Rapport De Design
Projet BRAIN

FILOCHE Léo
MORLOT-PINTA Louis
DE ZORDO Benjamin
LEGRAND Quentin

Ce rapport a pour objectif de définir les choix de conception de chaque sprint ainsi que de décrire la prise en compte des contraintes identifiées en analyse.

I. Présentation générale

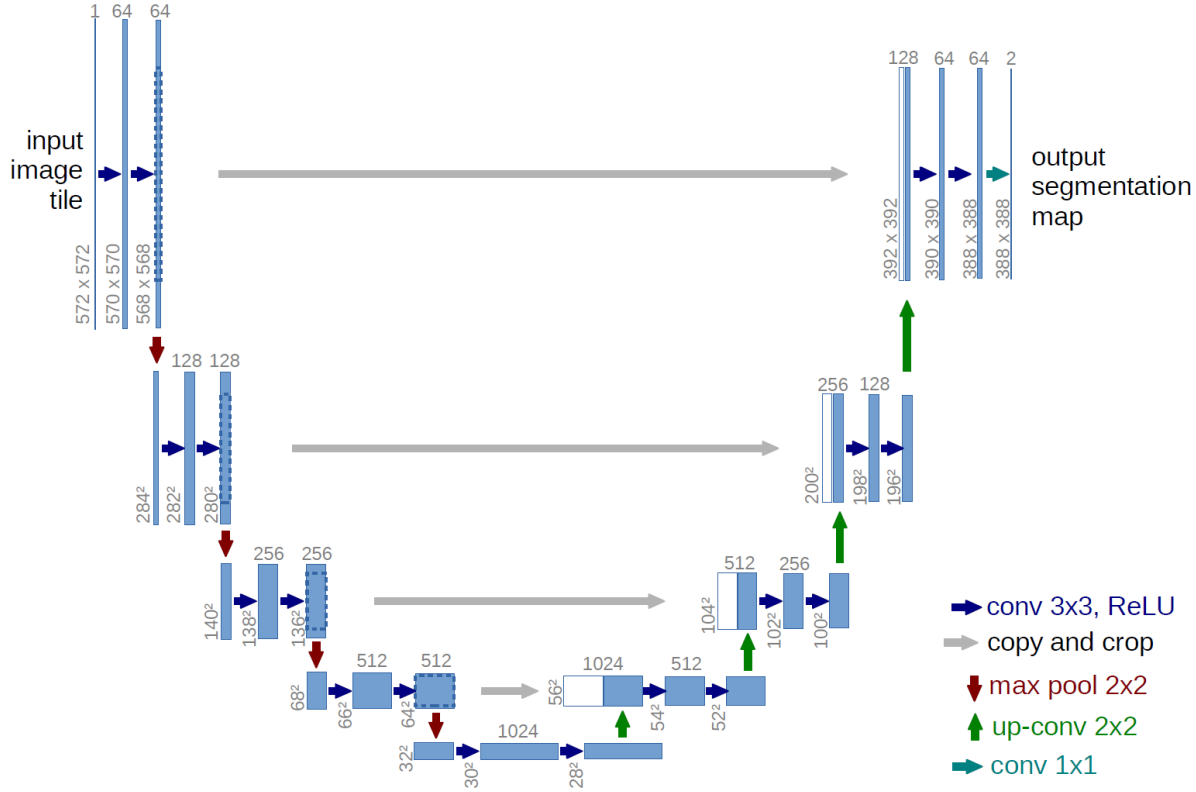
D'un point de vue IA nous avons utilisé le langage le plus utilisé dans ce domaine, Python, car il est approprié pour les calculs numériques que l'on retrouve en IA, et que la communauté d'IA le préfère à d'autres. Ainsi, nous avons utilisé un certain nombre de ces librairies. On peut citer NiBabel qui est un librairie pour gérer les fichiers NIFTI qui est l'extension des fichiers médicaux ou TensorFlow qui est une librairie de machine learning qui présente les fonctions classiques utilisées de métrique, de calcul etc. Pour cette partie, plusieurs contraintes fortes s'appliquent. L'IA doit être fiable, précise et efficace. Les prédictions produites doivent se dérouler dans un temps raisonnable pour l'utilisateur, et elles doivent être significatives (elles doivent assister réellement le docteur).

