

Segmentation médicale d'un cœur

Samy Madiou, Nadir Mohammed Bouakel, Soufiane Ham, Imad Idrissi

Apprentissage profond pour la vision par ordinateur (MTI881)

Travail remis au professeur Jose Dolz, 22 Décembre 2022

École de Technologie Supérieure (ÉTS), Montréal, QC

Abstract

Ce travail est réalisé dans le cadre du cours de maîtrise Apprentissage profond pour la vision par ordinateur (MTI881) et il consiste à créer un réseau de neurones profond capable de segmenter des images médicales de cœur du challenge ACDC (Automated Cardiac Diagnosis Challenge) [1] selon le myocarde, le ventricule gauche et le ventricule droite. Le but est de développer le modèle le plus performant quant à la précision de sa segmentation en explorant divers réseaux de neurones à convolution développés pour la segmentation d'images biomédicales. La finalité de ce projet fut une compétition avec d'autres équipes dans laquelle les meilleurs modèles de chaque équipe ont été testés sur des images. Le classement de la compétition a été déterminé selon le critère de la performance en évaluant les métriques suivantes: le coefficient Dice, la distance de Hausdorff (HDD) et la distance de surface moyenne (ASD).

1 Introduction

L'imagerie médicale joue un rôle essentiel dans le domaine de la santé. En effet, elle permet aux professionnels de la santé de mieux analyser et diagnostiquer leurs patients. L'étude des images médicales dépend principalement de l'interprétation visuelle et de l'expérience des radiologues. Cependant, cela peut être un exercice fastidieux pour les médecins. Par exemple, les images peuvent avoir une mauvaise qualité et il peut être difficile de situer une tumeur. Cela étant dit, l'utilisation de systèmes assistés par ordinateur peut être très utile et intéressante afin d'éviter ces problèmes. La segmentation d'image est une technique critique de l'analyse d'image. Son objectif est d'extraire des informations d'une image quelconque. Dans le domaine médical, les segments d'images correspondent souvent à différentes catégories de tissus, d'organes, ou d'autres structures biologiques du corps humain. Ainsi, des systèmes basés sur des modèles d'apprentissage machine profond peuvent assister les spécialistes en détectant les zones potentielles de tumeur avec un certain pourcentage erreur. Le présent rapport synthétise le travail effectué dans le cadre du cours MTI881

dispensé à l’École de Technologie Supérieure de Montréal. Le but de l’exercice est de mettre en oeuvre le meilleur réseau de neurones pour la segmentation d’images de coeur. Pour être plus précis, l’objectif de ce projet est de segmenter le coeur en trois parties. Ces parties sont le ventricule droit (VD), le ventricule gauche (VG) et le myocarde(MYO).

2 Travaux connexes

Famille des UNet: Le modèle UNet est un des modèles les plus utilisés pour la segmentation d’images médicales due à sa simplicité. De plus, il existe plusieurs variantes de ce modèle comme le ResidualUNet, le TransUNet [1] et le SwinUNet [2]. Ces deux derniers modèles font partie des plus performants pour la segmentation d’imagerie médicale. Pour être plus précis, le modèle TransUNet proposé par Chen et al. [1] le SwinUNet de Cao et al. [2] ont respectivement terminé en troisième et deuxième position pour le challenge ACDC. [3] [4].

3 Méthodologie

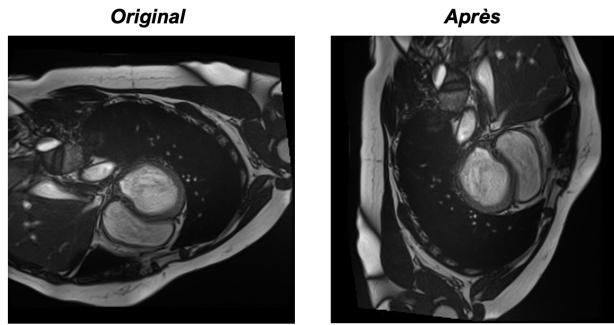
Beaucoup d’options se sont présentées pour le choix du modèle d’apprentissage pour le challenge. Cependant, il fallait d’abord s’intéresser à la nature des images en entrée avant de choisir un modèle. Les images sont de taille 256x256 et elles sont en 2D. De plus, notre étiquette est constituée de quatre classes. Ces classes sont [Arrière-plan, VG, VD, MYO], où l’arrière-plan à des composants diffère des autres classes. Par ailleurs, l’ensemble de données d’entraînement contient 1208 images et l’ensemble de validation en contient 90.

