Segmentation of Brain Tumors using UNet: With or Without Data Augmentation?

Lidia Luque
Faculty of Mathematics and Natural Sciences, University of Oslo

(Dated: December 17, 2021)

Tumor segmentation is a complex and time consuming but necessary task in clinical practice. Deep learning models, particularly CNN-based models have shown promise in making segmentation simpler and more reliable. Because of limited datasets, data augmentation is often used in the medical domain. In this project I assess the value of data augmentation by training a 3D convolution-based UNet model on MRI volumes of patients with brain tumors with and without data augmentation. The model without data augmentation converges quicker and achieves a higher Dice score of 0.780 compared to 0.768 without augmentation. More work is needed to explore whether different augmentation functions can achieve better results.

I. INTRODUCTION AND BACKGROUND

Glioblastomas are the most common type of brain cancer. Despite extensive research to develop new treatment methods, the prognosis is still bleak with median survival of 15-23 months and a 5-year survival rate of less than 6% [1]. Magnetic Resonance Imaging (MRI) is the standard imaging tool used for diagnostic proposes, surgical planning and monitoring of the disease's progression [2]. Typically, radiologists interpret the images and write a report describing the tumor and the visible changes since the last imaging session. The most precise method to quantify how the tumor has changed is to outline the different components of the tumor, or *segmenting* it, at different time points and from this calculate the change in volume. However, doing so presents several challenges.

Glioblastoma tumors have three different components: the active tumor, also called *enhancing tumor* due to it having high contrast in a certain kind of MRI sequence, the *necrotic core*, which is composed of non-active "dead" cells, and the *peritumoral edema*, the inflammation the surrounds the active tumor, which is also infiltrated by tumor cells[1]. These tumors are highly heterogeneous and the boundaries between these tissues can be very entangled and hard to define. This makes it very hard for humans, even highly trained radiologists, to outlines these borders in a consistent manner[3]. Radiologists are often not certain about their assessment, and even when they are, another radiologist might not agree with it. Another issue is time constrains. Nowadays, MRI machines typically take 3D images of the brain, with resolutions of 1mm being common. This results in volumes with a couple hundred slices -meaning 2D images, I will here use the terms slice and image interchangeably-, and the tumor can typically be seen in several dozens of them. Carefully outlining the tumor in each of these images is very time consuming. Radiologists are a scarce resource, and such careful outlining is thus only done when strictly necessary.

An obvious solution to this problem is to use machine learning methods to segment the tumor. This would to a large extent solve the manpower and time constrain issue, as well as providing a more homogeneous way of segmenting the tumors. While the algorithm will have its pitfalls, it will at least be consistent about them -unlike humans- making comparison between subjects easier, which in turn makes it possible to do better population studies based on these segmentations. Different machine learning techniques have been tried since the early 2000s, but it was not until Convolutional Neural Networks (CNN) began to be used a few years ago that the segmentations created by the trained networks started to approach an acceptable lever for use in medical research, and to a certain extent, clinical practice [4].

Training CNN's or other deep learning models with medical data comes with distinct challenges. For one, access to data is limited. Although glioblastoma is the most prevailing type of brain cancer, it is still -thankfully- a rare disease, and thus imaging data is scarce. Data protection concerns make it hard to share data, further reducing the pool of data a network can be trained on. At the same time, MRI machines are notoriously "inconsistent" across brands and even between models of the same brand. The contrast, the sharpness of the image, and the noise distribution are amongst the parameters that are machine-dependent. As an example: I have been told by experienced radiologists that they can tell will MRI-brand an image comes from just by looking at it. Relatively little data together with a lot of variance within the data presents a challenge when training a network, as it can quickly lead to over-fitting.

In this work, I try to asses the success of one method to reduce over-fitting, data augmentation. In short, a transform with random parameters is applied to the images each time they are used to train the network. Thus, the network sees more images -or at least more variation of the images in the dataset- which allows it to, in principle, reduce over-fitting and get better test accuracies. To test this I will use a CNN architecture called UNet, which was specifically made for biomedical segmentation -but

is now commonly used on many segmentation problems outside the medical domain. I will train this network using a public dataset which contains brain MRI images of glioblastoma patients together with the segmentations produced by radiologists, which act as the ground truth.

First, I will introduce convolutional neural networks and UNets in the theory section. Next, I will give an overview on the dataset used to train and test my network before moving on to a description of the methods used in this project. The results, including their discussion, will be presented afterwards, followed by a short conclusion based on the findings. All the code used in this project can be found in my GitHub [1].

II. THEORY

Here, I will assume knowledge about neural networks, including how they are trained through variations of the gradient descent algorithm, since these topics were covered in the previous project [5]. The discussion on CNNs is based on the lecture notes [6] and a very pedagogical article on *Towards Data Science* [7].

A. CNN

Convolutional Neural Networks are a subset of neural networks specifically designed to process images. Instead of flattening the images as you would in a simple feed-forward neural network, CNNs use convolution to learn spatial features in the training images. Typically, CNNs are used for classification problems, in which case these features might be flattened and passed through a feed forward layer to output a class. However, we are only going to use the feature learning part of the CNN, and so we will limit our discussion to it. Another disclaimer: We will only discuss the most simple type of CNN. While newer CNN architectures can be quite complex, the building blocks we discuss here are the backbone of any CNN. These two elements are the convolution layer -including an activation function- and a downsampling layer, which in the case of CNNs is often a max pooling layer.

