UNIVERSITÉ DE SHERBROOKE

IFT 780 - TP4

Segmentation d'images par résonance magnétique

Ait ichou Yoann

Amairi Tahar

Arsenault Pierre Daniel

Lien GitLab : https://depot.dinf.usherbrooke.ca/dinf/cours/h23/ift780/aity1101/projet_session_hiver.git

Faculté des sciences,
Département Informatique

18 avril 2023

Table des matières

1 Architectures

1.1 Fichiers modifiés

Les fichiers modifiés lors de la construction de nos réseaux sont les fichiers suivants :

- `src/manage/CNNTrainTestManager.py` : ajout du checkpointing et de la data augmentation, modification des noms des plots (à la place de fig1.png etc...) et leur emplacement de sauvegarde.
- `src/models/CNNBlocks.py` : ajout d'une nouvelle classe ResBottBlock.
- `src/models/CNNBaseModel.py` : modification du chemin de sauvegarde du modèle vers le dossier weights.
- `src/models/yourSegNet.py` : implémentation de yourSegNet.
- `src/models/yourUNet.py` : implémentation de yourUNet.
- `src/train.py` : ajout des différents arguments.
- `src/utils/utils.py` : sauvegarde les images augmentées.
- `src/models/Loss.py` : nouveau fichier incorporant les nouvelles losses.
- `figure` : nouveau dossier contenant les plots.
- `weights` : nouveau dossier contenant des modèles.
- `tp4.ipynb` : contient toutes les commandes utilisées tout au long de ce TP.

1.2 Illustration des architectures

Les blocs résiduels de yourSegNet et de yourUNet correspondent tous à l'architecture interne suivante, implémentant une nouvelle couche nommée RestBott :

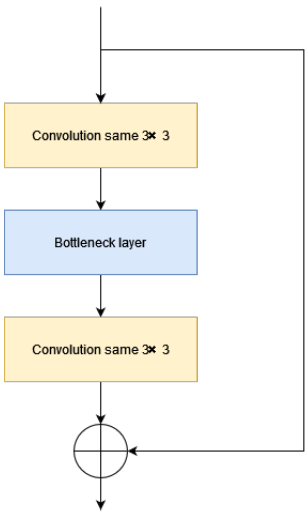


FIGURE 1 – Architecture des blocs RestBott résiduels

Dans notre architecture de yourUNet, nous avons fait le choix d'implémenter 3 blocs supplémentaires, tout en conservant l'architecture globale du UNet fourni :

- les blocs denses
- les blocs résiduels ResBott
- les blocs bottlenecks, compris dans les blocs résiduels
- les maxpooling, les convolutions transposées, et les liaisons skip propres au UNet

