Developing a Live Surgical Aid for Brain Tumor Resection using Augmented Reality and Deep Learning

Table of Contents

Abstract

Approximately 300,000 people are diagnosed with brain cancer each year, and only 20% survive, making brain cancer one of the deadliest cancers. In recent years, the development of minimally invasive surgeries has allowed for reduced recovery time and risk of infection, however, they do have their downsides; surgeons do not directly see the surgical site and are separated from preoperative scans from which they get the tumor location and other necessary data, resulting in a loss of visual and haptic feedback. To aid this issue and significantly increase the safety of minimally invasive brain tumor resections, a live surgical aid was constructed utilizing both deep learning and augmented reality. A deep learning model, using a 3D U-NET, was programmed and trained to automatically segment brain tumors in 3D from MRIs. A second deep learning model utilizing the RE-NET algorithm was constructed to segment cerebral vasculature from MRAs. A third deep learning model was then trained to recognize a target in real life, a 3D-printed head that would serve as the demonstration target for augmented reality. The 3D models were then implemented in Unity and uploaded to an Android-based augmented reality headset. This project has far-reaching and extensive applications, the main one being intraoperative aid during surgery by providing a comprehensive visualization of the tumor and vascular data, removing the need to switch perspectives between the surgical site and preoperative scans. This project can also be used as an effective aid in preoperative planning and surgical training.

Developing a Live Surgical Aid for Brain Tumor Resection using Augmented Reality and Deep Learning

In 2020, an estimated 308,000 people worldwide were diagnosed with brain cancer and of those people, 251,000 died (Siegel et al., 2021). Brain cancer is the tenth deadliest cancer for both men and women and only 36% of patients survive the five years following their diagnoses.

The most common treatment for brain cancer is surgical removal of the tumors through a craniotomy, and this is usually the only viable treatment that is needed (“Brain Tumor”, n.d.). Recent developments in surgical technique have allowed for a minimally invasive approach to tumor resection as opposed to the standard craniotomy (“Surgery for Brain”, n.d.). When compared with the craniotomy, these surgeries are much safer as they reduce the risk of infection and damaging brain matter (“Minimally Invasive”, 2022). However, current systems for minimally invasive surgeries require surgeons to rely solely on two-dimensional (2D) external displays showing pre-operative scans, as well as a camera feed from the probe the surgeon uses to perform the surgery, as opposed to being able to directly see the patient’s anatomy (Meola et al., 2017). This results in a loss of haptic feedback, the ability to create and feel pressure, possibly creating a disconnect with the surgery from being unable to see the surgical site in real life and having to rely on deciphering the location of the tumor from separate 2D screens (Meola et al., 2017). With all these issues, only experienced neurosurgeons and facilities with advanced technologies can perform minimally invasive tumor resections, and even still, the risks of damaging vascular or nervous tissue are high due to the visual disadvantages, the small surgical field, and the cognitive overload of the procedure on the surgeon (Meola et al., 2017).

Technologies, such as microscopes and endoscopes, have been developed to help combat some of the issues surgeons performing minimally invasive operations face (see Appendix A).

These utilities, however, separate the surgeon from the screens with preoperative scans, so the surgeon must continuously switch perspectives to be able to operate the tools.

To avoid the burden of switching perspectives and translating information from separate 2D screens to the real-life surgical site, a comprehensive neuronavigational system is needed to visualize all of the data in one place. Augmented Reality (AR), a term used to describe the combination of real-life and computer-generated content, has been proposed and developed as a viable solution for certain minimally invasive operations that require precision with low visibility (Salehahmadi & Hajialiasgari, 2019). AR could be a beneficial utility to aid in brain tumor surgery by visualizing the tumor as well as nervous and vascular structures in one comprehensive 3D view (see Appendix B).

AR through the use of a head-mounted display has been applied to surgeries. A recent surgery done at Johns Hopkins utilized AR for spinal fusion by displaying the exact positioning of the screws that were to be implanted into the spine on head-mounted AR goggles ("Johns Hopkins", 2021). Surgeons at St. Mary's hospital in London have also integrated AR into some of their surgical procedures by overlaying bones, muscles, and blood vessels onto the surgeon's field of view (Best, 2018). AR has also been applied to certain orthopedic surgeries, and surgeons who wore AR goggles were found to achieve a higher precision cut than those who did not (Jud et al., 2020).

For both traditional and minimally invasive procedures, AR had been proposed and tested on certain operations like spine surgery, but its application on cancer removal surgeries, specifically brain tumor resections, has yet to be developed and tested. In this project, an AR system was constructed to transform unlabeled 3D medical scans such as Magnetic Resonance Imaging (MRIs) into a comprehensive visualization of the tumor and peripheral structures in AR.

The AR system first segments a tumor from preoperative MRI scans using a trained deep learning model and generates a 3D segmentation of the tumor (see Appendix C). Then, cerebral vasculature is segmented from a Magnetic Resonance Angiography (MRA) to create a comprehensive patient-specific 3D model (see Appendix C). These 3D models are then put in Unity and a deep learning model is again used to recognize a certain physical marker, in this case a 3D-printed head to simulate a person, and superimpose the patient-specific models onto the head. This 3D model is displayed on the patient in the surgeon's field of view through an AR headset by utilizing physical markers and 3D location mapping. This project provided a comprehensive visualization of the tumor and surrounding vascular structures overlayed in correct positions and orientations on the patient to reduce the perceptual and cognitive burden of the surgery, maximize safety, and reduce risk of damage to surrounding tissue. The AR system can be effectively utilized for preoperative planning, medical and surgical training, and during surgical procedures.

Methods

There are no safety hazards for this project, as most of the project is programming. A possible concern is dizziness, which in an unlikely event can result from using virtual or augmented reality.

Materials

- Android Phone
- Computer
- Webcam
- MATLAB
- Unity Game Engine

- Vuforia
- Google Colab
- Android Development Kit
- BraTS2020 Dataset

  https://www.kaggle.com/datasets/awsaf49/brats20-dataset-training-validation
- TubeTK Healthy MR Database

  https://public.kitware.com/Wiki/TubeTK/Data
- Google Colab
- Slicer
- Github Repository: neurolabusc/nii2mesh
- 3D printer
- 2 kg Grey PLA filament
- 100 g Blue PLA filament
- Model Target Generator
- Visual Studio Code

Procedures

To begin the project, MATLAB was opened on a desktop computer. The file "BRATS_001.nii.gz" was imported from both the image and label folders from the BraTS2020 Dataset into the MATLAB project by dragging the file into the left sidebar.
The full model and label model were assigned to different file path variables by double clicking on their names in the sidebar and using the code shown in Appendix E.

The MATLAB nifitread() function was used to load the data.

The RGB value (255, 0, 0) was passed in as a colormap argument for volshow() to change the label to a different color and the alpha value 100 was passed into the alphamap argument of the full model to make it partially transparent so the tumor can easily be seen. After finishing the code, the MATLAB program was run using the green "Run" button at the top, and an interactive window opened with a 3D viewer displaying the full brain as well as the highlighted tumor (see Appendix E for code).

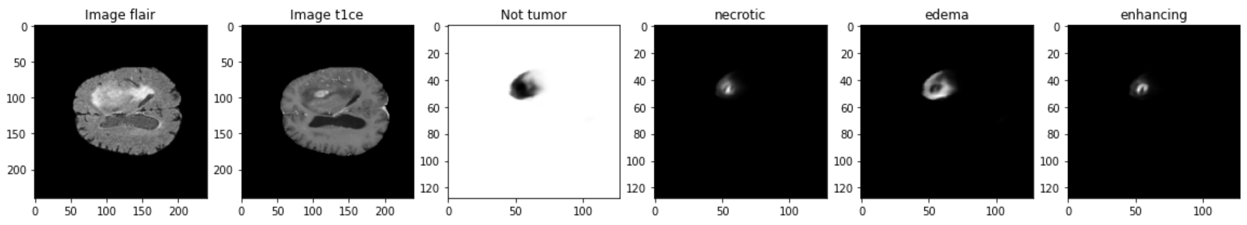
To construct the AI model that was able to outline, or segment, the brain tumor in each slice comprising the 3D model, Google Colab was opened in a web browser and a new notebook was created. The runtime of the notebook was set to GPU (Graphical Processing Unit) by clicking the Runtime dropdown, allowing access to a powerful GPU across the cloud.
The pre-installed Python libraries (cv2, glob, PIL, numpy, pandas, seaborn, matplotlib, keras, sklearn, and tensorflow) were imported into the notebook (see Appendix F for code). The libraries nilearn and nibabel were also installed.

The classes were defined using a dictionary; 0 is "not tumor", 1 is "necrotic core", 2 is "edema", and 3 is "enhancing". Each pixel of the images in the .nii files is already labeled with one of these values (see Appendix G).

The data was displayed in slices by loading a test .nii file from the BraTS Dataset with nibabel.load() and displaying the file with plt.imshow() as shown in Figure 1.

Figure 1

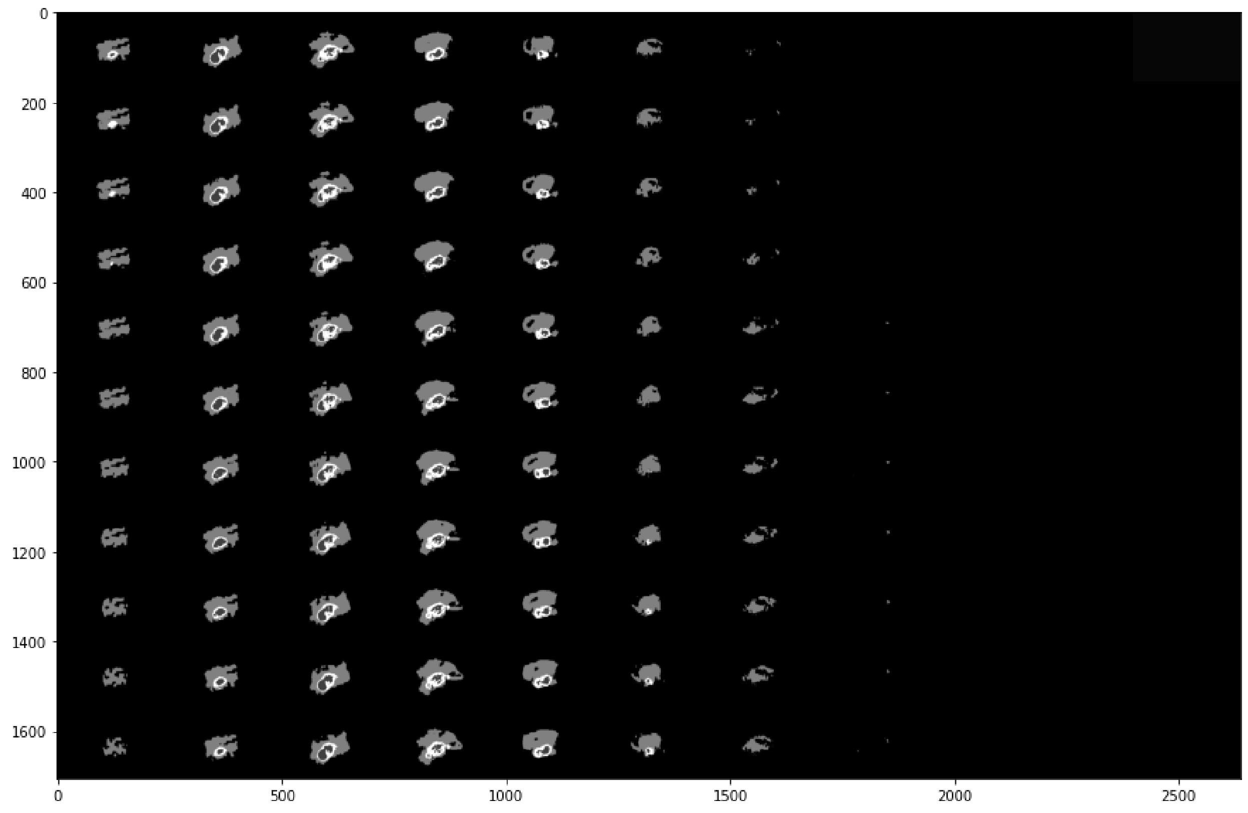
Slices of each class for a sample label



All slices of the data were also displayed at once, shown in Figure 2, to visualize in which slices was the brain placed, and which slices could be spliced off to reduce memory (see Appendix G).

Figure 2

All slices of a Label



The U-Net is the algorithm that seeks to predict the borders of the tumor. The U-Net algorithm was constructed in Keras, using a series of Convolutions, Max-Pooling, and Up-convolutions to alter the image so it is easier for the computer to process. Firstly, two convolutions were added to the input using Keras conv2D() method and then a Max Pooling layer with Keras MaxPooling2D(). This process was repeated three more times. Then, two convolutions and an up-convolution were added with the Keras method conv3DTranspose().

This process was repeated three more times. Finally, three convolutions were added to output the segmentation map. Appendix H shows the code for the algorithm.

The accuracy of the model was tested using the DiCE (Diverse Counterfactual Explanation) coefficient, which returns a value depending on how much the prediction overlaps with the real outline, or the ground truth. A function called diceCoefficient() was created, and it took in two arguments, truth and prediction (see Appendix I).

Keras.flatten() was called on both truth and prediction, and then the two were multiplied to find the intersection of truth and prediction. The following code shows this:

The value of two times the pixels in the intersection divided by the sum of the pixels in both truth and prediction were returned using 2*len(intersection)/(len(truth)+len(prediction)).

The entire BraTS Dataset was then split into three parts: 70% train, 15% validation, and 15% test (see Appendix J).

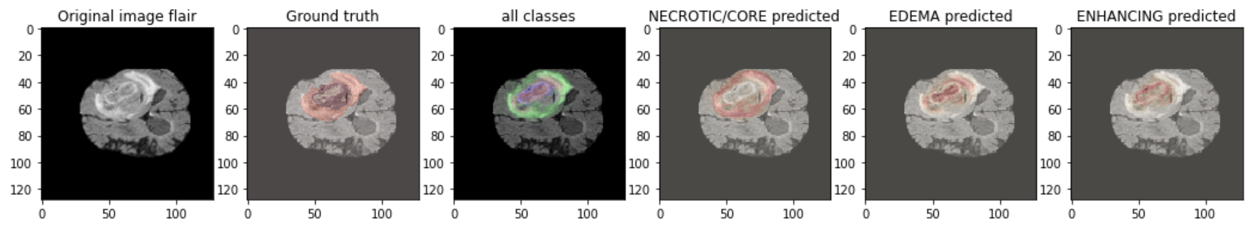
The train data, validation data, and test data sets were loaded in small batches using Keras DataGenerator. The DataGenerator() function was called on each of the train, validation, and test IDs (see Appendix K).

The model was trained with model.fit(), passing the training data, 35 epochs (number of cycles), and validation data as arguments.

A file was loaded in from the test dataset using nibabel.load() and the AI model was run on it with model.predict(). Matplotlib was used to display the prediction, shown in Figure 3, with plt.imshow() (see Appendix L and Appendix M).

