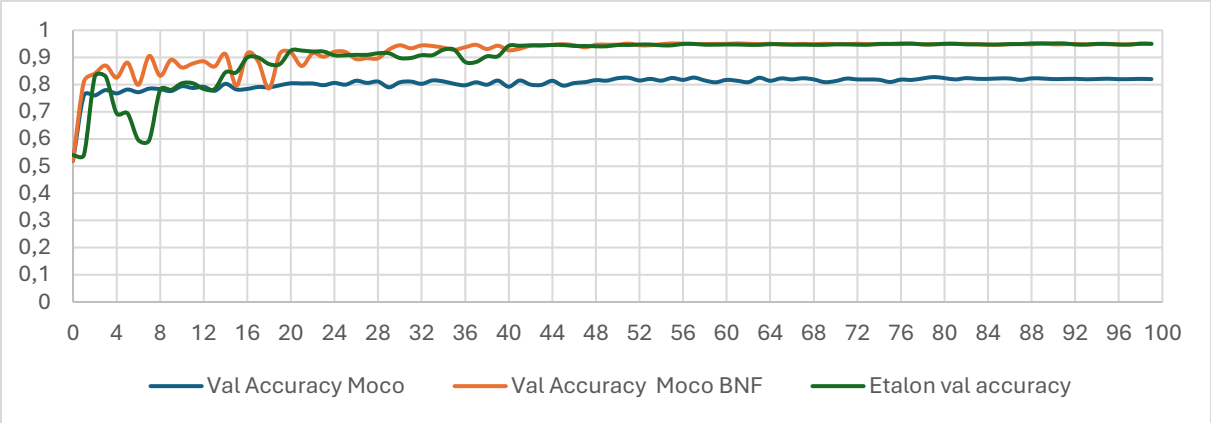


Graphique de comparaison générale entre les différents modèles MoCoV2 et le modèle étalon sur le dossier Quarter

Observations.

Les observations sont assez similaires à celles faites pour le dossier Full. Le modèle MoCoV2 ne donne pas une bonne précision, mais le modèle MoCoV2 BNF surpasse celle du modèle étalon. Cependant, on peut voir que l'écart entre les 2 modèles a diminué et que le modèle MoCo n'a pas une meilleure précision dans toutes les classes cette fois-ci.

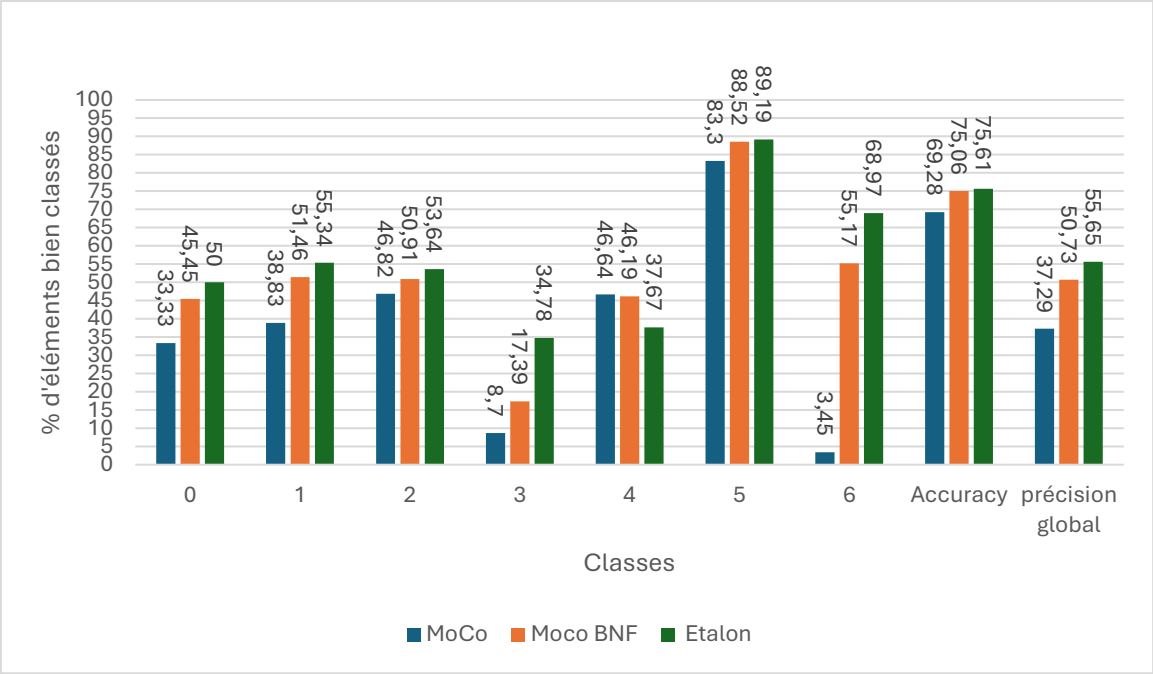
Avant de passer à l'analyse des résultats du « dataset DermaMNIST », il est intéressant d'analyser les courbes « d'accuracy » de l'ensemble de validation durant l'entrainement des divers modèles. Celle-ci nous donnera une idée de l'apprentissage des différents modèles.



Graphique comparatif entre « l'accuracy » sur l'ensemble de validation des différents modèles MoCo et celle du modèle étalon pour le dossier Full.

On peut faire plusieurs observations. Tout d'abord, comme pour SimCLR, la précision de MoCo BNF fluctue avant de se stabiliser vers « l'epoch » n°40, potentiellement pour les mêmes raisons que SimCLR. De plus, pour MoCo, on peut voir qu'il augmente de manière très lente son « accuracy » jusqu'à stagner vers « l'epoch » n°80. A ce stade, le modèle n'arrive pas à augmenter sa précision et stagne. Si on le compare au modèle étalon, on peut voir que les modèles SSL mettent plus de temps à arriver dans un état de stagnation et donc que leur apprentissage est plus long.

Maintenant, nous allons passer à l'analyse du « dataset DermaMNSIT » en commençant encore une fois par le dossier Full.

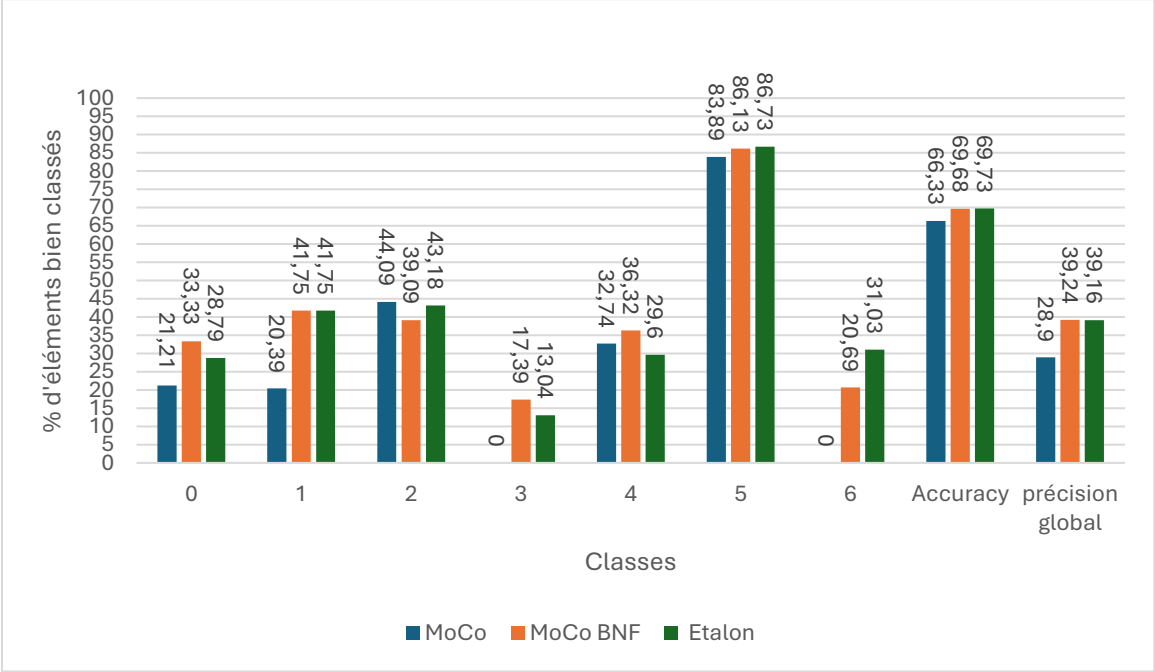


Graphique comparatif entre les modèles MoCo et le modèle étalon pour le dossier Full.

<u>Observations.</u>

On peut très vite se rendre compte que le modèle MoCo n'arrive pas à reconnaitre certaines classes, (les classes n°3 et n°6), et a une précision globale très mauvaise. D'un autre côté, le modèle MoCo BNF donne de meilleurs résultats mais encore une fois pour DermaMNIST, ceux-ci sont inférieurs aux résultats du modèle étalon. On peut noter que MoCo a une bien meilleure précision globale comparé à SimCLR (7% de différences) mais cette différence ne se matérialise malheureusement pas pour MoCo BNF, qui pour lui, est inférieure à SImCLR.

Nous pouvons maintenant terminer sur une courte analyse des résultats du dossier Quarter.



Graphique comparatif entre les modèles MoCo et le modèle étalon pour le dossier Quarter.

Observations.

On peut observer que MoCo a des résultats très similaires à SimCLR notamment pour les classes 3 et 6 (qui également, ont une précision de 0). Mais MoCo BNF arrive à dépasser très légèrement le modèle étalon. Même si celui-ci est, pour certaines classes, bien inférieur (par exemple, la classe n°6 où il y a plus de 10% de différence) .

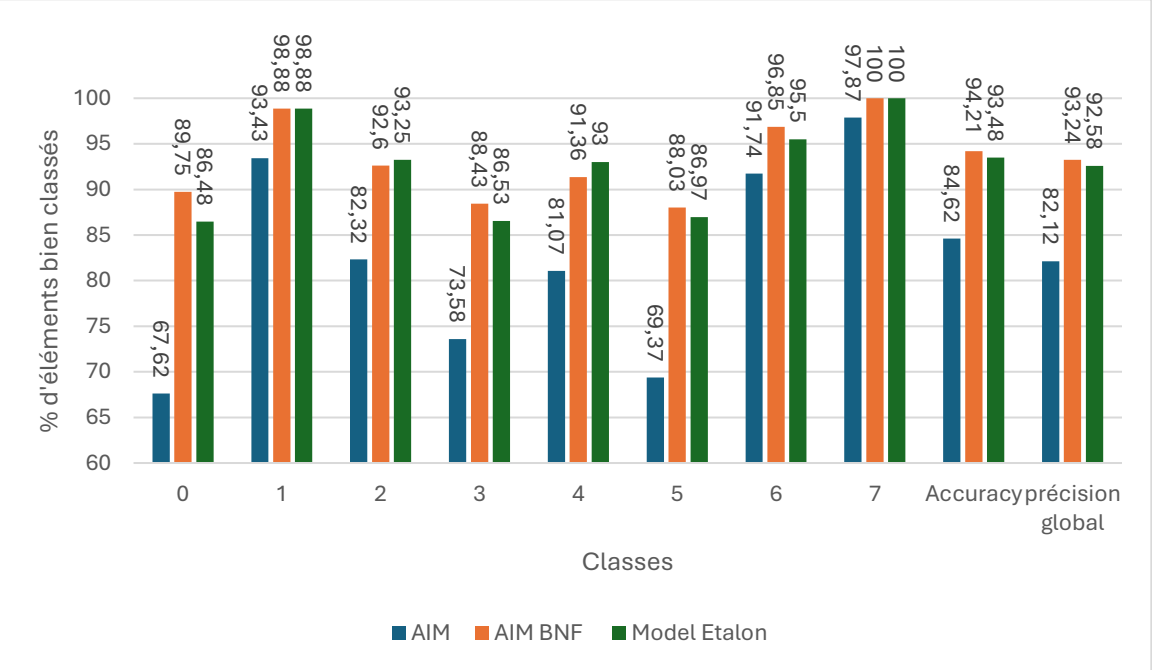
Observations générales sur MoCo.

Voici quelques observations qu'on peut faire sur MoCo.

- Si toutes les données sont labellisées et en suffisance, MoCo donnera de meilleurs résultats, supérieurs à un modèle supervisé (si on utilise « end to end fine-tuning »).
- S'il y a un déséquilibre de classes très important et qu'il y a des données labellisées en suffisance, alors MoCo donnera des résultats inférieurs à un modèle supervisé.
- S'il y a un déséquilibre de classes très important et qu'il y a n'y a pas de données labellisées en suffisance, alors MoCo donnera des résultats égaux ou légèrement supérieurs à un modèle supervisé.

7.5 Analyse des résultats du modèle AIM

Nous allons encore une fois commencer par une comparaison entre les différents modèles AIM et le modèle étalon sur le « datasets BloodMNIST », débutons avec le dossier Full.



Graphique de comparaison générale entre les différents modèles AIM et le modèle étalon sur le dossier Full.

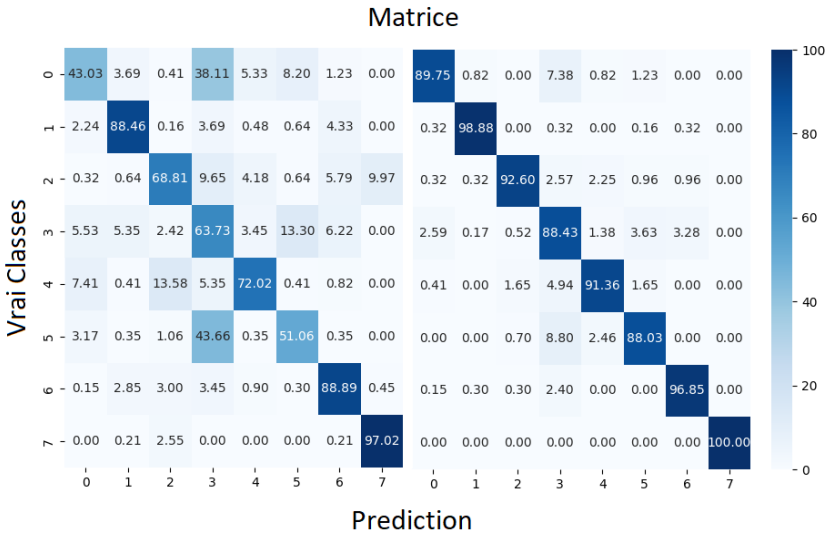
Observations.

La première observation qu'on peut effectuer est que le modèle AIM est, si on le compare aux autres modèles SSL (SimCLR, MoCo et MAE), celui avec la plus grande précision générale. Également, le modèle AIM BNF donne une très bonne précision générale, celle-ci dépasse légèrement le modèle étalon. Une dernière observation est que la différence de précision entre AIM et AIM BNF est la plus faible par rapport aux autres modèles SSL.
Nous allons maintenant terminer le « dataset BloodMNIST » en regardant les résultats du modèle sur le dossier Quarter.

Graphique de comparaison générale entre les différents modèles AIM et le modèle étalon sur le dossier Quarter .

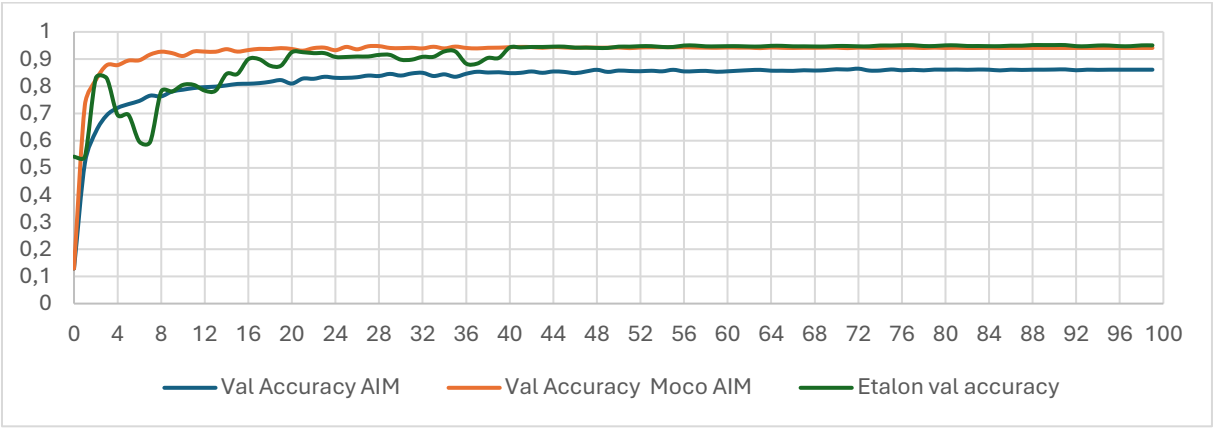
Observations.

Cette fois-ci, le modèle AIM se comporte assez différemment. Il n'a pas une bonne précision (SImCLR et MoCo sont supérieurs). En revanche, cette fois-ci, le gain entre AIM et AIM BNF est assez important et ce dernier dépasse le modèle étalon au niveau de la précision. Cela peut s'expliquer en observant le graphique ci-dessus. En effet, le modèle BNF, par son entrainement, réussit à réduire la confusion entre les classes n°0, 3 et 5 comme on peut le voir via les matrices de confusion. La précision des autres classes est également améliorée.



Comparaison des matrices de confusion des modèles AIM sur le dossier Quarter du « dataset BloodMnist ».

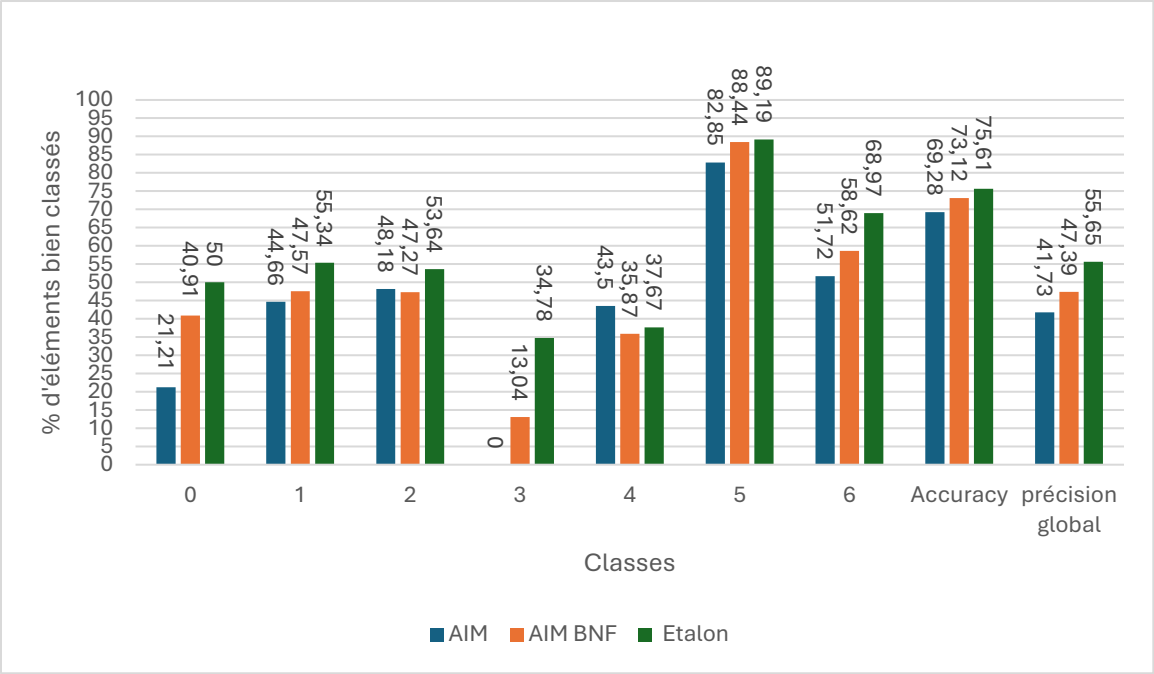
Avant de passer à l'analyse des résultats du « dataset DermaMNIST », il est intéressant d'analyser les courbes « d'accuracy » de l'ensemble de validation durant l'entrainement des divers modèles AIM. Celles-ci nous donneront une idée de l'apprentissage des différents modèles.



Graphique comparatif entre « l'accuracy » sur l'ensemble de validation des différents modèles AIM et celle du modèle étalon pour le dossier Full.

Comparée aux autres modèles, la précision des 2 modèles AIM est assez basse au départ, puis très vite, monte pour les « epochs » suivants. Ensuite, elle va graduellement augmenter de manière assez lente jusqu'à arriver, vers le quarantième « epoch », à un plateau. On peut également observer que les modèles ont peu d'oscillation durant leur entrainement.

Nous allons maintenant passer à l'analyse du « dataset DermaMNIST » et nous allons commencer sur les résultats du dossier Full.

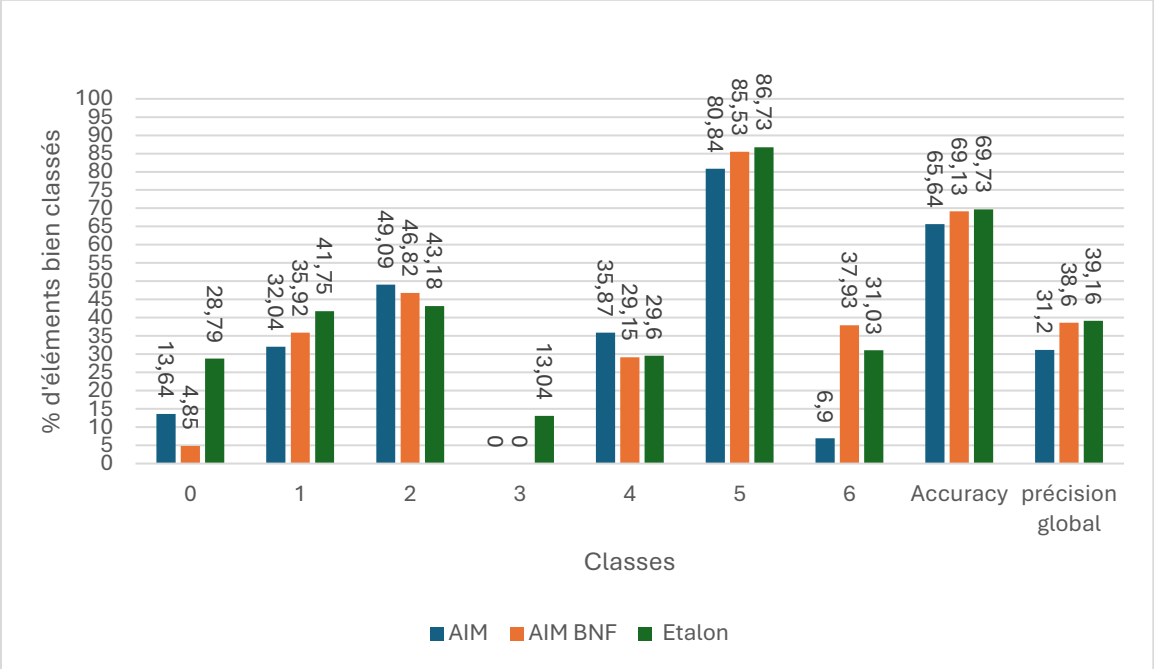


Graphique comparatif entre AIM et le modèle étalon pour le dossier Full.

Observations.

Nous pouvons observer une nouvelle fois que le modèle AIM est le plus précis parmi les autres modèles SSL équivalents et que le modèle AIM BNF est le moins précis. On peut l'expliquer assez facilement car ce dernier est également moins précis dans toutes les classes par rapport au modèle étalon. De plus, on observe encore une fois, un manque de gain de précision entre AIM et AIM BNF.

Nous allons maintenant terminer par le dossier Quarter



Graphique comparatif entre les modèles AIM et le modèle étalon pour le dossier Quarter.

Observations.

Les résultats obtenus pour le dossier Quarter sont assez intéressants. En effet, la précision de AIM BNF, même si celle-ci n'arrive pas à celle du modèle étalon, n'est pas la plus mauvaise des modèles SSL. Ce qui n'est pas la même chose par rapport à ce que nous avons observé pour le dossier Full. Mais comme pour le dossier Full, on peut voir que le gain entre les 2 modèles AIM est assez faible encore une fois. De plus, aucun d'eux n'arrive à classer correctement la classe n°3.

Observations générales sur AIM.

Voici quelques observations concernant AIM.

- En utilisant « le end to end finetuning », AIM peut atteindre voire dépasser le modèle étalon à condition que le déséquilibre de classes ne soit pas très important.
- N'a pas un gain de précision très important avec le « end to end fine-tuning ».
- A besoin de beaucoup de paramètres pour donner de bons résultats.
- Est lourd et long à entrainer.

7.6 Analyse des résultats du modèle MAE

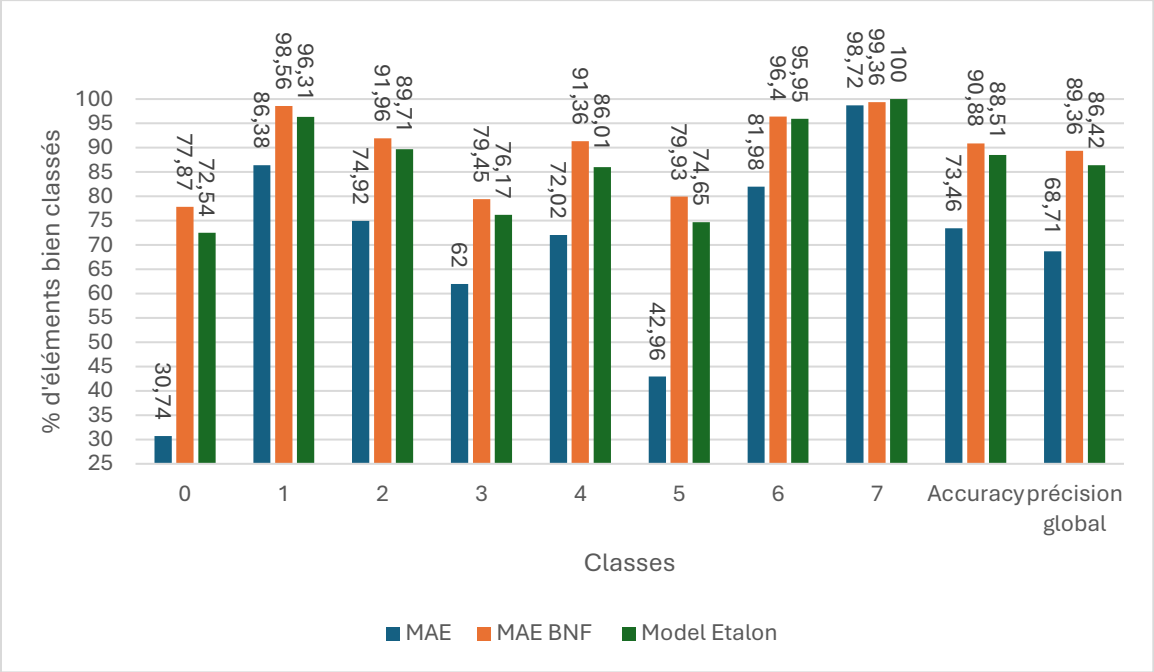
Pour terminer ces analyses, nous allons maintenant passer aux analyses du modèle MAE. Comme pour les autres modèles SSL, nous allons commencer par le « dataset BloodMNIST » ainsi que le dossier Full

Graphique de comparaison générale entre les différents modèles MAE et le modèle étalon sur le dossier Full.

Observations.

Une première observation qu'on peut faire est que la différence de précision entre certaines classes du modèle MAE et celles de MAE BNF est assez importante, notamment pour les classes 0, 3 et 5. De plus, le modèle MAE BNF a une précision globale supérieure au modèle étalon et chaque classe a une précision égale ou supérieure à celui-ci.

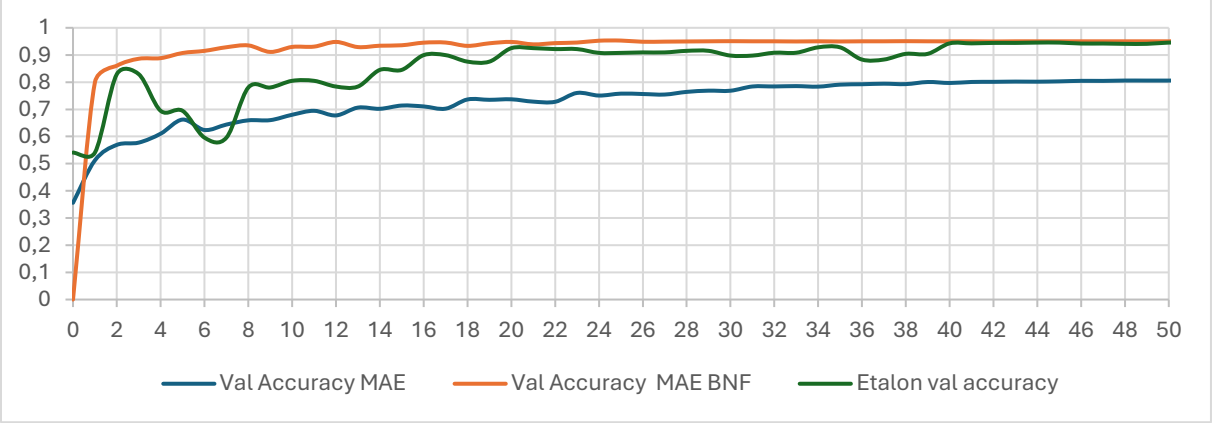
Cependant, le modèle AME est le moins performant de tous les modèles SSL en termes de précision globale. Encore une fois, comme pour les autres modèles, ce manque de précision vient des classes 0, 3 et 5 qui sont confondues entre elles.
Nous allons maintenant terminer le « dataset BloodMNIST » en regardant les résultats du modèle sur le dossier Quarter.



Graphique de comparaison générale entre les différents modèles MAE et le modèle étalon sur le dossier Quarter.

Observations.
Les observations sont assez similaires à celles faites pour le dossier Full. On se retrouve avec presque les mêmes remarques : le modèle MAE BNF est meilleur que le modèle étalon et le modèle MAE a la pire précision globale par rapport aux autres modèles. Le gain de précision entre les 2 modèles MAE est très impressionnant.

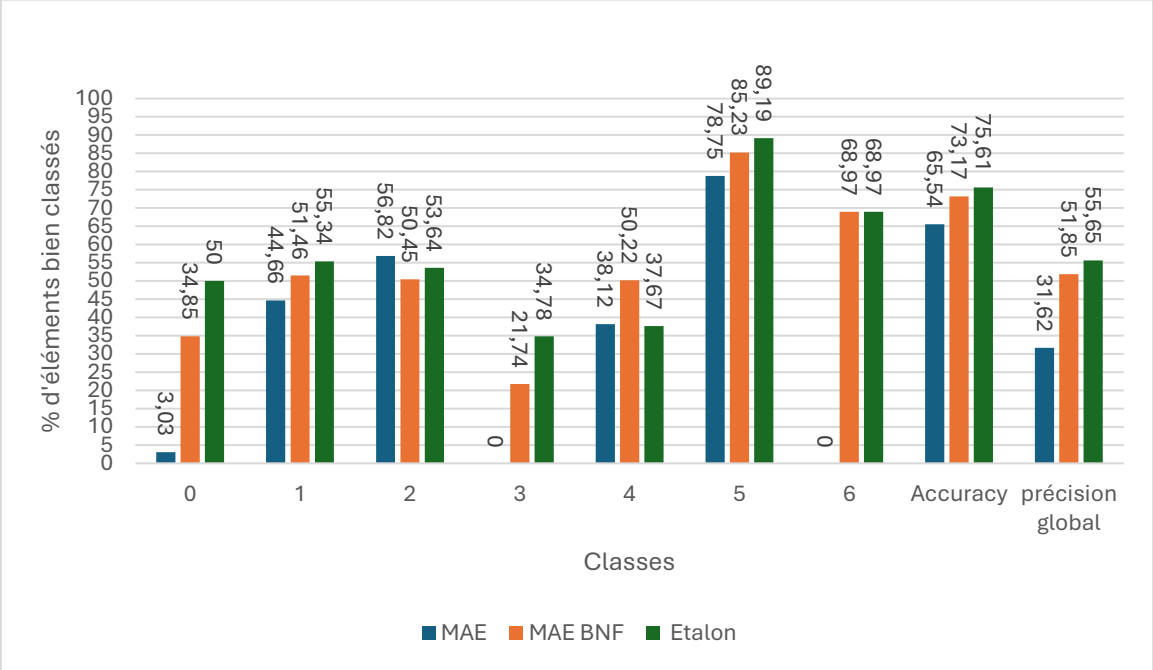
Avant de passer à l'analyse des résultats du « dataset DermaMNIST », il est intéressant d'analyser les courbes « d'accuracy » de l'ensemble de validation durant l'entrainement des divers modèles. Celles-ci nous donneront une idée de l'apprentissage des différents modèles.



Graphique comparatif entre « l'accuracy » sur l'ensemble de validation des différents modèles MAE et celle du modèle étalon pour le dossier Full.

On peut observer que le modèle MAE a une montée en précision assez lente jusqu'à un point de stagnation vers 40 « epochs ». Pour ce qui est de MAE BNF, après une très grosse amélioration de précision, celui-ci va fluctuer légèrement jusque 30 « epochs » puis stagner jusqu'à la fin de l'entrainement.

Pour terminer, nous allons passer à l'analyse du « dataset DermaMNSIT » en commençant encore une fois par le dossier Full.