Pour ce qui est de la partie web, nous avons porté rapidement nos choix sur FastAPI du point de vue backend, car c'est un framework d'API REST basé sur Swagger, facile d'utilisation. L'un de nos membres le connaissait déjà, et il est écrit en Python, même langage que pour les scripts d'IA, permettant un meilleur échange de données entre les parties. Nous avons choisi SQLite pour créer une base de données pour les utilisateurs. Pour le frontend, nous étions sous la contrainte de pouvoir intégrer facilement un visualiseur. Nous avons d'ailleurs commencé par cette étape : faire un état de l'art des visualiseurs web de fichiers d'IRM (format .nii). Après ces recherches, nous avons choisi d'utiliser la librairie Nifti-Reader Js. Après avoir tenté de l'intégrer dans un projet Angular sans succès, nous nous sommes rabattus sur Astro qui permet d'exécuter du javascript directement, tout en garantissant une bonne structure de projet. Docker a également été utilisé pour déployer le projet sur un serveur.

II. Présentation de la solution IA

a. Présentation générale

Pour le choix du framework nous avons décidé d'implémenter un réseau de neurones entièrement convolutif, Unet, qui est un réseau de neurones développé pour la segmentation d'images biomédicales. Il est capable de localiser et de distinguer les frontières des éléments composant une certaine image en faisant le focus sur une zone de l'image et de procéder à la classification sur chaque pixel.



L'architecture de U-NET est composée de deux chemins. Le premier est le chemin de contraction, aussi appelé encodeur. Il est utilisé pour capturer le contexte d'une image. Le second chemin est celui de l'expansion symétrique, aussi appelé décodeur. Il permet la localisation précise grâce à la convolution transposée.

Il s'agit en fait d'un assemblage de couches de convolution et de couches de "max pooling". Pendant la contraction les informations spatiales sont réduites tandis que les informations sur les caractéristiques sont augmentées.

Ainsi chaque Pixel est classifié et on est capable de séparer les différents éléments de l'image, ici les différentes zones de la tumeur.

b. Structure du modèle

L'architecture de framework choisi est le modèle UNet qui est une architecture de réseau neuronal convolutif largement utilisée dans les tâches de segmentation d'images, telles que l'analyse d'images médicales. Le modèle UNet a été implémenté en utilisant la bibliothèque Keras..

Le réseau comprend un chemin d'encodage et un chemin de décodage. Le chemin d'encodage du réseau consiste en une suite de couches de convolution qui réduisent la dimension spatiale de la carte d'entrée tout en augmentant le nombre de canaux de caractéristiques. Les couches de convolution sont généralement suivies d'une couche de pooling qui diminue la résolution spatiale de la carte de caractéristiques et réduit la complexité du modèle en éliminant les détails fins. Cette étape est répétée plusieurs fois pour obtenir une carte de caractéristiques avec une résolution spatiale très réduite, mais avec un grand nombre de canaux.

Le chemin de décodage, lui, est composé de couches de déconvolutions qui augmentent la résolution spatiale de la carte de caractéristiques. Les couches de déconvolution sont généralement suivies d'une couche de concaténation qui combine la carte de caractéristiques décodée avec la carte de caractéristiques correspondante dans le chemin d'encodage. Cette opération permet de restaurer les détails fins de l'image d'origine tout en maintenant la précision de la segmentation.

Le réseau prend en entrée des images de taille IMG_SIZE x IMG_SIZE x 2 (deux canaux) et retourne une carte de segmentation de taille IMG_SIZE x IMG_SIZE x 4 (quatre classes). Les deux canaux d'entrées correspondent à deux contrastes d'IRM et les quatres classes correspondent aux stades de la tumeur (cœur, l'œdème, la partie rehaussée et l'absence de tumeur). La fonction de perte utilisée pour entraîner le réseau est la perte de cross-entropy catégorielle et l'optimiseur utilisé est l'optimiseur Adam avec un taux d'apprentissage de 0,001. Le réseau est également évalué en utilisant plusieurs métriques telles que la précision, la sensibilité, la spécificité et le coefficient de Dice pour chaque classe de segmentation.

c. Data loader et generator

Le script de data loader est destiné à charger les données d'un ensemble de données de patients. Il utilise la bibliothèque os pour accéder aux fichiers, la bibliothèque sklearn pour diviser les données en ensembles d'entraînement, de validation et de test, ainsi qu'un module de variables personnalisé pour définir les différents chemins de données.