Figure 3

Class Predictions and Ground Truth



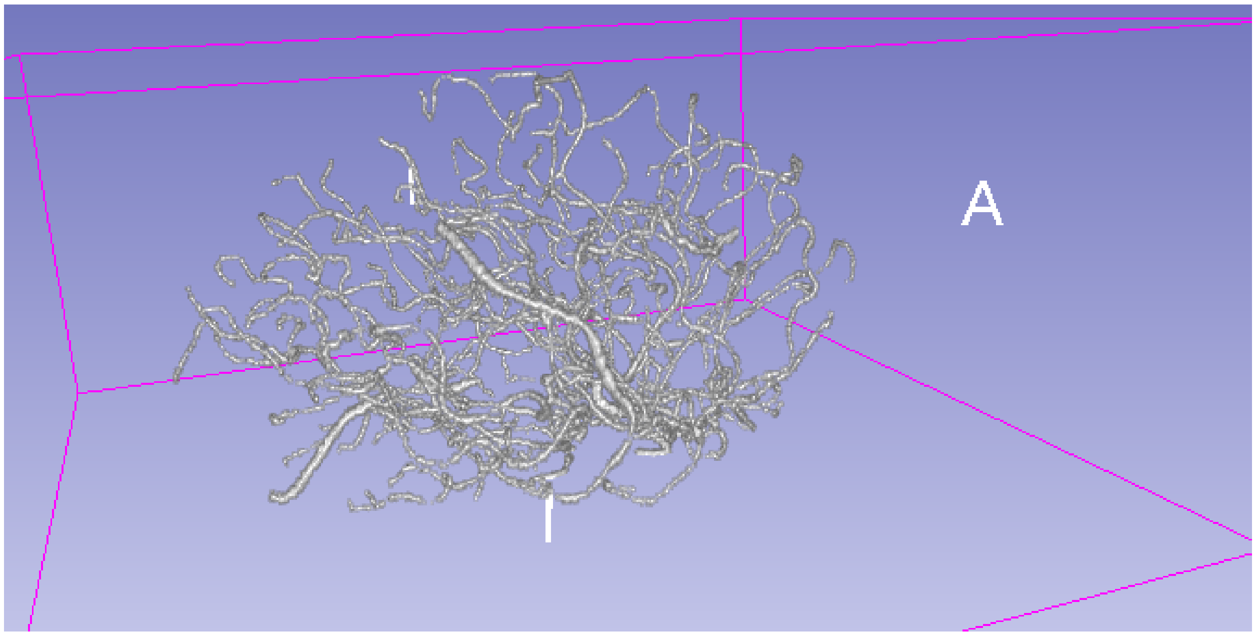
All the images from the segmented file were gathered into a NumPy array with np.array() and it was converted into a .nii file with nibabel.Nifti1Image(). The file was then saved on the computer as a .nii file.

The github repository neurolabusc/nii2mesh was then used to convert the segmented file, which contains the tumor boundaries, and the original, which contains the entire brain, to .obj files to be visualized in augmented reality later on.

Cerebral vasculature was generated from magnetic resonance angiographies (MRA) through deep learning to be visualized in augmented reality along with the brain and tumor. The dataset, TubeTK 100 healthy MRAs, was first downloaded from Kitware Medical datasets. The MRA training data had to be converted from the .mha file format to the .nii file format in order to be fed into the algorithm, so several helper scripts were programmed to automatically iterate through the folders and convert files using the SimpleITK library. The 42 labels (3D binary models of the vasculature), stored in .tre file format, also had to be converted into .nii files. A helper script was used to convert them into a .vtp poly data file, and then each one was manually inputted into the Slicer software, along with its respective .mha file. Each file was converted into a segmentation as shown in Figure 4, resized to the correct dimensions according to the respective .mha file, and then exported as a .nii file.

Figure 4

Cerebral Vasculature Label Rendered in 3D Slicer



The folders were rearranged and uploaded to Google Drive, so that the training folder contained 32 images and labels, and the test folder contained 10 images and labels. The rest of the MRAs were put in an unlabeled folder.

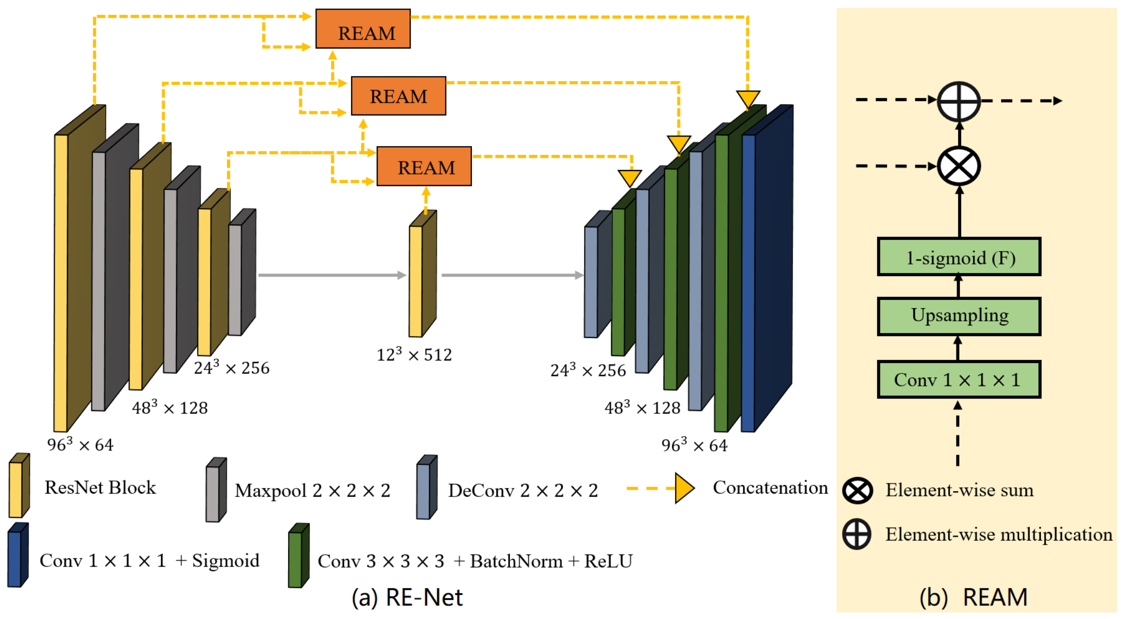
A new Google Colab notebook was opened and the runtime type was changed to Premium GPU. The necessary libraries were imported and installed and Google Drive was mounted (see Appendix N). To ensure the images and labels were generated correctly a function was written and called to visualize slices of the data.

The Visdom framework, a library developed by Meta, was used to collect, monitor, and plot several segmentation metrics as the model was training. The server was set up on a local tunnel and was kept on for the duration of the training. A plotter object was created with methods to create graphs of seven segmentation metrics: loss, accuracy, sensitivity, specificity, intersection over union (IoU), DiCE coefficient, precision, and area under curve (AUC) (see Appendix O).

A specialized deep learning algorithm, known as an RE Net (Reverse Edge Attention Network) was implemented for cerebrovascular segmentation; the algorithm structure is shown in Figure 5 and the program is in Appendix P.

Figure 5

RE-NET Structure



Note. From "Cerebrovascular Segmentation in MRA via Reverse Edge Attention Network," by H. Zhang, L. Xia, J. Yang, H. Hao, J. Liu, Y. Zhao, 2020, *MICCAI 2020, 12266*(1), p. 66-75 (https://doi.org/10.1007/978-3-030-59725-2_7).

This algorithm featured a series of convolution and max pooling layers followed by a series of convolutions, batch normalizations, and deconvolutions. A form of data augmentation known as Random Patch Crop was also programmed to increase the variability of training data and therefore the overall accuracy of te model. Arrays of size 96 by 96 by 96 voxels were randomly selected to be removed from a training sample (see Appendix Q).

A data loader function was also written to prepare the data to be loaded into the model during training. It first fetches the file paths into an array and then loads each file into a Numpy array from the Nifti file format using NiBabel functionality.

A metrics function was written to calculate the several metrics listed above using the true positives, true negatives, false positives, and false negatives. This function is called during evaluation in training (see Appendix R).

Several training values were then defined in a dictionary such as learning rate and number of epochs. Functions were written to adjust the learning rate of the model and save the model as a .pkl file at certain intervals or whenever it achieved the highest DiCE Coefficient.

The model was then trained on an Nvidia Tesla T4 GPU through Google Colab and used a parallel computing platform known as CUDA for faster training. As the model was trained, the evaluation metrics for each epoch were plotted through Visdom (see Appendix S).

After the model was trained, an MRA from the same patient whose MRI was segmented by the brain tumor segmentation model was loaded as a numpy array. The model with the best DiCE coefficient was run on the array and outputted a binary mask (see Appendix T). This was then saved as a .nii file and verified in Slicer compared to the ground truth to verify the prediction was accurate. The neurolabusc/nii2mesh github repository was again used to convert the cerebral vasculature into a .obj file to be visualized.

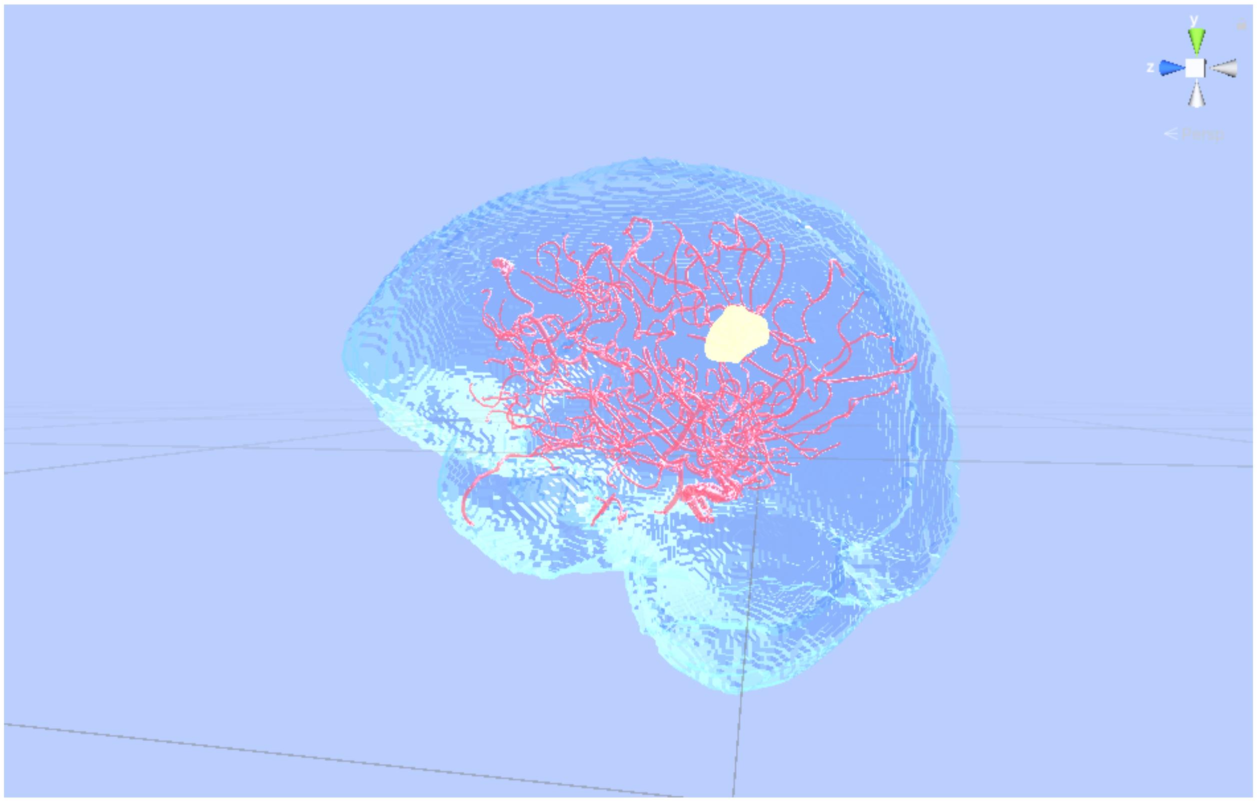
To simulate the brain, tumor, and cerebral vasculature being superimposed during training or live surgery, a head will be 3D printed for the models to be displayed inside of through augmented reality. To accomplish this, a head model was first designed in Fusion 360, and cutouts were made for colored blocks to be inserted to increase features and improve detection since the head is relatively uniform and featureless.

The head model was exported as a .obj file and imported into the Vuforia Model Target Generator. The Model Target Generator used Advanced Model Generation and trained a deep learning model using data from the head CAD so it would be able to recognize the head from all angles and superimpose 3D models on top of it with a greater degree of positional accuracy. After the model was trained, it was exported as a Unity Package.

A new Unity project was then created with Android Build Support installed, and Vuforia was added to the project. From the Vuforia menu, an AR Camera was added, and the ARCore requirement in its settings was set to “Optional”. The Unity Package was then dragged into the scene, and in the AR Camera settings, it was set as the main database. The 3D head model is now in the scene. The brain, tumor, and cerebral vasculature were then dragged in, and their transforms reset and aligned with the 3D head model, so they are positioned inside it. Different materials were created, and their colors, transparency, and emission were adjusted and applied to each of the 3D models for clearer visualization as shown in Figure 6.

Figure 6

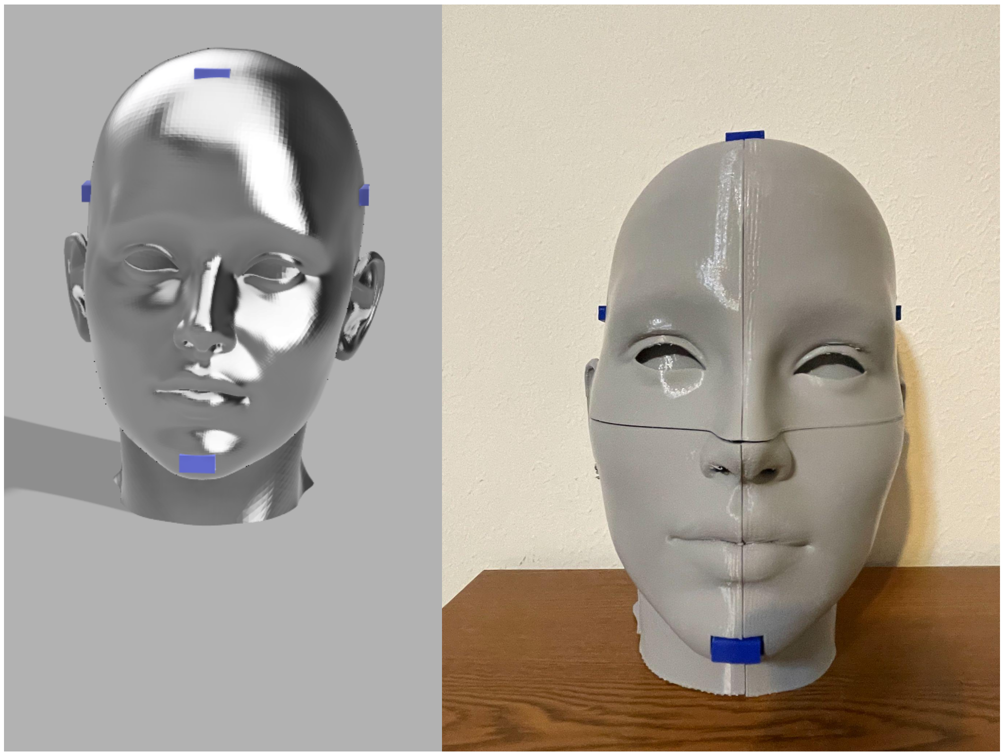
Brain, Tumor, and Cerebral Vasculature Rendered in Unity



For the detection to work, a physical head model must be present in real life, so the head CAD model was 3D printed as shown in Figure 7. The head model was resized to be 20cm by 28 cm by 24 cm and sliced into eight parts. The additional blocks that were previously added to increase the features of the head were also sliced out. Each of the eight head parts was printed with grey PLA filament, while the additional blocks were printed with blue PLA filament. After 3D printing was finished, the parts were glued together with Duco Cement and held with masking tape until the glue was set and cured.

Figure 7

CAD Head and 3D-Printed Head



An augmented reality headset also had to be printed to be able to better demonstrate how the visualizations would like in a surgical environment. The headset was printed and the Android phone was placed inside. After plugging in the Android phone to the PC and enabling USB debugging, the Unity project was then built and run on the Android phone.