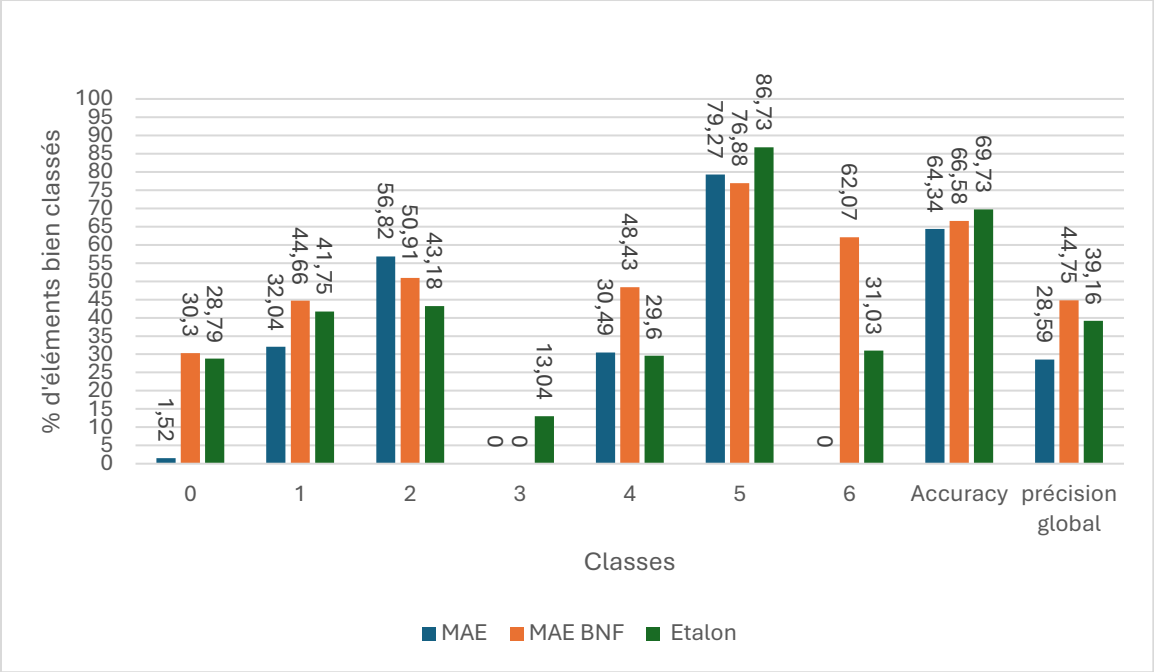


Graphique comparatif entre les modèles MAE et le modèle étalon pour le dossier Full.

Observations.

Le modèle MAE est encore une fois très peu performant et le deuxième, le moins bon de tous les modèles SSL (SimCLR a une précision légèrement inférieure à MAE), le modèle n'arrive pas à reconnaitre les classes 0, 3 et 6. Cependant, le modèle MAE BNF donne la meilleure précision possible des modèles SSL, même si celle-ci n'arrive pas à celle du modèle étalon. Une nouvelle fois, on observe que le « fine-tuning » joue un rôle important dans la précision du modèle pour MAE.

Nous pouvons maintenant terminer sur une courte analyse des résultats du dossier Quarter :



Graphique comparatif entre MAE et le modèle étalon pour le dossier Quarter.

Observations.

On peut faire des observations assez intéressantes.

D'abord le modèle MAE BNF a une très grande précision dépassant de loin celle du modèle étalon. Cependant, le modèle n'arrive pas à classer correctement la classe n°3 (qui est la plus compliquée à classer si on suit la précision du modèle étalon). On est également dans le cas où MAE est médiocre mais MAE BNF est le meilleur en termes de précision.

Observations générales sur MAE.

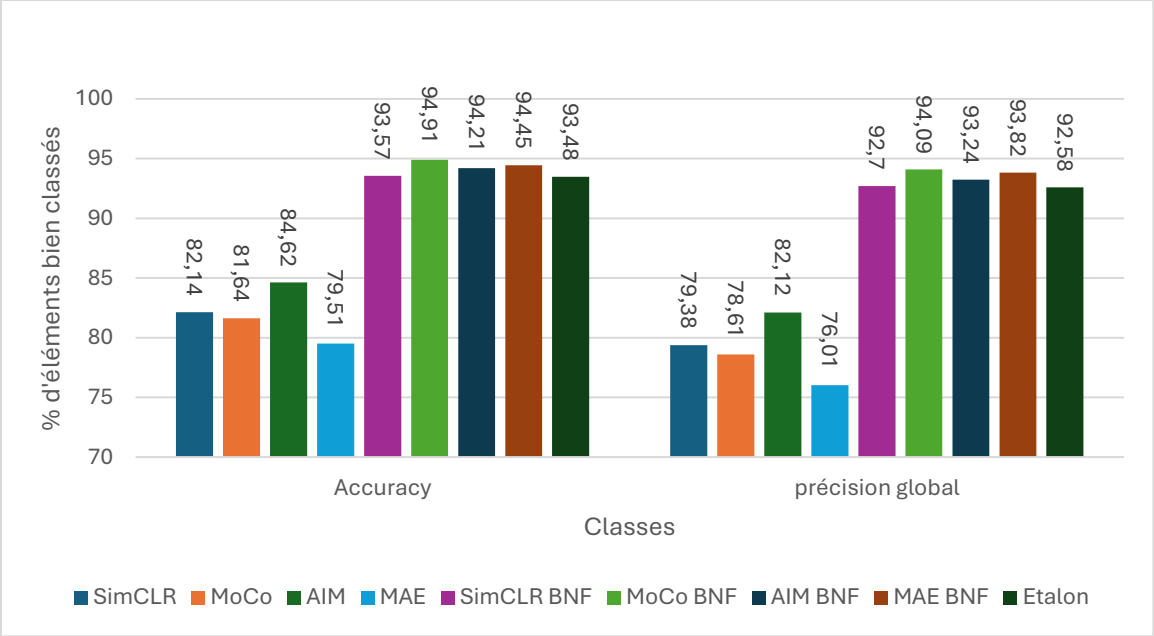
Voici quelques observations faites sur MAE.

- Pour le fine-tuning, l'utilisation de « extract feature for classifier » est à éviter et donnera toujours des résultats assez mauvais mais « end to end fine-tuning » donnera presque toujours d'excellents résultats.
- L'utilisation de « end to end fine-tuning » permet au modèle d'apprendre énormément.
- Le modèle est très lourd et lent à entrainer et fonctionne uniquement pour certaines tailles d'images.
- Il est moins affecté que d'autres modèles par le déséquilibre de classes.

7.7 Analyse globale des résultats

Cette partie est consacrée à analyser de manière globale la performance des différents modèles pour déterminer quel est le meilleur choix (s'il y en a un), mais également, si avec les données disponibles, on peut tirer des conclusions sur le meilleur cas d'utilisation des modèles.

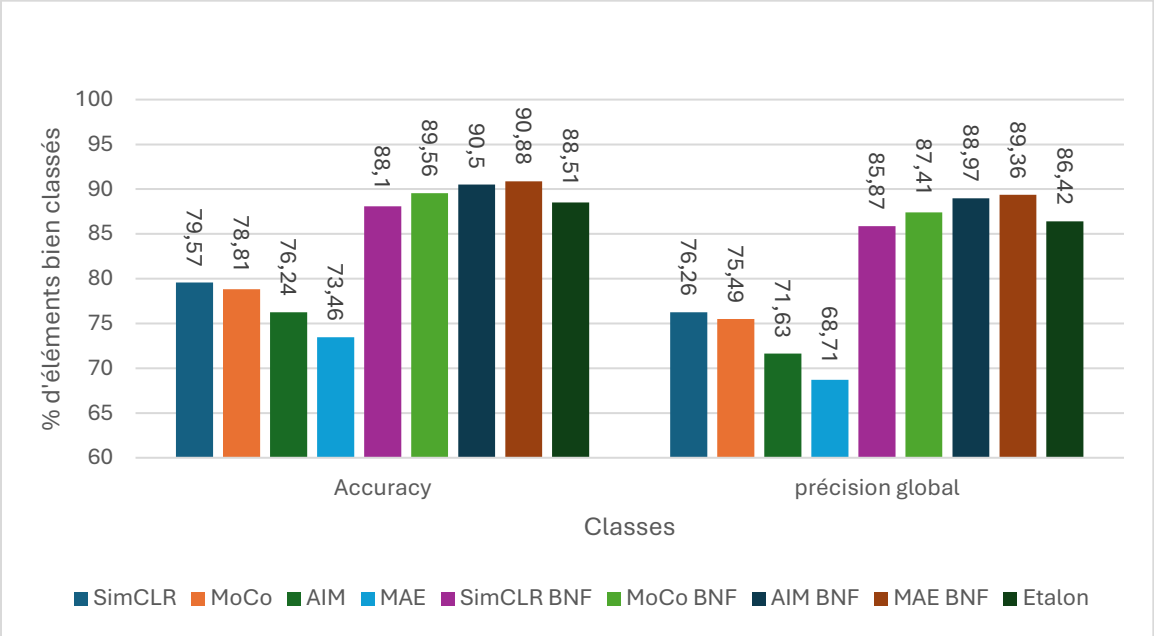
Nous allons évidemment regarder les résultats du « dataset BloodMNIST » avant de passer à DermaMNIST. Nous allons analyser les modèles en fonction des dossiers en commençant par le dossier Full.



Graphique de comparaison des modèles SSL pour BloodMNIST et le dossier Full

On peut tirer plusieurs conclusions de ce graphique. Les modèles SSL non NBF (SimCLR,MoCo,AIM,MAE) n'arrivent pas à dépasser un certain palier de précision (AIM est une exception). Il semble que la méthode « extract feature from classifier » ne soit pas adaptée pour avoir une bonne précision. Cependant, la méthode « end to end fine tuning » (les modèles BNF) permet d'obtenir une précision similaire, voir supérieure au modèle étalon. Le modèle le plus précis étant MoCo. Le modèle MoCo V2 serait donc à privilégier, il ne faut cependant pas oublier que le gain de précision est assez faible et que ce dernier prend beaucoup plus de temps et de ressources pour être entrainé.

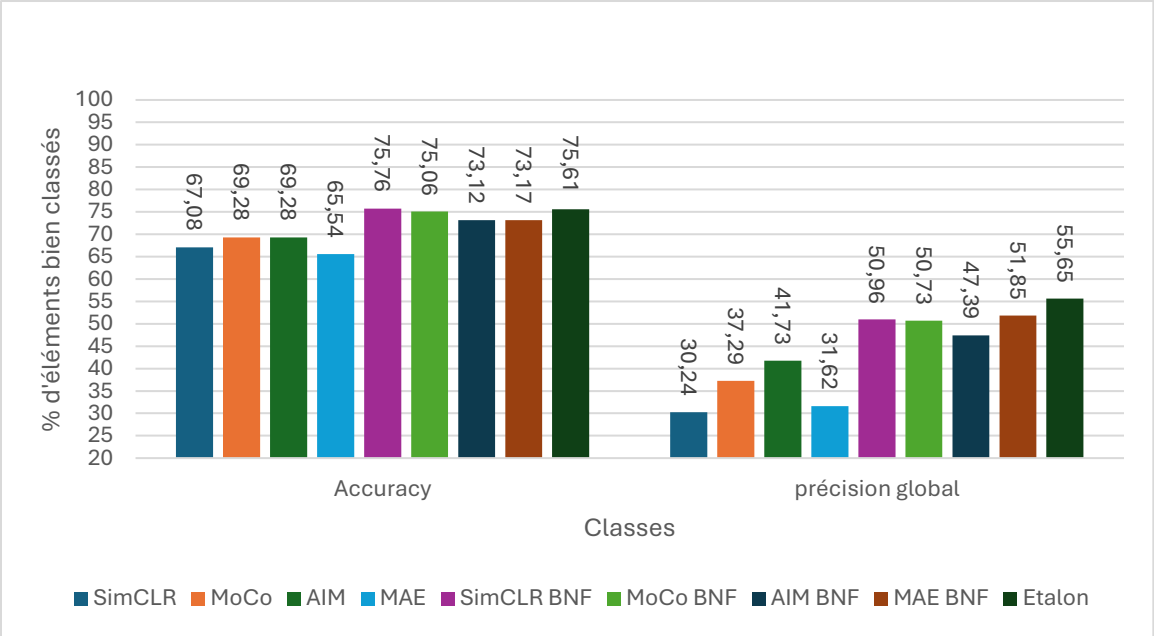
Ensuite, nous allons passer à l'analyse des résultats mais pour le dossier Quarter, toujours sur le « dataset BloodMNIST ».



Graphique de comparaison des modèles SSL pour BloodMNIST et le dossier Quarter

L'idée derrière le dossier Quarter était d'avoir un cas qui, en théorie, serait plus favorable aux divers modèles SSL. Mais quand on analyse les données, on se rend compte que même si les modèles SSL sont meilleurs, la différence n'est pas non plus très importante. La différence entre le modèle étalon et le meilleur modèle (MAE) n'est que de 2,94%. Une amélioration certes mais pas une amélioration importante. Encore une fois, seuls les modèles BNF donnent une bonne précision. Dans un cas où le nombre de données est limité et qu'on a un « dataset » avec un déséquilibre de classes moyen, alors un modèle SSL sera meilleur que du superviser classique.

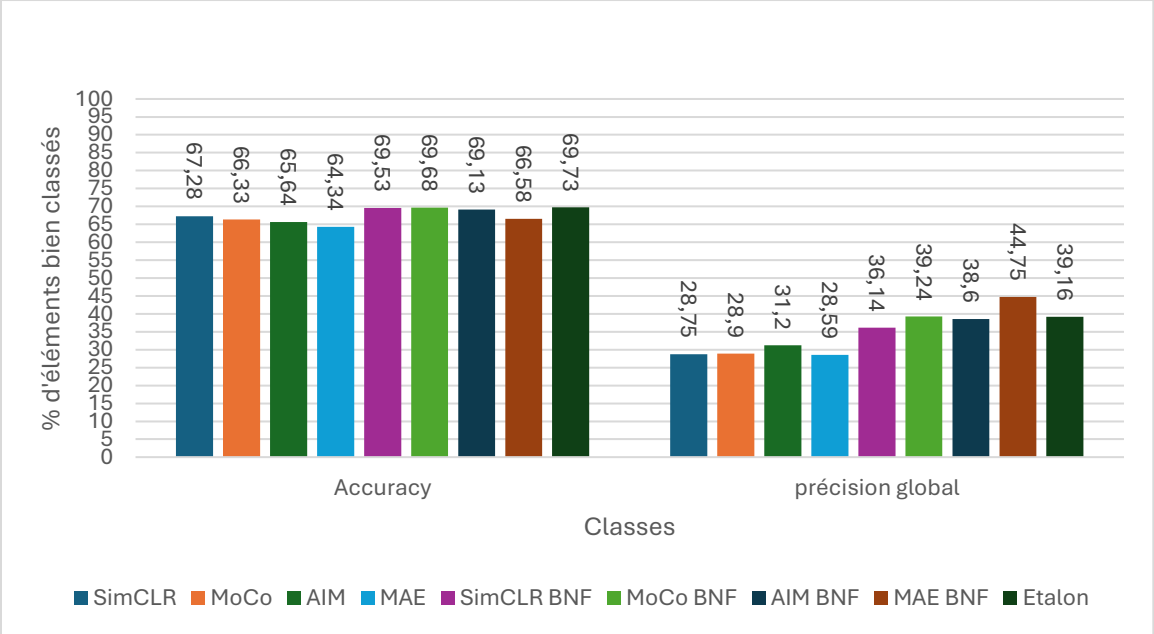
Nous allons maintenant passer au second « dataset » utilisé lors de ce travail : DermaMNIST. Celui-ci est assez différent de BlooDMNIST en plusieurs points: moins de données, déséquilibre de classes assez important, autre méthode utilisée pour récolter les données.



Graphique de comparaison des modèles SSL pour DermaMNIST et le dossier Full

On peut voir que bien que les modèles SSL BNF soient proches de la précision du modèle étalon aucun n'arrive à l'atteindre, de plus les modèles non BNF sont très en-dessous des précisions du modèle étalon. On peut en conclure, avec les diverses analyses effectuées dans ce travail, que le problème ici est le déséquilibre de classes durant l'entrainement des modèles SSL et que malgré l'entrainement d'un classificateur, celui-ci n'arrive pas à donner au modèle une bonne précision. On peut aussi voir le grand intérêt de comparer les différents modèles en utilisant la précision globale et non « l'accuracy » qui est plus sensible au déséquilibre de classes.

Pour terminer cette partie nous allons passer à l'analyse des résultats sur le dossier Quarter, après ça, nous passerons à une conclusion générale sur le travail.



Graphique de comparaison des modèles SSL pour DermaMNIST et le dossier Quarter

Dans ce cas-ci, on se retrouve avec moins de données et les modèles SSL peuvent faire mieux que le modèle étalon notamment MoCoBNF et MAE BNF, ce dernier étant meilleur que le modèle étalon. Cependant, il ne faut pas oublier que celui-ci n'arrive pas à reconnaitre une classe, le rendant moins intéressant. Dans un cas similaire, les modèles SSL semblent assez intéressants et il est probable que, s'il y avait moins de déséquilibres de classes, ceux-ci auraient de meilleures performances.

7.8 Quel fine tuning choisir ?

On peut maintenant répondre à cette question avec les données disponibles. Il est assez clair que la méthode « end to end fine tuning » est de loin la plus performante donnant à chaque fois de meilleurs résultats que la méthode « extract feature for classifier » et dépassant assez souvent le modèle étalon. Cependant, il ne faut pas perdre de vue que même la meilleure méthode demande aussi beaucoup plus de ressources pour être entrainée et que bien qu'assez inférieure, « extract feature for classifier » donne quand même des résultats assez impressionnants, surtout que le « classifier » n'a que 1.6 millions de paramètres entrainés avec des données labellisées, comparés aux 11.6 millions du modèle supervisé.

8 Analyse des résultats des fusions modèles

Avant de conclure ce travail, on peut encore effectuer quelques analyses sur les résultats des modèles, non pas sur les modèles de manière séparée, mais sur des modèles fusionnés. Quand on parle de modèles fusionnés, on entend pouvoir utiliser plusieurs modèles en même temps pour pouvoir donner un résultat à partir d'une entrée. L'idée de combiner des modèles entre eux et de pouvoir arriver à un meilleur résultat. Si on compare 2 modèles, on peut se rendre compte que bien que l'un d'eux soit moins précis, il a sûrement certaines classes pour lesquelles il a une meilleure précision. On essaie donc de prendre le plus performant des 2 modèles.

Il y a plusieurs manières de s'y prendre. On peut, par exemple, faire du modèle « stacking » [46] où l'idée est de prendre la sortie de plusieurs modèles qui va être utilisée comme entrée d'un autre modèle (un meta modèle) et qui donnera une prédiction. Pour ce travail, nous allons utiliser une forme de Bootstrap agrégation. Cela consiste à utiliser les prédictions faites par plusieurs modèles et de les combiner pour obtenir une prédiction finale. Cette combinaison peut être de différentes sortes. On peut faire par vote majoritaire (la sortie est celle majoritaire parmi tous les modèles) ou on peut combiner les différentes probabilités, cette dernière option a été choisie. La méthode utilisée est différente du Bootstrap[46] agrégation car il n'y a pas de modification sur les « datasets » étant donné qu'on utilise les modèles tels qu'ils ont été entrainés au point 7.

Maintenant, il faut déterminer 2 éléments essentiels. D'abord, comment combiner les modèles et quels sont les modèles à combiner. Pour le premier point, nous allons multiplier entre elles les probabilités données par chaque modèle. Concernant le second point, nous allons tester 3 fusions de modèles, MoCo et SimCLR car ils sont assez similaires dans leur construction et reposent tous les 2 sur un Resnet. Un modèle MAE et AIM car ils sont également très similaires et reposent tous les 2 sur un ViT. Nous terminerons cette partie avec un modèle qui fusionne tous les modèles SSL en un. Les modèles fusionnés seront appelés « modèles fusions ».

Les tests seront réalisés sur les 2 « datasets » mais pour que cette section reste relativement brève, uniquement sur les modèles BNF.

Pour éviter les répétitions, les sections d'analyse se concentreront surtout sur la comparaison des modèles fusions et des modèles étalons pour les différents « datasets » et dossiers.

8.1 Analyse Fusion SimCLR MoCo (FSM)

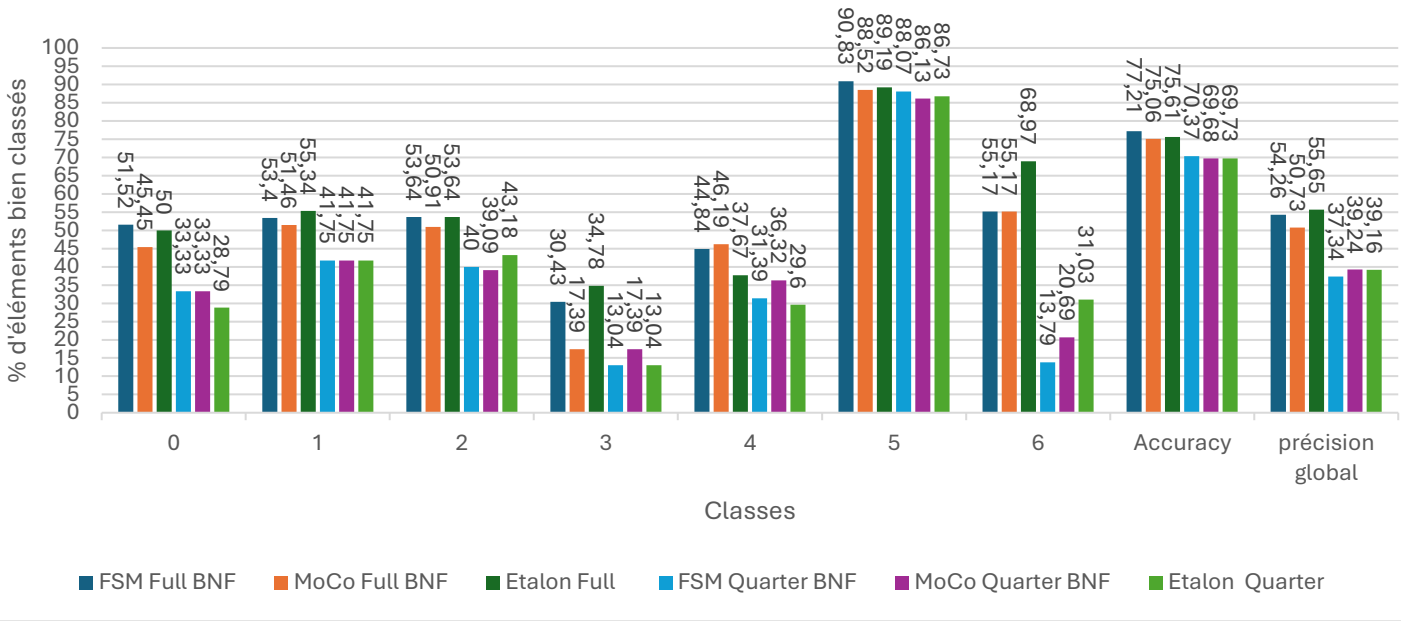
Nous allons commencer par un modèle qui fusionne SimCLR et MoCo, il sera appelé FSM.

Nous allons commencer par un test sur le « dataset BloodMNIST » et ensuite sur le « dataset DermaMNIST ».

Graphique de comparaison générale entre les modèles FSM, MoCo et les modèles étalons pour le « dataset BloodMNIST ».

Nous pouvons effectuer les observations suivantes. D'abord, les modèles FSM dépassent tous leurs modèles étalons équivalents d'au moins 1%. Cette amélioration provient des classes 0, 3 et pour le modèle FSM Full BNF également de la classe n°5. Cependant, le modèle FSM Full BNF a une précision très légèrement inférieure à celle du modèle MoCo (de 0.19%). Cela peut être expliqué car le modèle SimCLR est beaucoup moins performant que le modèle MoCo et que celui-ci fait descendre la précision du modèle FSM. En effet, le modèle SimCLR se trompe davantage que le modèle MoCo et il est fort probable qu'une bonne prédiction donnée par MoCo devienne mauvaise car elle est multipliée avec une mauvaise prédiction de SImCLR. Cela peut se produire si la prédiction de MoCo est bonne mais avec une certitude faible et que celle de SimCLR soit mauvaise mais que le modèle ait une très grande certitude dans son choix (même si celui-ci est incorrect). Pour finir, nous pouvons affirmer que les modèles FSM sont assez corrects comparés aux modèles étalons mais que parfois MoCo reste un meilleur choix.

Nous allons maintenant passer à l'analyse pour le « dataset DermaMNIST » pour les dossiers Full et Quater.



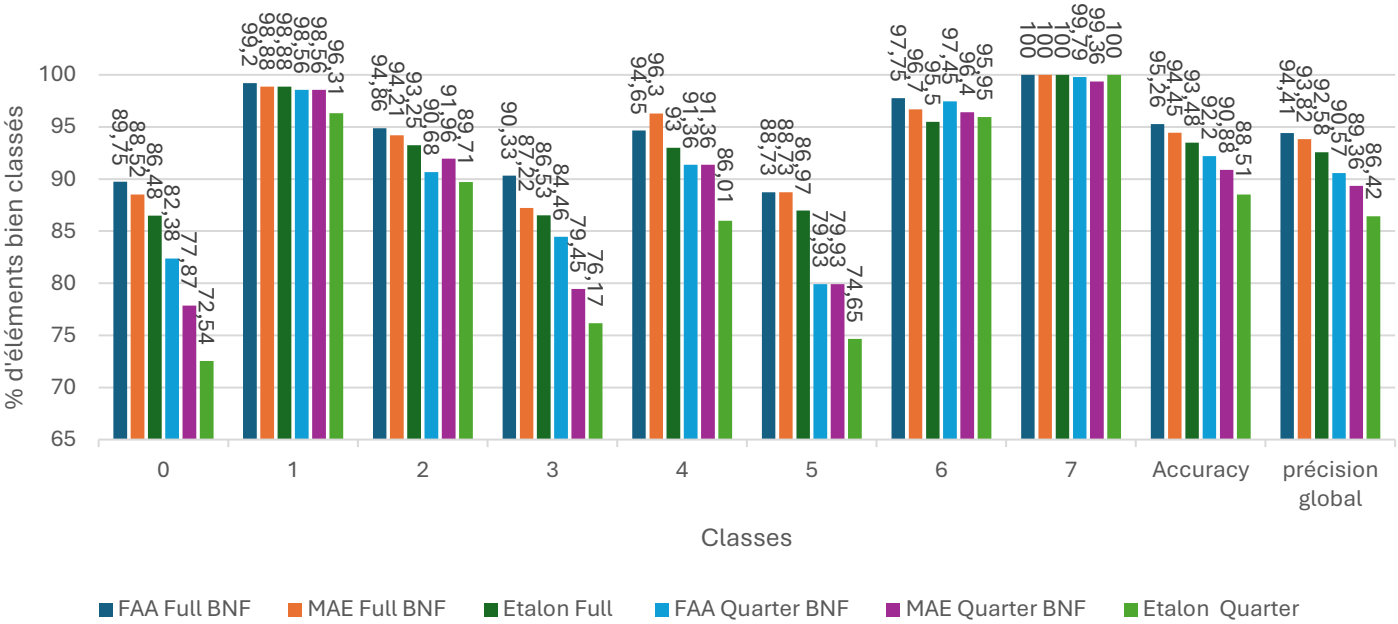
Graphique de comparaison générale entre les modèles FSM, MoCo et les modèles étalons pour le « dataset DermaMNIST ».

Nous pouvons effectuer les observations suivantes. En premier lieu, on se retrouve avec une conclusion opposée au « dataset BloodMNIST » étant donné que les modèles FSM ne font pas mieux que les modèles étalons. Cependant, le modèle FSM Full BNF fait beaucoup mieux que les autres modèles SSL mais le modèle FSM Quarter BNF est considéré comme étant le deuxième moins performant. Cela provient certainement du modèle SIMCLR qui est le plus médiocre. Encore une fois, la mauvaise précision des modèles est sûrement à attribuer au déséquilibre de classes important.

Le modèle FSM est, dans sa globalité, un bon modèle qui peut dans certains cas donner de très bons résultats. En revanche, il souffre de performances inférieures à MoCo dû à la relativement faible précision de SimCLR.

8.2 Analyse Fusion AIM MAE(FAA)

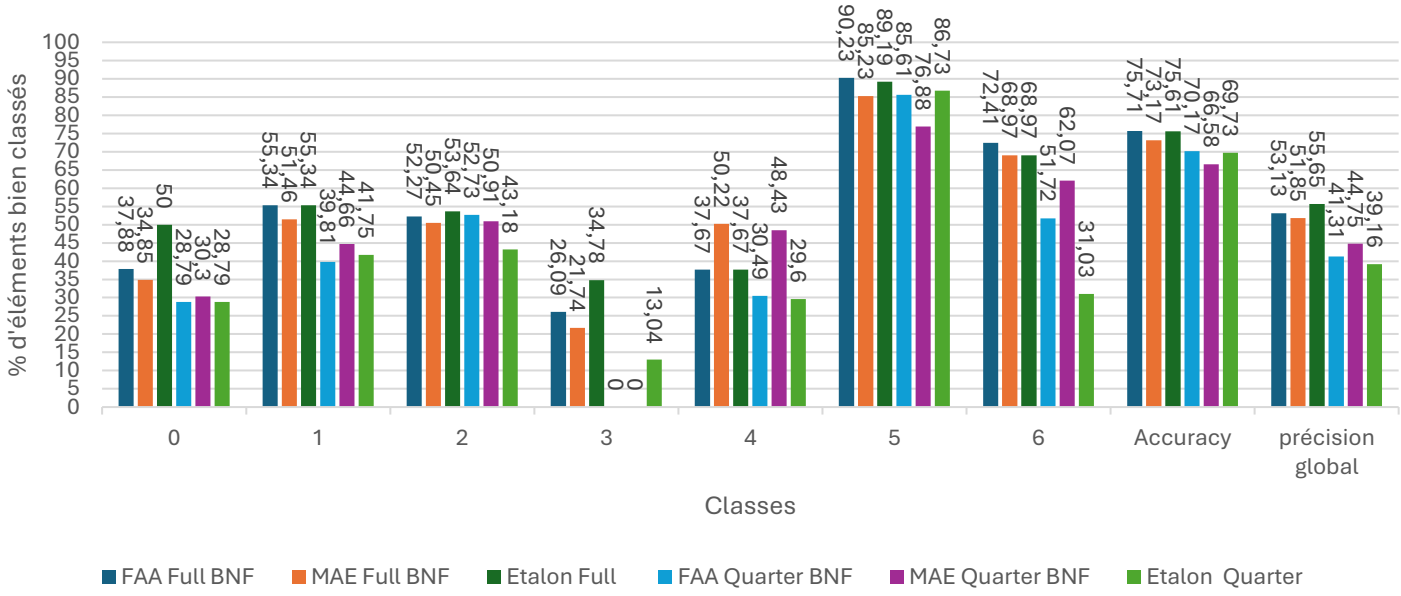
Nous allons continuer par un deuxième modèle qui fusionne cette fois-ci AIM et MAE. Il sera appelé FAA. Nous allons commencer par un test sur le « dataset BloodMNIST » et ensuite sur le « dataset DermaMNIST ».



Graphique de comparaison générale entre les modèles FAA, MAE et les modèles étalons pour le « dataset BloodMNIST ».

Le modèle FAA donne les meilleurs résultats tous modèles SSL confondus et pour tous dossiers confondus sur le « dataset BloodMNIST ». Cela peut s'expliquer par le fait que les modèles AIM et MAE donnent, en général, d'excellents résultats (quasiment les meilleurs de tous les modèles). On peut également voir que les modèles fusions ont une meilleure précision dans toutes les classes avec parfois des différences importantes (par exemple entre le FAA Quarter BNF et le modèle étalon Quarter pour la classe n°3, on a une différence de 89,29%, ce qui est tout simplement énorme). Cette excellente précision est probablement due au fait que les 2 modèles utilisés sont tous les 2 assez précis pour ce « dataset ».

Nous allons maintenant passer à l'analyse pour le « dataset DermaMNIST », pour les dossiers Full et Quarter.



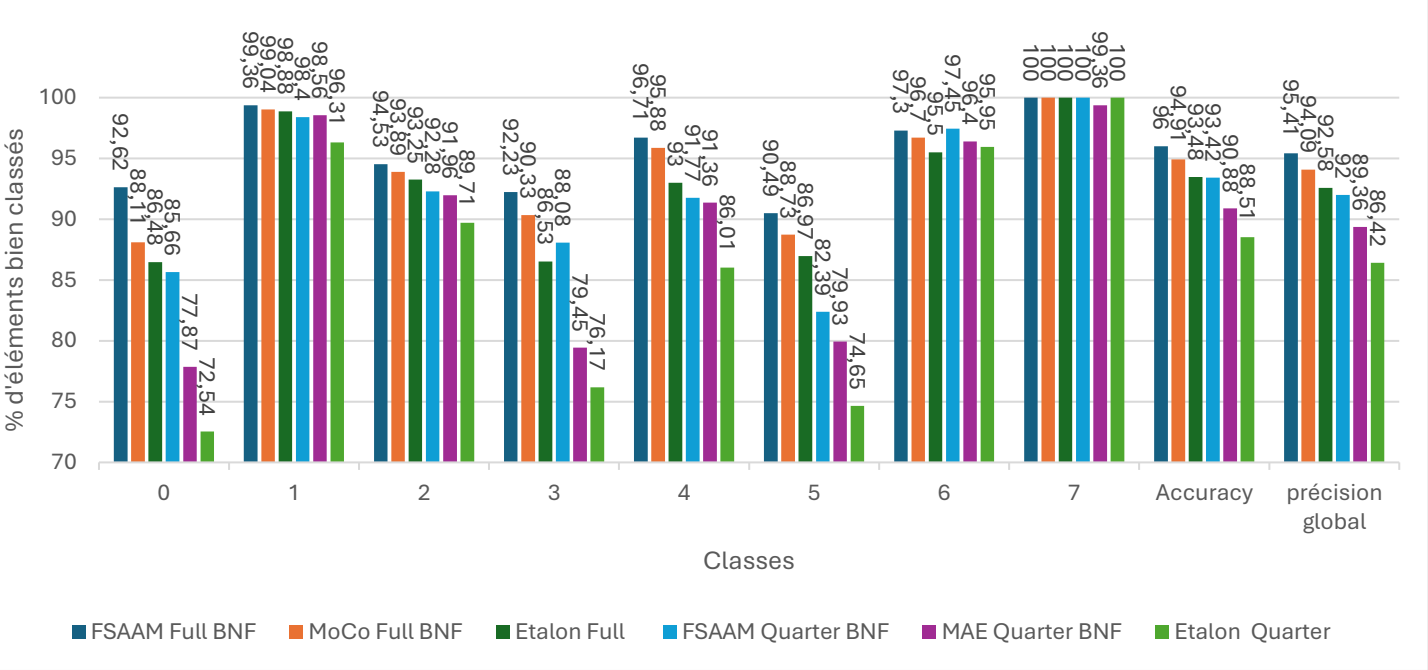
Graphique de comparaison générale entre les modèles FAA,MAE et les modèles étalons pour le « dataset » DermaMNIST.