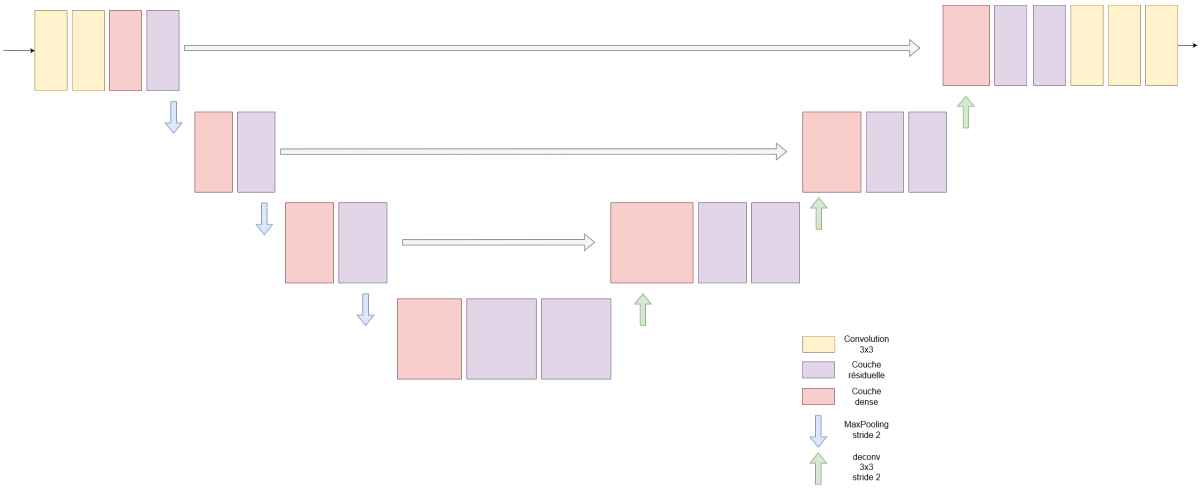


FIGURE 2 – Architecture de yourUnet

L'architecture de notre yourSegNet s'inspire de l'architecture du DeepLapV3 et de sa technique d'ASPP (Atreus Spatial Pyramid Pooling) qui a ici été implémentée en parallèle. Nous utilisons donc :

- les blocs résiduels
- les blocs bottlenecks, compris dans les blocs résiduels
- des max pooling pour réduire la taille des features maps
- un upsampling dans la couche finale pour retrouver les dimensions initiales

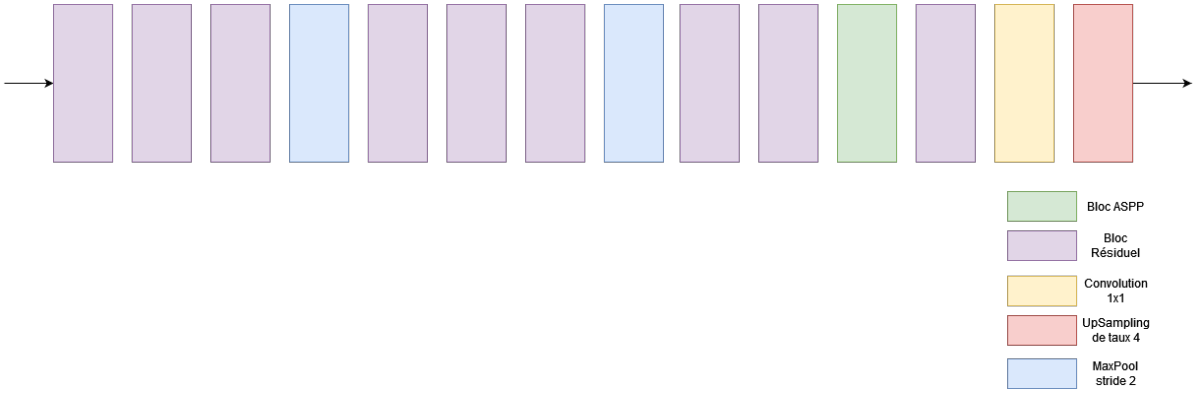


FIGURE 3 – Architecture de yourSegNet

1.3 Nouvelles losses

1.3.1 Dice loss

$$DiceLoss = 1 - \frac{2TP}{2TP + FN + FP}$$

$$DiceLoss = 1 - \frac{2|X \cap Y|}{|X| + |Y|} = 1 - DiceScore$$

Dans notre cas de figure, le Dice score a déjà été implémenté. Nous l'avons donc réutilisé dans notre calcul de la Dice Loss. Pour l'activer, il faut utiliser l'argument `-loss=dice`.

1.3.2 Tversky loss

$$TvLoss = 1 - \frac{TP}{TP + \alpha FN + \beta FP}$$

La Tversky loss est une version modifiée de la Dice loss. Pour son implémentation, nous avons récupéré des éléments de depuis ce git. Pour l'activer, il faut utiliser l'argument `-loss=tversky`.

