

Rapport de stage de fin d'études

Segmentation d'IRM 3D par méthodes évolutives et outillage de tiling/stitching pour l'imagerie médicale à grande échelle

August 26, 2025

Étudiant :

Poque Valentin valentin.poque@univ-tlse3.fr

Tuteurs :

Cussat-Blanc Sylvain
Cortacero Kévin
Mouyssset Sandrine

Mots-clés: CGP ; IA; Python; Traitement d'images

Remerciements :

J'aimerais remercier l'équipe pédagogique d'UPSSITECH dans un premier temps. Madame Ferrané qui a beaucoup œuvrée pour faire en sorte que ce stage réponde aux différentes attentes de la CTI malgré cette configuration particulière. Merci à Madame Mouysset également pour son suivi.

J'aimerais remercier également l'équipe REVA de l'Irit UT1 pour leur accueil, plus particulièrement Yuri Lavinas et Sylvain Cussat-Blanc pour leur suivi.

Merci également à Kévin Cortacero et ses associés au sein de la start-up Mantalys qui m'ont permis de réaliser mon quota de semaines en entreprises.

Je remercie encore une fois l'Irit UT1 et Sylvain d'avoir accepté cette configuration hybride lors du stage.

Contents

1	Introduction	1
1.1	Contexte du stage	1
1.2	Présentation brèves des deux sujets	1
1.2.1	IRIT UT1	1
1.2.2	Mantaly	1
2	Phase préliminaire : Documentation et acquisition de connaissances	2
2.1	GP vs CGP	2
2.2	Kartezio	5
2.3	MAGE	8
3	Phase 1 : Stage à l'IRIT Capitole	13
3.1	Objectif du stage et environnement de travail	13
3.2	Traitement seulement sur la coupe Axiale	14
3.2.1	Premier test sans traitement particulier	14
3.2.2	Nouvelle salve de tests avec utilisation de la Sliding Window	15
3.2.3	Traitement sur les images entières	17
3.3	Traitement sur les trois coupes en même temps	18
3.4	Transition sur l'outil MAGE	19
4	Phase 2 du stage : Travail au sein de la start-up Mantaly	20
4.1	Objectif du stage	20
4.2	Environnement de travail et état d'avancement du projet	20
4.3	État de l'art sur le stitching de grosse image	20
4.3.1	Stardist[14]	21
4.3.2	InstanSeg[15]	22
4.4	Mise en place du stitching pour Mantaplex	23
4.4.1	Fonctionnement globale attendu	23
4.4.2	Première et deuxième étapes : gestion avec chunks	25
4.4.3	Étape trois : généralisation du traitement 1D	29
4.4.4	Étape 4: Généralisation au traitement 2D	32

5 Phase 3 et suite du stage	35
Conclusion	36
Annexes	37

1 Introduction

1.1 Contexte du stage

Ce stage prend place en qualité de stage de fin d'études de ma formation SRI à l'école d'ingénieurs UPSSITECH à Toulouse. Il a pour but de professionnaliser l'étudiant en lui offrant une première expérience professionnelle longue durée de 5 à 6 mois.

Ce stage s'effectuera dans un contexte hybride car il doit répondre à certaines contraintes. Dans mon cas, le stage doit avoir lieu en entreprise car dans le cadre de notre formation la CTI impose un nombre de semaines de travail à effectuer en entreprise que je n'ai pas encore réalisé lors de mon stage précédent.

Or initialement ce stage devait se dérouler uniquement en laboratoire, il fallait donc trouver une solution pour y intégrer des semaines en entreprise. Grâce à la coopération d'Isabelle Ferrané, Sylvain Cussat-Blanc ainsi que Kévin Cortacero une solution a été trouvée.

Le stage sera séparé en deux parties :

- Première phase de trois mois dans le laboratoire d'informatique de l'IRIT UT1, encadré par M. Cussat-Blanc
- Deuxième phase de deux mois dans l'entreprise Mantalys de M. Cortacero

1.2 Présentation brèves des deux sujets

1.2.1 IRIT UT1

Le but de ce stage était de segmenter la neuromélanine sur des IRMs de cerveau en utilisant des outils tels que Kartezio et MAGE qui génèrent des pipelines transparents et interprétables grâce à la CGP.

1.2.2 Mantalys

Ici le but était d'optimiser le traitement des images de lames de microscope où l'on veut faire de la segmentation. En effet ce sont de très grosses images qui ne peuvent pas être affichées directement sur un PC avec les librairies classiques. Il faut donc mettre en place une méthodologie pour pouvoir appliquer la segmentation notamment en les découplant en images plus petites pré-traitement puis en les reconstruisant post-traitement.

2 Phase préliminaire : Documentation et acquisition de connaissances

Ces deux sujets nécessitent des connaissances solides dans des domaines niches que nous n'avons pas forcément abordés à UPSSITECH. Le plus important d'entre eux étant la programmation génétique (CGP et GP). Évidemment, toutes les bases nécessaires pour comprendre ces concepts ont été étudiées de long en large à UPSSITECH ce qui rend le travail de documentation plus simple. Je vais donc poser les bases sur les concepts de CGP et GP avant d'aborder plus en détail les articles lus.

2.1 GP vs CGP

CGP (Cartesian Genetic Programming) est une extension de GP (Genetic Programming) donc nous allons commencer par nous pencher sur GP dans un premier temps.

La programmation génétique (GP) vise à générer automatiquement des programmes informatiques [1]. Cette méthode vise à reproduire la théorie de l'évolution de Darwin en générant de nouvelles populations à partir d'une population initiale, généralement créée de manière aléatoire. Chaque individu est représenté par un arbre syntaxique contenant deux types de noeuds, les noeuds internes et les noeuds terminaux. Les noeuds internes sont des noeuds fonctionnels qui représentent donc des fonctions telles que « * », « + », « if » et bien d'autres. Tandis que les noeuds terminaux sont des variables ou des constantes comme nous allons le voir dans l'exemple qui suit.

Lors de la création de chaque individu, les combinaisons entre ces noeuds sont faites aléatoirement.

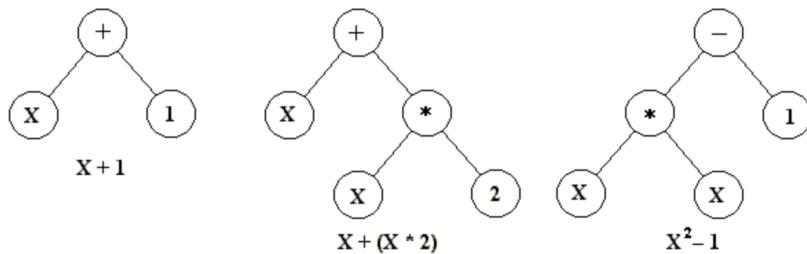


Figure 1: Example de résultat GP [2]

Voici un exemple de résultat obtenu sur différents individus. Sur le premier individu, en partant de la gauche, on voit que l'opérateur "+" est le seul utilisé et qu'il applique cette opération sur les noeuds terminaux x et 1 formant ainsi l'expression $x + 1$.

Pour le deuxième individu, le noeud fonctionnel "+" se trouve à la racine de l'arbre et applique l'opération "+" entre x et le résultat de l'opération "*" sur les noeuds terminaux x et 2. Ce qui donne l'expression $x + (x * 2)$.

Le dernier individu est plutôt similaire structurellement, nous avons l'opérateur "-" à la racine qui prend deux arguments, le noeud terminal 1 et le résultat du noeud fonctionnel "*" et des noeuds terminaux x et x . Ce qui donne l'expression $x^2 - 1$.

Chaque individu est ensuite évalué vis-à-vis d'une fonction de fitness définie au préalable selon le résultat souhaité. Une nouvelle population sera générée en gardant les meilleurs individus de la précédente et en appliquant des opérations de croisement et de mutation qui permettront d'explorer de nouvelles combinaisons de noeuds fonctionnels et terminaux tirées encore une fois aléatoirement

depuis la liste de nœuds disponibles. Le but étant de créer des individus encore plus proches de la fitness que les précédents. En général, le programme s'arrête lorsque le score de la fitness cible est atteint, lorsque nous avons dépassé le nombre de générations défini dans les paramètres ou bien lorsque la fitness stagne sur x générations (x étant une variable que nous avons définie).

Place maintenant à la CGP car c'est la notion centrale de mon stage. Comme vu précédemment, CGP (Cartesian Genetic Programming) est une extension de GP qui va donc l'améliorer et apporter de nouvelles possibilités.

Contrairement à GP classique, CGP encode les individus non plus sous forme d'arbre syntaxique mais bien de graphe dirigé acyclique [3].

Chaque individu est composé de deux éléments : le génotype et le phénotype.

CGP encode des fonctions mathématiques ou des pipelines de traitement, le tout est représenté sous la forme d'un génotype linéaire qui encode le phénotype sous la forme d'un graphe fonctionnel.

Un des principaux problèmes avec GP était le « Bloat » qui se caractérise par une croissance incontrôlée de la taille des programmes pendant le processus d'évolution sans y retrouver une amélioration dans la fitness [4].

CGP résout ce problème en contrôlant mieux la mutation et en autorisant les nœuds inactifs.

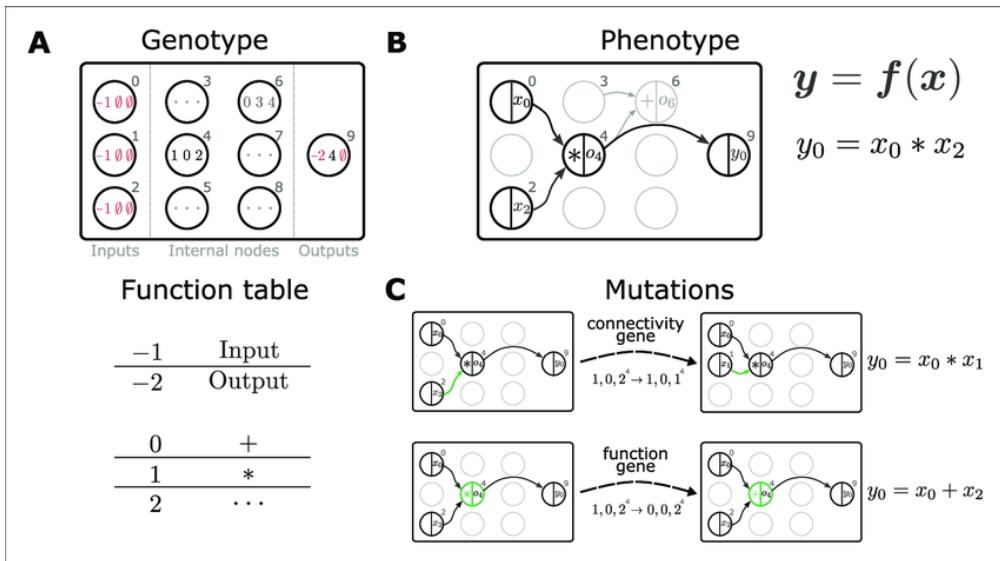


Figure 2: Exemple de CGP [5]

Dans la partie A de la figure 2 ci-dessus, nous avons un exemple de génotype en CGP.

Chaque nœud est représenté sous la forme d'un triplet (f, i_1, i_2) . Les valeurs possibles de f sont celles décrites dans la Function table. Les valeurs -1 et -2 sont des valeurs spéciales pour représenter les entrées et les sorties dans les colonnes Inputs et Outputs, ce sont des pointeurs sur des variables.

Dans notre exemple de génotype nous avons 3 nœuds en entrée, les nœuds $0, 1, 2$. Ainsi qu'un nœud en sortie, le nœud 9 . Dans un génotype, les nœuds représentent une connexion entre le résultat de deux nœuds et une fonction. Par exemple le nœud 2 applique la fonction 1 de la table de fonction aux nœuds 0 et 2 . Ce qui correspond à l'opération "*" sur les variables d'entrée des nœuds 0 et 2 .

Même principe pour le noeud 6 qui applique l'opération correspondant à la ligne 0 de la table des fonctions aux sorties des noeuds 3 et 4. Ce qui correspond à l'opération "+", on notera également que cette opération est ici effectuée sur le résultat de deux noeuds internes et non plus les noeuds d'entrée même si cela aurait été tout à fait possible. Il est intéressant de noter également que la sortie est définie comme étant la sortie du noeud 4. Cette désignation fait partie intégrante du génotype, au même titre que les fonctions et connexions des noeuds internes.

Comme pour la programmation génétique classique (GP), les premiers individus sont générés de manière aléatoire, qui dit individu dit génotype donc les connexions entre les noeuds d'entrée, les noeuds internes et les noeuds de sortie du génotype sont également générées de manière aléatoire. La génération suivante d'individus sera ensuite formée à partir des individus de la génération précédente ayant les meilleures performances selon la fonction de fitness.

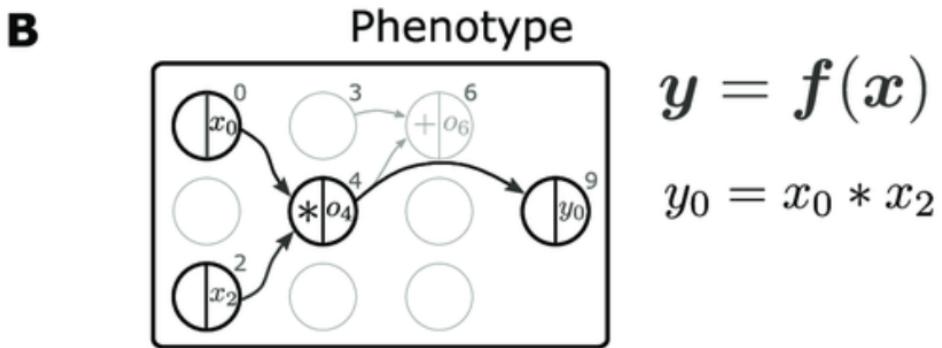


Figure 3: Focus sur le Phénotype [5]

Comme nous venons de le voir, le génotype en CGP décrit la structure codée d'un individu en spécifiant les fonctions présentes dans chaque noeud, leurs connexions, ainsi que les sorties du programme. Cependant, comme nous l'avons vu tout à l'heure, tous les noeuds ne sont pas utilisés pour obtenir la sortie. En effet, notre sortie était le noeud 4 donc tous les noeuds qui n'intervenaient pas dans le chemin menant le noeud 4 aux entrées n'étaient pas utilisés.

C'est là qu'intervient la notion de phénotype. Le phénotype correspond à la partie active du graphe, c'est le sous-graphe réellement utilisé pour produire la sortie. C'est-à-dire tous les noeuds qui permettent de relier le noeud de sortie du génotype au premier noeud avec lequel il est connecté. Lors de l'évaluation de l'individu, le graphe est lu en partant de la sortie et en suivant les connexions. Sont considérés comme inactifs tous les noeuds qui n'ont pas été sollicités pendant ce parcours. Ils font partie du génotype mais n'interviennent pas sur l'individu à cette étape de l'évolution.