Nous pouvons faire certaines observations. La première est que les modèles fusions ne donnent pas nécessairement de meilleurs résultats que les modèles étalons (pour le dossier Quarter, le modèle fusion est au-dessus du modèle étalon mais pour le dossier Full c'est l'inverse). Également, la précision des modèles est loin d'être la meilleure comparée aux autres modèles SSL. On a un comportement identique à celui de FSM. En effet, ici AIM est beaucoup moins bon que MAE et il tire la précision du modèle par le bas.

Le modèle FAA est très bon, même s'il rencontre quelques problèmes avec certains « datasets ». Il faut cependant garder à l'esprit que ces bons résultats sont en partie dû à MAE qui est un modèle bien plus complexe que les autres modèles SSL, lui donnant ainsi un certain avantage. Cependant, le modèle MAE offre tout de même une très bonne précision alors qu'il est similaire aux autres modèles SSL.

8.3 Analyse Fusion SimCLR AIM MAE MoCo (FSAAM)

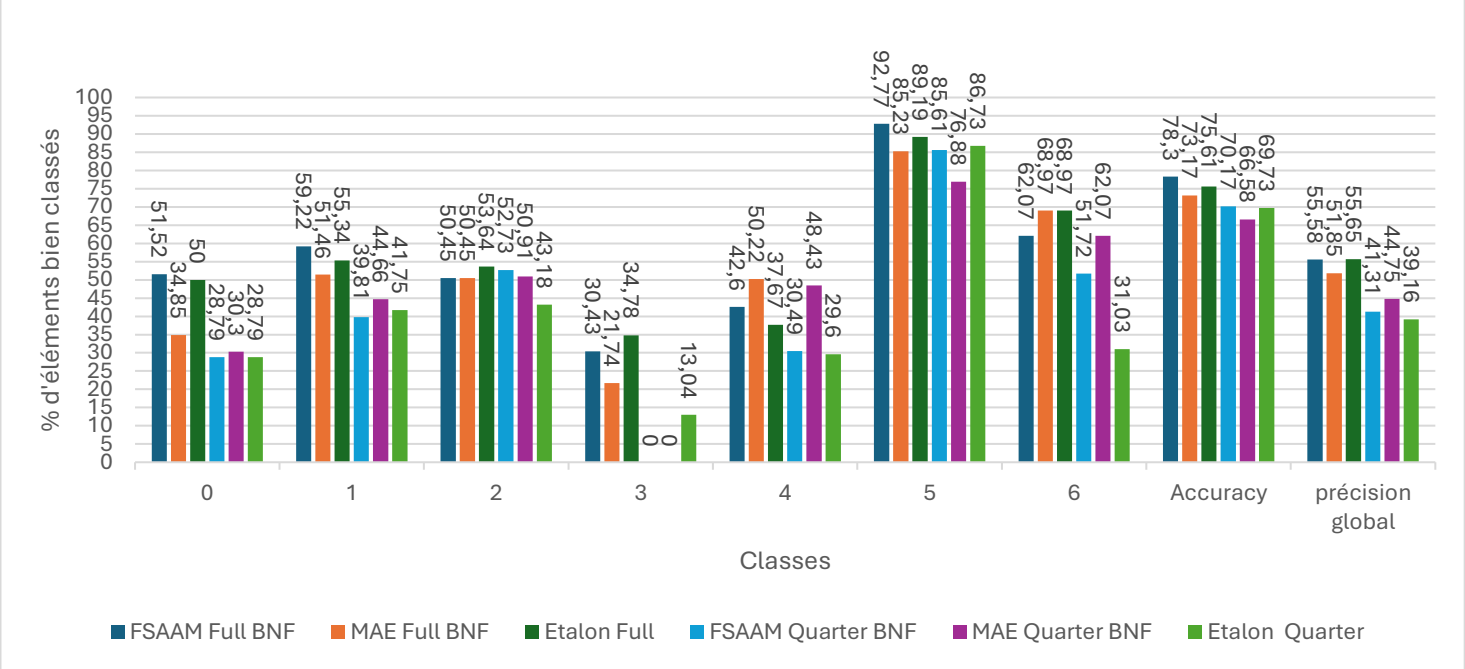
Nous allons terminer cette section sur un modèle qui, cette fois-ci, va regrouper tous les modèles que nous avons utilisés durant ce travail. Il sera nommé FSAAM. Comme pour les 2 précédentes analyses, nous allons commencer par analyser les résultats sur le « dataset BloodMNIST » pour finir sur le « dataset DermaMNIST ».



Graphique de comparaison générale entre les modèles FSM, de MoCo, de MAE et les modèles étalons pour le « dataset BloodMNIST ».

Nous pouvons effectuer les observations suivantes. D'abord, les modèles FSAAM dépassent tous leurs modèles étalons équivalents d'au moins 2.5%. De plus, le modèle FSAAM Quarter atteint presque le modèle étalon qui a été entrainé sur le dossier Full. Les modèles FSAAM donnent les meilleures précisions qui ont été observées dans ce travail pour leur dossier respectif (95,41% et 92%). Le plus impressionnant reste le modèle entrainé sur le dossier Quater car il a 5,58% de précision de plus ; la plus grande différence observée entre un modèle SSL et un modèle étalon. Ce modèle fusion met en valeur l'efficacité de « fusionner » les modèles SSL. Il est possible que l'utilisation de SImCLR ait potentiellement baissé la qualité du modèle. Cependant, on peut quand même observer que les classes les moins bonnes sont toujours les classes 0, 3 et 5 indiquant que le problème présent dans le modèle étalon n'est pas à 100% résolu.

Nous allons maintenant conclure cette partie en analysant le « dataset DermaMNIST » pour les dossiers Full et Quarter.



Graphique de comparaison générale entre les modèles FSM, de MAE et les modèles étalons pour le « dataset DermaMNIST ».

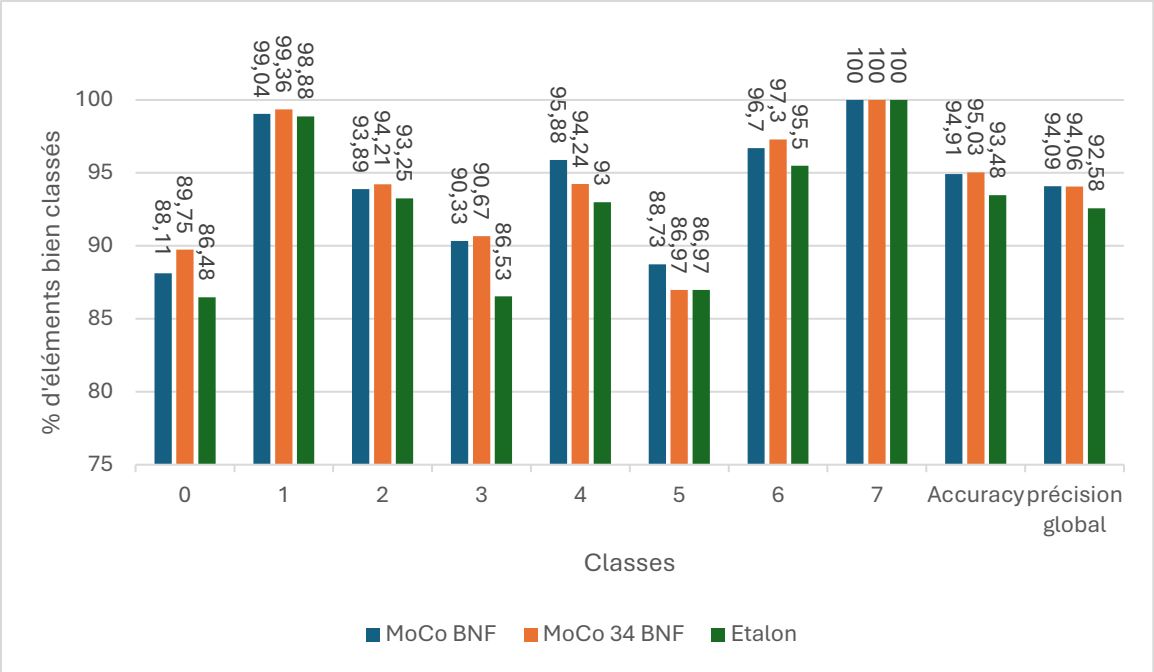
Nous pouvons faire quelques observations. La première est que le modèle FSAAM Full, bien que très proche du modèle étalon, ne le dépasse pas, il est bon de rappeler qu'aucun modèle SSL ne dépasse le modèle étalon dans ce cas de figure. Cependant, l'autre modèle FSAAM dépasse le modèle étalon mais n'est pas le meilleur modèle comparé aux autres modèles SSL (MAE fait mieux).

Pour conclure sur les modèles fusions, disons que ceux-ci sont une bonne manière d'améliorer les résultats des modèles SSL si plusieurs modèles sont disponibles. Il est tout à fait possible qu'avec plus de tests ou des modèles différents, on puisse arriver à de meilleurs résultats. Une idée potentielle qui n'a pas été testée est de créer un modèle qui ne prendrait que les meilleures parties de chaque modèle (c'est-à-dire qu'on ne prend la réponse du modèle que si celui-ci a la plus grande précision ou confiance dans sa réponse). Aussi, un méta modèle est envisageable mais est hors cadre de ce travail. Un autre avantage des modèles fusions est que la forme et la taille du modèle ne sont pas un frein pour pouvoir les combiner.

9 Augmentation du modèle

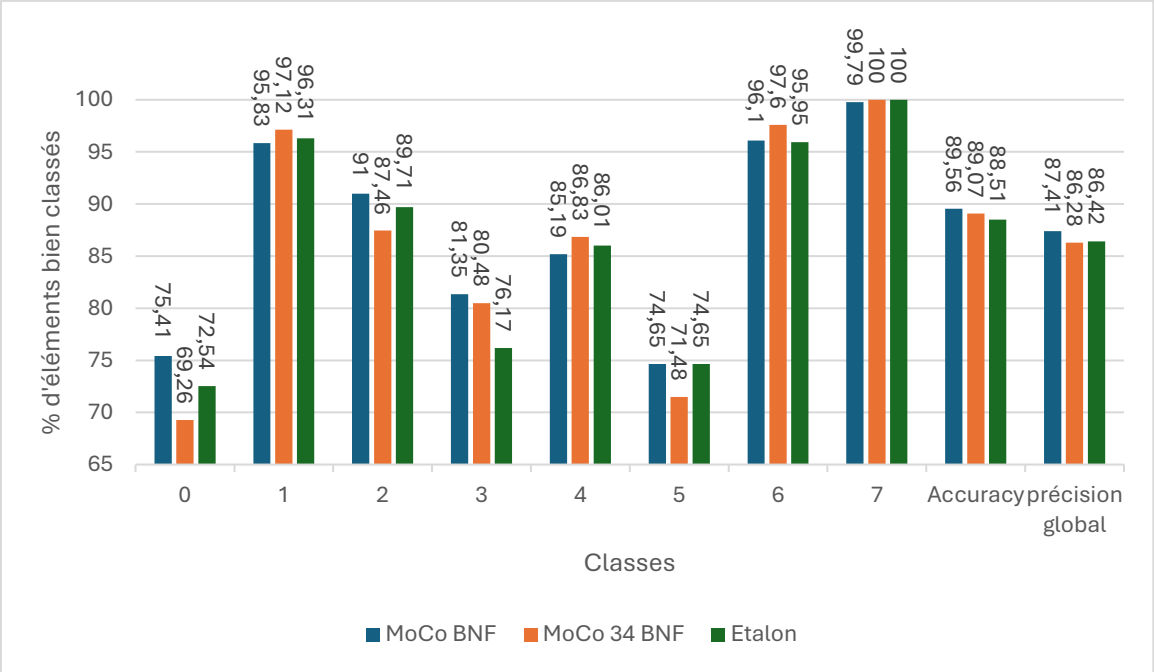
Après avoir analysé les différents modèles, on peut se poser une question, est-il possible d'améliorer les résultats des modèles en augmentant la taille du réseau de neurones et en le modifiant. Pour ce faire, nous allons modifier le réseau MoCo car celui-ci a donné les meilleurs résultats. La modification consiste à changer la « backbone » du modèle en la faisant passer d'un ResNet 18 à un ResNet 34 qui est beaucoup plus profond et devrait peut-être améliorer les résultats obtenus par le modèle.

Les tests seront effectués uniquement sur le « dataset BloodMNIST » et le dossier Full, ils devraient nous donner suffisamment de données pour voir si agrandir un réseau permet de nous fournir de meilleurs résultats.



Graphique de comparaison générale entre les modèles MoCo BNF, MoCo 34 BNF et étalon pour le « dataset BloodMNIST », dossier Full.

On peut effectuer plusieurs observations. D'abord, le modèle MoCo 34 BNF n'est pas meilleur que MoCo BNF (qui n'a qu'un ResNet 18) sur la précision globale même si l'écart est très très faible (0,03%). On peut également observer que pour certaines classes, un modèle est parfois meilleur que l'autre, par exemple pour la classe n°0 MoCo 34 BNF est meilleur que MoCo BNF mais on observe notamment le comportement inverse pour la classe n°4. Pour l'instant, il est clair que l'utilisation d'un ResNet plus grand n'apporte pas d'amélioration significative. Passons maintenant à l'analyse sur le dossier Quarter.



Graphique de comparaison générale entre les modèles MoCo BNF, MoCo 34 BNF et étalon pour le « dataset BloodMNIST », dossier Quarter.

Les observations qu'on peut effectuer sont les suivantes. La différence de précision entre les 2 modèles MoCo est bien plus importante, car cette fois-ci, MoCo 34 BNF n'arrive pas à atteindre la précision du modèle étalon (même si celui-ci est très proche). Comme pour le dossier Full, parfois, le modèle MoCo 34 BNF a de meilleurs résultats pour certaines classes et parfois des résultats bien en-dessous du modèle MoCo BNF, c'est vrai surtout pour les classes 0, 2 et 5.

Pour établir un résumé des résultats observés, ceux-ci sont légèrement inférieurs au modèle MoCo BNF, c'est dû au fait que le modèle MoCo 34 BNF semble mieux réussir l'apprentissage de certaines classes, mais cela se fait au détriment d'autres classes et il en résulte une précision moins bonne. Maintenant, on peut se demander pourquoi on a ce résultat ? On pourrait soumettre l'hypothèse que le résultat aurait été bien meilleur si les données avaient une taille plus grande car avec la structure de ResNet 34 l'image est réduite trop vite et donc on perd beaucoup d'informations et la forme n'est alors plus utilisable pas la suite du réseau.

10 Augmentation du nombre de données

Une autre méthode pour pouvoir améliorer les résultats de nos modèles serait d'utiliser plus de données. Pour cela, 2 possibilités : trouver de nouvelles données, utiliser de l'augmentation de données ou alors, créer des données synthétiques. Nous allons nous focaliser sur la première option, il est tout à fait possible de trouver de nouvelles données mais le mieux serait de trouver des données ayant les mêmes classes que celles que nous utilisons dans ce travail. Mais rien ne nous empêche de prendre des données qui sont similaires et de les proposer à notre réseau de neurones pour l'entrainement et il est plus que probable que ce dernier arrive à en extraire des représentations utiles qui pourront être utilisées par la suite pour la « downstream task ».

Pour ça, nous allons rechercher des données similaires au « dataset BloodMNIST », c'est-à-dire des images de cellules présentes dans le sang. Ces nouvelles données ne seront utilisées que lors de l'entrainement du modèle avec la tâche prétexte.

2 nouveaux « datasets » ont été utilisés (totalement ou en partie).

1) Blood Cell Images
2) Blood Cells Cancer (ALL) dataset

Le « dataset » n°1 regroupe des grandes images contenant plusieurs cellules, globules rouges, globules blancs et des plaquettes. Celles-ci peuvent être extraites grâce à des coordonnées fournies (avec étiquetage), comme montre l'exemple ci-dessous.

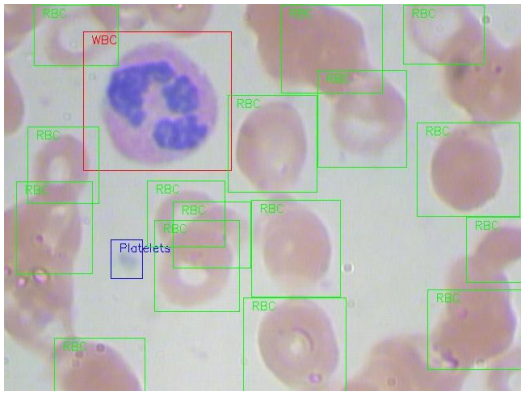


Figure 10.1 : image montrant les différentes cellules qui peuvent être extraites d'une image du « dataset » n°1.

Les cellules extraites sont : les globules blancs, les globules rouges et les plaquettes. Les images finales ont été redimensionnées en 28 px pour pouvoir être utilisées par nos modèles. On se retrouve avec 2 inconvénients. Le premier est que les cellules se chevauchent mais étant donné que ça arrive parfois dans « BloodMNIST », ce n'est pas préoccupant. Le second est que les coordonnées fournies donnent souvent des rectangles. Pour régler ce problème, nous allons prendre un carré qui englobe les coordonnées fournies pour chaque cellule, comme ça on est certain d'avoir des images carrées ; ceci pour ne pas avoir de problèmes de redimensionnement des images plus tard.

Le « dataset » n°2 contient des grandes images avec plusieurs cellules, cependant on ne possède pas de coordonnées pour pouvoir les récupérer. Et dû à leur nombre, le faire à la main n'est pas possible. Alors, nous avons décidé de découper l'image en patchs de 244 px et ensuite de redimensionner ces patchs en 28 px. Cependant, cette méthode nous fait perdre de l'information (2 bandes rectangulaires, une en bas et à droite). Ceci car les images d'origine ne sont pas carrées et notons aussi qu'il reste une échelle en bas à droite, rendant cette partie inutilisable.

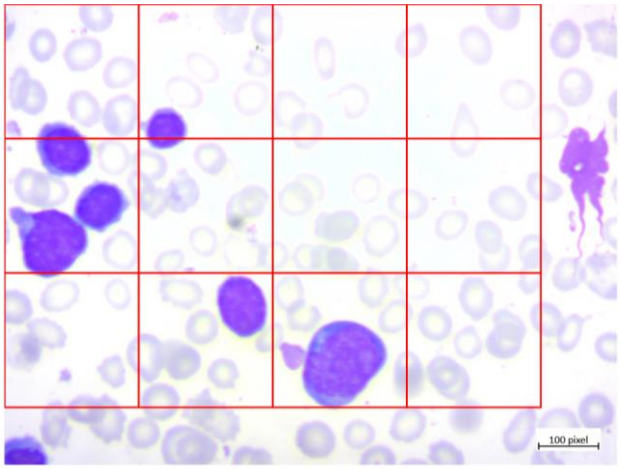


Figure 10.2 : image montrant les découpages d'une image du « dataset » n°2, on peut clairement voir les 2 bandes qui ne sont pas utilisées ainsi que l'échelle en bas à droite de l'image.

Le « dataset » contient 4 classes, une contenant des cellules provenant de patients sains, les 3 autres de patients atteints d'une autre forme de leucémie. Il a été choisi pour ce travail de prendre les cellules provenant de patients sains (car plus proche de « BloodMNIST » qui lui aussi provient de patients sains).

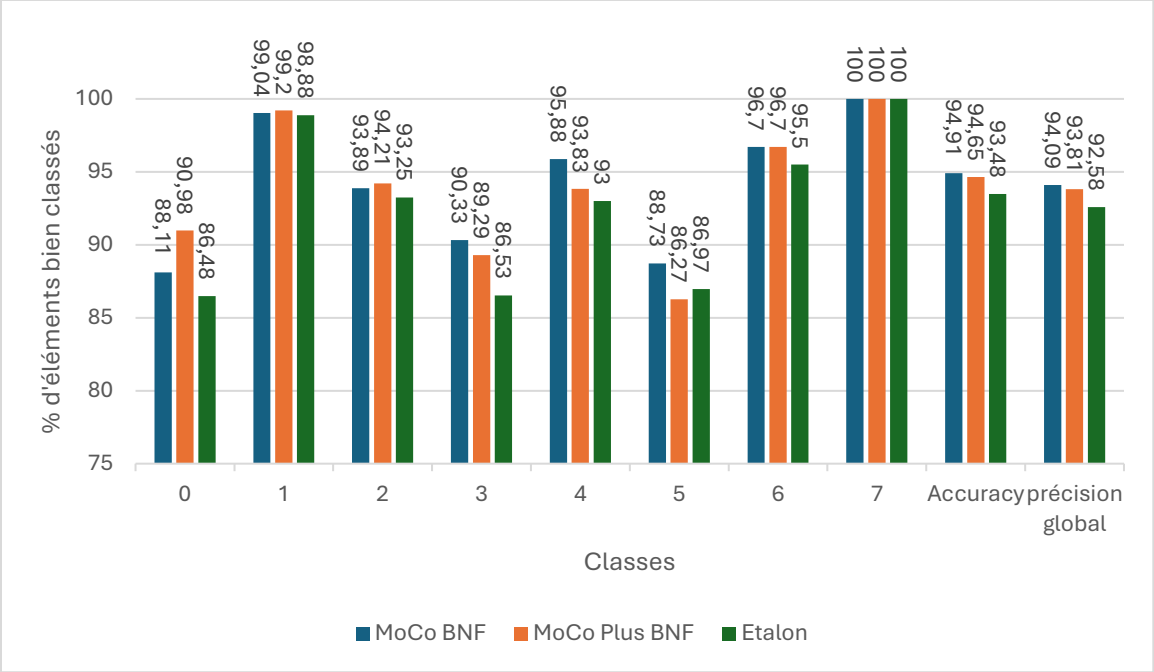
Une fois les images traitées (découpées, redimensionnées), on se retrouve pour le « dataset » n°1 avec 4889 images et 6144 images pour le « dataset » n°2 soit un total de 11 033 images. Ce qui est très proche du nombre d'images contenues dans « BloodMNIST » (11 959).

On peut remarquer, grâce aux images d'exemples, que d'une part les 2 « datasets » ont des images bien différentes, mais d'autre part qu'elles sont également bien différentes de celles de BloodMNIST. Malheureusement, le « dataset » n°1 ne donne pas la méthode utilisée pour la collecte de données ; cependant, les données devraient être assez similaires pour que le modèle SSL puisse les utiliser. (La différence majeure est la couleur mais le reste est plutôt similaire). Quant au « dataset » n°2, il a été créé en utilisant une caméra Zeiss sur un microscope avec un grossissement 100x. Pour BloodMNIST, les images ont été obtenues via un CellaVision DM96)[45] (un appareil pour visualiser le contenu du sang et voir les différentes cellules). Donc, une

méthode différente. Il est très difficile de trouver des « datasets » d'images de cellules sanguines qui utilisent tous les mêmes méthodes et le même équipement (car il n'y a pas de standards).

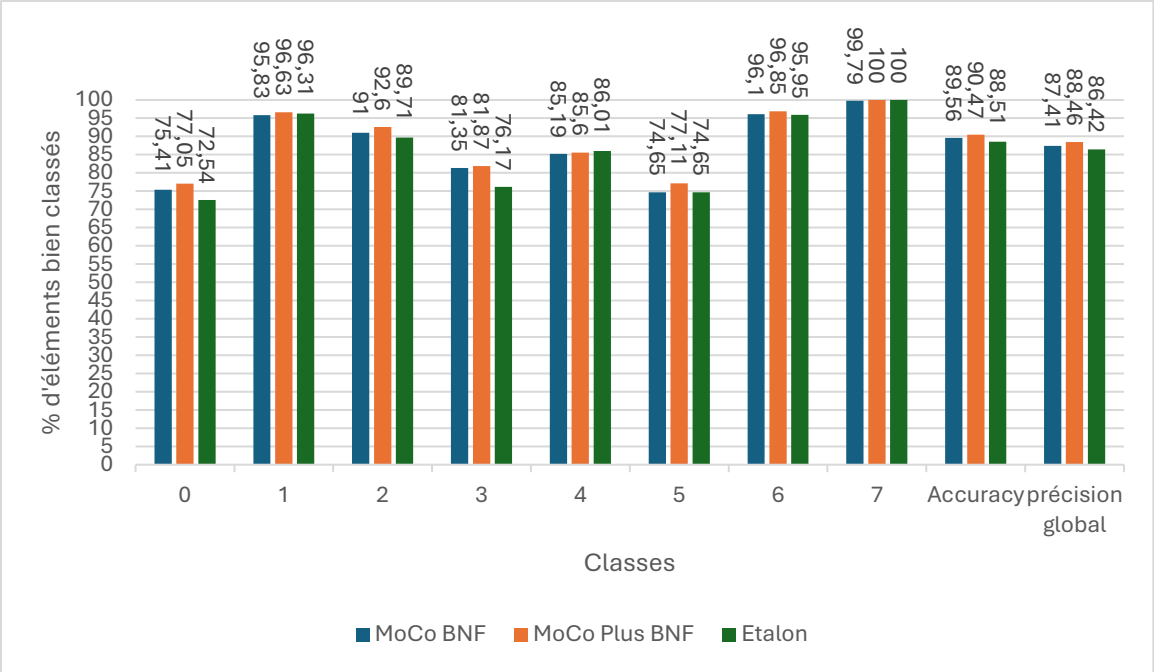
De ce fait, la décision a été prise de, malgré les différences, utiliser les 2 « datasets » et de voir les résultats.

Nous allons utiliser encore une fois MoCo pour les tests, pour ceux -ci nous allons tout d'abord tester MoCo puis MoCo BNF.



Graphique de comparaison générale entre les modèles MoCo BNF, MoCo Plus BNF et étalon pour le » dataset BloodMNIST », dossier Full.

On se retrouve avec un résultat assez similaire à celui du point n°9, le modèle a une précision qui est très proche mais inférieure à celle du modèle MoCo BNF. On obtient aussi des effets similaires sur la précision des classes, certains sont meilleurs pour un des modèles MoCo et d'autres sont moins bons. Il est possible que les images supplémentaires rendent l'apprentissage de certaines classes au modèle plus compliqué, donnant ainsi une précision légèrement plus faible. Dans une situation où on a beaucoup de données labellisées, utiliser davantage de données pour entrainer le modèle (autres données, similaires mais pas forcément des bonnes classes), ne semble pas avoir d'effet positif. Nous allons maintenant passer à l'analyse du modèle mais pour le dossier Quarter.



Graphique de comparaison générale entre les modèles MoCo BNF, MoCo Plus BNF et étalon pour le « dataset BloodMNIST », dossier Quarter.

Dans ce cas-ci, on se retrouve avec l'inverse des résultats obtenus pour le dossier Full, le modèle MoCo Plus BNF est très légèrement supérieur (de 1.05%). Cela s'explique car le modèle est quasiment meilleur dans toutes les classes (sauf la classe n°3). A l'analyse de ces résultats, nous pouvons émettre comme hypothèse que lorsqu'on a peu de données labellisées, rajouter d'autres données semble avoir un impact assez positif, même s'il faut garder à l'esprit que doubler les données disponibles n'a donné une amélioration que de 1.05%.

11 Conclusion

Après diverses observations et analyses, nous pouvons mettre en évidence plusieurs concepts.L'un d'entre eux est l'importance des modèles « self supervised learning », celle-ci risque d'augmenter dans les prochaines années. En effet, ces modèles peuvent fournir des résultats comparables et, suivant les circonstances, meilleurs que des modèles supervisés plus classiques. Notamment par l'utilisation de « end to end fine-tuning » pour entrainer le classificateur ou/et la présence en quantité suffisante de données non labellisées. Même si ceux-ci semblent souffrir de pas mal de problèmes lors de l'entrainement sur des données avec beaucoup de déséquilibre de classes.

Cette dernière situation est particulièrement présente dans certains milieux notamment celui de l'imagerie médicale où les modèles SSL ont montré de très bonnes performances.

Concernant le choix d'un modèle, celui-ci va déprendre de plusieurs facteurs mais dans une application similaire à celle développée dans ce travail, MAE semble être le meilleur choix, vu les résultats obtenus, avec en seconde position MoCo V2 .

Cependant, il faut garder à l'esprit que celui-ci, a comme chaque modèle, des avantages et des faiblesses et il faut toujours les prendre en considération pour le choix d'un modèle.

Concernant les modèles fusions, ils ont donné dans l'ensemble de très bons résultats.

Le meilleur modèle étant le modèle FSAAS, son utilisation est donc fortement recommandée. En revanche, on ne peut pas en dire autant pour l'augmentation du modèle qui n'a pas donné de résultats concluants. Concernant l'ajout de données supplémentaires, cela pourrait être exploré plus en profondeur, davantage que dans ce travail, car il semble que de meilleurs résultats ont été observés dans certains cas, notamment celui où le nombre de données labellisées est faible.

Néanmoins, lorsqu'on dispose de grandes quantités de données non labellisées et peu de données labellisées alors l'utilisation de modèles SSL peut donner des résultats fiables et surtout permet de gagner beaucoup de temps en ne labellisant pas des milliers de données supplémentaires permettant ainsi d'investir ce temps dans autre chose, comme la collecte de nouvelles données ou des tests d'hyperparamètres pour l'entrainement .