1.4 Courbes d'entraînement

Pour yourUNet, voici la courbe d'entraînement sans augmentation de données :

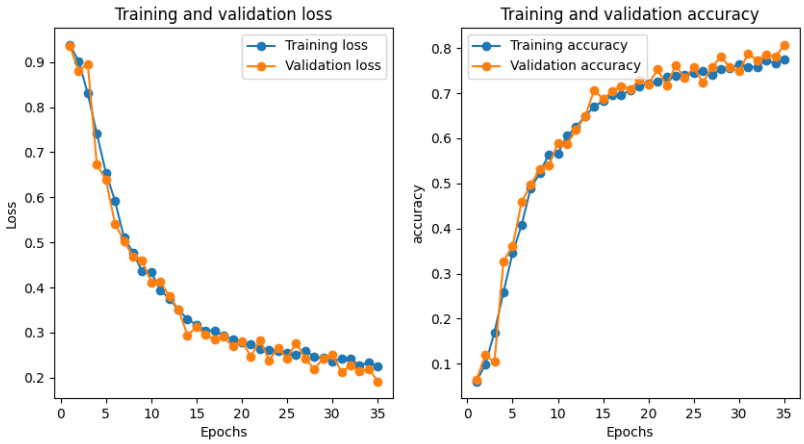


FIGURE 4 – Courbe d'entraînement du yourUNet

Avec cette architecture, nous parvenons à obtenir une justesse de validation de 0.8 au bout de 35 epochs. Comme nous allons le voir par la suite, ce modèle atteint d'excellentes justesses au bout d'un nombre d'epochs plus élevé que celui du yourSegNet. Notre justesse évolue assez rapidement lors des premières epochs, puis commence à ralentir vers la quinzième epoch. L'évolution de notre perte est satisfaisante, elle baisse bien au fur et à mesure que les epochs progressent.

Pour yourSegNet, voici la courbe d'entraînement sans augmentation de données :

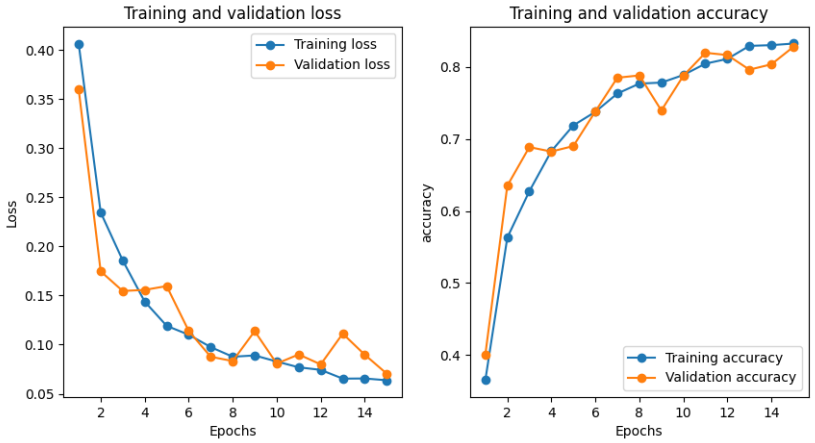


FIGURE 5 – Courbe d'entraînement du yourSegNet

On voit qu'en 15 epochs le modèle atteint une justesse de validation de 0.8. On constate que sa progression ralentit, mais elle pourrait probablement continuer d'augmenter. La courbe de perte est cohérente : elle diminue au cours des epochs. Les courbes d'entraînement avec les données modifiées sont dans la section "Augmentation des données".

1.5 Manuel d'exécution

De nouveaux arguments ont été ajoutés :

- `-save` : permet de sauvegarder les poids du modèle à la fin de l'entraînement pour effectuer de la prédiction. Les poids sont enregistrés dans le dossier `weights` sous le nom `ModelNamept`. Les poids enregistrés ne sont pas ceux du meilleur modèle mais uniquement ceux obtenus à la fin de l'entraînement. Pour obtenir ceux du meilleur modèle, il faut utiliser le checkpointing.
- `-plot` : permet d'afficher les figures (i.e les courbes d'apprentissage et les prédictions selon le cas) si l'environnement le permet. Les figures sont aussi enregistrées dans le dossier `figure`.

