Udacity AWS ML Engineer nanodegree – final project

Introduction

The objective of this study is to train and deploy a segmentation model on AWS SageMaker to automatically identify lower-grade gliomas (a form of brain tumour) from input MRI images. Glial cells are non-neuronal cells in the brain and spinal cord that provide support and protection for neurons. Brain tumours that start in these glial cells are known as Gliomas. The segmentation of gliomas is an important task for the early diagnosis of brain tumours. According to Buda et al. (2019), several studies have indicated that that tumour shape may be indicative of patient prognosis. However, obtaining tumour features requires manual segmentation of brain MRI images, which is a time-consuming and tedious process. Deep learning offers the potential to automate this process which provides important pre-operation information and may assist with the identification of tumour genomic subtype. Here the main goal is to develop an end-to-end solution rather than achieving the best model performance.

Dataset

The dataset used in this study is available from Kaggle. It consists of three channel (pre-contrast, FLAIR, post-contrast) MRI images along with the corresponding FLAIR abnormality segmentation masks (see Figure 1 for example). The MRI images originate from The Cancer Imaging Archive (TCIA) and includes 110 patients (pre-operative) from five institutions in The Cancer Genome Atlas (TCGA) lower-grade glioma collection. We're told that 15 of these patients had either the pre- or post-contrast image missing. In these cases, the missing data was replaced with a copy of the FLAIR image to ensure all inputs are three channels.

The binary segmentation masks were manually created by a medical school graduate by drawing the outline of FLAIR abnormality on the FLAIR channel (Buda et al., 2019). An experienced radiologist verified these masks and made modifications when necessary.

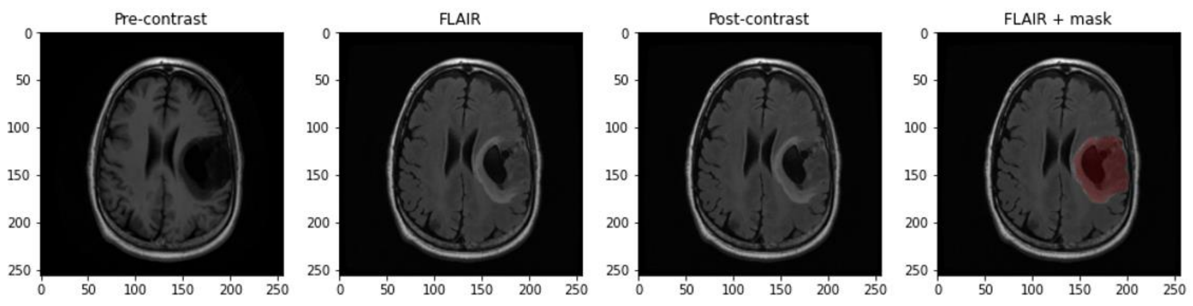


Figure 1 – Example three channel MRI data slice with segmentation mask (red)

Exploratory data analysis

In this section we take a closer look at the dataset to identify any issues that need to be considered when processing the data and training the model. Refer to the notebook named 01_data-prep-eda.ipynb for more details.

Example patient data

Both the input MRI images, and the segmentation masks are provided as 256 x 256-pixel tiff images. The original 3D MRI volumes were split into 2D slices, each 256 x 256 pixels in size. Each patient has a different number of slices, ranging from 20 to 88 slices. The slices for an example patient are shown in Figure 2 as three channel colour images.

One thing to note here is that some slices are very similar, since they come from the same patient (e.g., slices 19-21 in Figure 2). Therefore, I decided to make sure that all the data from one patient is in one dataset (e.g., only in the training data). Many people on Kaggle extract all the images from all patients and then randomly split the data, but I believe that this results in data leakage.

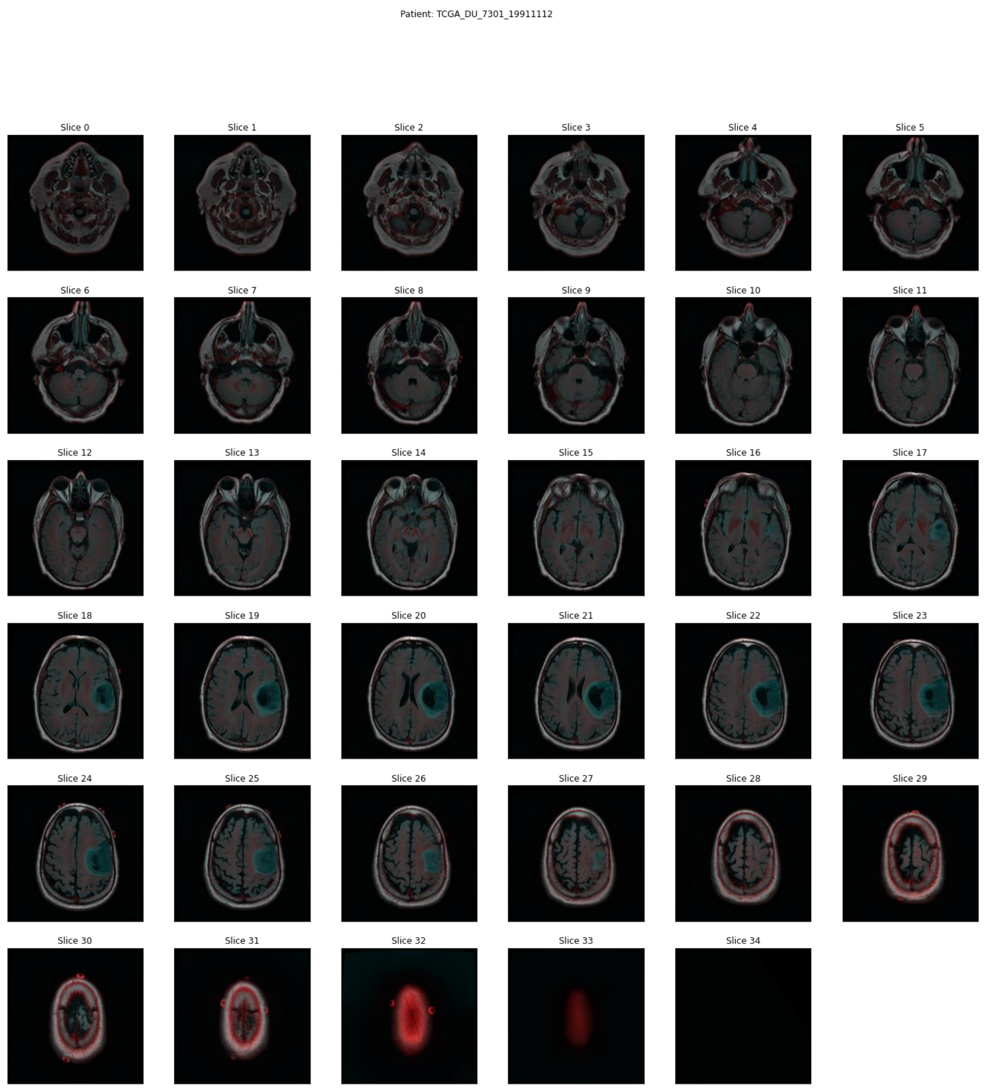


Figure 2 – 2D MRI slices from an example patient

Figures 3 shows the same patient data with the segmentation mask overlaid on top (where red indicates a tumour). Note in these examples the MRI image is shown in grayscale. An important note to make here is that not every slice has positive mask labels. In fact, most are 100% the negative class. This means that we have an imbalanced dataset which we need to consider during model training.