Une fonction pathListIntoIds transforme les noms de dossiers en identifiants et la fonction load_data utilise ces identifiants pour séparer les données en ensembles d'entraînement, de validation et de test. L'ensemble d'entraînement est divisé en 85% pour l'entraînement et 15% pour les tests.

Le script de Data generator implémente la classe DataGenerator, une sous-classe de keras.utils.Sequence, elle est utilisée pour générer des lots de données pour l'entraînement d'un modèle. Elle prend en entrée une liste d'ID et les utilise pour générer des paires d'entrée/sortie à utiliser lors de l'entraînement du modèle.

La classe a quatre méthodes :

- __init__: initialisation de la classe avec les paramètres requis tels que la taille de l'image, la taille du lot, le nombre de canaux et la liste d'ID.
- __len__: retourne le nombre de lots par epoch en fonction de la taille du lot et de la longueur de la liste d'ID.
- __getitem__: récupère un lot à partir des indexs du lot et de la liste d'ID et génère les paires d'entrée/sortie à partir de __data_generation.
- __data_generation: génère les paires d'entrée/sortie à partir de la liste d'ID récupérée dans __getitem__.

L'utilité de cette classe est de permettre de charger les données à la volée au lieu de les charger toutes en mémoire. Elle charge les données en petits lots, ce qui permet d'économiser de la mémoire et de permettre l'entraînement des modèles sur des données volumineuses.

d. Training du modèle

Pour des raisons de contraintes de temps il n'a pas été possible d'entraîner notre propre modèle, le script de training est donc responsable de charger un modèle UNet pré-entraîné. Ce modèle présente de très bonnes performances et est stocké dans un fichier h5.

La première partie du code définit des fonctions de perte et des métriques pour l'apprentissage du modèle, notamment des fonctions pour calculer le "dice coefficient" (une mesure de similarité entre deux images binaires), la précision, la sensibilité et la spécificité.

La deuxième partie du code charge le modèle pré-entraîné en utilisant la fonction load_model de Keras et en définissant les fonctions de perte et les métriques personnalisées en amont pour évaluer le modèle.

Enfin, la dernière partie du code charge les données d'historique d'entraînement du modèle pré-entraîné à partir d'un fichier CSV et trace les graphiques d'évaluation des performances du modèle, tels que la précision, le rappel, la perte et le coefficient de Dice.

e. Prédiction

Le script de prédiction est responsable de fournir la prédiction sous forme d'un fichier NIFTI à partir des données d'IRM sauvegardées dans le dossier patients.

A partir de deux contrastes la fonction predictByPath() va fournir la segmentation prédite sous forme de trois 3d array de shape (VOLUME_SLICE,IMG_SIZE ,IMG_SIZE) qui inscrit pour chaque voxels la probabilité que le voxels appartienne à la classe cœur, l'œdème ou à la partie rehaussée de la tumeur.

Ensuite , ces trois arrays sont fusionnés en un seul suivant les règles suivante :

- Si la probabilité d'appartenir au cœur est inférieure à 0.4 alors le voxel est de classe oedème.
- Si la probabilité d'appartenir au oedème est inférieure à 0.4 alors le voxel est de classe partie rehaussée.
- Si la probabilité d'appartenir à la partie rehaussée est inférieure à 0.4 alors le voxel n'appartient pas à la tumeur.

Cet ajustement permet d'affiner le résultat et surtout génère un énorme gain pour les métriques de vrai faux, rappel et précision.

Finalement Il nous reste à appliquer deux autres post traitements. Le premier à pour but de tourner l'image en manipulant l'array de prédiction, l'objectif et de remettre en bonne forme la prédiction pour correspondre au fichiers d'entrées pour le visualiseur. Pour ceci on procède à un agrandissement et une rotation de la matrice, l'agrandissement génère de nouvelles valeurs qui sont remplacées par les valeurs de classes correspondantes.