La représentation offerte par CGP est donc redondante et modulaire car elle permet d'avoir des noeuds inactifs pendant plusieurs générations qui peuvent devenir actifs plus tard à cause d'une mutation ou d'un croisement. Nous stockons donc des possibilités pour plus tard. De plus, chaque noeud, sous graphe ou groupe de noeuds peut devenir inactif après une mutation sans changer tout le système.

C'est ce que nous observons sur le schéma avec les noeuds 0 et 2 comportant les entrées x_0 et x_2 qui sont connectés au noeud 4 qui applique l'opération de multiplication et stocke le résultat dans o_4 . C'est pourquoi la sortie de ce phénotype vaut $y_0 = o_4 = x_0 * x_2$.

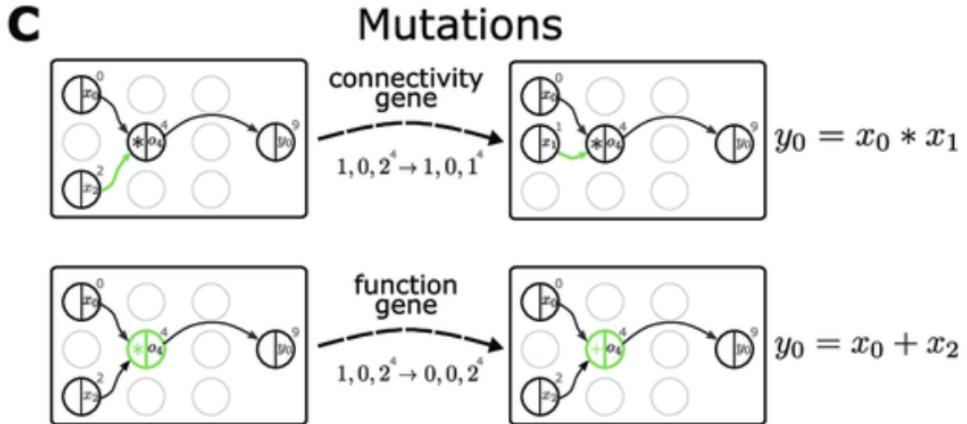


Figure 4: Focus sur la mutation [5]

En CGP, la mutation est le mécanisme principal permettant d'explorer l'espace de recherche. Elle permet de créer de nouveaux individus à partir du génotype d'un individu existant. Il existe deux types de mutations.

Le premier est la mutation de gène de connectivité, que l'on peut voir en haut sur le schéma ci-dessus.

Dans le cas d'une telle mutation, c'est la connexion entre les noeuds qui évolue, pas la fonction. Ce sont donc les entrées de la fonction qui changent comme nous pouvons le constater dans l'exemple où nous passons du triplet (1,0,2) à (1,0,1). Rappelons-nous que ces triplets sont écrits sous la forme (f, i_1, i_2). Ici le f ne bouge pas et ce sont les entrées qui sont modifiées pour explorer de nouvelles possibilités, ce noeud n'utilisera plus x_0 et x_2 mais x_0 et x_1 .

Le deuxième type de mutation est la mutation de gène de fonction, que l'on peut voir en bas sur le schéma.

Cette fois-ci c'est la fonction qui est modifiée, nous n'appliquons plus la fonction "*" mais "+" sans modifier i_1 et i_2 .

Ces deux mutations nous permettent d'ajouter deux nouvelles solutions à notre espace de solution qui offre des possibilités en plus pour le processus de sélection et d'évolution avec les expressions $y_0 = x_0 * x_1$ et $y_0 = x_0 + x_2$.

2.2 Kartezio

Kartezio est un framework développé par Kévin Cortacero, CEO de la start-up Mantalys et encadrant de la deuxième partie de mon stage. Je vais pour l'instant simplement présenter Kartezio de façon générale, je détaillerai plus tard dans ce rapport l'usage que j'en ai fait durant mon stage et pourquoi il a été utilisé. Le but de Kartezio est de générer des pipelines de traitement d'images médicales sans nécessiter beaucoup d'images d'entraînement car les annotations demandent du temps de travail à des chercheurs. Les datasets sont donc rares et souvent petits.

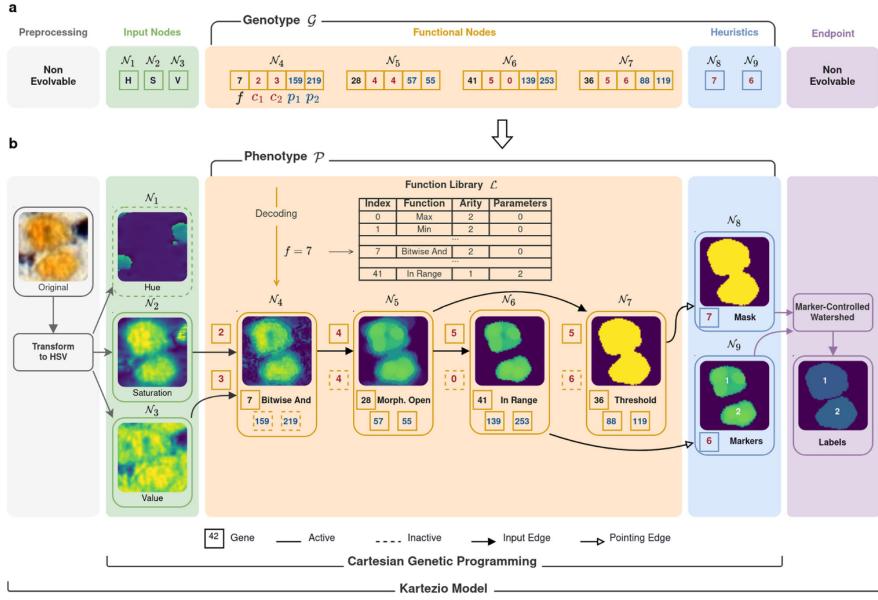


Figure 5: Schéma explicatif Kartezio [6]

Concentrons-nous d'abord sur la partie supérieure du schéma, la figure a. Nous retrouvons les éléments vus tout précédemment avec la CGP. À savoir le génotype contenant : les connexions avec les nœuds d'entrée, les nœuds fonctionnels, les nœuds de sortie.

Un gène représente un entier. Cela peut-être un gène f dont l'entier représente l'index de la fonction dans la librairie de fonctions. Cela peut également être un gène de connexions, comme les gènes c_1 et c_2 qui représentent respectivement la connexion aux nodes N_2 et N_3 qui sont des nodes d'entrée. Ces gènes peuvent aussi représenter des paramètres optionnels pour une fonction comme avec les gènes p_1 et p_2 .

Dans ce génotype, nous avons les noeuds allant de N_1 à N_3 représentant les nœuds d'entrée, de N_4 à N_7 les nœuds fonctionnels et enfin les noeuds N_8 et N_9 qui sont les noeuds de sortie sur lesquels sont appliqués l'heuristique afin d'évaluer la qualité des individus et guider le processus d'évolution.

Place maintenant à la deuxième partie du schéma, la partie b. Il y a deux composantes en plus avec Kartezio qui ne figurent pas dans CGP : le pré-procesing et le endpoint. Le pré-processing comme son nom l'indique est un processus qui se fait en amont et qui a pour but d'adpter les entrées à ce que l'utilisateur désire et juge le mieux pour le modèle. Ici le pré-processing choisi est la transformation de l'image en HSV et donc l'isolation des composantes H, S et V dans trois nouvelles images. Le endpoint s'applique en revanche à la fin du pipeline, il permet de conformer le résultat à ce que l'utilisateur attend. Par exemple, ici le Watershed contrôlé par marqueurs permet d'effectuer une segmentation d'instances pour séparer les deux cellules. Le endpoint et le pré-processing sont non évolutifs, ce sont des paramètres fixes choisis par l'utilisateur en amont.

Place à l'explication de la pipeline en elle-même. Une fois le preprocessing effectué, une première fonction est appliquée. C'est le noeud N_4 qui connecte la fonction Bitwise And ($f = 7$) aux noeuds N_2 et N_3 (c_1, c_2) avec les paramètres optionnels $p_1 = 159$ et $p_2 = 219$. La sortie de noeud est connectée à N_5 ($c_1 = c_2 = 4$) qui applique une ouverture ($f = 28$) avec les paramètres optionnels $p_1 = 57$ et $p_2 = 55$. Ensuite N_6 connecte le résultat de N_5 à la fonction in range ($f = 41$) avec les paramètres $p_1 = 139$ et $p_2 = 253$. Et enfin N_7 applique un threshold(seuil) ($f = 36$) sur le résultat de N_5 avec les paramètres optionnels $p_1 = 88$ et $p_2 = 119$. Tous les nœuds fonctionnels ont été traités, maintenant place aux nœuds de sortie. Ici nous avons deux nœuds de sortie, le nœud N_8 qui est

connecté au nœud N_7 ainsi que le nœud N_9 qui est connecté au nœud N_6 . Le nœud N_8 représente le masque binaire tandis que le noeud N_9 représente les marqueurs. Grâce à ces deux sorties, le endpoint peut appliquer le Watershed contrôlé par marqueurs afin d'obtenir une nouvelle image avec deux labels, un pour chaque instance. C'est cette image qui sera évaluée par l'heuristique afin de choisir les meilleurs individus pour la génération suivante.

Un des principaux avantages que présente Kartezio est que les pipelines générés sont entièrement interprétables et explicables car ce sont des pipelines sans aucune black box. Donc parfait pour des applications critiques dans des domaines tels que la médecine.

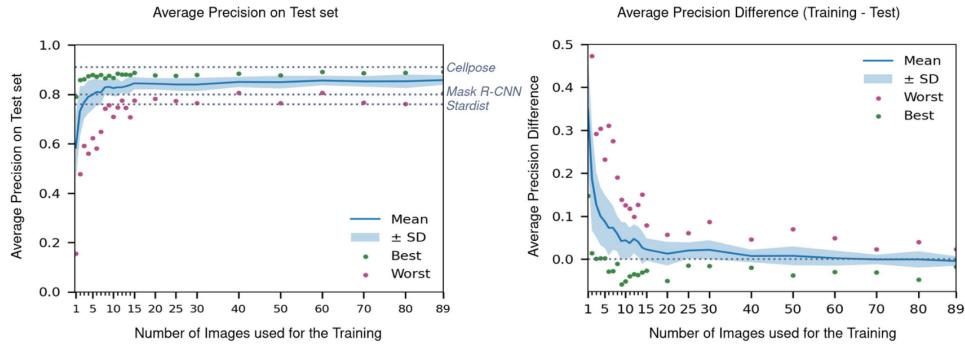


Figure 6: Comparatif de performances entre Kartezio et les autres technologies utilisées
[6]

Le graphique a nous montre plusieurs choses intéressantes. Premièrement, Kartezio, même avec peu d'images, rivalise avec les modèles comme Cellpose, Mask R-CNN ou Stardist qui utilisent tous du Deep Learning et qui sont donc black boxes et moins interprétables et explicables. Comme dit précédemment, ce genre de résultats avec si peu d'images est très utile dans le domaine biomédical car les images annotées sont précieuses.

Le graphique b met en avant l'évolution de l'overfitting par rapport au nombre d'images utilisées pour le train. Nous pouvons ici observer qu'au bout d'une dizaine ou d'une quinzaine d'images la généralisation au jeu de test commence à être correcte avec une différence de précision moyenne de 0.01 qui tend ensuite vers 0 quand nous nous approchons des 65 images. Cela montre encore une fois la capacité de Kartezio à être efficace sur des petits datasets grâce à cette robustesse à l'overfitting.

De plus, un des aspects de Kartezio qui nous intéressera, et j'y reviendrai plus tard, est que le traitement 3D est possible par empilement de slices 2D et reconstruction au niveau du Endpoint.

2.3 MAGE

MAGE (Multimodal Adaptive Graph Evolution) est un framework implémenté sous forme de package Julia [7]. L'idée globale derrière ce framework est d'implémenter une extension de CGP qui pourrait traiter plusieurs types de données différents dans une même pipeline. Par exemple, utiliser des fonctions nécessitant des strings en entrée, d'autres nécessitant des images etc. Une extension de CGP existait déjà pour gérer cette modalité. Cette extension est nommée Mixed-Type Cartesian Genetic Programming (MT-CGP). Cependant, elle présente quelques défauts sur lesquels je vais revenir plus tard dans les comparaisons qui seront effectuées avec MAGE. Je vais donc présenter l'article intitulé **Multimodal Adaptive Graph Evolution**, écrit par Camilo De La Torre, Sylvain Cussat-Blanc, Dennis G Wilson et Kevin Cortacero.[7]

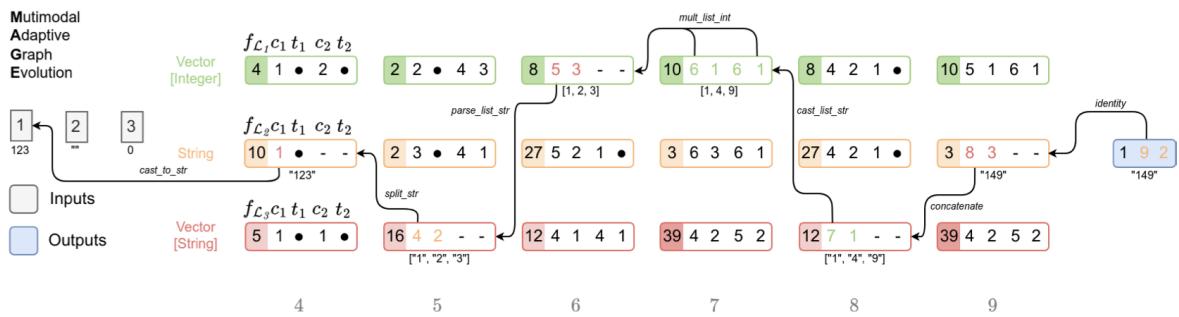


Figure 7: Exemple de graph de MAGE
[7]

Sur ce schéma, nous pouvons voir une implémentation de MAGE visant à résoudre le problème du *Squared Digits*. Le but étant ici de transformer l'integer 123 que le programme reçoit en entrée en une chaîne de caractère avec chaque digit de l'integer élevé au carré donc "149". Nous allons donc ici découvrir le fonctionnement de MAGE à travers cet exemple. La logique implémentée dans MAGE est la suivante: un chromosome par type de retour de fonction. Nous l'observons ici avec trois types de chromosomes distincts : Vector[Integer], String, Vector[String]. Chacun de ces types a ses propres fonctions et son propre sous-graphe, c'est comme ça que les conflits de type sont évités dans MAGE. Chaque noeud est formé comme suit : $(f_{L_i}, c_1, t_1, c_2, t_2)$ où :

- f_{L_i} est une fonction issue de la librairie L_i .
- c_1 représente l'indice du noeud auquel ce nœud est connecté
- c_2 idem que pour c_1
- t_1 correspond au type du chromosome qui contient les entrées
- t_2 idem que pour t_1

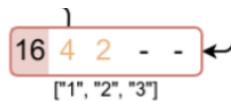


Figure 8: Zoom sur le noeud 5 de Vector[String]
[7]

Prenons l'exemple du nœud 5 de **Vector[String]**, dans ce cas nous avons $c_1 = 4$ ce qui signifie que la fonction d'indice 16 utilisée par ce nœud 5 va chercher son entrée au nœud numéro 4, le 4 est en gris en bas sur le schéma et tous les nœuds au-dessus de ce numéro 4 sont les nœuds 4 de leurs types respectifs. Pour le nœud 5, nous avons également $t_1 = 2$ ce qui signifie que le type du nœud auquel il est connecté est le type 2 qui est **String**. C'est grâce à la combinaison des paramètres c_1 et t_1 (également c_2 et t_2 , le cas échéant) que nous pouvons déterminer les connexions entre nœud dans le graphe. Ici notre nœud 5 des **Vector[String]** est connecté à la sortie de la fonction d'indice 10 des **String** car elle est la fonction utilisée dans le nœud 4 du type **String**.