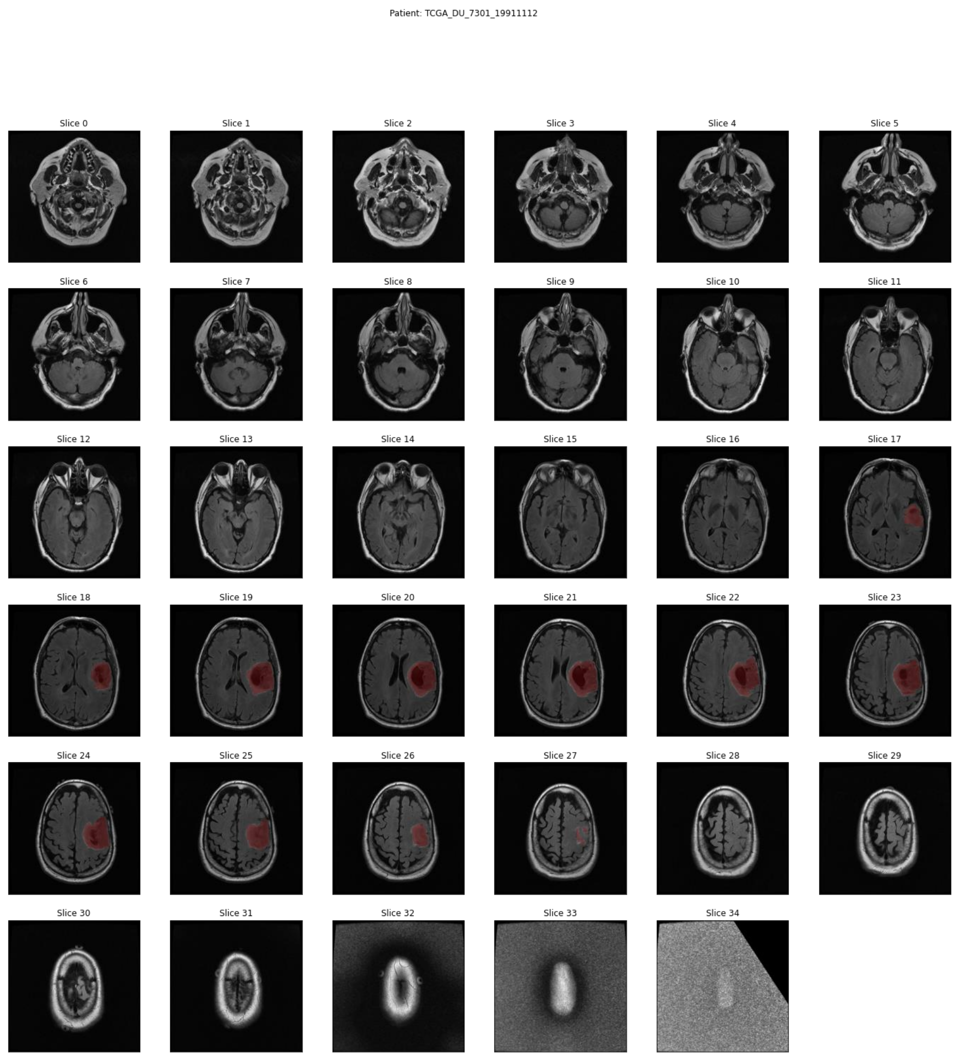


Figure 3 – MRI images (grayscale) with overlaid segmentation mask (red)

Data channels

We’re told in the dataset README file that the MRI channels are ordered as pre-contrast, FLAIR and post-contrast. The channel data for a few example slices are show in Figure 4. However, looking at the example, it looks more like FLAIR is in the first channel. We're told that the segmentation masks were picked using FLAIR, so I'm not sure the ordering they gave us is correct. In any case, it doesn't affect how we're going to train the model (as I’ll use all three channels), but it's an observation I made.

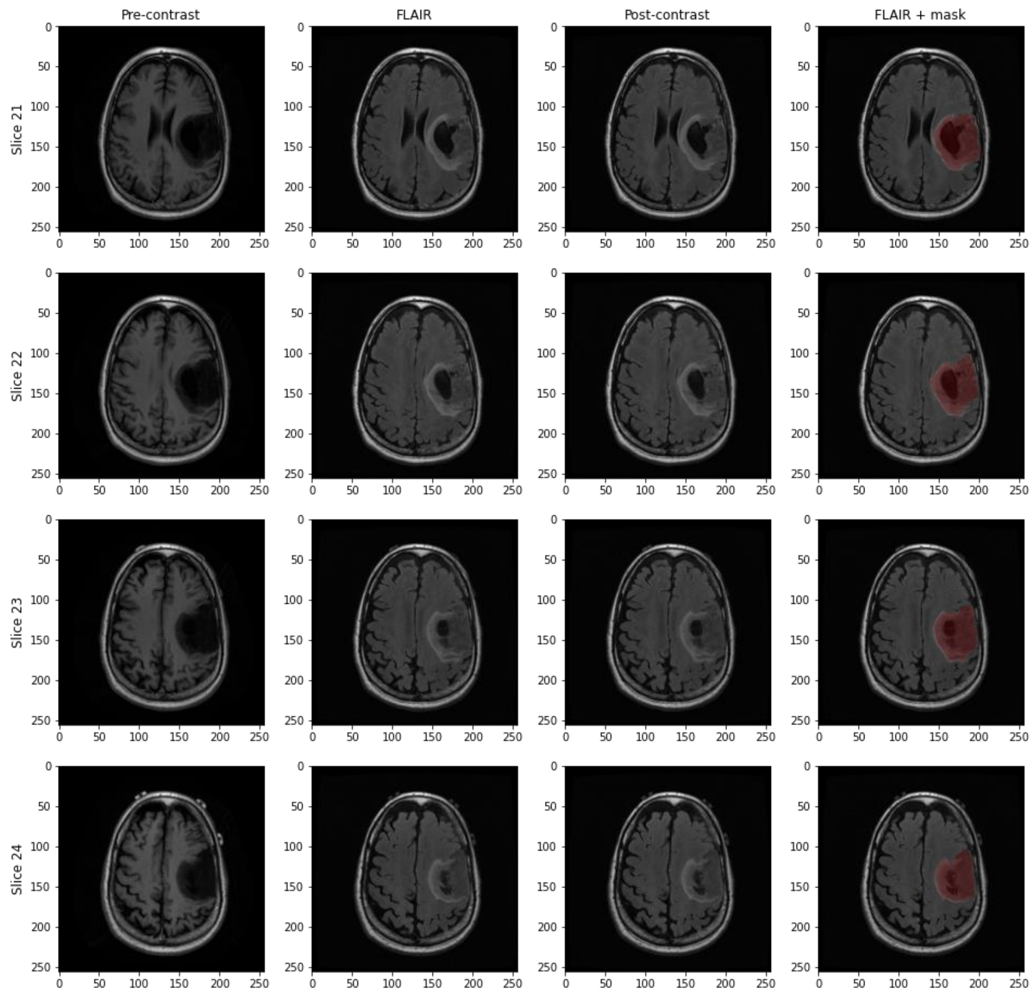


Figure 4 – Three channel input MRI images for several slices from an example patient. The right-hand image shows the FLAIR input channel with the segmentation mask overlay.

Dataset split

The provided data was not already split into training, validation and test datasets. I created a function to do this job, trying to keep the proportion of slices with and without tumour pixels approximately the same between the three datasets. To avoid data leakage, all data from the same patient was moved to one of the datasets.

Figure 5 shows the proportion of normal images (blue) and those with tumour present (orange) for the three datasets. Here the blue bar represents the proportion of mask slices that have 100% negative labels. We can see this is around 65% for the train and validation datasets, but slightly lower for the test dataset. Ideally, they would all be exactly the same, but this is difficult to achieve due to the way I'm splitting the data (i.e., keeping all data from one patient in a single dataset).

This means that only around 35% of the slices contain FLAIR abnormalities. Even the slices that do still only have a small percentage of pixels that are positive, so we have an imbalanced dataset. This is another consideration for the model training section.

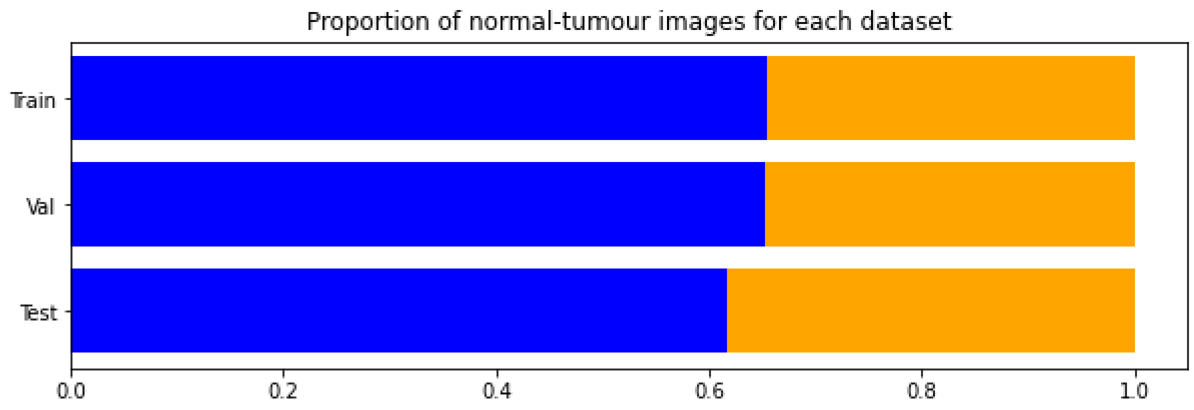


Figure 5 – Proportion of normal and tumour images in the three datasets

In total we have the following number of samples in each dataset:

- Training: 3236 images
- Validation: 359 images
- Test: 334 images

We only have 3236 samples available for training, which is a very small dataset. We will need to consider things like data augmentation to help our model to generalize to data it hasn't seen before.

Dataset statistics

I also found the minimum, maximum, mean and standard deviation of the pixel's values for each training image (split by channel). Figure 6 shows these results as histograms. A few things to note from this plot:

- **Min value:** The minimum value is generally zero. However, a small number of pre-contrast and post-contrast images have larger minimum values.
- **Max value:** Some of the images appear to have a maximum value of 0, meaning they are just blank slices.
- **Mean:** The mean generally lies in the range of 10-30 for most of the images. The mean is kept low by the black pixels that surround the brain MRI scan.