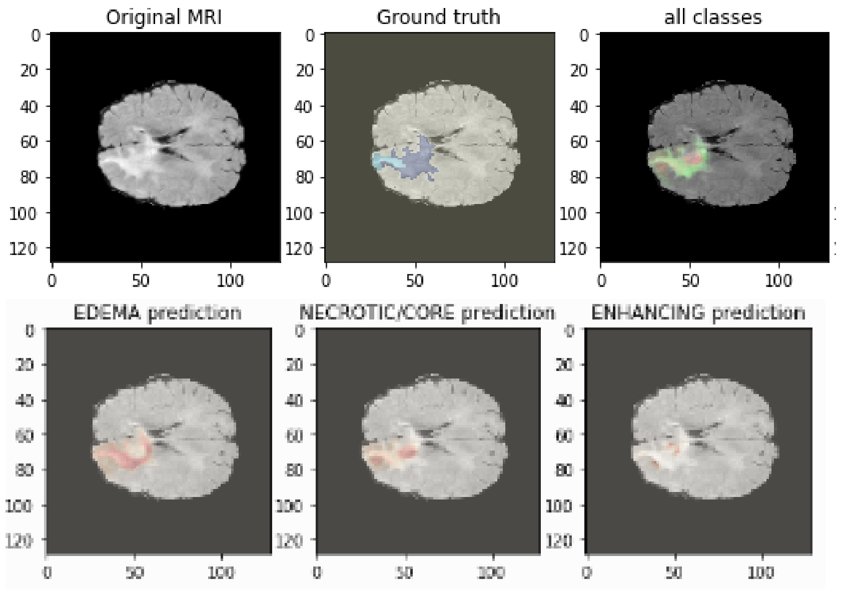
Results

Two deep learning models were programmed, trained, and evaluated: one for the 3D segmentation of brain tumors, and the other for the 3D segmentation of cerebral vasculature. Both of these models were evaluated with several benchmark metrics, specifically designed for medical segmentation, derived from four values: true positives, true negatives, false positives, and false negatives (see Appendix D). To calculate these values, the model prediction, or what the algorithm believes to be the segmentation, is compared with the ground truth, which is the segmentation that is manually done by an expert, and what the prediction should closely resemble.

The brain tumor segmentation model made predictions on three distinct classes: the tumor core/necrotic tissue, edema, and enhancing. Figure 8 shows actual predictions from a test MRI for each class compared with their ground truth values.

Figure 8

Brain Tumor Segmentation Predictions



The original MRI is overlaid on each panel, and then the classes as well as the ground truth are drawn on top of the original MRI.

Figure 9 shows both the ground truth and the prediction converted to a binary array so differences can easily be seen.

Figure 9

Ground Truth and Binary Prediction Comparison for Edema Class

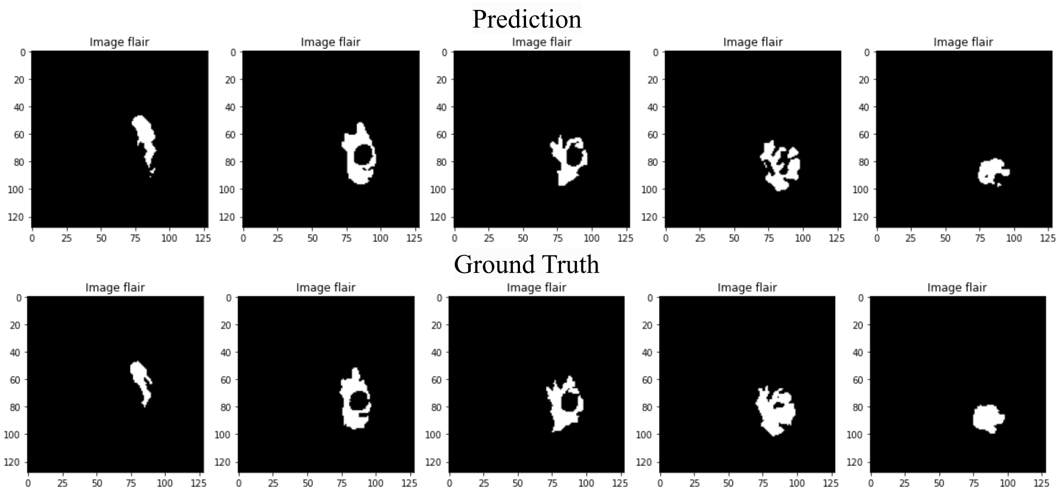


Table 1 communicates the mathematical formulas for the several segmentation metrics used in this project: loss, accuracy, mean Intersection Over Union (IOU), DiCE coefficient, precision, sensitivity, and specificity.

Table 1

Segmentation Metric Formulas

Metric	Formula	Minimized or Maximized
Loss	NA	Minimized
Accuracy	(TP+TN)/(TP+TN+FP+FN)	Maximized
IOU	TP/(TP+FP+FN)	Maximized
Dice coefficient	(2*TP)/(FP+FN+2*FP)	Maximized
Precision	TP/(TP+FP)	Maximized
Sensitivity	TP/(TP+FN)	Maximized
Specificity	TN/(FP+TN)	Maximized

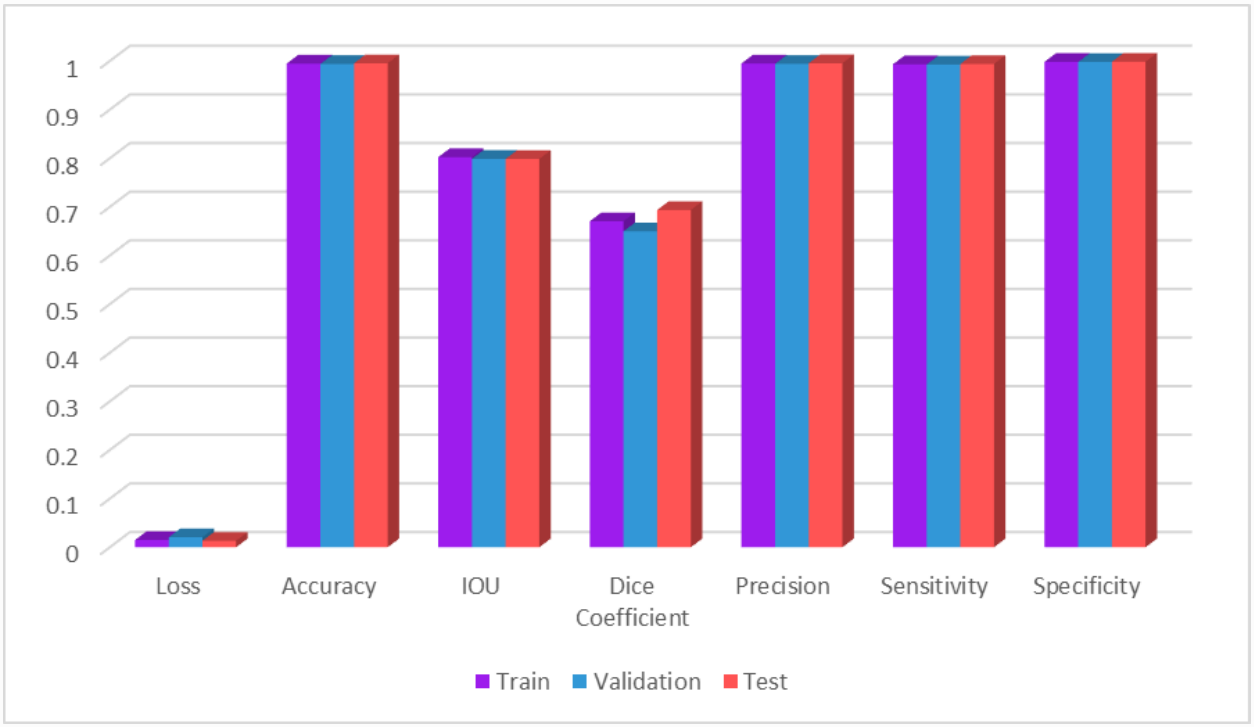
Note. The Loss metric is not computed with true/false positives and negatives, it is computed with an algorithm and used to adjust the learning rate of the model.

Each metric utilizes a different interpretation of true/false positives and negatives to evaluate the model on several fronts. The DiCE coefficient and the mean IOU are considered the best metrics in terms of overall model performance.

Figure 10 depicts the evaluation of the brain tumor segmentation model for the train, validation, and test datasets.

Figure 10

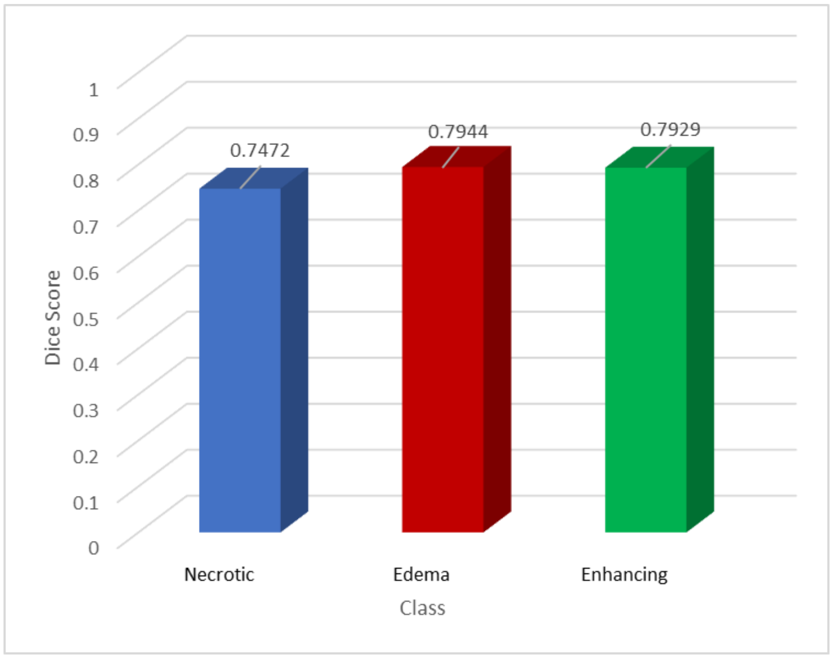
Segmentation Metrics for Train, Validation, and Test Datasets - Brain Tumor Segmentation



To further evaluate the brain tumor segmentation model, the DiCE coefficient for each class was calculated as shown in Figure 11 to discern which classes the model segments best, and which classes the model needs additional training on.

Figure 11

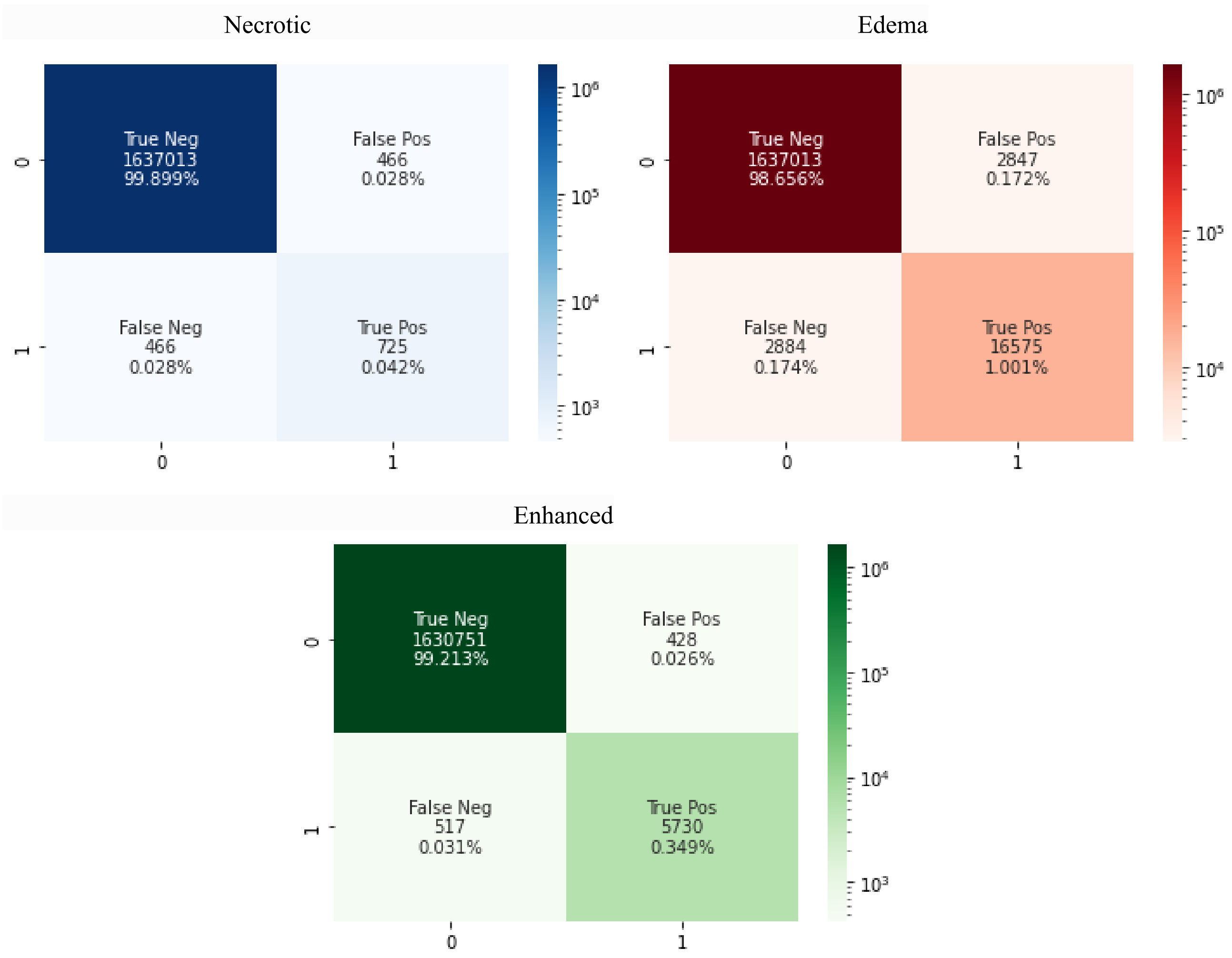
DiCE Coefficient Scores per Class



Looking further into the specific true/false positives and negatives for each class, three confusion matrices were generated, shown in Figure 12.

Figure 12

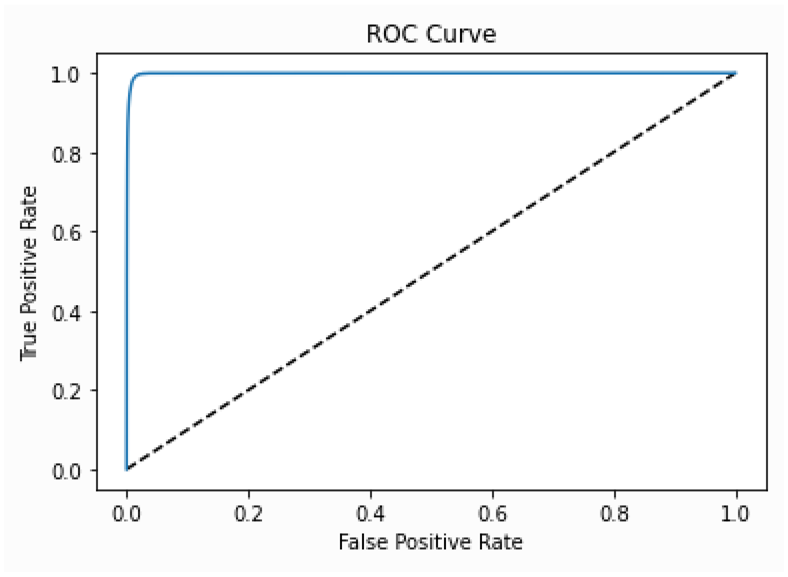
Confusion Matrices for each Class



These confusion matrices show the number of voxels (3D pixel) that fall into each category, as well as the percentage of the entire MRI those voxels make up (see Appendix D). One last test was done on the brain tumor segmentation model, and that was implementing a Receiver Operating Characteristic (ROC) curve for all classes combined, shown in Figure 13 to adjust the threshold at which the prediction assigns a 1 or 0 to each voxel.