[1] https://github.com/lidialuq/fys-stk4155/tree/main/Project3/code

1. Convolution layer

Convolution, in the image domain, is the application of a filter -also called kernel- to an image to get an activation. The easiest way to understand convolution is by looking at an example. Figure 1 shows a simple case where a 3x3 input is convoluted with a 2x2 kernel. We can imagine placing the kernel "on top" of the shaded area and computing the dot product of the kernel and the path of the input image it overlaps, yielding a scalar value. In this example, the output value at position (0,0) is thus given by $0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3 = 19$. We then slide the kernel one position to the right and preform the same operation, which yields 25. We repeat this until the kernel has swept through the entire input image. There are two parameters you can choose when doing a convolution with a given input image and kernel:

- The *stride length*, an integer which defines by how much the kernel will be moved in the x and y directions in order to compute each output value. In our simple CNN, this value is always 1. Larger strides, all else being equal, create smaller outputs.
- The *padding*, an integer ≤ 0 that defines how many rows and columns of zeroes should be added to the input image. Padding can be used to sample the pixels on the edges of the input image as often as if they were in other places. Padding also affects the size of the output image by virtue of changing the size of the input image.

The size of the output image, commonly called the *feature map*, (f_x, f_y) is given by $f_x, f_y = n_x - k_x + 1, n_y - k_y + 1$ where (n_x, n_y) is the size of the input image and (k_x, k_y) is the size of the kernel (assuming that the stride length is 1).

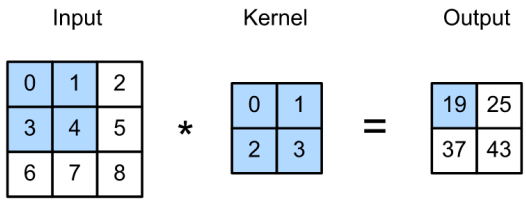

Figure 1. Example showing convolution of an 3x3 input with a 2x2 kernel without padding, giving a 2x2 output. Image retrieved from *Dive into deep learning* [8]

The size of the kernel is another design choice. While earlier CNNs used large kernels (AlexNet, which revolutionized CNNs when introduced in 2012 included 11x11 kernels [9]), smaller kernels are the standard nowadays, with 3x3 being a popular choice.

From this discussion so far, it would be logic to think

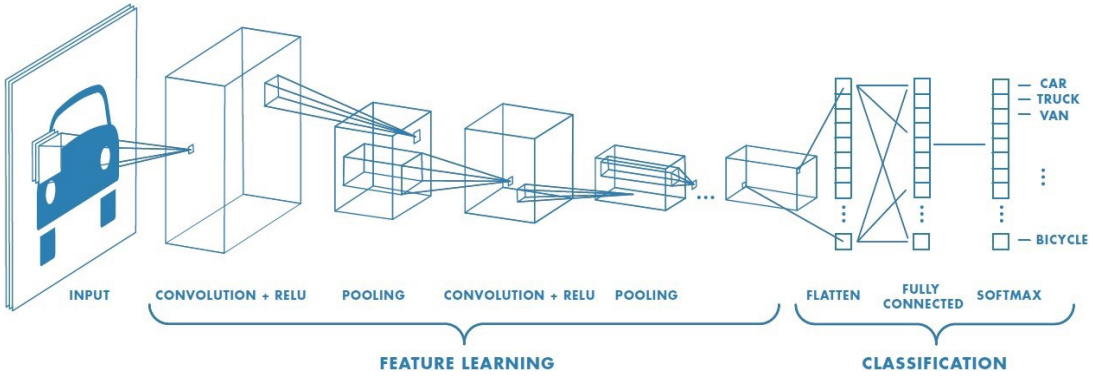

Figure 2. Schematic representation of a CNN using convolution layers with activations and pooling layers consecutively. The end classification is done by a feed forward neural network. Image retrieved from *Towards Data Science* [7]

that the actual values in the kernel are also a design choice. Indeed, different kernels will filter the image in different ways: there are kernels that blur or sharpen the input image, kernels chosen to detect the edges of the input image and many others. The beauty of a CNN is that the values of the kernel are learnable parameters! Through gradient descent, the values in the kernel -the weights- are learned so that the kernel extracts the features maps that are most "useful" to the network. Biases can be used in CNNs in much the same way they are used in neural networks, but in deep networks they are very often ignored, and empirically it seems it makes very little difference.

The input image might have multiple channels, for example when the input is an RGB image. In these cases, the kernel is three-dimensional and has the same depth as that of the input image. The size of the resulting feature map is independent on the number of channels of the input image -as long as the input image is two-dimensional, the feature map is also two-dimensional. What happens if the input "image" is a 3D volume, perhaps having also multiple channels? The kernel will in this case be four-dimensional, and the resulting feature map will be a 3D volume. It's hard to picture 4D convolution, but mathematically nothing much changes.

In case this wasn't complicated enough: we usually filter the input image by using several learnable kernels. This way, the network can learn several representations (feature maps) of the input image in one layer. This also means that one convolution layer will output usually output several feature maps -as many as kernels we use.

Any neural network needs a non-linear activation function to satisfy the Universal Approximation Theorem, and CNNs are not an exception. An activation function -ReLu and its variations are typical choices- is applied to each element of each feature map produced by the convolution step.

2. *Downsampling layer*

The downsampling layer reduced the size of the feature maps produced by the convolutional layer. It has two main roles:

- It reduces the size of the filter maps, making it possible to process the data with less computational resources.
- It summarises the feature map, which makes the model more robust to translational variations in the input image i.e. it helps the network recognise an object no matter where in the image it appears.