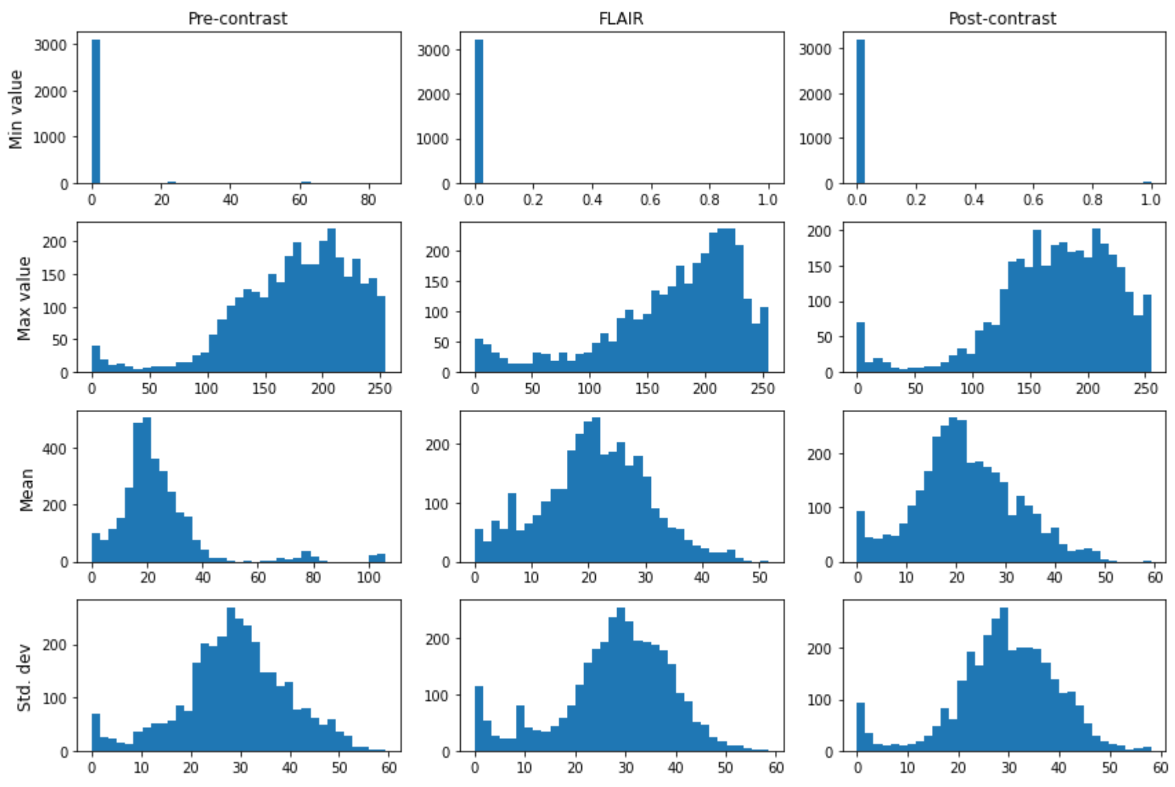


Figure 6 – Training image statistics (per channel) shown as histograms

EDA summary

To summarize some of the key points from this EDA section:

- We have a very small dataset with only 3236 samples for training. During training we will need to carefully consider methods such as data augmentation.
- All data slices from one patient should be kept in the same dataset to avoid data leakage. This has already been taken care of when I split the downloaded data.
- The majority (around 65%) of the slices have no positive pixel label. In addition, the slices that do have FLAIR abnormality present are still dominated by the negative class. Therefore, the dataset is imbalanced.

Methodology

This section describes the methods used for pre-processing of the data, model training, experiment tracking and deployment. For this work I use the MONAI open-source framework for processing of the medical images and for the segmentation models, while PyTorch Lightning is used to organize the training code.

Data processing

As identified in the EDA section, we're dealing with a very small dataset. Therefore, it's essential to use data augmentation techniques to provide more variation for the model to learn from.

In this project I'm making use of the MONAI package, which is a PyTorch-based framework for deep learning with medical data. This provides a convenient way to produce a transformation pipeline for medical data, including data loading and augmentation techniques. In this case the data is in the form of 2D .tif images, so they are relatively easy to work with. However, medical data will often come as 3D DICOM files which would be far more challenging without MONAI.

The pipeline I put together is detailed in the notebook named 02_train-and-deploy-sagemaker.ipynb in the notebooks folder. An example of the outputs from the transformation pipeline is provided in Figure 7. In summary, the pipeline includes:

- Loading of data from .tiff files
- Reordering the tensor dimensions so that the channel is first (as expected by PyTorch)
- Data normalization
- Transfer of data to GPU device (optional)
- Random horizontal flips
- Random data rotation
- Random spatial crop
- Random scale intensity change
- Random amplitude shift
- Addition of Gaussian noise

Note that MONAI has a nice option that allows us to move the data to the GPU as part of this pipeline. Putting the data on GPU can save significant time during training, as it doesn't need to keep moving the data between RAM and the GPU. This is a good option in this case as we have a small dataset. Also, it means that the steps prior to this (which are all deterministic) don't need to be repeated on every epoch. Note that steps that are stochastic (i.e., random) in nature need to come after the transfer to GPU entry in the pipeline. The MONAI transformations have been designed to be able to be run on GPU, so the random processes (e.g., random crop, amplitude shifts) will be applied on each epoch.

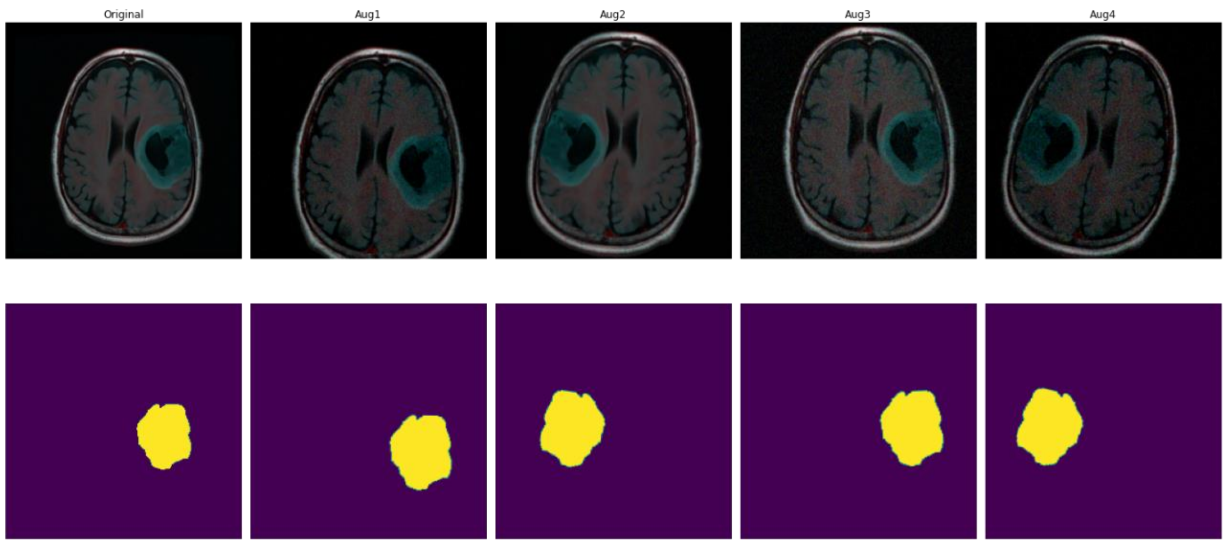


Figure 7 – Example data augmentation applied to one MRI image and corresponding segmentation mask. The left column shows the original MRI and mask, with the next four columns showing different augmented views output from the MONAI transformation pipeline.

Model training

In this project I'm using PyTorch Lightning, a framework built on top of PyTorch, to help organize and structure the model training code. This consists of three main components:

- **The LightningDataModule**: This is a Python class used to organize things related to data loading and processing (such as PyTorch Datasets and DataLoaders). My dataloader class (named BrainMRIData) can be found in the brain_datamodule.py file inside the project repo.
- **The LightningModule**: The LightningModule class inherits everything from torch.nn.Module and adds many more useful features on top. Here we can configure what happens during training and validation steps, without having to write all of the normal code that does things such as clearing the gradients, running back propagation and updating parameters (these are all taken care of for us). My code is contained in the BrainMRIModel class in the brain_model.py module.
- The PyTorch Lightning **Trainer**: This largely automates the training process and allows us to easily apply best practices.

Model architecture

In the current work, two different segmentation model architectures have been included for testing. This includes the well-known UNet (Ronneberger et al., 2015) architecture which has been successfully utilized for a wide range of segmentation tasks. This model is implemented using the MONAI UNet class. A hyperparameter search is run to determine the best model hyperparameters for this task (see later section for details).

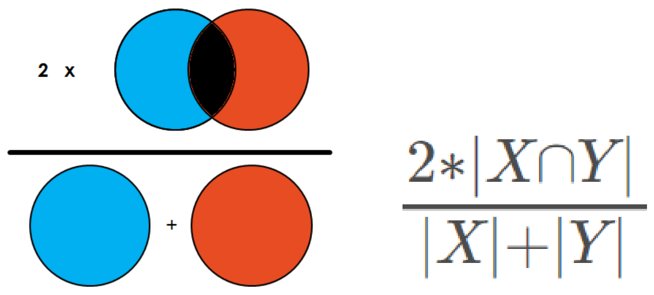
The second architecture tested is the Attention UNet based on the work by Oktay et al. (2018). This is again implemented using MONAI with the AttentionUnet class. The addition of the attention mechanism to the original UNet architecture can help the network learn where to focus on an image, which may improve performance.