12 Annexes

12.1 Figures

1) *Réseaux de neurones convolutifs : concepts et fonctionnement*

Figure 1.1 : (01-07-2024)

URL : https://fr.wikipedia.org/wiki/R%C3%A9seau_de_neurones_artificiels (remarque, celle-ci a été modifiée pour y inclure le biais)

Figure 1.2 : implémentation de la fonction sigmoïde dans le logiciel GeoGebra (par moi-même)

Figure 1.3 : (01-07-2024) [47]

Figure 1.4 : exemple de 2 kernels, (par moi-même)

Figure 1.5 :(10-12-2024) image d'un chat après l'utilisation d'un filtre horizontal et vertical URL : https://setosa.io/ev/image-kernels/ (image prise par moi-même)

Figure 1.6 : (29-06-24)

URL : https://jamesmccaffrey.wordpress.com/2018/05/30/convolution-image-size-filter-size-padding-and-stride/convolution_math/

Figure 1.7 : (29-06-24)

URL : https://jamesmccaffrey.wordpress.com/2018/05/30/convolution-image-size-filter-size-padding-and-stride/convolution_math/

Figure 1.8 : [19]

Figure 1.9 : (29-06-24)Pooling Illustration URL: https://www.digitalocean.com/community/tutorials/pooling-in-convolutional-neural-networks

Figure 1.10 : (17-07-2024) URL :https://pytorch.org/docs/stable/generated/torch.nn.ReLU.html

Figure 1.11 : (05-06-2024)[21]

Figure 1.12 : (05-06-2024)[21]

Figure 1.13 : (05-06-2024)[21]

Figure 1.14 : (06-12-2024)[35]

Figure 1.15 : (06-12-2024)[35]

2) *Application de l'IA en imagerie médicale*

Figure 2.1 : (22-04-24) URL :https://www.cliniciansbrief.com/article/hairline-bone-fracture

Figure 2.2 : (22-04-24) URL :https://www.norimagerie.com/teleradiographie-du-crane-face-et-profil

Figure 2.3 : (22-04-24) URL :https://www.norimagerie.com/teleradiographie-du-crane-face-et-profil

Figure 2.4 : (22-04-24) [49]

Figure 2.5 : image provenant du dataset BloodMNIST [50]

Figure 2.6 : (03-08-2024) images du Titanic et de l'Olympique URL : https://www.jmilford-titanic.com/2014/03/differences-between-olympic-and-titanic.html

Figure 2.7 : (02-07-24) URL :https://thispersondoesnotexist.com/

Figure 2.8 : (08-08-2024) [18]

3) Fonctionnement de l'apprentissage auto-supervisé

Figure 3.1 : SSL (21-06-24) URL :https://www.v7labs.com/blog/self-supervised-learning-guide

Figure 3.2 : SSL (21-06-24)[8]

Figure 3.3 : SSL (21-06-24)[8]

Figure 3.4 : SSL (21-06-24)[8]

Figure 3.5 : SSL (21-06-24)[8]

Figure 3.6 : SSL (21-06-24)[8]

Figure 3.7 : SSL (21-06-24)[8]

Figure 3.8 : SSL (21-06-24)[8]

Figure 3.9 : SSL (21-06-24)[8] URL :https://www.cliniciansbrief.com/article/hairline-bone-fracture

Figure 3.10 : image d'un chat après avoir retiré une partie de l'image , (par moi-même)

4) Comparaison des différents types de tâches prétextes

Figure 4.1 : exemple de 3 transformations sur une image de chat, (par moi-même)

Figure 4.2 : (15-05-2024) [9]

Figure 4.3 : (30-06-2024) [10]

Figure 4.4 : (30-06-2024) [12]

Figure 4.5 : (30-06-2024) [12]

Figure 4.6 : (30-06-2024) [13]

Figure 4.7 : (30-06-2024) [13]

Figure 4.8 : (30-06-2024) [13]

5) Choix du dataset et modèle basique pour comparaison

Figure 5.0 : (08-08-24) MedMNIST [50]

Figure 5.1 : (22-04-24) Image provenant du dataset BloodMNIST [50]

Figure 5.2 : (22-04-24) Image provenant du dataseRetinaMNIST [50]

Figure 5.3 : (17-04-24) URL :https://alexlenail.me/NN-SVG/LeNet.html)

6) *Implémentation*

7) *Analyse des résultats des modèles*

Figure 7.1 : (30-06-2024) [10]

8) *Augmentation du nombre de données*

Figure 10.1 : (29-11-2024) Image d'exemple de découpages fournis par le dataset. [42]

Figure 10.2 : (29-11-2024) Exemple de découpage effectuer sur le dataset n°2.[10]

12.2 Bibliographie

[1] (17-07-2024) URL : https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html

[2] (17-07-2024) URL : https://pytorch.org/docs/stable/generated/torch.nn.MaxPool2d.html

[3](17-07-2024) URL : https://pytorch.org/docs/stable/generated/torch.nn.AvgPool2d.html

[4](17-07-2024) URL : https://pytorch.org/docs/stable/generated/torch.nn.Dropout.html

[5](17-07-2024) URL : https://pytorch.org/docs/stable/generated/torch.nn.ReLU.html

[6](17-07-2024) URL : https://en.wikipedia.org/wiki/Cross-entropy

[7](24-03-2024) Dive into the Details of Self-Supervised Learning for Medical Image Analysis URL : https://arxiv.org/abs/2209.12157

[8](20-10-2024) Self-supervised learning for medical image classification: a systematic review and implementation guidelines URL : https://www.nature.com/articles/s41746-023-00811-0

[9](15-05-2024) A Simple Framework for Contrastive Learning of Visual Representations URL :

https://arxiv.org/abs/2002.05709

[10](30-06-2024) Momentum Contrast for Unsupervised Visual Representation Learning URL : https://arxiv.org/abs/1911.05722v3

[11](30-06-2024) Improved Baselines with Momentum Contrastive Learning URL : https://arxiv.org/abs/2003.04297

[12](30-06-2024) Scalable Pre-training of Large Autoregressive Image Models URL : https://arxiv.org/abs/2401.08541

[13](30-06-2024) Masked Autoencoders Are Scalable Vision Learners URL : https://arxiv.org/abs/2111.06377

[14](30-06-2024) Self-supervised Learning: A Succinct Review URL : https://link.springer.com/article/10.1007/s11831-023-09884-2

[15](29-07-2024) AsymMirai: Interpretable Mammography-based Deep Learning Model for 1–5-year Breast Cancer Risk Prediction URL : https://pubs.rsna.org/doi/10.1148/radiol.232780

[16](30-07-2024) https://www.kaggle.com/datasets/sovitrath/diabetic-retinopathy-2015-data-colored-resized

[17] (1-08-2024) Survey on deep learning with class imbalance URL : https://journalofbigdata.springeropen.com/articles/10.1186/s40537-019-0192-5

[18] (1-08-2024) Trends in Fracture Incidence: A Population-Based Study Over 20 Years

URL : https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3929546/

[19] (29-06-2024) A Study of CNN and Transfer Learning in Medical Imaging: Advantages, Challenges, Future Scope URL : https://www.mdpi.com/2071-1050/15/7/5930

[20] (03-08-2024) ImagNet URL : https://image-net.org/download.php

[21] (05-08-2024) Deep Residual Learning for Image Recognition URL : https://ieeexplore.ieee.org/document/7780459

[22] (05-08-2024) Dropout: A Simple Way to Prevent Neural Networks from Overfitting URL : https://www.cs.toronto.edu/~rsalakhu/papers/srivastava14a.pdf

[23](07-08-2024) The Origin and Value of Disagreement Among Data Labelers: A Case Study of the Individual Difference in Hate Speech Annotation URL : https://arxiv.org/abs/2112.04030

[24](07-08-2024) Assessing Inter-Annotator Agreement for Medical Image Segmentation URL : https://www.ncbi.nlm.nih.gov/pmc/articles/PMC10062409/

[25](08-08-2024) Fonction SoftMax URL : https://en.wikipedia.org/wiki/Softmax_function

[26] (08-08-2024) Deep Sparse Rectifier Neural Networks URL : https://proceedings.mlr.press/v15/glorot11a

[27] (08-08-2024) Dermatologist-level classification of skin cancer with deep neural networks URL :https://www.nature.com/articles/nature21056

[28](08-08-2024) Generative adversarial network in medical imaging: A review

URL :https://www.sciencedirect.com/science/article/abs/pii/S1361841518308430

[29] (08-08-2024) Gaussian blur

[30](16-12-2024)Exemple de cout pour labélisée des données URL : https://cloud.google.com/ai-platform/data-labeling/pricing?hl=fr

URL : https://en.wikipedia.org/wiki/Gaussian_blur

[31](09-08-2024) URL : https://docs.lightly.ai/self-supervised-learning/tutorials/package/tutorial_moco_memory_bank.html#imports

[32](08-08-2024) URL : https://en.wikipedia.org/wiki/Kernel_(image_processing)

[33] (13-08-2024) Generative Adversarial Nets URL : https://arxiv.org/abs/1406.2661

[34](13-08-2024) The CIFAR-10 dataset URL : https://www.cs.toronto.edu/~kriz/cifar.html

[35](13-08-2024) AN IMAGE IS WORTH 16X16 WORDS: TRANSFORMERS FOR IMAGE RECOGNITION AT SCALE URL : https://arxiv.org/pdf/2010.11929

[36](10-10-2024) Image Data Augmentation for Deep Learning: A Survey URL : https://arxiv.org/abs/2204.08610

[37](15-10-2024) Learning Rate URL : https://en.wikipedia.org/wiki/Learning_rate

[38](15-10-2024) https://en.wikipedia.org/wiki/Mathematical_optimization

[39](16-10-2024) WHAT IS ARTIFICIAL INTELLIGENCE? (John McCarthy) URL : https://www-formal.stanford.edu/jmc/whatisai.pdf

[40](18-10-2024)Max Pooling URL : https://fr.wikipedia.org/wiki/Max_pooling

[41](08-08-2024)MIT Deep Learning Book URL : https://github.com/janishar/mit-deep-learning-book-pdf?tab=readme-ov-file

[42](27-11-2024) Blood Cell Images
URL :https://www.kaggle.com/datasets/paultimothymooney/blood-cells/discussion/63703

[43](27-11-2024) Blood Cells Cancer (ALL) dataset

URL : https://www.kaggle.com/datasets/mohammadamireshraghi/blood-cell-cancer-all-4class

[44](29-11-2024) Couche linéaire
URL :https://pytorch.org/docs/stable/generated/torch.nn.Linear.html

[45](04-12-2024)BloodMNIST URL :
https://www.sciencedirect.com/science/article/pii/S2352340920303681

[46](27-11-2024) Fusion de model URL :

https://fr.wikipedia.org/wiki/Bootstrap_aggregating

[47](16-12-2024) Réseaux de neurones URL :

https://en.wikipedia.org/wiki/Neural_network_(machine_learning)

[48](16-12-2024) Imagerie médical URL :

https://en.wikipedia.org/wiki/Medical_imaging

[49](16-12-2024) Erreur dans le dataset Cifar 10 URL :https://labelerrors.com/

[50](16-12-2024)Dataset MedMnist URL : https://medmnist.com/

12.3 Données

12.3.1 Données des modèles SSL

Les tableaux ci-dessous reprennent les données brutes obtenues lors de ce travail pour les différents modèles SSL SimCLR, MoCo, AIM, MAE, ainsi que pour le modèle étalon:

Dataset blood dossier full

Normal (backbone frozen)	Etalon	SimCLR	MoCo	AIM	MAE
0	86,48	60,66	61,07	67,62	51,23
1	98,88	90,38	90,87	93,43	91,35
2	93,25	78,46	80,39	82,32	85,53
3	86,53	68,05	67,7	73,58	64,42
4	93	84,77	81,48	81,07	76,54
5	86,97	61,97	57,04	69,37	51,76
6	95,5	92,64	92,04	91,74	88,29
7	100	98,09	98,3	97,87	98,94
Accuracy	93,48	82,14	81,64	84,62	79,51
Accuracy moyenne	92,58	79,38	78,61	82,12	76,01

BNF (backbone not frozen)	Etalon	SimCLR BNF	MoCo BNF	AIM BNF	MAE BNF
0	86,48	84,02	88,11	89,75	88,52
1	98,88	98,4	99,04	98,88	98,88
2	93,25	92,93	93,89	92,6	94,21
3	86,53	88,08	90,33	88,43	87,22
4	93	95,47	95,88	91,36	96,3
5	86,97	88,03	88,73	88,03	88,73
6	95,5	94,89	96,7	96,85	96,7
7	100	99,79	100	100	100
Accuracy	93,48	93,57	94,91	94,21	94,45
Accuracy moyenne	92,58	92,7	94,09	93,24	93,82

Dataset blood dossier quarter

Normal(backbone frozen)	Etalon	SimCLR	MoCo	AIM	MAE
0	72,54	59,43	57,38	43,03	30,74
1	96,31	87,98	88,3	88,46	86,38
2	89,71	74,28	77,17	68,81	74,92
3	76,17	64,94	62,69	63,73	62
4	86,01	80,25	79,84	72,02	72,02
5	74,65	52,82	49,3	51,06	42,96
6	95,95	92,49	91,14	88,89	81,98
7	100	97,87	98,09	97,02	98,72
Accuracy	88,51	79,57	78,81	76,24	73,46
Accuracy moyenne	86,42	76,26	75,49	71,63	68,71

BNF (backbone not frozen)	Etalon	SimCLR BNF	MoCo BNF	AIM BNF	MAE BNF
0	72,54	77,05	75,41	81,97	77,87
1	96,31	93,91	95,83	97,44	98,56
2	89,71	89,07	91	88,42	91,96
3	76,17	79,97	81,35	81,17	79,45
4	86,01	83,13	85,19	89,71	91,36
5	74,65	69,01	74,65	77,82	79,93
6	95,95	95,05	96,1	95,2	96,4
7	100	99,79	99,79	100	99,36
Accuracy	88,51	88,1	89,56	90,5	90,88
Accuracy moyenne	86,42	85,87	87,41	88,97	89,36

Dataset derma dossier full

Normal(backbone frozen)	Etalon	SimCLR	MoCo	AIM	MAE
0	50	30,3	33,33	21,21	3,03
1	55,34	26,21	38,83	44,66	44,66
2	53,64	31,36	46,82	48,18	56,82
3	34,78	0	8,7	0	0
4	37,67	34,53	46,64	43,5	38,12
5	89,19	85,83	83,3	82,85	78,75
6	68,97	3,45	3,45	51,72	0
Accuracy	75,61	67,08	69,28	69,28	65,54
Accuracy moyenne	55,65	30,24	37,29	41,73	31,62

BNF (backbone not frozen)	Etalon	SimCLR BNF	MoCo BNF	AIM BNF	MAE BNF
0	50	50	45,45	40,91	34,85
1	55,34	54,37	51,46	47,57	51,46
2	53,64	47,27	50,91	47,27	50,45
3	34,78	17,39	17,39	13,04	21,74
4	37,67	42,15	46,19	35,87	50,22
5	89,19	90,38	88,52	88,44	85,23
6	68,97	55,17	55,17	58,62	68,97
Accuracy	75,61	75,76	75,06	73,12	73,17
Accuracy moyenne	55,65	50,96	50,73	47,39	51,85

Dataset derma dossier quarter

Normal(backbone frozen)	Etalon	SimCLR	MoCo	AIM	MAE
0	28,79	30,3	21,21	13,64	1,52
1	41,75	27,18	20,39	32,04	32,04
2	43,18	30,91	44,09	49,09	56,82
3	13,04	0	0	0	0
4	29,6	25,11	32,74	35,87	30,49
5	86,73	87,77	83,89	80,84	79,27
6	31,03	0	0	6,9	0
Accuracy	69,73	67,28	66,33	65,64	64,34
Accuracy moyenne	39,16	28,75	28,9	31,2	28,59

BNF (backbone not frozen)	Etalon	SimCLR BNF	MoCo BNF	AIM BNF	MAE BNF
0	28,79	37,88	33,33	4,85	30,3
1	41,75	37,86	41,75	35,92	44,66
2	43,18	39,55	39,09	46,82	50,91
3	13,04	13,04	17,39	0	0
4	29,6	30,49	36,32	29,15	48,43
5	86,73	87,25	86,13	85,53	76,88
6	31,03	6,9	20,69	37,93	62,07
Accuracy	69,73	69,53	69,68	69,13	66,58
Accuracy moyenne	39,16	36,14	39,24	38,6	44,75

12.3.2 Données des modèles SSL Fusions

Les tableaux ci-dessous reprennent les données brutes obtenues lors de ce travail pour les différents modèles fusions : FSM,FAA,FSAAM, ainsi que pour le modèle étalon :

Dataset blood dossier full

BNF (backbone not frozen)	Etalon	FSM BNF	FAA BNF	FSAAM BNF
0	86,48	87,3	89,75	92,62
1	98,88	98,72	99,2	99,36
2	93,25	93,89	94,86	94,53
3	86,53	90,5	90,33	92,23
4	93	96,71	94,65	96,71
5	86,97	88,03	88,73	90,49
6	95,5	96,1	97,75	97,3
7	100	100	100	100
Accuracy	93,48	94,71	95,26	96
Accuracy moyenne	92,58	93,9	94,41	95,41

Dataset blood dossier quarter

BNF (backbone not frozen)	Etalon	FSM BNF	FAA BNF	FSAAM BNF
0	72,54	79,51	82,38	85,66
1	96,31	96,47	98,56	98,4
2	89,71	90,35	90,68	92,28
3	76,17	82,73	84,46	88,08
4	86,01	85,6	91,36	91,77
5	74,65	73,94	79,93	82,39
6	95,95	95,95	97,45	97,45
7	100	100	99,79	100
Accuracy	88,51	90,12	92,2	93,42
Accuracy moyenne	86,42	88,07	90,57	92

Dataset derma dossier full

BNF (backbone not frozen)	Etalon	FSM BNF	FAA BNF	FSAAM BNF
0	50	51,52	37,88	51,52
1	55,34	53,4	55,34	59,22
2	53,64	53,64	52,27	50,45
3	34,78	30,43	26,09	30,43
4	37,67	44,84	37,67	42,6
5	89,19	90,83	90,23	92,77
6	68,97	55,17	72,41	62,07
Accuracy	75,61	77,21	75,71	78,3
Accuracy moyenne	55,65	54,26	53,13	55,58

Dataset derma dossier quarter

BNF (backbone not frozen)	Etalon	FSM BNF	FAA BNF	FSAAM BNF
0	28,79	33,33	28,79	33,33
1	41,75	41,75	39,81	44,66
2	43,18	40	52,73	45
3	13,04	13,04	0	13,04
4	29,6	31,39	30,49	30,49
5	86,73	88,07	85,61	89,34
6	31,03	13,79	51,72	24,14
Accuracy	69,73	70,37	70,17	71,97
Accuracy moyenne	39,16	37,34	41,31	40

12.3.3 Données des modèles MoCo 34

Les tableaux ci-dessous reprennent les données brutes obtenues lors de ce travail pour les différents modèles MoCo 34 ainsi que pour le modèle étalon :

Dataset blood dossier full

Normal(backbone frozen)	Etalon	MoCo 34
0	86,48	54,92
1	98,88	86,86
2	93,25	73,63
3	86,53	61,66
4	93	82,72
5	86,97	49,65
6	95,5	89,19
7	100	95,96
Accuracy	93,48	77,43
Accuracy moyenne	92,58	74,32

BNF (backbone not frozen)	Etalon	MoCo 34 BNF
0	86,48	89,75
1	98,88	99,36
2	93,25	94,21
3	86,53	90,67
4	93	94,24
5	86,97	86,97
6	95,5	97,3
7	100	100
Accuracy	93,48	95,03
Accuracy moyenne	92,58	94,06

Dataset blood dossier quarter

BNF (backbone not frozen)	Etalon	MoCo 34 BNF
0	72,54	69,26
1	96,31	97,12
2	89,71	87,46
3	76,17	80,48
4	86,01	86,83
5	74,65	71,48
6	95,95	97,6
7	100	100
Accuracy	88,51	89,07
Accuracy moyenne	86,42	86,28

Tableaux reprenant les données brutes obtenues lors de ce travail pour le modèle MoCo Plus, ainsi que pour le modèle étalon :

Dataset blood dossier full

BNF (backbone not frozen)	Etalon	MoCo Plus BNF
0	86,48	90,98
1	98,88	99,2
2	93,25	94,21
3	86,53	89,29
4	93	93,83
5	86,97	86,27
6	95,5	96,7
7	100	100
Accuracy	93,48	94,65
Accuracy moyenne	92,58	93,81

Dataset blood dossier quarter

BNF (backbone not frozen)	Etalon	MoCo Plus BNF
0	72,54	77,05
1	96,31	96,63
2	89,71	92,6
3	76,17	81,87
4	86,01	85,6
5	74,65	77,11
6	95,95	96,85
7	100	100
Accuracy	88,51	90,47
Accuracy moyenne	86,42	88,46

12.4 Code python des différents modèles :

12.4.1 Etalon

```
# Chaque bloc possède un lien vers la source originale, formaté comme ceci :
#Source: []

# Rappels avant de lancer le programme :
#   - bien vérifier le dataset et le dossier utilisé.
#   - bien vérifier les paramètres et les dossiers de sauvegarde.
#   - faire attention de ne pas lancer le modèle alors qu'il a déjà été entraîné.
#   - Si le premier epoch est très lent, c'est normal, il suffit d'attendre un peu que
les données soient chargées.

# Cette cellule contient les imports nécessaires au bon fonctionnement du code.

import torch
import torchvision
import torch.nn as nn
```

```
import torch.nn.functional as F
import torch.optim as optim

import torchvision.transforms.v2 as transformsV2
from torchsummary import summary
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision import datasets, models

import os
import numpy as np
import matplotlib.pyplot as plt
import random

from PIL import Image


import seaborn as sns
import pandas as pd

from sklearn.metrics import confusion_matrix

torch.__version__
In [ ]:
# Fonction utilisée pendant le programme pour donner une seed aux différentes
fonctions aléatoires.
# A mettre au début de chaque cellule.
# Source : [https://pytorch.org/docs/stable/notes/randomness.html]
def stop_random():
  seed=621
  torch.manual_seed(seed)
  torch.cuda.manual_seed(seed)
  random.seed(seed)
  np.random.seed(seed)
  torch.backends.cudnn.deterministic = True
  torch.backends.cudnn.benchmark = False

In [ ]:
# Configuration générale. Attention à bien choisir les bons paramètres et le bon
emplacement de sauvegarde.
stop_random()
nb_Epoch = 100
batch_Size = 128
taille_Image = 28
augmentation=False
dossier = "Full"
dataset = "Blood"#Derma
BNF = False
#Nomenclature:Nom_Model#Dossier#Epoch#BNF
model_Path = './PytorchModelV3/'+"Res18#"+dossier
+"#"+dataset+str(nb_Epoch) +"#" +str(BNF)+".pth"
path_Train = dataset+"_"+dossier+"/Dataset/train"
path_Val =  dataset+"_"+dossier+"/Dataset/val"
nb_Classe = len(next(os.walk(dataset+"_"+dossier+"/test"))[1])
print("Model choisi:RES18, avec le dossier: " +dossier+ ", avec "+
str(nb_Epoch) + " epoch")
print("Emplacement: " +model_Path)
In [ ]:
stop_random()
```

```
transform_Train = transformsV2.Compose([
   transformsV2.ToTensor(),

])

# Si l'on a besoin de faire de l'augmentation de données.
if(augmentation):
   print("augmentation:")
   transform_Train = transformsV2.Compose([
   transformsV2.RandomResizedCrop(size=28, scale=(0.3, 1.0), ratio=(1, 1)),
   transformsV2.RandomHorizontalFlip(),
   transformsV2.ToTensor(),
   transformsV2.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
])

transform = transformsV2.Compose([
   transformsV2.ToTensor(),
])

trainset= datasets.ImageFolder(root=path_Train, transform=transform_Train)
valset = datasets.ImageFolder(root=path_Val, transform=transform)

print(transform_Train)
In [ ]:
#Source:[https://pytorch.org/hub/pytorch_vision_resnet/]
# On récupère un modèle ResNet18 non entraîné.
stop_random()
import torch
import torch.nn as nn
model = torch.hub.load('pytorch/vision:v0.10.0', 'resnet18',
pretrained=False)
num_features = model.fc.in_features
model.fc = nn.Linear(num_features, nb_Classe)
device = torch.device('cuda:0')
net = model
net.to(device)
In [ ]:
#Source:[https://saturncloud.io/blog/how-to-use-class-weights-with-focal-
loss-in-pytorch-for-imbalanced-multiclass-classification/]
stop_random()
# Chargement des données et gestion des poids des classes.
trainloader = torch.utils.data.DataLoader(trainset,
batch_size=batch_Size,shuffle=True, num_workers=4)
valloader = torch.utils.data.DataLoader(valset,
batch_size=batch_Size,shuffle=False, num_workers=4)
classes = trainset.classes
num_classes = len(trainset.classes)

nom_Classe= trainset.targets
training_Nb_Exemple = torch.bincount(torch.tensor(nom_Classe))
total = len(trainset)


class_weights = []
for exemple in training_Nb_Exemple:
   poid = (total-exemple) / (total)
   class_weights.append(poid)
   print(poid)
```

```
#Source:[https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLos
s.html]
class_weights = torch.tensor(class_weights,dtype=torch.float32).cuda()
criterion = nn.CrossEntropyLoss(weight=class_weights)
optimizer = optim.SGD(net.parameters(), lr=0.1, momentum=0.9)
In [ ]:
#Source:[https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html]
#Source:[https://saturncloud.io/blog/calculating-the-accuracy-of-pytorch-
models-every-epoch/]

stop_random()
import numpy as np
#training
t_accuracy_tab = []
t_loss_tab = []

#validation
v_accuracy_tab = []
v_loss_tab = []

epochs = []

nb_correct= 0
nb_exemple = 0
lr = 0.1
optimizer = optim.SGD(net.parameters(), lr, momentum=0.9)

for epoch in range(nb_Epoch):
  t_accuracy = 0
  t_loss = 0

  v_accuracy = 0
  v_loss = 0

  nb_exemple = 0
  nb_correct = 0
  net.train()

  if(epoch%20==0):
    lr = lr/2
    print(lr)
    if(lr < 0.0005):
      lr = 0.0005
    optimizer = optim.SGD(net.parameters(), lr, momentum=0.9)

  for i, data in enumerate(trainloader, 0):

    inputs, labels = data[0].to(device), data[1].to(device)# GPU
    optimizer.zero_grad()
    outputs = net(inputs)

    loss = criterion(outputs, labels)
    loss.backward()
    optimizer.step()
    t_loss += loss.item()
    _, predicted = torch.max(outputs, 1)

    nb_exemple += labels.size(0)
    nb_correct += (predicted == labels).sum().item()
```

```
  t_loss = t_loss/len(trainloader)#Moyenne de la loss sur tout l'epoch
  t_accuracy = 100*(nb_correct /nb_exemple)#Accuracy moyenne sur un epoch
  t_accuracy_tab.append(t_accuracy)
  t_loss_tab.append(t_loss)

  nb_exemple = 0
  nb_correct = 0

  net.eval()

  if(epoch%5 == 0):   #Collecte de données
    with torch.inference_mode():
      for data in valloader:
        inputs, labels = data[0].to(device), data[1].to(device)
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        v_loss += loss.item()
        _, predicted = torch.max(outputs, 1)

        nb_exemple += labels.size(0)
        nb_correct += (predicted == labels).sum().item()
    v_loss  =  v_loss/len(valloader)
    v_accuracy = 100*(nb_correct /nb_exemple)
    v_accuracy_tab.append(v_accuracy)
    v_loss_tab.append(v_loss)
  else:
    v_accuracy_tab.append(v_accuracy)
    v_loss_tab.append(v_loss)

  epochs.append(epoch)

  print(f'Epoch    {epoch+1}:T_Accuracy    =    {t_accuracy:.2f}%    T_Loss:=
{(t_loss):.2f}')
  print(f'Epoch    {epoch+1}:V_Accuracy    =    {v_accuracy:.2f}%    V_Loss:=
{(v_loss):.2f}')


#loss_Brut = t_loss_tab
t_loss_temp= np.array(t_loss_tab)
t_accuracy_temp= np.array(t_accuracy_tab)

v_loss_temp= np.array(v_loss_tab)
v_accuracy_temp= np.array(v_accuracy_tab)
In [ ]:
# Graphique des données et enregistrement des données au format Excel.
stop_random()
print('Entrainement fini')
plt.plot(epochs, t_loss_temp, label='Training_Loss')
plt.plot(epochs, v_loss_temp, label='Validation_Loss')
plt.title('Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()

plt.title('Accuracy')
plt.plot(epochs, t_accuracy_temp, label='Training_Accuracy')
plt.plot(epochs, v_accuracy_temp, label='Validation_Accuracy')
```

```
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()

#Enregistrement
import csv
titre                                                                    =
['training_loss','validation_loss','training_accuracy','validation_accuracy']

tableaux_complet                                                         =
list(zip(t_loss_temp,v_loss_temp,t_accuracy_temp,v_accuracy_temp))
path_save_data  =  './Entrainement/'+  dataset  +  "/Res18#"+dossier
+"#"+str(nb_Epoch) +"#" +str(BNF)+".csv"
print(path_save_data)
with open(path_save_data, mode='w', newline='', encoding='utf-8') as fichier:
    writer = csv.writer(fichier)
    writer.writerow(titre)
    for i in range(len(t_loss_temp)):
        temp1 =f"{t_loss_temp[i]:.5f}".replace('.', ',')
        temp2 =f"{v_loss_temp[i]:.5f}".replace('.', ',')
        temp3 =f"{t_accuracy_temp[i]:.5f}".replace('.', ',')
        temp4 =f"{v_accuracy_temp[i]:.5f}".replace('.', ',')
        writer.writerow([temp1, temp2, temp3, temp4])
print("Fichier sauvegarder: "+ path_save_data)
In [ ]:
#Enregistrement du model
torch.save(net.state_dict(), model_Path)
In [ ]:
#Chargement du model
net.load_state_dict(torch.load(model_Path))
In [ ]:
# Partie test : va tester le modèle sur l'ensemble du dossier de test, et donner les
prédictions par classe ainsi qu'une matrice de confusion.
stop_random()
print("NB de classe: " + str(nb_Classe))
image_transform = transform
total = 0
nb_true_pred = 0
total_AC = 0
taille_Dossier= [0 for x in range(nb_Classe)]
prediction= [[0 for x in range(nb_Classe)] for y in range(nb_Classe)]
model = net
model.eval()
image_folder = dataset+'_Full/test/'
items = os.listdir(image_folder)
print(len(items))

for j in range(0,len(items)):
    image_folder = dataset+'_Full/test/'
    image_folder = image_folder + "[" + str(j) + "]"
    nb_true_pred = 0
    taille_Dossier[j] = len(os.listdir(image_folder))
    for filename in os.listdir(image_folder):
        if filename.endswith(".png"):#Avant Jpeg
            image_path = os.path.join(image_folder, filename)
            image = Image.open(image_path).convert('RGB')
            image_tensor = image_transform(image)
            image_tensor = image_tensor.to(device)
```

```
        image_tensor = image_tensor.unsqueeze(0)
        with torch.inference_mode():
          output = model(image_tensor)
          probabilities = F.softmax(output, dim=1)
        class_names = classes
        _, predicted_class = torch.max(output, 1)
        predicted_class_name = class_names[predicted_class.item()]
        confidence = torch.max(probabilities).item() * 100
        true_class=os.path.basename(os.path.dirname(image_path))
        temp = predicted_class_name
        temp = temp.replace('[','')
        temp = temp.replace(']','')

        prediction[j][int(temp)] =prediction[j][int(temp)] +1
        if(true_class == predicted_class_name):
          nb_true_pred = nb_true_pred +1
  print("Classe:"+ str(j) + f" % de bonne prédiction:" +
str((nb_true_pred/taille_Dossier[j])* 100) +"%" )
  total = total + taille_Dossier[j]
  total_AC = total_AC + nb_true_pred

print("Accuray général:" +str(( total_AC / total)*100)+"%")
for j in range(0,len(items)):
  for i in range (0,len(items)):
    prediction[j][i] = (prediction[j][i]/taille_Dossier[j])*100

matrice_C = np.array(prediction)
plt.figure(figsize=(6, 6))
sns.heatmap(matrice_C, annot=True, fmt='.2f', cmap='Blues')
plt.xlabel('Prédiction')
plt.ylabel('Vrai classe')
plt.title('Matrice')
image_Path = 'M_image_'+dataset+'/'+str(nb_Epoch) +"_" + "EPOCH"+"_" +
dossier + "RES18.png"
plt.savefig(image_Path)
plt.show()
In [ ]:
# Utilisé pour copier plus simplement les résultats dans un fichier Excel.
total2=0
for j in range(0,nb_Classe):
  total2 = total2 + prediction[j][j]
  temp =f"{prediction[j][j]:.2f}".replace('.', ',')
  print(temp)
temp =f"{(total_AC / total)*100:.2f}".replace('.', ',')
print(temp)
temp =f"{total2/nb_Classe:.2f}".replace('.', ',')
print(temp)
In [ ]:
```

