BRAIN TUMOR AUTO-SEGMENTATION FOR MAGNETIC RESONANCE IMAGING (MRI)

CSSM 502: ADVANCED DATA ANALYSIS PYTHON PROJECT

Shayan Rahimi Shahmirzadi,

Omer Gokalp Ulku,

Reza Sayyari

In this project we will try to build a neural network to automatically segment tumor regions in the brain, using MRI (Magnetic Resonance Imaging) scans.

Here we will be focusing on MRIs. We will walk through some of the steps of training a deep learning model for segmentation.

The MRI scan is one of the most common image modalities that we encounter in the radiology field.

Other data modalities include:

- Computer Tomography (CT
- Ultrasound
- X-Rays

What is an MRI scan?

Magnetic resonance imaging (MRI) is a type of scan that uses strong magnetic fields and radio waves to produce detailed images of the inside of the body.

An MRI scanner is a large tube that contains powerful magnets. You lie inside the tube during the scan.

An MRI scan can be used to examine almost any part of the body, including the:

- brain and spinal cord
- bones and joints
- breasts
- heart and blood vessels
- internal organs, such as the liver, womb, or prostate gland

The results of an MRI scan can be used to help diagnose conditions, plan treatments, and assess how effective previous treatment has been.

How does an MRI scan work?

Most of the human body is made up of water molecules, which consist of hydrogen and oxygen atoms.

At the center of each hydrogen atom is an even smaller particle called a proton. Protons are like tiny magnets and are very sensitive to magnetic fields.

When you lie under the powerful scanner magnets, the protons in your body line up in the same direction, in the same way that a magnet can pull the needle of a compass.

Short bursts of radio waves are then sent to certain areas of the body, knocking the protons out of alignment.

When the radio waves are turned off, the protons realign. This sends out radio signals, which are picked up by receivers.

These signals provide information about the exact location of the protons in the body.

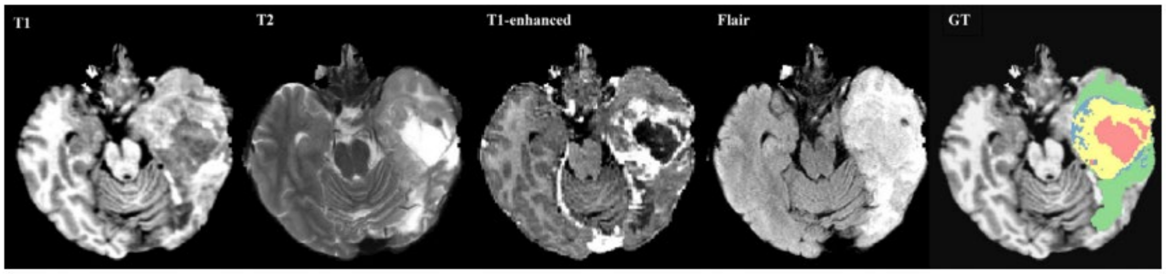
They also help to distinguish between the various types of tissue in the body, because the protons in different types of tissue realign at different speeds and produce distinct signals.

In the same way that millions of pixels on a computer screen can create complex pictures, the signals from the millions of protons in the body are combined to create a detailed image of the inside of the body.

1. Dataset

In this project, we will build a multi-class segmentation model by identifying 3 different abnormalities in each image: edemas, non-enhancing tumors, and enhancing tumors.

The images below can be analyzed with neural networks individually and be combined into a single 3D volume to make predictions.



1.2 MRI Data Processing

MR images are mostly in DICOM format.

- The DICOM format is the output format for most commercial MRI scanners. This type of data can be processed using the pydicom Python library.

We will be using the data from the Decathlon 10 Challenge. This data has been mostly pre-processed for the competition participants, however in real practice, MRI data needs to be significantly pre-preprocessed before we can use it to train our models.

1.3 Exploring the Dataset

Our dataset is stored in the NifTI-1 Format and we will be using the NiBabel library to interact with the files, we have access to a total of 484 training images which we will be splitting into training (80%) and validation (20%) datasets.