Voici maintenant les commandes utilisées pour obtenir les courbes d'apprentissage :

- `!python train.py -model=yourUNet -num-epochs=35 -batch_size=14 -plot -save -lr=0.001 -loss=dice -enable_checkpoint`
- `!python train.py -model=yourSegNet -num-epochs=15 -batch_size=7 -plot -save -lr=0.001 -loss=tversky -enable_checkpoint`

2 Checkpointing

2.1 Fichiers modifiés

Pour implémenter le checkpointing, il était nécessaire de modifier les fichiers ci-dessous :

- `src/manage/CNNTrainTestManager.py`
- `src/train.py`

Le premier étant l'interface permettant de gérer l'entraînement et le test du modèle, il contient le code pour effectuer le checkpointing. Par conséquent, deux nouvelles fonctions ont été ajoutées : `checkpoint` et `load_checkpoint_fn`. Des modifications ont été aussi incorporées à la classe `CNNTrainTestManager` pour pouvoir gérer le checkpointing. Le second fichier contient l'ajout des arguments pour activer le checkpointing via la ligne de commande.

2.2 Manuel d'exécution

Deux arguments ont été ajoutés pour gérer le checkpointing :

- `-enable_checkpoint` : permet d'activer le checkpointing durant l'entraînement du modèle. A travers les epochs, seul le modèle ayant la meilleure justesse de validation est enregistré dans le dossier `weights` sous le nom `ModelName_checkpoint_weights.pt`. Les autres paramètres (hors des poids du modèle) sont enregistrés sous le nom `ModelName_checkpoint.pt`.
- `-load_checkpoint` : permet de reprendre l'entraînement depuis un checkpoint chargé depuis le dossier `weights`.

Voici un exemple :

- `!python train.py -model=UNet -num-epochs=5 -enable_checkpoint`
- `!python train.py -model=UNet -num-epochs=10 -load_checkpoint`

Techniquement dans cet exemple, on pourra reprendre l'entraînement à la 6ème epoch lors de l'exécution de la seconde commande.

3 Augmentation des données

3.1 Fichiers modifiés

Pour ajouter l'augmentation de données, les fichiers utils.utils.py et train.py ont été modifiés. Nous avons ajouté la fonction `extract_data_augment`. Cette fonction produit et sauvegarde 4 versions (image d'entrée, image normalisée, image modifiée avec le premier bloc d'augmentation de données et image modifiée avec le premier bloc d'augmentation de données) d'un certain nombre d'image différentes. Dans train.py, l'argument `data_aug` a été modifié et l'argument `n_extract_data_augment` a été ajouté. `data_aug` est maintenant un nombre représentant le type de d'augmentation et `n_extract_data_augment` est le nombre d'image qu'on veut extraire. S'il est plus grand que 0, il n'y aura pas d'entraînement, mais seulement des images modifiées qui seront téléchargées. C'est dans ce fichier que la fonction `extract_data_augment` est appelée.

3.2 Types d'augmentation

D'abord, notre transformation de base contient une Normalisation(0.5, 0.5). Cette transformation est réalisée même sans data augmentation. Elle normalise les données avec une moyenne de 0.5 et un std de 0.5. Nous obtenons de meilleurs résultats avec celle-ci. Comme le base_transform fourni en comportait une, nous avons l'avons également utilisé dans notre transformation de base.

Ensuite, le premier bloc d'augmentation de données est composé d'une réflexion verticale de probabilité de 0.2, d'une réflexion horizontale de probabilité de 0.2 et d'une normalisation de moyenne 0.5 et de d'écart-type de 0.5.

Puis, le deuxième bloc d'augmentation de données est composé d'une rotation de -20 à 20 degré suivi d'un bruit gaussien (gaussian blur) avec un masque de grandeur 5 et d'une normalisation de moyenne 0.5 et de d'écart-type de 0.5.

La figure 6 comporte 5 exemples de données modifiées dans 4 états différents (de gauche à droite) : l'image brute, l'image normalisée, image ayant été modifiée par le premier bloc d'augmentation de données et l'image ayant été modifiée par le deuxième bloc.

Notons que nous avons essayé un random crop, mais les résultats étaient mauvais.