12.4.2 SimCLR

```
# Chaque bloc possède un lien vers la source originale, formaté comme ceci : #Source: []

# Rappels avant de lancer le programme :
#   - bien vérifier le dataset et le dossier utilisé.
#   - bien vérifier les paramètres et les dossiers de sauvegarde.
#   - faire attention de ne pas lancer le modèle alors qu'il a déjà été entraîné.
#   - Si le premier epoch est très lent, c'est normal, il suffit d'attendre un peu que les données soient chargées.
```

```
# Cette cellule contient les imports nécessaires au bon fonctionnement du code.

import torch
import torchvision
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torchvision.transforms as transforms
from torchsummary import summary
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision import datasets, models

import os
import random
import copy
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image


import seaborn as sns
import pandas as pd

from sklearn.metrics import confusion_matrix
import pytorch_lightning as pl

from lightly.data import LightlyDataset
from lightly.loss import NTXentLoss
from lightly.models.modules.heads import MoCoProjectionHead
from lightly.models.utils import (
    batch_shuffle,
    batch_unshuffle,
    deactivate_requires_grad,
    update_momentum,
)
#Import spécifique à chaque model
from lightly.loss import NTXentLoss
from lightly.models.modules import SimCLRProjectionHead
from lightly.transforms.simclr_transform import SimCLRTransform
from lightly.transforms import SimCLRTransform, utils
import csv

torch.__version__
In [ ]:
# Fonction utilisée pendant le programme pour donner une seed aux différentes fonctions aléatoires.
# A mettre au début de chaque cellule.
# Source : [https://pytorch.org/docs/stable/notes/randomness.html]
def stop_random():
    seed=621
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    random.seed(seed)
    np.random.seed(seed)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False

In [ ]:
# Configuration générale. Attention à bien choisir les bons paramètres et les bon emplacement de sauvegarde.

stop_random()
nb_Epoch = 100
batch_Size = 128
taille_Image = 28
augmentation=False
```

```
num_workers = 4
dossier="Full"
dataset="Blood"#Derma
BNF =True
#Nomenclature:Nom_Model#Dossier#Epoch#BNF
Nom_Model = "SimCLR"
model_Path = './PytorchModelV3/'+"SimCLR_Model#"+"Full" +"#"+dataset+str(nb_Epoch) +"#"
+"FALSE"+".pth"
path_Class = './PytorchModelV3/'+"SimCLR_Class#"+dossier +"#"+dataset+str(nb_Epoch) +"#"
+str(BNF)+".pth"
path_Train_SSL_M = dataset+"_SSL"
path_Train = dataset+"_"+dossier+"/Dataset/train"
path_val = dataset+"_"+dossier+"/Dataset/val"
nb_Classe = len(next(os.walk(dataset+"_"+dossier+"/test"))[1])
print("Model choisi SimCLR, avec le dossier: " +dossier+ ", avec "+ str(nb_Epoch) + " epoch")
print("Emplacement: " +model_Path)
print("Emplacement: " +path_Class)
In [ ]:
#Source:[https://docs.lightly.ai/self-supervised-
learning/tutorials/package/tutorial_moco_memory_bank.html]
#Source:[https://docs.lightly.ai/self-supervised-learning/examples/simclr.html]

stop_random()
transform = SimCLRTransform(input_size=28, gaussian_blur=0.0)


train_classifier_transforms = torchvision.transforms.Compose(
    [
        torchvision.transforms.ToTensor(),
    ])

if(augmentation):# Si l'on a besoin de faire de l'augmentation de données.
    print("augmentation:")
    train_classifier_transforms = transforms.Compose([
    transforms.RandomResizedCrop(size=28, scale=(0.3, 1.0), ratio=(1, 1)),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
])


val_transforms = torchvision.transforms.Compose(
    [
        torchvision.transforms.ToTensor(),
    ])



dataset_train_model = LightlyDataset(input_dir=path_Train_SSL_M, transform=transform)
dataset_train_classifier = LightlyDataset(input_dir=path_Train, transform=train_classifier_transforms)
dataset_val = LightlyDataset(input_dir=path_val, transform=val_transforms)

print(train_classifier_transforms)
In [ ]:
#Source:[https://docs.lightly.ai/self-supervised-
learning/tutorials/package/tutorial_moco_memory_bank.html]

# Explication sur les paramètres des dataloaders :
#     shuffle --> mélange les images, à utiliser surtout pour les données d'entraînement du modèle SSL.
#     drop_last --> sinon, parfois, l'entraînement peut planter à un epoch.

stop_random()
dataloader_train_model = torch.utils.data.DataLoader(
    dataset_train_model,
    batch_size=batch_Size,
```

```
    shuffle=True,
    drop_last=True,
    num_workers=num_workers,
    persistent_workers=True,
)

dataloader_train_classifier = torch.utils.data.DataLoader(
    dataset_train_classifier,
    batch_size=batch_Size,
    shuffle=True,
    drop_last=True,
    num_workers=num_workers,
    persistent_workers=True,
)

dataloader_val = torch.utils.data.DataLoader(
    dataset_val,
    batch_size=batch_Size,
    shuffle=True,
    drop_last=False,
    num_workers=num_workers,
    persistent_workers=True,
)
In [ ]:
#Source:[https://docs.lightly.ai/self-supervised-learning/examples/simclr.html]
stop_random()
class SimCLR(pl.LightningModule):
    def __init__(self):
        super().__init__()
        resnet = torchvision.models.resnet18()
        self.backbone = nn.Sequential(*list(resnet.children())[:-1])
        self.projection_head = SimCLRProjectionHead(512, 1024, 1024)
        self.criterion = NTXentLoss()
        self.train_losses = []

    def forward(self, x):
        x = self.backbone(x).flatten(start_dim=1)
        z = self.projection_head(x)
        return z

    def training_step(self, batch, batch_index):
        (x0, x1) = batch[0]
        z0 = self.forward(x0)
        z1 = self.forward(x1)
        loss = self.criterion(z0, z1)
        self.log('train_loss', loss, on_step=True, on_epoch=True, prog_bar=False, logger=False)
        return loss

    def configure_optimizers(self):
        optim = torch.optim.SGD(self.parameters(), lr=0.06)
        return optim

    def on_train_epoch_end(self):
        epoch_loss = self.trainer.callback_metrics['train_loss']
        self.train_losses.append(epoch_loss.item())
        print(f'Epoch {self.current_epoch}: Train Loss: {epoch_loss.item()}')

    def on_train_end(self):# Affiche la loss à la fin de l'entraînement.
        print("LOSS:")
        for i in self.train_losses:
            temp =f"{i:.5f}".replace('.', ',')
            print(temp )
In [ ]:
#Source:[https://saturncloud.io/blog/how-to-use-class-weights-with-focal-loss-in-pytorch-for-imbalanced-
multiclass-classification/]
```

```
stop_random()
import torch.optim as optim
trainset = dataset_train_classifier
# Parce que les dataloaders de Lightly n'ont pas l'attribut "targets", donc on est obligé d'utiliser cette méthode.

nb_Element = [0] * nb_Classe
i = 0
total = 0
for sous_d in os.listdir(path_Train):
    sous_d = os.path.join(path_Train, sous_d)
    nb_Element[i] = len(os.listdir(sous_d))
    i = i +1
for k in range(0, nb_Classe):
    total = total + nb_Element[k]

class_weights = []
for exemple in nb_Element:
    poid = (total-exemple) / (total)
    class_weights.append(poid)
    print(poid)


#Source:[https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html]

class_weights = torch.tensor(class_weights,dtype=torch.float32).cuda()
criterion_ = nn.CrossEntropyLoss(weight=class_weights)
print(class_weights)
In [ ]:
#Source:[https://docs.lightly.ai/self-supervised-
learning/tutorials/package/tutorial_moco_memory_bank.html]
class Classifier(pl.LightningModule):
    def __init__(self, backbone):
        super().__init__()
        self.backbone = backbone


        self.train_losses_tab = []
        self.train_accuracy_tab = []
        self.val_losses_tab = []
        self.val_accuracy_tab = []

        self.train_losses = 0
        self.train_accuracy = 0
        self.val_losses = 0
        self.val_accuracy = 0

        self.t_iteration = 0
        self.v_iteration = 0

        # freeze the backbone
        if(not(BNF)):
            deactivate_requires_grad(backbone)
        self.fc1 = nn.Linear(512, 2048)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(2048, nb_Classe)

        self.validation_step_outputs = []
        self.criterion = criterion_

    def forward(self, x):
        x = self.backbone(x).flatten(start_dim=1)
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        y_hat = x
```

```
        return y_hat

    def training_step(self, batch, batch_idx):
        x, y, _ = batch
        y_hat = self.forward(x)
        loss =  self.criterion(y_hat, y)

        # Calcul et enregistrement des données utiles.
        t_acc = (y_hat.argmax(dim=1) == y).float().mean()
        self.train_losses = self.train_losses + loss.item()
        self.train_accuracy = self.train_accuracy + t_acc.item()
        self.t_iteration =  self.t_iteration +1
        return loss

    def on_train_epoch_end(self):

        self.train_losses_tab.append(self.train_losses/self.t_iteration)
        self.train_accuracy_tab.append(self.train_accuracy/self.t_iteration)
        self.train_losses = 0
        self.train_accuracy = 0
        self.t_iteration = 0


    def on_train_end(self):

        self.train_losses= np.array(self.train_losses_tab)
        self.train_accuracy= np.array(self.train_accuracy_tab)

        self.val_losses= np.array(self.val_losses_tab)
        self.val_accuracy= np.array(self.val_accuracy_tab)

        #Enregistrement
        titre = ['training_loss','validation_loss','training_accuracy','validation_accuracy']

        tableaux_complet = list(zip(self.train_losses,self.train_accuracy,self.val_losses,self.val_accuracy))
        path_save_data = './Entrainement/'+ dataset + "/SimCLR_Class#"+dossier +"#"+str(nb_Epoch) +"#"
+str(BNF)+".csv"
        print(path_save_data)
        with open(path_save_data, mode='w', newline='', encoding='utf-8') as fichier:
            writer = csv.writer(fichier)
            writer.writerow(titre)
            for i in range(len(self.train_losses)):
                temp1 =f"{self.train_losses[i]:.5f}".replace('.', ',')
                temp2 =f"{self.val_losses[i]:.5f}".replace('.', ',')
                temp3 =f"{self.train_accuracy[i]:.5f}".replace('.', ',')
                temp4 =f"{self.val_accuracy[i]:.5f}".replace('.', ',')
                writer.writerow([temp1, temp2, temp3, temp4])

    def custom_histogram_weights(self):
        for name, params in self.named_parameters():
            self.logger.experiment.add_histogram(name, params, self.current_epoch)

    def validation_step(self, batch, batch_idx):
        x, y, _ = batch
        y_hat = self.forward(x)
        loss =  self.criterion(y_hat, y)
        v_acc = (y_hat.argmax(dim=1) == y).float().mean()

        self.val_losses =self.val_losses + loss.item()
        self.val_accuracy = self.val_accuracy + v_acc.item()
        self.v_iteration = self.v_iteration +1

    def on_validation_epoch_end(self):

        self.val_losses_tab.append(self.val_losses/self.v_iteration)
```

```
        self.val_accuracy_tab.append(self.val_accuracy/self.v_iteration)

        self.val_losses = 0
        self.val_accuracy = 0
        self.v_iteration = 0

    def configure_optimizers(self):
        optim = torch.optim.SGD(self.parameters(), lr=0.01, momentum=0.9)
        scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optim, T_max=self.trainer.max_epochs)
        return [optim], [scheduler]
In [ ]:
torch.set_float32_matmul_precision('medium')# Réduit légèrement la précision, mais améliore la rapidité de
l'entraînement.
In [ ]:
#Entrainement du model
stop_random()
model = SimCLR()
print("Entrainement du model: " + Nom_Model)
trainer = pl.Trainer(max_epochs=nb_Epoch, devices=1, accelerator="gpu")
trainer.fit(model, dataloader_train_model)

torch.save(model.state_dict(), model_Path)
In [ ]:
#Chargement du model
model = SimCLR()
model.load_state_dict(torch.load(model_Path))
print(model_Path)
In [ ]:
#Source:[https://docs.lightly.ai/self-supervised-
learning/tutorials/package/tutorial_moco_memory_bank.html]
stop_random()
import contextlib
import os
import sys

print("Entrainement du classifieur pour le model: " + Nom_Model)
if(not(BNF)):
    model.eval()
    print("Eval")
else:
    model.train()
    print("Train")

log_file_path = 'training_output.log'# PyTorch n'aime pas afficher beaucoup de texte, donc on le stocke dans
un fichier à la place.
with open(log_file_path, 'w') as log_file_path:
    with contextlib.redirect_stdout(log_file_path), contextlib.redirect_stderr(log_file_path):
        classifier = Classifier(model.backbone)
        trainer = pl.Trainer(max_epochs=nb_Epoch, devices=1, accelerator="gpu")
        trainer.fit(classifier, dataloader_train_classifier, dataloader_val)

print("FINITO")


#Sauvegarde le classificateur
torch.save(classifier.state_dict(), path_Class)
In [ ]:
#Charge le classificateur
classi =  Classifier(model.backbone)
classi.load_state_dict(torch.load(path_Class))
classifier= classi
In [ ]:
stop_random()
# Partie test : va tester le modèle sur l'ensemble du dossier de test, et donner les prédictions par classe ainsi
qu'une matrice de confusion.
```

```
import torch.nn.functional as F
import seaborn as sns
import pandas as pd
from sklearn.metrics import confusion_matrix


print("Model choisi:" + Nom_Model +",Avec "+ str(nb_Epoch)+" Epoch,et le dossier " +dossier)
print(model_Path)
print("NB de classe: " + str(nb_Classe))


class_names=['[0]', '[1]', '[2]', '[3]', '[4]', '[5]', '[6]' ,'[7]']
if(nb_Classe == 8):
   class_names=['[0]', '[1]', '[2]', '[3]', '[4]', '[5]', '[6]' ,'[7]']
else:
   class_names=['[0]', '[1]', '[2]', '[3]', '[4]', '[5]', '[6]']
total = 0
nb_true_pred = 0
total_AC = 0
taille_dossier= [0 for x in range(nb_Classe)]

predictions= [[0 for x in range(nb_Classe)] for y in range(nb_Classe)]


for j in range(0,nb_Classe):
   image_folder = dataset +"_Full/test/"

   image_folder = image_folder + "[" + str(j) + "]"
   nb_true_pred = 0
   taille_dossier[j] = len(os.listdir(image_folder))
   classifier.eval()


   for filename in os.listdir(image_folder):
      if filename.endswith(".png"):
         image_path = os.path.join(image_folder, filename)

         image = Image.open(image_path).convert('RGB')
         image_tensor = val_transforms(image)
         image_tensor = image_tensor.unsqueeze(0)

         with torch.no_grad():

            output = classifier(image_tensor)
            probabilities = F.softmax(output, dim=1)

            _, predicted_class = torch.max(output, 1)
            predicted_class_name = class_names[predicted_class.item()]
            confidence = torch.max(probabilities).item() * 100
            true_class=os.path.basename(os.path.dirname(image_path))
            temp = predicted_class_name
            temp = temp.replace('[','')
            temp = temp.replace(']','')
            predictions[j][int(temp)] = predictions[j][int(temp)] +1
            if(true_class == predicted_class_name):
               nb_true_pred = nb_true_pred +1
   print("Classe:"+ str(j) + f" % de bonne prédiction:" + str((nb_true_pred/taille_dossier[j])* 100) +"%" )
   total = total + taille_dossier[j]
   total_AC = total_AC + nb_true_pred

print("Accuray général:" +str(( total_AC / total)*100)+"%")

for j in range(0,nb_Classe):
   for i in range (0,nb_Classe):
      predictions[j][i] = (predictions[j][i]/taille_dossier[j])*100
```

```
matrice_C = np.array(predictions)
plt.figure(figsize=(6, 6))
sns.heatmap(matrice_C, annot=True, fmt='.2f', cmap='Blues')
plt.xlabel('Prédiction')
plt.ylabel('Vrai classe')
plt.title('Matrice')
image_Path = 'M_image_'+dataset+'/'+str(nb_Epoch) +"_" + "EPOCH"+"_" + dossier +"_"+
Nom_Model+"_"+str(BNF)+".png"
plt.savefig(image_Path)
plt.show()
In [ ]:
# Utilisé pour copier plus simplement les résultats dans un fichier Excel.
total2=0
for j in range(0,nb_Classe):
    total2 = total2 + predictions[j][j]
    temp =f"{predictions[j][j]:.2f}".replace('.', ',')
    print(temp)
temp =f"{(total_AC / total)*100:.2f}".replace('.', ',')

print(temp)
temp =f"{total2/nb_Classe:.2f}".replace('.', ',')
print(temp)
```

12.4.3 Moco

```
# Chaque bloc possède un lien vers la source originale, formaté comme ceci : #Source: []

# Rappels avant de lancer le programme :
#   - bien vérifier le dataset et le dossier utilisé.
#   - bien vérifier les paramètres et les dossiers de sauvegarde.
#   - faire attention de ne pas lancer le modèle alors qu'il a déjà été entraîné.
#   - Si le premier epoch est très lent, c'est normal, il suffit d'attendre un peu que les données soient chargées.

# Cette cellule contient les imports nécessaires au bon fonctionnement du code.

import torch
import torchvision
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torchvision.transforms as transforms
from torchsummary import summary
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision import datasets, models

import os
import random
import copy
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image


import seaborn as sns
import pandas as pd

from sklearn.metrics import confusion_matrix
import pytorch_lightning as pl

from lightly.data import LightlyDataset
from lightly.loss import NTXentLoss
from lightly.models.modules.heads import MoCoProjectionHead
from lightly.models.utils import (
```

```
    batch_shuffle,
    batch_unshuffle,
    deactivate_requires_grad,
    update_momentum,
)
#Import spécifique à chaque model
from lightly.loss import NTXentLoss
from lightly.models.modules import MoCoProjectionHead
from lightly.models.utils import deactivate_requires_grad, update_momentum
from lightly.transforms.moco_transform import MoCoV2Transform
from lightly.utils.scheduler import cosine_schedule


torch.__version__
In [ ]:
# Fonction utilisée pendant le programme pour donner une seed aux différentes fonctions aléatoires.
# A mettre au début de chaque cellule.
# Source : [https://pytorch.org/docs/stable/notes/randomness.html]
def stop_random():
    seed=621
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    random.seed(seed)
    np.random.seed(seed)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False

In [ ]:
# Configuration générale. Attention à bien choisir les bons paramètres et les bon emplacement de sauvegarde.
stop_random()
nb_Epoch = 100
batch_Size = 128
taille_Image = 28
augmentation=False
num_workers = 4
dossier="Full"
dataset="Blood"#Derma
BNF = False
#Nomenclature:Nom_Model#Dossier#Epoch#BNF
Nom_Model = "MoCo"
model_Path = './PytorchModelV3/'+"MoCo_Model#"+"Full" +"#"+dataset+str(nb_Epoch) +"#"
+"FALSE"+".pth"
path_Class = './PytorchModelV3/'+"MoCo_Class#"+dossier +"#"+dataset+str(nb_Epoch) +"#"
+str(BNF)+".pth"
path_Train_SSL_M = dataset+"_SSL"
path_Train = dataset+"_"+dossier+"/Dataset/train"
path_val =  dataset+"_"+dossier+"/Dataset/val"
nb_Classe = len(next(os.walk(dataset+"_"+dossier+"/test"))[1])
print("Model choisi MoCo, avec le dossier: " +dossier+ ", avec "+ str(nb_Epoch) + " epoch")
print("Emplacement: " +model_Path)
print("Emplacement: " +path_Class)
In [ ]:
#Source:[https://docs.lightly.ai/self-supervised-
learning/tutorials/package/tutorial_moco_memory_bank.html]
#Source:[https://docs.lightly.ai/self-supervised-learning/examples/moco.html]

transform = MoCoV2Transform(input_size=28,gaussian_blur=0.0,)

train_classifier_transforms = torchvision.transforms.Compose(
    [
        torchvision.transforms.ToTensor(),
    ])

if(augmentation):# Si l'on a besoin de faire de l'augmentation de données.
    print("augmentation:")
```

```
    train_classifier_transforms = transforms.Compose([
    transforms.RandomResizedCrop(size=28, scale=(0.3, 1.0), ratio=(1, 1)),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
])


val_transforms = torchvision.transforms.Compose(
    [
        torchvision.transforms.ToTensor(),
    ])



dataset_train_model = LightlyDataset(input_dir=path_Train_SSL_M, transform=transform)
dataset_train_classifier = LightlyDataset(input_dir=path_Train, transform=train_classifier_transforms)
dataset_val = LightlyDataset(input_dir=path_val, transform=val_transforms)

print(train_classifier_transforms)
In [ ]:
#Source:[https://docs.lightly.ai/self-supervised-
learning/tutorials/package/tutorial_moco_memory_bank.html]

# Explication sur les paramètres des dataloaders :
#     shuffle --> mélange les images, à utiliser surtout pour les données d'entraînement du modèle SSL.
#     drop_last --> sinon, parfois, l'entraînement peut planter à un epoch.

stop_random()
dataloader_train_model = torch.utils.data.DataLoader(
    dataset_train_model,
    batch_size=batch_Size,
    shuffle=True,
    drop_last=True,
    num_workers=num_workers,
    persistent_workers=True,
)

dataloader_train_classifier = torch.utils.data.DataLoader(
    dataset_train_classifier,
    batch_size=batch_Size,
    shuffle=True,
    drop_last=True,
    num_workers=num_workers,
    persistent_workers=True,
)

dataloader_val = torch.utils.data.DataLoader(
    dataset_val,
    batch_size=batch_Size,
    shuffle=False,
    drop_last=False,
    num_workers=num_workers,
    persistent_workers=True,
)
In [ ]:
#Source:[https://docs.lightly.ai/self-supervised-learning/examples/moco.html]
stop_random()
class Moco(pl.LightningModule):
    def __init__(self):
        super().__init__()
        resnet = torchvision.models.resnet18()
        self.backbone = nn.Sequential(*list(resnet.children())[:-1])
        self.projection_head = MoCoProjectionHead(512, 1024, 1024)
```

```
        self.backbone_momentum = copy.deepcopy(self.backbone)
        self.projection_head_momentum = copy.deepcopy(self.projection_head)

        deactivate_requires_grad(self.backbone_momentum)
        deactivate_requires_grad(self.projection_head_momentum)

        self.criterion = NTXentLoss(memory_bank_size=(4096, 1024))
        self.train_losses = []

    def forward(self, x):
        query = self.backbone(x).flatten(start_dim=1)
        query = self.projection_head(query)
        return query

    def forward_momentum(self, x):
        key = self.backbone_momentum(x).flatten(start_dim=1)
        key = self.projection_head_momentum(key).detach()
        return key

    def training_step(self, batch, batch_idx):
        momentum = cosine_schedule(self.current_epoch, nb_Epoch, 0.996, 1)
        update_momentum(self.backbone, self.backbone_momentum, m=momentum)
        update_momentum(self.projection_head, self.projection_head_momentum, m=momentum)
        x_query, x_key = batch[0]
        query = self.forward(x_query)
        key = self.forward_momentum(x_key)
        loss = self.criterion(query, key)
        self.log("train_loss_ssl", loss)
        self.log('train_loss', loss, on_step=True, on_epoch=True, prog_bar=False, logger=False)
        return loss


    def configure_optimizers(self):
        optim = torch.optim.SGD(self.parameters(), lr=0.06)
        return optim

    def on_train_epoch_end(self):
        epoch_loss = self.trainer.callback_metrics['train_loss']
        self.train_losses.append(epoch_loss.item())
        print(f'Epoch {self.current_epoch}: Train Loss: {epoch_loss.item()}')

    def on_train_end(self):# Affiche la loss à la fin de l'entraînement.
        print("LOSS:")
        for i in self.train_losses:
            temp =f"{i:.5f}".replace('.', ',')
            print(temp )
In [ ]:
#Source:[https://saturncloud.io/blog/how-to-use-class-weights-with-focal-loss-in-pytorch-for-imbalanced-multiclass-classification/]
stop_random()
import torch.optim as optim
trainset = dataset_train_classifier
# Parce que les dataloaders de Lightly n'ont pas l'attribut "targets", donc on est obligé d'utiliser cette méthode.


nb_Element = [0] * nb_Classe
i = 0
total = 0
for sous_d in os.listdir(path_Train):
    sous_d = os.path.join(path_Train, sous_d)
    nb_Element[i] = len(os.listdir(sous_d))
    i = i +1
for k in range(0, nb_Classe):
    total = total + nb_Element[k]
```

```
class_weights = []
for exemple in nb_Element:
   poid = (total-exemple) / (total)
   class_weights.append(poid)
   print(poid)


#Source:[https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html]

class_weights = torch.tensor(class_weights,dtype=torch.float32).cuda()
criterion_ = nn.CrossEntropyLoss(weight=class_weights)
print(class_weights)
In [ ]:
#Source:[https://docs.lightly.ai/self-supervised-
learning/tutorials/package/tutorial_moco_memory_bank.html]
class Classifier(pl.LightningModule):
   def __init__(self, backbone):
      super().__init__()
      self.backbone = backbone

      self.train_losses_tab = []
      self.train_accuracy_tab = []
      self.val_losses_tab = []
      self.val_accuracy_tab = []

      self.train_losses = 0
      self.train_accuracy = 0
      self.val_losses = 0
      self.val_accuracy = 0

      self.t_iteration = 0
      self.v_iteration = 0

      # freeze the backbone
      if(not(BNF)):
         deactivate_requires_grad(backbone)
      self.fc1 = nn.Linear(512, 2048)
      self.relu = nn.ReLU()
      self.fc2 = nn.Linear(2048, nb_Classe)

      self.validation_step_outputs = []
      self.criterion = criterion_

   def forward(self, x):
      x = self.backbone(x).flatten(start_dim=1)
      x = self.fc1(x)
      x = self.relu(x)
      x = self.fc2(x)
      y_hat = x
      return y_hat

   def training_step(self, batch, batch_idx):
      x, y, _ = batch
      y_hat = self.forward(x)
      loss =  self.criterion(y_hat, y)
      # Calcul et enregistrement des données utiles.

      t_acc = (y_hat.argmax(dim=1) == y).float().mean()
      self.train_losses = self.train_losses + loss.item()
      self.train_accuracy = self.train_accuracy + t_acc.item()
      self.t_iteration =  self.t_iteration +1
      return loss

   def on_train_epoch_end(self):
```

```
        self.train_losses_tab.append(self.train_losses/self.t_iteration)
        self.train_accuracy_tab.append(self.train_accuracy/self.t_iteration)
        self.train_losses = 0
        self.train_accuracy = 0
        self.t_iteration = 0


    def on_train_end(self):

        self.train_losses= np.array(self.train_losses_tab)
        self.train_accuracy= np.array(self.train_accuracy_tab)

        self.val_losses= np.array(self.val_losses_tab)
        self.val_accuracy= np.array(self.val_accuracy_tab)

        #Enregistrement
        import csv
        titre = ['training_loss','validation_loss','training_accuracy','validation_accuracy']

        tableaux_complet = list(zip(self.train_losses,self.train_accuracy,self.val_losses,self.val_accuracy))
        path_save_data = './Entrainement/'+ dataset + "/Moco_Class#"+dossier +"#"+str(nb_Epoch) +"#"
+str(BNF)+".csv"
        print(path_save_data)
        with open(path_save_data, mode='w', newline='', encoding='utf-8') as fichier:
            writer = csv.writer(fichier)
            writer.writerow(titre)
            for i in range(len(self.train_losses)):
                temp1 =f"{self.train_losses[i]:.5f}".replace('.', ',')
                temp2 =f"{self.val_losses[i]:.5f}".replace('.', ',')
                temp3 =f"{self.train_accuracy[i]:.5f}".replace('.', ',')
                temp4 =f"{self.val_accuracy[i]:.5f}".replace('.', ',')
                writer.writerow([temp1, temp2, temp3, temp4])

    def custom_histogram_weights(self):
        for name, params in self.named_parameters():
            self.logger.experiment.add_histogram(name, params, self.current_epoch)

    def validation_step(self, batch, batch_idx):
        x, y, _ = batch
        y_hat = self.forward(x)
        loss =  self.criterion(y_hat, y)
        v_acc = (y_hat.argmax(dim=1) == y).float().mean()

        self.val_losses =self.val_losses + loss.item()
        self.val_accuracy = self.val_accuracy + v_acc.item()
        self.v_iteration = self.v_iteration +1

    def on_validation_epoch_end(self):

        self.val_losses_tab.append(self.val_losses/self.v_iteration)
        self.val_accuracy_tab.append(self.val_accuracy/self.v_iteration)

        self.val_losses = 0
        self.val_accuracy = 0
        self.v_iteration = 0

    def configure_optimizers(self):
        optim = torch.optim.SGD(self.parameters(), lr=0.01, momentum=0.9)
        scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optim, T_max=self.trainer.max_epochs)
        return [optim], [scheduler]
In [ ]:
torch.set_float32_matmul_precision('medium')# Réduit légèrement la précision, mais améliore la rapidité de
l'entraînement.
In [ ]:
#Entrainement du model
```

```
stop_random()
model = Moco()
print("Entrainement du model: " + Nom_Model)
trainer = pl.Trainer(max_epochs=nb_Epoch, devices=1, accelerator="gpu")
trainer.fit(model, dataloader_train_model)

torch.save(model.state_dict(), model_Path)
In [ ]:
#Chargement du model
model = Moco()
model.load_state_dict(torch.load(model_Path))
print(model_Path)
In [ ]:
#Source:[https://docs.lightly.ai/self-supervised-
learning/tutorials/package/tutorial_moco_memory_bank.html]
stop_random()
import contextlib
import os
import sys

print("Entrainement du classifieur pour le model: " + Nom_Model)

if(not(BNF)):
   model.eval()
   print("Eval")
else:
   model.train()
   print("Train")

log_file_path = 'training_output.log'# PyTorch n'aime pas afficher beaucoup de texte, donc on le stocke dans
un fichier à la place.
with open(log_file_path, 'w') as log_file_path:
   with contextlib.redirect_stdout(log_file_path), contextlib.redirect_stderr(log_file_path):
      classifier = Classifier(model.backbone)
      trainer = pl.Trainer(max_epochs=nb_Epoch, devices=1, accelerator="gpu")
      trainer.fit(classifier, dataloader_train_classifier, dataloader_val)

print("FINITO")


#Sauvegarde le classificateur
torch.save(classifier.state_dict(), path_Class)
In [ ]:
#Charge le classificateur
classi =  Classifier(model.backbone)
classi.load_state_dict(torch.load(path_Class))
classifier= classi
In [ ]:
stop_random()
# Partie test : va tester le modèle sur l'ensemble du dossier de test, et donner les prédictions par classe ainsi
qu'une matrice de confusion.
import torch.nn.functional as F
import seaborn as sns
import pandas as pd
from sklearn.metrics import confusion_matrix


print("Model choisi:" + Nom_Model +",Avec "+ str(nb_Epoch)+" Epoch,et le dossier " +dossier)
print(model_Path)
print("NB de classe: " + str(nb_Classe))


class_names=['[0]', '[1]', '[2]', '[3]', '[4]', '[5]', '[6]' ,'[7]']
if(nb_Classe == 8):
   class_names=['[0]', '[1]', '[2]', '[3]', '[4]', '[5]', '[6]' ,'[7]']
```

```
else:
   class_names=['[0]', '[1]', '[2]', '[3]', '[4]', '[5]', '[6]']

total = 0
nb_true_pred = 0
total_AC = 0
taille_dossier= [0 for x in range(nb_Classe)]

predictions= [[0 for x in range(nb_Classe)] for y in range(nb_Classe)]


for j in range(0,nb_Classe):
   image_folder = dataset +"_Full/test/"

   image_folder = image_folder + "[" + str(j) + "]"
   nb_true_pred = 0
   taille_dossier[j] = len(os.listdir(image_folder))
   classifier.eval()

   for filename in os.listdir(image_folder):
      if filename.endswith(".png"):
         image_path = os.path.join(image_folder, filename)

         image = Image.open(image_path).convert('RGB')
         image_tensor = val_transforms(image)
         image_tensor = image_tensor.unsqueeze(0)

         with torch.no_grad():

            output = classifier(image_tensor)
            probabilities = F.softmax(output, dim=1)

            _, predicted_class = torch.max(output, 1)
            predicted_class_name = class_names[predicted_class.item()]

            confidence = torch.max(probabilities).item() * 100
            true_class=os.path.basename(os.path.dirname(image_path))
            temp = predicted_class_name
            temp = temp.replace('[','')
            temp = temp.replace(']','')
            predictions[j][int(temp)] = predictions[j][int(temp)] +1
            if(true_class == predicted_class_name):
               nb_true_pred = nb_true_pred +1
   print("Classe:"+ str(j) + f" % de bonne prédiction:" + str((nb_true_pred/taille_dossier[j])* 100) +"%" )
   total = total + taille_dossier[j]
   total_AC = total_AC + nb_true_pred

print("Accuray général:" +str(( total_AC / total)*100)+"%")

for j in range(0,nb_Classe):
   for i in range (0,nb_Classe):
      predictions[j][i] = (predictions[j][i]/taille_dossier[j])*100
matrice_C = np.array(predictions)
plt.figure(figsize=(6, 6))
sns.heatmap(matrice_C, annot=True, fmt='.2f', cmap='Blues')
plt.xlabel('Prédiction')
plt.ylabel('Vrai classe')
plt.title('Matrice')
image_Path = 'M_image_'+dataset+'/'+str(nb_Epoch) +"_" + "EPOCH"+"_" + dossier +"_"+
Nom_Model+"_"+str(BNF)+".png"
print(image_Path)
plt.savefig(image_Path)
plt.show()
In [ ]:
# Utilisé pour copier plus simplement les résultats dans un fichier Excel.
```

```
total2=0
for j in range(0,nb_Classe):
    total2 = total2 + predictions[j][j]
    temp =f"{predictions[j][j]:.2f}".replace('.', ',')
    print(temp)
temp =f"{(total_AC / total)*100:.2f}".replace('.', ',')

print(temp)
temp =f"{total2/nb_Classe:.2f}".replace('.', ',')
print(temp)
```