1.4 Data Preprocessing using patches

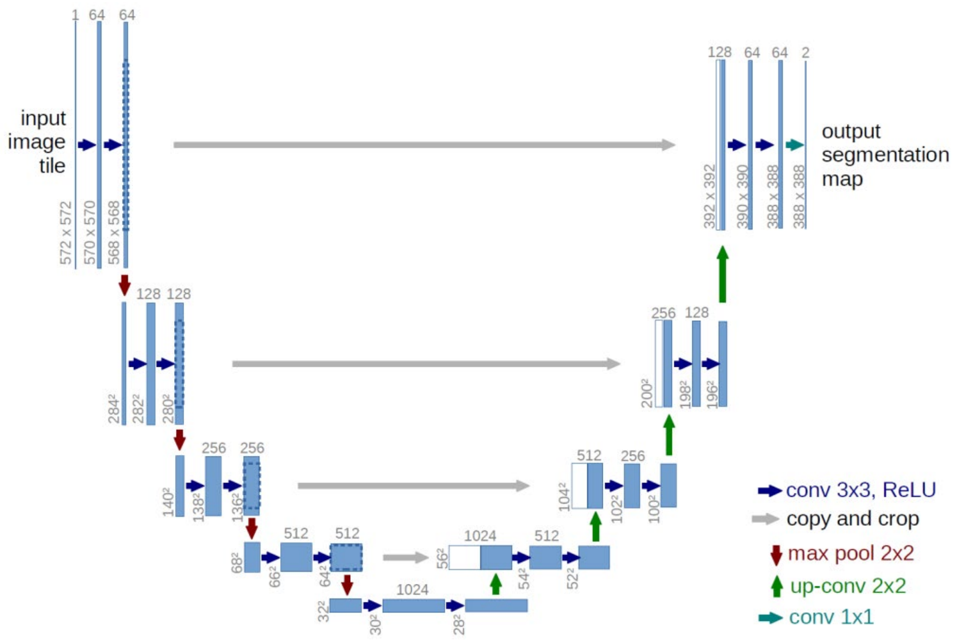
Although we have the dataset provided, we still have to do some minor pre-processing before feeding the data to our model.

We are going to first generate "patches" of our data which you can think of as sub-volumes of the whole MR images and since the values in MR images cover a very wide range, we will standardize the values to have a mean of zero and standard deviation of 1.

2 Model: 3D U-Net

Now we will be building a 3D U-net as our model.

- This architecture will take advantage of the volumetric shape of MR images and is one of the best performing models for this task.



3.1 Dice Similarity Coefficient

Aside from the architecture, one of the most important elements of any deep learning method is the choice of our loss function.

A natural choice that you may be familiar with is the cross-entropy loss function.

- However, this loss function is not ideal for segmentation tasks due to heavy class imbalance (there are typically not many positive regions).

A much more common loss for segmentation tasks is the Dice similarity coefficient, which is a measure of how well two contours overlap.

- The Dice index ranges from 0 (complete mismatch)
- To 1 (perfect match).

In general, for two sets A and B, the Dice similarity coefficient is defined as:

$$\mathrm{DSC}(A,B) = \frac{2 \times |A \cap B|}{|A| + |B|}.$$

Here we can interpret A and B as sets of voxels, A being the predicted tumor region and B being the ground truth.

Our model will map each voxel to 0 or 1

- 0 means it is a background voxel
- 1 means it is part of the segmented region.

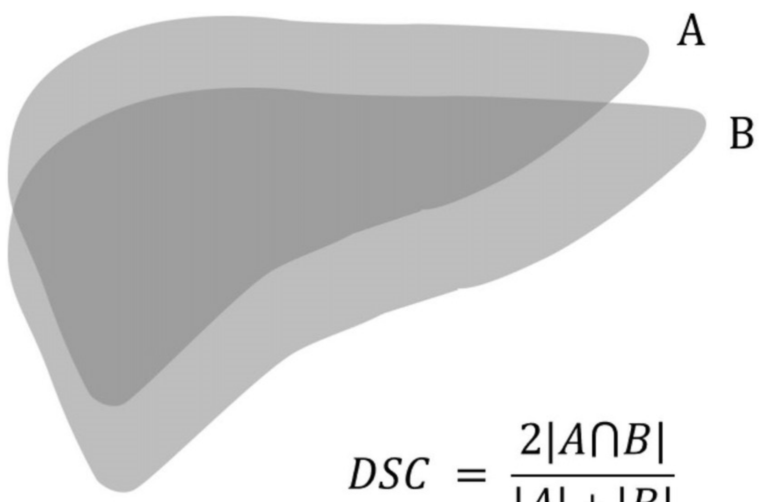
In the dice coefficient, the variables in the formula are:

- x : the input image
- $f(x)$: the model output (prediction)
- y : the label (actual ground truth)

The dice coefficient "DSC" is:

$$\mathrm{DSC}(f, x, y) = \frac{2 \times \sum i, j f(x) ij \times yij + \epsilon}{\sum i, j f(x) ij + \sum i, j y_{ij} + \epsilon}$$

- ϵ is a small number that is added to avoid division by zero



$$DSC = \frac{2|A \cap B|}{|A| + |B|}$$

DSC: Dice similarity coefficient

Dice Coefficient for Multiple classes

Now that we have the single class case, we can think about how to approach the multi class context.

- For this task, we want segmentations for each of the 3 classes of abnormality (edema, enhancing tumor, non-enhancing tumor).
- This will give us 3 different dice coefficients (one for each abnormality class).
- To combine these, we can just take the average. We can write that the overall dice coefficient is:

$$DC(f,x,y) = \frac{1}{3}\left(DC_1(f,x,y) + DC_2(f,x,y) + DC_3(f,x,y)\right)$$

- *DC1*, *DC2* and *DC3* are edema, enhancing tumor, and non-enhancing tumor dice coefficients.

$$DC(f,x,y) = \frac{1}{N}\sum_{c=1}^{C}\left(DC_c(f,x,y)\right)$$

In this case, with three categories, *C=3*

3.2 Soft Dice Loss

While the Dice Coefficient makes intuitive sense, it is not the best for training.

- This is because it takes in discrete values (zeros and ones).
- The model outputs *probabilities* that each pixel is, say, a tumor or not, and we want to be able to backpropagate through those outputs.

Therefore, we need an analogue of the Dice loss which takes real valued input. This is where the **Soft Dice loss** comes in. The formula is:

$$\mathcal{L}_{Dice}(p,q) = 1 - \frac{2 \times \sum_{i,j} p_{ij} q_{ij} + \epsilon}{\left(\sum_{i,j} p_{ij}^2\right) + \left(\sum_{i,j} q_{ij}^2\right) + \epsilon}$$

- p is our predictions
- q is the ground truth
- In practice each q_i will either be 0 or 1.
- ϵ is a small number that is added to avoid division by zero

The soft Dice loss ranges between

- 0: perfectly matching the ground truth distribution q
- 1: complete mismatch with the ground truth.

You can also check that if p_i and q_i are each 0 or 1, then the soft Dice loss is just one minus the dice coefficient.

Multi-Class Soft Dice Loss

We have explained the single class case for simplicity, but the multi-class generalization is exactly the same as that of the dice coefficient.

- Since we have already implemented the multi-class dice coefficient, we will jump directly to the multi-class soft dice loss.

For any number of categories of diseases, the expression becomes:

$$\mathcal{L}_{Dice}(p,q) = 1 - \frac{1}{N} \sum_{c=1}^{C} \frac{2 \times \sum_{i,j} p_{cij} q_{cij} + \epsilon}{\left(\sum_{i,j} p_{cij}^2\right) + \left(\sum_{i,j} q_{cij}^2\right) + \epsilon}$$

4 Create and Train the model

Once we have finished implementing the soft dice loss, we can create the model!

- We'll use the `unet_model_3d` function in `utils` which we implemented
- This creates the model architecture and compiles the model with the specified loss functions and metrics.

4.1 Training on a Large Dataset

In order to facilitate the training on the large dataset:

- We have pre-processed the entire dataset into patches and stored the patches in the `h5py` format.
- We also wrote a custom Keras `Sequence` class which can be used as a `Generator` for the keras model to train on large datasets.

5 Evaluation

Now that we have a trained model, we'll learn to extract its predictions and evaluate its performance on scans from our validation set.

5.1 Overall Performance

First we measure the overall performance on the validation set.

- We can do this by calling the keras evaluate generator function and passing in the validation generator

Using the validation set for testing

- Note: since we didn't do cross validation tuning on the final model, it's okay to use the validation set.
- For real life implementations, however, it would be wiser to do cross validation as usual to choose hyperparameters and then use a hold out test set to assess performance

5.2 Patch-level predictions

When applying the model, we will want to look at segmentations for individual scans (entire scans, not just the sub-volumes)

- This will be a bit complicated because of our sub-volume approach.
- First we keep things simple and extract model predictions for sub-volumes.
- We can use the sub-volume which we extracted at the beginning of the assignment.