Comment MAGE résout le problème des *squared digits* ? Il applique dans un premier temps la fonction *cast_to_str* sur l'entrée 123 ce qui donne la chaîne de caractères "123" ensuite la fonction *split_str* qui isole chaque caractère de cette liste dans un **Vector[String]** ["1","2","3"]. La fonction *parse_list_str* transforme ensuite le **Vector[String]** en **Vector[Integer]** [1,2,3]. La fonction *mult_list_int* prend en entrée [1,2,3] deux fois et multiplie ces deux vecteurs termes à termes ce qui donne [1,4,9]. Ce **Vector[Integer]** va être casté en **Vector[String]** par la fonction *cast_list_str* ce qui donne ["1","4","9"]. La fonction *concatenate* nous permet d'obtenir la chaîne de caractères attendue "149" et ce "149" est retourné à l'organe gérant l'évolution grâce à la fonction *identity*.

Effectuons maintenant une comparaison entre MAGE et MT-CGP sur certains problèmes du dataset PSB2 (Programm Synthesis Benchmark 2). Le PSB2 est un benchmark contenant plusieurs problèmes classiques de la littérature dont le *Squared digit* fait partie. Ce benchmark est beaucoup utilisé pour évaluer la qualité des outils en CGP. L'objectif de MAGE était d'améliorer l'approche multi types de CGP qui avait déjà été entreprise par MT-CGP c'est pourquoi il fait office de référence de comparaison.

Conditions de test :

- 20 runs indépendantes par problème
- Considéré comme solved si tous les tests de validation sont passés
- Stratégie d'évolution ($1 + \lambda$)
- Initialisation du graphe aléatoire
- Fitness : nombre de tests correctement résolus

Les paramètres utilisés pour chaque run sont récapitulés dans la figure 9 ci-dessous.

Parameter	MAGE	MT-CGP
Generations	20000	20000
Lambda	10	10
Nodes	30	30*types
Mutation rate	1	1
Output Mutation rate	-	-
Arity	3	3
n train samples	200	200
n test samples	2000	2000

Figure 9: Paramètres de test pour chaque run [7]

Analysons maintenant les résultats obtenus sur la figure 10 ci-dessous.

Problem	Solved		Generation	
	MAGE	MT-CGP	MAGE	MT-CGP
Basement	12	1	4278	18339
CamelCase	20	15	461	10514
Coin Sums	11	0	10684	-
FizzBuzz	17	0	6763	-
Fuelcost	6	0	1191	-
GCD	5	0	2217	0

Figure 10: Comparatif MAGE vs MT-CGP sur le dataset PSB2 [7]

Ce tableau met en évidence le fait que MAGE obtient de bien meilleurs résultats en termes de nombre de runs réussies sur les problèmes Basement et CamelCase. MAGE réussit douze fois plus de runs que MT-CGP pour le problème de Basement et 1.5 fois plus de runs pour le problème CamelCase. De plus, pour ces deux mêmes problèmes MAGE obtient son résultat en effectuant beaucoup moins de générations : Un peu plus de quatre fois moins pour le problème Basement et plus de 22 fois moins de générations pour le problème CamelCase. Plus important encore, MAGE arrive à résoudre des problèmes sur lesquels MT-CGP échoue systématiquement (Coinsums, FizzBuzz, Fuelcost, GCD).

Le dataset PSB2 est un dataset contenant des problèmes de programmation symbolique. Or MAGE a été développé pour pallier notamment les difficultés rencontrées par CGP et MT-CGP sur les problèmes de traitement d'images. C'est pourquoi nous allons maintenant découvrir un article que j'ai également étudié pendant mon travail bibliographique. Les auteurs se concentrent cette fois sur l'application de MAGE aux problèmes du dataset Atari Games qui sont des problèmes de traitement d'image.[8]

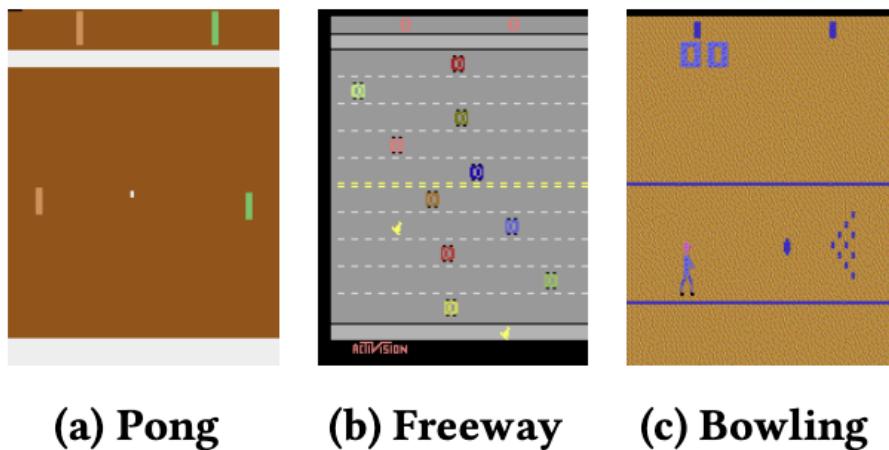


Figure 11: Atari games utilisés pour cette article [8]

Voici, sur la figure ci-dessus, les trois jeux utilisés pour mettre à l'épreuve MAGE.

Le jeu **Pong** est une adaptation du célèbre jeu de Pingpong. Ici le but est de déplacer verticalement la raquette (les petites barres oranges et vertes de chaque côté) que l'on contrôle pour renvoyer la

Segmentation d'IRM 3D par méthodes évolutives et outillage de tiling/stitching pour l'imagerie 10 médicale à grande échelle

balle à la raquette de l'adversaire. Aucune action supplémentaire n'est nécessaire, la balle est renvoyée automatiquement une fois qu'elle touche la raquette.

Dans le jeu **Freeway**, le joueur contrôle un poulet qui doit traverser une route composée de 10 voies où des voitures circulent. Le but est d'éviter les voitures lors de la traversée de cette route. Le score augmente à chaque traversée réussie. Les actions possibles pour le joueur sont soit d'avancer soit de reculer.

Le troisième et dernier jeu est le **Bowling**, le but du jeu est tout simplement de lancer une boule pour faire tomber des quilles. Le joueur contrôle l'alignement horizontal entre le lanceur de la boule et les quilles mais également le timing du lancer.

Game	Action Set	n_{out}	Screen Crop
Pong	{0, 4, 5}	3	[15 : 77, 1 : 84]
Freeway	{0, 1, 2}	3	[11 : 74, 6 : 84]
Bowling	{0, 1, 2, 3}	4	[14 : 72, 6 : 78]

Figure 12: Actions disponibles pour l'agent dans chaque jeu [8]

Dans la figure ci-dessus, nous pouvons observer les actions disponibles pour chaque jeu ainsi que la proportion de l'image qui est gardée pour le traitement.

Action Set représente la liste des actions disponibles pour chaque Atari game, c'est une sélection effectuée par les auteurs de l'article. Le paramètre n_{out} représente le nombre d'actions disponibles pour notre cas. Et **Screen Crop** la partie de l'image totale qui sera utilisée dans MAGE.

Pour **Pong**, les actions disponibles dans Atari sont NOOP, FIRE, LEFT, RIGHT, LEFTFIRE, RIGHTFIRE et les auteurs gardent 0,4,5 ce qui correspond aux actions NOOP, LEFTFIRE, RIGHTFIRE. Ces actions dans cette configuration précise, signifient respectivement : rester immobile, bouger la raquette vers le haut, bouger la raquette vers le bas.[8]

Pour **Freeway**, les actions disponibles sont NOOP, UP, DOWN et les auteurs utilisent les trois. Ce qui permet respectivement de : rester immobile, avancer sur la route, reculer sur la route.

Et enfin pour le jeu du **Bowling** les actions disponibles sont NOOP, FIRE, UP, DOWN, UPFIRE, DOWNFIRE mais les auteurs gardent seulement les quatre premières actions qui permettent respectivement de : ne rien faire, lancer la boule, se déplacer vers le haut (gauche de la piste), se déplacer vers le bas (droite de la piste).

La reward est fixée en fonction de la frame actuelle et du jeu. Pour **Pong** la reward est basée sur le fait de gagner des points en réussissant à marquer le point en faisant passer la balle derrière la raquette adverse ou d'en perdre en laissant passer la balle derrière notre raquette.

Dans **Freeway**, les récompenses sont glanées en réussissant à traverser la route.

Et enfin dans le cas du **Bowling**, la récompense est maximale en faisant tomber les quilles.

Analysons maintenant ce que nous apprend cet article sur les résultats obtenus sur chaque jeu.

Pour **Pong**, les résultats sont mitigés. La meilleure politique établie par MAGE obtient un score de 21 qui est le score maximal pour ce jeu. Cependant, cette politique se généralise très mal pour les autres scénarios. Le score moyen sur les 100 seeds est de ($\mu_{100\text{seeds}} = 0.84$). De plus, l'écart-type sur ces 100 seeds est de ($\sigma_{100\text{seeds}} = 21.1$) et le meilleur score sur ces 100 seeds est $\text{best}_{100\text{seeds}} = 21$. Nous voyons bien ici que la moyenne est très faible alors que le score maximum est très grand ce qui montre qu'il n'a pas été atteint beaucoup de fois et c'est confirmé par l'écart-type qui nous permet de conclure que sur ces 100 runs les résultats doivent ressembler à un nombre élevé de runs quasi nuls et une run qui maximise le score. Après analyse, les auteurs ont remarqué que cette politique exploite des comportements répétitifs liés aux seeds sur lesquels MAGE s'est entraîné. C'est donc un optimum local. Nous noterons tout de même que cette situation met en avant le caractère interprétable et explicable des comportements appliqués par MAGE. En effet, les auteurs ont pu découvrir ce local optimum grâce à l'analyse des graphes produits par MAGE en plus des données statistiques vues juste avant.

Place maintenant à **Freeway**, il faut savoir que le score moyen réalisé par les humains à ce jeu est de 29.6.[9] D'autre part le résultat moyen obtenu par les méthodes d'apprentissage par renforcement utilisant les réseaux de neurones profonds obtient un score moyen de 34.[10] Dans ce cas-là, la politique optimale trouvée par MAGE obtient les statistiques suivantes : $\mu_{100\text{seeds}} = 22.5$, $\sigma_{100\text{seeds}} = 1.6$, $\text{best}_{100\text{seeds}} = 26$. Nous pouvons constater ici que le score max est de 26 mais le score moyen est de 22.6 ce qui montre une généralisation plutôt correcte pour ce jeu. Le score moyen est inférieur au score moyen humain mais est jugé satisfaisant par les auteurs de l'article notamment car MAGE n'utilise aucun réseau de neurones ni aucune technologie black box. L'analyse de la politique a montré que l'agent avait tendance à vouloir traverser jusqu'en haut de l'écran le plus vite possible, plus de détails sur la manière dont l'agent fait son choix avec les éléments qu'ils jugent être des zones d'intérêts sont disponibles dans l'article.[8]

Pour le dernier jeu du **Bowling**, MAGE a obtenu de meilleurs résultats que pour les deux précédents jeux. Le score moyen d'un humain est 160.7 [9] et le score moyen des méthodes d'apprentissage par renforcement utilisant les réseaux de neurones profonds est de 200 [11]. Sur les jeux de tests, MAGE rivalise avec l'état de l'art avec un score moyen de $\mu_{\text{trainseed}} = 211$ et un écart-type de $\sigma_{\text{trainseed}} = 11.5$. Une fois que cette politique est généralisée aux autres seeds nous constatons une baisse des résultats avec un score moyen de $\mu_{100\text{seeds}} = 172,2$, un écart-type de $\sigma_{100\text{seeds}} = 37.4$ et un meilleur score de $\text{best}_{100\text{seeds}} = 223$. Certes, le score moyen obtenu par MAGE ne dépasse pas les performances des méthodes de l'état de l'art utilisant les RNNs mais il dépasse le score moyen obtenu de par l'humain.

Les auteurs de l'article finissent par conclure que GP a du potentiel pour la mise en place de politique de contrôle visuel nécessitant de la transparence. En soulignant qu'il est possible d'obtenir des politiques performantes tout en maintenant la transparence totale dans les choix effectués pour générer le pipeline. Ils notent également une limite de robustesse des agents face à la stochasticité des environnements Atari. Ils évoquent aussi le fait que cette robustesse n'était pas un objectif spécifique pendant l'entraînement et que prendre en compte cette variable pendant cette phase pourrait rendre les solutions plus robustes.

3 Phase 1 : Stage à l'IRIT Capitole

3.1 Objectif du stage et environnement de travail

L'enjeu du projet était de procéder à la segmentation d'un pigment dans le cerveau appelé Neuromélanine à partir d'images IRM de patients et témoins. Ce pigment pourrait être une piste pour étudier la maladie de Parkinson. Étant dans un contexte médical, nous ne pouvons pas nous permettre d'utiliser des outils contenant des blackboxes et devons nous concentrer sur des outils les plus explicables, transparents et interprétables possibles. Nous utiliserons donc des générateurs de pipelines 2D/3D tels que Kartezio ou MAGE. Pour ce qui est de l'environnement, le stage s'est déroulé à l'IRIT-UT Capitole au sein de l'équipe REVA [12]. Les outils que nous utilisons reposent donc sur GP et CGP que nous avons vus précédemment. Deux jours de présentiel étaient imposés, le lundi et le mercredi. Les réunions d'équipe avaient lieu le lundi et le mercredi me permettant de travailler quelques fois avec Gaudillat Elliott qui travaillait sur un projet lié au mien ainsi que sur l'implémentation de la 3D dans l'outil Kartezio. Il y avait des réunions hebdomadaires ou bimensuelles selon les besoins. Avec Yuri Lavinas, chercheur postdoctoral travaillant à l'IRIT ou bien Sylvain Cussat-Blanc mon maître de stage et responsable de l'équipe REVA.