12.4.4 MAE

```
# Chaque bloc possède un lien vers la source originale, formaté comme ceci : #Source: []

# Rappels avant de lancer le programme :
#   - bien vérifier le dataset et le dossier utilisé.
#   - bien vérifier les paramètres et les dossiers de sauvegarde.
#   - faire attention de ne pas lancer le modèle alors qu'il a déjà été entraîné.
#   - Si le premier epoch est très lent, c'est normal, il suffit d'attendre un peu que les données soient chargées.

# Cette cellule contient les imports nécessaires au bon fonctionnement du code.

import torch
import torchvision
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torchvision.transforms as transforms
from torchsummary import summary
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision import datasets, models

import os
import random
import copy
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image


import seaborn as sns
import pandas as pd

from sklearn.metrics import confusion_matrix
import pytorch_lightning as pl

from lightly.data import LightlyDataset
from lightly.loss import NTXentLoss
from lightly.models.modules.heads import MoCoProjectionHead
from lightly.models.utils import (
    batch_shuffle,
    batch_unshuffle,
    deactivate_requires_grad,
    update_momentum,
)
#Import spécifique à chaque model

from lightly.models import utils
from lightly.models.modules import MAEDecoderTIMM, MaskedVisionTransformerTIMM
from lightly.transforms import MAETransform
```

```
import csv

torch.__version__
In [ ]:
# Fonction utilisée pendant le programme pour donner une seed aux différentes fonctions aléatoires.
# A mettre au début de chaque cellule.
# Source : [https://pytorch.org/docs/stable/notes/randomness.html]
def stop_random():
  seed=621
  torch.manual_seed(seed)
  torch.cuda.manual_seed(seed)
  random.seed(seed)
  np.random.seed(seed)
  torch.backends.cudnn.deterministic = True
  torch.backends.cudnn.benchmark = False

In [ ]:
# Configuration générale. Attention à bien choisir les bons paramètres et les bon emplacement de sauvegarde.

stop_random()
nb_Epoch = 50
batch_Size = 128
taille_Image = 28
augmentation=False
num_workers = 4
dossier="Full"
dataset="Blood"#Derma
BNF = False
#Nomenclature:Nom_Model#Dossier#Epoch#BNF
Nom_Model = "MAE"
model_Path = './PytorchModelV3/'+"MAE_Model#"+"Full" +"#"+dataset+str(nb_Epoch) +"#"
+"FALSE"+".pth"
path_Class = './PytorchModelV3/'+"MAE_Class#"+dossier +"#"+dataset+str(nb_Epoch) +"#"
+str(BNF)+".pth"
path_Train_SSL_M = dataset+"_SSL"
path_Train = dataset+"_"+dossier+"/Dataset/train"
path_val = dataset+"_"+dossier+"/Dataset/val"
nb_Classe = len(next(os.walk(dataset+"_"+dossier+"/test"))[1])
print("Model choisi MAE, avec le dossier: " +dossier+ ", avec "+ str(nb_Epoch) + " epoch")
print("Emplacement: " +model_Path)
print("Emplacement: " +path_Class)
In [ ]:
#Source:[https://docs.lightly.ai/self-supervised-
learning/tutorials/package/tutorial_moco_memory_bank.html]
#Source:[https://docs.lightly.ai/self-supervised-learning/examples/mae.html]


from torchvision import datasets
from lightly.transforms import MAETransform


transform = MAETransform(input_size = 224)
input_Size=224

train_classifier_transforms = torchvision.transforms.Compose(
  [
    transforms.Resize(224),
    torchvision.transforms.ToTensor(),
  ])

if(augmentation):# Si l'on a besoin de faire de l'augmentation de données.
  print("augmentation:")
  train_classifier_transforms = transforms.Compose([
  transforms.RandomResizedCrop(size=28, scale=(0.3, 1.0), ratio=(1, 1)),
  transforms.RandomHorizontalFlip(),
```

```
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
])


val_transforms = torchvision.transforms.Compose(
    [
        transforms.Resize(224),
        torchvision.transforms.ToTensor(),
    ])



dataset_train_model = LightlyDataset(input_dir=path_Train_SSL_M, transform=transform)
dataset_train_classifier = LightlyDataset(input_dir=path_Train, transform=train_classifier_transforms)
dataset_val = LightlyDataset(input_dir=path_val, transform=val_transforms)

print(train_classifier_transforms)
In [ ]:
#Source:[https://docs.lightly.ai/self-supervised-
learning/tutorials/package/tutorial_moco_memory_bank.html]

# Explication sur les paramètres des dataloaders :
#      shuffle --> mélange les images, à utiliser surtout pour les données d'entraînement du modèle SSL.
#      drop_last --> sinon, parfois, l'entraînement peut planter à un epoch.


stop_random()
dataloader_train_model = torch.utils.data.DataLoader(
    dataset_train_model,
    batch_size=batch_Size,
    shuffle=True,
    drop_last=True,
    num_workers=num_workers,
    persistent_workers=True,#New
)

dataloader_train_classifier = torch.utils.data.DataLoader(
    dataset_train_classifier,
    batch_size=batch_Size,
    shuffle=True,
    drop_last=True,
    num_workers=num_workers,
    persistent_workers=True,
)

dataloader_val = torch.utils.data.DataLoader(
    dataset_val,
    batch_size=batch_Size,
    shuffle=False,
    drop_last=False,
    num_workers=num_workers,
    persistent_workers=True,
)
In [ ]:
#https://github.com/lucidrains/vit-pytorch/blob/main/README.md
#Model MAE
# Note: The model and training settings do not follow the reference settings
# from the paper. The settings are chosen such that the example can easily be
# run on a small dataset with a single GPU.

from vit_pytorch import ViT
import pytorch_lightning as pl
import torch
import torchvision
```

```
from timm.models.vision_transformer import vit_base_patch32_224
from torch import nn

from lightly.models import utils
from lightly.models.modules import MAEDecoderTIMM, MaskedVisionTransformerTIMM
from lightly.transforms import MAETransform

from timm import create_model


class MAE(pl.LightningModule):
  def __init__(self):
    super().__init__()
    self.train_losses = []
    num_classes = 8
    vit = vit_base_patch32_224()
    decoder_dim = 512

    self.mask_ratio = 0.75
    self.patch_size = vit.patch_embed.patch_size[0]
    self.backbone = MaskedVisionTransformerTIMM(vit=vit)
    self.sequence_length = self.backbone.sequence_length
    self.decoder = MAEDecoderTIMM(
      num_patches=vit.patch_embed.num_patches,
      patch_size=self.patch_size,
      embed_dim=vit.embed_dim,
      decoder_embed_dim=decoder_dim,
      in_chans=3,
      decoder_depth=1,
      decoder_num_heads=16,
      mlp_ratio=4.0,
      proj_drop_rate=0.0,
      attn_drop_rate=0.0,
    )
    self.criterion = nn.MSELoss()

  def forward_encoder(self, images, idx_keep=None):
    return self.backbone.encode(images=images, idx_keep=idx_keep)

  def forward_decoder(self, x_encoded, idx_keep, idx_mask):
    # build decoder input
    batch_size = x_encoded.shape[0]
    x_decode = self.decoder.embed(x_encoded)
    x_masked = utils.repeat_token(
      self.decoder.mask_token, (batch_size, self.sequence_length)
    )
    x_masked = utils.set_at_index(x_masked, idx_keep, x_decode.type_as(x_masked))

    # decoder forward pass
    x_decoded = self.decoder.decode(x_masked)

    # predict pixel values for masked tokens
    x_pred = utils.get_at_index(x_decoded, idx_mask)
    x_pred = self.decoder.predict(x_pred)
    return x_pred

  def training_step(self, batch, batch_idx):
    views = batch[0]
    images = views[0]  # views contains only a single view
    batch_size = images.shape[0]
    idx_keep, idx_mask = utils.random_token_mask(
      size=(batch_size, self.sequence_length),
      mask_ratio=self.mask_ratio,
      device=images.device,
    )
```

```
        x_encoded = self.forward_encoder(images=images, idx_keep=idx_keep)
        x_pred = self.forward_decoder(
            x_encoded=x_encoded, idx_keep=idx_keep, idx_mask=idx_mask
        )

        # get image patches for masked tokens
        patches = utils.patchify(images, self.patch_size)
        # must adjust idx_mask for missing class token
        target = utils.get_at_index(patches, idx_mask - 1)

        loss = self.criterion(x_pred, target)
        self.log('train_loss', loss, on_step=True, on_epoch=True, prog_bar=False, logger=False)#Log
        return loss

    def configure_optimizers(self):
        optim = torch.optim.AdamW(self.parameters(), lr=1.5e-4)
        return optim

    def on_train_epoch_end(self):
        epoch_loss = self.trainer.callback_metrics['train_loss']
        self.train_losses.append(epoch_loss.item())
        print(f'Epoch {self.current_epoch}: Train Loss: {epoch_loss.item()}')

    def on_train_end(self):# Affiche la loss à la fin de l'entraînement.
        print("LOSS:")
        for i in self.train_losses:
            temp =f"{i:.5f}".replace('.', ',')
            print(temp )
In [ ]:
#Source:[https://saturncloud.io/blog/how-to-use-class-weights-with-focal-loss-in-pytorch-for-imbalanced-multiclass-classification/]
stop_random()
import torch.optim as optim
trainset = dataset_train_classifier
# Parce que les dataloaders de Lightly n'ont pas l'attribut "targets", donc on est obligé d'utiliser cette méthode.


nb_Element = [0] * nb_Classe
i = 0
total = 0
for sous_d in os.listdir(path_Train):
    sous_d = os.path.join(path_Train, sous_d)
    nb_Element[i] = len(os.listdir(sous_d))
    i = i +1
for k in range(0, nb_Classe):
    total = total + nb_Element[k]

class_weights = []
for exemple in nb_Element:
    poid = (total-exemple) / (total)
    class_weights.append(poid)
    print(poid)


#Source:[https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html]

class_weights = torch.tensor(class_weights,dtype=torch.float32).cuda()
criterion_ = nn.CrossEntropyLoss(weight=class_weights)
print(class_weights)
In [ ]:
#Source:[https://docs.lightly.ai/self-supervised-learning/tutorials/package/tutorial_moco_memory_bank.html]
class Classifier(pl.LightningModule):
    def __init__(self, backbone):
        super().__init__()
```

```
        self.backbone = backbone


        self.train_losses_tab = []
        self.train_accuracy_tab = []
        self.val_losses_tab = []
        self.val_accuracy_tab = []

        self.train_losses = 0
        self.train_accuracy = 0
        self.val_losses = 0
        self.val_accuracy = 0

        self.t_iteration = 0
        self.v_iteration = 0

        # freeze the backbone
        if(not(BNF)):
            deactivate_requires_grad(backbone)
        self.fc1 = nn.Linear(768, 2048)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(2048, nb_Classe)

        self.validation_step_outputs = []
        self.criterion = criterion_

    def forward(self, x):
        x = self.backbone(x).flatten(start_dim=1)
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        y_hat = x
        return y_hat

    def training_step(self, batch, batch_idx):
        x, y, _ = batch
        y_hat = self.forward(x)
        loss =  self.criterion(y_hat, y)
        # Calcul et enregistrement des données utiles.

        t_acc = (y_hat.argmax(dim=1) == y).float().mean()
        self.train_losses = self.train_losses + loss.item()
        self.train_accuracy = self.train_accuracy + t_acc.item()
        self.t_iteration =  self.t_iteration +1
        return loss

    def on_train_epoch_end(self):

        self.train_losses_tab.append(self.train_losses/self.t_iteration)
        self.train_accuracy_tab.append(self.train_accuracy/self.t_iteration)
        self.train_losses = 0
        self.train_accuracy = 0
        self.t_iteration = 0


    def on_train_end(self):

        self.train_losses= np.array(self.train_losses_tab)
        self.train_accuracy= np.array(self.train_accuracy_tab)

        self.val_losses= np.array(self.val_losses_tab)
        self.val_accuracy= np.array(self.val_accuracy_tab)

        #Enregistrement
        titre = ['training_loss','validation_loss','training_accuracy','validation_accuracy']
```

```
        tableaux_complet = list(zip(self.train_losses,self.train_accuracy,self.val_losses,self.val_accuracy))
        path_save_data = './Entrainement/'+ dataset + "/MAE_Class#"+dossier +"#"+str(nb_Epoch) +"#"
+str(BNF)+".csv"
        print(path_save_data)
        with open(path_save_data, mode='w', newline='', encoding='utf-8') as fichier:
            writer = csv.writer(fichier)
            writer.writerow(titre)
            for i in range(len(self.train_losses)):
                temp1 =f"{self.train_losses[i]:.5f}".replace('.', ',')
                temp2 =f"{self.val_losses[i]:.5f}".replace('.', ',')
                temp3 =f"{self.train_accuracy[i]:.5f}".replace('.', ',')
                temp4 =f"{self.val_accuracy[i]:.5f}".replace('.', ',')
                writer.writerow([temp1, temp2, temp3, temp4])

    def custom_histogram_weights(self):
        for name, params in self.named_parameters():
            self.logger.experiment.add_histogram(name, params, self.current_epoch)

    def validation_step(self, batch, batch_idx):
        x, y, _ = batch
        y_hat = self.forward(x)
        loss =  self.criterion(y_hat, y)
        v_acc = (y_hat.argmax(dim=1) == y).float().mean()

        self.val_losses =self.val_losses + loss.item()
        self.val_accuracy = self.val_accuracy + v_acc.item()
        self.v_iteration = self.v_iteration +1

    def on_validation_epoch_end(self):

        self.val_losses_tab.append(self.val_losses/self.v_iteration)
        self.val_accuracy_tab.append(self.val_accuracy/self.v_iteration)

        self.val_losses = 0
        self.val_accuracy = 0
        self.v_iteration = 0

    def configure_optimizers(self):
        optim = torch.optim.SGD(self.parameters(), lr=0.01, momentum=0.9)
        scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optim, T_max=self.trainer.max_epochs)
        return [optim], [scheduler]
In [ ]:
torch.set_float32_matmul_precision('medium')# Réduit légèrement la précision, mais améliore la rapidité de
l'entraînement.
In [ ]:
#Entrainement du model
stop_random()

model = MAE()


print("Entrainement du model: " + Nom_Model)
trainer = pl.Trainer(max_epochs=nb_Epoch, devices=1, accelerator="gpu")
trainer.fit(model, dataloader_train_model)

torch.save(model.state_dict(), model_Path)
In [ ]:
#Chargement du model
model = MAE()
model.load_state_dict(torch.load(model_Path))
print(model_Path)
In [ ]:
#Source:[https://docs.lightly.ai/self-supervised-
learning/tutorials/package/tutorial_moco_memory_bank.html]
```

```
stop_random()
import contextlib
import os
import sys

print("Entrainement du classifieur pour le model: " + Nom_Model)

if(not(BNF)):
   model.eval()
   print("Eval")
else:
   model.train()
   print("Train")

log_file_path = 'training_output.log'# PyTorch n'aime pas afficher beaucoup de texte, donc on le stocke dans un fichier à la place.
with open(log_file_path, 'w') as log_file_path:
   with contextlib.redirect_stdout(log_file_path), contextlib.redirect_stderr(log_file_path):
      classifier = Classifier(model.backbone)
      trainer = pl.Trainer(max_epochs=nb_Epoch, devices=1, accelerator="gpu")
      trainer.fit(classifier, dataloader_train_classifier, dataloader_val)

print("FINITO")


#Sauvegarde le classificateur
torch.save(classifier.state_dict(), path_Class)
In [ ]:
#Charge le classificateur
classi =  Classifier(model.backbone)
classi.load_state_dict(torch.load(path_Class))
classifier= classi
In [ ]:
stop_random()
# Partie test : va tester le modèle sur l'ensemble du dossier de test, et donner les prédictions par classe ainsi qu'une matrice de confusion.
import torch.nn.functional as F
import seaborn as sns
import pandas as pd
from sklearn.metrics import confusion_matrix


print("Model choisi:" + Nom_Model +",Avec "+ str(nb_Epoch)+" Epoch,et le dossier " +dossier)
print(model_Path)
print("NB de classe: " + str(nb_Classe))


class_names=['[0]', '[1]', '[2]', '[3]', '[4]', '[5]', '[6]' ,'[7]']
if(nb_Classe == 8):
   class_names=['[0]', '[1]', '[2]', '[3]', '[4]', '[5]', '[6]' ,'[7]']
else:
   class_names=['[0]', '[1]', '[2]', '[3]', '[4]', '[5]', '[6]']

total = 0
nb_true_pred = 0
total_AC = 0
taille_dossier= [0 for x in range(nb_Classe)]

predictions= [[0 for x in range(nb_Classe)] for y in range(nb_Classe)]


for j in range(0,nb_Classe):
   image_folder = dataset +"_Full/test/"

   image_folder = image_folder + "[" + str(j) + "]"
```

```
    nb_true_pred = 0
    taille_dossier[j] = len(os.listdir(image_folder))
    classifier.eval()


    for filename in os.listdir(image_folder):
        if filename.endswith(".png"):
            image_path = os.path.join(image_folder, filename)

            image = Image.open(image_path).convert('RGB')
            image_tensor = val_transforms(image)
            image_tensor = image_tensor.unsqueeze(0)

            with torch.no_grad():

                output = classifier(image_tensor)
                probabilities = F.softmax(output, dim=1)

                _, predicted_class = torch.max(output, 1)
                predicted_class_name = class_names[predicted_class.item()]

                confidence = torch.max(probabilities).item() * 100
                true_class=os.path.basename(os.path.dirname(image_path))
                temp = predicted_class_name
                temp = temp.replace('[','')
                temp = temp.replace(']','')
                predictions[j][int(temp)] = predictions[j][int(temp)] +1
                if(true_class == predicted_class_name):
                    nb_true_pred = nb_true_pred +1
    print("Classe:"+ str(j) + f" % de bonne prédiction:" + str((nb_true_pred/taille_dossier[j])* 100) +"%" )
    total = total + taille_dossier[j]
    total_AC = total_AC + nb_true_pred

print("Accuray général:" +str(( total_AC / total)*100)+"%")

for j in range(0,nb_Classe):
    for i in range (0,nb_Classe):
        predictions[j][i] = (predictions[j][i]/taille_dossier[j])*100
matrice_C = np.array(predictions)
plt.figure(figsize=(6, 6))
sns.heatmap(matrice_C, annot=True, fmt='.2f', cmap='Blues')
plt.xlabel('Prédiction')
plt.ylabel('Vrai classe')
plt.title('Matrice')
image_Path = 'M_image_'+dataset+'/'+str(nb_Epoch) +"_" + "EPOCH"+"_" + dossier +"_"+
Nom_Model+"_"+str(BNF)+".png"
plt.savefig(image_Path)
plt.show()
In [ ]:
# Utilisé pour copier plus simplement les résultats dans un fichier Excel.
total2=0
for j in range(0,nb_Classe):
    total2 = total2 + predictions[j][j]
    temp =f"{predictions[j][j]:.2f}".replace('.', ',')
    print(temp)
temp =f"{(total_AC / total)*100:.2f}".replace('.', ',')

print(temp)
temp =f"{total2/nb_Classe:.2f}".replace('.', ',')
print(temp)
```

12.4.5 AIM

```
# Chaque bloc possède un lien vers la source originale, formaté comme ceci : #Source: []

# Rappels avant de lancer le programme :
#   - bien vérifier le dataset et le dossier utilisé.
#   - bien vérifier les paramètres et les dossiers de sauvegarde.
#   - faire attention de ne pas lancer le modèle alors qu'il a déjà été entraîné.
#   - Si le premier epoch est très lent, c'est normal, il suffit d'attendre un peu que les données soient chargées.

# Cette cellule contient les imports nécessaires au bon fonctionnement du code.

import torch
import torchvision
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torchvision.transforms as transforms
from torchsummary import summary
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision import datasets, models

import os
import random
import copy
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image


import seaborn as sns
import pandas as pd

from sklearn.metrics import confusion_matrix
import pytorch_lightning as pl

from lightly.data import LightlyDataset
from lightly.loss import NTXentLoss

from lightly.models.modules.heads import MoCoProjectionHead
from lightly.models.utils import (
    batch_shuffle,
    batch_unshuffle,
    deactivate_requires_grad,
    update_momentum,
)
#Import spécifique à chaque model
from lightly.models import utils
from lightly.models.modules import AIMPredictionHead, MaskedCausalVisionTransformer
from lightly.transforms import AIMTransform
import csv

torch.__version__
In [ ]:
# Fonction utilisée pendant le programme pour donner une seed aux différentes fonctions aléatoires.
# A mettre au début de chaque cellule.
# Source : [https://pytorch.org/docs/stable/notes/randomness.html]
def stop_random():
    seed=621
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    random.seed(seed)
    np.random.seed(seed)
    torch.backends.cudnn.deterministic = True
```

```
    torch.backends.cudnn.benchmark = False
```

In [ ]:

```
# Configuration générale. Attention à bien choisir les bons paramètres et les bon emplacement de sauvegarde.

stop_random()
nb_Epoch = 100
batch_Size = 128
taille_Image = 28
augmentation=False
num_workers = 4
dossier="Full"
dataset="Blood"#Derma
BNF = False
#Nomenclature:Nom_Model#Dossier#Epoch#BNF
Nom_Model = "AIM"
model_Path = './PytorchModelV3/'+"AIM_Model#"+"Full" +"#"+dataset+str(nb_Epoch) +"#"
+"FALSE"+".pth"
path_Class = './PytorchModelV3/'+"AIM_Class#"+dossier +"#"+dataset+str(nb_Epoch) +"#"
+str(BNF)+".pth"
path_Train_SSL_M = dataset+"_SSL"
path_Train = dataset+"_"+dossier+"/Dataset/train"
path_val =  dataset+"_"+dossier+"/Dataset/val"
nb_Classe = len(next(os.walk(dataset+"_"+dossier+"/test"))[1])
print("Model choisi AIM, avec le dossier: " +dossier+ ", avec "+ str(nb_Epoch) + " epoch")
print("Emplacement: " +model_Path)
print("Emplacement: " +path_Class)
```

In [ ]:

```
#Source:[https://docs.lightly.ai/self-supervised-
learning/tutorials/package/tutorial_moco_memory_bank.html]
#Source:[https://docs.lightly.ai/self-supervised-learning/examples/aim.html]

from torchvision import datasets
from lightly.transforms import AIMTransform

input_Size=28



transform = AIMTransform(input_size = 28)

train_classifier_transforms = torchvision.transforms.Compose(
    [
        torchvision.transforms.ToTensor(),
    ])

if(augmentation):# Si l'on a besoin de faire de l'augmentation de données.
    print("augmentation:")
    train_classifier_transforms = transforms.Compose([
    transforms.RandomResizedCrop(size=28, scale=(0.3, 1.0), ratio=(1, 1)),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
])


val_transforms = torchvision.transforms.Compose(
    [
        torchvision.transforms.ToTensor(),
    ])

dataset_train_model = LightlyDataset(input_dir=path_Train_SSL_M, transform=transform)
dataset_train_classifier = LightlyDataset(input_dir=path_Train, transform=train_classifier_transforms)
dataset_val = LightlyDataset(input_dir=path_val, transform=val_transforms)
```

```
print(train_classifier_transforms)
In [ ]:
#Source:[https://docs.lightly.ai/self-supervised-
learning/tutorials/package/tutorial_moco_memory_bank.html]

# Explication sur les paramètres des dataloaders :
#     shuffle --> mélange les images, à utiliser surtout pour les données d'entraînement du modèle SSL.
#     drop_last --> sinon, parfois, l'entraînement peut planter à un epoch.

stop_random()
dataloader_train_model = torch.utils.data.DataLoader(
    dataset_train_model,
    batch_size=batch_Size,
    shuffle=True,
    drop_last=True,
    num_workers=num_workers,
    persistent_workers=True,#New
)

dataloader_train_classifier = torch.utils.data.DataLoader(
    dataset_train_classifier,
    batch_size=batch_Size,
    shuffle=True,
    drop_last=True,
    num_workers=num_workers,
    persistent_workers=True,
)

dataloader_val = torch.utils.data.DataLoader(
    dataset_val,
    batch_size=batch_Size,
    shuffle=False,
    drop_last=False,
    num_workers=num_workers,
    persistent_workers=True,
)
In [ ]:
#Source:[https://docs.lightly.ai/self-supervised-learning/examples/aim.html]

import pytorch_lightning as pl
import torch
import torchvision
from torch import nn

from lightly.models import utils
from lightly.models.modules import AIMPredictionHead, MaskedCausalVisionTransformer
from lightly.transforms import AIMTransform

class AIM(pl.LightningModule):
    def __init__(self) -> None:
        super().__init__()
        self.train_losses = []
        vit = MaskedCausalVisionTransformer(
            img_size=28,
            patch_size=4,
            embed_dim=280,
            depth=12,
            num_heads=1,
            qk_norm=False,
            class_token=False,
            no_embed_class=True,
        )
        utils.initialize_2d_sine_cosine_positional_embedding(
            pos_embedding=vit.pos_embed, has_class_token=vit.has_class_token
        )
```

```
        self.patch_size = vit.patch_embed.patch_size[0]
        self.num_patches = vit.patch_embed.num_patches

        self.backbone = vit
        self.projection_head = AIMPredictionHead(

            input_dim=vit.embed_dim, output_dim=3 * self.patch_size**2, num_blocks=1
        )

        self.criterion = nn.MSELoss()

    def training_step(self, batch, batch_idx):
        views, targets = batch[0], batch[1]
        images = views[0]
        batch_size = images.shape[0]

        mask = utils.random_prefix_mask(
            size=(batch_size, self.num_patches),
            max_prefix_length=self.num_patches - 1,
            device=images.device,
        )
        features = self.backbone.forward_features(images, mask=mask)

        features = self.backbone._pos_embed(features)
        predictions = self.projection_head(features)


        patches = utils.patchify(images, self.patch_size)
        patches = utils.normalize_mean_var(patches, dim=-1)

        loss = self.criterion(predictions, patches)
        self.log('train_loss', loss, on_step=True, on_epoch=True, prog_bar=False, logger=False)#Log
        return loss

    def configure_optimizers(self):
        optim = torch.optim.AdamW(self.parameters(), lr=1.5e-4)
        return optim

    def on_train_epoch_end(self):
        epoch_loss = self.trainer.callback_metrics['train_loss']
        self.train_losses.append(epoch_loss.item())
        print(f'Epoch {self.current_epoch}: Train Loss: {epoch_loss.item()}')

    def on_train_end(self):# Affiche la loss à la fin de l'entraînement.
        print("LOSS:")
        for i in self.train_losses:
            temp =f"{i:.5f}".replace('.', ',')
            print(temp )
In [ ]:
#Source:[https://saturncloud.io/blog/how-to-use-class-weights-with-focal-loss-in-pytorch-for-imbalanced-multiclass-classification/]
stop_random()
import torch.optim as optim
trainset = dataset_train_classifier
#Car les dataloaders de lightly n'ont pas l'attribut targets, donc on est obligé d'utiliser cette méthode

nb_Element = [0] * nb_Classe
i = 0
total = 0
for sous_d in os.listdir(path_Train):
    sous_d = os.path.join(path_Train, sous_d)
    nb_Element[i] = len(os.listdir(sous_d))
    i = i +1
for k in range(0, nb_Classe):
    total = total + nb_Element[k]
```

```
class_weights = []
for exemple in nb_Element:
    poid = (total-exemple) / (total)
    class_weights.append(poid)
    print(poid)


#Source:[https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html]

class_weights = torch.tensor(class_weights,dtype=torch.float32).cuda()
criterion_ = nn.CrossEntropyLoss(weight=class_weights)
print(class_weights)
In [ ]:
#Source:[https://docs.lightly.ai/self-supervised-
learning/tutorials/package/tutorial_moco_memory_bank.html]
class Classifier(pl.LightningModule):
    def __init__(self, backbone):
        super().__init__()
        self.backbone = backbone


        self.train_losses_tab = []
        self.train_accuracy_tab = []
        self.val_losses_tab = []
        self.val_accuracy_tab = []

        self.train_losses = 0
        self.train_accuracy = 0
        self.val_losses = 0
        self.val_accuracy = 0

        self.t_iteration = 0
        self.v_iteration = 0

        self.fc0 = nn.Linear(49000, 512)
        deactivate_requires_grad(self.fc0)
        # freeze the backbone
        if(not(BNF)):
            deactivate_requires_grad(backbone)
        self.fc1 = nn.Linear(512, 2048)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(2048, nb_Classe)

        self.validation_step_outputs = []
        self.criterion = criterion_

    def forward(self, x):
        x = self.backbone(x).flatten(start_dim=1)
        x = self.fc0(x)
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        y_hat = x
        return y_hat

    def training_step(self, batch, batch_idx):
        x, y, _ = batch
        y_hat = self.forward(x)
        loss =  self.criterion(y_hat, y)

        # Calcul et enregistrement des données utiles.
        t_acc = (y_hat.argmax(dim=1) == y).float().mean()
        self.train_losses = self.train_losses + loss.item()
        self.train_accuracy = self.train_accuracy + t_acc.item()
```

```
        self.t_iteration =  self.t_iteration +1
        return loss

    def on_train_epoch_end(self):

        self.train_losses_tab.append(self.train_losses/self.t_iteration)
        self.train_accuracy_tab.append(self.train_accuracy/self.t_iteration)
        self.train_losses = 0
        self.train_accuracy = 0
        self.t_iteration = 0


    def on_train_end(self):

        self.train_losses= np.array(self.train_losses_tab)
        self.train_accuracy= np.array(self.train_accuracy_tab)

        self.val_losses= np.array(self.val_losses_tab)
        self.val_accuracy= np.array(self.val_accuracy_tab)

        #Enregistrement
        titre = ['training_loss','validation_loss','training_accuracy','validation_accuracy']

        tableaux_complet = list(zip(self.train_losses,self.train_accuracy,self.val_losses,self.val_accuracy))
        path_save_data = './Entrainement/'+ dataset + "/AIM_Class#"+dossier +"#"+str(nb_Epoch) +"#"
+str(BNF)+".csv"
        print(path_save_data)
        with open(path_save_data, mode='w', newline='', encoding='utf-8') as fichier:
            writer = csv.writer(fichier)
            writer.writerow(titre)
            for i in range(len(self.train_losses)):
                temp1 =f"{self.train_losses[i]:.5f}".replace('.', ',')
                temp2 =f"{self.val_losses[i]:.5f}".replace('.', ',')
                temp3 =f"{self.train_accuracy[i]:.5f}".replace('.', ',')
                temp4 =f"{self.val_accuracy[i]:.5f}".replace('.', ',')
                writer.writerow([temp1, temp2, temp3, temp4])

    def custom_histogram_weights(self):
        for name, params in self.named_parameters():
            self.logger.experiment.add_histogram(name, params, self.current_epoch)

    def validation_step(self, batch, batch_idx):
        x, y, _ = batch
        y_hat = self.forward(x)
        loss =  self.criterion(y_hat, y)
        v_acc = (y_hat.argmax(dim=1) == y).float().mean()

        self.val_losses =self.val_losses + loss.item()
        self.val_accuracy = self.val_accuracy + v_acc.item()
        self.v_iteration = self.v_iteration +1

    def on_validation_epoch_end(self):

        self.val_losses_tab.append(self.val_losses/self.v_iteration)
        self.val_accuracy_tab.append(self.val_accuracy/self.v_iteration)

        self.val_losses = 0
        self.val_accuracy = 0
        self.v_iteration = 0

    def configure_optimizers(self):
        optim = torch.optim.SGD(self.parameters(), lr=0.01, momentum=0.9)
        scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optim, T_max=self.trainer.max_epochs)
        return [optim], [scheduler]
In [ ]:
```

```
torch.set_float32_matmul_precision('medium')# Réduit légèrement la précision, mais améliore la rapidité de l'entraînement.
In [ ]:
#Entrainement du model
stop_random()

model = AIM()


print("Entrainement du model: " + Nom_Model)
trainer = pl.Trainer(max_epochs=nb_Epoch, devices=1, accelerator="gpu")
trainer.fit(model, dataloader_train_model)

torch.save(model.state_dict(), model_Path)
In [ ]:
#Chargement du model
model =  AIM()
model.load_state_dict(torch.load(model_Path))
print(model_Path)
In [ ]:
#Source:[https://docs.lightly.ai/self-supervised-learning/tutorials/package/tutorial_moco_memory_bank.html]
stop_random()
import contextlib
import os
import sys

print("Entrainement du classifieur pour le model: " + Nom_Model)

if(not(BNF)):
   model.eval()
   print("Eval")
else:
   model.train()
   print("Train")

log_file_path = 'training_output.log'# PyTorch n'aime pas afficher beaucoup de texte, donc on le stocke dans un fichier à la place.
with open(log_file_path, 'w') as log_file_path:
   with contextlib.redirect_stdout(log_file_path), contextlib.redirect_stderr(log_file_path):
      classifier = Classifier(model.backbone)
      trainer = pl.Trainer(max_epochs=nb_Epoch, devices=1, accelerator="gpu")
      trainer.fit(classifier, dataloader_train_classifier, dataloader_val)

print("FINITO")


#Sauvegarde le classificateur
torch.save(classifier.state_dict(), path_Class)
In [ ]:
#Charge le classificateur
classi =  Classifier(model.backbone)
classi.load_state_dict(torch.load(path_Class))
classifier= classi
In [ ]:
stop_random()
# Partie test : va tester le modèle sur l'ensemble du dossier de test, et donner les prédictions par classe ainsi qu'une matrice de confusion.
import torch.nn.functional as F
import seaborn as sns
import pandas as pd
from sklearn.metrics import confusion_matrix


print("Model choisi:" + Nom_Model +",Avec "+ str(nb_Epoch)+" Epoch,et le dossier " +dossier)
```

```
print(model_Path)
print("NB de classe: " + str(nb_Classe))


class_names=['[0]', '[1]', '[2]', '[3]', '[4]', '[5]', '[6]' ,'[7]']
if(nb_Classe == 8):
   class_names=['[0]', '[1]', '[2]', '[3]', '[4]', '[5]', '[6]' ,'[7]']
else:
   class_names=['[0]', '[1]', '[2]', '[3]', '[4]', '[5]', '[6]']
total = 0
nb_true_pred = 0
total_AC = 0
taille_dossier= [0 for x in range(nb_Classe)]

predictions= [[0 for x in range(nb_Classe)] for y in range(nb_Classe)]


for j in range(0,nb_Classe):
   image_folder = dataset +"_Full/test/"

   image_folder = image_folder + "[" + str(j) + "]"
   nb_true_pred = 0
   taille_dossier[j] = len(os.listdir(image_folder))
   classifier.eval()

   #print(taille_dossier[j])
   for filename in os.listdir(image_folder):
      if filename.endswith(".png"):
         image_path = os.path.join(image_folder, filename)

         image = Image.open(image_path).convert('RGB')
         image_tensor = val_transforms(image)
         image_tensor = image_tensor.unsqueeze(0)

         with torch.no_grad():

            output = classifier(image_tensor)
            probabilities = F.softmax(output, dim=1)

            _, predicted_class = torch.max(output, 1)
            predicted_class_name = class_names[predicted_class.item()]

            confidence = torch.max(probabilities).item() * 100
            true_class=os.path.basename(os.path.dirname(image_path))
            temp = predicted_class_name
            temp = temp.replace('[','')
            temp = temp.replace(']','')
            predictions[j][int(temp)] = predictions[j][int(temp)] +1
            if(true_class == predicted_class_name):
               nb_true_pred = nb_true_pred +1
   print("Classe:"+ str(j) + f" % de bonne prédiction:" + str((nb_true_pred/taille_dossier[j])* 100) +"%" )
   total = total + taille_dossier[j]
   total_AC = total_AC + nb_true_pred

print("Accuray général:" +str(( total_AC / total)*100)+"%")

for j in range(0,nb_Classe):
   for i in range (0,nb_Classe):
      predictions[j][i] = (predictions[j][i]/taille_dossier[j])*100
matrice_C = np.array(predictions)
plt.figure(figsize=(6, 6))
sns.heatmap(matrice_C, annot=True, fmt='.2f', cmap='Blues')
plt.xlabel('Prédiction')
plt.ylabel('Vrai classe')
plt.title('Matrice')
```

```
image_Path = 'M_image_'+dataset+'/'+str(nb_Epoch) +"_" + "EPOCH"+"_" + dossier +"_"+
Nom_Model+"_"+str(BNF)+".png"
plt.savefig(image_Path)
plt.show()
In [ ]:
# Utilisé pour copier plus simplement les résultats dans un fichier Excel.
total2=0
for j in range(0,nb_Classe):
   total2 = total2 + predictions[j][j]
   temp =f"{predictions[j][j]:.2f}".replace('.', ',')
   print(temp)
temp =f"{(total_AC / total)*100:.2f}".replace('.', ',')

print(temp)
temp =f"{total2/nb_Classe:.2f}".replace('.', ',')
print(temp)
```

12.4.6 Modèle Fusion

```
# Chaque bloc possède un lien vers la source originale, formaté comme ceci : #Source: []

# Rappels avant de lancer le programme :
#   - bien vérifier le dataset et le dossier utilisé.
#   - bien vérifier les paramètres et les dossiers de sauvegarde.
#   - faire attention de ne pas lancer le modèle alors qu'il a déjà été entraîné.
#   - Si le premier epoch est très lent, c'est normal, il suffit d'attendre un peu que les données soient chargées.

# Cette cellule contient les imports nécessaires au bon fonctionnement du code.

import torch
import torchvision
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torchvision.transforms as transforms
from torchsummary import summary
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision import datasets, models

import os
import random
import copy
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image


import seaborn as sns
import pandas as pd

from sklearn.metrics import confusion_matrix
import pytorch_lightning as pl

from lightly.data import LightlyDataset
from lightly.loss import NTXentLoss
#from lightly.models import ResclasGenerator
from lightly.models.modules.heads import MoCoProjectionHead
from lightly.models.utils import (
   batch_shuffle,
```

```
    batch_unshuffle,
    deactivate_requires_grad,
    update_momentum,
)
#Import spécifique à chaque model
#Moco:
from lightly.loss import NTXentLoss
from lightly.models.modules import MoCoProjectionHead
from lightly.models.utils import deactivate_requires_grad, update_momentum
from lightly.transforms.moco_transform import MoCoV2Transform
from lightly.utils.scheduler import cosine_schedule

#SimCLR:
from lightly.loss import NTXentLoss
from lightly.models.modules import SimCLRProjectionHead
from lightly.transforms.simclr_transform import SimCLRTransform
from lightly.transforms import SimCLRTransform, utils

#AIM
from lightly.models import utils
from lightly.models.modules import AIMPredictionHead, MaskedCausalVisionTransformer
from lightly.transforms import AIMTransform
#MAE
from lightly.models import utils
from lightly.models.modules import MAEDecoderTIMM, MaskedVisionTransformerTIMM
from lightly.transforms import MAETransform
import csv
torch.__version__
```

In [ ]:

```
# Fonction utilisée pendant le programme pour donner une seed aux différentes fonctions aléatoires.
# A mettre au début de chaque cellule.
# Source : [https://pytorch.org/docs/stable/notes/randomness.html]
def stop_random():
    seed=621
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    random.seed(seed)
    np.random.seed(seed)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False
```

In [ ]:

```
# Configuration générale. Attention à bien choisir les bons paramètres et les bon emplacement de
sauvegarde.
stop_random()
nb_Epoch = 100
batch_Size = 128
taille_Image = 28
BNF = True
dossier="Full"
dataset="Derma"
Moco_path_Class = './PytorchModelV3/'+"MoCo_Class#"+dossier +"#"+dataset+str(nb_Epoch) +"#"
+str(BNF)+".pth"
SimCLR_path_Class = './PytorchModelV3/'+"SimCLR_Class#"+dossier +"#"+dataset+str(nb_Epoch) +"#"
+str(BNF)+".pth"
AIM_path_Class = './PytorchModelV3/'+"AIM_Class#"+dossier +"#"+dataset+str(nb_Epoch) +"#"
+str(BNF)+".pth"
MAE_path_Class = './PytorchModelV3/'+"MAE_Class#"+dossier +"#"+dataset+str(50) +"#"
+str(BNF)+".pth"
```

```
print(Moco_path_Class)
path_val =  dataset+"_"+dossier+"/Dataset/val"
path_Train = dataset+"_"+dossier+"/Dataset/train"
nb_Classe = len(next(os.walk(dataset+"_"+dossier+"/test"))[1])


Model_Actif = "FSAAM"
#FSM Sim et MoCo
#FAA Aim et MAE
#FSAAM Sim,AIM,MAE,MoCo
```

