



Classification des Tumeurs Cérébrales

sur STM32 avec x-cube-ai

BASKARAN Amalan
12001105
INSTRUMENTATION an2
06/02/2024

La détection et la caractérisation des tumeurs cérébrales sont des enjeux majeurs et complexes en science-médecine, car elles conditionnent le diagnostic, le pronostic et le traitement des patients. Les gliomes, les adénomes hypophysaires et les méningiomes sont les tumeurs cérébrales les plus courantes et ont des propriétés morphologiques et génétiques diverses. L'imagerie par résonance magnétique (IRM) est la technique la plus utilisée pour l'identification des tumeurs cérébrales.

Dans ce projet, je vais partir sur une base de données avec des images de différentes tumeurs cérébrales : Glioma, Meningioma, tumeurs hypophysaires et un sujet sain. L'objectif est d'utiliser une technique d'apprentissage profond (Réseau neuronal convolutif [CNN]) pour entraîner un modèle capable de reconnaître les tumeurs sur des images inédites. Ce modèle sera ensuite intégré à un stm32 pour développer un système de détection des tumeurs cérébrales avec l'image donnée avec une interface.

I. Construction du Modèle CNN

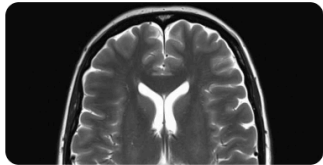
La construction d'un modèle nécessite des données complètes avec un niveau de confiance et d'utilisation élevés, car il s'agit d'une classification médicale. Ce type de modèle doit être capable de distinguer les différentes catégories de maladies ou de symptômes à partir des données fournies. Il faut donc s'assurer que les données sont de bonne qualité, représentatives et cohérentes.

Dans le cadre du cours de Diagnostic par apprentissage, nous avons découvert le site KAGGLE, une plateforme incontournable pour les passionnés de data science. Ce site propose des jeux de données variés et de qualité, qui permettent de s'entraîner à nos modèles.

J'ai décidé de partir sur une base de données qui a déjà subi un post traitement, c'est-à-dire suppression des images aberrantes et une unification des taille de l'image.

J'ai décidé d'utiliser le Dataset suivant :

Brain tumors 256x256
A Refined Brain Tumor Image Dataset with Grayscale Normalization and Zoom



Usability ⓘ
8.75

License
CC0: Public Domain

Ce Dataset a une note d'utilisabilité de 8.75/10 avec une license CC0 qui dit :

You can copy, modify, distribute and perform the work, even for commercial purposes, all without asking permission. See **Other Information** below.

<https://www.kaggle.com/datasets/thomasdubail/brain-tumors-256x256>

Ce dataset contient les éléments suivants :





 glioma_tumor 901 files	 meningioma_tumor 913 files	 normal 438 files	 pituitary_tumor 844 files
--	--	--	---

Figure : Dataset

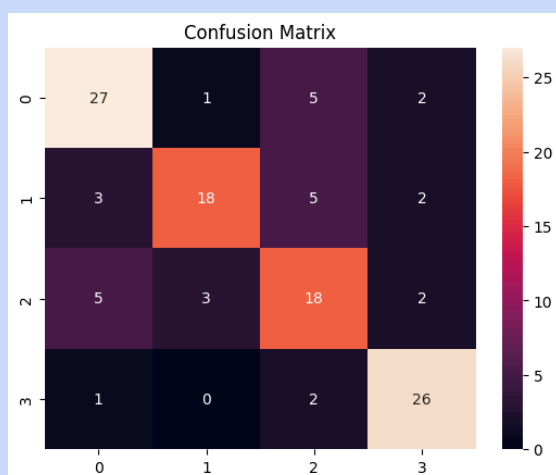
Ce dossier contient des images RGB de 256x256 pixels qui ont été normalisées. Pour entraîner notre modèle, nous devons connaître les contraintes de notre carte stm32, qui a une limite de mémoire à respecter.

- Ultralow-power STM32L4 Series MCUs based on ARM® Cortex®-M4 core with 1 Mbyte of Flash memory and 128 Kbytes of SRAM, in LQFP100 package

Selon les exigences de la carte, nous ne pouvons pas utiliser plus de 1 Mo de mémoire flash. Cela signifie que nous devons créer un modèle lightweight qui optimise l'espace et les performances. Nous devons donc réduire la complexité et la taille du modèle.