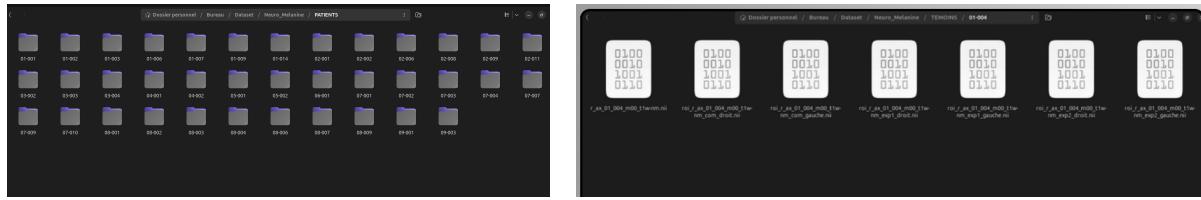


Figure 13: Architecture du Dataset

Le dataset se présente tel quel : deux dossiers, un pour les PATIENTS et un pour les TEMOINS. À l'intérieur de ces dossiers il y a un dossier par individu nommé via un identifiant. Et à l'intérieur de chacun de ces dossiers il y a sept fichiers différents que nous allons parcourir de la gauche vers la droite:

- Le premier fichier contient l'IRM complet
- Le deuxième contient la fusion des masques droits effectués par les deux experts.
- Le troisième fichier contient la fusion des masques gauches effectués par les deux experts.
- Le quatrième fichier contient le masque effectué par le premier expert correspondant au pigment de neuromélanine sur la partie droite
- Le cinquième fichier contient le masque effectué par le premier expert correspondant au pigment de neuromélanine sur la partie gauche
- Le sixième fichier contient le masque effectué par le deuxième expert correspondant au pigment de neuromélanine sur la partie droite
- Le septième fichier contient le masque effectué par le deuxième expert correspondant au pigment de neuromélanine sur la partie gauche

Le choix a donc été fait de fusionner les masques droits et gauches de chaque expert afin d'obtenir un seul masque complet pour chaque individu.

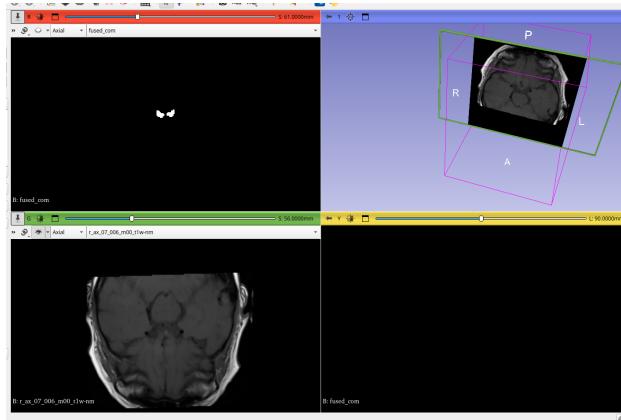


Figure 14: Visualisation de l'IRM complet et du masque dans Slicer 3D

Ce sont donc les deux principaux fichiers que nous utiliserons, le masque servira de vérité terrain et l'IRM complet sera le sujet sur lequel nous appliquerons la segmentation. Tous les fichiers évoqués depuis le début de cette partie sont au format *.nii*. Ce format est courant en neuroimagerie et contient des images 3D avec les axes suivants: Axial, Coronal, Sagittal correspondant aux coupes du cerveau. Chacun de ces fichiers contient 181 slices par axe.

3.2 Traitement seulement sur la coupe Axiale

Une des étapes primordiales maintenant est la préparation du dataset. Comme dit précédemment, le masque a été fusionné dans un seul fichier pour chaque patient. Tous ces fichiers ont été convertis en PNG dans un premier temps. Nous avons donc pour chaque fichier *nii* un dossier du même nom contenant des slices PNG allant de slice_000 à slice_180 regroupant donc nos 181 slices. De plus, Sylvain m'a conseillé de concentrer le traitement sur une zone en particulier de l'IRM de 50x50 pixels car le pigment se trouve obligatoirement dans cette zone s'il est présent.

3.2.1 Premier test sans traitement particulier

Une fois le dataset préparé et adapté à Kartezio j'ai effectué une première salve de tests avec les fonctions disponibles dans Kartezio sans choix particulier afin d'évaluer la situation.

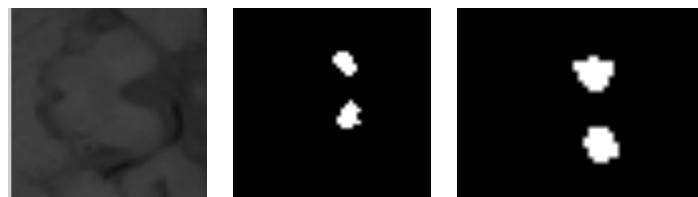


Figure 15: Entrée VS Vérité terrain

Voici un exemple de résultat dans cette configuration sur une slice où le pigment est censé être présent. Nous avons besoin de métriques pour évaluer la qualité de la prédiction même si ce n'est pas facilité par le fait que la vérité terrain n'est pas exacte. En effet, comme mentionné précédemment, les masques ont été effectués par deux experts différents et pourtant les masques qui en résultent ne sont pas les mêmes et une combinaison des deux résultats a été utilisée pour obtenir notre vérité terrain. Le problème avec cette configuration est que nous ne pouvons pas

dévaluer une segmentation sauf erreur grotesque qui saute aux yeux. L'idéal serait d'instaurer des feedbacks réguliers avec les experts pour qu'ils donnent leur avis sur certaines prédictions obtenues.

3.2.2 Nouvelle salve de tests avec utilisation de la Sliding Window

Une nouvelle salve de tests a été effectuée avec

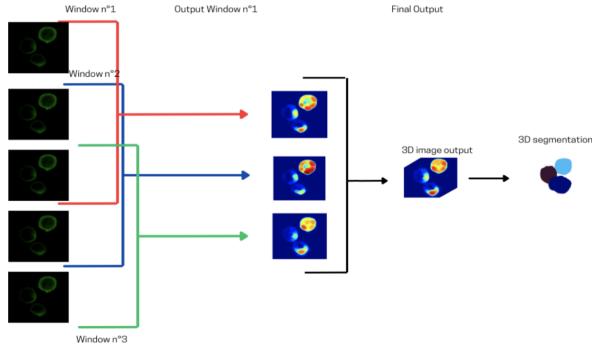


Figure 16: Schéma de fonctionnement de la Sliding Window

Le but de cette composante est d'ajouter du contexte pour chaque slice. Ce contexte est obtenu en ajoutant la slice précédente et la slice suivante pendant le traitement de la slice actuelle.

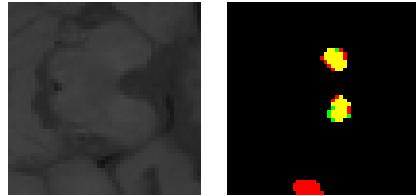


Figure 17: Entrée VS Superposition Vérité terrain / masque

Voici les paramètres avec lesquels ces résultats ont été obtenus:

- Le modèle a été entrainé sur 3 images de train et testé sur ces mêmes images
- Slices 40 à 80 en entrée (les slices où le pigment se situe pour ces images sont comprises entre 55 et 65)
- 30 000 itérations

J'ai implémenté un script qui crée un dossier nommé en fonction des paramètres du modèle, par exemple : *Tester_sur_le_dataset_train_3_individu_30000_slice40to80* cela nous permet de nous retrouver au niveau des tests. De plus, dans ce dossier il y a un fichier *txt* contenant encore une fois tous les paramètres du modèle ainsi que les métriques individu par individu et sur l'échantillon d'individu total. Il y a ensuite un dossier par individu contenant pour chacun d'entre eux les slices utilisées en entrée par le modèle ainsi que les masques produits en sortie, et enfin un dossier contenant la superposition entre la prédiction et la vérité terrain pour chaque slice.

Nous pouvons observer dans la figure ci-dessus le code couleur que j'ai mis en place pour les fichiers générés en sortie. En vert, la vérité terrain. En rouge, la prédiction. Et enfin en jaune, l'intersection entre la prédiction et la vérité terrain. Nous identifions ici une segmentation incorrecte de notre

modèle pour la partie basse de l'image. Ce n'est pas très inquiétant dans la mesure où ce genre d'erreur pourra être géré avec un post-traitement plus tard par exemple. De même, nous notons ici que pour les pigments notre segmentation diffère de quelques pixels. Nous entrons ici dans la zone de flou que je notifiais précédemment où il faudrait soumettre ce résultat à une contre-analyse des experts.

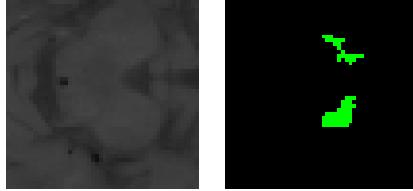


Figure 18: Entrée VS Superposition Vérité terrain / masque

Voici un cas un peu plus problématique pour ce même individu où nous ne détectons rien alors que nous avons une vérité terrain qui le demande. Pour cet individu nous avons une précision de 42% et un recall de 50%. Cela rejoint ce que nous observions sur ces exemples, le recall faible vient du fait que comme vu sur la figure 18 nous avons un nombre de faux négatifs élevé sur certaines slices. Sur les 6 slices de cet exemple où nous sommes censés détecter les pigments, nous en avons deux où nous ne détectons rien ce qui fait augmenter le nombre de faux négatifs en flèche et donc baisse drastiquement la valeur du recall. La faible précision vient plutôt du comportement que nous observons sur la Figure 17, avec des détectons sur les bordures là où la vérité terrain ne comporte rien. Cela fait augmenter le nombre de faux positifs et donc baisser la précision.

NB: rappel formules:

- Recall : $\frac{TP}{TP+FN}$ où TP représente les vrais positifs et FN les faux négatifs
- Précision: $\frac{TP}{TP+FP}$ où FP représente les faux positifs

Il faudrait générer plusieurs runs avec les mêmes paramètres afin de pouvoir généraliser ses comportements. Mais en raison de contraintes logicielles ce n'était pas possible. C'est pourquoi j'ai préféré tester plusieurs configurations sur de très petits échantillons. En m'assurant également de rendre toutes ces métriques et résultats visuels accessibles facilement pour que quelqu'un puisse récupérer le projet et analyser les résultats sans avoir besoin de se plonger dans le code.

Le fait que la phase de train soit très longue ne me permet pas d'utiliser beaucoup d'images pour mes tests car je ne peux pas me permettre de ne pas utiliser le PC pendant plusieurs heures. C'est pourquoi quand j'avais besoin de faire des tests conséquents (au moins 10 000 itérations) je laissais le modèle tourner la nuit sur mon PC. Dans tous les cas, je ne pouvais pas utiliser le dataset entier donc je me cantonnais la plupart du temps à quelques images de train et je testais sur ces mêmes images. Je ne testais normalement pas sur la partie testing du dataset car vu le faible nombre d'images que j'utilisais pour le train, la généralisation du traitement ne pouvait pas être de très grande qualité.

Cependant, j'ai pu réaliser un test avec les 16 images de train et le tester sur 3 images du dataset de test. J'ai laissé tourner le modèle un week-end entier afin de rendre cela possible et voici les résultats. Rappel des paramètres:

- Le modèle a été entrainé sur 16 images de train et 3 images de test issues du dataset de test
- Slices 40 à 80 en entrée (les slices où le pigment se situe pour ces images sont comprises entre 55 et 65)

- 30 000 itérations

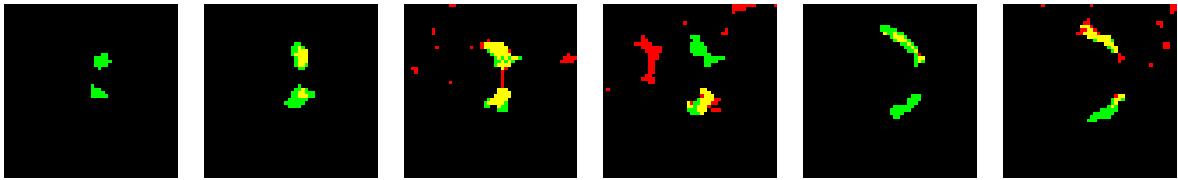


Figure 19: Superposition vérité terrain / prédition

Voici les résultats obtenus pour le premier individu de test avec cette configuration sur les slices où nous sommes censés localiser les pigments. Elles sont affichées dans l'ordre croissant du numéro de slice, 59 à gauche 64 à droite. Il y a eu un problème lors de la run et les FP et les TP n'ont pas été bien calculés dans la fonction de superposition donc les statistiques de Précision et Recall ne sont pas exploitables. Nous allons donc faire une analyse visuelle des superpositions car je n'ai pas eu le temps de régler le problème de comptage. Nous voyons bien ici que le comportement obtenu avec le dataset de train se généralise avec difficulté sur le dataset de test. Aucune partie du pigment de la vérité terrain n'est reconnue sur la slice 59, sur la slice 60 les pigments segmentés contiennent bien moins de pixels. La slice 61 contient l'essentiel des pixels contenus dans la vérité terrain avec quelques faux positifs aux alentours. Pour les trois dernières slices à chaque fois une des parties du pigment, gauche ou droite est mal reconnue.

Encore une fois il est difficile de tirer une conclusion sur une seule run, il faudrait généraliser le fonctionnement sur plusieurs runs pour pouvoir se positionner sur la pertinence de la configuration. Nous noterons quand même ici que le modèle a été entraîné sur les 16 images de train, c'est-à-dire le maximum pour la répartition actuelle du dataset qui est de 16 images de train et 40 de test. Cette répartition avait été effectuée par mon maître de stage et se justifie par le fait qu'aussi bien KarteZio et MAGE ont des prédispositions à généraliser efficacement les comportements appris sur de petits jeux de données. Ces frameworks n'ont pas besoin de très gros datasets d'entraînement. Une des pistes peut tout de même être de modifier la répartition train/test de notre dataset.