The most common way of downsampling in CNN is using a pooling function, more specifically max pooling. It is a simple operation: A window is slid trough the input feature map -just as the kernel was in the convolution layer- and the largest value in the portion of the feature map covered by the window is chosen. The output of the max-pooling layer depends on the window size and the stride length. With a window size of 2x2 and a stride length of 2, both common choices, the output feature map height and width are half of the input's. The reason why max-pooling is so popular is that it has the added benefit of acting as a noise filter. The intuition behind this is that max-pooling chooses the largest activation in the feature maps and discards the other "noisy" activations. Other pooling layers, like average pooling, do not have this benefit. Max-pooling is also a computationally cheap operation. There is another way of downsampling that, while not commonly used in 'vanilla' CNNs, is used in a CNN-based model called UNet, which we will discuss in a coming section.

3. Putting it together

Figure 2 shows a schematic representation of a CNN. First, a convolution layer with ReLu as the activation function is applied to the three channel input. Next, a pooling layer is used. This combination of convolution + activation + pooling is repeated several times. This step is called feature leaning or feature extraction. Note that the depth of the blocks symbolizes the number of features, that increase after each convolutional layer. The resulting feature maps are flattened and passed on to the classifier, a feed forward neural network.

B. UNet

As we've seen, feature extraction is what makes CNNs such powerful tools in Computer Vision. It is a way of learning an abstract representation of an image, which can then be used to classify it, but also to do *Semantic Segmentation* i.e. a pixel-wise classification. In 2015, Ronneberger et al. revolutionized the world of semantic segmentation by introducing a novel CNN-based architecture called UNet [10]. UNets consists of of two parts: an encoder, which learns a feature representation of the input image, and a symmetric decoder, that uses these feature representations to create the segmentation. The name UNet comes from the shape of the schematic representation of this model, shown in figure 3. While there are many UNet variations, I will base my discussion on the UNet I have used in this project. In this section I will give an overview on the different components of this UNet. More in-depth details are given in section IV B.

1. Encoder

The encoder is shown as the downward path in figure 3. The goal of the encoder is to extract feature maps from the input images. It consists of a user-defined number of layers, which include a convolution to extract features, and activation layer (or several) and a way to downsample the feature maps. While the first UNet used a max-pool layer for that propose, later UNet architectures, including the one I use in this project, use convolution with a stride length of 2. The size of the resulting feature maps is the same as it would be if we were to use max-pooling with the same stride length, i.e. a stride length of 2 reduced the output dimensions by half. Unfortunately, I haven't been able to find a satisfactory answer as to why downsampling via convolution would be preferred. Empirically however, it does seem to preform better in medical imaging segmentation, and indeed, a model similar to the one implemented in this project has won this year's BraTS challenge [11]. After the last downsampling layer, we typically have a large number of small feature maps.

2. Decoder

The decoder upsamples the feature maps and creates the output segmentation. The upsampling is done by transpose convolutions, which increase the size of the features (which ultimately become the output segmentation). In a transform convolution, the input image is enlarged by adding padding. We then convolve the modified input image with a kernel, making sure that we choose the kernel size, stride length and padding to the original image so that the output image is larger than the input feature map. Just as in normal convolution, the weights in the kernel are a trainable parameter. The network effectively learns the best way to upsample the feature maps with the ultimate goal of producing an output segmentation as similar as possible to the ground truth segmentation. However, using only the last feature maps produced by the encoder would give a very fuzzy output segmentation, as these feature maps have little positional information. To solve this issue, UNet architectures use *skip connections* where the feature maps produced by the encoder are concatenated with the upsampled features in the decoder. Then, convolutions followed by activation functions are applied to these feature maps, to create feature maps that contain information on not only classification, but also position. Finally, in the output layer, a convolution is used to reduce the number of channels (features) to the number of classes, in our case three.

III. DATASET

The data I use in this work is the public dataset provided for the 2016 and 2017 Brain Tumor Segmentation (BraTS) challenges[2]. It consists of 750 MRI scans of patients diagnosed with Glioblastoma (or a lower-grade Glioma). Each scan consists of four 3D images (volumes) of the patients brain taken with different MRI settings -so-called sequences. These four sequences are: T1-weighted (T1), post-Gadolinium contrast T1-weighted (T1c), T2-weighted (T2) and Fluid-Attenuated Inversion Recovery (FLAIR). To understand this project it suffices to think of these four sequences as images with different

[2]The dataset was downloaded from `https://drive.google.com/drive/folders/1HqEgzS8BV2c7xYNrZdEAnrHk7osJJ--2`