Afin d'implémenter un système de classification d'images sur une carte stm32, j'ai opté pour un modèle simplifié et de faible dimension. Cela a entraîné une baisse de la performance de mon système, mesurée par le taux de précision. Dans le secteur médical, il serait nécessaire d'atteindre une précision proche de 100% et de disposer d'un grand volume de données pour pouvoir déployer un tel système. Mais comme je mène ce projet sur une carte stm32 et que je souhaite réaliser une classification d'images, j'ai décidé de me passer de ces exigences.

Pour ma part, j'ai effectué les étapes suivantes pour préparer les données. J'ai sélectionné 150 images pour chaque classe et je les ai transformées en images en niveaux de gris. Ensuite, j'ai redimensionné les images à une dimension de 55x55 pixels. j'ai effectué le réseau de neurones convolutif de manique qu'on a appris en cours de diagnostic par apprentissage. Après avoir entraîné on obtient la matrice de confusion et score suivante :



```
score = CNN.evaluate(X_test, y_test)
print('Test Accuracy: {}'.format(score[1]))

4/4 [=====] - 0s 36
Test Accuracy: 0.7416666746139526
```

Figure : Matrice de confusion & Score

Pour sauvegarder le modèle après l'entraînement, nous pouvons choisir entre deux formats : le format Keras (avec l'extension .h5) ou le format TensorFlow Lite (avec l'extension .tflite). La principale différence entre ces deux formats est que le modèle au format TensorFlow Lite occupe moins d'espace que le modèle au format Keras pour les mêmes performances d'où j'ai privilégié model tensorflow lite.

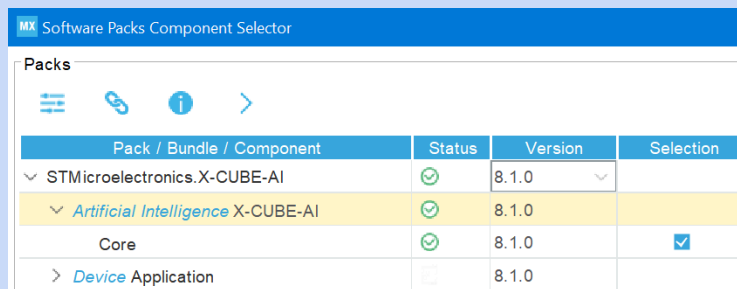
Remarque : Lorsque j'ai essayé de sauvegarder mon modèle directement sous le format .tflite, j'ai rencontré des problèmes. Pour les éviter, j'ai d'abord sauvegardé mon modèle en .h5, puis je l'ai converti en .tflite avec le convertisseur TensorFlow Lite. Ainsi, j'ai pu utiliser mon modèle dans mon application sans souci.

en finale j'ai obtenu mon modèle .tflite avec une taille de 819 ko qui satisfait bien les contraintes posées par notre carte STM32.

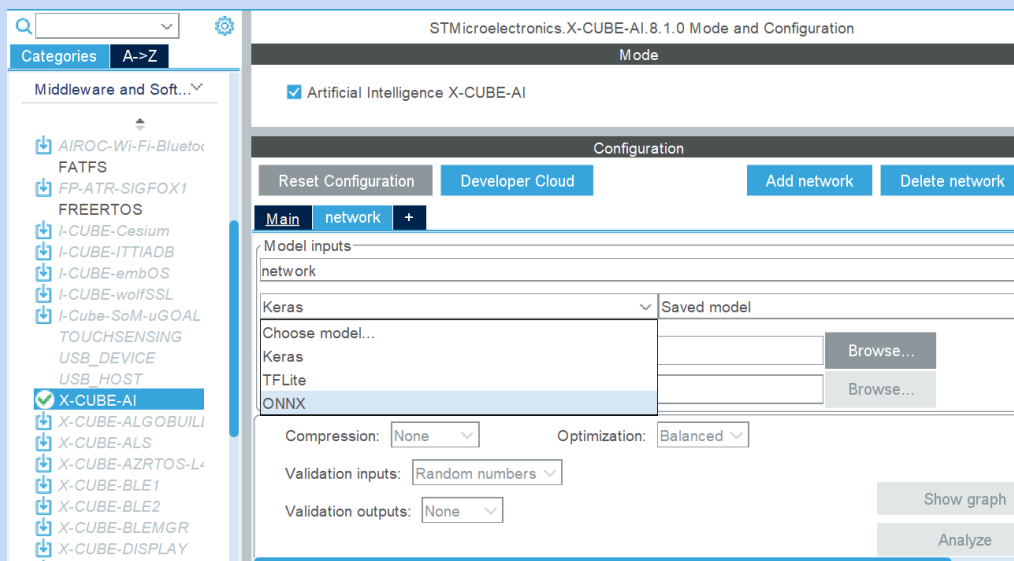
Vous trouverez en jointe les codes utilisés pour générer le modèle, convertir le modèle et testé le modèle en python.