3.2.3 Traitement sur les images entières

Bien que le rescaling des images sur la zone d'intérêt de 50x50 pixels montre de bons résultats, il est également intéressant de voir si avec les images entières nous obtenons de meilleurs résultats. Le fait d'utiliser les images entières permet d'avoir plus de contexte et peut permettre de mieux repérer les pigments en ajoutant de la nuance entre la zone d'intérêt et le reste. Je n'ai pu faire qu'un seul test car l'utilisation des images entières rend l'entraînement très long à paramètres équivalents. Voici les paramètres utilisés:

- Le modèle a été entrainé sur 16 images de train et 3 images de test issues du dataset de train
- Slices 40 à 80 en entrée (les slices où le pigment se situe pour ces images sont comprises entre 55 et 65)
- 10 000 itérations

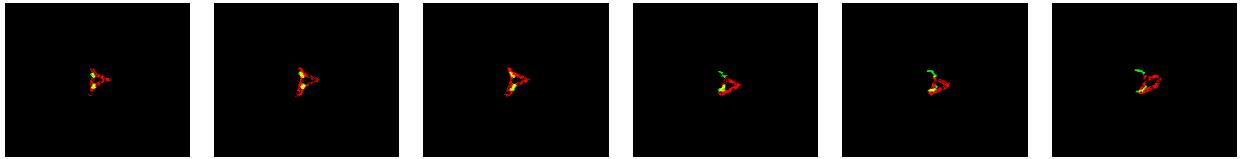


Figure 20: Superposition vérité terrain / prédition

Voici les résultats obtenus pour le premier individu de test avec cette configuration sur les slices où nous sommes censés localiser les pigments. Elles sont affichées dans l'ordre croissant du numéro de slice, 60 à gauche 65 à droite. Le Recall pour cet individu est de 48% ce qui est cohérent car on peut voir que la plupart des pixels du masque sur les trois premières slices sont segmentés. Cependant pour les trois dernières slices, la partie gauche (haut) du pigment n'est pas segmentée. Il y a donc relativement peu de faux négatifs vu que ce sont les seules slices où il fallait détecter les pigments.

La Précision est beaucoup plus basse, 6%. En effet, nous observons bien les faux positifs représentés en rouge sur chaque slice.

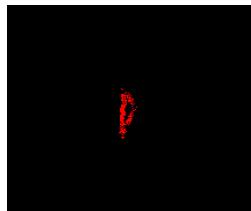


Figure 21: Slice 57 de ce même individu

De plus, il y a une dizaine de slices comme celle-ci. Nous y voyons une segmentation de pigment qui ne devrait pas avoir lieu ce qui amène à une explosion du nombre de faux positifs. Cela fait donc drastiquement baisser la valeur de la Précision.

Place maintenant à une autre configuration prenant en compte cette fois les trois axes.

3.3 Traitement sur les trois coupes en même temps

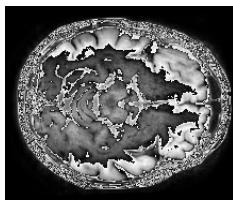


Figure 22: Vue axiale

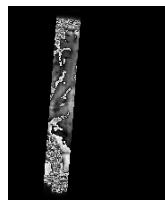


Figure 23: Vue sagittale



Figure 24: Vue coronale

Voici les différentes vues de la même slice selon chaque axe: axial, sagittal, coronal. Chaque vue représente un axe différent de l'IRM de cerveau entier. Dans la partie précédente, seule la coupe axiale était utilisée. Or, comme nous pouvons le voir ci-dessous avec ces trois figures représentant une slice du masque contenant la zone à segmenter. Selon chaque axe le masque n'a pas la même forme et ne se situe pas à la même slice, donc il faut traiter les trois axes simultanément pour perdre le moins d'informations possible.



Figure 25: Masque vue axiale slice 60



Figure 26: Masque vue sagittale slice 80

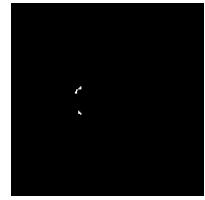


Figure 27: Masque vue coronale slice 100

Pour prendre en compte le traitement des trois axes il a fallu de nouveau adapter le dataset pour qu'il puisse exploiter au mieux les fonctionnalités déjà présentes dans Kartezio. Des readers étaient déjà implémentées dans Kartezio afin de prétraiter des images 3D au format *tiff* en créant une pile d'image 2D par coupe. Un trainer 3D existe également permettant d'entraîner le modèle à partir de ces données d'entrées. Les fichiers *nii* ont donc été convertis en fichiers *tiff* puis quelques ajustements ont été effectués dans le code de Kartezio en local afin de rendre le modèle utilisable. Malheureusement, la multiplication par trois du nombre de slices à traiter rendait les tests impossibles pour mon PC même avec des petites fractions du dataset de quelques images. Le code est fonctionnel, prêt à tourner et les résultats seront formatés dans un fichier *.zip* organisé les rendant facile à analyser. Le résultat attendu pour cette configuration à 3 axes est une meilleure segmentation grâce au nouveau contexte. En effet, grâce aux trois piles d'images 2D (1 par axe) et la sliding window vu précédemment le contexte pour chaque slice sera déterminant.

Si nous résumons les facteurs sur lesquels nous pourrons jouer dans le futur pour améliorer la segmentation :

- Paramètres internes Kartezio(nombre de runs, decoder, endpoint, preprocessing)
- Taille des images (croppé ou non)
- Raisonnement par zone d'intérêt

Des approches différentes peuvent être envisagées sur les sets d'entraînement et de tests par exemple nous pourrions entraîner le modèle sur les images croppées et le tester sur des images de tests entières ou inversement.

3.4 Transition sur l'outil MAGE

Kartezio et MAGE étant l'un comme les autres des implémentations de CGP il serait intéressant de les comparer pour voir quel framework nous permet d'avoir les meilleurs résultats. MAGE étant codé en Julia, qui est un langage que je ne connaissais pas, la prise en main était compliquée. De plus, j'étais pris par le temps donc je n'ai pas eu le temps de finir mon implémentation. J'ai quasiment fini la préparation et les premiers traitements du dataset, il reste à créer la bibliothèque de fonctions que nous souhaitons rendre disponible pour l'évolution.

4 Phase 2 du stage : Travail au sein de la start-up Mantalys

4.1 Objectif du stage

L'enjeu de ce stage est d'établir un processus de Stitching pour l'application Mantaplex. Mantaplex traite de très grosses images de type WSI (Whole Slide Image). Ce type d'image représente l'entièreté d'une lame de microscope. Le problème étant que ce sont des images d'une résolution extrêmement élevée donnant des images dont les tailles sont dans l'ordre du GB.[13] Le fait que ces images soient si lourdes pose une préoccupation majeure, comment les charger en RAM ? Pour effectuer les tâches que requiert habituellement ce type d'image comme de la segmentation par exemple, un chargement en RAM est nécessaire. Dans le cas des WSI aucune technologie accessible ne permet d'effectuer ce genre de tâches avec une machine accessible (pas de puissants calculateurs) sans avoir recours au Tiling et au Stitching. Le tiling consiste à décomposer la WSI en plusieurs petites images appelées tiles pour effectuer la segmentation sur chaque petite tile ce qui permet de charger seulement des portions de l'image entière en RAM. Le stitching intervient après et consiste à réassembler ces tiles contenant les cellules segmentées pour reformer la WSI initiale mais cette fois avec la segmentation effectuée.

4.2 Environnement de travail et état d'avancement du projet

Mantalys est composé de trois associés :

- Brienne : Post-doc en Biologie
- Thomas : Ingénieur SRI diplômé d'UPSSITECH
- Kévin : Docteur en IA, créateur de Kartezio et également ingénieur SRI diplômé d'UPSSITECH et également responsable de mon stage

La start-up ayant été fondée récemment, le Fablab d'Esquirol sert actuellement de bureau. Brienne étant responsable de toute la partie biologie du projet ne code pas donc nous étions trois à coder. Kévin et Thomas travaillaient sur d'autres aspects du projet pendant que je me chargeais de mettre en place le stitching sous la tutelle de Kévin. Au moment de mon arrivée le 15 juin, le tiling était déjà implémenté et il ne restait plus qu'à faire le stitching. Pour ce qui est de l'organisation de la semaine nous étions sur deux jours de présentiel, le mardi et le jeudi généralement et le reste de la semaine en distanciel. L'organisation mise en place pour ma tâche s'inspirait du TDD (Test Driving Development). C'est-à-dire qu'il y a eu un important brainstorming avec Kévin les premiers jours pour que je comprenne bien tous les enjeux et les implications de cette tâche au sein du produit final. Puis la tâche a été découpée en plusieurs parties, pour chacune de ces parties Kévin écrivait des tests correspondants au comportement qu'il attendait et je développais donc dans le but de satisfaire ces tests au fur et à mesure. Cette approche permet de découper le travail pour ne pas s'éparpiller et améliorer la compréhension globale du projet au fur et à mesure.

4.3 État de l'art sur le stitching de grosse image

Après que Kévin m'a expliqué comment il imaginait que nous pouvions élaborer le stitching et les différentes approches qu'il avait envisagé, il m'a donné quelques librairies à étudier. Le but étant de regarder comment la concurrence a décidé de traiter ce problème afin de s'inspirer du meilleur

Segmentation d'IRM 3D par méthodes évolutives et outillage de tiling/stitching pour l'imagerie 20 médicale à grande échelle

de chaque approche. Cette partie s'est faite avant même que j'ai accès au code déjà écrit par Kévin et Thomas afin que je ne sois pas biaisé dans mon approche.

Les deux librairies à étudier étaient **Stardist** et **InstanSeg** et plus précisément comment elles géraient le cas des très grosses images. L'étude de ces librairies allait aussi me permettre d'en apprendre plus sur les enjeux du tiling et du stitching avec lesquels je n'étais pas familier.

4.3.1 Stardist[14]

Je me suis concentré sur le fichier `stardist/stardist/models/base.py` car il contenait la fonction `predict_instances_big` que j'avais identifiée comme étant le cœur du problème. En effet, Stardist ne gère pas que les très grosses images, il gère aussi les images classiques donc cette fonction n'est pas spécialement mise en avant dans les docs.

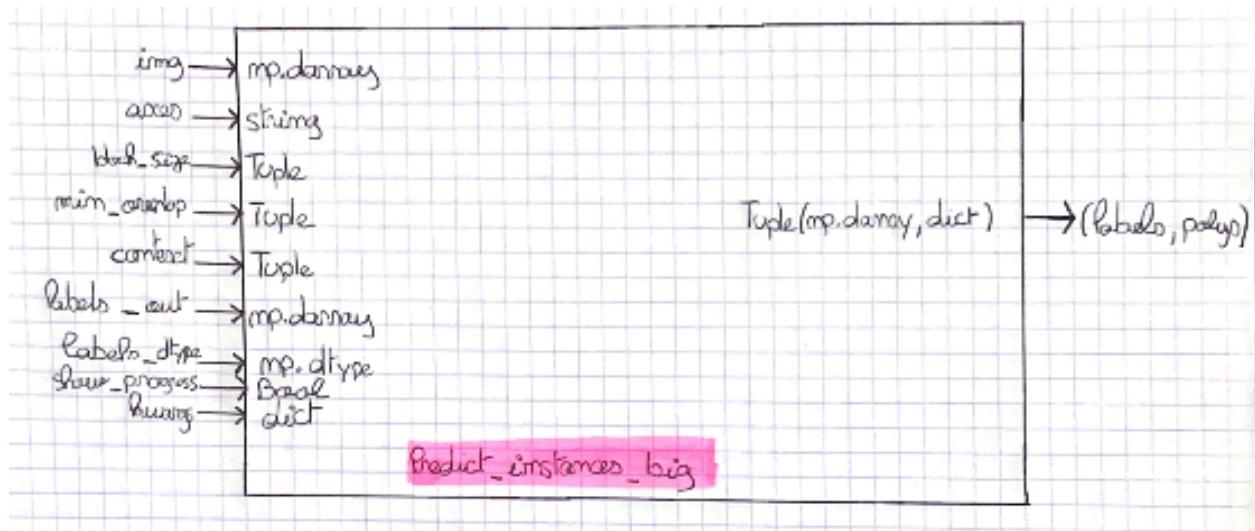


Figure 28: Diagramme de classe de la fonction `predict_instances_big`

Passons en revue chacun des paramètres ensemble:

- `img` est un tableau numpy représentant l'image entière que nous voulons traiter sur laquelle il n'est pas possible de faire la segmentation directement.
- `axes` est une chaîne de caractères de type "XY", "YX", "XYZ" etc qui a pour but d'indiquer quels axes de l'image correspondent à quelles dimensions
- `block_size` est un int/tuple de int qui détermine la taille des blocks (tiles) dans lesquelles l'image va être découpée.
- `min_overlap` est un int ou tuple de int qui représente la valeur de recouvrement en pixels qu'il doit y avoir entre les tiles nous revidendrons sur ce concept clé plus tard qui permet d'assurer que chaque objet en bord de tile est vu dans au moins une tile
- `context` est également un int ou tuple de int qui représente un peu comme l'overlap, une zone de recouvrement mais qui cette fois n'est présente que lorsque la prédiction est effectuée avec le modèle puis est retirée après. Cela permet de pallier le problème récurrent des CNNs au voisinage des bordures où les prédictions sont de très mauvaises qualités à cause des glitches dus aux effets de bords.

- `labels_out` est un tableau numpy représentant le tableau dans lequel les labels seront écrits pendant la prédiction. Si ce paramètre vaut `False`, la fonction retournera seulement le dictionnaire de polygones et aucun tableau de labels. Et si `labels_out = None` la fonction allouera un tableau numpy de la bonne taille elle-même pour les labels.
- `label_dtype` représente le type que doit avoir le label retourné par la fonction seulement si `labels_out` est `None` sinon ce paramètre ne sera pas pris en compte.
- `show_progress` est un booléen permettant l'affichage ou non d'une barre de progression pendant le traitement du block
- `kwargs` est un dictionnaire comportant les arguments qui seront transmis à la fonction `predict_instances` qui est appelée à l'intérieur de `predict_instances_big`

Observons dans la figure ci-dessous un exemple d'appel à la fonction `predict_instances_big`.

Example

```
>>> img.shape
(20000, 20000)
>>> labels, polys = model.predict_instances_big(img, axes='YX', block_size=4096,
min_overlap=128, context=128, n_tiles=(4,4))
```

Figure 29: Exemple d'appel de la fonction `predict_instances_big` de Stardist[14]

Dans cet exemple l'image totale est une image de forme $(20000, 20000)$ ce qui représente une image de 20 000 par 20 000 pixels. Les axes sont dans l'ordre "YX" ce qui veut dire que la forme de l'image est traitée comme telle $(Y, X) (20000, 20000)$ donc $Y=X=20000$. Cette image sera découpée en blocs de 4096×4096 pixels dans la fonction `predict_instances_big`. Le recouvrement doit être pris en compte donc cela donnera $\frac{20000-128}{4096-128} = 5$ soit 5 (en X) x 5 (en Y) = 25 blocs. Le calcul se base sur le fait que chaque bloc a besoin de voir 128 pixels (overlap) du bloc suivant/précédent.