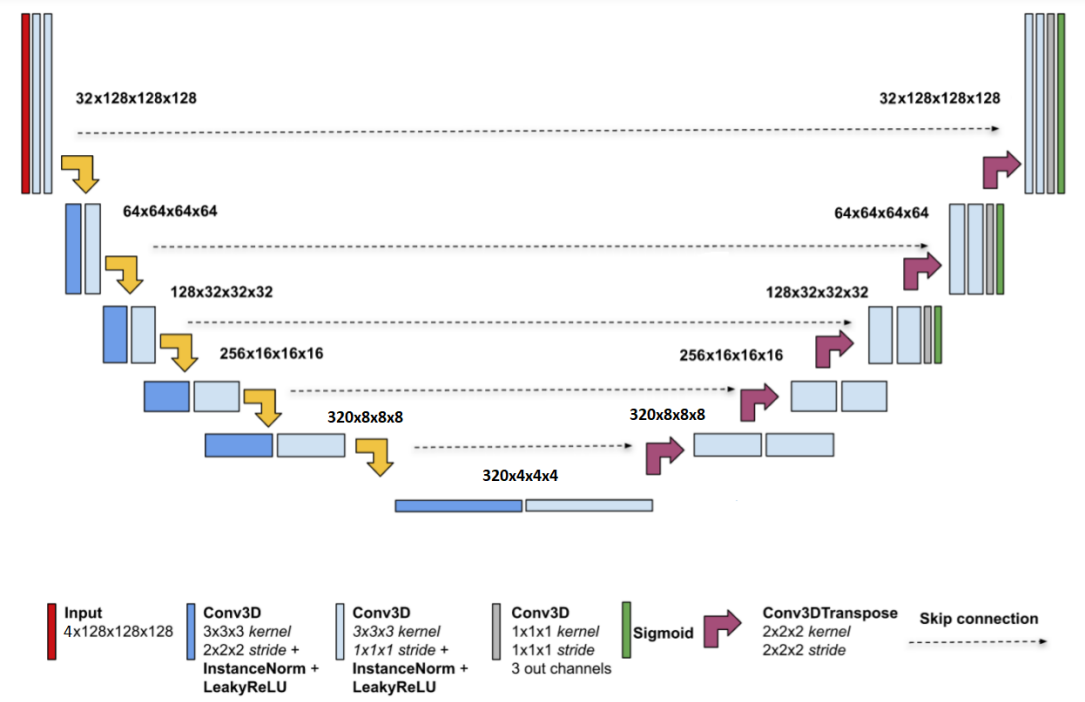

Figure 3. Schematic representation of the UNet used in this project. Image adapted from Futurega et al.[11]

kinds of "filters" that they show different parts of the tumor. While the raw volumes recorded by the MRI machines have varying resolution, in the BraTS dataset all volumes have been interpolated to 240 × 240 × 155 pixels. All volumes have also been skull-striped to protect the privacy of the patients. The data is acquired from 19 different hospitals in an effort to capture the range of differences in the MRI volumes expected in the "real world". The data is split into 484 scans to train and 266 to test the network with. [12]

The dataset also includes 750 ground-truth segmentations outlined by radiologists -one for every patient. The radiologists can use all four sequences to come up with a tumor segmentation, which includes three classes: enhancing tumor, necrotic core and edema. To sum up: The dataset consists of data from 750 patients, each with four volumes and one expert segmentation with 3 classes.

IV. IMPLEMENTATION

The code in this project uses the PyTorch library to implement data handling and model training. I have also used the MONAI library, a PyTorch-based framework for deep learning in medical imaging, mainly for data preprocessing and model implementation.

A. Data pre-processing and augmentation

The first step of data pre-processing is to stack all four sequences for each patient. Thus, we have a 4x240x240x155 size tensor for each patient. Next, we remove the background by cropping the borders of the image until the first non-zero voxel is encountered. This reduces the size of the image without reducing the image quality. Since our network requires all input images to be the same size, we have to interpolate all images to the same size, in this case 4x128x128x128. These steps are also done on the ground truth segmentation so that it still matches the input data. The next step is to normalize the data. This is done individually for each channel and each patient by computing the mean and the standard deviation, excluding the background voxels, and then removing the mean and dividing by the standard deviation for all non-background voxels. We choose not to normalize the background such that the value of those voxels remains zero. All these steps are applied to both the train and the test datasets.

Last but not least, we change the segmentation classes to make the training process simpler. The classes we use to train the model are: Whole Tumor (WT), which includes the enhancing tumor, the necrotic core and the edema, Tumor Core (TC), which includes the necrotic core and the enhancing tumor, and the Enhancing Tumor (ET). Each of these three classes is given a channel so that the ground truth segmentation consists of a a binary

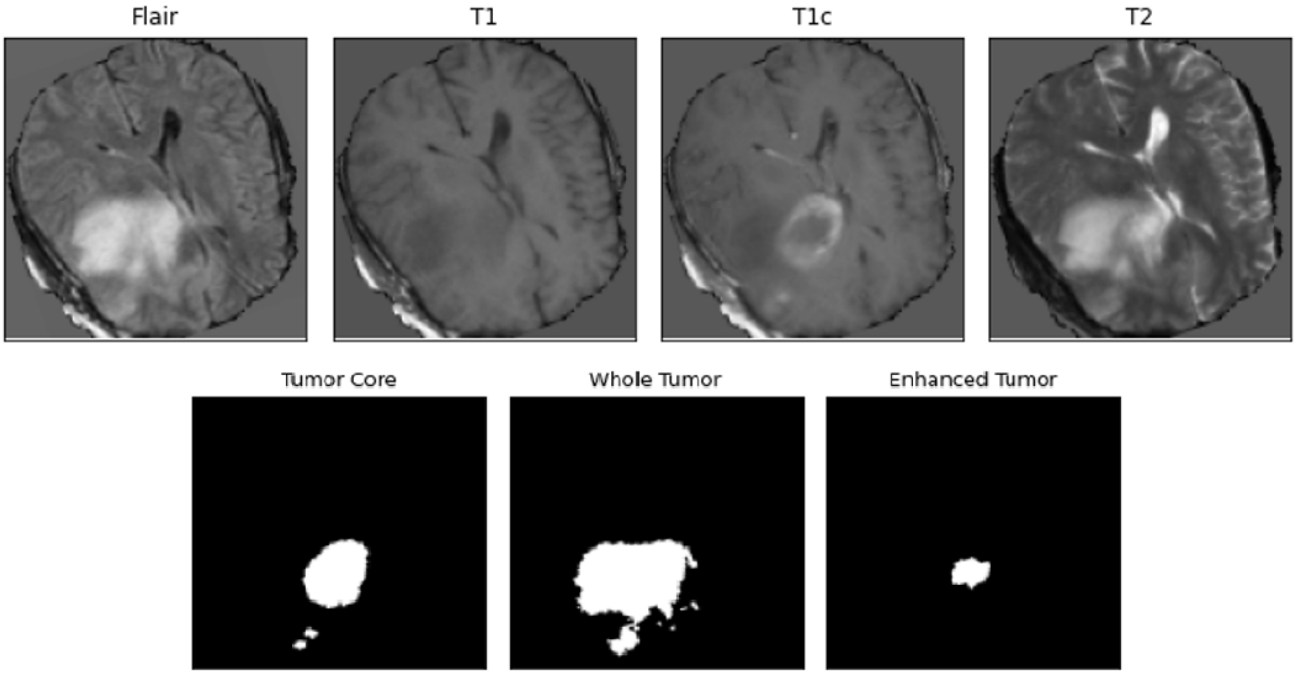

Figure 4. The top row shows one slice of the input data after pre-processing and augmentation. The bottom show shows the corresponding ground truth segmentations.