Le second à pour objectif de supprimer les défauts de prédiction. Le principe est de vérifier s' il n'existe pas de rassemblement de voxels de la même classe trop petits. Si c'est le cas on les considère comme des défauts et les supprimons.

f. Evaluation

Ce script a pour objectif de calculer les métriques implémentées pour évaluer les performances du modèle. Il utilise donc la métrique de perte de DICE pour mesurer la similarité entre les prédictions du modèle et les annotations manuelles des tumeurs cérébrales. Il évalue également d'autres métriques de performance telles que l'exactitude, la précision, la sensibilité et la spécificité.

Le recadrage ou croping est une technique de pré-traitement pour éliminer les parties d'une image qui ne sont pas pertinentes pour la tâche en question. Cela permet de réduire la taille de l'image et de simplifier le traitement de l'information. Par exemple, pour les images médicales, on peut recadrer autour de la région d'intérêt pour éliminer le bruit de fond et se concentrer sur la zone de la tumeur.

La normalisation des données est un autre pré-traitement utilisé pour réduire les effets de la variation des données en ajustant la plage de valeurs des données d'entrée à une plage spécifique. Cela peut être utile pour éviter les problèmes liés à des données non normalisées telles que la saturation de la fonction d'activation, la convergence lente et la diminution des performances de classification.

Un dernier script de séparation permet de calculer la proportion de chaque classe présente dans une donnée patient et de créer un training set en s'assurant que les données qui le comprennent sont les données dont les proportions sont les plus équilibrées. Cela permet de s'assurer que le modèle s'entraîne sur les cas les plus généraux et non sur des cas très spécifiques, ce qui pourrait mener à un sur apprentissage et donc de moins bonnes performances.

g. Prétraitement

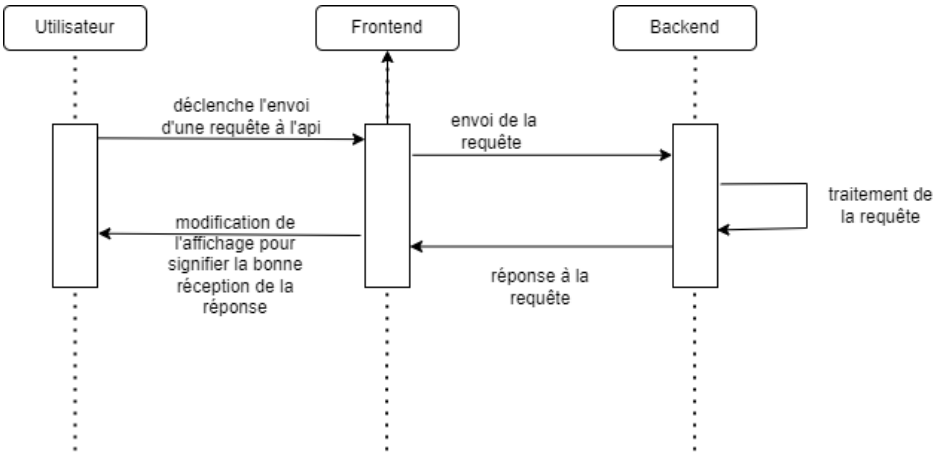
Dans l'objectif d'entraîner notre propre modèle nous avions commencé à implémenter certains pré traitement. Ces pré-traitements sont des étapes importantes en intelligence artificielle (IA) car ils permettent d'améliorer la qualité des données d'entrée et donc d'améliorer les performances des modèles.

III. Présentation de la solution WEB

En ce qui concerne l'application WEB, le développement a été dirigé par la volonté de joindre les différentes parties (front, api et ia) le plus rapidement possible, de sorte à ce qu'il n'y ait pas de conflits lors de leur jonction (éviter un aspect "big bang").