Figure 13

ROC Curve

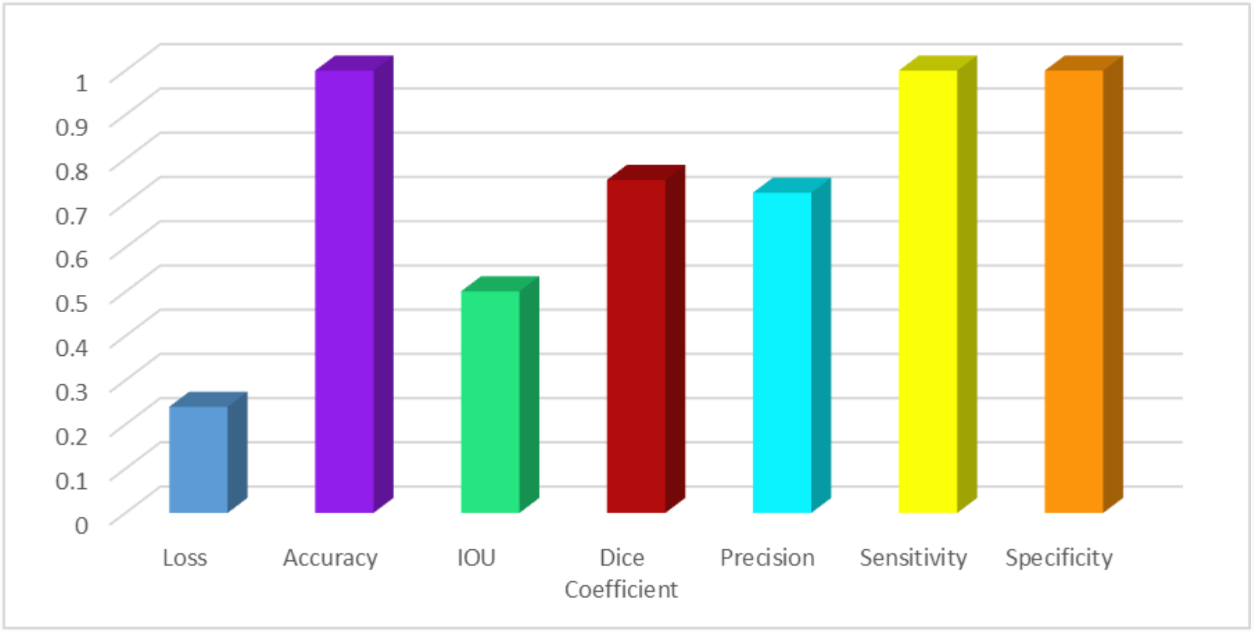


The ROC curve analyzes and depicts the resulting change and tradeoff of true and false positives as the threshold changes. The model's prediction is an array of probabilities, of how likely it thinks a certain voxel is part of a tumor or its surroundings. From this prediction, a threshold value of usually 0.5 is selected, and the prediction is converted into a binary array of 1s and 0s based on that threshold. The lower that threshold, the more true and false positives the model will hold, and the higher the threshold, the fewer true and false positives the model will hold. Implementing the ROC Curve allows for a suitable threshold to be found, one that maximizes the number of true positives without greatly increasing the number of false positives. The dotted diagonal line represents a random classifier, a model that has zero training and makes random predictions. The blue line represents the ROC curve of the brain tumor segmentation model; the closer the line is to the left corner, the more accurate the model is considered.

For the evaluation of the cerebrovascular segmentation model, most of the same metrics were used as shown in Figure 14.

Figure 14

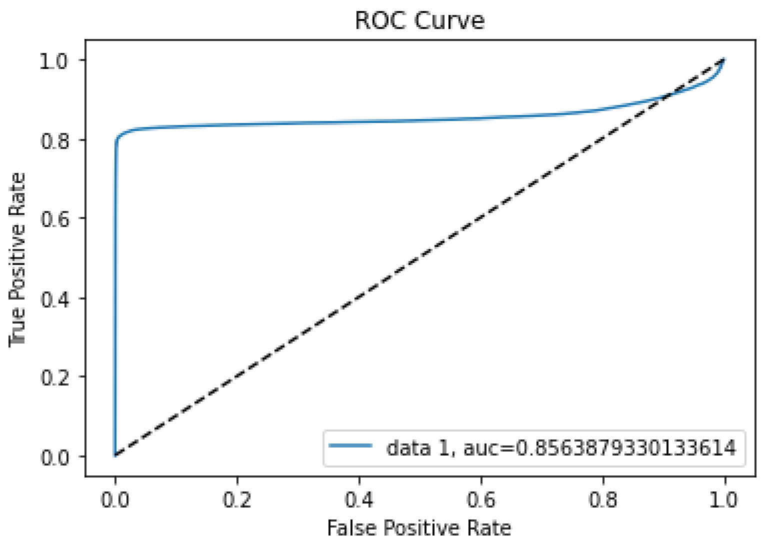
Segmentation Metrics - Cerebrovascular Segmentation



An ROC Curve, shown in Figure 15, was generated based on the sensitivity and specificity metrics, and the Area Under Curve (AUC) was calculated to be 0.856.

Figure 15

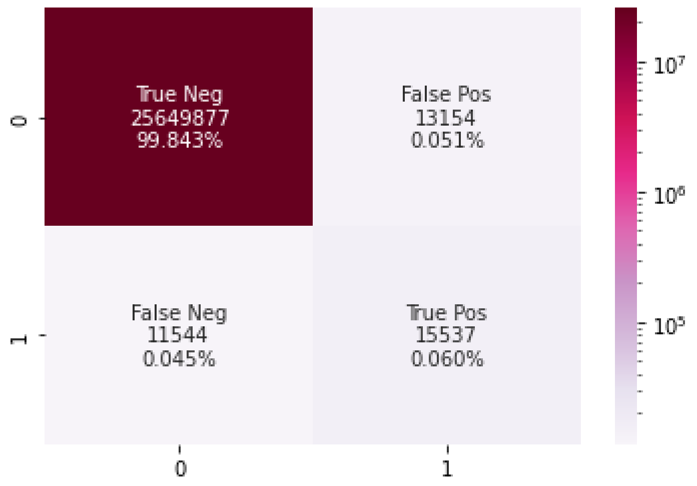
ROC Curve for Cerebrovascular Segmentation Model



Additionally, a confusion matric shown in Figure 16 was plotted, displayed the true/false positives and negatives.

Figure 16

Confusion Matrix for Cerebrovascular Segmentation Model



Figures 17 and 18 shows a sample test prediction compared against the actual vasculature in both slice view and 3D view.

Figure 17

Slices of Cerebrovascular Segmentation Truth and Predictions

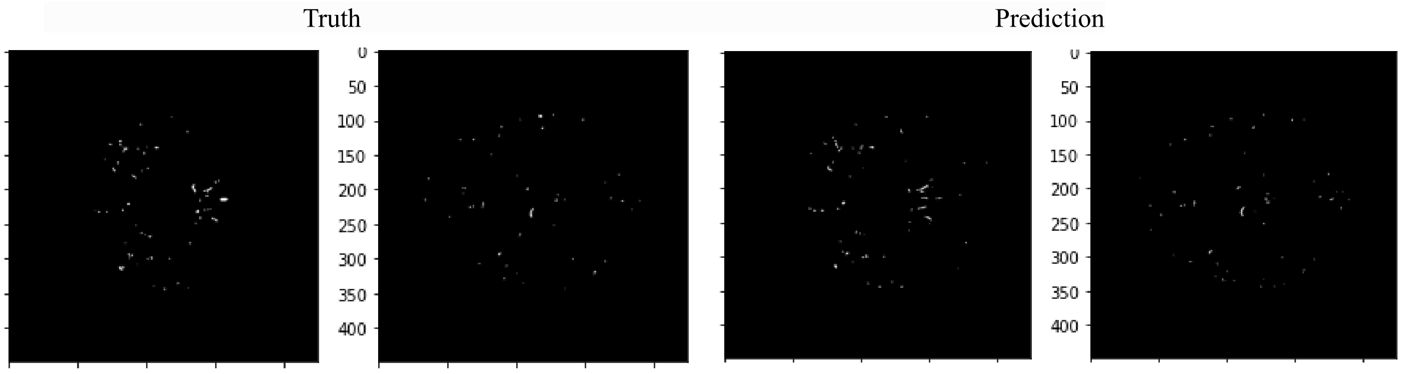
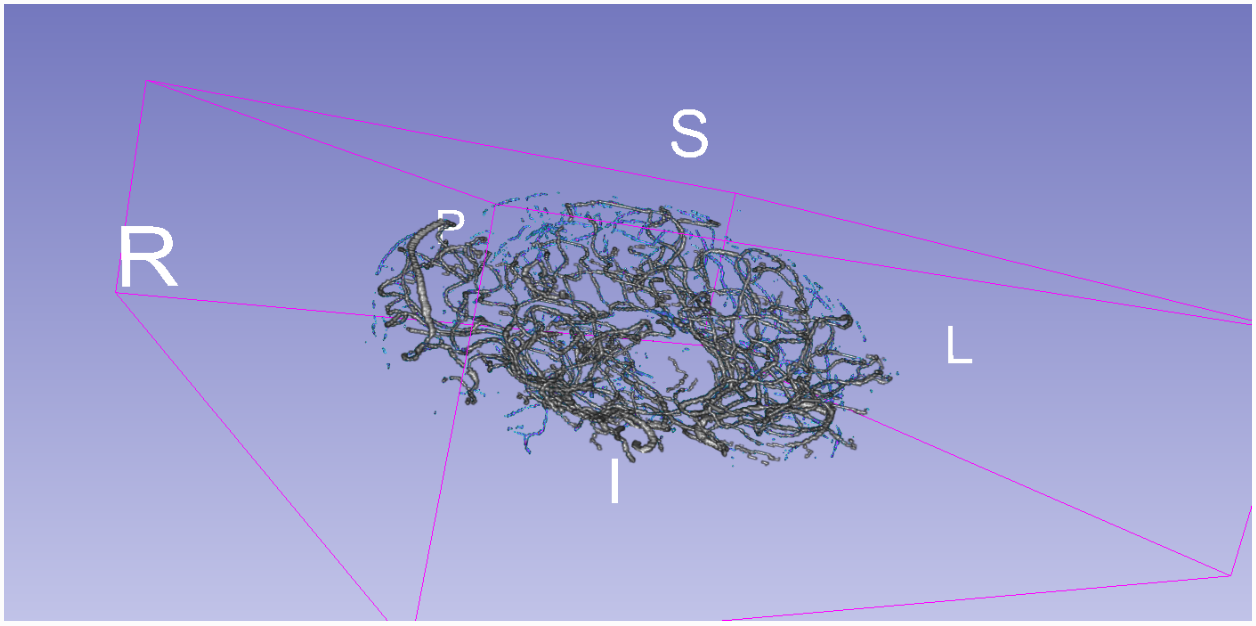


Figure 18

3D Rendering of Cerebral Vasculature Label and Prediction in Slicer



Discussion

The brain tumor segmentation model output similar results in terms of metrics for the train, validation, and test datasets as seen in Figure 10. The dice coefficient score was 0.69 for the brain tumor segmentation model, which implies the model is exceptionally accurate, however, some incorrect predictions are present. Taking into consideration the size of the tumor as well as the application of this project, certain slight inaccuracies, for example, pixels between the class boundaries which is where the model made most of the incorrect predictions, are negligible.

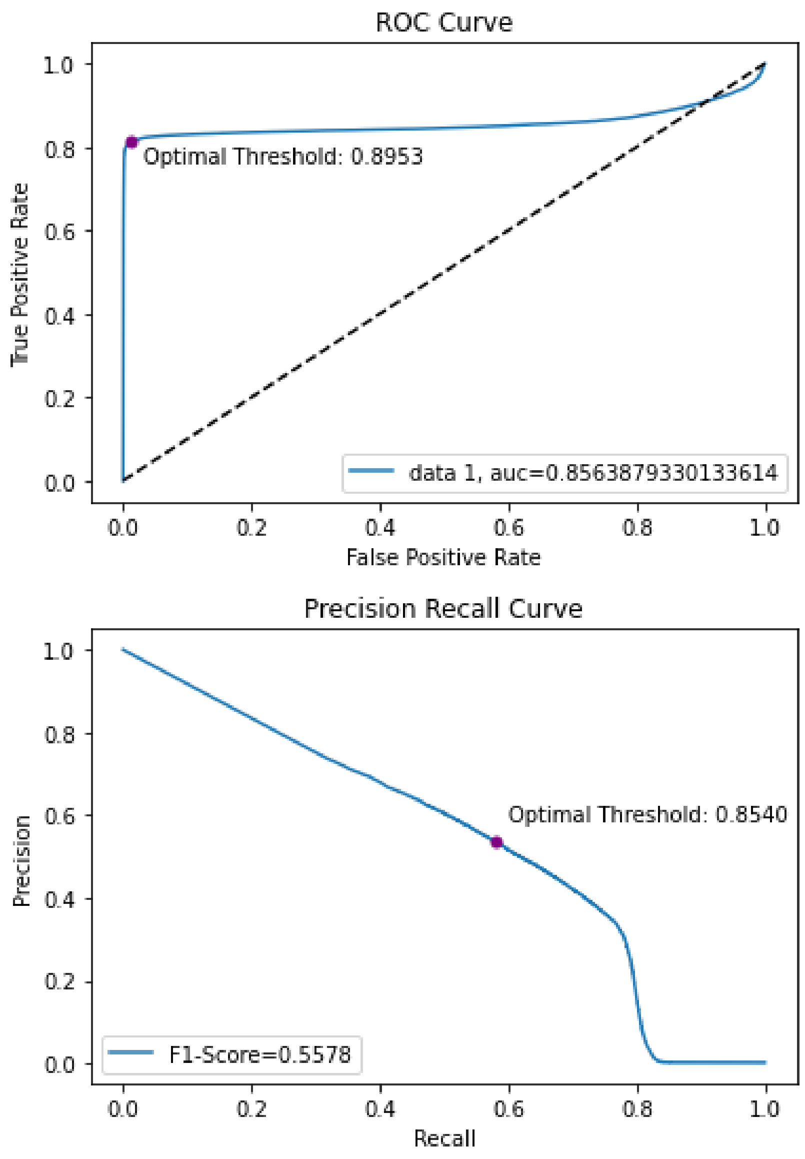
Of the three classes, necrotic, edema, and enhancing, the edema class was segmented the most accurately with a dice score of 0.7944; the enhancing class also had a close dice score of 0.7929 as shown in Figure 11. The necrotic class had a slightly lower dice score of 0.7472 most likely due to unclear boundaries on the MRI between the edema and tumor core. The tumor is the core on the inside, the edema, which is a buildup of fluid, surrounds the tumor, and enhancing refers to the section surrounding the edema that is easily visible in a contrast-enhanced MRI due to the concentration of blood vessels. Slight gradient differences between these classes can lead to some model inaccuracies, but these are usually negligible for the purpose of this project.

As for the confusion matrices in Figure 12, voxels in the true negative category compromise an overwhelming percentage because the tumor is relatively small compared to the entire scale of the MRI. With the heatmap on a logarithmic scale, larger percentages are colored darker, and it can be concluded that both the true negative and positive values are much greater than the false negative and positive values. The brain tumor segmentation model has a near-perfect ROC curve shown in Figure 13, as it maximizes the Area Under the Curve (AUC) signifying an extremely accurate model.