Loss function

Here the DiceFocalLoss from MONAI was used as the loss function for training. This combines the Dice loss and the Focal loss. Using the focal loss can help the model to concentrate on more difficult samples, rather than applying equal weighting to all the samples. This can help in cases where we have imbalanced datasets, such as encountered here.

Metric

The performance of the model is measured using the Dice score (or coefficient), which measures the similarity between two sets of data (in this case the predicted segmentation mask and the manually labelled mask). The Dice score is illustrated in Figure 8. Put simply, it is two times the area of overlap between the two images divided by the total number of pixels in the images.



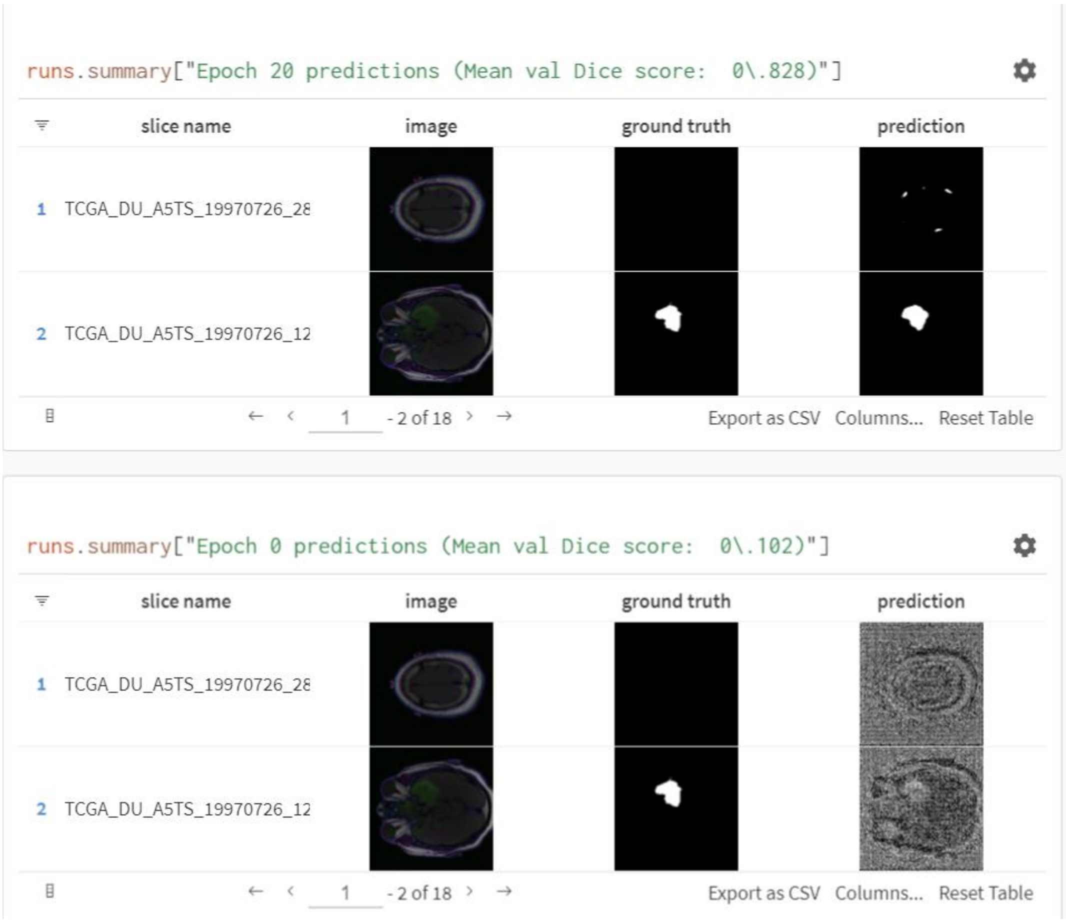
$$\frac{2*|X \cap Y|}{|X|+|Y|}$$

Figure 8 – Illustration of the Dice score (Source)

The Dice coefficient is very similar to the Intersection over Union (IoU). Scores ranges from 0 to 1, with 1 meaning perfect prediction. The use of pixel accuracy will not be a good indicator of performance in this case due to the class imbalance of the data (i.e., the model could predict all pixels to be the negative class and it would achieve a very high pixel accuracy score).

Experiment tracking

I included the option to track all experiments using Weights and Biases in the code. This is great for tracking things such as losses/metrics during training, system information (such as CPU/GPU usage) and even artifacts such as the model checkpoints. For this I also created a custom PyTorch Lightning callback (BrainSegPredictionLogger inside the brain_model.py module) to output example predictions from the validation dataset as the training progresses. To avoid logging too many things, this is set to only log the predictions if the validation Dice score improves. An example of this output is shown in Figure 9.



```
runs.summary["Epoch 20 predictions (Mean val Dice score:  0\.828)"]
```

	slice name	image	ground truth	prediction
1	TCGA_DU_A5TS_19970726_28			
2	TCGA_DU_A5TS_19970726_12			

← ‹ 1 - 2 of 18 › → Export as CSV Columns... Reset Table

```
runs.summary["Epoch 0 predictions (Mean val Dice score:  0\.102)"]
```

	slice name	image	ground truth	prediction
1	TCGA_DU_A5TS_19970726_28			
2	TCGA_DU_A5TS_19970726_12			

← ‹ 1 - 2 of 18 › → Export as CSV Columns... Reset Table

Figure 9 – Example prediction logging of validation data in Weights and Biases.

Hyperparameter tuning

An important part of any machine learning project is determining a suitable set of hyperparameter for the model. In the case of the UNet architecture, this includes things such as the learning rate, dropout rate, batch size and number of filters per layer. In this project I've included two different methods of performing the hyperparameter search:

1. **AWS SageMaker hyperparameter tuning job** – Here a job is submitted to AWS to run a hyperparameter search
2. **Optuna –** An open-source package that can run the hyperparameter search on a local machine

AWS SageMaker

An example of how to run a hyperparameter tuning job with AWS SageMaker is included in the notebook named 02_train-and-deploy.ipynb. However, a small dataset like this isn't ideal for running in the cloud if you have your own GPU enabled machine. Each trial (model with unique set of hyperparameters) of the search is run on a new EC2 instance, so each time you need to wait for the EC2 instance to start (which may take some time with spot instances), retrieve the Docker image and download the training data. In this case the startup process takes longer than training the model (which we need to pay for). As highlighted in the notebook, training on an AWS instance also seems to be much slower than when run locally, even when the EC2 instance has a better GPU card.

As a result of the issues highlighted above, I just ran the search for two trials and deployed the best of these models to demonstrate the process. For a much larger dataset (where training takes much longer than the time to setup the EC2 instance) it may make sense to use SageMaker for hyperparameter tuning.

Optuna

As a result of these issues, I decided to run my hyperparameter search outside of SageMaker on my local machine (note that although we can run SageMaker training jobs on a local machine, we cannot do this for hyperparameter optimization). We can still deploy the model on SageMaker, as discussed later in the report.

The Optuna open-source package was used for running the hyperparameter search. This can be run using the file named optuna-hpo.py in the scripts folder. By default, this will track the experiments in Weights and Biases, with a custom callback created to only upload the model artifact when the best model score (measured by the validation Dice score) improves (rather than saving the model from every trial run).

The hyperparameter search space is defined in the objective function inside this script. The search space used is defined in Table 1.

Hyperparameter	Search space	Sampling type
Dropout rate	[0.0, 0.5]	Uniform
Learning rate	[0.000001, 0.1]	Log-uniform
Batch size	[16,32]	Categorical
Number filters block 1	[8,16,32]	Categorical

Table 1 – Hyperparameter search space

Results

A summary of the hyperparameter search and the predictions from the best model are provided in this section.

HPO analysis

A separate hyperparameter search was run for the UNet and attention UNet architectures. The analysis of these can be found in the 03a_optuna-hpo-1_unet_analysis.ipynb and 03b_optuna-hpo-

2_unet-attention_analysis.ipynb notebooks respectively. The results can also be explored in the Weights and Biases project.

UNet

A total of 100 trials of the hyperparameter space were run using the Tree-structured Parzen Estimator (TPE) hyperparameter search option, which is based on Bayesian reasoning. Figure 10 shows the optimization history for this search, where the objective is to maximize the validation Dice score. The first 10 trials are sampled randomly, resulting in wide variability in the objective value. After this we can see the gradual improvement in model performance as the search converges on the best set of hyperparameters.