a. Sprint 1 : Mise en place de l'architecture principale

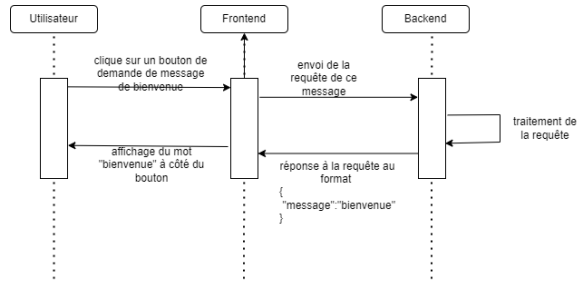
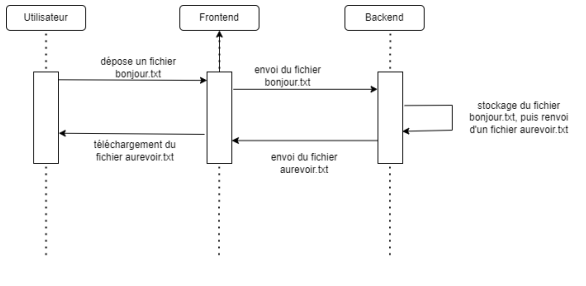
L'objectif de ce premier sprint a été de mettre en place une première structure, afin que chaque partie soit déjà présente et possède les premières fonctionnalités essentielles. Notre objectif était donc dans un premier temps d'avoir une page web sur laquelle il était possible d'envoyer une requête pour recevoir un message de la part de l'API, témoignant du succès de la connexion entre ces deux parties. Cela peut se traduire par la séquence suivante.



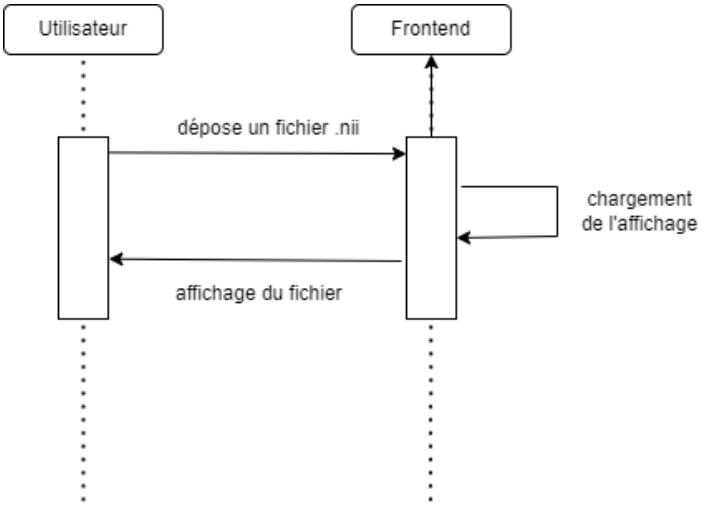
Séquence simple de l'action d'une requête* de la part de l'utilisateur

*: la requête est soit la demande d'une donnée textuelle, soit le dépôt d'un fichier

Dans un premier temps, l'idée a été d'effectuer des requêtes afin de recevoir des JSON "simples" (quelques clés, quelques valeurs), puis dans un second temps nous avons voulu que l'utilisateur puisse envoyer des fichiers à l'API, afin de les déposer sur le serveur, et d'en recevoir en retour.

	
exemple de séquence d'envoi d'une requête "textuelle"	exemple de séquence d'envoi de requête de fichier

Ce sprint a aussi été l'occasion d'intégrer une première version du visualiseur, afin de respecter les deux contraintes majeures du frontend, qui sont l'envoi/la réception de fichiers .nii, et leur visualisation.