4.3.2 InstanSeg[15]

L'essentiel de la gestion des grosses images se fait dans le fichier `Instanseg/instanseg/utils/tiling.py`. Ce fichier contient la fonction `_sliding_window_inference`. Cette fonction fait appel aux fonctions suivantes:

- `_chops`: permet de récupérer les indexs des tiles
- `_tiles_from_chops`: récupère les tiles à partir de leurs indexs
- `predictor`: prédit la segmentation sur chaque tile
- `_stitch`: reconstruit l'image entière avec la segmentation effectuée sur les tiles

Ce qui nous intéresse ici est donc plutôt la fonction `_stitch` pour voir comment est géré la reconstruction au niveau de l'overlap. Cette fonction utilise une liste par côté (haut, bas, gauche, droite) et si la tile touche un de ces bords les labels sur la bordure sont supprimés.

4.4 Mise en place du stitching pour Mantaplex

4.4.1 Fonctionnement globale attendu

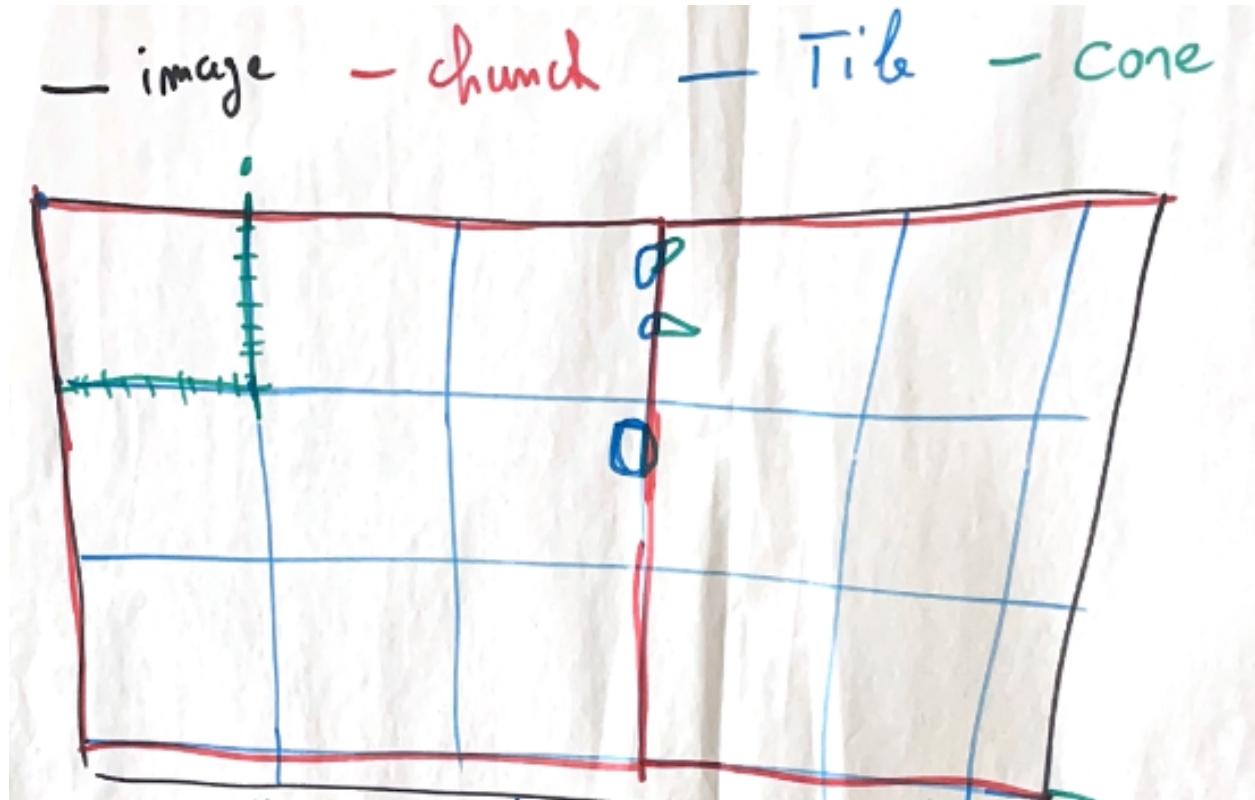


Figure 30: Schéma de la problématique globale

Comme nous pouvons le voir sur la figure ci-dessus nous pouvons voir en rouge ce que nous allons appeler à partir de maintenant "Chunks" sachant qu'un chunk est un regroupement de plusieurs tiles. Ici les tiles sont les carrés bleus, les chunks les carrés rouges. Les questions de recouvrement entre tiles sont déjà prises en compte par le code déjà élaboré par Thomas et Kévin. L'enjeu principal est la gestion des cellules visibles sur la figure ci-dessous.

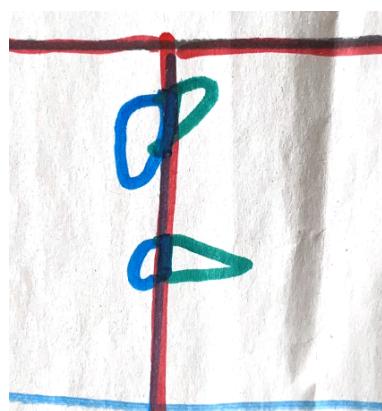


Figure 31: Zoom sur les cellules problématiques

En effet, le principal risque lors d'un stitching que ce soit avec des tiles ou des chunks est la gestion

Segmentation d'IRM 3D par méthodes évolutives et outillage de tiling/stitching pour l'imagerie 23 médicale à grande échelle

des cellules en bordures. Il y a deux cas possibles :

- Il n'y a pas d'overlap, auquel cas la cellule est simplement coupé en deux.
- Il y a de l'overlap, auquel cas la cellule risque d'être traité deux fois car chaque chunk la traitera vu qu'elle est dans la zone d'overlap.

C'est la principale préoccupation qui guidera le développement de cette fonctionnalité, quels outils et stratégies utilisés pour s'assurer que chaque cellule en bordure est reconstruite sur un seul chunk dans l'image finale. Comme mentionné précédemment, la méthode utilisée est le TDD (Test driving development). Nous allons donc voir dans la figure ci-dessous les différentes étapes clés qui ont été définies pour cette tâche.

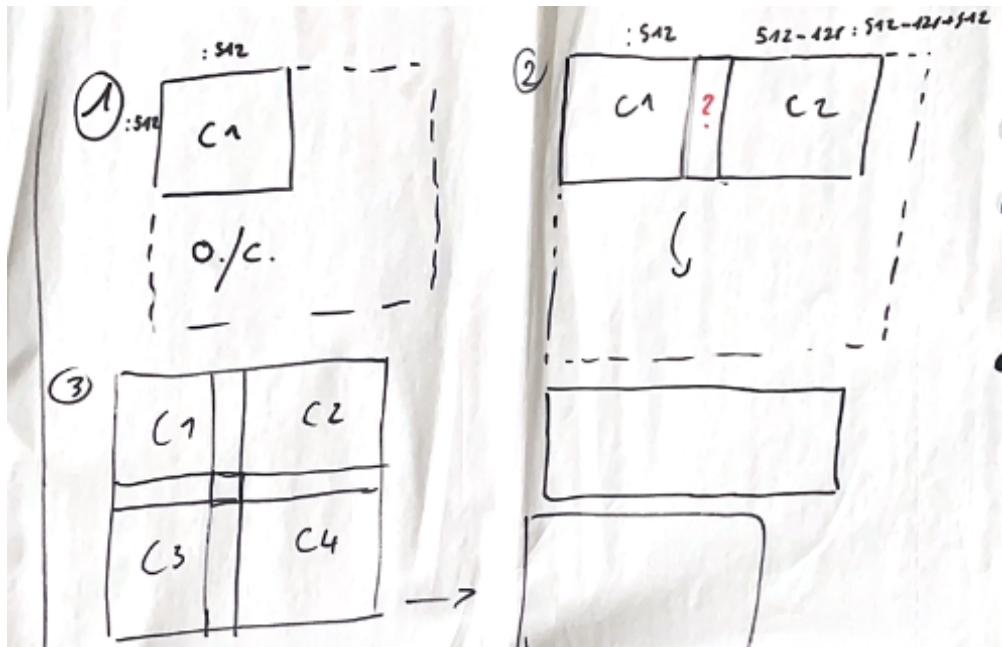


Figure 32: Étapes intermédiaires fixées pour l'élaboration de cette tâche

Les trois principales étapes sont la gestion d'un seul chunk, la gestion de deux chunks, et enfin la gestion de 4 chunks qui doit donc être généralisable à n'importe quel nombre de chunks. Nous allons détailler les enjeux de chaque étape dans sa sous-partie respective.

4.4.2 Première et deuxième étapes : gestion avec chunks

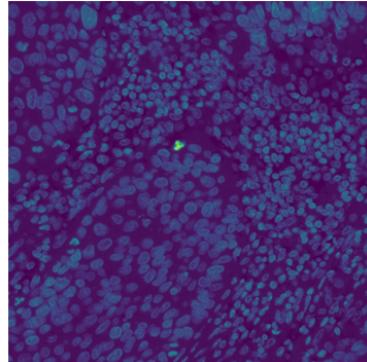


Figure 33: Image entière avant découpage

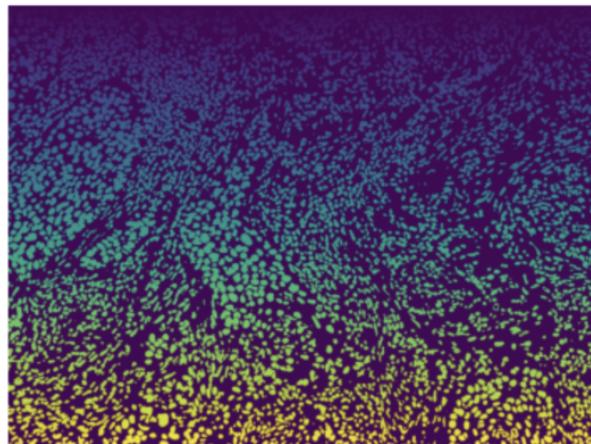


Figure 34: Image labellisée

Voici le point de départ. Nous pouvons ici voir l'image originale ainsi que l'image labellisée. Dans notre fonction d'entrée pour la fonction de stitching nous n'aurons malheureusement aucune de ces deux images. Le but étant de pouvoir généraliser le traitement à n'importe quelle image, la reconstruction nécessite d'être agnostique. C'est pour ça que nous n'avons que les chunks que nous voulons traiter en entrée et non pas l'image entière. Les chunks sont des parties de l'image labellisée donc nous avons déjà accès aux labels dans notre fonction de stitching. Je vais vous présenter le profil de cette fonction de stitching en me basant sur le schéma de la figure ci-dessous.



Figure 35: Schéma fonction de stitching

Passons en revue chacun des paramètres ensemble:

- chunk_1 est un np.array contenant l'image que chunk_1 représente avec sa bbox plus son overlap
- chunk_2 est un np.array représentant l'équivalent de chunk_1 mais pour le chunk 2
- coord_chunk_1 est une instance de la classe Chunk qui contient les coordonnées de début et de fin du chunk en x et y
- idem pour coord_chunk_2
- overlap est un int représentant le nombre de tile de recouvrement voulu entre chaque chunk cette donnée nous sera utile plus tard pour retrouver la bbox des chunks notamment
- chunk_size est un int représentant la taille du chunk qui doit servir seulement en cas de debug car pour le fonctionnement classique nous avons besoin d'être agnostique pour pouvoir généraliser.
- tile_size est un int représentant la taille d'une tile ce qui permet grâce à overlap de calculer l'overlap en pixels

Algorithm 1 Décision de placement du label

```

unique_labels1 ← labels_chunk1
for label in unique_labels1 do
    if label = 0 then
        continue
    end if
    polygon, centroid ← process_mask(chunk_1, label)
    if centroid is None then
        continue
    end if
    x, y ← centroid
    if x < chunk_1_valid_x then
        polygons1 ← polygons1 ∪ {polygon}
        centroids1 ← centroids1 ∪ {centroid}
        valid_labels1 ← valid_labels1 ∪ {label}
    end if
end for

```

Voici ci-dessus [1] la manière dont j'ai choisi de gérer la prise de décision concernant le chunk 1. Pour chaque label (cellule) de chunk_1 nous appelons une fonction qui nous retourne son polygone (contour) et son centroïde (centre). Nous comparons le centroïde de ce label à la plus grande valeur possible de x dans chunk_1 sans être dans la partie overlap, nous nous cantonnons donc à la bbox de chunk_1. Si le centroïde du label fait partie de la bbox du chunk_1 alors nous décidons d'ajouter ce label, ce polygone et ce centroïde à la liste des labels, polygones et centroïdes valides pour chunk_1. Nous appliquerons exactement la même logique pour chunk_2 sauf que la condition cette fois sera $x > overlap * tile_size$. Car le calcul du centroïde est effectué dans le repère local de chunk_2 et non plus le repère de l'image global donc la première coordonnées de x est 0 sauf que ce 0 correspond à la zone d'overlap gauche de chunk_2. C'est pourquoi la bbox de chunk_2 commence à partir de $0 + overlap$, overlap étant exprimé en pixels. Nous remarquerons que le fait que nous ne raisonnions qu'en x ici est dû au fait que nous n'avons que notre image globale qui

est coupée en deux verticalement. De plus pour chunk_1 nous n'avons à gérer que le bord droit et pour chunk_2 seulement le gauche car l'autre bord est la limite de l'image totale donc il n'y a pas de sélection à faire. Dans la même situation avec trois chunks découpés verticalement et donc alignés horizontalement, le traitement du chunk du milieu (2) aurait nécessité la prise en compte de la bordure à sa gauche (chunk_1) mais également la bordure avec le chunk à sa droite (chunk_3).

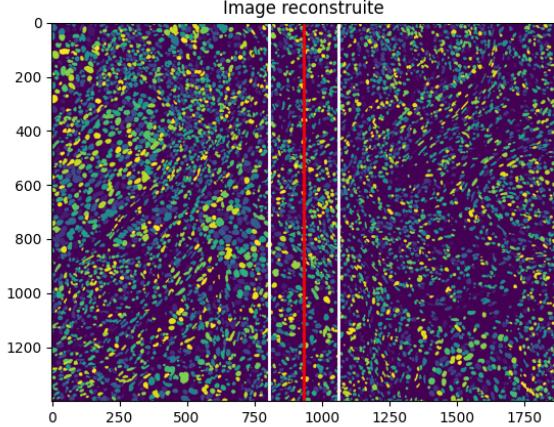


Figure 36: Image globale avec ligne de découpage

Ici nous avons une bonne visualisation du découpage en chunks. La ligne rouge au milieu représente la bordure entre les deux chunks donc délimite leurs bbox respectives. Chunk_1 va de 0 à 934 (ligne rouge), et le chunk_2 de 934 à 1868. Le chunk_1 avec overlap va lui de 0 à 1062(deuxième ligne blanche). Tandis que le chunk_2 avec overlap commence à 806 (première ligne blanche) et se termine à 1868, la fin de l'image. L'ajout de cette overlap est vital pour assurer la qualité de la segmentation lors des étapes précédentes mais il est également vital de gérer les complications que ça implique pour la reconstruction.