II. Implementation du modele sur STM32

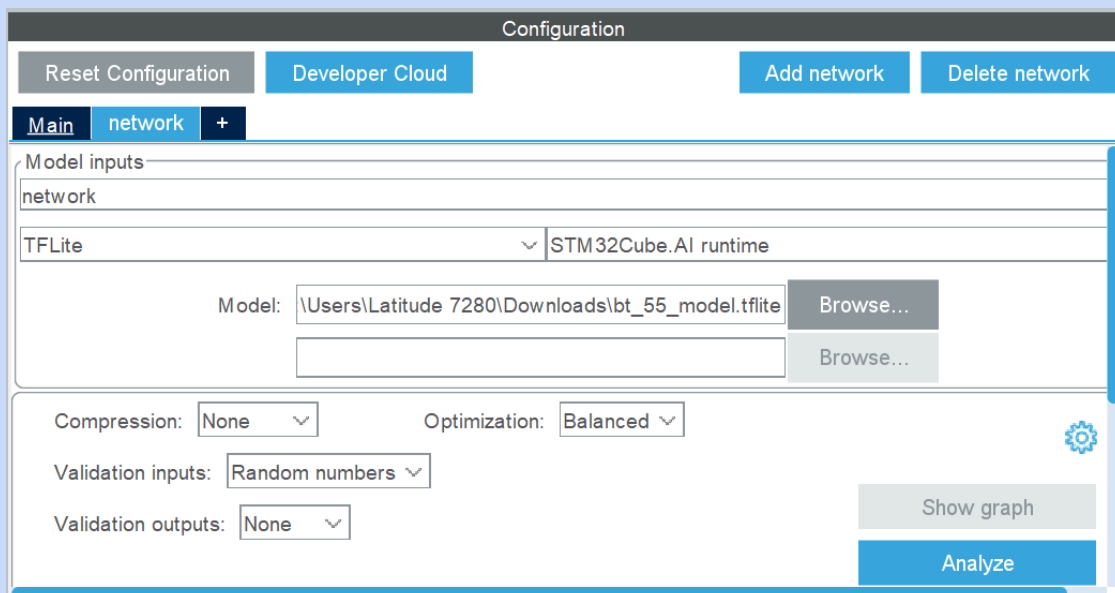
Dans CubeMx nous activons x-cube-ai core dans la fenêtre software packs :



choisissons x-cuba-ai dans middleware, on aura une fenêtre comme ceci :



comme on veut déployer un modèle TFLite, on choisit notre modèle :



Ensuite il faudra cliquer sur analyse pour voir notre modèle est compatible


```

Model file:      bt_55_model.tflite
Total Flash:     864764 B (844.50 KiB)
  Weights:       834044 B (814.50 KiB)
  Library:       30720 B (30.00 KiB)
Total Ram:       65532 B (64.00 KiB)
  Activations:   65532 B (64.00 KiB)
  Library:       0 B
  Input:         12100 B (11.82 KiB included in Activations)
  Output:        172 B (included in Activations)
Done
Analyze complete on AI model

```

Si ce message apparaît, cela veut dire que notre stm32 est capable de tourner notre modèle. sinon il y aura un message en rouge indiquant que la taille dépasse la taille de mémoire flash.