3.3 Effet de l'augmentation des données

Voici les augmentations de données faites sur 5 images différentes :

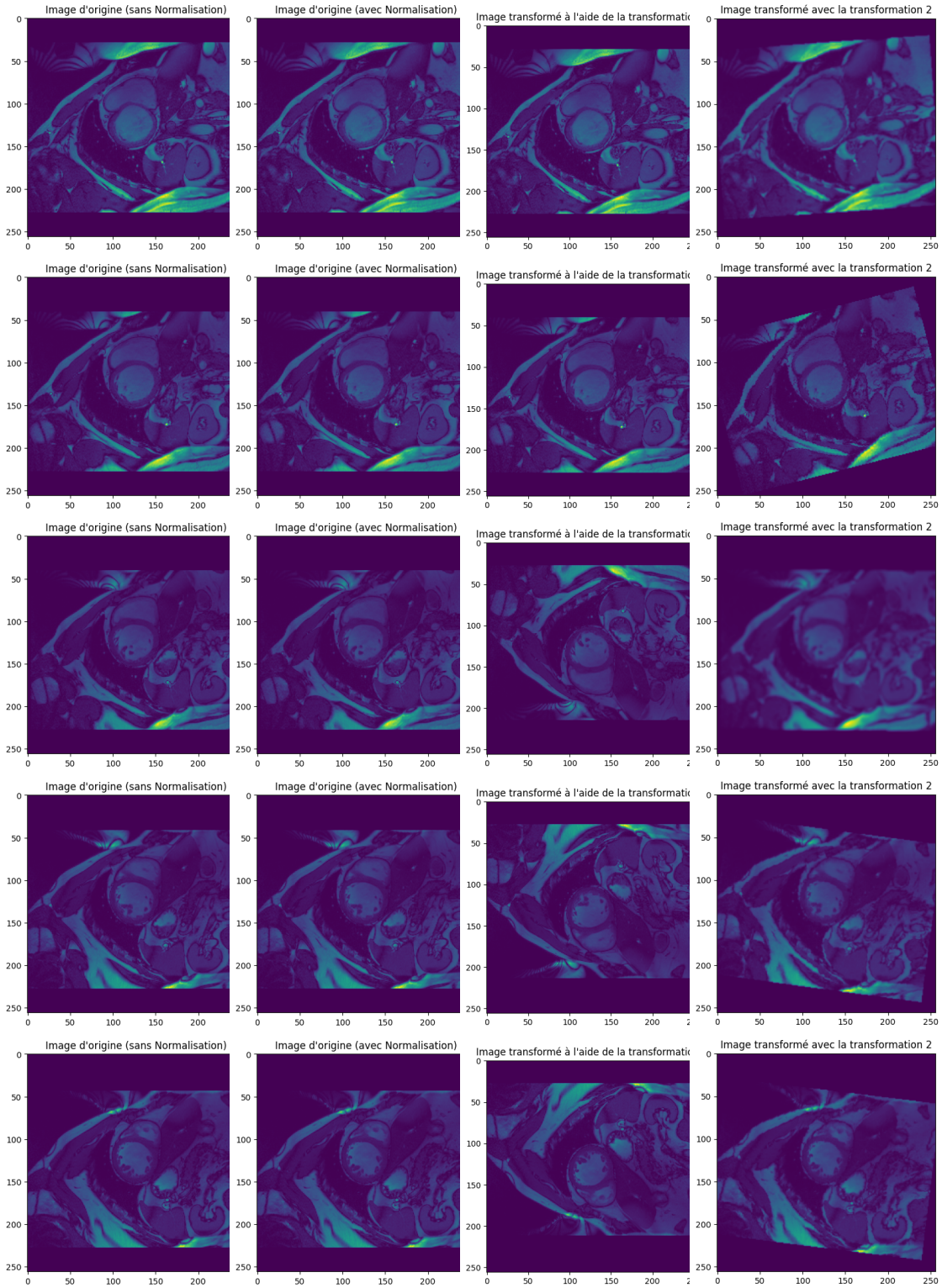
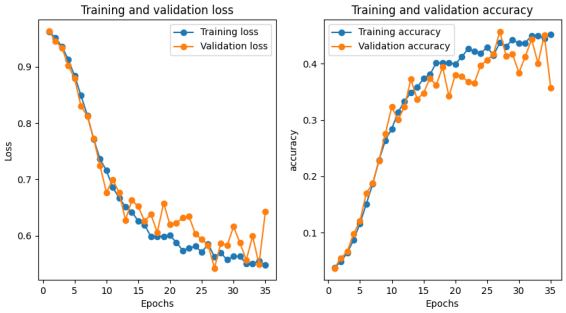
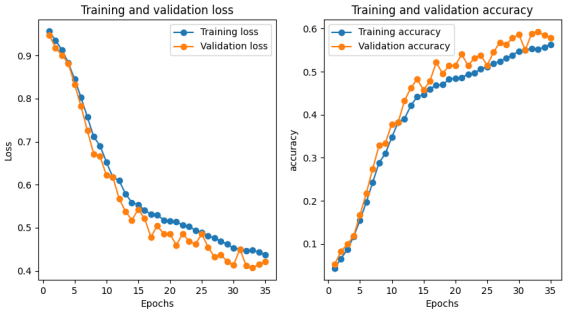