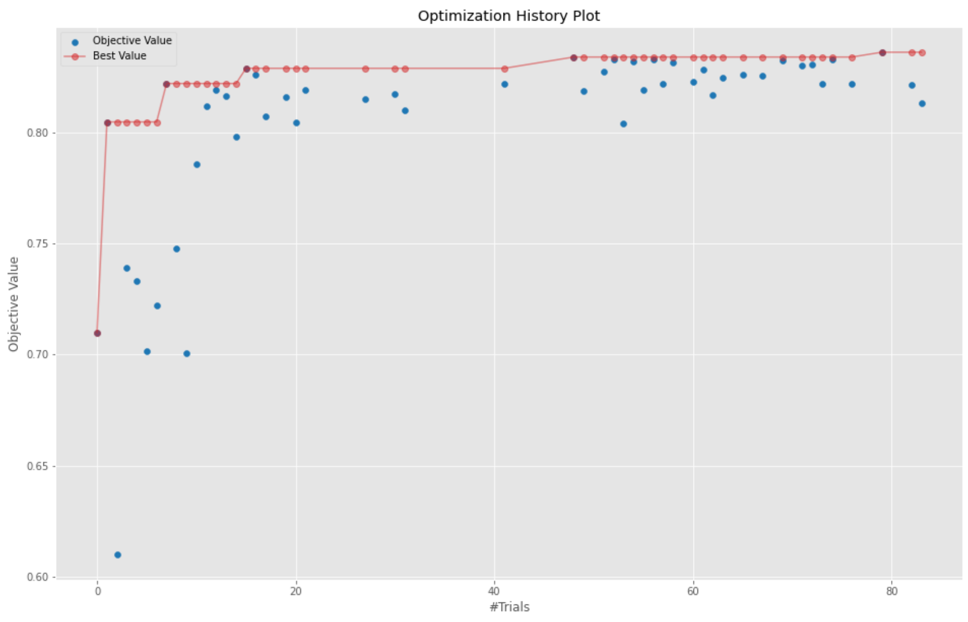


Figure 10 – Optimization history of the TPE hyperparameter search for the UNet architecture. Blue dots are best validation Dice score for each trial, while the red line indicates the best score (produced using Optuna)

Figure 11 shows the effect of the different hyperparameters on the model performance. The colour of the points represents the epoch number, so we can see how the search starts to converge on a learning rate of around 10^{-3}.

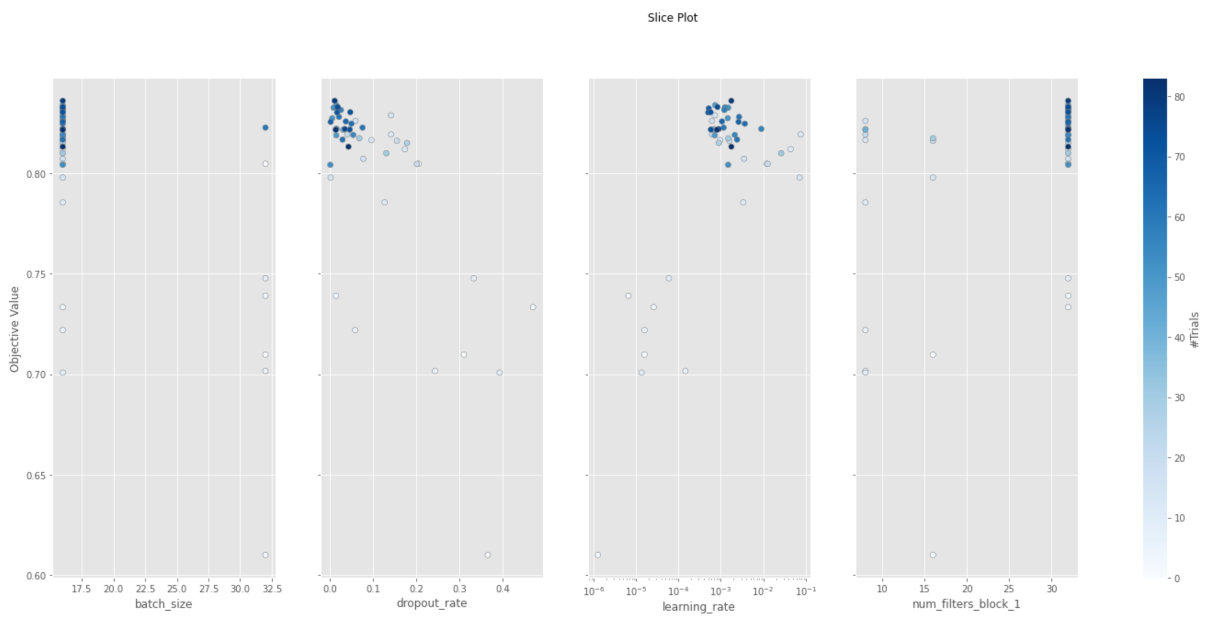


Figure 11 – Variation in model performance with hyperparameter values (produced using Optuna)

The highest validation Dice score was produced by trial 84, resulting in a score of 0.839. The hyperparameters used for this trial were:

- Batch size: 16
- Dropout rate: 0.019747
- Learning rate: 0.001087
- Number filters in block 1: 32

Further details and plots for this trial are available in the Weights and Biases run page. For instance, various plots from the training history are shown in Figure 12, including the validation Dice score and training/validation losses.

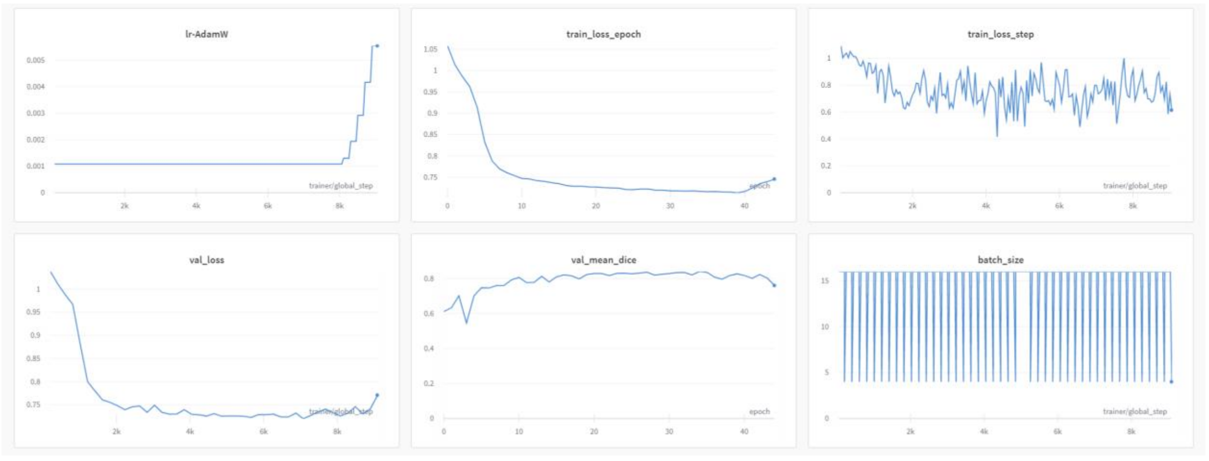


Figure 12 - Training history for trail 84 from the UNet hyperparameter search (from Weights & Biases)

Useful information about the system performance, such as CPU and GPU usage, is also logged and illustrated in Figure 13.

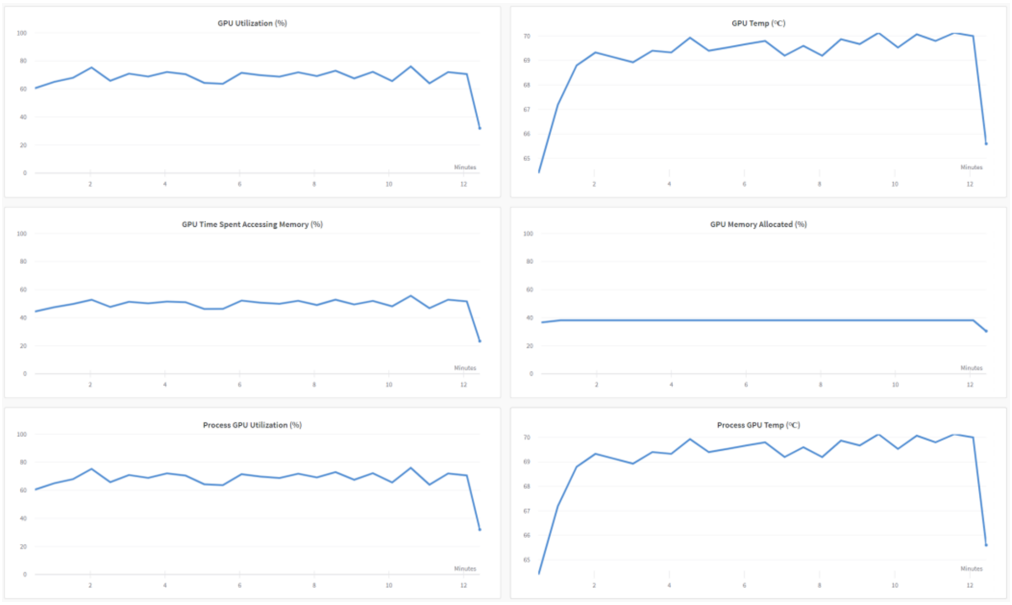


Figure 13 – System details logged in Weights and Biases

We can also visualize the distribution of gradients at different points during training for different layers in the UNet architecture, as shown in Figure 14.