The same metrics were used to evaluate the cerebrovascular segmentation model as shown in Figure 14. The dice coefficient score, calculated from predictions on the validation dataset, was 0.75 signifying a high degree of accuracy in prediction. An ROC Curve shown in Figure 15 was also plotted for the cerebrovascular segmentation model and the AUC was calculated to be 0.856, meaning the prediction correlates well with the ground truth at a certain threshold. Analyzing the shape of the ROC curve, the threshold of 0.5 currently set to produce the binary mask from the prediction is likely to not be the most effective threshold. To find the optimal threshold, the G-mean and Precision Recall Curve were utilized as shown in Figure 19. The G-mean is the square root of recall times precision, and is plotted on the ROC Curve. The Precision-Recall curve is separate graph plotted with recall/sensitivity on the x-axis and precision on the y-axis. The F or F1-score is then calculated and plotted against the curve to find the optimum threshold.

Figure 19

G-Mean and Precision-Recall Thresholds



Calculating the G-mean returned an optimal threshold of 0.8953, while the F-Score returned an optimal threshold of 0.5578. Both thresholds were able to significantly reduce the amount of false positives, thus increasing the overall accuracy of the prediction.

Applications and Next Steps

This project will be continued to further optimize the AR end of the system; implementing locational trackers and devices will greatly enhance the positioning and orientation of the superimposition. Rather than placing the burden solely on the prediction of the AI, actual locational arguments from the camera and several nearby markers will be used to create a 3D rendering of the scene, and from there the exact position and orientation of the models can be determined.

This project has far-reaching and extensive applications, the main purpose being intended for the live and intraoperative use in craniotomies and minimally invasive brain tumor resections. Surgeons can wear the AR headsets to clearly see an adjustable and comprehensive visualization of the position of the tumor, cerebral vasculature, and other data. This removes the need for the surgeon to constantly switch perspectives, between the surgical site/endoscope feed and the separate 2D screens. Additionally, the AR system removes the cognitive burden of locating the tumor through multiple perspectives, rather, it shows its location directly in the surgeon's field of view. The surgical aid can easily be adjusted to aid in other tumor removal surgeries, such as for pancreatic and colon cancers. This novel AR surgical aid can also be used for preoperative planning, as rendering the tumor, vasculature, etc. in 3D will allow for more effective planning and collaboration. A third important application of this surgical aid and AR in general is its use in medical training, so medical students can visualize how the surgery occurs through a more interactive experience. Lastly, visualizing the tumor in 3D can allow for better communication between doctors and patients, so patients can gain a better understanding of their situation.

References

Best, J. (2018). *Augmented reality in the operating theater: How surgeons are using Microsoft's HoloLens to make operations better*. ZDNET. https://www.zdnet.com/article/augmented-reality-in-the-operating-theatre-how-surgeons-are-using-microsofts-hololens-to-make/

Brain Tumor: Statistics. (2022). Cancer.Net. https://www.cancer.net/cancer-types/brain-tumor/statistics

Jasenovcova, L. (2022). *What is augmented reality and how does AR work*. Resco. https://www.resco.net/blog/what-is-augmented-reality-and-how-does-ar-work/

Johns Hopkins Performs Its First Augmented Reality Surgeries in Patients. (2021). Johns Hopkins Medicine. https://www.hopkinsmedicine.org/news/articles/johns-hopkins-performs-its-first-augmented-reality-surgeries-in-patients

Jud, L., Fotouhi, J., Andronic, O., Aichmair, A., Osgood, G., Navab, N., & Farshad, M. (2020). Applicability of augmented reality in orthopedic surgery – A systematic review. *BMC Musculoskeletal Disorders, 21*(103). https://bmcmusculoskeletdisord.biomedcentral.com/articles/10.1186/s12891-020-3110-2

Le, J. (2021). *How to do Semantic Segmentation using Deep Learning*. Nanonets. https://nanonets.com/blog/how-to-do-semantic-segmentation-using-deep-learning/

Meola, A., Cutolo, F., Carbone, M., Cagnazzo, F., Ferrari, M., & Ferrari, V. (2017). Augmented Reality in Neurosurgery: A Systematic Review. *Neurosurgical Review, 40*(4), 537-548. https://doi.org/10.1007/s10143-016-0732-9

Minimally Invasive Brain Tumor Surgery. (2022). Pacific Neuroscience Institute. https://www.pacificneuroscienceinstitute.org/brain-tumor/treatment/minimally-invasive-brain-surgery/#tab-gravity-assisted

Salehahmadi, F., & Hajialiasgari, F. (2019). Grand Adventure of Augmented Reality in Landscape of Surgery. *World Journal of Plastic Surgery, 8*(2). https://doi.org/10.29252/wjps.8.2.135

Siegel, R., L., Miller, K., D., Fuchs, H., E., & Jemal, A. (2021). Cancer Statistics 2021. *Cancer Journal for Clinicians, 71*(1), 7-33. https://doi.org/10.3322/caac.21654

Surgery for Brain Tumours. (2019). Cancer Research UK. https://www.cancerresearchuk.org/about-cancer/brain-tumours/treatment/surgery/remove-brain-tumour

What is Deep Learning? (n.d.). Mathworks. Retrieved September 26, 2022 from https://www.mathworks.com/discovery/deep-learning.html

What is SLAM (Simultaneous Localization and Mapping)? (n.d.). Geoslam. Retrieved September 26, 2022 from https://geoslam/us/what-is-slam/

Appendix A

Current Neuronavigation Systems

Neuronavigation is the concept of using technology within surgery to navigate through patient anatomy. There are several utilities for neuronavigation, the most basic being the use of preoperative scans. Surgeons will utilize preoperative scans such as Magnetic Resonance Imaging and Computed Tomography (CT), displayed on a separate screen to localize the tumor. In addition to this, some surgeries will utilize intraoperative scans, or scans taken during surgery. In some situations, surgeons will use 5-aminolevulinic acid (5-ALA), which concetrates in the cancerous cells of the tumor. This allows surgeons to see additional parts of the tumor that appear similar to healthy tissue in the brain.

This live surgical aid is a form of neuronavigation that aids the surgeon by visualizing the tumor as well as other data in 3D, essentially replicating what 5-ALA does, but with a more interactive and customizable methodology.

For craniotomies, the live surgical aid will be implemented through a head mounted display, as usually the surgeon is not utilizing other visual equipment. For minimally invasive surgeries, however, surgeons utilize microscopes and endoscopes to see the perspective of the probe in the small orofice in the brain. The live surgical will have to be slightly adjusted to perform superimposition upon the camera feed by using locational trackers, rather than the prediction of a deep learning model.

Appendix B

Deep Learning and Augmented Reality

Deep learning uses a complex neural network, a series of interconnected layers that function similar to the human brain, to learn complex patterns within data, usually an image (“What is Deep Learning”, n.d.). Deep learning involves both an algorithm, the neural network in this case, and data. Deep learning is best suitable for large amounts of data, and algorithms have hundreds of layers as opposed to standard machine learning networks that only contain a couple layers in the neural network (“What is Deep Learning”, n.d.). Deep learning is often applied to computer vision, the task of interpreting an image, due to the large amounts of data in an image. In image processing, the goal is often to identify if there is an object in the photo, locate where the object is, or accurately outline the object in the image. Outlining an object in the image is often used in medical applications, for example outlining a tumor, and is known as segmentation. Segmentation is a resource-intensive process, and requires specialized deep neural networks known as convolutional neural networks (CNN) (Le, 2021). A convolutional neural network is a special type of neural network that alters the image at each layer to make the data more manageable and to better understand the image. The performance of a CNN is dependent upon its architecture as well as the amount of data it is trained on; designing the optimal CNN using the various image convolution operations that exist can be a challenge.

Deep learning is also extensively used in augmented reality; accurately superimposing 3D models into real life requires CNNs that are able to pinpoint where a model should be placed in an image. AR requires three components: sensors, algorithms, and output devices (Jasenovcova, 2022). AR can utilize multiple positional sensors, such as GPS and gyro, as well as LiDAR for more accurate results, but basic superimposition requires only a single camera.

The type of AR being performed determines the amount of sensors needed. Marker-based AR requires only a camera and functions through locating a specific symbol, and then layering an object with reference to that symbol (Jasenovcova, 2022). Marker-based AR is very accurate, but it requires the use of a specific symbol to work, therefore, it is not applicable in all areas. Marker-less superimposition AR improves its accuracy with more sensors. This type of AR superimposes a 3D model or image on an object. The difference between the object and the marker is that the marker is easy to recognize, like a barcode or QR code, while the object could be anything, like a person. All types of AR require an algorithm, the actual artificial intelligence that is determining the location of the markers or objects and then overlaying the model in 3D (Jasenovcova, 2022). This algorithm is called simultaneous localization and mapping (SLAM) ("What is SLAM", n.d.). SLAM is able to generate 3D environments from its surroundings as well as localize itself in that environment through use of only a camera. SLAM allows for the 3D model projected and oriented the right way when displayed on the headset. As for the algorithm behind SLAM, it is composed of fine-tuned CNNs that are able to extrapolate 3D data from a single image through use of depth perception ("What is SLAM", n.d.). The CNN first recognizes the location where the augment must be displayed, this can be a marker akin to something resembling a QR code, or it can be an actual object like a person. After the system has identified that the marker or object is present, it must extract and locate the features from the object; for example, the AR system will first locate the person, and then locate the arms, legs, and head. Now the AR system can accurately position the model over the features, such as superimposing a skeleton on a person. From here, the system repeatedly performs the same detection and positioning of the model, updating the headset display in real time.

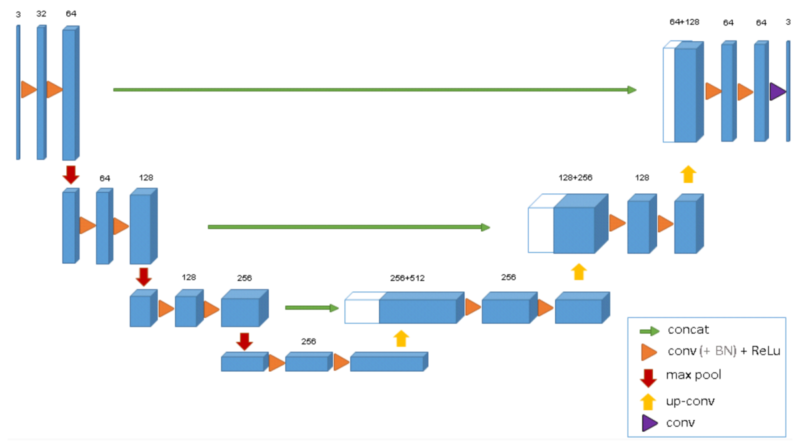
Appendix C

Deep Learning Algorithms

The 3D U-NET is the convolutional neural network (CNN) used for brain tumor segmentation. This algorithm features two parts: an encoder and a decoder. The encoder comprises several layers of convolutions as well as max pooling every two convolutions. The decoder features layers of convolutions and up-convolutions or transpose convolutions every two convolution layers. This algorithm essentially reduces the features of an image through convolutions in the encoder, and then increases the features again in the decoder. The repeated change fo the image through the weights of each layer allows for a segmentation map to be produced from the input image. The 3D U-NET additionally features skip connections to prevent degradation due to the depth of the algorithm. The skip connections transfer the convoluted image to the equally sized layers across the U-structure, essentially skipping a portion of the encoder and decoder. A diagram of the U-NET is shown in Figure C1.

Figure C1

3D U-NET Diagram

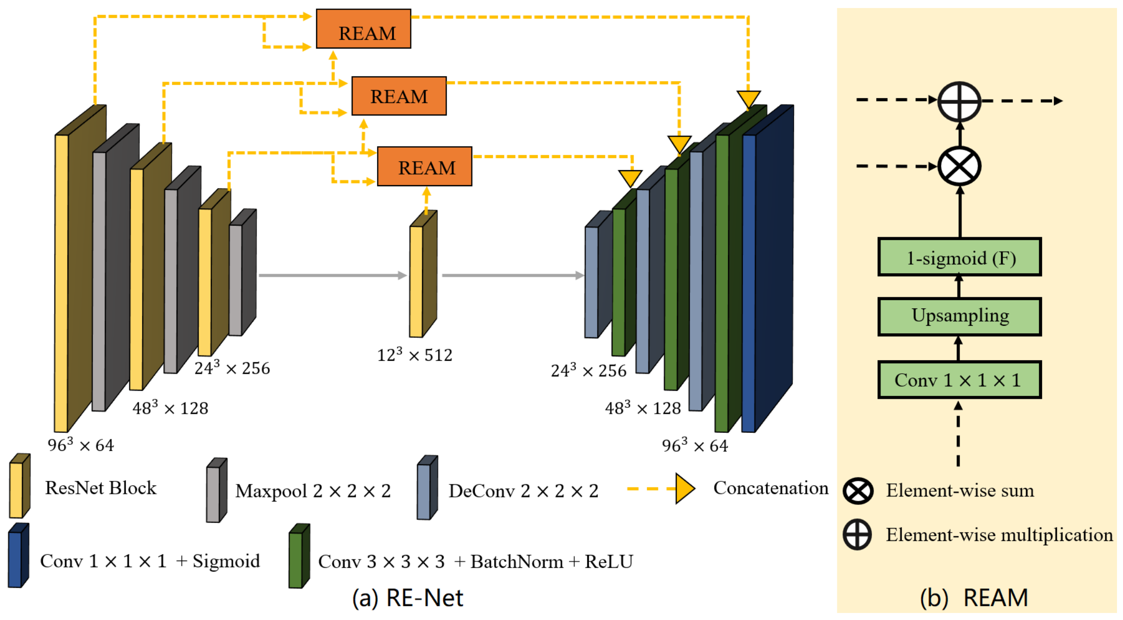


Note. From "Review: 3D U-Net — Volumetric Segmentation," by S. H. Tsang, 2019, *Towards Data Science*.

The Reverse Edge Attention network (RE-Net) was the CNN used for cerebrovascular segmentation. This algorithm functions similar to the U-NET, as it features both an encoder and decoder, however this algorithm is more optimized for the specific application of blood vessel segmentation. The RE-Net also features skip connections, however a special operation known as the Reverse Edge Attention Module (REAM) is embedded in each skip connection to extract segmentation edges from the encoder. Figure C2 depicts a diagram of the RE-Net.

Figure C2

RE-Net Diagram



Note. From “Cerebrovascular Segmentation in MRA via Reverse Edge Attention Network,” by H. Zhang, L. Xia, J. Yang, H. Hao, J. Liu, Y. Zhao, 2020, *MICCAI 2020, 12266*(1), p. 66-75 (https://doi.org/10.1007/978-3-030-59725-2_7).

Appendix D

Segmentation Metrics

Several segmentation metrics were used in the evaluation of both the brain tumor and cerebrovascular segmentation models. These main metrics were accuracy, loss, DiCE Coefficient, mean Intersection Over Union (IOU), sensitivity, specificity, precision, and Area Under Curve (AUC). Each of these metrics except loss is calculated from four numbers: true positives, true negatives, false positives, and false negatives. In the case of brain tumor segmentation, each voxel is assigned a value from 0-3, with zero representing that specific voxel is not a tumor, one representing that the specific voxel is a tumor core, and so on. For the ground truth, each of these voxels is assigned by an expert radiologist, so it is considered the correct label. The AI also assigns each voxel a value from zero through three depending on what it predicts for each voxel. Its prediction is then compared with the ground truth, and true/false positives and negatives are calculated. A true positive refers to one specific voxel of the entire MRI that both the AI and the expert predicted was a tumor, so a value from one through 3. A true negative is a specific voxel that both the AI and expert predicted wasn't a tumor. Both of these are correct predictions. A false positive, however, is when an AI predicts a certain voxel to be a tumor, but it actually isn't. In the same way, a false negative is when the AI predicts a certain voxel to not be a tumor, but it actually is. True/false positives and negatives are then used to calculate the metrics listed above. In general for segmentation, DiCE coefficient and mean IOU are the most commonly used metrics.