3x128x128x128 tensor.

Data augmentation is the process of transforming the input data so as to artificially expand the dataset during the training phase. This prevents over-fitting and can make a model more robust to changes in the input data. In this project, I have chosen to compare a model trained with data augmentation to the same model trained without data augmentation. These are the data augmentation techniques I have used:

- Randomly flip the volumes along the x and y axis with a probability of 0.5. The same flip is applied to the ground truth.
- Scale the intensity of input image by $v = v(1 + f)$ where f is a random value $\in (-0.1, 0.1)$. That is, the intensity is randomly increased or decreased by up to 10%.
- Shift the intensity of the input image with a factor times the standard deviation of the image such that $v = v + f \cdot std(v)$, with f being a random value $\in (-0.1, 0.1)$.
- Randomly rotate the input image along the z axis and apply the same rotation to the ground truth. Use zero padding outside grid values.
- Apply a random shear of up to 0.2 radiants (equivalent to 11 degrees). Apply the same shear to the ground truth. Use zero padding outside grid values.

Figure 4 shows an example of input data after pre-processing and augmentation, with the bottom row showing the corresponding ground truths for all three classes. Note that while only one slice is shown, the inputs to the model are volumes.

B. Model implementation

Figure 3 shows the UNet implementation used in this project. The model is implemented in MONAI. I used the DynUNet class (Dynamic UNet), which is a flexible UNet implementation. I have based the choice of the network parameters on previous work on UNet segmentation, specially the work of Isensee et al. [13] and Futrega et al. [11].

In the first layer of the encoder, the four-dimensional input (4 channels with 128x128x128 volumes) is passed through two convolutional layers, each with 3x3x3 kernels and 1x1x1 stride. Note that in this network, because of the input dimensions, all convolutions and transpose convolutions are done in three dimensions. After each of these convolutions, we preform instance normalization followed by passing the output through a Leaky ReLu

with a negative slope set to 0.01. We will call this convolution, instance normalization and activation block a convolution block. The outputs of this layer are 32 feature maps with size 128x128x128 (both convolutions are done with 32 kernels). In the second layer, we start by downsampling the feature maps by half. To that end a convolution with 3x3x3 kernels and 2x2x2 stride is used, followed by instance normalization and LeakyReLU. Next use a convolution block with 3x3x3 kernels and 1x1x1 stride. The feature maps are doubled in this layer, with the output being 64 feature maps of 64x64x64 size (64 kernels are used in each convolution). We use the same downsampling and convolution block layer four more times, decreasing the volume size and increasing the number of features at each layer (but the last, where we no longer increase the number of features because of computational constrains). At the bottom of the UNet we have 320 features of 4x4x4 size.

The first decoder layer consists of a transpose convolution that upsamples the feature maps to 8x8x8 in size. Next, the feature maps with the same size from the encoder are concatenated. Two consecutive convolution blocks with 320 3x3x3 kernels and 1x1x1 stride are applied. The output thus remains 320 features maps with 8x8x8 in size. This same layer is repeated, this time using 256 kernels. The same layer, with decreasing number of kernels, is repeated four more times. In the last layer, a simple convolution with three 1x1x1 kernel and 1x1x1 stride is used to reduce the number of feature maps from 32 to 3, which is the number of output classes. A Sigmoid activation function and a 0.5 cutoff is applied, so that the output consists of three binary channels of 128x128x128 size, which was the size of the input.

This network is somehow special in that it uses deep supervision. This means that the feature map outputs from the two layers before the output layer also go through the same steps as the output layer in order to create the segmentation maps: convolution with 3 1x1x1 kernel and 1x1x1 stride, Sigmoid and a 0.5 cutoff. We thus have not one but three segmentation outputs that we can calculate the loss from. Lee et al. finds that deep supervision can help, even in relatively shallow networks such as this, by acting as regularization [14]. Empirically, Futrega et al. show that having two outputs to deeply supervise the training archives the best results [11].

C. Training and validation

The goal of our network is to produce segmentations that are as similar as possible to the ground truth segmentations. To that end we will use the Dice score, a widely used metric that measures how similar two images are, to evaluate the network. Given a predicted segmentation map p and a ground truth segmentation map y the Dice score is defined by

$$D_{score} = \frac{1}{i}\sum^{i}\frac{y_i p_i}{y_i + p_i} \tag{1}$$

where the index i runs over the segmentation classes. In our case, $i = 3$ and the Dice score is an average of the individual Dice scores for each class. Thus, when the images are exactly equal, $D = 1$ and when the segmentations do not overlap at all, $D = 0$.

A simple modification makes it possible to use the Dice score as a loss:

$$D_{loss} = 1 - D_{score} \tag{2}$$

The loss function used to train this network is an average of the Dice losses for the three outputs -the output segmentation map and the two outputs for deep supervision. Weighting them equally has empirically been found to give the best results [11].

Adam with a weight decay of $1e-5$ is used as the optimizer. In order for the network to converge in a reasonable time-frame, the starting learning rates have to be relatively high. Thus, a learning rate scheduler is used which reduces that starting learning rate by 0.05 every 75 epochs. The network is trained for 200 epochs with a mini-batch size of 3. This is the highest batch size that the memory of the GPU this network was trained on could cope with -the memory usage is a whooping 10.5GB even with such a small batch size. Validation on the test dataset is preformed after each epoch, and the trained model is saved if the test validation Dice score is higher than the previous highest Dice score.