In [ ]:

```
#Source:[https://docs.lightly.ai/self-supervised-
learning/tutorials/package/tutorial_moco_memory_bank.html]

# Explication sur les paramètres des dataloaders :
#     shuffle --> mélange les images, à utiliser surtout pour les données d'entraînement du modèle SSL.
#     drop_last --> sinon, parfois, l'entraînement peut planter à un epoch.

stop_random()

val_transforms_MAE = torchvision.transforms.Compose(
  [
    transforms.Resize(224),
    torchvision.transforms.ToTensor(),
  ])

val_transforms = torchvision.transforms.Compose(
  [
    torchvision.transforms.ToTensor(),
  ])


import torch.optim as optim

#Car les dataloaders de lightly n'ont pas l'attribut targets, donc on est obligé d'utiliser cette méthode
#Ici ne sert a pas a grand chose, mais garder pour la continuiter des scripts
nb_Element = [0] * nb_Classe
i = 0
total = 0
for sous_d in os.listdir(path_Train):
  sous_d = os.path.join(path_Train, sous_d)
  nb_Element[i] = len(os.listdir(sous_d))
  i = i +1
for k in range(0, nb_Classe):
  total = total + nb_Element[k]

class_weights = []
for exemple in nb_Element:
  poid = (total-exemple) / (total)
  class_weights.append(poid)
  print(poid)
#Source:[https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html]
class_weights = torch.tensor(class_weights,dtype=torch.float32).cuda()
criterion_ = nn.CrossEntropyLoss(weight=class_weights)
print(class_weights)
```

In [ ]:

```
#MoCo Model
stop_random()
```

```
class Moco(pl.LightningModule):
  def __init__(self):
    super().__init__()
    resnet = torchvision.models.resnet18()
    self.backbone = nn.Sequential(*list(resnet.children())[:-1])
    self.projection_head = MoCoProjectionHead(512, 1024, 1024)

    self.backbone_momentum = copy.deepcopy(self.backbone)
    self.projection_head_momentum = copy.deepcopy(self.projection_head)

    deactivate_requires_grad(self.backbone_momentum)
    deactivate_requires_grad(self.projection_head_momentum)

    self.criterion = NTXentLoss(memory_bank_size=(4096, 1024))
    self.train_losses = []

  def forward(self, x):
    query = self.backbone(x).flatten(start_dim=1)
    query = self.projection_head(query)
    return query

  def forward_momentum(self, x):
    key = self.backbone_momentum(x).flatten(start_dim=1)
    key = self.projection_head_momentum(key).detach()
    return key

  def training_step(self, batch, batch_idx):
    momentum = cosine_schedule(self.current_epoch, nb_Epoch, 0.996, 1)
    update_momentum(self.backbone, self.backbone_momentum, m=momentum)
    update_momentum(self.projection_head, self.projection_head_momentum, m=momentum)
    x_query, x_key = batch[0]
    query = self.forward(x_query)
    key = self.forward_momentum(x_key)
    loss = self.criterion(query, key)
    self.log("train_loss_ssl", loss)
    self.log('train_loss', loss, on_step=True, on_epoch=True, prog_bar=False, logger=False)#Log
    return loss


  def configure_optimizers(self):
    optim = torch.optim.SGD(self.parameters(), lr=0.06)
    return optim

  def on_train_epoch_end(self):
    epoch_loss = self.trainer.callback_metrics['train_loss']
    self.train_losses.append(epoch_loss.item())
    print(f'Epoch {self.current_epoch}: Train Loss: {epoch_loss.item()}')

  def on_train_end(self):
    print("LOSS:")
    for i in self.train_losses:
      temp =f"{i:.5f}".replace('.', ',')
      print(temp )
```

In [ ]:

```
#Source:[https://docs.lightly.ai/self-supervised-learning/examples/simclr.html]
#Model SimCLR
stop_random()
class SimCLR(pl.LightningModule):
  def __init__(self):
```

```
        super().__init__()
        resnet = torchvision.models.resnet18()
        self.backbone = nn.Sequential(*list(resnet.children())[:-1])
        self.projection_head = SimCLRProjectionHead(512, 1024, 1024)
        self.criterion = NTXentLoss()
        self.train_losses = []

    def forward(self, x):
        x = self.backbone(x).flatten(start_dim=1)
        z = self.projection_head(x)
        return z

    def training_step(self, batch, batch_index):
        (x0, x1) = batch[0]
        z0 = self.forward(x0)
        z1 = self.forward(x1)
        loss = self.criterion(z0, z1)
        self.log('train_loss', loss, on_step=True, on_epoch=True, prog_bar=False, logger=False)#Log
        return loss

    def configure_optimizers(self):
        optim = torch.optim.SGD(self.parameters(), lr=0.06)#0.06
        return optim

    def on_train_epoch_end(self):
        epoch_loss = self.trainer.callback_metrics['train_loss']
        self.train_losses.append(epoch_loss.item())
        print(f'Epoch {self.current_epoch}: Train Loss: {epoch_loss.item()}')

    def on_train_end(self):
        print("LOSS:")
        for i in self.train_losses:
            temp =f"{i:.5f}".replace('.', ',')
            print(temp )
```

In [ ]:

```
#Source:[https://docs.lightly.ai/self-supervised-learning/examples/aim.html]

import pytorch_lightning as pl
import torch
import torchvision
from torch import nn

from lightly.models import utils
from lightly.models.modules import AIMPredictionHead, MaskedCausalVisionTransformer
from lightly.transforms import AIMTransform

class AIM(pl.LightningModule):#
    def __init__(self) -> None:
        super().__init__()
        self.train_losses = []
        vit = MaskedCausalVisionTransformer(
            img_size=28,
            patch_size=4,
            embed_dim=280,
            depth=12,
            num_heads=1,
            qk_norm=False,
            class_token=False,
            no_embed_class=True,
```

```
        )
        utils.initialize_2d_sine_cosine_positional_embedding(
            pos_embedding=vit.pos_embed, has_class_token=vit.has_class_token
        )
        self.patch_size = vit.patch_embed.patch_size[0]
        self.num_patches = vit.patch_embed.num_patches

        self.backbone = vit
        self.projection_head = AIMPredictionHead(

            input_dim=vit.embed_dim, output_dim=3 * self.patch_size**2, num_blocks=1
        )

        self.criterion = nn.MSELoss()

    def training_step(self, batch, batch_idx):
        views, targets = batch[0], batch[1]
        images = views[0]
        batch_size = images.shape[0]

        mask = utils.random_prefix_mask(
            size=(batch_size, self.num_patches),
            max_prefix_length=self.num_patches - 1,
            device=images.device,
        )
        features = self.backbone.forward_features(images, mask=mask)

        features = self.backbone._pos_embed(features)
        predictions = self.projection_head(features)


        patches = utils.patchify(images, self.patch_size)
        patches = utils.normalize_mean_var(patches, dim=-1)

        loss = self.criterion(predictions, patches)
        self.log('train_loss', loss, on_step=True, on_epoch=True, prog_bar=False, logger=False)#Log
        return loss

    def configure_optimizers(self):
        optim = torch.optim.AdamW(self.parameters(), lr=1.5e-4)
        return optim

    def on_train_epoch_end(self):
        epoch_loss = self.trainer.callback_metrics['train_loss']
        self.train_losses.append(epoch_loss.item())
        print(f'Epoch {self.current_epoch}: Train Loss: {epoch_loss.item()}')

    def on_train_end(self):
        print("LOSS:")
        for i in self.train_losses:
            temp =f"{i:.5f}".replace('.', ',')
            print(temp )
```

In [ ]:

```
#Source:[https://docs.lightly.ai/self-supervised-learning/examples/mae.html]
#Model MAE


from vit_pytorch import ViT
import pytorch_lightning as pl
```

```
import torch
import torchvision
from timm.models.vision_transformer import vit_base_patch32_224
from torch import nn

from lightly.models import utils
from lightly.models.modules import MAEDecoderTIMM, MaskedVisionTransformerTIMM
from lightly.transforms import MAETransform

from timm import create_model


class MAE(pl.LightningModule):
  def __init__(self):
    super().__init__()
    self.train_losses = []
    num_classes = 8
    vit = vit_base_patch32_224()
    decoder_dim = 512

    self.mask_ratio = 0.75
    self.patch_size = vit.patch_embed.patch_size[0]
    self.backbone = MaskedVisionTransformerTIMM(vit=vit)
    self.sequence_length = self.backbone.sequence_length
    self.decoder = MAEDecoderTIMM(
      num_patches=vit.patch_embed.num_patches,
      patch_size=self.patch_size,
      embed_dim=vit.embed_dim,
      decoder_embed_dim=decoder_dim,
      in_chans=3,#NEW
      decoder_depth=1,
      decoder_num_heads=16,
      mlp_ratio=4.0,
      proj_drop_rate=0.0,
      attn_drop_rate=0.0,
    )
    self.criterion = nn.MSELoss()

  def forward_encoder(self, images, idx_keep=None):
    return self.backbone.encode(images=images, idx_keep=idx_keep)

  def forward_decoder(self, x_encoded, idx_keep, idx_mask):
    # build decoder input
    batch_size = x_encoded.shape[0]
    x_decode = self.decoder.embed(x_encoded)
    x_masked = utils.repeat_token(
      self.decoder.mask_token, (batch_size, self.sequence_length)
    )
    x_masked = utils.set_at_index(x_masked, idx_keep, x_decode.type_as(x_masked))

    # decoder forward pass
    x_decoded = self.decoder.decode(x_masked)

    # predict pixel values for masked tokens
    x_pred = utils.get_at_index(x_decoded, idx_mask)
    x_pred = self.decoder.predict(x_pred)
    return x_pred

  def training_step(self, batch, batch_idx):
```

```
        views = batch[0]
        images = views[0]  # views contains only a single view
        batch_size = images.shape[0]
        idx_keep, idx_mask = utils.random_token_mask(
            size=(batch_size, self.sequence_length),
            mask_ratio=self.mask_ratio,
            device=images.device,
        )
        x_encoded = self.forward_encoder(images=images, idx_keep=idx_keep)
        x_pred = self.forward_decoder(
            x_encoded=x_encoded, idx_keep=idx_keep, idx_mask=idx_mask
        )

        # get image patches for masked tokens
        patches = utils.patchify(images, self.patch_size)
        # must adjust idx_mask for missing class token
        target = utils.get_at_index(patches, idx_mask - 1)

        loss = self.criterion(x_pred, target)
        self.log('train_loss', loss, on_step=True, on_epoch=True, prog_bar=False, logger=False)#Log
        return loss

    def configure_optimizers(self):
        optim = torch.optim.AdamW(self.parameters(), lr=1.5e-4)
        return optim

    def on_train_epoch_end(self):
        epoch_loss = self.trainer.callback_metrics['train_loss']
        self.train_losses.append(epoch_loss.item())
        print(f'Epoch {self.current_epoch}: Train Loss: {epoch_loss.item()}')

    def on_train_end(self):
        print("LOSS:")
        for i in self.train_losses:
            temp =f"{i:.5f}".replace('.', ',')
            print(temp )
```

In [ ]:

```
#Source:[https://docs.lightly.ai/self-supervised-
learning/tutorials/package/tutorial_moco_memory_bank.html]
#Classificateur  Moco
class Classifier_MoCo(pl.LightningModule):
    def __init__(self, backbone):
        super().__init__()
        self.backbone = backbone


        self.train_losses_tab = []
        self.train_accuracy_tab = []
        self.val_losses_tab = []
        self.val_accuracy_tab = []

        self.train_losses = 0
        self.train_accuracy = 0
        self.val_losses = 0
        self.val_accuracy = 0

        self.t_iteration = 0
        self.v_iteration = 0
```

```
        # freeze the backbone
        if(not(BNF)):
            deactivate_requires_grad(backbone)
        self.fc1 = nn.Linear(512, 2048)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(2048, nb_Classe)

        self.validation_step_outputs = []
        self.criterion = criterion_

    def forward(self, x):
        x = self.backbone(x).flatten(start_dim=1)
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        y_hat = x
        return y_hat

    def training_step(self, batch, batch_idx):
        x, y, _ = batch
        y_hat = self.forward(x)
        loss =  self.criterion(y_hat, y)


        t_acc = (y_hat.argmax(dim=1) == y).float().mean()
        self.train_losses = self.train_losses + loss.item()
        self.train_accuracy = self.train_accuracy + t_acc.item()
        self.t_iteration =  self.t_iteration +1
        return loss

    def on_train_epoch_end(self):

        self.train_losses_tab.append(self.train_losses/self.t_iteration)
        self.train_accuracy_tab.append(self.train_accuracy/self.t_iteration)
        self.train_losses = 0
        self.train_accuracy = 0
        self.t_iteration = 0


    def on_train_end(self):

        self.train_losses= np.array(self.train_losses_tab)
        self.train_accuracy= np.array(self.train_accuracy_tab)

        self.val_losses= np.array(self.val_losses_tab)
        self.val_accuracy= np.array(self.val_accuracy_tab)


        import csv
        titre = ['training_loss','validation_loss','training_accuracy','validation_accuracy']

        tableaux_complet = list(zip(self.train_losses,self.train_accuracy,self.val_losses,self.val_accuracy))
        path_save_data = './Entrainement/'+ dataset + "/Moco_Class#"+dossier +"#"+str(nb_Epoch) +"#"
+str(BNF)+".csv"
        print(path_save_data)
        with open(path_save_data, mode='w', newline='', encoding='utf-8') as fichier:
            writer = csv.writer(fichier)
            writer.writerow(titre)
            for i in range(len(self.train_losses)):
```

```
            temp1 =f"{self.train_losses[i]:.5f}".replace('.', ',')
            temp2 =f"{self.val_losses[i]:.5f}".replace('.', ',')
            temp3 =f"{self.train_accuracy[i]:.5f}".replace('.', ',')
            temp4 =f"{self.val_accuracy[i]:.5f}".replace('.', ',')
            writer.writerow([temp1, temp2, temp3, temp4])

  def custom_histogram_weights(self):
    for name, params in self.named_parameters():
      self.logger.experiment.add_histogram(name, params, self.current_epoch)

  def validation_step(self, batch, batch_idx):
    x, y, _ = batch
    y_hat = self.forward(x)
    loss =  self.criterion(y_hat, y)
    v_acc = (y_hat.argmax(dim=1) == y).float().mean()

    self.val_losses =self.val_losses + loss.item()
    self.val_accuracy = self.val_accuracy + v_acc.item()
    self.v_iteration = self.v_iteration +1

  def on_validation_epoch_end(self):

    self.val_losses_tab.append(self.val_losses/self.v_iteration)
    self.val_accuracy_tab.append(self.val_accuracy/self.v_iteration)

    self.val_losses = 0
    self.val_accuracy = 0
    self.v_iteration = 0

  def configure_optimizers(self):
    optim = torch.optim.SGD(self.parameters(), lr=0.01, momentum=0.9)
    scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optim, T_max=self.trainer.max_epochs)
    return [optim], [scheduler]
```

In [ ]:

```
#Source:[https://docs.lightly.ai/self-supervised-
learning/tutorials/package/tutorial_moco_memory_bank.html]
#Classificateur  SimCLR
class Classifier_SimCLR(pl.LightningModule):#Pour les variables ne pas oubliez les SELF
  def __init__(self, backbone):
    super().__init__()
    self.backbone = backbone

    self.train_losses_tab = []
    self.train_accuracy_tab = []
    self.val_losses_tab = []
    self.val_accuracy_tab = []

    self.train_losses = 0
    self.train_accuracy = 0
    self.val_losses = 0
    self.val_accuracy = 0

    self.t_iteration = 0
    self.v_iteration = 0

    # freeze the backbone
    if(not(BNF)):
      deactivate_requires_grad(backbone)
    self.fc1 = nn.Linear(512, 2048)
```

```
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(2048, nb_Classe)

        self.validation_step_outputs = []
        self.criterion = criterion_

    def forward(self, x):
        x = self.backbone(x).flatten(start_dim=1)
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        y_hat = x
        return y_hat

    def training_step(self, batch, batch_idx):
        x, y, _ = batch
        y_hat = self.forward(x)
        loss =  self.criterion(y_hat, y)


        t_acc = (y_hat.argmax(dim=1) == y).float().mean()
        self.train_losses = self.train_losses + loss.item()
        self.train_accuracy = self.train_accuracy + t_acc.item()
        self.t_iteration =  self.t_iteration +1
        return loss

    def on_train_epoch_end(self):

        self.train_losses_tab.append(self.train_losses/self.t_iteration)
        self.train_accuracy_tab.append(self.train_accuracy/self.t_iteration)
        self.train_losses = 0
        self.train_accuracy = 0
        self.t_iteration = 0


    def on_train_end(self):

        self.train_losses= np.array(self.train_losses_tab)
        self.train_accuracy= np.array(self.train_accuracy_tab)

        self.val_losses= np.array(self.val_losses_tab)
        self.val_accuracy= np.array(self.val_accuracy_tab)

        titre = ['training_loss','validation_loss','training_accuracy','validation_accuracy']

        tableaux_complet = list(zip(self.train_losses,self.train_accuracy,self.val_losses,self.val_accuracy))
        path_save_data = './Entrainement/'+ dataset + "/SimCLR_Class#"+dossier +"#"+str(nb_Epoch) +"#"
+str(BNF)+".csv"
        print(path_save_data)
        with open(path_save_data, mode='w', newline='', encoding='utf-8') as fichier:
            writer = csv.writer(fichier)
            writer.writerow(titre)
            for i in range(len(self.train_losses)):
                temp1 =f"{self.train_losses[i]:.5f}".replace('.', ',')
                temp2 =f"{self.val_losses[i]:.5f}".replace('.', ',')
                temp3 =f"{self.train_accuracy[i]:.5f}".replace('.', ',')
                temp4 =f"{self.val_accuracy[i]:.5f}".replace('.', ',')
                writer.writerow([temp1, temp2, temp3, temp4])
```

```
  def custom_histogram_weights(self):
    for name, params in self.named_parameters():
      self.logger.experiment.add_histogram(name, params, self.current_epoch)

  def validation_step(self, batch, batch_idx):
    x, y, _ = batch
    y_hat = self.forward(x)
    loss =  self.criterion(y_hat, y)
    v_acc = (y_hat.argmax(dim=1) == y).float().mean()

    self.val_losses =self.val_losses + loss.item()
    self.val_accuracy = self.val_accuracy + v_acc.item()
    self.v_iteration = self.v_iteration +1

  def on_validation_epoch_end(self):

    self.val_losses_tab.append(self.val_losses/self.v_iteration)
    self.val_accuracy_tab.append(self.val_accuracy/self.v_iteration)

    self.val_losses = 0
    self.val_accuracy = 0
    self.v_iteration = 0

  def configure_optimizers(self):
    optim = torch.optim.SGD(self.parameters(), lr=0.01, momentum=0.9)
    scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optim, T_max=self.trainer.max_epochs)
    return [optim], [scheduler]
```

In [ ]:

```
#Source:[https://docs.lightly.ai/self-supervised-
learning/tutorials/package/tutorial_moco_memory_bank.html]
#Classificateur AIM
class Classifier_AIM(pl.LightningModule):#Pour les variables ne pas oubliez les SELF
  def __init__(self, backbone):
    super().__init__()
    self.backbone = backbone


    self.train_losses_tab = []
    self.train_accuracy_tab = []
    self.val_losses_tab = []
    self.val_accuracy_tab = []

    self.train_losses = 0
    self.train_accuracy = 0
    self.val_losses = 0
    self.val_accuracy = 0

    self.t_iteration = 0
    self.v_iteration = 0

    self.fc0 = nn.Linear(49000, 512)
    deactivate_requires_grad(self.fc0)
    # freeze the backbone
    if(not(BNF)):
      deactivate_requires_grad(backbone)
    self.fc1 = nn.Linear(512, 2048)
    self.relu = nn.ReLU()
    self.fc2 = nn.Linear(2048, nb_Classe)
```

```
        self.validation_step_outputs = []
        self.criterion = criterion_

    def forward(self, x):
        x = self.backbone(x).flatten(start_dim=1)
        x = self.fc0(x)
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        y_hat = x
        return y_hat

    def training_step(self, batch, batch_idx):
        x, y, _ = batch
        y_hat = self.forward(x)
        loss =  self.criterion(y_hat, y)


        t_acc = (y_hat.argmax(dim=1) == y).float().mean()
        self.train_losses = self.train_losses + loss.item()
        self.train_accuracy = self.train_accuracy + t_acc.item()
        self.t_iteration =  self.t_iteration +1
        return loss

    def on_train_epoch_end(self):

        self.train_losses_tab.append(self.train_losses/self.t_iteration)
        self.train_accuracy_tab.append(self.train_accuracy/self.t_iteration)
        self.train_losses = 0
        self.train_accuracy = 0
        self.t_iteration = 0


    def on_train_end(self):

        self.train_losses= np.array(self.train_losses_tab)
        self.train_accuracy= np.array(self.train_accuracy_tab)

        self.val_losses= np.array(self.val_losses_tab)
        self.val_accuracy= np.array(self.val_accuracy_tab)

        #Enregistrement
        titre = ['training_loss','validation_loss','training_accuracy','validation_accuracy']

        tableaux_complet = list(zip(self.train_losses,self.train_accuracy,self.val_losses,self.val_accuracy))
        path_save_data = './Entrainement/'+ dataset + "/AIM_Class#"+dossier +"#"+str(nb_Epoch) +"#"
+str(BNF)+".csv"
        print(path_save_data)
        with open(path_save_data, mode='w', newline='', encoding='utf-8') as fichier:
            writer = csv.writer(fichier)
            writer.writerow(titre)
            for i in range(len(self.train_losses)):
                temp1 =f"{self.train_losses[i]:.5f}".replace('.', ',')
                temp2 =f"{self.val_losses[i]:.5f}".replace('.', ',')
                temp3 =f"{self.train_accuracy[i]:.5f}".replace('.', ',')
                temp4 =f"{self.val_accuracy[i]:.5f}".replace('.', ',')
                writer.writerow([temp1, temp2, temp3, temp4])

    def custom_histogram_weights(self):
```

```
        for name, params in self.named_parameters():
            self.logger.experiment.add_histogram(name, params, self.current_epoch)

    def validation_step(self, batch, batch_idx):
        x, y, _ = batch
        y_hat = self.forward(x)
        loss =  self.criterion(y_hat, y)
        v_acc = (y_hat.argmax(dim=1) == y).float().mean()

        self.val_losses =self.val_losses + loss.item()
        self.val_accuracy = self.val_accuracy + v_acc.item()
        self.v_iteration = self.v_iteration +1

    def on_validation_epoch_end(self):

        self.val_losses_tab.append(self.val_losses/self.v_iteration)
        self.val_accuracy_tab.append(self.val_accuracy/self.v_iteration)

        self.val_losses = 0
        self.val_accuracy = 0
        self.v_iteration = 0

    def configure_optimizers(self):
        optim = torch.optim.SGD(self.parameters(), lr=0.01, momentum=0.9)
        scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optim, T_max=self.trainer.max_epochs)
        return [optim], [scheduler]
```

In [ ]:

```
#Source:[https://docs.lightly.ai/self-supervised-
learning/tutorials/package/tutorial_moco_memory_bank.html]
#Classificateur MAE
class Classifier_MAE(pl.LightningModule):#Pour les variables ne pas oubliez les SELF
    def __init__(self, backbone):
        super().__init__()
        self.backbone = backbone
        #Marche très bien

        self.train_losses_tab = []
        self.train_accuracy_tab = []
        self.val_losses_tab = []
        self.val_accuracy_tab = []

        self.train_losses = 0
        self.train_accuracy = 0
        self.val_losses = 0
        self.val_accuracy = 0

        self.t_iteration = 0
        self.v_iteration = 0

        # freeze the backbone
        if(not(BNF)):
            deactivate_requires_grad(backbone)
        self.fc1 = nn.Linear(768, 2048)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(2048, nb_Classe)

        self.validation_step_outputs = []
        self.criterion = criterion_
```

```
    def forward(self, x):
        x = self.backbone(x).flatten(start_dim=1)
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        y_hat = x
        return y_hat

    def training_step(self, batch, batch_idx):
        x, y, _ = batch
        y_hat = self.forward(x)
        loss =  self.criterion(y_hat, y)

        t_acc = (y_hat.argmax(dim=1) == y).float().mean()
        self.train_losses = self.train_losses + loss.item()
        self.train_accuracy = self.train_accuracy + t_acc.item()
        self.t_iteration =  self.t_iteration +1
        return loss

    def on_train_epoch_end(self):

        self.train_losses_tab.append(self.train_losses/self.t_iteration)
        self.train_accuracy_tab.append(self.train_accuracy/self.t_iteration)
        self.train_losses = 0
        self.train_accuracy = 0
        self.t_iteration = 0


    def on_train_end(self):

        self.train_losses= np.array(self.train_losses_tab)
        self.train_accuracy= np.array(self.train_accuracy_tab)

        self.val_losses= np.array(self.val_losses_tab)
        self.val_accuracy= np.array(self.val_accuracy_tab)

        #Enregistrement
        titre = ['training_loss','validation_loss','training_accuracy','validation_accuracy']

        tableaux_complet = list(zip(self.train_losses,self.train_accuracy,self.val_losses,self.val_accuracy))
        path_save_data = './Entrainement/'+ dataset + "/MAE_Class#"+dossier +"#"+str(nb_Epoch) +"#"
+str(BNF)+".csv"
        print(path_save_data)
        with open(path_save_data, mode='w', newline='', encoding='utf-8') as fichier:
            writer = csv.writer(fichier)
            writer.writerow(titre)
            for i in range(len(self.train_losses)):
                temp1 =f"{self.train_losses[i]:.5f}".replace('.', ',')
                temp2 =f"{self.val_losses[i]:.5f}".replace('.', ',')
                temp3 =f"{self.train_accuracy[i]:.5f}".replace('.', ',')
                temp4 =f"{self.val_accuracy[i]:.5f}".replace('.', ',')
                writer.writerow([temp1, temp2, temp3, temp4])

    def custom_histogram_weights(self):
        for name, params in self.named_parameters():
            self.logger.experiment.add_histogram(name, params, self.current_epoch)

    def validation_step(self, batch, batch_idx):
        x, y, _ = batch
```

```
        y_hat = self.forward(x)
        loss =  self.criterion(y_hat, y)
        v_acc = (y_hat.argmax(dim=1) == y).float().mean()

        self.val_losses =self.val_losses + loss.item()
        self.val_accuracy = self.val_accuracy + v_acc.item()
        self.v_iteration = self.v_iteration +1

    def on_validation_epoch_end(self):

        self.val_losses_tab.append(self.val_losses/self.v_iteration)
        self.val_accuracy_tab.append(self.val_accuracy/self.v_iteration)

        self.val_losses = 0
        self.val_accuracy = 0
        self.v_iteration = 0

    def configure_optimizers(self):
        optim = torch.optim.SGD(self.parameters(), lr=0.01, momentum=0.9)
        scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optim, T_max=self.trainer.max_epochs)
        return [optim], [scheduler]
```

In [ ]:

```
torch.set_float32_matmul_precision('medium')
model_Moco =Moco()
model_SimClR =SimCLR()
model_AIM =AIM()
model_MAE =MAE()
#Charge les différents classificateurs

class_Moco =  Classifier_MoCo(model_Moco.backbone)
class_SimCLR =  Classifier_SimCLR(model_SimClR.backbone)
class_AIM =  Classifier_AIM(model_AIM.backbone)
class_MAE =  Classifier_MAE(model_MAE.backbone)

class_Moco.load_state_dict(torch.load(Moco_path_Class))
class_SimCLR.load_state_dict(torch.load(SimCLR_path_Class))
class_AIM.load_state_dict(torch.load(AIM_path_Class))
class_MAE.load_state_dict(torch.load(MAE_path_Class))
```

In [ ]:

```
stop_random()
# Partie test : va tester le modèle sur l'ensemble du dossier de test, et donner les prédictions par classe ainsi qu'une matrice de confusion.
import torch.nn.functional as F
import seaborn as sns
import pandas as pd
from sklearn.metrics import confusion_matrix


print("NB de classe: " + str(nb_Classe))

class_names=['[0]', '[1]', '[2]', '[3]', '[4]', '[5]', '[6]' ,'[7]']
if(nb_Classe == 8):
    class_names=['[0]', '[1]', '[2]', '[3]', '[4]', '[5]', '[6]' ,'[7]']
else:
    class_names=['[0]', '[1]', '[2]', '[3]', '[4]', '[5]', '[6]']

total = 0
nb_true_pred = 0
total_AC = 0
```

```
taille_dossier= [0 for x in range(nb_Classe)]

predictions= [[0 for x in range(nb_Classe)] for y in range(nb_Classe)]
i = 0

for j in range(0,nb_Classe):
    image_folder = dataset +"_Full/test/"

    image_folder = image_folder + "[" + str(j) + "]"
    nb_true_pred = 0
    taille_dossier[j] = len(os.listdir(image_folder))

    class_Moco.eval()
    class_SimCLR.eval()
    class_AIM.eval()
    class_MAE.eval()


    output_SimCLR=None
    output_MoCo=None
    output_AIM=None
    output_MAE=None

    probabilities_MoCo=None
    probabilities_SimCLR=None
    probabilities_AIM=None
    probabilities_MAE=None
    probabilities=None


    for filename in os.listdir(image_folder):
        if filename.endswith(".png"):
            image_path = os.path.join(image_folder, filename)
            image = Image.open(image_path).convert('RGB')
            image_tensor = val_transforms(image)
            image_tensor = image_tensor.unsqueeze(0)
            if(Model_Actif == "FAA" or Model_Actif == "FSAAM"):
                #Pour MAE car image en 224 px
                image_tensor_MAE = val_transforms_MAE(image)
                image_tensor_MAE = image_tensor_MAE.unsqueeze(0)

            with torch.no_grad():

                if(Model_Actif == "FSM" or Model_Actif == "FSAAM"):
                    output_MoCo  = class_Moco(image_tensor)
                    output_SimCLR= class_SimCLR(image_tensor)

                    probabilities_MoCo= F.softmax(output_MoCo, dim=1)
                    probabilities_SimCLR= F.softmax(output_SimCLR, dim=1)
                    probabilities= probabilities_SimCLR*probabilities_MoCo

                if(Model_Actif == "FAA" or Model_Actif == "FSAAM"):
                    output_AIM= class_AIM(image_tensor)
                    output_MAE= class_MAE(image_tensor_MAE)

                    probabilities_AIM = F.softmax( output_AIM, dim=1)
                    probabilities_MAE = F.softmax( output_MAE, dim=1)
                    probabilities= probabilities_MAE*probabilities_AIM
```

```
            if(Model_Actif == "FSAAM"):
               probabilities= probabilities_SimCLR*probabilities_MoCo*probabilities_MAE*probabilities_AIM
            _, predicted_class = torch.max(probabilities, dim=1)
            predicted_class_name = class_names[predicted_class.item()]

            true_class=os.path.basename(os.path.dirname(image_path))
            temp = predicted_class_name
            temp = temp.replace('[','')
            temp = temp.replace(']','')
            predictions[j][int(temp)] = predictions[j][int(temp)] +1
            if(true_class == predicted_class_name):
               nb_true_pred = nb_true_pred +1
   print("Classe:"+ str(j) + f" % de bonne prédiction:" + str((nb_true_pred/taille_dossier[j])* 100) +"%" )
   total = total + taille_dossier[j]
   total_AC = total_AC + nb_true_pred

print("Accuray général:" +str(( total_AC / total)*100)+"%")

for j in range(0,nb_Classe):
   for i in range (0,nb_Classe):
      predictions[j][i] = (predictions[j][i]/taille_dossier[j])*100
matrice_C = np.array(predictions)
plt.figure(figsize=(6, 6))
sns.heatmap(matrice_C, annot=True, fmt='.2f', cmap='Blues')
plt.xlabel('Prédiction')
plt.ylabel('Vrai classe')
plt.title('Matrice')
image_Path = 'M_Image_Fusion/'+str(nb_Epoch) +"_" + "EPOCH"+"_" + dossier +"_"+
dataset+"_"+Model_Actif+"_"+str(BNF)
plt.savefig(image_Path)
plt.show()
```

In [ ]:

```
# Utilisé pour copier plus simplement les résultats dans un fichier Excel.
total2=0
for j in range(0,nb_Classe):
   total2 = total2 + predictions[j][j]
   temp =f"{predictions[j][j]:.2f}".replace('.', ',')
   print(temp)
temp =f"{(total_AC / total)*100:.2f}".replace('.', ',')

print(temp)
temp =f"{total2/nb_Classe:.2f}".replace('.', ',')
print(temp)
```

12.4.7 MoCo Plus

```
# Chaque bloc possède un lien vers la source originale, formaté comme ceci : #Source: []

# Rappels avant de lancer le programme :
#   - bien vérifier le dataset et le dossier utilisé.
#   - bien vérifier les paramètres et les dossiers de sauvegarde.
#   - faire attention de ne pas lancer le modèle alors qu'il a déjà été entraîné.
#   - Si le premier epoch est très lent, c'est normal, il suffit d'attendre un peu que les données soient chargées.

# Cette cellule contient les imports nécessaires au bon fonctionnement du code.


import torch
```

```
import torchvision
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torchvision.transforms as transforms
from torchsummary import summary
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision import datasets, models

import os
import random
import copy
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image


import seaborn as sns
import pandas as pd

from sklearn.metrics import confusion_matrix
import pytorch_lightning as pl

from lightly.data import LightlyDataset
from lightly.loss import NTXentLoss
from lightly.models.modules.heads import MoCoProjectionHead
from lightly.models.utils import (
    batch_shuffle,
    batch_unshuffle,
    deactivate_requires_grad,
    update_momentum,
)
#Import spécifique à chaque model
from lightly.loss import NTXentLoss
from lightly.models.modules import MoCoProjectionHead
from lightly.models.utils import deactivate_requires_grad, update_momentum
from lightly.transforms.moco_transform import MoCoV2Transform
from lightly.utils.scheduler import cosine_schedule


torch.__version__
```