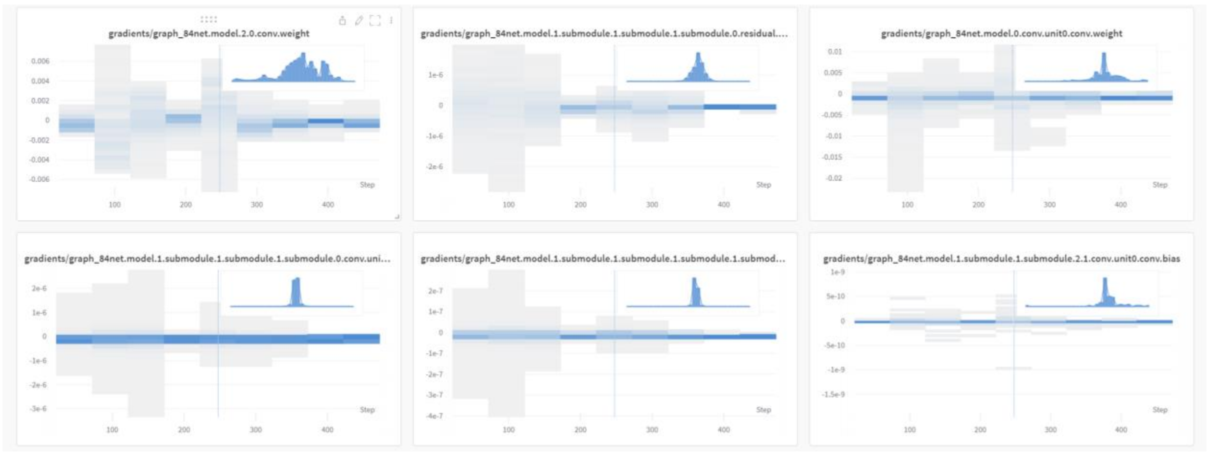
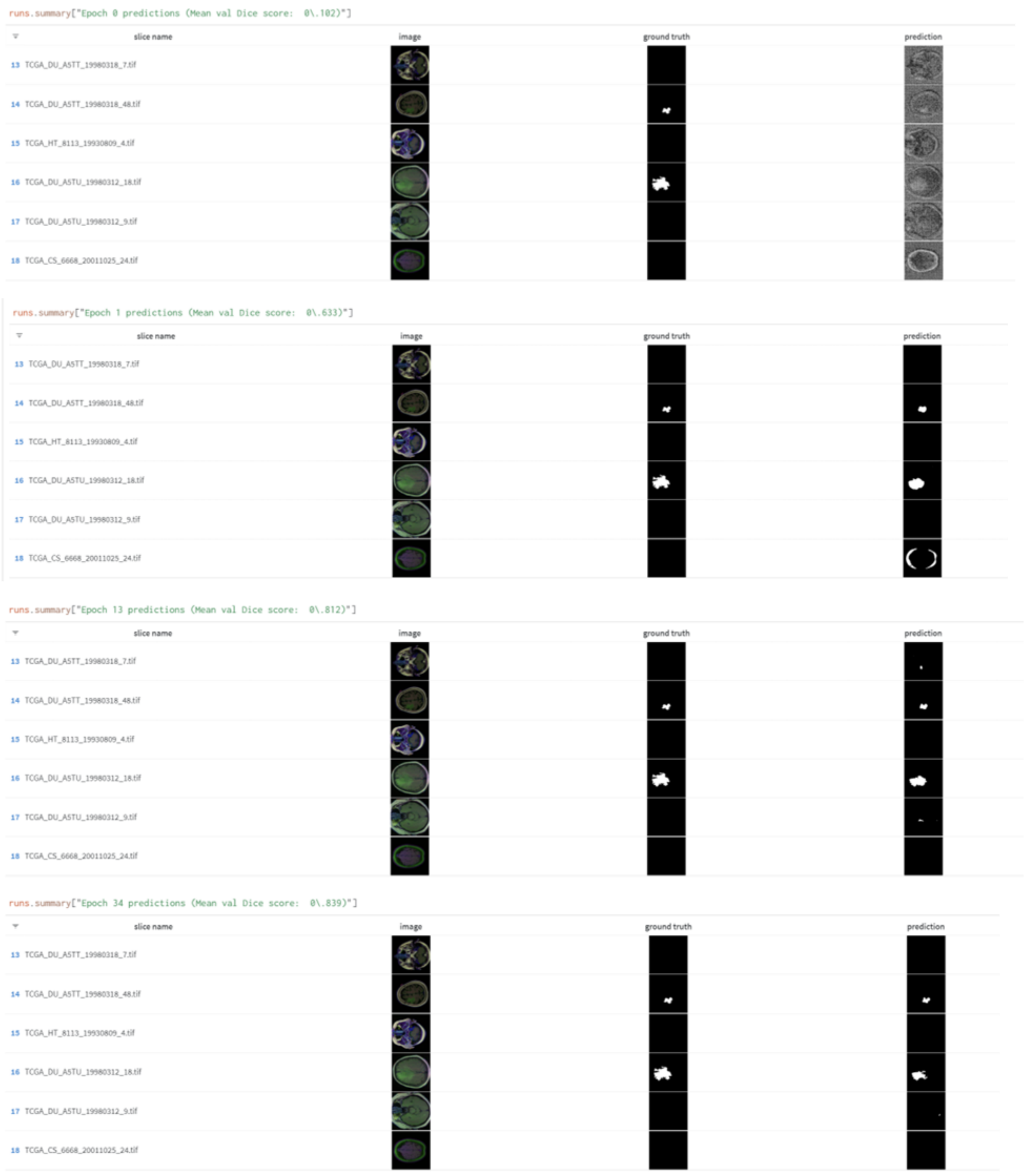


Figure 14 – Gradient distribution for different layers in the UNet model during training

I also included a custom callback in my training code that outputs example predictions on the validation data if the validation Dice score improved. This allows us to visualize how the model performance changes as the epochs progress. The example in Figure 15 shows that before the model has learned anything (epoch 0), it outputs something like an edge detector image. After just two epochs of training the model has already started to learn something useful, although it still has some major errors. By epoch 34 we reach the best performance, which captures the details of the tumour slices quite well (while also not predicting tumours on the slices where none exist).



```
runs.summary["Epoch 0 predictions (Mean val Dice score:  0\.102)"]
```

	slice name	image	ground truth	prediction
13	TCGA_DU_A5TT_19980318_7.tif			
14	TCGA_DU_A5TT_19980318_48.tif			
15	TCGA_HT_8113_19930809_4.tif			
16	TCGA_DU_A5TU_19980312_18.tif			
17	TCGA_DU_A5TU_19980312_9.tif			
18	TCGA_CS_6668_20011025_24.tif			

```
runs.summary["Epoch 1 predictions (Mean val Dice score:  0\.633)"]
```

	slice name	image	ground truth	prediction
13	TCGA_DU_A5TT_19980318_7.tif			
14	TCGA_DU_A5TT_19980318_48.tif			
15	TCGA_HT_8113_19930809_4.tif			
16	TCGA_DU_A5TU_19980312_18.tif			
17	TCGA_DU_A5TU_19980312_9.tif			
18	TCGA_CS_6668_20011025_24.tif			

```
runs.summary["Epoch 13 predictions (Mean val Dice score:  0\.812)"]
```

	slice name	image	ground truth	prediction
13	TCGA_DU_A5TT_19980318_7.tif			
14	TCGA_DU_A5TT_19980318_48.tif			
15	TCGA_HT_8113_19930809_4.tif			
16	TCGA_DU_A5TU_19980312_18.tif			
17	TCGA_DU_A5TU_19980312_9.tif			
18	TCGA_CS_6668_20011025_24.tif			

```
runs.summary["Epoch 34 predictions (Mean val Dice score:  0\.839)"]
```

	slice name	image	ground truth	prediction
13	TCGA_DU_A5TT_19980318_7.tif			
14	TCGA_DU_A5TT_19980318_48.tif			
15	TCGA_HT_8113_19930809_4.tif			
16	TCGA_DU_A5TU_19980312_18.tif			
17	TCGA_DU_A5TU_19980312_9.tif			
18	TCGA_CS_6668_20011025_24.tif			

Figure 15 – Validation predictions at different stages during training of trial 84 (from Weights and Biases

Attention UNet

So far, the attention UNet has not been able to produce better results than the conventional UNet. Although it has not run for the same 100 trials yet, the optimization history in Figure 16 doesn't look like it will reach that level (although we should run it for longer to confirm).