FIGURE 6 – Effet des augmentations de données sur 5 images différentes

Nous avons testé les augmentations de données sur nos modèles. Nous voyons que l'augmentation de données diminue notre score. Il s'agit d'un comportement normal puisque nous ne concaténons pas les données transformées avec les données d'origine. En effet, nous fournissons seulement les données transformées aux modèles. Les figures 6 à 10 montrent les courbes d'entraînements réalisées avec de l'augmentation de données.

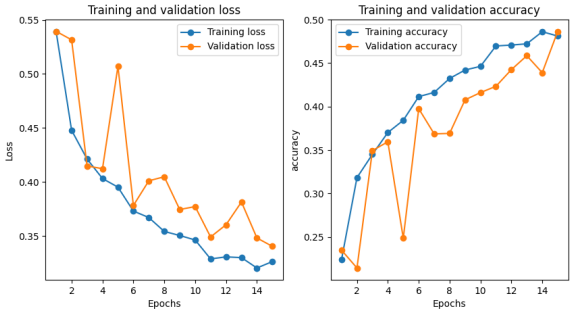


(a) Data augmentation 1

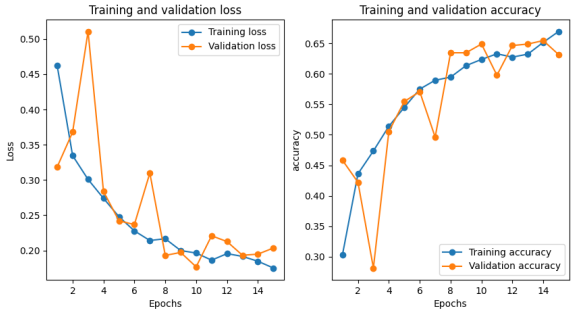


(b) Data augmentation 2

FIGURE 7 – Modèle yourUNet



(a) Data augmentation 1



(b) Data augmentation 2

FIGURE 8 – Modèle yourSegnet

3.4 Manuel d'exécution

Si on veut visualiser l'effet de l'augmentation de données sur 5 images, il faut entrer :

- `!python train.py -n_extract_data_augment 5`

Pour recourir à l'augmentation de données dans notre entraînement (sans concaténation) avec le modèle yourSegNet, par exemple, il faut entrer :

- `!python train.py -model=yourSegNet -num-epochs=15 -batch_size=7 -plot -save -lr=0.001 -loss tversky -enable_checkpoint -data_aug 1` (mode 1)
- `!python train.py -model=yourSegNet -num-epochs=15 -batch_size=7 -plot -save -lr=0.001 -loss tversky -enable_checkpoint -data_aug 1` (mode 2)