In addition to these metrics, a Reciever Operating Characteristic (ROC) curve, as well as a Precision Recall Curve, was plotted to identify optimal thresholds. The AI will return probabilities on the likelihood of each voxel being a positive, and then a threshold value, usually

0.5, is used to convert the prediction to binary. The ROC and Precision Recall curves will identify a threshold other than 0.5 that will minimize false positives and false negatives. This was done for both brain tumor and cerebrovascular segmentation. Lastly, several confusion matrices were plotted using the true/false positives and negatives. Confusion matrices are a way to visually represent the number of true/false positives and negatives, and the color is assigned based on their proportions.

Appendix E

Visualizing Volumes in MATLAB

```
#Assigning both the tumor segmentation and the full MRI to
variables in the MATLAB program
filepath_segmentation = “BraTS_Training_001_seg.nii”
filepath_full = “BraTS_Training_001_flair.nii”
#Loading the data from the file paths, tumorVol represents just
the tumor volume while fullVol represents the entire brain
volume
tumorVol = function niftiread(filepath_segmentation)
fullVol = function niftiread(filepath_full)
#Assign variables values to be passed in as arguments
alpha = 100
color = (255, 0, 0)
#volshow() will display the volumes in an interactive window
volshow(alphamap=alpha,colormap=color,renderer = “Isosurface”)
```

Appendix F

Importing Libraries for Brain Tumor Segmentation

```
import os
import cv2
import glob
import PIL
import numpy as np
import pandas as pd
import seaborn as sb
import shutil
import matplotlib.pyplot as plt
Import skimage
from PIL import Image, ImageOps
import nilearn as nl
import nibabel as nib
import keras
import keras.backend as K
import tensorflow as tf
from tensorflow.keras.utils import plot_model
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from tensorflow.keras.layers.experimental import preprocessing
```

Appendix G

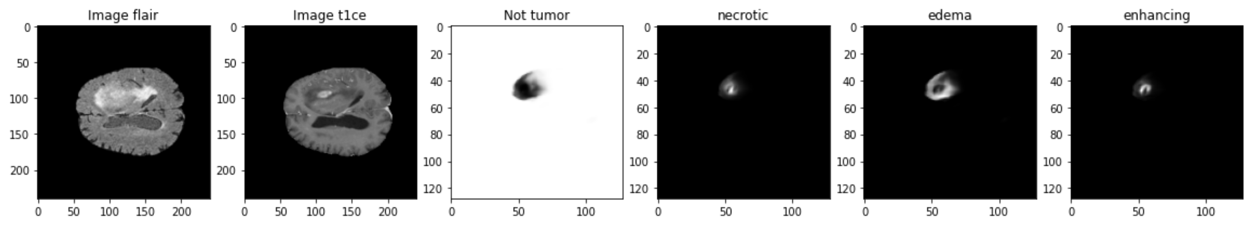
Visualizing Tumor Data in Slices

```
    classes = {
        0 : 'NOT tumor',
        1 : 'NECROTIC/CORE',
        2 : 'EDEMA',
        3 : 'ENHANCING'
    }
    test_image = nib.load(‘filepath’).get_fdata
    fig, (num1) = plt.subplots(1, figsize = (20, 10))
    ax1.imshow(test_image, cmap = 'gray')
nottumor = nib.load('/content/brainpred1.nii').get_fdata()
necrotic = nib.load('/content/brainpred2.nii').get_fdata()
edema = nib.load('/content/brainpred3.nii').get_fdata()
enhancing = nib.load('/content/brainpred4.nii').get_fdata()
ogflair = nib.load('/content/ogflair.nii').get_fdata()
ogce = nib.load('/content/ogce.nii').get_fdata()

#print(necrotic.shape)
#print(ogflair.shape)

fig, (ax1, ax2, ax3, ax4, ax5, ax6) = plt.subplots(1,6, figsize
= (20, 10))
slice_w = 25

ax1.imshow(ogflair[:,:,ogflair.shape[0]//2-slice_w], cmap =
'gray')
```

```
ax1.set_title('Image flair')

ax2.imshow(ogce[:,:,ogce.shape[0]//2-slice_w], cmap = 'gray')
ax2.set_title('Image t1ce')

ax3.imshow(nottumor[nottumor.shape[0]//2-slice_w,:,:], cmap =
'gray')
ax3.set_title('Not tumor')

ax4.imshow(necrotic[necrotic.shape[0]//2-slice_w,:,:], cmap =
'gray')
ax4.set_title('necrotic')

ax5.imshow(edema[edema.shape[0]//2-slice_w,:,:,], cmap = 'gray')
ax5.set_title('edema')

ax6.imshow(enhancing[enhancing.shape[0]//2-slice_w,:,:,], cmap =
'gray')
ax6.set_title('enhancing')
```

Figure G1

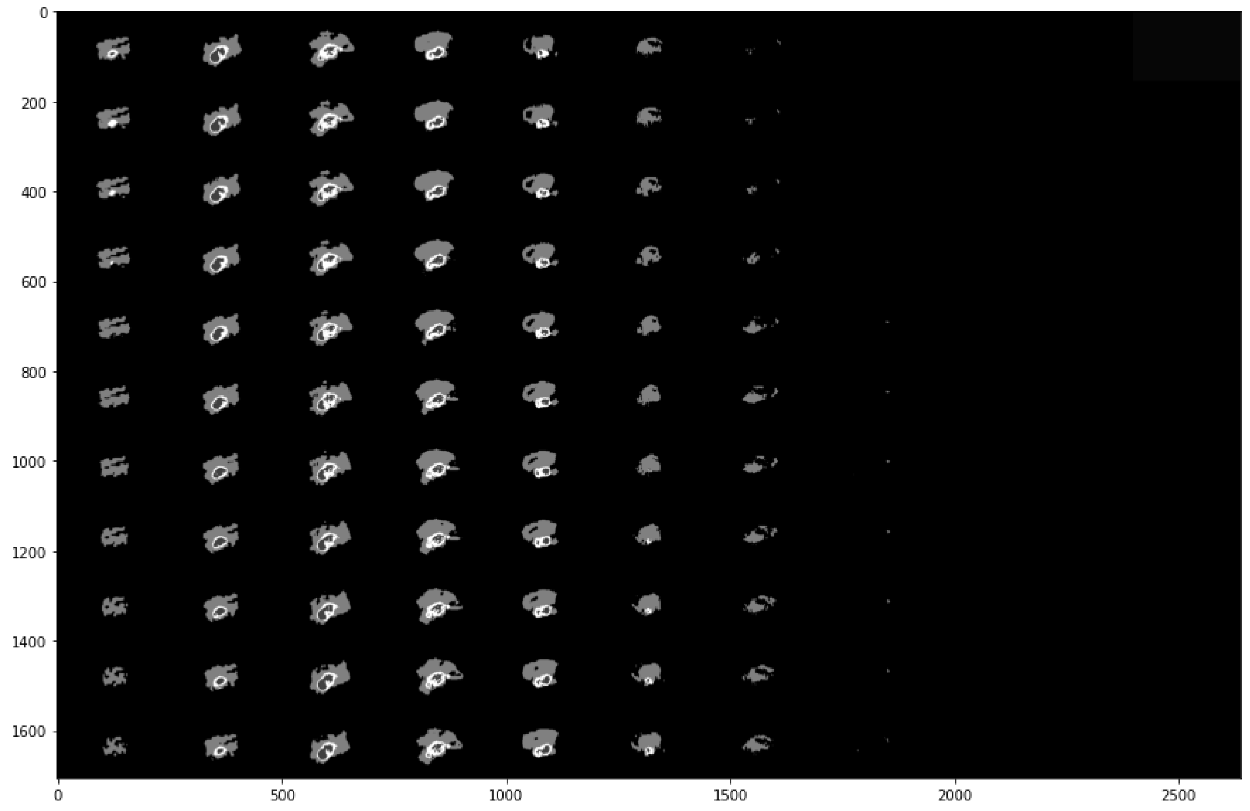
Tumor Label per Class



```
fig, ax1 = plt.subplots(1, 1, figsize = (15,15))
ax1.imshow(rotate(montage(test_mask[60:-60,:,:]), 90,
resize=True), cmap ='gray')
```

Figure G2

All Slices of Label Visualized



Appendix H

Programming the 3D U-NET

```
#defines function that builds the algorithm
#Rectified Linear Unit (ReLU) is the activation function that
allows the algorithm to easily identify correlation
#padding controls what happens to the border of the image when
it is convoluted
def build_unet(inputs, ker_init, dropout):
conv = Conv2D(32, 3, activation = 'relu', padding = 'same',
kernel_initializer = ker_init)(inputs)
conv = Conv2D(32, 3, activation = 'relu', padding = 'same',
kernel_initializer = ker_init)(conv)
pool = MaxPooling2D(pool_size=(2, 2))(conv)

conv2 = Conv2D(64, 3, activation = 'relu', padding = 'same',
kernel_initializer = ker_init)(pool)
conv2 = Conv2D(64, 3, activation = 'relu', padding = 'same',
kernel_initializer = ker_init)(conv2)
pool2 = MaxPooling2D(pool_size=(2, 2))(conv2)

conv3 = Conv2D(128, 3, activation = 'relu', padding = 'same',
kernel_initializer = ker_init)(pool2)
conv3 = Conv2D(128, 3, activation = 'relu', padding = 'same',
kernel_initializer = ker_init)(conv3)
```

```
pool3 = MaxPooling2D(pool_size=(2, 2))(conv3)


conv4 = Conv2D(256, 3, activation = 'relu', padding = 'same',
kernel_initializer = ker_init)(pool3)
conv4 = Conv2D(256, 3, activation = 'relu', padding = 'same',
kernel_initializer = ker_init)(conv4)
pool4 = MaxPooling2D(pool_size=(2, 2))(conv4)
conv5 = Conv2D(512, 3, activation = 'relu', padding = 'same',
kernel_initializer = ker_init)(pool4)
conv5 = Conv2D(512, 3, activation = 'relu', padding = 'same',
kernel_initializer = ker_init)(conv5)
drop5 = Dropout(dropout)(conv5)


up7 = Conv2D(256, 2, activation = 'relu', padding = 'same',
kernel_initializer = ker_init)(UpSampling2D(size =
(2,2))(drop5))
merge7 = concatenate([conv3,up7], axis = 3)
conv7 = Conv2D(256, 3, activation = 'relu', padding = 'same',
kernel_initializer = ker_init)(merge7)
conv7 = Conv2D(256, 3, activation = 'relu', padding = 'same',
kernel_initializer = ker_init)(conv7)
```

```
up8 = Conv2D(128, 2, activation = 'relu', padding = 'same',
kernel_initializer = ker_init)(UpSampling2D(size =
(2,2))(conv7))
merge8 = concatenate([conv2,up8], axis = 3)
conv8 = Conv2D(128, 3, activation = 'relu', padding = 'same',
kernel_initializer = ker_init)(merge8)
conv8 = Conv2D(128, 3, activation = 'relu', padding = 'same',
kernel_initializer = ker_init)(conv8)

up9 = Conv2D(64, 2, activation = 'relu', padding = 'same',
kernel_initializer = ker_init)(UpSampling2D(size =
(2,2))(conv8))
merge9 = concatenate([conv,up9], axis = 3)
conv9 = Conv2D(64, 3, activation = 'relu', padding = 'same',
kernel_initializer = ker_init)(merge9)
conv9 = Conv2D(64, 3, activation = 'relu', padding = 'same',
kernel_initializer = ker_init)(conv9)

up = Conv2D(32, 2, activation = 'relu', padding = 'same',
kernel_initializer = ker_init)(UpSampling2D(size =
(2,2))(conv9))
merge = concatenate([conv1,up], axis = 3)
conv = Conv2D(32, 3, activation = 'relu', padding = 'same',
kernel_initializer = ker_init)(merge)
```

```
conv = Conv2D(32, 3, activation = 'relu', padding = 'same',
kernel_initializer = ker_init)(conv)
conv10 = Conv2D(4, (1,1), activation = 'softmax')(conv)

return Model(inputs = inputs, outputs = conv10)
input_layer = Input((IMG_SIZE, IMG_SIZE, 2))
model = build_unet(input_layer, 'he_normal', 0.2)
```

Appendix I

Dice Coefficient Function

```
def dice_coef(y_true, y_pred, smooth=1.0):
    class_num = 4
    for i in range(class_num):
        y_true_f = K.flatten(y_true[:,:,:,i])
        y_pred_f = K.flatten(y_pred[:,:,:,i])
        intersection = K.sum(y_true_f * y_pred_f)
        loss = ((2. * intersection + smooth) / (K.sum(y_true_f)
    + K.sum(y_pred_f) + smooth))
        if i == 0:
           total_loss = loss
        else:
           total_loss = total_loss + loss
        total_loss = total_loss / class_num
        return total_loss
```

Appendix J

Generating Train, Validation, and Test Datasets

```
train_and_val_directories = [f.path for f in
os.scandir(TRAIN_DATASET_PATH) if f.is_dir()]


def pathListIntoIds(dirList):
     x = []
     for i in range(0,len(dirList)):
         x.append(dirList[i][dirList[i].rfind('/')+1:])
     return x


train_and_test_ids = pathListIntoIds(train_and_val_directories);
train_test_ids, val_ids =
     train_test_split(train_and_test_ids,test_size=0.2)
train_ids, test_ids =
     train_test_split(train_test_ids,test_size=0.15)
```

Appendix K

Generating Keras DataGenerators

```
class DataGenerator(keras.utils.Sequence):
     'Generates data for Keras'
     def __init__(self, list_IDs, dim=(IMG_SIZE,IMG_SIZE),
     batch_size = 1, n_channels = 2, shuffle=True):
             'Initialization'
             self.dim = dim
             self.batch_size = batch_size
             self.list_IDs = list_IDs
             self.n_channels = n_channels
             self.shuffle = shuffle
             self.on_epoch_end()

     def __len__(self):
       return int(np.floor(len(self.list_IDs) /self.batch_size))
     def __getitem__(self, index):
           indexes =
     self.indexes[index*self.batch_size:(index+1)*self.batch_siz
     e]

           Batch_ids = [self.list_IDs[k] for k in indexes]
           X, y = self.__data_generation(Batch_ids)
           return X, y
```

```
    def on_epoch_end(self):
        self.indexes = np.arange(len(self.list_IDs))
        if self.shuffle == True:
            np.random.shuffle(self.indexes)


    def __data_generation(self, Batch_ids):
       X = np.zeros((self.batch_size*VOLUME_SLICES, *self.dim,
self.n_channels))
       y = np.zeros((self.batch_size*VOLUME_SLICES, 240, 240))
       Y = np.zeros((self.batch_size*VOLUME_SLICES, *self.dim,
    4))
       for c, i in enumerate(Batch_ids):
         case_path = os.path.join(TRAIN_DATASET_PATH, i)
         data_path = os.path.join(case_path, f'{i}_flair.nii')
         flair = nib.load(data_path).get_fdata()


         data_path = os.path.join(case_path, f'{i}_t1ce.nii')
         ce = nib.load(data_path).get_fdata()


         data_path = os.path.join(case_path, f'{i}_seg.nii')
         seg = nib.load(data_path).get_fdata()


         for j in range(VOLUME_SLICES):
                X[j +VOLUME_SLICES*c,:,:,0] =
      cv2.resize(flair[:,:,j+VOLUME_START_AT], (IMG_SIZE,
      IMG_SIZE));
                X[j +VOLUME_SLICES*c,:,:,1] =
      cv2.resize(ce[:,:,j+VOLUME_START_AT], (IMG_SIZE,
      IMG_SIZE));
                y[j +VOLUME_SLICES*c] =
      seg[:,:,j+VOLUME_START_AT];


          y[y==4] = 3;
          mask = tf.one_hot(y, 4);
          Y = tf.image.resize(mask, (IMG_SIZE, IMG_SIZE));
          return X/np.max(X), Y
training_generator = DataGenerator(train_ids)
valid_generator = DataGenerator(val_ids)
test_generator = DataGenerator(test_ids)
K.clear_session()
history =  model.fit(training_generator,
                          epochs=35,
                          steps_per_epoch=len(train_ids),
                          callbacks= callbacks,
                          validation_data = valid_generator
                          )
model.save("model.h5")
```