3.1 Augmentation des données

L’augmentation de données est une technique très populaire dans le domaine de l’apprentissage automatique. Elle est utilisée pour augmenter la quantité et la qualité des données disponibles pour l’entraînement d’un modèle. En effet, elle consiste à générer de nouvelles données à partir de celles existantes, en utilisant différentes techniques de transformation et de modification. Un des principaux avantages de l’augmentation de données est justement le fait qu’on évite la généralisation. En effet, l’augmentation de données aide à réduire le risque de surapprentissage en fournissant au modèle une variété supplémentaire de données d’entraînement. Également, en augmentant les données, nous pouvons entraîner des modèles d’apprentissage machine de manière plus efficace afin d’obtenir de meilleurs résultats. Cela étant dit, nous avons donc décidé d’utiliser cette technique, car nous avions peu de données. Nous avions en notre possession uniquement 1208 images pour l’entraînement, ce qui est clairement insuffisant. Pour ce faire, nous avions utilisé la librairie python ”Albumentation” [2], qui offre diverses techniques et transformations pour l’augmentation de données.

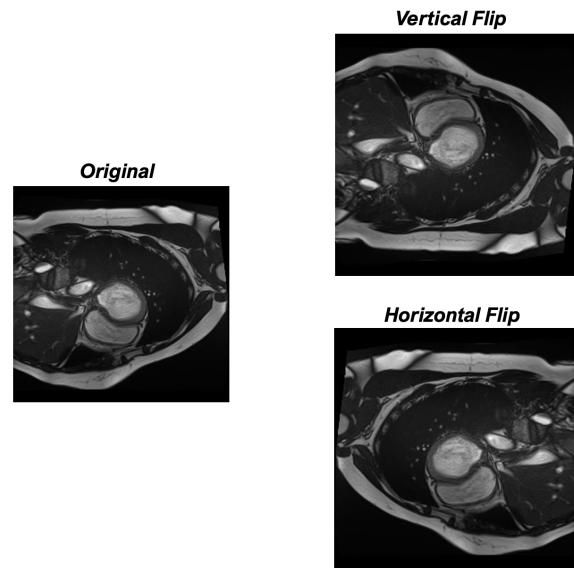
RandomRotate90 : Elle permet de faire pivoter aléatoirement une image de 0, 90, 180 ou 270 degrés.

Figure 1: Rotation Aléatoire



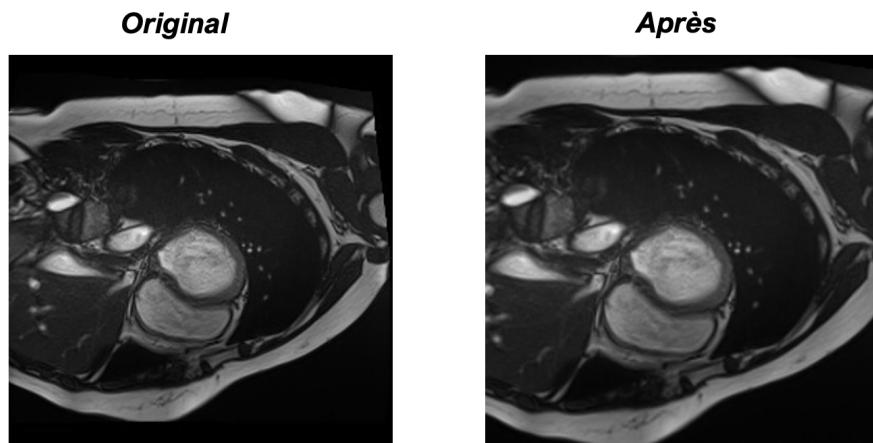
VerticalFlip et HorizontalFlip : Ces deux fonctions retournent l'image par rapport à l'axe vertical et horizontal comme l'effet d'un miroir.

Figure 2: Vertical and Horizontal Flip



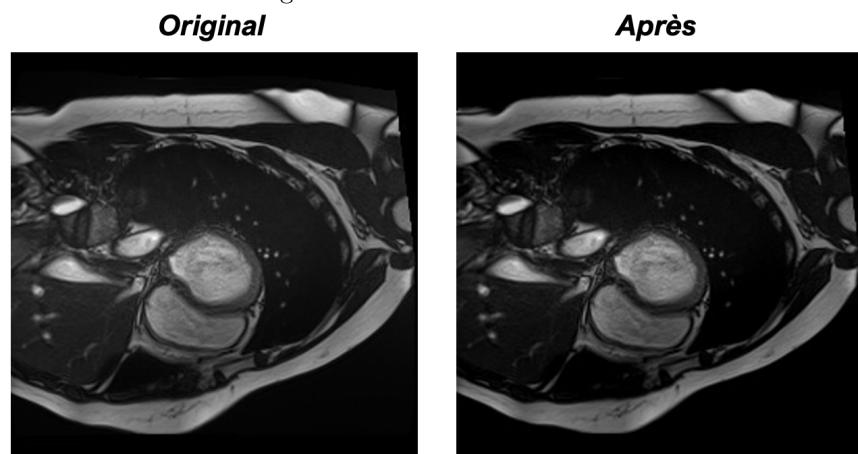
CenterCrop : La fonction CenterCrop (recadrage au centre) est utilisée pour recadrer une image en ne gardant que la partie centrale de celle-ci. Cela signifie que les bords de l'image sont coupés et que seule la partie centrale de l'image est conservée. Nous avons décidé d'enlever 32 pixels à la longueur et la largeur de l'image.