In order to accelerate training, I use the automatic mixed precision package provided in PyTorch. It automatically detects whether an operation needs the dynamic range using float32 gives (typically backpropagation), and uses float16 for the remaining operations (for example convolutions).

V. RESULTS AND DISCUSSION

Ideally, we would do careful hyperparameter tuning for the UNet with and without augmentation, allowing us to get the highest scores possible with these models. However, the computational complexity of the network makes this an impossible persuit. Training a simple network took between 17 and 24 hours, depending on whether we used augmentation or not (and also which other processes where running on the CPU, as loading data to the GPU seemed to be the bottleneck). This means that I

was only able to train four networks, two with augmentation and two without. I choose to change the initial learning rate, as this is a parameter that tends to have a very significant impact on the convergence. Table I shows the maximum average Dice scores attained on the test dataset for each of the model and learning rate combinations. It also shows the epoch number at which that maximum average Dice score was attained.

Model	*Max Dice*	*Epoch*
Augmentation, lr=1e-3	0.760	138
Augmentation, lr=5e-4	0.767	139
No augmentation, lr=1e-3	0.777	133
No augmentation, lr=5e-4	0.780	93

Table I. Maximum evaluation Dice scores for each model. The epoch number at the max evaluation Dice score is also shown.

Somewhat unexpectedly, the network without data augmentation preforms better than the network with augmentation, as measured by the Dice score. Augmentation accounts for by far the largest difference in the scores, with the initial learning rate having very little effect on both the Dice scores and the epoch at which the best score is attained. Why would augmentation not only not help, but actually decrease the network's accuracy? It seems like the way I have augmented the input images does not reflect the variation in the real data. Perhaps using the same augmentations with less "strength" -lower the factor for the intensity changes and the angles for the rotation and shear- would have helped. Another option to try would be to lower the probability of the data being augmented (it is set to one in most of my augmentation functions). There are also many other augmentation functions that might reflect the variation in the real data more accurately. Adding noise and doing elastic transformations have been shown to work for a similar dataset and model [10]. Another relevant question is whether the test dataset reflects the real world data a model would be applied to. Because the BraTS dataset is public, it is often used to train models with the hope that they can be used in clinical practice. The best thing would be to use transfer learning for this purpose -train first on BraTS and then keep training the model on hospital-specific data. However, the reality is that 1) getting access to patient data is a complicated process and 2) the clinicians using the model probably don't have the knowledge or time to train it. Ideally, we would have a trained model that works out of the box. That's obviously a tall order, but in such a context data it could be advantage to use strong data augmentation *even when* it hurts the performance of the model on the test dataset available. A more "general" model -higher bias, lower variance- has the potential to do better on significantly different datasets.

Figure 5 shows the evaluation Dice scores at each epoch for the best preforming network with and without augmentation, with an initial learning rate of 5e-4 which, as

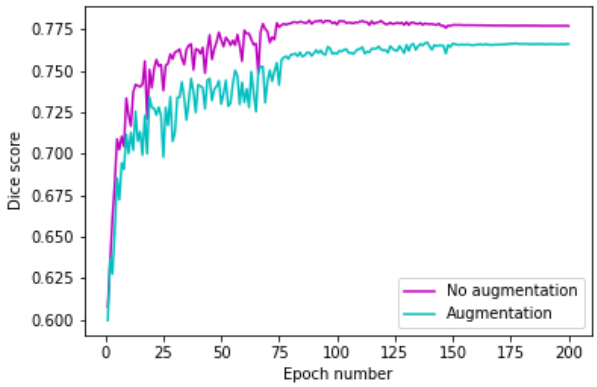

Figure 5. Evaluation Dice scores at each epoch with and without data augmentation. The initial learning rate is set to 5e-4.

shown in table I, preforms slightly better than the higher initial learning rate. The model without data augmentation preforms better at all epochs. The Dice scores for both models show a significant decrease in variability after epoch 75, and again after epoch 150. This is due to the learning rate being decreased at those epochs. The model without data augmentation shows some signs of over-fitting after epoch $\sim$ 100, after which the test Dice score shows a slight decrease as the training continues. However, this effect is much smaller than the overall difference in the Dice scores between the models. Also, using the model at the epoch with the highest test scores instead of using the model at the end of training helps avoid over-fitting.

It is also interesting to look at the average Dice scores for each class individually. Figure 6 shows the average Dice scores for each class for every epoch. At the epoch with the overall highest Dice score, epoch 93, the Dice scores for each class are: WT=0.910, TC=0.830, ET=0.601. The models does *much* better at segmenting the entire tumor that it does in segmenting the Tumor Core, and the Enhanced Tumor (which is a part of the TC) is even harder to segment. There are two reasons for this. First and foremost, separating tumor from brain tissue is a relatively straight forward task, while the differences between the different tumor tissue sub-types are much more intricate. Indeed, experienced radiologists also struggle with this task. Secondly, the classes are imbalanced. There are many more voxels in the WT class than in the ET, and not all images have ET at all. There is simply less data to train the algorithm to segment ET than TC and specially WT. Changing the loss to account for this class imbalance, for instance by giving more weight to the loss for the ET class, could help improve the segmentation accuracies for this class.