Une fois le traitement des cellules en bordure appliqué comme vu dans l'algorithme plus haut[1]. J'ai pu faire mes premiers essais de reconstruction en prenant pour repère une cellule en bordure de chunk afin de vérifier son traitement.

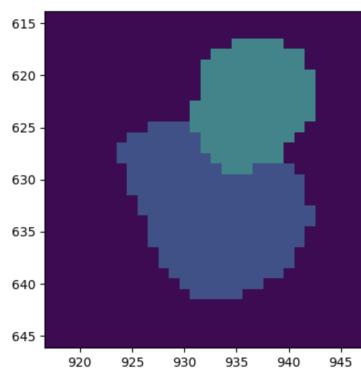


Figure 37: Cellule repère

La cellule ci-dessus est l'objectif à atteindre.

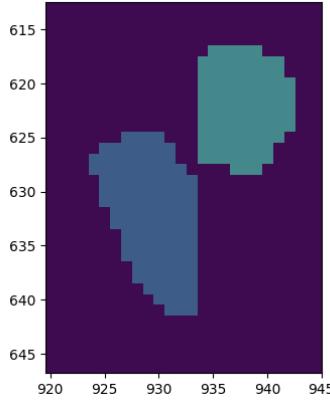


Figure 38: Premier essai de reconstruction

Ci-dessus, nous pouvons voir le résultat du premier test de reconstruction. Le problème saute aux yeux, la cellule est coupée en deux en 934 qui est, pour rappel, la limite entre les deux chunks. Nous sommes en plein dans le phénomène que nous souhaitions éviter. Cette ligne coupe la cellule en deux et ignore la partie de la cellule qui est dans l'autre chunk. Il s'ensuit donc bien évidemment une phase de test de l'algorithme [1] pour s'assurer de la source du problème : problème du choix de gestion des cellules en bordure ou problème de reconstruction ?

Pour tester l'algorithme j'ai donc affiché respectivement le chunk_1 seul et le chunk_2 seul et les polygones en bordure étaient bien entiers donc c'était vraiment l'étape de fusion des deux chunks qui bloquait.

La reconstruction que j'appliquais à ce moment-là était une concaténation de chunk_1 et chunk_2 ainsi qu'une mise à zéro ensuite des labels qui n'étaient pas dans la liste des labels_valides de chaque chunk.

J'ai donc poursuivi dans cette voie pour un second test mais en essayant d'ajouter l'overlap également dans la reconstruction, ce qui a donné le résultat que vous pouvez trouver dans la figure ci-dessous.

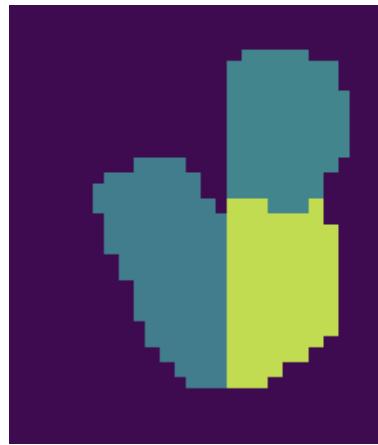


Figure 39: Deuxième essai avec overlap

Nous noterons ici une légère amélioration avec la deuxième partie inférieure de la cellule qui apparaît. Néanmoins ce résultat m'a permis de remarquer quelque chose qui m'avait échappé, ce n'est pas

très visible sur la figure mais il y a une nuance de couleur entre la partie supérieure droite de la cellule et la partie inférieure gauche. Ce qui veut dire que cette cellule comporte trois labels alors qu'elle n'est censée en comporter que deux. Cela m'a permis de me rendre compte que je ne pouvais pas reconstruire l'image finale en me servant de mes chunks directement car quand bien même j'aurais la donnée me permettant de savoir que cette cellule est sur le bord du chunk_1 doit être traité par chunk_1, je ne pourrais pas l'afficher dans chunk_1 vu qu'il dépasserait. Il faut donc se détacher des chunks pour la reconstruction et utiliser les labels et les polygones. J'ai donc élaboré une nouvelle approche consistant à parcourir la liste de polygones et de labels valides de chaque chunk et à dessiner tous les polygones avec les labels correspondants dans une image vide de la bonne taille. Tout en n'oubliant pas d'effectuer un changement de repère pour les polygones de chunk_2 en les basculant du repère local vers le repère de l'image entière reconstruite. Cette approche nous permet de nous affranchir de la limite spatiale du chunk dans la reconstruction tout en conservant le traitement nous permettant de nous assurer que deux chunks ne vont pas traiter la même cellule.

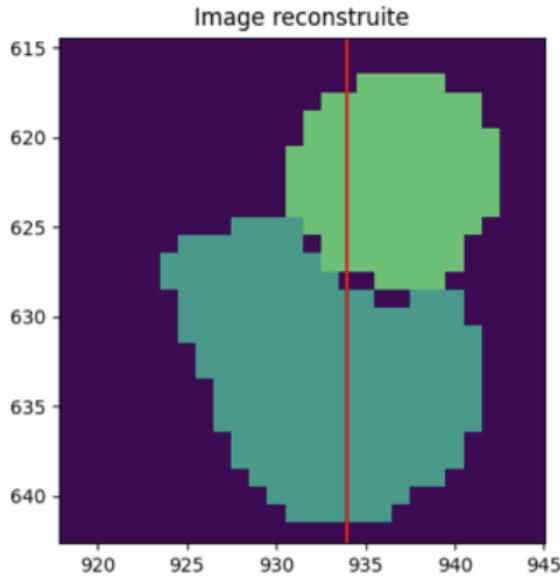


Figure 40: Résultat obtenu avec la nouvelle approche

Nous pouvons voir ici que nous obtenons un résultat totalement conforme à celui qui nous servait de repère .[37]

La ligne rouge représente la frontière théorique entre les deux chunks vu qu'elle est pile au milieu de l'image comme nous pouvions le voir sur cette image. [36]

4.4.3 Étape trois : généralisation du traitement 1D

Comme nous pouvions le voir dans l'algorithme précédent 1 le traitement était pensé pour gérer seulement deux chunks et n'était pas généralisé. De plus c'était du traitement chunk par chunk au niveau des labels sans automatisation donc ce n'était absolument pas viable pour la suite. Voilà pourquoi une nouvelle fonction a été pensée en traitant les chunks via une liste, c'est tout l'enjeu de cette étape trois : être capable de généraliser le traitement des chunks du moment que nous restons en une dimension.

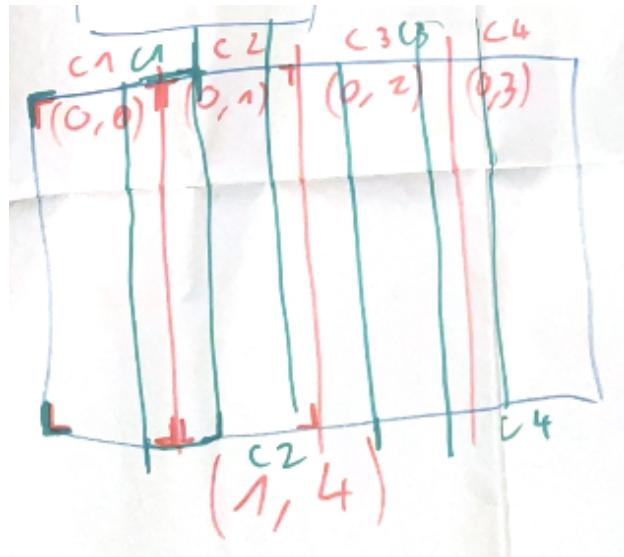


Figure 41: Objectif de l'étape 3

Cette situation implique toujours le traitement de l'overlap entre les chunks mais de manière plus subtile que dans l'étape d'avant. Précédemment si la coordonnée x du centroïde était dans la zone sans overlap du chunk_1 alors le traitement lui était dévolué. Sinon s'il était après la zone d'overlap, c'était le chunk_2 qui était chargé du traitement. Cette méthode n'est plus viable dans notre situation car il faut aussi gérer les chunks qui ne sont pas aux extrémités. Comme nous pouvons le voir dans l'algo simplifié ci-dessous [2], la manière de gérer le tout a changée. Premier changement majeur: les chunks sont maintenant sous forme de liste[chunk_infos][chunk_np_array] et la dimension donnée sous forme de grille. La grille nous permet de récupérer le nombre de colonnes et de lignes, dans cette étape le nombre de lignes sera toujours à 1. C'est le nombre de colonnes qui déterminera le nombre de chunks. Le principe de l'algorithme est le suivant : les centroïdes et les polygones sont calculés avec un offset leurs permettant d'être dans le bon repère qui sera aussi utilisé pour la reconstruction dans l'image finale. Ce offset correspond aux coordonnées de départ du chunk traité. Ensuite il y a trois cas possibles. Soit le chunk est à gauche donc déjà dans le bon repère et nous fixons l'offset à 0. Le polygone est à traiter si son centre est dans la bounding box du chunk, la limite en x de la bounding box est récupérée grâce à la fonction get_valid_xmax qui retourne le x le plus grand toléré par la bbox du chunk(zone sans l'overlap). Le même principe est appliqué pour le chunk le plus à droite sauf que nous ne nous concentrons pas sur le xmax mais le xmin qui représente la partie gauche du chunk et donc la partie bénéficiant de l'overlap que nous voulons ici retirer de la bounding box. De même pour la dernière condition qui gère tous les chunks centraux qui ne sont donc pas aux extrémités. Ces chunks là ne gardent que les polygones dont le centre est dans leurs bboxes. Cette gestion nous assure que chaque polygone ne sera traité qu'une fois.

Algorithm 2 Placement généralisé des labels dans chaque chunk

```

label_max ← 0
for chunk = 0 to row - 1 do
    offset ← 0
    x_offset ← chunk_list_output[chunk][0].x_start
    for all label ∈ labels_uniques do
        (polygon, centroid) ← process_mask(chunk_relabel, label, x_offset, y_offset=0)
        if centroid is None then
            continue
        end if
        (x, y) ← centroid
        if chunk_list_output[chunk][0].position == 0 (N.B chunk est à gauche) then
            offset ← 0
            if x ≤ chunk_list_output[chunk][0].get_valid_xmax(overlap × tile_size) then
                Ajouter polygon, centroid, label à leurs listes respectives
            end if
        else if chunk_list_output[chunk][0].position == row - 1(N.B chunk est à droite) then
            offset ← chunk_list_output[chunk][0].x_start
            if x ≥ chunk_list_output[chunk][0].get_valid_xmin(overlap × tile_size) then
                Ajouter polygon, centroid, label
            end if
        else
            offset ← chunk_list_output[chunk][0].x_start
            if get_valid_xmin(overlap × tile_size) ≤ x ≤ get_valid_xmax(overlap × tile_size)
            then
                Ajouter polygon, centroid, label
            end if
        end if
    end for
    label_max ← max(chunk_relabel)
    Dessiner les polygones valides dans reconstructed avec l'offset
end for

```

Nous pouvons observer la manière dont le découpage est effectué lorsque le nombre de chunk demandé est de six. Le début du chunk est représenté en vert et la fin en rouge.

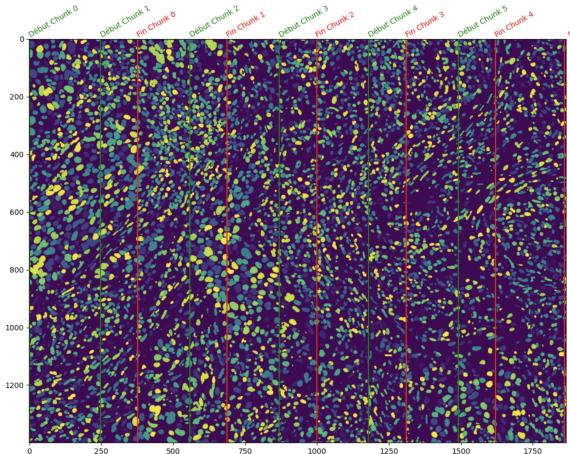


Figure 42: Découpage avec six chunks

Le nombre de cellules identifiées était égal à celui de l'image complète dans le fichier test ce qui permet de valider ce fonctionnement.

Toutefois il était important de vérifier la robustesse du traitement et comment il allait se comporter si nous lui enlevions un des chunks de la liste par exemple le chunk_0. Le comportement attendu est une reconstruction sans le chunk_0 mais qui ne change rien aux autres chunks.

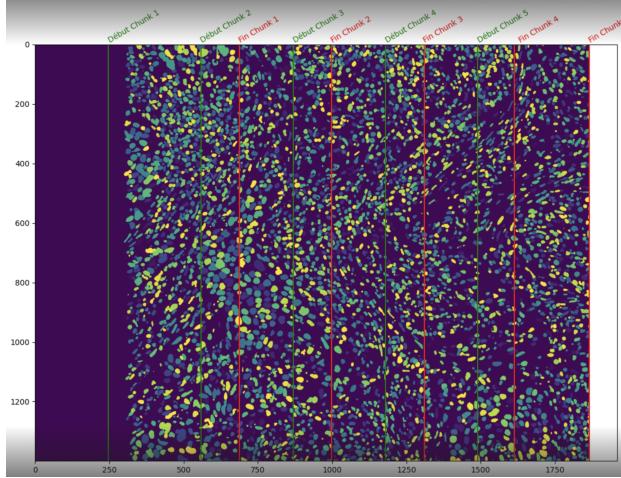


Figure 43: Découpage avec 6 chunks mais retrait du chunk_0

Nous observons bien le comportement attendu ce qui montre la robustesse du code.

4.4.4 Étape 4: Généralisation au traitement 2D

Avant toute chose au début de cette étape je me suis rendu compte que les conditions de mon algo précédent n'étaient pas exactes. Avec des conditions telles qu'elles les cellules étant pile en bordure de chunk allaient être comptées deux fois. Il a fallu réévaluer les conditions pour s'assurer que les frontières entre chunk ne seraient traitées qu'une fois.[4] Pour faire cela nous implémentons la logique suivante. Si le chunk est celui le plus à gauche, il ne gère pas sa bordure droite. Si le chunk est le plus à droite, il gère sa bordure gauche. Et si le chunk est au milieu il gère seulement sa bordure gauche.