Figure 3: Center Crop



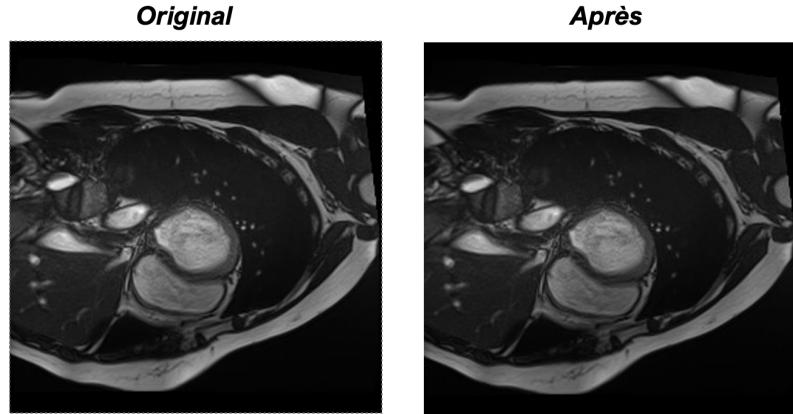
RandomBrightness : Cette transformation est utilisée pour ajuster la luminosité (augmenter ou diminuer) d'une image de manière aléatoire.

Figure 4: Luminosité aléatoire



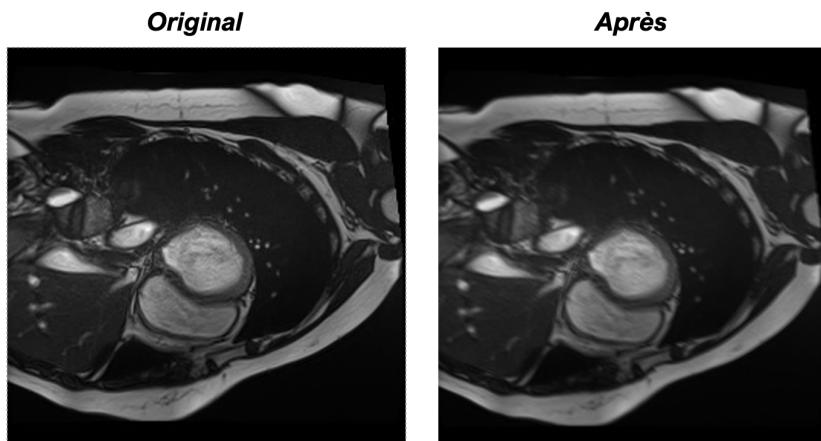
RandomContrast : Cette transformation est utilisée pour ajuster le contraste (augmenter ou diminuer) d'une image de manière aléatoire.

Figure 5: Contraste aléatoire



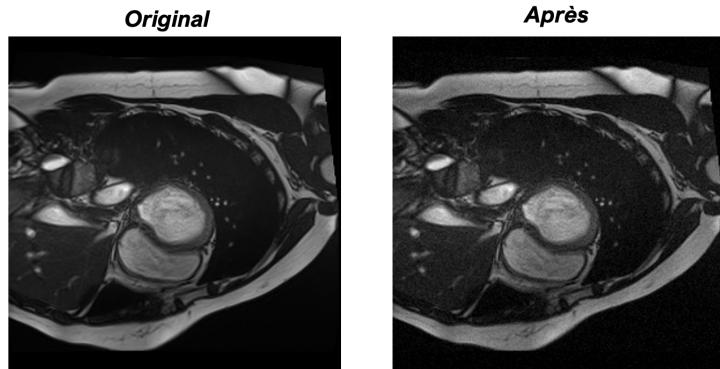
MotionBlur : Cette transformation permet d'ajouter un flou de mouvement à une image.

Figure 6: Motion Blur



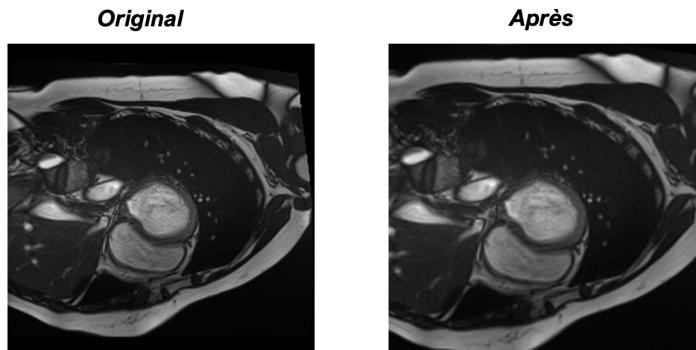
GaussNoise : Cette transformation permet d'ajouter du bruit gaussien à une image. Le bruit gaussien est un type de bruit aléatoire qui suit une distribution normale.

Figure 7: Gaussian Noise



CoarseDropout : Cette fonction masque différentes régions aléatoires dans l'image.

Figure 8: Coarse Dropout



Ces transformations sont donc appliquées sur les images et leurs étiquettes de façon aléatoire avec une probabilité de 0,4. Il y a donc 40% de chance qu'on applique chaque transformation sur les images. Ainsi, grâce à l'augmentation de donnée, nous avons pu entraîner le modèle avec près de 5000 images.