Last but not least we will look at the actual segmen-

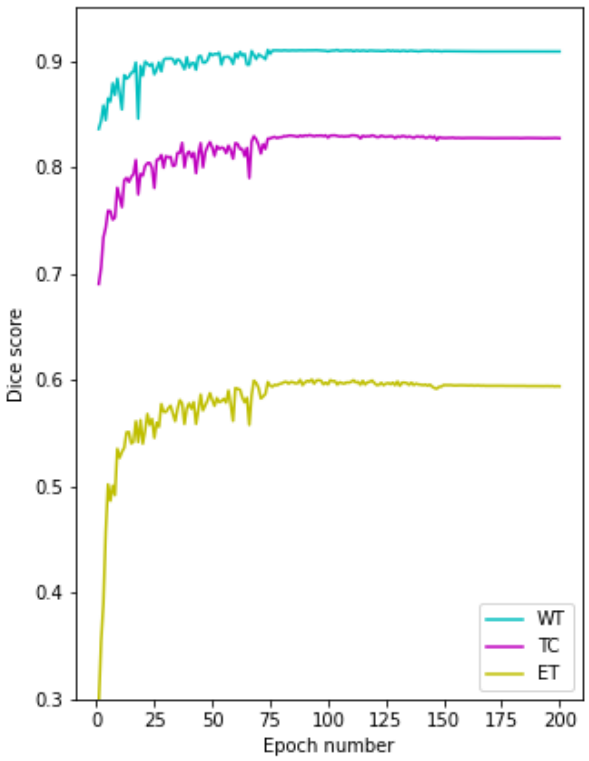

Figure 6. Evaluation Dice scores at each epoch for each of the three classes Whole Tumor (WT), Tumor Core (TC) and Enhancing Tumor (ET). using the model without data augmentation and an initial learning rate of 5e-4.

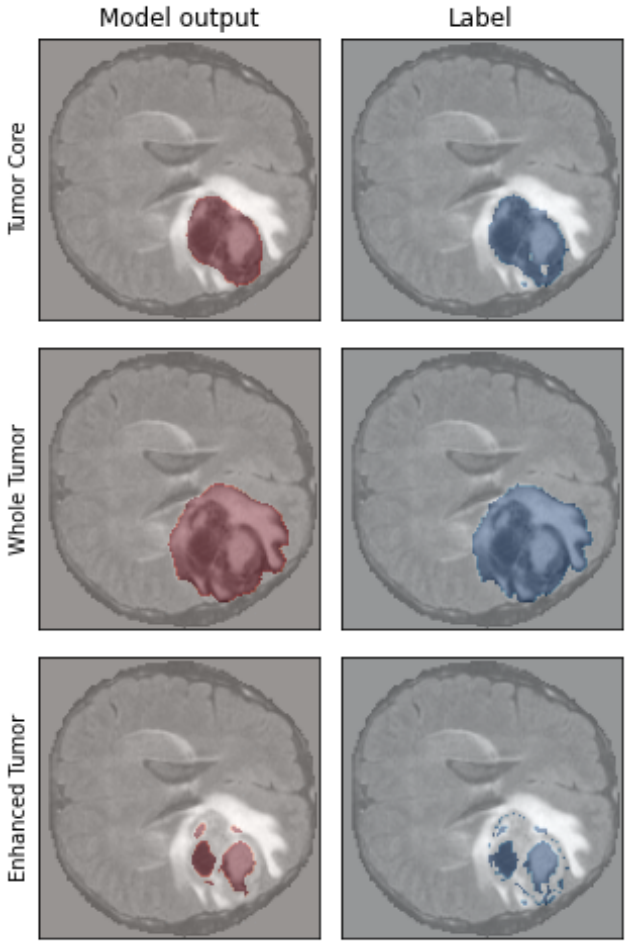

Figure 7. Output from the best preforming model shown side by side with the ground truth segmentation (label) for patient nr. 10. A Flair sequence is used as the background to give context to the segmentations.

tations produced by our best preforming trained model. Figure 7 shows the output segmentations for each class on the left, and the ground truth segmentation on the right for one patient. The segmentation are superposed on the Flair image to give context. Visually, the WT segmentation appears almost perfect. The only difference seems to be that the edges of the model output segmentation are somewhat smoother than the ground truth edges. The same "smoothing out" problem is present in the TC and specially in the ET, where the model misses a pixelated ring of this class around the mail ET segmentation. However, how similar would other radiologist's segmentations be compared to our ground truth? It would be interesting to have several expert segmentations of the ET to see what the real difference between this model and human segmentations is. Figure 8 shows the segmentations for another patient. In shows a case in which the model strongly over-segments the TC and ET classes. The WT class is still segmented quite accurately. [3]

VI. CONCLUSION

In this project I have implemented a UNet capable of segmenting brain tumors in MRI volumes. The goal was to experiment with data augmentation, which is commonly used in medical imaging, but often without testing its efficacy. The results show that the best-preforming network trained with data augmentation preformed worse than the one trained on non-augmented

[3] More example segmentations can be found at `https://github.com/lidialuq/fys-stk4155/tree/main/Project3/figures`