De plus j'ai activé uart1 avec un DMA circulaire sur uart_rx afin de faire les débogages.

En ce qui concerne le code STM nous facilite donne des exemples de fonctionnement de x-cube-ai.

Pour effectuer une prédiction nous utilisant 3 fonction qui sont donnés par STM qui sont :

- AI_Init() //créer une instance du modèle
- AI_Run() // génère la sortie en fonction des input données
- argmax() //trouver la classe

ces 3 fonction sont sous une licence

si nous fouillons le fichier Network.h on trouve les information sur les entrée sortie du modèle :

```

48 #define AI_NETWORK_IN_1_FORMAT      AI_BUFFER_FORMAT_FLOAT
49 #define AI_NETWORK_IN_1_HEIGHT      (55)
50 #define AI_NETWORK_IN_1_WIDTH       (55)
51 #define AI_NETWORK_IN_1_CHANNEL     (1)
52 #define AI_NETWORK_IN_1_SIZE        (55 * 55 * 1)
53 #define AI_NETWORK_IN_1_SIZE_BYTES  (12100)

```

```

68 #define AI_NETWORK_OUT_1_FORMAT      AI_BUFFER_FORMAT_FLOAT
69 #define AI_NETWORK_OUT_1_CHANNEL     (43)
70 #define AI_NETWORK_OUT_1_SIZE        (43)
71 #define AI_NETWORK_OUT_1_SIZE_BYTES  (172)

```

On peut notamment trouver la taille d'entrée de notre modèle qui est 55x55x1 donc c'est bien notre image en niveau de gris de 55x55.

X-cube-ai nous permet de simplifier le processus d'importation de notre modèle en acceptant un vecteur comme entrée au lieu d'une matrice. Ainsi, même si nous avons entraîné notre modèle avec des données de dimension quelconque, l'entrée de modèle sous CubeIDE serait forcément un vecteur donc on peut facilement importer et appliquer des données depuis l'extérieur avec uart par exemple.

```
66 float aiInData[AI_NETWORK_IN_1_SIZE];
67 float aiOutData[AI_NETWORK_OUT_1_SIZE];
```

Figure : Déclaration des vecteur I/O du modèle.

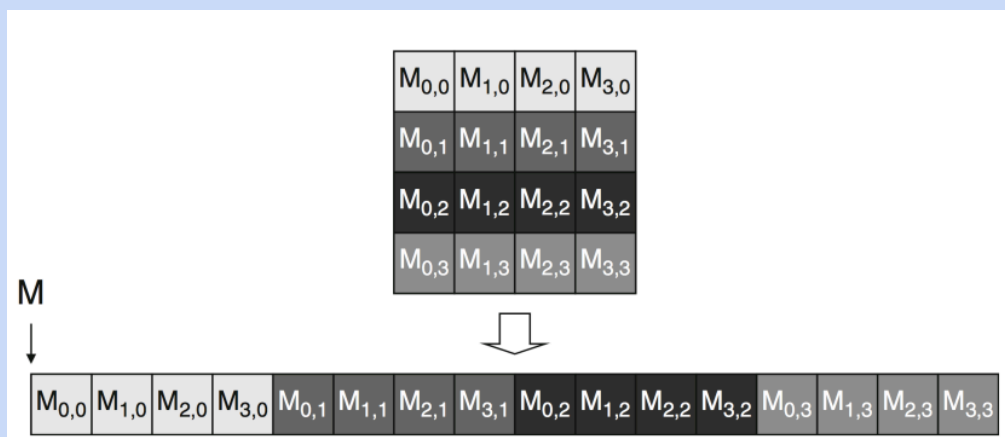


Figure : Image to vecteur d'entrée

Une façon de représenter l'image d'entrée est d'utiliser un vecteur de dimension 3025. Ce vecteur contient les valeurs des pixels de l'image, qui a une taille de 55x55x1. Ainsi, chaque élément du vecteur correspond à un pixel de l'image.

Pour avancer dans le projet, nous devons vérifier que notre modèle fonctionne correctement avec des données aléatoires. j'ai effectué ce test avec succès, ce qui me permet de passer à l'étape suivante.