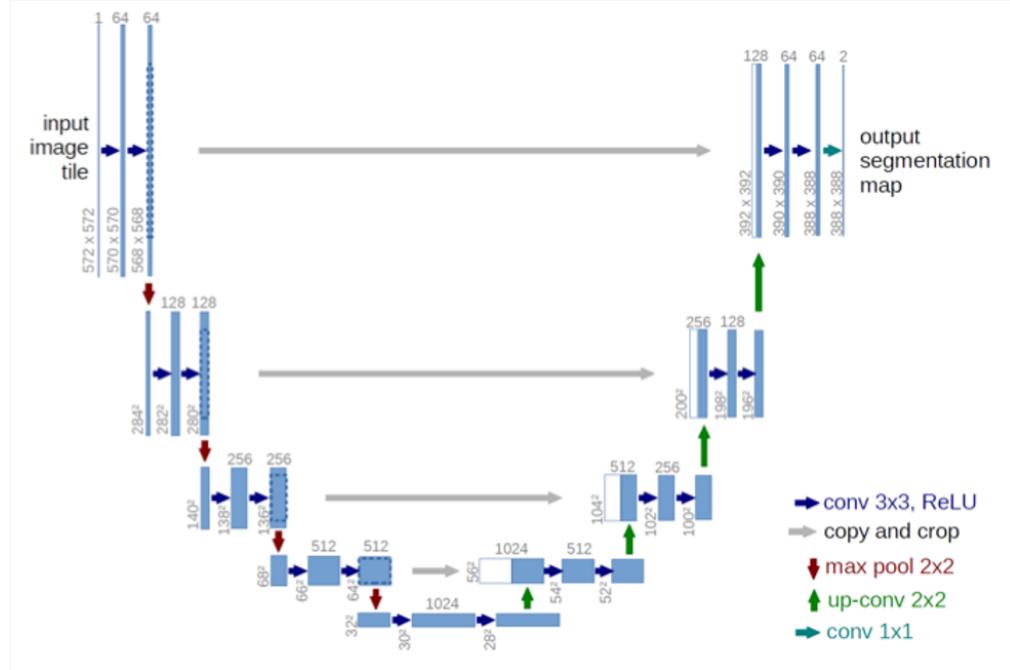
3.2 Architecture des modèles

Voici les architectures des modèles que nous avons développés ou que nous avons considéré développer:

3.2.1 UNet

C'est un modèle qui a un peu plus de 30 millions de paramètres (selon notre configuration). Il a été développé par Olaf Ronneberger, Philipp Fischer, et Thomas Brox en 2015. Le modèle est composé d'un encodeur et un décodeur. Dans le bloc encodeur, on retrouve une série de couches convolutionnelles qui permettent de réduire les dimensions de l'image reçue en entrée. Le bloc décodeur contient des couches de suréchantillonage ou "upsampling" et une série de couches convolutionnelles. [5]

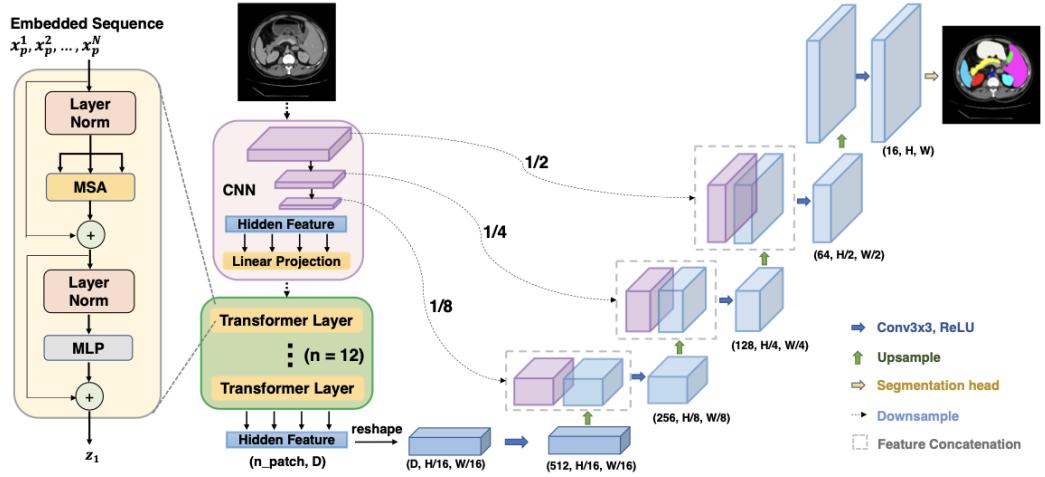
Figure 9: Architecture du modèle UNet



3.2.2 TransUNet

L'architecture du TransUNet ressemble beaucoup à celle du UNet. Ce modèle contient aussi des blocs encodeur et décodeur et des connexions de sauts entre les couches d'encodeur/décodeur. Contrairement au UNet, il y a une traduction d'image à image qui est faite avant de prédire le masque de segmentation. L'image est traduite dans un domaine synthétique où les objets sont plus clairement visibles et plus faciles à segmenter.