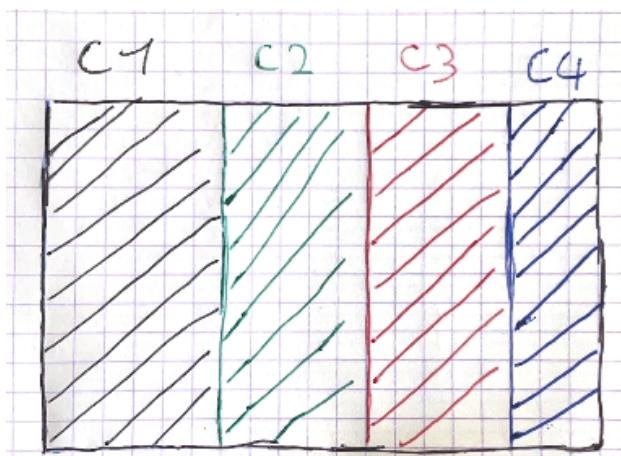


Figure 44: Gestion de la bordure pour quatre chunks

Voici un exemple avec C1, C2, C3, C4 respectivement chunk_1, chunk_2, chunk_3 et chunk_4. Ici nous avons C1 qui ne gère pas sa bordure droite car C2 la gère. C2 ne gère pas sa bordure droite car C3 la gère. De même pour C3 qui ne gère pas sa bordure droite car C4 la gère. Grâce à cette logique nous assurons le traitement unique de chaque bordure.

Place maintenant au traitement 2D, ce traitement en 2D implique une grille de chunk avec colonnes et lignes. Si nous avons une grille (2,4) cela signifie que nous aurons 8 chunks (2 colonnes de 4 chunks). Cette étape est fusionnée avec une étape d'optimisation du code car le traitement en y des polygones implique d'au minimum doubler le nombre de conditions pour gérer les polygones.(Il y a y fois le nombre de conditions en x en plus). C'est pour cela que les structures de données ont été repensées afin que des classes comme Chunk2D ou ChunkShape aient des attributs tels que context qui est une bounding box des coordonnées du chunk avec overlap ou bien core qui représente cette fois la zone effective du chunk sans l'overlap. Une fonction *is_inside* qui prend en paramètres des coordonnées x et y et vérifie si elles appartiennent au core. Cela nous donne l'algorithme simplifié ci-dessous.

Algorithm 3 Placement généralisé des labels dans chaque chunk 2D

```

label_max ← 0
for all chunk ∈ chunks do
    if chunk.array is None then
        continue
    end if
    labels_1 ← unique(chunk.array)
    chunk_relabel ← where(chunk.array ≠ 0, chunk.array + label_max, 0)
    labels_2 ← unique(chunk_relabel)
    for all label ∈ labels_2 do
        if label = 0 then
            continue
        end if
        (polygon, centroid) ← process_mask(chunk_relabel, label, x_offset = chunk.shape.context[0], y_offset = chunk.shape.context[1])
        if centroid is None then
            continue
        end if
        (x, y) ← centroid
        if chunk.shape.is_inside(x, y) then
            Initialiser chunk_data
            Ajouter polygon, centroid, label à chunk_data
            Dessiner les polygones dans reconstructed avec x_offset = chunk.shape.context[0] et
            y_offset = chunk.shape.context[1]
        end if
    end for
    label_max ← label_max + max(chunk.array)
end for
Appliquer randomize_labels à reconstructed
Afficher reconstructed avec imshow sibesoin

```

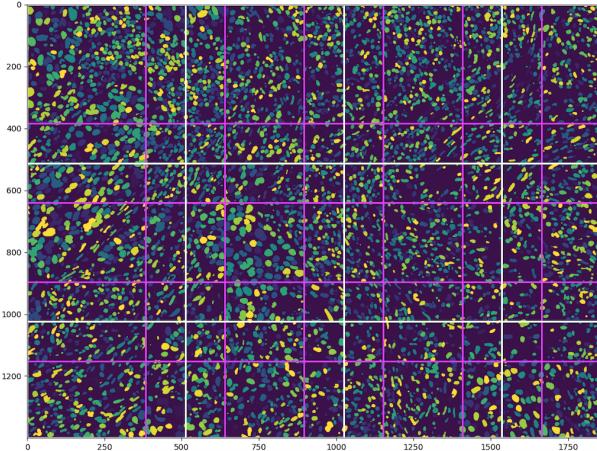


Figure 45: Résultat du traitement 2D

Voici un exemple de résultat avec une grille de (3,4) chunks. La grille est un paramètre calculé automatiquement à partir de la taille du chunk que nous fixons. Ici nous avons choisi d'avoir des chunks de 512 par 512 pixels. Dans cette situation, s'il n'est pas possible de découper l'image en chunk de la taille que nous demandons alors nous créons le maximum de chunk de la bonne taille et si la partie de l'image restante est plus grande que la moitié de la taille d'un chunk alors nous créons un chunk exprès pour cette partie et sinon nous l'ajoutons simplement au dernier chunk en l'agrandissant. Cela nous permet de toujours respecter les dimensions de l'image initiale et de ne pas perdre de données. Notre nombre de cellules détectées reste correct, les cellules sont bien reconstruites même lorsqu'elles sont en bordure de chunk.

Nous avons ensuite effectué un nouveau test de robustesse en utilisant la fonction *expand_labels* de la librairie skimage.[16] Cette fonction applique une dilatation contextuelle à chaque label de l'image. Cela signifie que le label va s'étendre dans l'image selon la valeur de son noyau qui est une valeur en pixels que nous fixons en paramètre. Ce label s'étendra donc de ce nombre de pixels si cela n'implique pas d'empêter sur un autre label auquel cas il ne s'étendra simplement pas. Cet ajout nous permet de voir si notre code et notre fonctionnement de reconstruction restent robustes face à ce changement.

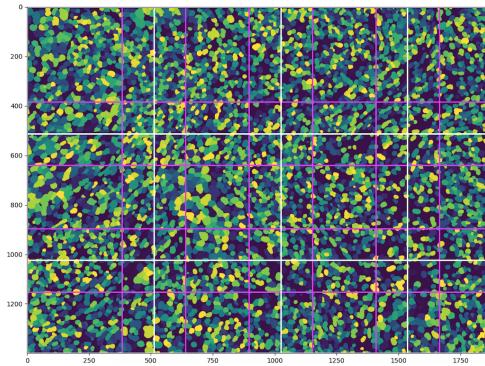


Figure 46: Résultat du traitement 2D avec expansion

Vous pouvez voir ci-dessus [46] la figure avec les labels étendus et une fois de plus, la reconstruction ne rogne aucun label.

5 Phase 3 et suite du stage

Le stage se finit le 15 septembre donc un peu moins d'un mois après la date de rendu de ce rapport. Je vais donc ici brièvement expliquer le sujet que je vais développer dans le mois à venir.

L'objectif est de fusionner les deux premières phases du stage en essayant d'appliquer le stitching et le tiling dans Kartezio pour effectuer la segmentation des pigments de Neuromélanine dans les IRMs.

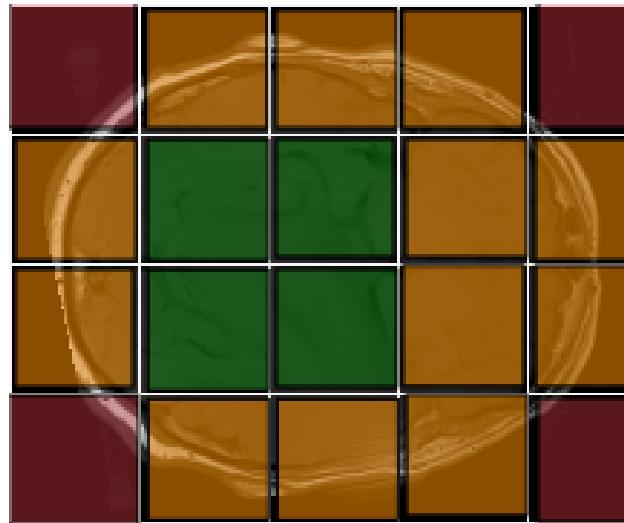


Figure 47: Zone d'intérêt sur une slice axial d'IRM

L'idée est de mettre en place des zones d'intérêts et de pondérer l'entraînement avec le poids de chaque zone. En rouge les zones sans data, en orange les zones qui en contiennent sans contenir pour autant les pigments que nous cherchons à segmenter. Et en vert les tiles contenant potentiellement les pigments à segmenter. L'idée serait donc d'intégrer la notion d'overlap entre ces tiles lors du traitement. Le fonctionnement est semblable à la sliding window mais dans la slice elle-même.

Conclusion

J'ai beaucoup appris lors de stage que ce soit techniquement ou personnellement. J'ai acquis de nouvelles connaissances liés à la programmation génétique (GP et CGP). J'ai pu les appliquer un problème concret. J'ai également eu l'occasion de développer et d'approfondir certaines compétences que j'avais acquises pendant mon cursus scolaire notamment avec l'utilisation de modèle d'IA et de bibliothèques de traitements d'images. J'ai également pu solidifier mes compétences de programmation et de réflexion algorithmique notamment sur la deuxième phase du stage.

Ce stage m'a aussi permis d'explorer deux formes d'encadrement différents, un environnement de recherche avec plus de liberté, moins de contrainte et donc moins de repères. Et un autre plus entrepreneurial avec plus de cadre, un cahier des charges très précis. Ces deux expériences différentes m'ont permis d'en apprendre plus sur ma manière de travailler, de me remettre en question sur ce que je faisais de mal et de développer des compétences organisationnelles différentes.

Annexes

Algorithm 4 Correction de l'algorithme de placement

```
label_max ← 0
for chunk = 0 to row - 1 do
    offset ← 0
    x_offset ← chunk_list_output[chunk][0].x_start
    for all label ∈ labels_uniques do
        (polygon, centroid) ← process_mask(chunk_relabel, label, x_offset, y_offset=0)
        if centroid is None then
            continue
        end if
        (x, y) ← centroid
        if chunk_list_output[chunk][0].position == 0 (N.B chunk est à gauche) then
            offset ← 0
            if x < chunk_list_output[chunk][0].get_valid_xmax(overlap × tile_size) then
                Ajouter polygon, centroid, label à leurs listes respectives
            end if
        else if chunk_list_output[chunk][0].position == row - 1 (N.B chunk est à droite) then
            offset ← chunk_list_output[chunk][0].x_start
            if x >= chunk_list_output[chunk][0].get_valid_xmin(overlap × tile_size) then
                Ajouter polygon, centroid, label
            end if
        else
            offset ← chunk_list_output[chunk][0].x_start
            if get_valid_xmin(overlap × tile_size) ≤ x < get_valid_xmax(overlap × tile_size)
            then
                Ajouter polygon, centroid, label
            end if
        end if
    end for
    label_max ← max(chunk_relabel)
    Dessiner les polygones valides dans reconstructed avec l'offset
end for
```

References

- [1] W. Banzhaf, “Genetic programming,” in *Encyclopedia of Information Systems*, H. Bidgoli, Ed., vol. 2, [Online]. Available: <https://www.sciencedirect.com/science/article/abs/pii/B008043076700557X>, San Diego, CA, USA: Academic Press, 2003, pp. 137–146.
- [2] D. H. H., *Genetic algorithms and evolutionary computation faq*, TalkOrigins Archive, [Online]. Available: <https://www.talkorigins.org/faqs/genalg/genalg.html>, Mar. 2001.
- [3] J. F. Miller, “Cartesian genetic programming: Its status and future,” *Genetic Programming and Evolvable Machines*, vol. 21, no. 1–2, pp. 181–215, Jun. 2020, [Online]. Available: <https://link.springer.com/article/10.1007/s10710-019-09360-6>.
- [4] L. Trujillo, L. Muñoz, E. Galván-López, and S. Silva, “Neat genetic programming: Controlling bloat naturally,” *Information Sciences*, vol. 333, pp. 21–43, Jan. 2016, [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0020025515008038>.
- [5] J. Jordan, M. Schmidt, W. Senn, and M. A. Petrovici, “Evolving interpretable plasticity for spiking networks,” *eLife*, vol. 10, e66273, Oct. 2021, [Online]. Available: <https://www.researchgate.net/publication/355714324>.
- [6] K. Cortacero, B. McKenzie, S. Müller, *et al.*, “Evolutionary design of explainable algorithms for biomedical image segmentation,” *Nature Communications*, vol. 14, no. 7112, 2023. DOI: 10.1038/s41467-023-42664-x. [Online]. Available: <https://www.nature.com/articles/s41467-023-42664-x>.
- [7] C. De La Torre, Y. Lavinas, K. Cortacero, H. Luga, D. G. Wilson, and S. Cussat-Blanc, “Multimodal adaptive graph evolution for program synthesis,” in *Parallel Problem Solving from Nature – PPSN XVIII: 18th International Conference, PPSN 2024, Hagenberg, Austria, September 14–18, 2024, Proceedings, Part I*, Hagenberg, Austria: Springer-Verlag, 2024, pp. 306–321, ISBN: 978-3-031-70054-5. DOI: 10.1007/978-3-031-70055-2_19. [Online]. Available: https://doi.org/10.1007/978-3-031-70055-2_19.
- [8] C. D. L. Torre, G. Nadizar, Y. Lavinas, H. Luga, D. Wilson, and S. Cussat-Blanc, “Evolution of inherently interpretable visual control policies,” in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO ’25)*, ACM, 2025, pp. 358–367. DOI: 10.1145/3712256.3726332. [Online]. Available: <https://doi.org/10.1145/3712256.3726332>.
- [9] V. Mnih, K. Kavukcuoglu, D. Silver, *et al.*, “Human-level control through deep reinforcement learning,” *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.
- [10] W. Dabney, G. Ostrovski, D. Silver, and R. Munos, “Implicit quantile networks for distributional reinforcement learning,” in *Proceedings of the 35th International Conference on Machine Learning (ICML)*, PMLR, 2018, pp. 1096–1105.
- [11] A. P. Badia, B. Piot, S. Kapturowski, *et al.*, “Agent57: Outperforming the Atari Human Benchmark,” in *Proceedings of the 37th International Conference on Machine Learning (ICML)*, PMLR, 2020, pp. 507–517.
- [12] Site web de l’équipe REVA, [Online]. Available: <https://www.irit.fr/departement/calcul-intensif-simulation-optimisation/rev/>.
- [13] E. Jenkinson and O. Arandjelović, “Whole slide image understanding in pathology: What is the salient scale of analysis?” *BioMedInformatics*, vol. 4, no. 1, pp. 489–518, 2024, ISSN: 2673-7426. DOI: 10.3390/biomedinformatics4010028. [Online]. Available: <https://www.mdpi.com/2673-7426/4/1/28>.
- [14] U. Schmidt, M. Weigert, C. Broaddus, and G. Myers, *Stardist: Object detection with star-convex shapes*, <https://github.com/stardist/stardist>, Accessed: 2025-07-31, 2018.

- [15] T. Goldsborough *et al.*, “Instanseg: An embedding-based instance segmentation algorithm optimized for accurate, efficient and portable cell segmentation,” *arXiv preprint arXiv:2408.15954*, 2024. DOI: 10.48550/arXiv.2408.15954. [Online]. Available: <https://doi.org/10.48550/arXiv.2408.15954>.
- [16] S. van der Walt, J. L. Schönberger, J. Nunez-Iglesias, *et al.*, “Scikit-image: Image processing in python,” *PeerJ*, vol. 2, e453, 2014, [Online]. Available: <https://peerj.com/articles/453/>. DOI: 10.7717/peerj.453.