Appendix L

Test Data Helper Functions

```
def imageLoader(path):
    image = nib.load(path).get_fdata()
    X = np.zeros((self.batch_size*VOLUME_SLICES, *self.dim,
self.n_channels))
    for j in range(VOLUME_SLICES):
        X[j +VOLUME_SLICES*c,:,:,0] =
cv2.resize(image[:,:,j+VOLUME_START_AT], (IMG_SIZE, IMG_SIZE));
        X[j +VOLUME_SLICES*c,:,:,1] =
cv2.resize(ce[:,:,j+VOLUME_START_AT], (IMG_SIZE, IMG_SIZE));

        y[j +VOLUME_SLICES*c] = seg[:,:,j+VOLUME_START_AT];
    return np.array(image)


def loadDataFromDir(path, list_of_files, mriType, n_images):
    scans = []
    masks = []
    for i in list_of_files[:n_images]:
        fullPath = glob.glob( i + '/*'+ mriType +'*')[0]
        currentScanVolume = imageLoader(fullPath)
        currentMaskVolume = imageLoader( glob.glob( i +
'/*seg*')[0] )
```

```
        for j in range(0, currentScanVolume.shape[2]):
            scan_img = cv2.resize(currentScanVolume[:,:,j],
dsize=(IMG_SIZE,IMG_SIZE),
interpolation=cv2.INTER_AREA).astype('uint8')
            mask_img = cv2.resize(currentMaskVolume[:,:,j],
dsize=(IMG_SIZE,IMG_SIZE),
interpolation=cv2.INTER_AREA).astype('uint8')
            scans.append(scan_img[..., np.newaxis])
            masks.append(mask_img[..., np.newaxis])
    return np.array(scans, dtype='float32'), np.array(masks,
dtype='float32')
```

Appendix M

Loading and Predicting an Unlabeled MRI

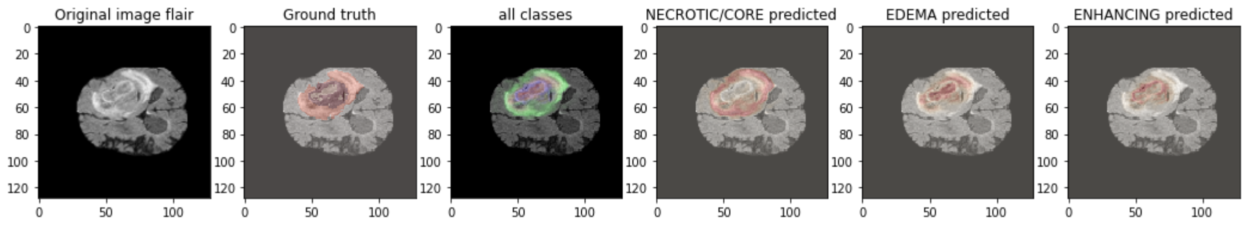
```
def predictByPath(case_path,case):
    files = next(os.walk(case_path))[2]
    X = np.empty((VOLUME_SLICES, IMG_SIZE, IMG_SIZE, 2))

    vol_path = os.path.join(case_path,
f'BraTS20_Training_{case}_flair.nii')
    flair=nib.load(vol_path).get_fdata()

    vol_path = os.path.join(case_path,
f'BraTS20_Training_{case}_t1ce.nii');
    ce=nib.load(vol_path).get_fdata()
    for j in range(VOLUME_SLICES):
        X[j,:,:,0] = cv2.resize(flair[:,:,j+VOLUME_START_AT],
(IMG_SIZE,IMG_SIZE))
        X[j,:,:,1] = cv2.resize(ce[:,:,j+VOLUME_START_AT],
(IMG_SIZE,IMG_SIZE))
    something = model.predict(X/np.max(X), verbose=1)
    ni_img1 = nib.Nifti1Image(something[:,:,:,0],
affine=np.eye(4))
    ni_img2 = nib.Nifti1Image(something[:,:,:,1],
affine=np.eye(4))
```

```
    ni_img3 = nib.Nifti1Image(something[:,:,:,2], 
affine=np.eye(4))
    ni_img4 = nib.Nifti1Image(something[:,:,:,3], 
affine=np.eye(4))
    ogflair = nib.Nifti1Image(flair, affine=np.eye(4))
    ogce = nib.Nifti1Image(ce, affine=np.eye(4))
    nib.save(ni_img1, "brainpred1.nii")
    nib.save(ni_img2, "brainpred2.nii")
    nib.save(ni_img3, "brainpred3.nii")
    nib.save(ni_img4, "brainpred4.nii")
    nib.save(ogflair, "ogflair.nii")
    nib.save(ogce, "ogce.nii")
    return something

def showPredictsById(case, start_slice = 60):
    path = 
f"drive/MyDrive/BRATSData/train/MICCAI_BraTS2020_TrainingData/Br
aTS20_Training_{case}"
    gt = nib.load(os.path.join(path, 
f'BraTS20_Training_{case}_seg.nii')).get_fdata()
    origImage = nib.load(os.path.join(path, 
f'BraTS20_Training_{case}_flair.nii')).get_fdata()
    p = predictByPath(path,case)
    core = p[:,:,:,1]
    edema= p[:,:,:,2]
    enhancing = p[:,:,:,3]
    plt.figure(figsize=(18, 50))
    f, axarr = plt.subplots(1,6, figsize = (18, 50))
    for i in range(6):
          axarr[i].imshow(cv2.resize(origImage[:,:,start_slice+V
          OLUME_START_AT], (IMG_SIZE, IMG_SIZE)), cmap="gray",
          interpolation='none')
     axarr[0].imshow(cv2.resize(origImage[:,:,start_slice+VOLUME
_START_AT], (IMG_SIZE, IMG_SIZE)), cmap="gray")
    axarr[0].title.set_text('Original image flair')
    curr_gt=cv2.resize(gt[:,:,start_slice+VOLUME_START_AT],
(IMG_SIZE, IMG_SIZE), interpolation = cv2.INTER_NEAREST)
    axarr[1].imshow(curr_gt, cmap="Reds", interpolation='none',
alpha=0.3) # ,alpha=0.3,cmap='Reds'
    axarr[1].title.set_text('Ground truth')
    axarr[2].imshow(p[start_slice,:,:,1:4], cmap="Reds",
interpolation='none', alpha=0.3)
    axarr[2].title.set_text('all classes')
    axarr[3].imshow(edema[start_slice,:,:], cmap="OrRd",
interpolation='none', alpha=0.3)
    axarr[3].title.set_text(f'{SEGMENT_CLASSES[1]} predicted')
    axarr[4].imshow(core[start_slice,:,], cmap="OrRd",
interpolation='none', alpha=0.3)
```

```
    axarr[4].title.set_text(f'{SEGMENT_CLASSES[2]} predicted')
    axarr[5].imshow(enhancing[start_slice,:,], cmap="OrRd", 
interpolation='none', alpha=0.3)
    axarr[5].title.set_text(f'{SEGMENT_CLASSES[3]} predicted')
    plt.show()


showPredictsById(case=test_ids[0][-3:])
```

Figure M1

Predictions per Class Compared With Ground Truth



Appendix N

Importing Libraries for Cerebrovascular Segmentation

```
import torch

import os

import sys

import time

import torch

import torch.nn as nn

import numpy as np

import nibabel as nib

from torch.utils.data import Dataset

from torchvision import transforms

from random import randint

from scipy import ndimage


from torch.utils.data import DataLoader

import glob

!pip install SimpleITK

import SimpleITK as sitk

import random

!pip install mxnet

import visdom
```

Appendix O

Initializing Visdom through a Local Tunnel

```
! npm install -g localtunnel
get_ipython().system_raw('/usr/local/bin/python -m pip install
visdom')
get_ipython().system_raw('/usr/local/bin/python -m visdom.server
-port 6006 >> visdomlog.txt 2>&1 &')
get_ipython().system_raw('lt --port 6006 >> url.txt 2>&1 &')
import time
time.sleep(5)
! cat url.txt
import visdom
time.sleep(5)
vis = visdom.Visdom(port='6006')
print(vis)
time.sleep(3)
vis.text('testing')
! cat visdomlog.txt
class AverageMeter(object):

    def __init__(self):
        self.reset()

    def reset(self):
        self.val = 0
        self.avg = 0
        self.sum = 0
        self.count = 0

    def update(self, val, n=1):
        self.val = val
        self.sum += val * n
        self.count += n
        self.avg = self.sum / self.count

class VisdomLinePlotter(object):
    def __init__(self, env_name='main'):
        self.viz = vis
        self.env = env_name
        self.plots = {}
    def plot(self, var_name, split_name, title_name, x, y):
        if var_name not in self.plots:
            self.plots[var_name] =
self.viz.line(X=np.array([x,x]), Y=np.array([y,y]),
env=self.env, opts=dict(
                legend=[split_name],
                title=title_name,
                xlabel='Epochs',
                ylabel=var_name
            ))
        else:
            self.viz.line(X=np.array([x]), Y=np.array([y]),
env=self.env, win=self.plots[var_name], name=split_name, update
= 'append')
global plotter
plotter = VisdomLinePlotter(env_name='main')
```

Appendix P

Programming the RE-NET

```
nonlinearity = partial(F.relu, inplace=True)


def downsample():
    return nn.MaxPool3d(kernel_size=2, stride=2)


def deconv(in_channels, out_channels):
    return nn.ConvTranspose3d(in_channels, out_channels,
kernel_size=2, stride=2)


def initialize_weights(*models):
    for model in models:
        for m in model.modules():
            if isinstance(m, nn.Conv3d) or isinstance(m,
nn.Linear):
                nn.init.kaiming_normal(m.weight)
                if m.bias is not None:
                    m.bias.data.zero_()
            elif isinstance(m, nn.BatchNorm3d):
                m.weight.data.fill_(1)
                m.bias.data.zero_()


class ResEncoder(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(ResEncoder, self).__init__()
        self.conv1 = nn.Conv3d(in_channels, out_channels//2,
kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm3d(out_channels//2)
        self.conv2 = nn.Conv3d(out_channels//2, out_channels,
kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm3d(out_channels)
        self.relu = nn.ReLU(inplace=False)
        self.conv1x1 = nn.Conv3d(in_channels, out_channels,
kernel_size=1)

    def forward(self, x):
        #residual = self.conv1x1(x)
        out = self.relu(self.bn1(self.conv1(x)))
        out = self.relu(self.bn2(self.conv2(out)))
        out = self.conv1x1(x)
        out = self.relu(out)
        return out

class Decoder(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(Decoder, self).__init__()
        self.conv = nn.Sequential(
            nn.Conv3d(in_channels, out_channels, kernel_size=3,
padding=1),
            nn.BatchNorm3d(out_channels),
            nn.ReLU(inplace=True),
            nn.Conv3d(out_channels, out_channels, kernel_size=3,
padding=1),
            nn.BatchNorm3d(out_channels),
            nn.ReLU(inplace=True)
        )
    def forward(self, x):
        out = self.conv(x)
        return out

class RE_Net(nn.Module):
    def __init__(self):

        super(RE_Net, self).__init__()
        self.encoder1 = ResEncoder(1, 32)
        self.encoder2 = ResEncoder(32, 64)
        self.encoder3 = ResEncoder(64, 128)
        self.bridge = ResEncoder(128, 256)

        self.conv1_1 = nn.Conv3d(256, 1, kernel_size=1)
        self.conv2_2 = nn.Conv3d(128, 1, kernel_size=1)
        self.conv3_3 = nn.Conv3d(64, 1, kernel_size=1)


        self.convTrans1 = nn.ConvTranspose3d(1, 1,
kernel_size=2, stride=2)
        self.convTrans2 = nn.ConvTranspose3d(1, 1,
kernel_size=2, stride=2)
        self.convTrans3 = nn.ConvTranspose3d(1, 1,
kernel_size=2, stride=2)


        self.decoder3 = Decoder(256, 128)
        self.decoder2 = Decoder(128, 64)
        self.decoder1 = Decoder(64, 32)
        self.down = downsample()
        self.up3 = deconv(256, 128)
        self.up2 = deconv(128, 64)
        self.up1 = deconv(64, 32)
        self.final = nn.Conv3d(32, 1, kernel_size=1, padding=0)
        initialize_weights(self)

    def forward(self, x):
        enc1 = self.encoder1(x)
        down1 = self.down(enc1)


        enc2 = self.encoder2(down1)
```

```
down2 = self.down(enc2)


con3_3 = self.conv3_3(enc2)

convTrans3 = self.convTrans3(con3_3)

x3 = -1 * (torch.sigmoid(convTrans3)) + 1

x3 = x3.expand(-1, 32, -1, -1, -1).mul(enc1)

x3 = x3 + enc1


enc3 = self.encoder3(down2)

down3 = self.down(enc3)


con2_2 = self.conv2_2(enc3)

convTrans2 = self.convTrans2(con2_2)

x2 = -1 * (torch.sigmoid(convTrans2)) + 1

x2 = x2.expand(-1, 64, -1, -1, -1).mul(enc2)

x2 = x2 + enc2

bridge = self.bridge(down3)

conv1_1 = self.conv1_1(bridge)

convTrans1 = self.convTrans1(conv1_1)

x = -1 * (torch.sigmoid(convTrans1)) + 1

x = x.expand(-1, 128, -1, -1, -1).mul(enc3)

x = x + enc3


up3 = self.up3(bridge)
```

```
        up3 = torch.cat((up3, x), dim=1)
        dec3 = self.decoder3(up3)

        up2 = self.up2(dec3)
        up2 = torch.cat((up2, x2), dim=1)
        dec2 = self.decoder2(up2)

        up1 = self.up1(dec2)
        up1 = torch.cat((up1, x3), dim=1)
        dec1 = self.decoder1(up1)
        final = self.final(dec1)
        final = F.sigmoid(final)
        return final
```