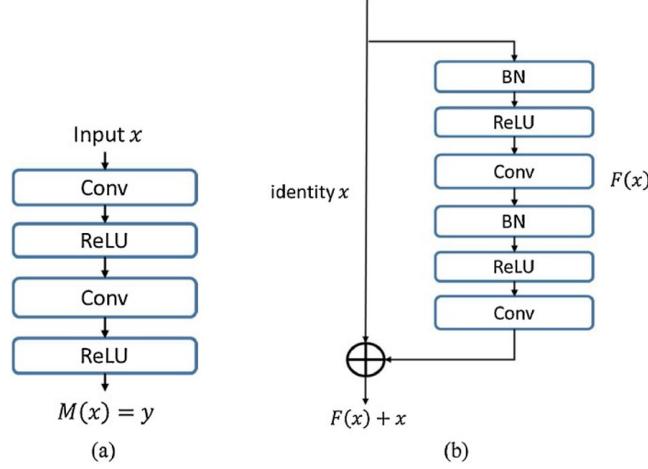
Figure 10: Architecture du modèle TransUNet



3.2.3 ResidualUNet

Encore une fois, le ResidualUNet n'est pas bien différent du UNet. Ce qui différencie les deux modèles est que dans le ResidualUNet, on retrouve des couches résiduelles, ce qui permet au modèle d'apprendre le résiduel entre l'entrée et la sortie de la couche. Ces couches résiduelles permettent aussi d'éviter le problème de la disparition des gradients qui est un phénomène qui peut se produire durant l'entraînement où les gradients des paramètres par rapport à la fonction de coût tendent vers le zéro. Cela peut se produire lorsque les paramètres sont initialisés avec de grandes valeurs ou lorsque l'architecture du réseau comporte un grand nombre de couches, ce qui est le cas avec le ResidualUNet qui contient un peu plus de 70 millions de paramètres (selon notre configuration). En bref, nous avons seulement développé et testé le UNet et le ResidualUNet dû à la limite de temps pour développer un modèle fonctionnel et assez performant pour tenter de gagner le challenge ACDC.

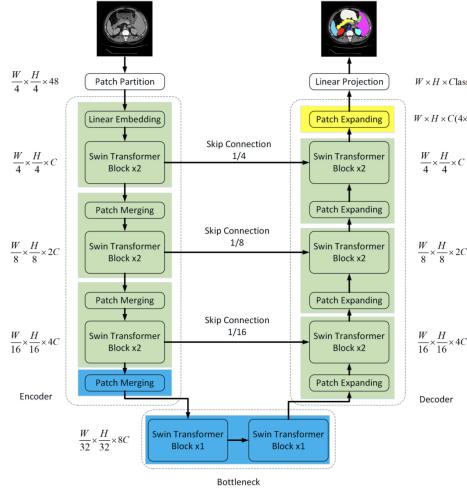
Figure 11: Architecture du modèle ResidualUNet



3.2.4 SwinUNet

La différence entre le SwinUNet et le UNet est que dans le SwinUNet une classification des images avec une approche d'apprentissage autosupervisé est effectuée avant de prédire le masque de segmentation.

Figure 12: Architecture du modèle SwinUNet



3.3 Hyperparamètres

Les hyperparamètres sont des paramètres qui permettent de contrôler le processus d'apprentissage. Ils sont choisis par les développeurs avant de commencer l'entraînement du modèle. D'ailleurs, il est crucial de choisir les bonnes valeurs pour ces derniers afin d'avoir les meilleures performances du modèle sur les images qu'il n'a jamais vu.

Dans notre cas, pour configurer notre modèle on a utilisé des hyperparamètres à plusieurs niveaux. Tout d'abord, nous avons le "batch size" qui permet de déterminer le nombre d'échantillons d'image à utiliser par itération durant la phase d'entraînement et de validation. Le modèle est alors entraîné sur un batch de données à la fois. Après plusieurs tentatives, nous avons remarqué que plus le batch size est grand, plus le modèle est capable de traiter rapidement de grands lots d'images, mais cela risque d'entraîner une instabilité dans la convergence et une augmentation d'erreur du modèle. Cependant, nous avons utilisé un batch size à la fois assez grand pour permettre une convergence rapide et assez petit pour éviter l'augmentation d'erreur afin d'augmenter nos performances. Après avoir testé différentes tailles de lot pour l'entraînement et la validation, nous avons finalement choisi une taille de 8-12 images par lot pour l'entraînement et un lot de 3-6 images pour la validation.