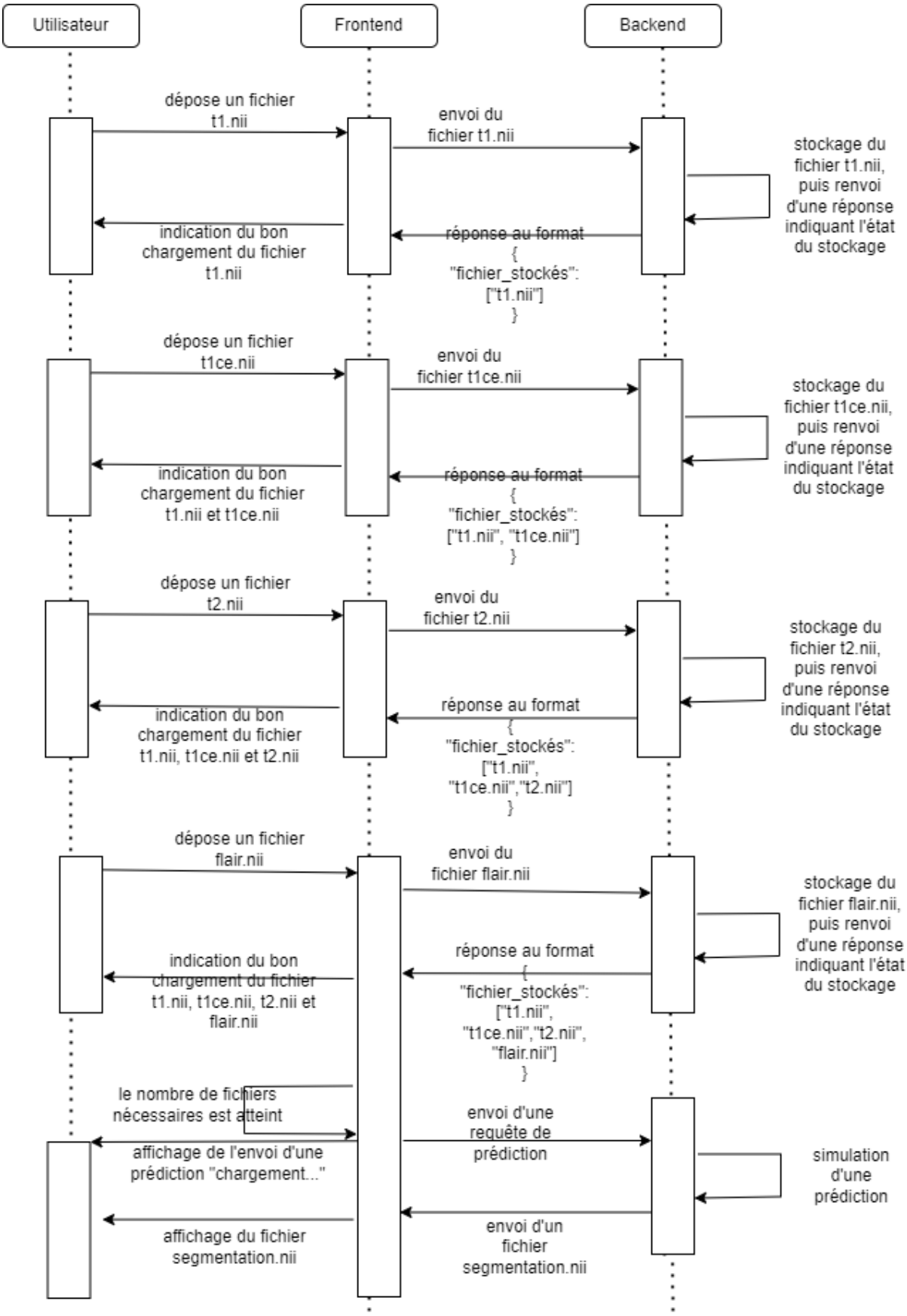


Séquence simple de la visualisation d'un fichier d'IRM sur l'interface

b. Sprint 2 : Développement de la séquence de prédiction

L'objectif de ce second sprint a été de mettre en place l'interface utilisateur pour simuler un processus de prédiction de bout en bout. L'idée est que l'utilisateur dépose 4 fichiers d'IRM (de type t1, t2,t2ce et flair), avant d'en recevoir une prédiction associée.

La séquence se traduit de la manière suivante.