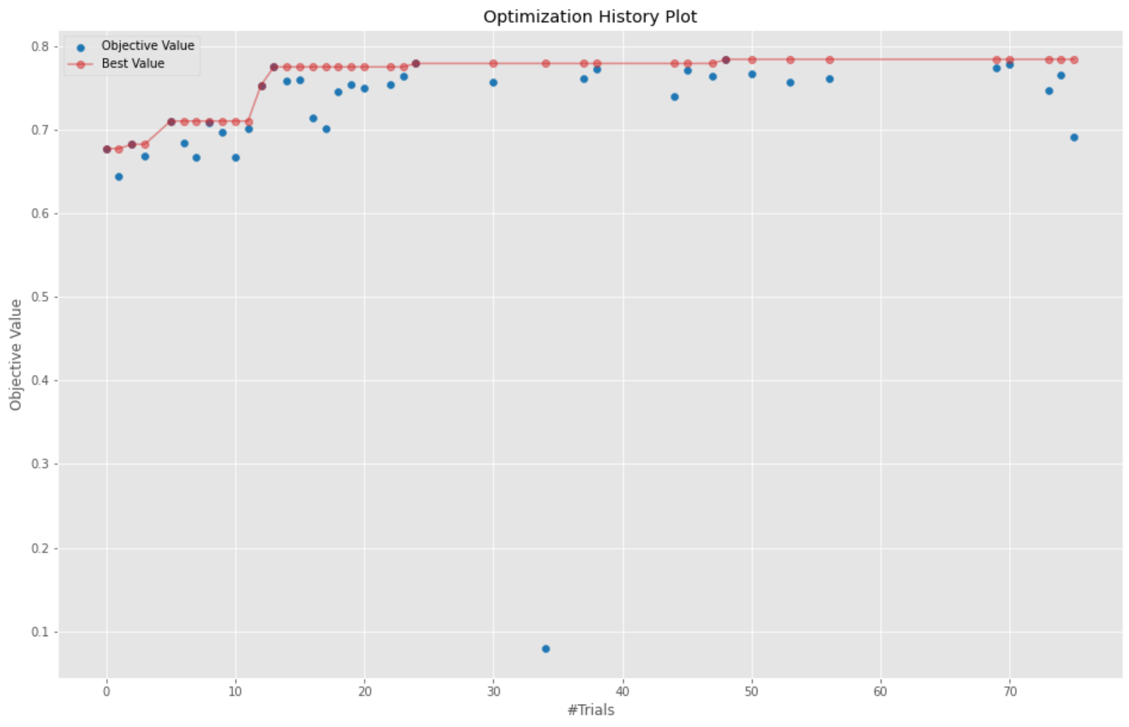


Figure 16 – Optimization history of hyperparameter search run for attention UNet (produced using Optuna

Model predictions

In the notebook named 04_deploy-best-optuna-model.ipynb you will find some analysis of the predictions made by the best model on the validation data (see next section for further details about the deployment process).

The three images with the lowest validation Dice score are displayed in Figure 17. We can see that in each of these cases (in the notebook you will see it's true for the ten worst examples) that the model is predicting tumours for slices where the ground truth shows no tumour labels (represented by yellow shading in these figures).

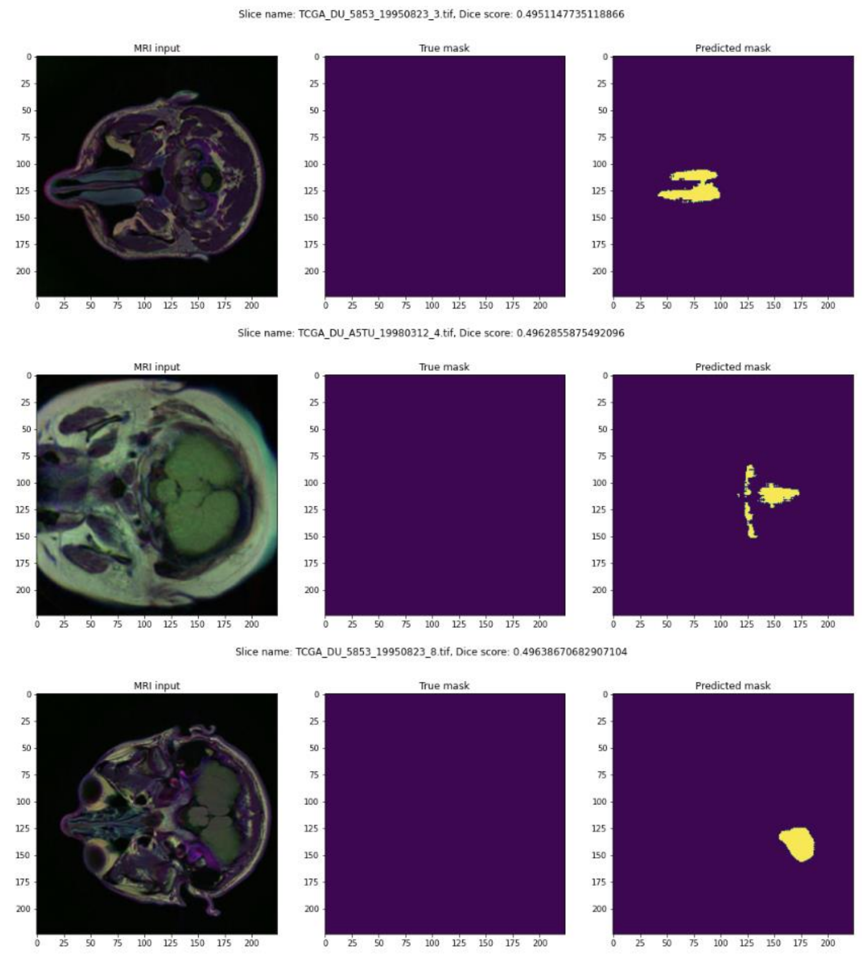

Slice name: TCGA_DU_5853_19950823_3.tif, Dice score: 0.4951147735118866
MRI input
True mask
Predicted mask
Slice name: TCGA_DU_A5TU_19980312_4.tif, Dice score: 0.4962855875492096
MRI input
True mask
Predicted mask
Slice name: TCGA_DU_5853_19950823_8.tif, Dice score: 0.49638670682907104
MRI input
True mask
Predicted mask

Figure 17 – The three worst predictions made by the best model on the validation dataset (as measured by the Dice score)

Figure 18 shows the three worst results (lowest Dice score) where positive labels exist in the ground truth mask:

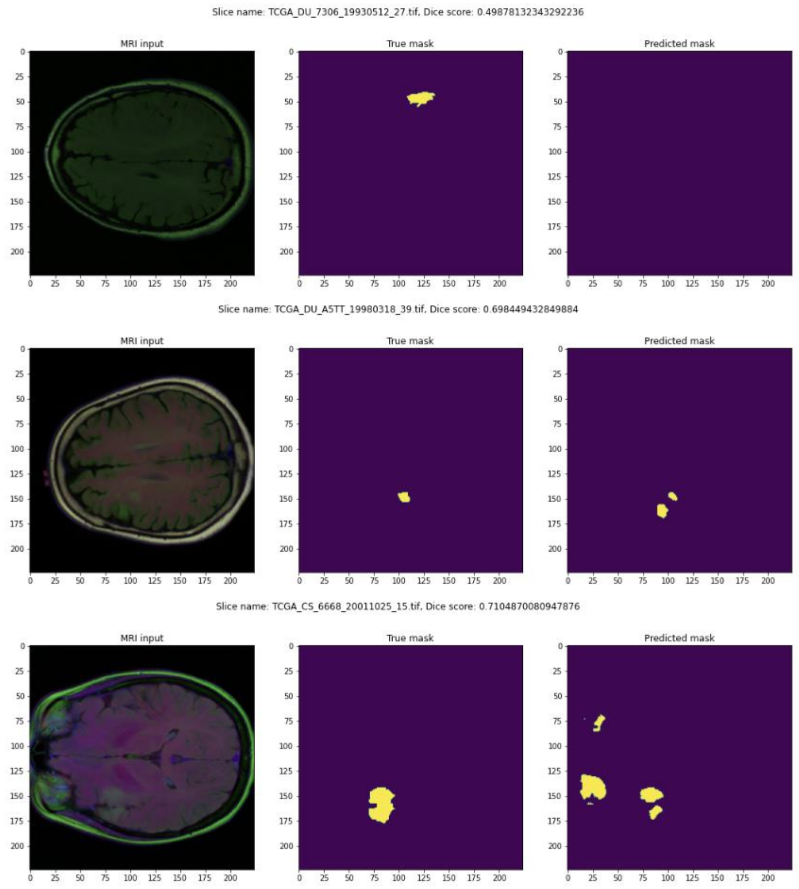

Slice name: TCGA_DU_7306_19930512_27.tif, Dice score: 0.49878132343292236
MRI input
True mask
Predicted mask
Slice name: TCGA_DU_A5TT_19980318_39.tif, Dice score: 0.698449432849884
MRI input
True mask
Predicted mask
Slice name: TCGA_CS_6668_20011025_15.tif, Dice score: 0.7104870080947876
MRI input
True mask
Predicted mask

Figure 18 – The three worst predictions made by the best model on the validation dataset (as measured by the Dice score) for samples with positive labels in the ground truth mask

The three best model predictions on the validation dataset (for slices where the ground truth mask includes positive labels) are displayed in Figure 19, with a highest Dice score of 0.969.