Nous voulons importer une image depuis l'extérieur, par exemple depuis votre pc par une liaison uart, avant d'importer il faudra être sûr que chaque pixel soit importé correctement.

III. Envoie des données sur Uart & Débogage

Comme nous travaillons sur un gros taille de vecteur (3025), il est important de vérifier la bonne transmission, Pour cela IDE Keil Uvision propose des meilleurs outils de débogage en temps réel. Ainsi j'ai utilisé la librairie pyserial de python afin d'envoyer mes données sur l'uart.

```
1  import serial
2  ser = serial.Serial()
3  ser.baudrate = 115200
4  ser.port = 'COM10'
5  ser.open()
6  data_array = [1, 4, 7, 12, 45, 78, 91]
7  values = bytearray(data_array)
8  ser.write(values)
9  ser.close()
```

Figure : Code Test UART

Comme nous avons configuré notre UART1 avec une DMA circulaire sur RX, nous créons une variable rx_data afin de stocker les valeurs reçu :

```
uint8_t rxdata[7];
```

ensuite nous attendons la réception des données :

```
HAL_UART_Init(&huart1);
HAL_UART_Receive_DMA(&huart1, (uint8_t*)rxdata, sizeof(rxdata));
```

on pourra utiliser aussi la méthode suivant qui est de retransmettre les données reçu sur UART pour vérifier bon fonctionnement :

```
void HAL_UART_RxHalfCpltCallback(UART_HandleTypeDef *huart)
{
    HAL_UART_Transmit(&huart1, rxdata, sizeof(rxdata), 200);
    HAL_UART_Receive_DMA(&huart1, (uint8_t *)&rxdata, sizeof(rxdata));
}
```

Cette approche comporte beaucoup d'inconvénients pour notre application, car elle ne nous permet pas de contrôler les données numériques que nous envoyons et ignore également le fonctionnement interne de ce processus.

Le meilleur solution est d'ouvrir le <debug session> sur Keil
cette session permet de voir les mémoire, registres,
observer les variables en temps réel et donne aussi un code assembleur.



c'est un très bon outil comparé à la méthode précédente. voici comment se présente l'outils:

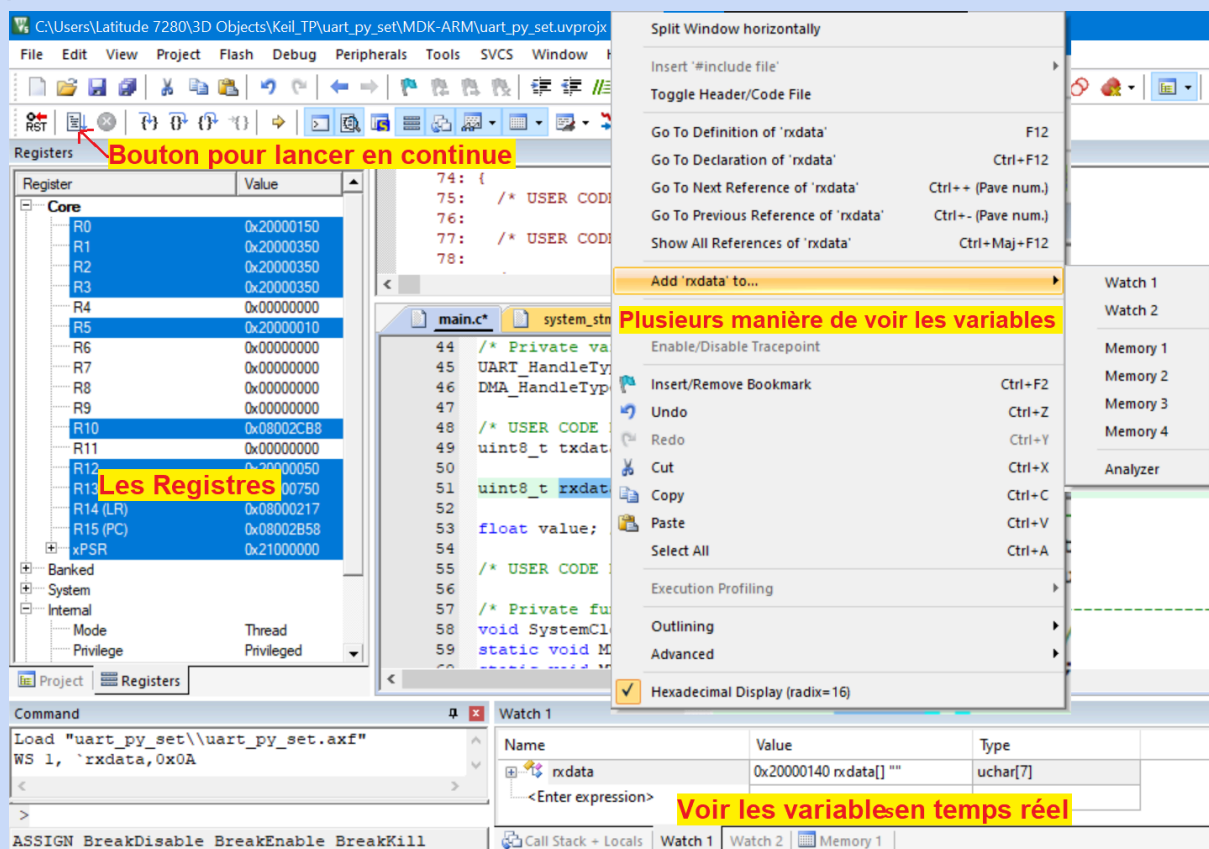


Figure : Débogages sur Keil

une fois lancé le script sur Keil, nous lançons notre script python et observe les éléments suivantes dans la fenêtre watch1 de Keil :

Name	Value	Type
rxdata	0x20000140 rxdata[] "□□\a\f-N["	uchar[7]
[0]	1	uchar
[1]	4	uchar
[2]	7	uchar
[3]	12	uchar
[4]	45 '-'	uchar
[5]	78 'N'	uchar
[6]	91 '['	uchar

Figure : Données reçu

Ce genre de débogage nous aide à vérifier les types de données et à confirmer que la colonne value contient bien les valeurs qu'on a transmises avec python.

```
arr = np.random.randint(0, 100, size=7)
values = bytearray(arr.tolist())
ser.write(values)
```

Figure : Code pour envoyer un vecteur aléatoire numpy

Remarque : *Il faudra convertir un vecteur numpy en un vecteur normal (avec .tolist()) avant d'écrire sur serial port sinon il écrira avec une décalage.*

```
22 image = Image.open('im_55.jpg')
23 image = image.convert("L") # Convert the image to grayscale
24 numpydata = asarray(image)
25 x = numpydata.reshape(-1) #2D to 1D
26 values = bytearray(x.tolist())
27 ser.write(values)
```

Figure : Code pour envoyer une image

Avec ces 3 tests, les données ont été transmises correctement.
lorsqu'on transmet une image de 55x55 (vecteur de 3025 éléments)

```
51 uint8_t rxdata[3025];
```

On note que le système prend environ une trentaine de secondes pour transmettre ces 3025 éléments par liaison uart.

Une fois que les test sont réussi nous passons au création d'interface pour être facilement utilisé par un utilisateur.

IV. Interface avec PyQt

Pour transmettre des données de type texte, un terminal suffit, mais pour une image, il faut une interface plus adaptée pour faciliter l'usage. Ainsi, nous pouvons améliorer l'expérience utilisateur en créant une interface dédiée à l'envoi d'images.

J'ai d'abord essayé de créer une interface qt/c++, mais je me suis vite rendu compte que c'était trop compliqué. J'ai donc opté pour PyQt, qui me semblait plus simple et plus adapté à mes besoins. Mon objectif était de créer mon interface entièrement en code python sans recourir aux logiciels comme QTCREATOR. J'ai donc suivi des tutoriels sur la gestion des layouts et les widgets de base que j'utilise dans l'interface.

J'ai conçu une interface qui se compose de deux parties. La première partie gère la communication par port série, et la deuxième partie permet l'envoi d'images. La partie communication par port série qui permet à l'utilisateur de configurer les paramètres d'une liaison série, tels que le port COM, le débit en baudes, le bit de parité et le nombre de bits ainsi pour établir une connexion avec la carte STM32. L'interface affichera uniquement les port COM disponibles.