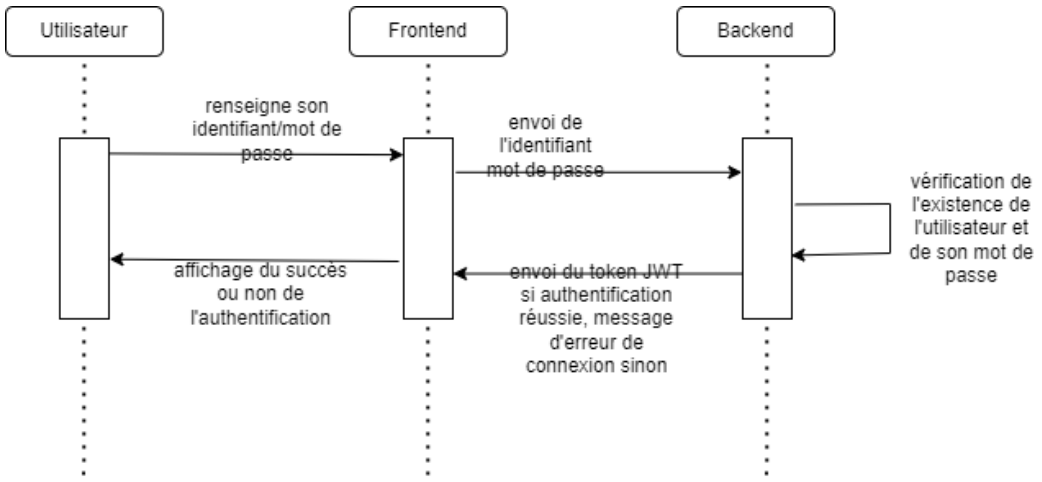


Séquence complète d’un processus de prédiction sur l’application

Bien que la séquence paraisse importante, il ne s'agit en fait que de la combinaison des deux "briques" d'actions construites au sprint précédent, qui sont l'échange de requêtes JSON et de fichiers, et l'affichage d'un fichier .nii. A ce stade, ce fichier n'est qu'un "stub", et ne correspond pas à la segmentation des fichiers donnés en entrée.

De plus, ce sprint a été l'occasion d'améliorer le visualiseur, en donnant la possibilité de changer de coupe (axiale, coronale, sagittale).

Enfin un fonctionnement processus très classique d'authentification a été mis en place.



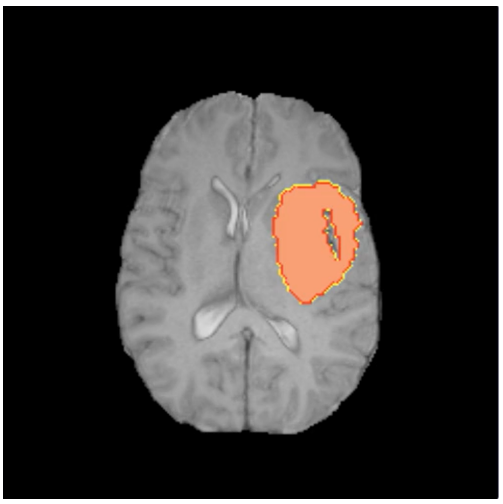
Séquence de l'authentification d'un utilisateur

Cette authentification permet au docteur d'effectuer la prédiction décrite ci-dessus. Sans celle-ci, cela lui est impossible. Ce choix de structure permet de facilement distinguer les utilisateurs, et de stocker temporairement pour chacun les fichiers qu'ils déposent sans les mélanger (on ne veut pas qu'Alice reçoive la prédiction de Bob).

c. Sprint 3 : Unification

Toujours dans une optique de jonction des différentes parties, ce sprint fut, comme son nom l'indique, dédié à l'ajout de l'IA développée par l'équipe du même nom. Concrètement, cette équipe a généré un modèle, et écrit des scripts se basant sur ce modèle, permettant une prédiction en fonction des fichiers (t1,t1ce,flair et t2) donnés en entrée. Cette opération a pris lieu dans le backend, où il a fallu incorporer ces scripts et ce modèle à l'API. D'un point de vue séquentiel, cela change peu de choses. Simplement, au lieu de "simulation d'une prédiction", on a "appel du script de prédiction".

En parallèle de cette unification, les développeurs de la partie frontend ont développé un système de calques pour pouvoir superposer plusieurs images d'IRM, et notamment d'images de segmentation. Cela s'intègre dans la volonté de proposer une expérience complète lors de la prédiction sur l'application (le fichier de segmentation vient directement se superposer aux images déposées par l'utilisateur– on parle d'images d'anatomie).



Exemple de l'ajout de superposition de l'image de prédiction aux images d'anatomie

d. Sprint 4 : Déploiement

L'application répondant aux exigences demandées, ce sprint fut l'occasion de déployer l'application, afin d'avoir une démonstration en conditions "réelles" de Visual Gliome. Afin d'y parvenir, il a été question de Dockeriser l'application. Concrètement, des fichier Dockerfile ont été écrits pour le frontend et pour le backend, afin de les containeriser. Ensuite, un fichier docker-compose a aussi été écrit, dans le but de coordonner les containers.
Une fois cela fait, un de nos membres a hébergé l'application.
D'un point de vue utilisateur, ce sprint lui permet de ne pas avoir à exécuter lui-même l'application en local, et de seulement se rendre sur http://visualgliome.bdezordo.com/ pour l'utiliser.