D'autre part, il fallait aussi trouver les bons réglages pour notre fonction de coût. Tout d'abord, notre fonction de perte est une combinaison de deux fonctions de coût qui ont chacune un poids w . Ce poids permet d'accorder plus de pénalisation au modèle lorsque le modèle effectue de mauvaises prédictions. Dans notre situation, on a voulu augmenter la pénalisation sur le facteur du **Dice Loss** afin de concentrer le modèle sur l'aspect de similitudes au masque de segmentation. Après plusieurs réglages de poids, nous avons choisi les poids suivants $w_{DiceLoss} = 0.6$ et $w_{CE} = 0.4$. Ces réglages ont changé de manière considérable la performance sur notre modèle UNet et ResidualUNet. Les modèles réalisaient environ 83-85% de Dice moyen sur les images de test.

Ensuite, il fallait choisir un taux d'apprentissage. Le taux d'apprentissage est le réglage le plus important dans l'apprentissage machine. En effet, il permet de contrôler la vitesse à laquelle le modèle apprend sur les images d'entraînement. Le choix de ce dernier est crucial, car il peut grandement impacter l'apprentissage du modèle. Un taux trop grand peut faire en sorte que le modèle ne converge jamais pour minimiser les pertes. Sinon, un taux trop petit risque de prendre beaucoup trop de temps pour converger. De plus, avoir un très petit taux risque de pousser le modèle à rester bloquer sur un minimum local. Par conséquent, il faut choisir un taux d'apprentissage qui soit à la fois assez élevé pour converger rapidement, mais assez petit pour éviter pour bien entraîner le modèle. Cependant, il n'y a pas de solution miracle pour trouver le bon taux d'apprentissage, car ce dernier dépend de plusieurs facteurs comme la complexité du modèle, la qualité des images d'entraînement, etc. La meilleure stratégie pour trouver le taux optimal est l'essai/erreur jusqu'à trouver le taux qui offre les meilleures performances. Après avoir utilisé plusieurs techniques d'optimisation du taux d'apprentissage, on a abouti à un $lr=0.001$

avec l'optimisateur ***Adam***.

En résumé, les hyperparamètres sont des paramètres très importants pour l'apprentissage d'un modèle. Ces derniers ont un gros impact sur la performance de notre modèle. Pour trouver les meilleures combinaisons de paramètres, il suffit d'effectuer plusieurs tests avec différents réglages, pour avoir un modèle performant sur les images de l'ensemble de tests. C'était la phase la plus longue dans le projet, et selon nous, nous aurions pu avoir de meilleures performances avec des réglages différents.

3.4 Fonctions de coût

Pour évaluer notre modèle d'apprentissage, il faut utiliser une fonction de coût qui mesure l'erreur du modèle en comparant les prédictions du modèle aux images GT. Cette fonction permet d'ajuster les poids du modèle de manière à minimiser l'erreur. D'ailleurs, la fonction de coût est utilisée avec la descente de gradient pour propager l'erreur dans le modèle pour ajuster les paramètres du modèle pour minimiser la fonction de coût.

Tout d'abord, nous avons commencé par utiliser seulement l'entropie croisée comme fonction de coût dans l'entraînement. Mais celle-ci n'était pas optimale puisqu'elle ne maximisait pas la métrique de similitude de Dice. Pour ce faire, on a ajouté le Diceloss à notre fonction de coût en la combinant avec l'entropie croisée pour maximiser le taux de similitude. [6]

$$\begin{aligned} Loss &= w_{dice} * L_{dice} + w_{CE} * L_{CE} \\ L_{dice} &: 1 - \text{Dice}(s_{pred}, onehot_{labels}) \\ L_{CE} &: \text{CrossEntropy}(pred, labels) \end{aligned}$$

En ajoutant des poids aux fonctions de coûts ça nous permet de mieux contrôler quelle fonction pénalise le plus le modèle. Dans notre situation, on a donné un plus petit poids pour l'entropie croisée que le diceloss, pour la simple et bonne raison c'est que l'entropie croisée favorise la classe du background, car les classes dans nos images étaient déséquilibrées, il y avait environ 75% de background par image. De plus, en accordant plus d'importance au diceloss, ça forçait notre modèle à mieux s'entraîner pour maximiser le dice. D'ailleurs, notre fonction de diceloss ne tient pas compte du background. Celle-ci calcule la dice de chaque classe pour ensuite faire une moyenne de toutes les classes :

$$\begin{aligned} L_{dice} &= 1 - \frac{1}{C} * \sum_{c=0}^C \text{dice}(GT_c, pred_c) \\ C &= Classes - 1 \end{aligned}$$

Dans la validation, on a uniquement utilisé le diceloss comme fonction de coût afin de maximiser le dice et effectuer notre condition d'arrêt afin d'éviter le surapprentissage de notre modèle. De plus, nous avons une simple fonction d'arrêt, on précise un terme de patience = 3, si l'erreur sur la validation ne diminue plus après trois itérations, on arrête notre apprentissage et on sauvegarde notre modèle.