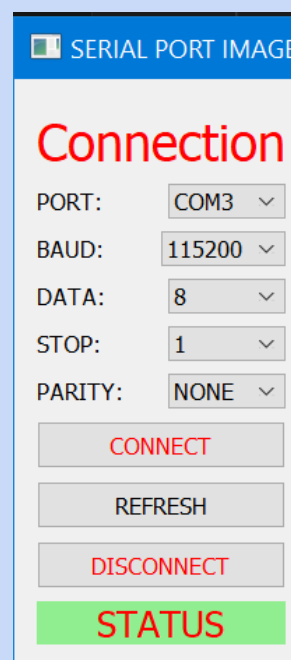


Figure : Interface partie communication

La deuxième partie permet de parcourir nos fichiers afin de choisir une image pour envoyer vers la carte stm32.

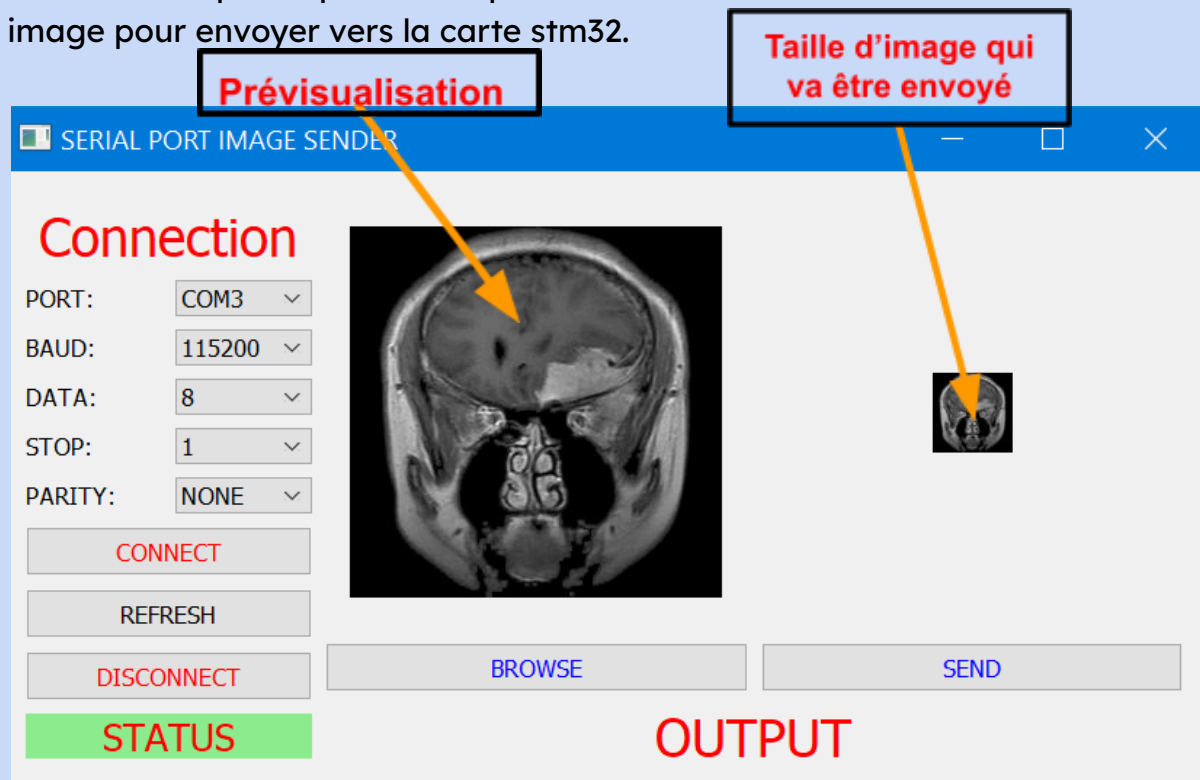
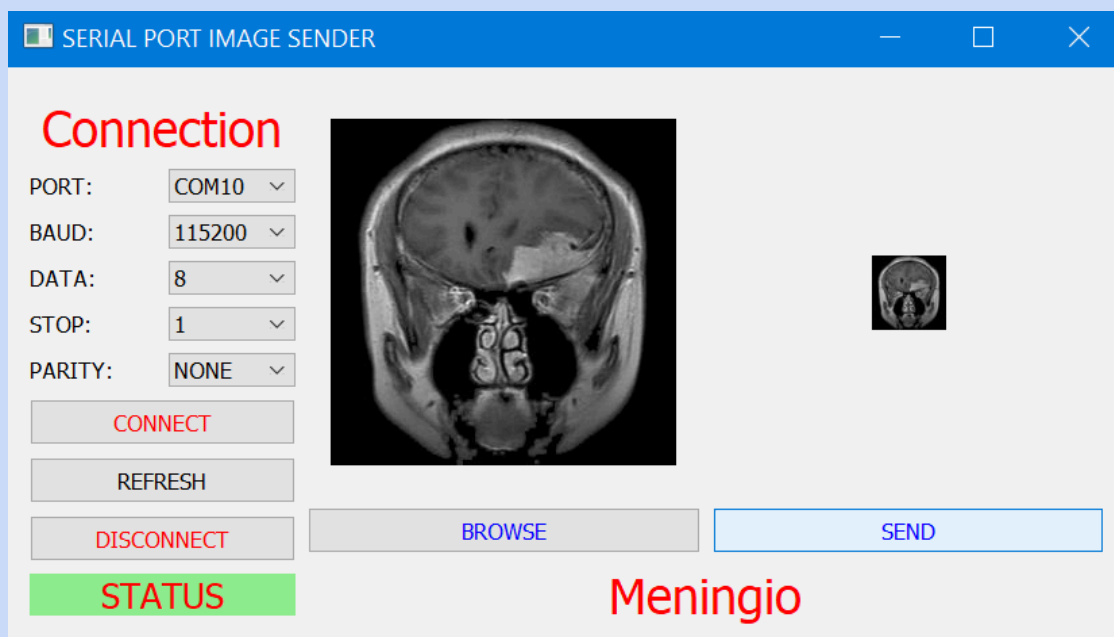