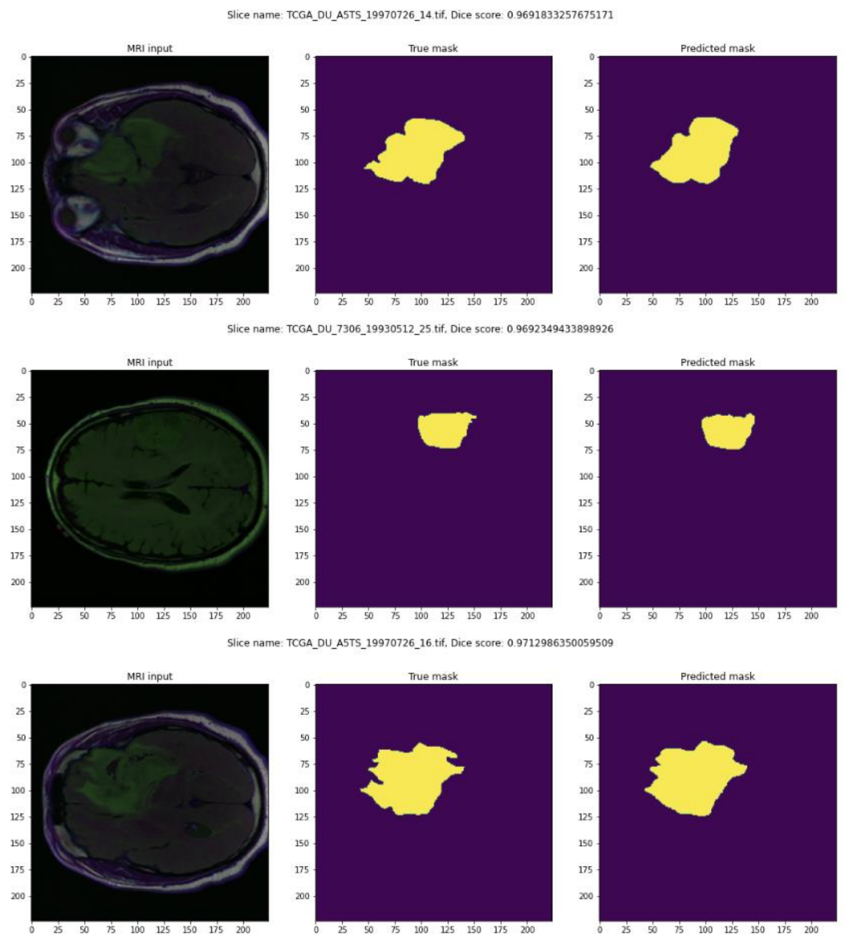

Slice name: TCGA_DU_A5TS_19970726_14.tif, Dice score: 0.9691833257675171
MRI input
True mask
Predicted mask
Slice name: TCGA_DU_7306_19930512_25.tif, Dice score: 0.9692349433898926
MRI input
True mask
Predicted mask
Slice name: TCGA_DU_A5TS_19970726_16.tif, Dice score: 0.9712986350059509
MRI input
True mask
Predicted mask

Figure 19 – The three best predictions made by the best model on the validation dataset (as measured by the Dice score) for samples including positive labels in the ground truth mask

Note that ultimately, we want to perform these predictions and analysis on the test dataset. However, this should only be done once we are fully happy with the model and have finished running all our tests.

Model deployment

Although the Optuna HPO was run outside of SageMaker, we can still use the SDK to deploy the resulting best model. Here we need to use the PyTorchModel class available in the sagemaker package. This process is documented in the 04_deploy-best-optuna-model.ipynb notebook. In this case, the model is deployed as an endpoint that can perform inference in real-time. A better choice for this case may be to implement a batch transform or deploy using a Lambda function.

Discussion

Here I include some details on issues that I encountered during this project and areas where I can improve things in the future.

Issues encountered

Many issues have been encountered during this project and have been documented in the various notebooks in the project. Several issues are discussed in more details below.

Nan loss

At some stage I had an issue where the models would start training okay, but would suddenly fail due to the Dice Focal loss returning Nan. It turned out that this was caused by using mixed precision (16-bit) training. This option is made easy to implement by PyTorch Lightning and helps speed up training by reducing the required memory bandwidth and the time taken to run operations. However, in this case it appears that the model I'm using is not numerically stable, so further testing is required to determine the cause. For now, I continued my training with full 32-bit precision.

Passing Boolean arguments to SageMaker entry point

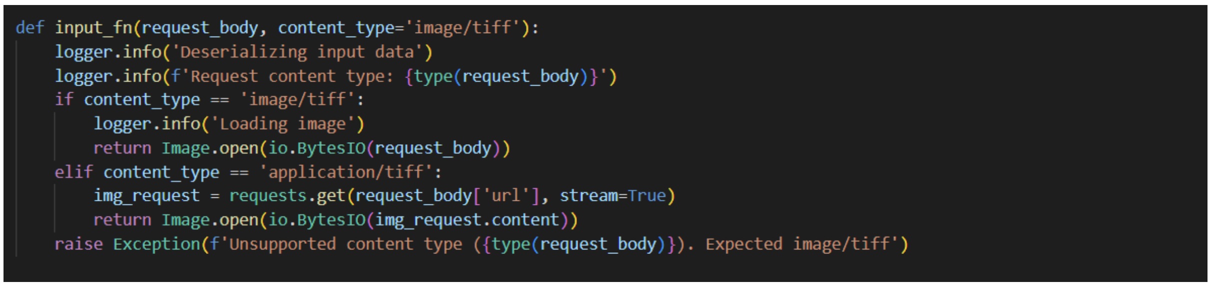
An issue when passing arguments to the entry point that runs on SageMaker is that argparse does not accept Boolean arguments. Even more frustratingly, if you try to do this it will run without error (but everything will use True, even if you specify False). To get around this issue, I include the str2bool function inside the train.py script used as the entry point for training on SageMaker. Note that this was taken from this repository and converts an input string into a Boolean.

Loading models with model_fn in SageMaker deployments

Initially I kept the building of the model architecture outside of the BrainMRIModel class so that any of the models from MONAI could easily be used. However, this caused an issue when trying to load the best model as part of a deployment in SageMaker. To deploy a model, we must provide a function named model_fn, which tells SageMaker how to load the compressed model file. However, this only accepts one input argument model_dir, which is the location of the trained model file. In my case I had issues that several of the hyperparameters could not be tracked by PyTorch Lightning (including the model network and loss function), so needed to be instantiated before loading the model checkpoint. Since there was no way of passing the required hyperparameters to reconstruct the same model architecture, I had to build this process inside of the BrainMRIModel class.

Image serialization for deployment

One problem I have not been able to resolve is serializing the tiff images in bytes format. The code I tried to use is shown below in Figure 20. This worked on my previous project (replacing tiff with jpeg), but in this case I run into some form of Pickle error.



```
def input_fn(request_body, content_type='image/tiff'):
    logger.info('Deserializing input data')
    logger.info(f'Request content type: {type(request_body)}')
    if content_type == 'image/tiff':
        logger.info('Loading image')
        return Image.open(io.BytesIO(request_body))
    elif content_type == 'application/tiff':
        img_request = requests.get(request_body['url'], stream=True)
        return Image.open(io.BytesIO(img_request.content))
    raise Exception(f'Unsupported content type ({type(request_body)}). Expected image/tiff')
```

Figure 20 - Example input_fn code for SageMaker inference where input tiff image is serialized as bytes

For now, I've passed NumPy data since this is an accepted form of serialized data that SageMaker will deal automatically with.

Areas for improvement

Several things can be tried in the future to further improve the results and usefulness of the code. This includes:

Try different model architectures

There are many alternative model architectures that can be tested for segmentation. These include:

- **Vision Transformer (ViT):** Transformers have produced state-of-the-art results in many different fields, so may be worth trying here (although maybe the dataset is too small). MONAI include an implementation as shown here.
- **UNETR:** This modified version of the UNet uses transformers in the encoder part of the network. It is available here in MONAI.

Create custom Docker image

Currently every time I run a test on a local SageMaker container it needs to install all the additional Python dependencies. Although this is better than running on an AWS SageMaker container (which needs to start an EC2 instance and download the container image each time), it still slows down development. To further speed things up, I could extend the base PyTorch container that I'm using with the additional packages and push the new image to the AWS Elastic Container Registry (ECR).

Oversampling

The dataset is highly imbalanced, with the number of pixels without tumour vastly outnumbering those with the tumour label. One option that could be tested is oversampling the slices with the positive class to reduce the class imbalance.

Conclusions

In this project I've demonstrated how to build an end-to-end workflow for deploying a segmentation model on AWS SageMaker. In this case we saw how to do this to predict lower-grade gliomas from MRI images. Due to the small dataset size, running training jobs on the cloud was found to be sub-optimal. Instead, a local hyperparameter search was performed using Optuna and the best model deployed using the SageMaker SDK. So far, the standard UNet model produced the best validation Dice score of 0.839, but experiments with newer model architectures may lead to improved results.

References

Buda, M., Saha, A. and Mazurowski, M.A., 2019. Association of genomic subtypes of lower-grade gliomas with shape features automatically extracted by a deep learning algorithm. Computers in biology and medicine, 109, pp.218-225. 1906.03720.pdf (arxiv.org)

Kaggle Brain MRI Segmentation dataset: Brain MRI segmentation | Kaggle

Oktay, O., Schlemper, J., Folgoc, L.L., Lee, M., Heinrich, M., Misawa, K., Mori, K., McDonagh, S., Hammerla, N.Y., Kainz, B. and Glocker, B., 2018. Attention u-net: Learning where to look for the pancreas. arXiv preprint arXiv:1804.03999.

Ronneberger, O., Fischer, P. and Brox, T., 2015, October. U-net: Convolutional networks for biomedical image segmentation. In International Conference on Medical image computing and computer-assisted intervention (pp. 234-241). Springer, Cham.