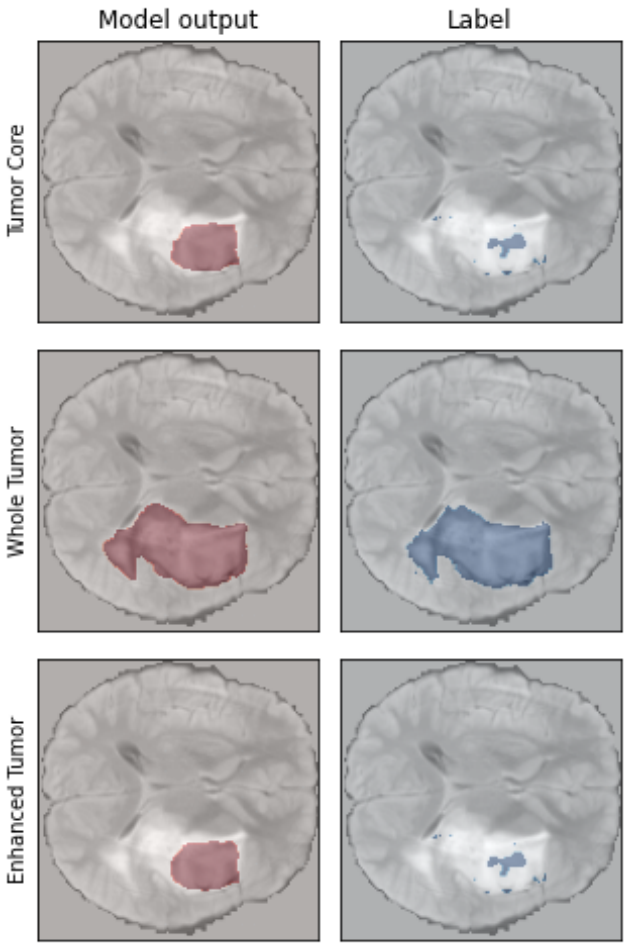

Figure 8. Output from the best preforming model shown side by side with the ground truth segmentation (label) for patient nr. 50. A Flair sequence is used as the background to give context to the segmentations.

data. The best mean Dice scores achieved was 0.768 with data augmentation and 0.780 without. The model without data augmentation also trained faster, achieving the best mean Dice score after only 93 epochs compared to 139 for the model with augmentation. Data augmentation is also computationally heavy, adding several hours of training time. Due to long training times I was not able to test other forms of data augmentation. It is likely that lighter data augmentation and the use of other, more realistic data augmentation functions would produce better results.

The best preforming model, a UNet trained with a starting learning rate of 5e-4 without data augmentation, preforms very well for the Whole Tumor class, but poorly on the Enhanced Tumor class. The achieved Dice scores are 0.910 for Whole Tumor, 0.830 for Tumor Core, 0.601 for Enhanced Tumor. Class imbalance likely plays a role in this difference. Future work should explore using a weighted Dice loss or other weighted losses to counteract this imbalance.

[1] A. Shergalis, A. Bankhead, U. Luesakul, N. Muangsin, and N. Neamati, "Current challenges and opportunities in treating glioblastoma," *Pharmacological reviews*, vol. 70, no. 3, pp. 412–445, 2018.

[2] A. A. of Neuro-Surgeons, "Glioblastoma multiforme."

[3] M. Visser, D. Müller, R. van Duijn, M. Smits, N. Verburg, E. Hendriks, R. Nabuurs, J. Bot, R. Eijgelaar, M. Witte, *et al.*, "Inter-rater agreement in glioma segmentations on longitudinal mri," *NeuroImage: Clinical*, vol. 22, p. 101727, 2019.

[4] E. S. Biratu, F. Schwenker, Y. M. Ayano, and T. G. Debelee, "A survey of brain tumor segmentation and classification algorithms," *Journal of Imaging*, vol. 7, no. 9, 2021.

[5] F. Hoftun, A. Ravn, and L. Luque, "Neural networks, logistic and linear regression for data classification and regression problems," pp. 1–6, 11 2021.

[6] M. Hjorth-Jensen, "Applied data analysis and machine learning, fys-stk3155/4155 at the university of oslo, norway," 2021.

[7] S. Saha, "A comprehensive guide to convolutional neural networks the eli5 way," 2018.

[8] D. into Deep Learning, "Convolutions for images."

[9] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, pp. 1097–1105, 2012.

[10] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," in *International Conference on Medical image computing and computer-assisted intervention*, pp. 234–241, Springer, 2015.

[11] M. Futrega, A. Milesi, M. Marcinkiewicz, and P. Ribalta, "Optimized u-net for brain tumor segmentation," 2021.

[12] M. Antonelli, A. Reinke, S. Bakas, K. Farahani, AnnetteKopp-Schneider, B. A. Landman, G. Litjens, B. Menze, O. Ronneberger, R. M. Summers, B. van Ginneken, M. Bilello, P. Bilic, P. F. Christ, R. K. G. Do, M. J. Gollub, S. H. Heckers, H. Huisman, W. R. Jarnagin, M. K. McHugo, S. Napel, J. S. G. Pernicka, K. Rhode, C. Tobon-Gomez, E. Vorontsov, H. Huisman, J. A. Meakin, S. Ourselin, M. Wiesenfarth, P. Arbelaez, B. Bae, S. Chen, L. Daza, J. Feng, B. He, F. Isensee, Y. Ji, F. Jia, N. Kim, I. Kim, D. Merhof, A. Pai, B. Park, M. Perslev, R. Rezaiifar, O. Rippel, I. Sarasua, W. Shen, J. Son, C. Wachinger, L. Wang, Y. Wang, Y. Xia, D. Xu, Z. Xu, Y. Zheng, A. L. Simpson, L. Maier-Hein, and M. J. Cardoso, "The medical segmentation decathlon," 2021.

[13] F. Isensee, J. Petersen, A. Klein, D. Zimmerer, P. F. Jaeger, S. Kohl, J. Wasserthal, G. Koehler, T. Norajitra, S. Wirkert, *et al.*, "nnu-net: Self-adapting framework for u-net-based medical image segmentation," *arXiv preprint arXiv:1809.10486*, 2018.

[14] C.-Y. Lee, S. Xie, P. Gallagher, Z. Zhang, and Z. Tu, "Deeply-Supervised Nets," in *Proceedings of the Eighteenth International Conference on Artificial Intelligence and Statistics*, vol. 38 of *Proceedings of Machine Learning Research*, (San Diego, California, USA), pp. 562–570, PMLR, 09–12 May 2015.