In [ ]:

```
# Fonction utilisée pendant le programme pour donner une seed aux différentes fonctions aléatoires.
# A mettre au début de chaque cellule.
# Source : [https://pytorch.org/docs/stable/notes/randomness.html]
def stop_random():
    seed=621
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    random.seed(seed)
    np.random.seed(seed)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False
```

In [ ]:

```
# Configuration générale. Attention à bien choisir les bons paramètres et les bon emplacement de
sauvegarde.
```

```
stop_random()
nb_Epoch = 100
batch_Size = 128
taille_Image = 28
augmentation=False
num_workers = 4
dossier="Quarter"
dataset="Blood"
BNF = True
#Nomenclature:Nom_Model#Dossier#Epoch#BNF
Nom_Model = "MoCo"
model_Path = './PytorchModelV3/'+"MoCo_Model_PLUS"+"Full" +"#"+dataset+str(nb_Epoch) +"#"
+"FALSE"+".pth"
path_Class = './PytorchModelV3/'+"MoCo_Class_PLUS"+dossier +"#"+dataset+str(nb_Epoch) +"#"
+str(BNF)+".pth"
path_Train_SSL_M = dataset+"_SSL_Plus"
path_Train = dataset+"_"+dossier+"/Dataset/train"
path_val =  dataset+"_"+dossier+"/Dataset/val"
nb_Classe = len(next(os.walk(dataset+"_"+dossier+"/test"))[1])
print("Model choisi MoCo, avec le dossier: " +dossier+ ", avec "+ str(nb_Epoch) + " epoch")
print("Emplacement: " +model_Path)
print("Emplacement: " +path_Class)
```

In [ ]:

```
#Source:[https://docs.lightly.ai/self-supervised-
learning/tutorials/package/tutorial_moco_memory_bank.html]
#Source:[https://docs.lightly.ai/self-supervised-
learning/tutorials/package/tutorial_moco_memory_bank.html]

transform = MoCoV2Transform(input_size=28,gaussian_blur=0.0,)

train_classifier_transforms = torchvision.transforms.Compose(
    [
        torchvision.transforms.ToTensor(),
    ])

if(augmentation):# Si l'on a besoin de faire de l'augmentation de données.
    print("augmentation:")
    train_classifier_transforms = transforms.Compose([
    transforms.RandomResizedCrop(size=28, scale=(0.3, 1.0), ratio=(1, 1)),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
])


val_transforms = torchvision.transforms.Compose(
    [
        torchvision.transforms.ToTensor(),
    ])



dataset_train_model = LightlyDataset(input_dir=path_Train_SSL_M, transform=transform)
dataset_train_classifier = LightlyDataset(input_dir=path_Train, transform=train_classifier_transforms)
dataset_val = LightlyDataset(input_dir=path_val, transform=val_transforms)

print(train_classifier_transforms)
```

In [ ]:

```
#Source:[https://docs.lightly.ai/self-supervised-
learning/tutorials/package/tutorial_moco_memory_bank.html]

# Explication sur les paramètres des dataloaders :
#     shuffle --> mélange les images, à utiliser surtout pour les données d'entraînement du modèle SSL.
#     drop_last --> sinon, parfois, l'entraînement peut planter à un epoch.

stop_random()
dataloader_train_model = torch.utils.data.DataLoader(
    dataset_train_model,
    batch_size=batch_Size,
    shuffle=True,
    drop_last=True,
    num_workers=num_workers,
    persistent_workers=True,
)

dataloader_train_classifier = torch.utils.data.DataLoader(
    dataset_train_classifier,
    batch_size=batch_Size,
    shuffle=True,
    drop_last=True,
    num_workers=num_workers,
    persistent_workers=True,
)

dataloader_val = torch.utils.data.DataLoader(
    dataset_val,
    batch_size=batch_Size,
    shuffle=False,
    drop_last=False,
    num_workers=num_workers,
    persistent_workers=True,
)
```

In [ ]:

```
#Source:[https://docs.lightly.ai/self-supervised-learning/examples/moco.html]
stop_random()
class Moco(pl.LightningModule):
    def __init__(self):
        super().__init__()
        resnet = torchvision.models.resnet18()
        self.backbone = nn.Sequential(*list(resnet.children())[:-1])
        self.projection_head = MoCoProjectionHead(512, 1024, 1024)

        self.backbone_momentum = copy.deepcopy(self.backbone)
        self.projection_head_momentum = copy.deepcopy(self.projection_head)

        deactivate_requires_grad(self.backbone_momentum)
        deactivate_requires_grad(self.projection_head_momentum)

        self.criterion = NTXentLoss(memory_bank_size=(4096, 1024))
        self.train_losses = []

    def forward(self, x):
        query = self.backbone(x).flatten(start_dim=1)
        query = self.projection_head(query)
        return query

    def forward_momentum(self, x):
```

```
        key = self.backbone_momentum(x).flatten(start_dim=1)
        key = self.projection_head_momentum(key).detach()
        return key

    def training_step(self, batch, batch_idx):
        momentum = cosine_schedule(self.current_epoch, nb_Epoch, 0.996, 1)
        update_momentum(self.backbone, self.backbone_momentum, m=momentum)
        update_momentum(self.projection_head, self.projection_head_momentum, m=momentum)
        x_query, x_key = batch[0]
        query = self.forward(x_query)
        key = self.forward_momentum(x_key)
        loss = self.criterion(query, key)
        self.log("train_loss_ssl", loss)
        self.log('train_loss', loss, on_step=True, on_epoch=True, prog_bar=False, logger=False)#Log
        return loss


    def configure_optimizers(self):
        optim = torch.optim.SGD(self.parameters(), lr=0.06)
        return optim

    def on_train_epoch_end(self):
        epoch_loss = self.trainer.callback_metrics['train_loss']
        self.train_losses.append(epoch_loss.item())
        print(f'Epoch {self.current_epoch}: Train Loss: {epoch_loss.item()}')

    def on_train_end(self):# Affiche la loss à la fin de l'entraînement.
        print("LOSS:")
        for i in self.train_losses:
            temp =f"{i:.5f}".replace('.', ',')
            print(temp )
```

In [ ]:

```
#Source:[https://saturncloud.io/blog/how-to-use-class-weights-with-focal-loss-in-pytorch-for-imbalanced-multiclass-classification/]
stop_random()
import torch.optim as optim
trainset = dataset_train_classifier
#Car les dataloaders de lightly n'ont pas l'attribut targets, donc on est obligé d'utiliser cette méthode


nb_Element = [0] * nb_Classe
i = 0
total = 0
for sous_d in os.listdir(path_Train):
    sous_d = os.path.join(path_Train, sous_d)
    nb_Element[i] = len(os.listdir(sous_d))
    i = i +1
for k in range(0, nb_Classe):
    total = total + nb_Element[k]

class_weights = []
for exemple in nb_Element:
    poid = (total-exemple) / (total)
    class_weights.append(poid)
    print(poid)


#Source:[https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html]
```

```
class_weights = torch.tensor(class_weights,dtype=torch.float32).cuda()
criterion_ = nn.CrossEntropyLoss(weight=class_weights)
print(class_weights)
```

In [ ]:

```
#Source:[https://docs.lightly.ai/self-supervised-
learning/tutorials/package/tutorial_moco_memory_bank.html]
class Classifier(pl.LightningModule):
  def __init__(self, backbone):
    super().__init__()
    self.backbone = backbone


    self.train_losses_tab = []
    self.train_accuracy_tab = []
    self.val_losses_tab = []
    self.val_accuracy_tab = []

    self.train_losses = 0
    self.train_accuracy = 0
    self.val_losses = 0
    self.val_accuracy = 0

    self.t_iteration = 0
    self.v_iteration = 0

    # freeze the backbone
    if(not(BNF)):
      deactivate_requires_grad(backbone)
    self.fc1 = nn.Linear(512, 2048)
    self.relu = nn.ReLU()
    self.fc2 = nn.Linear(2048, nb_Classe)

    self.validation_step_outputs = []
    self.criterion = criterion_

  def forward(self, x):
    x = self.backbone(x).flatten(start_dim=1)
    x = self.fc1(x)
    x = self.relu(x)
    x = self.fc2(x)
    y_hat = x
    return y_hat

  def training_step(self, batch, batch_idx):
    x, y, _ = batch
    y_hat = self.forward(x)
    loss =  self.criterion(y_hat, y)
    #Calcule et enregistrement des données utiles

    t_acc = (y_hat.argmax(dim=1) == y).float().mean()
    self.train_losses = self.train_losses + loss.item()
    self.train_accuracy = self.train_accuracy + t_acc.item()
    self.t_iteration =  self.t_iteration +1
    return loss

  def on_train_epoch_end(self):

    self.train_losses_tab.append(self.train_losses/self.t_iteration)
    self.train_accuracy_tab.append(self.train_accuracy/self.t_iteration)
```

```
        self.train_losses = 0
        self.train_accuracy = 0
        self.t_iteration = 0


    def on_train_end(self):

        self.train_losses= np.array(self.train_losses_tab)
        self.train_accuracy= np.array(self.train_accuracy_tab)

        self.val_losses= np.array(self.val_losses_tab)
        self.val_accuracy= np.array(self.val_accuracy_tab)

        #Enregistrement
        import csv
        titre = ['training_loss','validation_loss','training_accuracy','validation_accuracy']

        tableaux_complet = list(zip(self.train_losses,self.train_accuracy,self.val_losses,self.val_accuracy))
        path_save_data = './Entrainement/'+ dataset + "/Moco_Class_PLUS#"+dossier +"#"+str(nb_Epoch)
+"#" +str(BNF)+".csv"
        print(path_save_data)
        with open(path_save_data, mode='w', newline='', encoding='utf-8') as fichier:
            writer = csv.writer(fichier)
            writer.writerow(titre)
            for i in range(len(self.train_losses)):
                temp1 =f"{self.train_losses[i]:.5f}".replace('.', ',')
                temp2 =f"{self.val_losses[i]:.5f}".replace('.', ',')
                temp3 =f"{self.train_accuracy[i]:.5f}".replace('.', ',')
                temp4 =f"{self.val_accuracy[i]:.5f}".replace('.', ',')
                writer.writerow([temp1, temp2, temp3, temp4])

    def custom_histogram_weights(self):
        for name, params in self.named_parameters():
            self.logger.experiment.add_histogram(name, params, self.current_epoch)

    def validation_step(self, batch, batch_idx):
        x, y, _ = batch
        y_hat = self.forward(x)
        loss =  self.criterion(y_hat, y)
        v_acc = (y_hat.argmax(dim=1) == y).float().mean()

        self.val_losses =self.val_losses + loss.item()
        self.val_accuracy = self.val_accuracy + v_acc.item()
        self.v_iteration = self.v_iteration +1

    def on_validation_epoch_end(self):

        self.val_losses_tab.append(self.val_losses/self.v_iteration)
        self.val_accuracy_tab.append(self.val_accuracy/self.v_iteration)

        self.val_losses = 0
        self.val_accuracy = 0
        self.v_iteration = 0

    def configure_optimizers(self):
        optim = torch.optim.SGD(self.parameters(), lr=0.01, momentum=0.9)
        scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optim, T_max=self.trainer.max_epochs)
        return [optim], [scheduler]
```

In [ ]:

```
torch.set_float32_matmul_precision('medium')# Réduit légèrement la précision, mais améliore la rapidité de l'entraînement.
```

In [ ]:

```
#Entrainement du model
stop_random()
model = Moco()
print("Entrainement du model: " + Nom_Model)
trainer = pl.Trainer(max_epochs=nb_Epoch, devices=1, accelerator="gpu")
trainer.fit(model, dataloader_train_model)

torch.save(model.state_dict(), model_Path)
```

In [ ]:

```
#Chargement du model
model = Moco()
model.load_state_dict(torch.load(model_Path))
print(model_Path)
```

In [ ]:

```
#Source:[https://docs.lightly.ai/self-supervised-learning/tutorials/package/tutorial_moco_memory_bank.html]
stop_random()
import contextlib
import os
import sys

print("Entrainement du classifieur pour le model: " + Nom_Model)

if(not(BNF)):
  model.eval()
  print("Eval")
else:
  model.train()
  print("Train")

log_file_path = 'training_output.log'# PyTorch n'aime pas afficher beaucoup de texte, donc on le stocke dans un fichier à la place.
with open(log_file_path, 'w') as log_file_path:
  with contextlib.redirect_stdout(log_file_path), contextlib.redirect_stderr(log_file_path):
    classifier = Classifier(model.backbone)
    trainer = pl.Trainer(max_epochs=nb_Epoch, devices=1, accelerator="gpu")
    trainer.fit(classifier, dataloader_train_classifier, dataloader_val)

print("FINITO")


#Sauvegarde le classificateur
torch.save(classifier.state_dict(), path_Class)
```

In [ ]:

```
#Charge le classificateur
classi =  Classifier(model.backbone)
classi.load_state_dict(torch.load(path_Class))
classifier= classi
```

In [ ]:

```
stop_random()
# Partie test : va tester le modèle sur l'ensemble du dossier de test, et donner les prédictions par classe ainsi qu'une matrice de confusion.
import torch.nn.functional as F
import seaborn as sns
import pandas as pd
from sklearn.metrics import confusion_matrix
```

```
print("Model choisi:" + Nom_Model +",Avec "+ str(nb_Epoch)+" Epoch,et le dossier " +dossier)
print(model_Path)
print("NB de classe: " + str(nb_Classe))


class_names=['[0]', '[1]', '[2]', '[3]', '[4]', '[5]', '[6]' ,'[7]']
if(nb_Classe == 8):
   class_names=['[0]', '[1]', '[2]', '[3]', '[4]', '[5]', '[6]' ,'[7]']
else:
   class_names=['[0]', '[1]', '[2]', '[3]', '[4]', '[5]', '[6]']
image_transform = transform
total = 0
nb_true_pred = 0
total_AC = 0
taille_dossier= [0 for x in range(nb_Classe)]

predictions= [[0 for x in range(nb_Classe)] for y in range(nb_Classe)]


for j in range(0,nb_Classe):
   image_folder = dataset +"_Full/test/"

   image_folder = image_folder + "[" + str(j) + "]"
   nb_true_pred = 0
   taille_dossier[j] = len(os.listdir(image_folder))
   classifier.eval()

   for filename in os.listdir(image_folder):
      if filename.endswith(".png"):
         image_path = os.path.join(image_folder, filename)

         image = Image.open(image_path).convert('RGB')
         image_tensor = val_transforms(image)
         image_tensor = image_tensor.unsqueeze(0)

         with torch.no_grad():

            output = classifier(image_tensor)
            probabilities = F.softmax(output, dim=1)

            _, predicted_class = torch.max(output, 1)
            predicted_class_name = class_names[predicted_class.item()]

            confidence = torch.max(probabilities).item() * 100
            true_class=os.path.basename(os.path.dirname(image_path))
            temp = predicted_class_name
            temp = temp.replace('[','')
            temp = temp.replace(']','')
            predictions[j][int(temp)] = predictions[j][int(temp)] +1
            if(true_class == predicted_class_name):
               nb_true_pred = nb_true_pred +1
   print("Classe:"+ str(j) + f" % de bonne prédiction:" + str((nb_true_pred/taille_dossier[j])* 100) +"%" )
   total = total + taille_dossier[j]
   total_AC = total_AC + nb_true_pred

print("Accuray général:" +str(( total_AC / total)*100)+"%")
```

```
for j in range(0,nb_Classe):
   for i in range (0,nb_Classe):
      predictions[j][i] = (predictions[j][i]/taille_dossier[j])*100
matrice_C = np.array(predictions)
plt.figure(figsize=(6, 6))
sns.heatmap(matrice_C, annot=True, fmt='.2f', cmap='Blues')
plt.xlabel('Prédiction')
plt.ylabel('Vrai classe')
plt.title('Matrice')
image_Path = 'M_image_'+dataset+'/'+str(nb_Epoch) +"_" + "EPOCH"+"_" + dossier +"_"+
Nom_Model+"_PLUS_"+str(BNF)+".png"
print(image_Path)
plt.savefig(image_Path)
plt.show()
```

In [ ]:

```
# Utilisé pour copier plus simplement les résultats dans un fichier Excel.
total2=0
for j in range(0,nb_Classe):
   total2 = total2 + predictions[j][j]
   temp =f"{predictions[j][j]:.2f}".replace('.', ',')
   print(temp)
temp =f"{(total_AC / total)*100:.2f}".replace('.', ',')

print(temp)
temp =f"{total2/nb_Classe:.2f}".replace('.', ',')
print(temp)
```

12.4.8 MoCo 34

```
# Chaque bloc possède un lien vers la source originale, formaté comme ceci : #Source: []

# Rappels avant de lancer le programme :
#   - bien vérifier le dataset et le dossier utilisé.
#   - bien vérifier les paramètres et les dossiers de sauvegarde.
#   - faire attention de ne pas lancer le modèle alors qu'il a déjà été entraîné.
#   - Si le premier epoch est très lent, c'est normal, il suffit d'attendre un peu que les données soient chargées.

# Cette cellule contient les imports nécessaires au bon fonctionnement du code.


import torch
import torchvision
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torchvision.transforms as transforms
from torchsummary import summary
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision import datasets, models

import os
import random
import copy
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
```

```
import seaborn as sns
import pandas as pd

from sklearn.metrics import confusion_matrix
import pytorch_lightning as pl

from lightly.data import LightlyDataset
from lightly.loss import NTXentLoss
from lightly.models.modules.heads import MoCoProjectionHead
from lightly.models.utils import (
    batch_shuffle,
    batch_unshuffle,
    deactivate_requires_grad,
    update_momentum,
)
#Import spécifique à chaque model
from lightly.loss import NTXentLoss
from lightly.models.modules import MoCoProjectionHead
from lightly.models.utils import deactivate_requires_grad, update_momentum
from lightly.transforms.moco_transform import MoCoV2Transform
from lightly.utils.scheduler import cosine_schedule


torch.__version__
```

In [ ]:

```
# Fonction utilisée pendant le programme pour donner une seed aux différentes fonctions aléatoires.
# A mettre au début de chaque cellule.
# Source : [https://pytorch.org/docs/stable/notes/randomness.html]
def stop_random():
    seed=621
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    random.seed(seed)
    np.random.seed(seed)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False
```

In [ ]:

```
# Configuration générale. Attention à bien choisir les bons paramètres et les bon emplacement de
sauvegarde.

stop_random()
nb_Epoch = 100
batch_Size = 128
taille_Image = 28
augmentation=False
num_workers = 4
dossier="Quarter"
dataset="Blood"
BNF = True
#Nomenclature:Nom_Model#Dossier#Epoch#BNF
Nom_Model = "MoCo"
model_Path = './PytorchModelV3/'+"MoCo_34_Model#"+"Full" +"#"+dataset+str(nb_Epoch) +"#"
+"FALSE"+".pth"
path_Class = './PytorchModelV3/'+"MoCo_34_Class#"+dossier +"#"+dataset+str(nb_Epoch) +"#"
+str(BNF)+".pth"
path_Train_SSL_M = dataset+"_SSL"
path_Train = dataset+"_"+dossier+"/Dataset/train"
```

```
path_val =  dataset+"_"+dossier+"/Dataset/val"
nb_Classe = len(next(os.walk(dataset+"_"+dossier+"/test"))[1])
print("Model choisi MoCo, avec le dossier: " +dossier+ ", avec "+ str(nb_Epoch) + " epoch")
print("Emplacement: " +model_Path)
print("Emplacement: " +path_Class)
```

In [ ]:

```
#Source:[https://docs.lightly.ai/self-supervised-
learning/tutorials/package/tutorial_moco_memory_bank.html]
#Source:[https://docs.lightly.ai/self-supervised-
learning/tutorials/package/tutorial_moco_memory_bank.html]

transform = MoCoV2Transform(input_size=28,gaussian_blur=0.0,)

train_classifier_transforms = torchvision.transforms.Compose(
    [
        torchvision.transforms.ToTensor(),
    ])

if(augmentation):# Si l'on a besoin de faire de l'augmentation de données.
    print("augmentation:")
    train_classifier_transforms = transforms.Compose([
    transforms.RandomResizedCrop(size=28, scale=(0.3, 1.0), ratio=(1, 1)),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
])


val_transforms = torchvision.transforms.Compose(
    [
        torchvision.transforms.ToTensor(),
    ])



dataset_train_model = LightlyDataset(input_dir=path_Train_SSL_M, transform=transform)
dataset_train_classifier = LightlyDataset(input_dir=path_Train, transform=train_classifier_transforms)
dataset_val = LightlyDataset(input_dir=path_val, transform=val_transforms)

print(train_classifier_transforms)
```

In [ ]:

```
#Source:[https://docs.lightly.ai/self-supervised-
learning/tutorials/package/tutorial_moco_memory_bank.html]

# Explication sur les paramètres des dataloaders :
#     shuffle --> mélange les images, à utiliser surtout pour les données d'entraînement du modèle SSL.
#     drop_last --> sinon, parfois, l'entraînement peut planter à un epoch.

stop_random()
dataloader_train_model = torch.utils.data.DataLoader(
    dataset_train_model,
    batch_size=batch_Size,
    shuffle=True,
    drop_last=True,
    num_workers=num_workers,
    persistent_workers=True,
)

dataloader_train_classifier = torch.utils.data.DataLoader(
```

```
    dataset_train_classifier,
    batch_size=batch_Size,
    shuffle=True,
    drop_last=True,
    num_workers=num_workers,
    persistent_workers=True,
)

dataloader_val = torch.utils.data.DataLoader(
    dataset_val,
    batch_size=batch_Size,
    shuffle=False,
    drop_last=False,
    num_workers=num_workers,
    persistent_workers=True,
)
```

In [ ]:

```
#Source:[https://docs.lightly.ai/self-supervised-learning/examples/moco.html]
stop_random()
class Moco(pl.LightningModule):
    def __init__(self):
        super().__init__()
        resnet = torchvision.models.resnet34()
        self.backbone = nn.Sequential(*list(resnet.children())[:-1])
        self.projection_head = MoCoProjectionHead(512, 1024, 1024)

        self.backbone_momentum = copy.deepcopy(self.backbone)
        self.projection_head_momentum = copy.deepcopy(self.projection_head)

        deactivate_requires_grad(self.backbone_momentum)
        deactivate_requires_grad(self.projection_head_momentum)

        self.criterion = NTXentLoss(memory_bank_size=(4096, 1024))
        self.train_losses = []

    def forward(self, x):
        query = self.backbone(x).flatten(start_dim=1)
        query = self.projection_head(query)
        return query

    def forward_momentum(self, x):
        key = self.backbone_momentum(x).flatten(start_dim=1)
        key = self.projection_head_momentum(key).detach()
        return key

    def training_step(self, batch, batch_idx):
        momentum = cosine_schedule(self.current_epoch, nb_Epoch, 0.996, 1)
        update_momentum(self.backbone, self.backbone_momentum, m=momentum)
        update_momentum(self.projection_head, self.projection_head_momentum, m=momentum)
        x_query, x_key = batch[0]
        query = self.forward(x_query)
        key = self.forward_momentum(x_key)
        loss = self.criterion(query, key)
        self.log("train_loss_ssl", loss)
        self.log('train_loss', loss, on_step=True, on_epoch=True, prog_bar=False, logger=False)#Log
        return loss


    def configure_optimizers(self):
```

```
        optim = torch.optim.SGD(self.parameters(), lr=0.06)
        return optim

    def on_train_epoch_end(self):
        epoch_loss = self.trainer.callback_metrics['train_loss']
        self.train_losses.append(epoch_loss.item())
        print(f'Epoch {self.current_epoch}: Train Loss: {epoch_loss.item()}')

    def on_train_end(self):# Affiche la loss à la fin de l'entraînement.
        print("LOSS:")
        for i in self.train_losses:
            temp =f"{i:.5f}".replace('.', ',')
            print(temp )
```

In [ ]:

```
#Source:[https://saturncloud.io/blog/how-to-use-class-weights-with-focal-loss-in-pytorch-for-imbalanced-multiclass-classification/]
stop_random()
import torch.optim as optim
trainset = dataset_train_classifier
#Car les dataloaders de lightly n'ont pas l'attribut targets, donc on est obligé d'utiliser cette méthode


nb_Element = [0] * nb_Classe
i = 0
total = 0
for sous_d in os.listdir(path_Train):
    sous_d = os.path.join(path_Train, sous_d)
    nb_Element[i] = len(os.listdir(sous_d))
    i = i +1
for k in range(0, nb_Classe):
    total = total + nb_Element[k]

class_weights = []
for exemple in nb_Element:
    poid = (total-exemple) / (total)
    class_weights.append(poid)
    print(poid)


#Source:[https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html]

class_weights = torch.tensor(class_weights,dtype=torch.float32).cuda()
criterion_ = nn.CrossEntropyLoss(weight=class_weights)
print(class_weights)
```

In [ ]:

```
#Source:[https://docs.lightly.ai/self-supervised-learning/tutorials/package/tutorial_moco_memory_bank.html]
class Classifier(pl.LightningModule):
    def __init__(self, backbone):
        super().__init__()
        self.backbone = backbone

        self.train_losses_tab = []
        self.train_accuracy_tab = []
        self.val_losses_tab = []
        self.val_accuracy_tab = []

        self.train_losses = 0
        self.train_accuracy = 0
```

```
        self.val_losses = 0
        self.val_accuracy = 0

        self.t_iteration = 0
        self.v_iteration = 0

        # freeze the backbone
        if(not(BNF)):
            deactivate_requires_grad(backbone)
        self.fc1 = nn.Linear(512, 2048)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(2048, nb_Classe)

        self.validation_step_outputs = []
        self.criterion = criterion_

    def forward(self, x):
        x = self.backbone(x).flatten(start_dim=1)
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        y_hat = x
        return y_hat

    def training_step(self, batch, batch_idx):
        x, y, _ = batch
        y_hat = self.forward(x)
        loss =  self.criterion(y_hat, y)
        #Calcule et enregistrement des données utiles

        t_acc = (y_hat.argmax(dim=1) == y).float().mean()
        self.train_losses = self.train_losses + loss.item()
        self.train_accuracy = self.train_accuracy + t_acc.item()
        self.t_iteration =  self.t_iteration +1
        return loss

    def on_train_epoch_end(self):

        self.train_losses_tab.append(self.train_losses/self.t_iteration)
        self.train_accuracy_tab.append(self.train_accuracy/self.t_iteration)
        self.train_losses = 0
        self.train_accuracy = 0
        self.t_iteration = 0


    def on_train_end(self):

        self.train_losses= np.array(self.train_losses_tab)
        self.train_accuracy= np.array(self.train_accuracy_tab)

        self.val_losses= np.array(self.val_losses_tab)
        self.val_accuracy= np.array(self.val_accuracy_tab)

        #Enregistrement
        import csv
        titre = ['training_loss','validation_loss','training_accuracy','validation_accuracy']

        tableaux_complet = list(zip(self.train_losses,self.train_accuracy,self.val_losses,self.val_accuracy))
```

```
        path_save_data = './Entrainement/'+ dataset + "/Moco_34_Class#"+dossier +"#"+str(nb_Epoch) +"#"
+str(BNF)+".csv"
        print(path_save_data)
        with open(path_save_data, mode='w', newline='', encoding='utf-8') as fichier:
            writer = csv.writer(fichier)
            writer.writerow(titre)
            for i in range(len(self.train_losses)):
                temp1 =f"{self.train_losses[i]:.5f}".replace('.', ',')
                temp2 =f"{self.val_losses[i]:.5f}".replace('.', ',')
                temp3 =f"{self.train_accuracy[i]:.5f}".replace('.', ',')
                temp4 =f"{self.val_accuracy[i]:.5f}".replace('.', ',')
                writer.writerow([temp1, temp2, temp3, temp4])

    def custom_histogram_weights(self):
        for name, params in self.named_parameters():
            self.logger.experiment.add_histogram(name, params, self.current_epoch)

    def validation_step(self, batch, batch_idx):
        x, y, _ = batch
        y_hat = self.forward(x)
        loss =  self.criterion(y_hat, y)
        v_acc = (y_hat.argmax(dim=1) == y).float().mean()

        self.val_losses =self.val_losses + loss.item()
        self.val_accuracy = self.val_accuracy + v_acc.item()
        self.v_iteration = self.v_iteration +1

    def on_validation_epoch_end(self):

        self.val_losses_tab.append(self.val_losses/self.v_iteration)
        self.val_accuracy_tab.append(self.val_accuracy/self.v_iteration)

        self.val_losses = 0
        self.val_accuracy = 0
        self.v_iteration = 0

    def configure_optimizers(self):
        optim = torch.optim.SGD(self.parameters(), lr=0.01, momentum=0.9)
        scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optim, T_max=self.trainer.max_epochs)
        return [optim], [scheduler]
```

In [ ]:

```
torch.set_float32_matmul_precision('medium')# Réduit légèrement la précision, mais améliore la rapidité
de l'entraînement.
```

In [ ]:

```
#Entrainement du model
stop_random()
model = Moco()
print("Entrainement du model: " + Nom_Model)
trainer = pl.Trainer(max_epochs=nb_Epoch, devices=1, accelerator="gpu")
trainer.fit(model, dataloader_train_model)

torch.save(model.state_dict(), model_Path)
```

In [ ]:

```
#Chargement du model
model = Moco()
model.load_state_dict(torch.load(model_Path))
print(model_Path)
```

In [ ]:

```
#Source:[https://docs.lightly.ai/self-supervised-learning/tutorials/package/tutorial_moco_memory_bank.html]
stop_random()
import contextlib
import os
import sys

print("Entrainement du classifieur pour le model: " + Nom_Model)

if(not(BNF)):
  model.eval()
  print("Eval")
else:
  model.train()
  print("Train")

log_file_path = 'training_output.log'# PyTorch n'aime pas afficher beaucoup de texte, donc on le stocke dans un fichier à la place.
with open(log_file_path, 'w') as log_file_path:
  with contextlib.redirect_stdout(log_file_path), contextlib.redirect_stderr(log_file_path):
    classifier = Classifier(model.backbone)
    trainer = pl.Trainer(max_epochs=nb_Epoch, devices=1, accelerator="gpu")
    trainer.fit(classifier, dataloader_train_classifier, dataloader_val)

print("FINITO")


#Sauvegarde le classificateur
torch.save(classifier.state_dict(), path_Class)
```

In [ ]:

```
#Charge le classificateur
classi =  Classifier(model.backbone)
classi.load_state_dict(torch.load(path_Class))
classifier= classi
```

In [ ]:

```
stop_random()
# Partie test : va tester le modèle sur l'ensemble du dossier de test, et donner les prédictions par classe ainsi qu'une matrice de confusion.
import torch.nn.functional as F
import seaborn as sns
import pandas as pd
from sklearn.metrics import confusion_matrix


print("Model choisi:" + Nom_Model +",Avec "+ str(nb_Epoch)+" Epoch,et le dossier " +dossier)
print(model_Path)
print("NB de classe: " + str(nb_Classe))


class_names=['[0]', '[1]', '[2]', '[3]', '[4]', '[5]', '[6]' ,'[7]']
if(nb_Classe == 8):
  class_names=['[0]', '[1]', '[2]', '[3]', '[4]', '[5]', '[6]' ,'[7]']
else:
  class_names=['[0]', '[1]', '[2]', '[3]', '[4]', '[5]', '[6]']
image_transform = transform

total = 0
nb_true_pred = 0
total_AC = 0
```

```
taille_dossier= [0 for x in range(nb_Classe)]

predictions= [[0 for x in range(nb_Classe)] for y in range(nb_Classe)]


for j in range(0,nb_Classe):
   image_folder = dataset +"_Full/test/"

   image_folder = image_folder + "[" + str(j) + "]"
   nb_true_pred = 0
   taille_dossier[j] = len(os.listdir(image_folder))
   classifier.eval()

   for filename in os.listdir(image_folder):
      if filename.endswith(".png"):
         image_path = os.path.join(image_folder, filename)

         image = Image.open(image_path).convert('RGB')
         image_tensor = val_transforms(image)
         image_tensor = image_tensor.unsqueeze(0)

         with torch.no_grad():

            output = classifier(image_tensor)
            probabilities = F.softmax(output, dim=1)

            _, predicted_class = torch.max(output, 1)
            predicted_class_name = class_names[predicted_class.item()]

            confidence = torch.max(probabilities).item() * 100
            true_class=os.path.basename(os.path.dirname(image_path))
            temp = predicted_class_name
            temp = temp.replace('[','')
            temp = temp.replace(']','')
            predictions[j][int(temp)] = predictions[j][int(temp)] +1
            if(true_class == predicted_class_name):
               nb_true_pred = nb_true_pred +1
   print("Classe:"+ str(j) + f" % de bonne prédiction:" + str((nb_true_pred/taille_dossier[j])* 100) +"%" )
   total = total + taille_dossier[j]
   total_AC = total_AC + nb_true_pred

print("Accuray général:" +str(( total_AC / total)*100)+"%")

for j in range(0,nb_Classe):
   for i in range (0,nb_Classe):
      predictions[j][i] = (predictions[j][i]/taille_dossier[j])*100
matrice_C = np.array(predictions)
plt.figure(figsize=(6, 6))
sns.heatmap(matrice_C, annot=True, fmt='.2f', cmap='Blues')
plt.xlabel('Prédiction')
plt.ylabel('Vrai classe')
plt.title('Matrice')
image_Path = 'M_image_'+dataset+'/'+str(nb_Epoch) +"_" + "EPOCH"+"_" + dossier +"_"+
Nom_Model+"_50_"+str(BNF)+".png"
print(image_Path)
plt.savefig(image_Path)
plt.show()
```

In [ ]:

```
# Utilisé pour copier plus simplement les résultats dans un fichier Excel.
```

```
total2=0
for j in range(0,nb_Classe):
   total2 = total2 + predictions[j][j]
   temp =f"{predictions[j][j]:.2f}".replace('.', ',')
   print(temp)
temp =f"{(total_AC / total)*100:.2f}".replace('.', ',')

print(temp)
temp =f"{total2/nb_Classe:.2f}".replace('.', ',')
print(temp)
```