Appendix Q

Random Patch Extraction

```
args = {
    'train_patch_size_x': 96,
    'train_patch_size_y': 96,
    'train_patch_size_z': 96,
}
def standardization_intensity_normalization(dataset, dtype):
    mean = dataset.mean()
    std  = dataset.std()
    return ((dataset - mean) / std).astype(dtype)

def extractPatch(d, patch_size_x, patch_size_y, patch_size_z, x,
y, z):
    patch = d[x - patch_size_x // 2:x + patch_size_x // 2, y -
patch_size_y // 2:y + patch_size_y // 2,
            z - patch_size_z // 2:z + patch_size_z // 2]
    return patch
patchs_size = (args["train_patch_size_x"],
args["train_patch_size_y"], args['train_patch_size_z'])

def load_dataset(root_dir, train=True):
    images = []
    groundtruth = []
    if train:
        sub_dir = 'train'
    else:
        sub_dir = 'test'

    images_path = os.path.join(root_dir, sub_dir, 'train')
    groundtruth_path = os.path.join(root_dir, sub_dir, 'label')

    for file in glob.glob(os.path.join(images_path, '*.nii')):
        images.append(file)
    for file in glob.glob(os.path.join(groundtruth_path,
'*.nii')):
        groundtruth.append(file)
    return images, groundtruth

def RandomPatchCrop(image, label, patch_in_size, patch_gd_size):
    if (patch_in_size[0] % patch_gd_size[0] != 0 or
patch_in_size[1] % patch_gd_size[1] != 0 or patch_in_size[2] %
            patch_gd_size[2] != 0):
        sys.exit("ERROR : randomPatchsAugmented patchs size
error 1")
    if (patch_in_size[0] < patch_gd_size[0] or patch_in_size[1]
< patch_gd_size[1] or patch_in_size[2] <
            patch_gd_size[2]):
```

```
        sys.exit("ERROR : randomPatchsAugmented patchs size
error 2")
    x = randint(patchs_size[0] // 2,
                image.shape[0] - patch_in_size[0] // 2)
    y = randint(patchs_size[1] // 2, image.shape[1] -
patch_in_size[1] // 2)
    z = randint(patchs_size[2] // 2, image.shape[2] -
patch_in_size[2] // 2)
    r0 = randint(0, 3)
    r1 = randint(0, 3)
    r2 = randint(0, 3)
    patchs_in = extractPatch(image, patch_in_size[0],
patch_in_size[1], patch_in_size[2], x, y, z)
    patchs_gd = extractPatch(label, patch_gd_size[0],
patch_gd_size[1], patch_gd_size[2], x, y, z)
    patchs_in = np.rot90(patchs_in, r0, (0, 1))
    patchs_in = np.rot90(patchs_in, r1, (1, 2))
    patchs_in = np.rot90(patchs_in, r2, (2, 0))
    patchs_gd = np.rot90(patchs_gd, r0, (0, 1))
    patchs_gd = np.rot90(patchs_gd, r1, (1, 2))
    patchs_gd = np.rot90(patchs_gd, r2, (2, 0))
    return patchs_in, patchs_gd
class Data(Dataset):
    def __init__(self,
                 root_dir,
                 train=True,
                 rotate=40,
                 flip=True,
                 random_crop=True,
                 scale1=512):
        self.root_dir = root_dir
        self.train = train
        self.rotate = rotate
        self.flip = flip
        self.random_crop = random_crop
        self.transform = transforms.ToTensor()
        self.resize = scale1
        self.images, self.groundtruth =
load_dataset(self.root_dir, self.train)
    def __len__(self):
        return len(self.images)

    def __getitem__(self, idx):
        img_path = self.images[idx]
        gt_path = self.groundtruth[idx]

        image = nib.load(img_path)
        image = image.get_data().astype(np.float32)
        label = nib.load(gt_path)
        label = label.get_data().astype(np.float32)
        ImagePatch, LablePatch = RandomPatchCrop(image, label,
patchs_size,patchs_size)
        ImagePatch =
standardization_intensity_normalization(ImagePatch, 'float32')
        image =
torch.from_numpy(np.ascontiguousarray(ImagePatch)).unsqueeze(0)
        label =
torch.from_numpy(np.ascontiguousarray(LablePatch)).unsqueeze(0)
        return image, label
```

Appendix R

Metric Functions: AUC, ACC, SEN, SPE, IOU, DSC, PRE

```
def metrics_3d(pred, gt):
    pred = np.where(pred > 150, 255.0, 0)
    FP, FN, TP, TN = numeric_score(pred, gt)
    auc=1-0.5*(FP/(FP+TN+1e-10)+FN/(FN+TP+1e-10))
    acc = (TP + TN) / (TP + FP + FN + TN + 1e-10)
    sen = TP / (TP + FN + 1e-10)  # recall sensitivity
    spe = TN / (TN + FP + 1e-10)
    iou = TP / (TP + FN + FP + 1e-10)
    dsc = 2.0 * TP / (TP * 2.0 + FN + FP + 1e-10)
    pre = TP / (TP + FP + 1e-10)

    return auc, acc, sen, spe, iou, dsc, pre
def metrics3d(pred, label, batch_size):

    pred = (pred.data.cpu().numpy() * 255).astype(np.uint8)

    label = (label.data.cpu().numpy() * 255).astype(np.uint8)



    auc, acc, sen, spe, iou, dsc, pre = 0, 0, 0, 0, 0, 0, 0
    for i in range(batch_size):
        img = pred[i, :, :, :]
        gt = label[i, :, :, :]

        AUC, ACC, SEN, SPE, IOU, DSC, PRE = metrics_3d(img, gt)

        auc += AUC
        acc += ACC
        sen += SEN
        spe += SPE
        iou += IOU
        dsc += DSC
        pre += PRE
    return auc, acc, sen, spe, iou, dsc, pre
```

Appendix S

Training the Cerebrovascular Segmentation Model

```
args = {
    'root': '/content/drive/MyDrive/3dBrainTumorSegmentation/',
    'data_path':
'/content/drive/MyDrive/3dBrainTumorSegmentation/cervascseg/',
    'epochs': 4000,
    'lr': 0.0001,
    'snapshot': 100,
    'test_step': 1,
    'ckpt_path':
'/content/drive/MyDrive/3dBrainTumorSegmentation/checkpoints/',
    'batch_size': 4,
}
#Visdom
X, Y = 0, 1.0
x_acc, y_acc = 0, 0
x_sen, y_sen = 0, 0
x_spe, y_spe = 0, 0
x_iou, y_iou = 0, 0
x_dsc, y_dsc = 0, 0
x_pre, y_pre = 0, 0
x_auc, y_auc = 0, 0
```

```
x_testsen, y_testsen = 0.0, 0.0
x_testdsc, y_testdsc = 0.0, 0.0

plotter.plot('Loss', 'train', 'Loss', X, Y)
plotter.plot('ACC', 'train', 'Accuracy', x_acc, y_acc)
plotter.plot('SEN', 'train', 'Sensitivity', x_sen, y_sen)
plotter.plot('SPE', 'train', 'Specificity', x_spe, y_spe)
plotter.plot('IOU', 'train', 'Intersection Over Union', x_iou,
y_iou)
plotter.plot('DSC', 'train', 'Dice Coefficient', x_dsc, y_dsc)
plotter.plot('PRE', 'train', 'Precision', x_pre, y_pre)
plotter.plot('SEN', 'test', 'Sensitivity', x_testsen, y_testsen)
plotter.plot('Loss', 'test', 'Loss', X, Y)
plotter.plot('DSC', 'test', 'Dice Coefficient', x_testdsc,
y_testdsc)
plotter.plot('AUC', 'train', 'Area Under Curve', x_auc, y_auc)

os.environ["CUDA_VISIBLE_DEVICES"] = "0,1"

def save_ckpt(net, iter):
    if not os.path.exists(args['ckpt_path']):
        os.makedirs(args['ckpt_path'])
    torch.save(net, args['ckpt_path'] +
'X-netPatchEnhanced_Dice' + iter + '.pkl')
```

```
    print("{} Saved model to:{}".format("\u2714",
args['ckpt_path']))


def adjust_lr(optimizer, base_lr, iter, max_iter, power=0.9):
    lr = base_lr * (1 - float(iter) / max_iter) ** power
    for param_group in optimizer.param_groups:
        param_group['lr'] = lr


def train():
    net = RE_Net().cuda()
    net = nn.DataParallel(net).cuda()
    optimizer = optim.Adam(net.parameters(), lr=args['lr'],
weight_decay=0.0005)
    print("{}{}{}{}".format(" " * 8, "\u250f", "\u2501" * 61,
"\u2513"))
    print("{}{}{}{}".format(" " * 8, "\u2503", " " * 22 + "
Start Straining " + " " * 22, "\u2503"))
    print("{}{}{}{}".format(" " * 8, "\u2517", "\u2501" * 61,
"\u251b"))


    iters = 1
    best_sen, best_dsc = 0., 0.
    for epoch in range(args['epochs']):
        net.train()
        train_data = Data(args['data_path'], train=True)
        batchs_data = DataLoader(train_data,
batch_size=args['batch_size'], num_workers=8, shuffle=True)
        for idx, batch in enumerate(batchs_data):
            image = batch[0].type(torch.FloatTensor).cuda()
            label = batch[1].cuda()
            optimizer.zero_grad()
            pred = net(image)
            loss = dice_coeff_loss(pred, label)
            loss.backward()
            optimizer.step()
            auc, acc, sen, spe, iou, dsc, pre = metrics3d(pred,
label, pred.shape[0])
            print(
                '{0:d}:{1:d}] \u2501\u2501\u2501
loss:{2:.10f}\tacc:{3:.4f}\tsen:{4:.4f}\tspe:{5:.4f}\tiou:{6:.4f
}\tdsc:{7:.4f}\tpre:{8:.4f}'.format
                (epoch + 1, iters, loss.item(), acc /
pred.shape[0], sen / pred.shape[0], spe / pred.shape[0],iou /
pred.shape[0], dsc / pred.shape[0], pre / pred.shape[0]))
            iters += 1
            #visdom
            X, x_acc, x_sen, x_spe, x_iou, x_dsc, x_pre,x_auc =
iters, iters, iters, iters, iters, iters, iters,iters
```

```
            Y, y_acc, y_sen, y_spe, y_iou, y_dsc, y_pre,y_auc=
loss.item(), acc / pred.shape[0], sen / pred.shape[0], spe / \

pred.shape[0], iou / \

pred.shape[0], dsc / pred.shape[0], pre / pred.shape[0],auc /
pred.shape[0]

            plotter.plot('Loss', 'train', 'Loss', X, Y)
            plotter.plot('ACC', 'train', 'Accuracy', x_acc,
y_acc)
            plotter.plot('SEN', 'train', 'Sensitivity', x_sen,
y_sen)
            plotter.plot('SPE', 'train', 'Specificity', x_spe,
y_spe)
            plotter.plot('IOU', 'train', 'Intersection Over
Union', x_iou, y_iou)
            plotter.plot('DSC', 'train', 'Dice Coefficient',
x_dsc, y_dsc)
            plotter.plot('PRE', 'train', 'Precision', x_pre,
y_pre)
            plotter.plot('AUC', 'train', 'Area Under Curve',
x_auc, y_auc)
```

```
        adjust_lr(optimizer, base_lr=args['lr'], iter=epoch,
max_iter=args['epochs'], power=0.9)

        if (epoch + 1) % args['snapshot'] == 0:
            save_ckpt(net, str(epoch + 1))

        #eval
        if (epoch + 1) % args['test_step'] == 0:
            test_auc,test_acc, test_sen, test_spe, test_iou,
test_dsc, test_pre = model_eval(net, iters)
        if test_sen >= best_sen and (epoch + 1) >= 500:
            save_ckpt(net, "best_SEN")
            best_sen = test_sen
        if test_dsc > best_dsc:
            save_ckpt(net, "best_DSC")
            best_dsc = test_dsc
        print(
            "Average SEN:{0:.4f}, average SPE:{1:.4f},  average
IOU:{2:.4f},average DSC:{3:.4f},average PRE:{4:.4f}".format(
                test_sen, test_spe, test_iou, test_dsc,
test_pre))

def model_eval(net, iters):
    print("{}{}{}{}".format(" " * 8, "\u250f", "\u2501" * 61,
"\u2513"))
    print("{}{}{}{}".format(" " * 8, "\u2503", " " * 23 + "
Start Testing " + " " * 23, "\u2503"))
    print("{}{}{}{}".format(" " * 8, "\u2517", "\u2501" * 61,
"\u251b"))
    test_data = Data(args['data_path'],train = False)
    batchs_data = DataLoader(test_data, batch_size=1)

    net.eval()
    AUC ,ACC, SEN, SPE, IOU, DSC, PRE = [], [], [], [], [],
[],[]
    file_num = 0
    for idx, batch in enumerate(batchs_data):
        image = batch[0].float().cuda()
        label = batch[1].cuda()
        pred_val = net(image)

        loss2 = dice_coeff_loss(pred_val, label)
        print("Before metrics")
        auc, acc, sen, spe, iou, dsc, pre = metrics3d(pred_val,
label, pred_val.shape[0])
```

```
        print("--- test ACC:{0:.4f} test SEN:{1:.4f} test 
SPE:{2:.4f} test IOU:{3:.4f} test DSC:{4:.4f} test PRE:{5:.4f} 
test AUC:{6:.4f}".format(acc, sen, spe, iou, dsc, pre, auc))
        ACC.append(acc)
        SEN.append(sen)
        SPE.append(spe)
        IOU.append(iou)
        DSC.append(dsc)
        PRE.append(pre)
        AUC.append(auc)
        file_num += 1
        print(acc)
        print(ACC)



        X,x_testsen, x_testdsc = iters,iters, iters
        Y,y_testsen, y_testdsc = loss2.item(),sen / 
pred_val.shape[0], dsc / pred_val.shape[0]



        plotter.plot('SEN', 'test', 'Sensitivity', x_testsen, 
y_testsen)
        plotter.plot('Loss', 'test', 'Loss', X, Y)
        plotter.plot('DSC', 'test', 'Dice Coefficient',
x_testdsc, y_testdsc)


        return np.mean(AUC),np.mean(ACC), np.mean(SEN),
np.mean(SPE), np.mean(IOU), np.mean(DSC), np.mean(PRE)

train()
```

Appendix T

Model Evaluation and Generating Predictions

```
os.environ['CUDA_VISIBLE_DEVICES'] = "0"
args = {
    'test_path':
'/content/drive/MyDrive/3dBrainTumorSegmentation/cervascsegtest/
test/',
    'pred_path':
'/content/drive/MyDrive/3dBrainTumorSegmentation/cervascsegtest/
predict/'
}
if not os.path.exists(args['pred_path']):
    os.makedirs(args['pred_path'])
def load_3d():
    test_images = []
    test_labels = []
    for file in glob.glob(os.path.join(args['test_path'],
'image', '*.nii')):
        basename = os.path.basename(file)
        file_name = basename[:-8]
        image_name = os.path.join(args['test_path'], 'image',
basename)
        label_name = os.path.join(args['test_path'], 'label',
'label'+basename[-7:-4]+'.mha')
```

```
        test_images.append(image_name)
        test_labels.append(label_name)
    return test_images, test_labels
def load_net():
    net =
torch.load('/content/drive/MyDrive/3dBrainTumorSegmentation/chec
kpoints/best.pkl')
    return net
def save_prediction(pred, gt, filename='', spacing=None):
    e = torch.gt(pred, 0.1)
    count = torch.sum(e)
    print(count)
    print(pred)
    mean = torch.mean(pred)
    print(mean)
    threshold = 0.00001

    binary_tensor = torch.where(pred > threshold,
torch.tensor(1.0).cuda(), torch.tensor(0.0).cuda())

    print(binary_tensor)
    e = torch.gt(binary_tensor, 0.05)
    count = torch.sum(e)
    print(count)
    print(binary_tensor.shape)
    e = torch.gt(pred, 0)
    count = torch.sum(e)
    print(count)

    save_path = args['pred_path'] + 'pred/'
    if not os.path.exists(save_path):
        os.makedirs(save_path)
        print("Make dirs success!")

    mask = (binary_tensor.data.cpu().numpy()).astype(np.uint8)
    print(mask.shape)
    mask = mask.squeeze(0).squeeze(0)
    print(mask.shape)
    count = 0

    print(mask.shape)
    image = nib.Nifti1Image(mask, np.eye(4))
    nib.save(image,
'/content/drive/MyDrive/3dBrainTumorSegmentation/cervascsegtest/
predict/pred/pred.nii')

def save_label(label, index, spacing=None):
    label_path = args['pred_path'] + 'label/'
    if not os.path.exists(label_path):
        os.makedirs(label_path)
    label = sitk.GetImageFromArray(label)
    if spacing is not None:
        label.SetSpacing(spacing)
    sitk.WriteImage(label, os.path.join(label_path, index +
".mha"))
def predict():
    net = load_net()
    images, labels = load_3d()
    with torch.no_grad():
        net.eval()
        for i in tqdm(range(len(images))):
            name_list = images[i].split('/')
            index = name_list[-1][:-4]
            image = sitk.ReadImage(images[i])

            image = nib.load(images[i])
            image = image.get_data().astype(np.float32)
            print(image.shape)
            image =
standardization_intensity_normalization(image, 'float32')
            print(image.shape)
            label = sitk.ReadImage(labels[i])
            label =
sitk.GetArrayFromImage(label).astype(np.int64)
            save_label(label, index)
            image = np.ascontiguousarray(image)
            print(image.shape)
            image =
torch.from_numpy(image).unsqueeze(0).unsqueeze(0)
            print(image.shape)
            image = image.cuda()
            with torch.no_grad():
              output = net(image)

            save_prediction(output, label, filename=index +
'_pred', spacing=None)

predict()
```