3.5 Optimisateurs

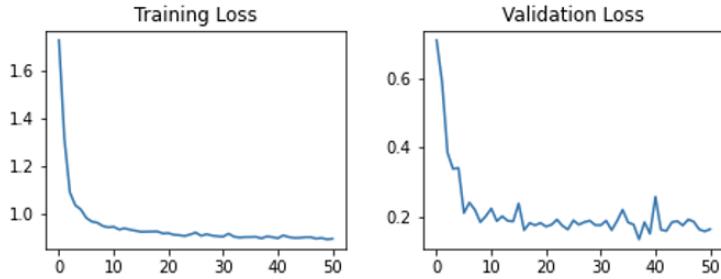
Les optimisateurs [7] sont primordiaux dans l'apprentissage machine, car ils permettent d'améliorer la convergence vers des minimums globaux. Il existe une variété d'optimisateurs qui ont chacun leurs avantages et inconvénients. Cependant, nous avons décidé d'utiliser les optimisateurs Adam et SGD combiné avec Momentum. De plus, afin de faire varier le taux d'apprentissage et améliorer la convergence, nous avons utilisé différents types de Scheduler.

L'optimisateur SGD est un algorithme qui est utilisé pour améliorer la convergence et la performance de notre modèle. Ce dernier permet d'avoir une meilleure généralisation de notre modèle, mais il peut être long à converger.

Il est efficace d'utiliser des Scheduler pour augmenter la vitesse de conver-

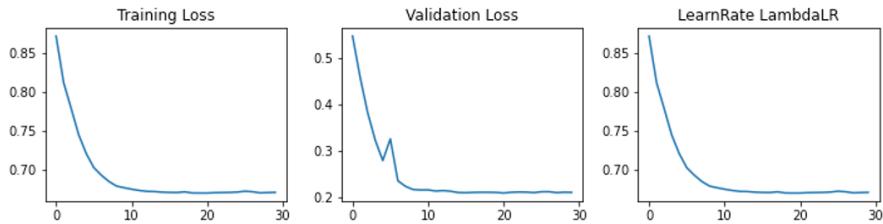
Figure 13: Entrainement avec l'optimisateur SGD

`lr:0.001 train_batch_size:8 val_batch:3 regul_dice:1 optimizer:SGD`



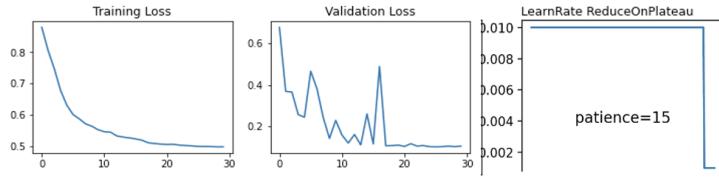
gence. Cependant, nous avons testé l'optimisateur avec un Scheduler *personnalisé* [7], celui-ci change le taux d'apprentissage par un facteur à chaque epoch. Le désavantage de ce Scheduler personnalisé est qu'après un certain nombre d'epoch, le taux d'apprentissage devient très petit et le modèle n'apprend plus rien.

Figure 14: Lamda LR
`lr:0.1 train_batch_size:16 val_batch:10 optimizer:SGD`



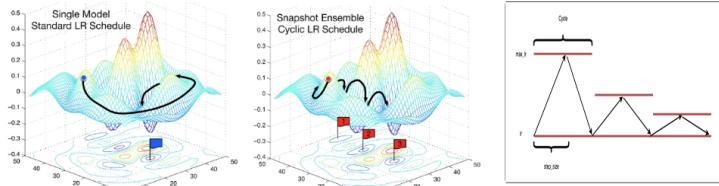
D'autre part, on a essayé avec le Scheduler ***ReduceLROnPlateau*** [7]. Ce dernier permet de diminuer le taux d'apprentissage lorsque l'erreur de validation commence à plafonner après un certain nombre d'epoch. Ce Scheduler est très efficace, car il permet de diminuer le taux de manière automatique si on stagne à un niveau d'erreur. Le plus important avec ce Scheduler est de bien choisir le bon taux de patience, car si on a un taux de patience vraiment grand, le modèle peut faire du surapprentissage avant que le taux d'apprentissage soit diminué.

Figure 15: ReduceOnPlateau LR



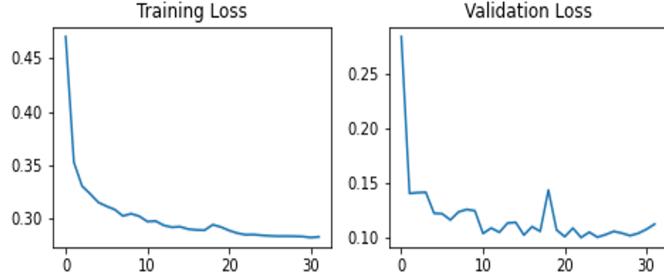
De plus, on a testé le Scheduler ***Cyclic*** [7]. Celui-ci permet de définir un taux d'apprentissage cyclique qui varie de manière triangulaire au cours de l'entraînement du modèle. Cela permet de converger plus rapidement en prenant le chemin le plus rapide qui mène au minimum global. Si on avait utilisé un Scheduler standard, on aurait fait un long détour pour aboutir sur le minimum global. Le graphique ci-dessous illustre ce comportement :

Figure 16: Cyclique LR



À la fin, on a décidé de garder l'optimiseur ***Adam*** [7], car il n'a pas réellement besoin de Scheduler, puisque ce dernier gère seulement le taux d'apprentissage. Il est très utilisé par la communauté scientifique. Bien que celui-ci soit rapide à converger, il n'offre pas une meilleure généralisation que l'optimiseur SGD. Par contre, il offre des performances très satisfaisantes. Cependant, c'est l'optimiseur Adam sans Scheduler qui a donné les meilleures prédictions.