Figure : Interface avant l'envoi

A chaque fois qu'une image est envoyée la communication série sera fermé (fait exprès dans code) donc il faudra connecter à chaque fois.

V. Résultat Final

Test avec quelques images :


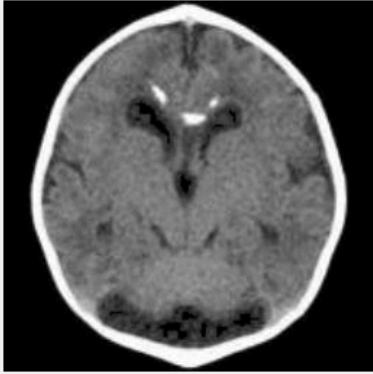


SERIAL PORT IMAGE SENDER

Connection

PORT: COM10
BAUD: 115200
DATA: 8
STOP: 1
PARITY: NONE

CONNECT
REFRESH
DISCONNECT
STATUS



BROWSESEND


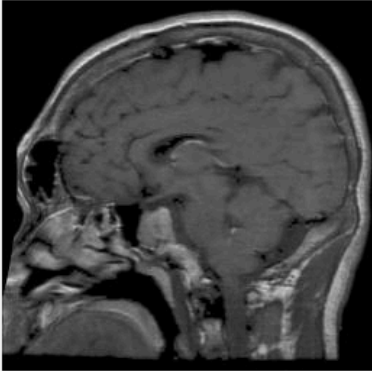
Glioma

SERIAL PORT IMAGE SENDER

Connection

PORT: COM10
BAUD: 115200
DATA: 8
STOP: 1
PARITY: NONE

CONNECT
REFRESH
DISCONNECT
STATUS



BROWSESEND

Pituitar

VI. Conclusion

Ce projet a été une occasion pour moi de développer les compétences nécessaires avec les STM32 pour réaliser un système embarqué. J'ai pu apprendre à gérer les différentes étapes du projet, à établir une communication efficace entre les composants (PC \Leftrightarrow STM32) et à créer une interface utilisateur adaptée.