We can extract predictions by calling `model.predict` on the patch.

- We will add an `images_per_batch` dimension, since the `predict` method is written to take in batches.
- The dimensions of the input should be `(images_per_batch, num_channels, x_dim, y_dim, z_dim)`.
- We use [numpy.expand dims]() to add a new dimension as the zero-th dimension by setting axis=0

Currently, each element of `patch_pred` is a number between 0.0 and 1.0.

- Each number is the model's confidence that a voxel is part of a given class.
- We use a threshold of 0.5.

The model is covering some of the relevant areas, but it's definitely not perfect.

- To quantify its performance, we can use per-pixel sensitivity and specificity.

5.3 Running on entire scans

As of now, our model just runs on patches, but what we really want to see is our model's result on a whole MRI scan.

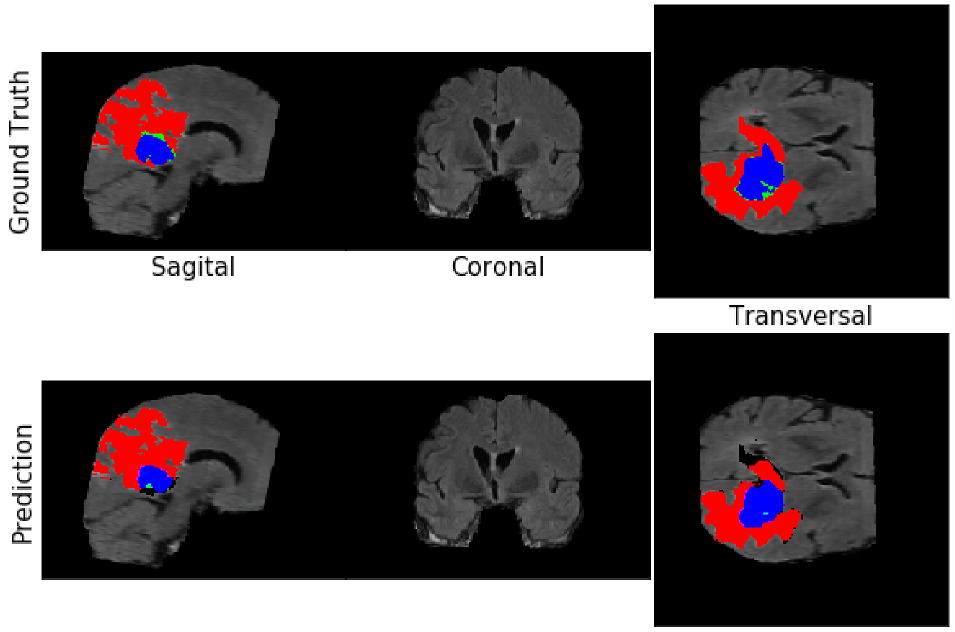
- To do this, we generate patches for the scan.
- Then we run the model on the patches.
- Then combine the results together to get a fully labeled MR image.

The output of our model will be a 4D array with 3 probability values for each voxel in our data.

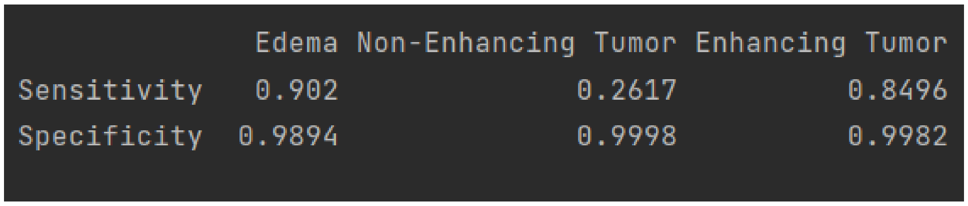
- We then can use a threshold (which you can find by a calibration process) to decide whether or not to report a label for each voxel.

We have written a function that stitches the patches together: `predict_and_viz(image, label, model, threshold)`

- Inputs: an image, label and model.
- Ouputs: the model prediction over the whole image, and a visual of the ground truth and prediction.
- The first and second predictions takes about 7 to 8 minutes to run.



Final Results:



```
              Edema Non-Enhancing Tumor Enhancing Tumor
Sensitivity   0.902              0.2617          0.8496
Specificity  0.9894              0.9998          0.9982
```