Figure 17: Adam
lr:0.001 train_batch_size:15 val_batch:6 optimizer:ADAM



4 Résultats

Pendant l’entraînement, nous avons seulement testé les modèles UNet et ResidualUNet avec différentes configurations. Le tableau suivant montre les Dice obtenus pour chaque modèle et chaque classe:

Modèle	Paramètres	Classe 1	Classe 2	Classe 3	Moyenne
ResidualUNet	lr=0.01, opt=Adam, epoch=20	0,83	0,84	0,93	0,8700
ResidualUNet	lr=0.01, opt=SGD, epoch=33	0,79	0,74	0,88	0,8033
UNet	lr=0.001, opt=SGD+CyclicLr, epoch=22	0,80	0,79	0,90	0,7233
UNet	lr=0.001, opt=SGD+LamdaLr, epoch=22	0,69	0,68	0,80	0,7233
UNet	lr=0.005, opt=Adam, epoch=62	0,83	0,82	0,92	0,8567
UNet	lr=0.001, opt=Adam, epoch=30	0,85	0,85	0,93	0,8767

Table 1: Tableau des résultats dice obtenus sur les images de tests

Comme on peut le voir, les deux modèles ont été très performants. Nous avons passé beaucoup plus de temps à entraîner le ResidualUNet, car il avait presque deux fois plus de paramètres. Pour le challenge, nous avons finalement choisi le modèle UNet, car les valeurs que rentraitait sa fonction de coût étaient plus basses.

Métrique	Classe 1	Classe 2	Classe 3	Moyenne
Dice	0,67	0,77	0,85	0,76
HD	8,41	5,64	4,57	6,21
ASD	2,51	1,93	1,79	2,07

Table 2: Tableau des résultats obtenus lors du challenge

5 Conclusion

Pour conclure, nous avons choisi d'utiliser le modèle Unet. Sur ce modèle, nous avons fait de l'augmentation de donnée. Cela permet d'augmenter la quantité de données disponibles pour l'entraînement du modèle. On a dû utiliser différentes techniques comme le RandomRotate90, CenterCrop, GaussNoise, etc. afin d'avoir des données de qualité. De plus, grâce aux hyperparamètre, le modèle Unet a obtenu les meilleurs résultats durant l'entraînement avec un score Dice moyen de 0.87. Toutefois, il serait intéressant de voir la performance de notre modèle avec d'autres tâches dans le même domaine d'imagerie médicale. Cependant, nous proposerons une implémentation plus élaborée afin de maximiser nos chances d'avoir de meilleurs résultats.

References

- [1] Olivier Bernard, Alain Lalande, Clement Zotti, Frederick Cervenansky, Xin Yang, Pheng-Ann Heng, Irem Cetin, Karim Lekadir, Oscar Camara, Miguel Angel Gonzalez Ballester, et al. Deep learning techniques for automatic mri cardiac multi-structures segmentation and diagnosis: is the problem solved? *IEEE transactions on medical imaging*, 37(11):2514–2525, 2018.
- [2] Alexander Buslaev, Vladimir I. Iglovikov, Eugene Khvedchenya, Alex Parinov, Mikhail Druzhinin, and Alexandr A. Kalinin. Albumentations: Fast and flexible image augmentations. *Information*, 11(2), 2020.
- [3] Hu Cao, Yueyue Wang, Joy Chen, Dongsheng Jiang, Xiaopeng Zhang, Qi Tian, and Manning Wang. Swin-unet: Unet-like pure transformer for medical image segmentation. *arXiv preprint arXiv:2105.05537*, 2021.
- [4] Jieneng Chen, Yongyi Lu, Qihang Yu, Xiangde Luo, Ehsan Adeli, Yan Wang, Le Lu, Alan L Yuille, and Yuyin Zhou. Transunet: Transformers make strong encoders for medical image segmentation. *arXiv preprint arXiv:2102.04306*, 2021.
- [5] Huimin Huang, Lanfen Lin, Ruofeng Tong, Hongjie Hu, Qiaowei Zhang, Yutaro Iwamoto, Xianhua Han, Yen-Wei Chen, and Jian Wu. Unet 3+: A full-scale connected unet for medical image segmentation. In *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1055–1059. IEEE, 2020.
- [6] Shruti Jadon. A survey of loss functions for semantic segmentation. In *2020 IEEE Conference on Computational Intelligence in Bioinformatics and Computational Biology (CIBCB)*, pages 1–7. IEEE, 2020.
- [7] Aliasghar Mortazi. Optimization algorithms for deep learning based medical image segmentations. 2019.