Sebastian Borgeaud dit Avocat

Brain tumour segmentation using Convolutional Neural Networks

Computer Science Tripos – Part II

Fitzwilliam College

March 31, 2017

Proforma

Name:	**Sebastian Borgeaud dit Avocat**
College:	**Fitzwilliam College**
Project Title:	**Brain tumour segmentation using Convolutional Neura**
Examination:	**Computer Science Tripos – Part II, June 2017**
Word Count:	**INSERT**
Project Originator:	Duo Wang
Supervisor:	Dr. Mateja Jamnik & Duo Wang

[1]This word count was computed by `detex diss.tex | tr -cd '0-9A-Za-z \n' | wc -w`

Declaration

I, Sebastian Borgeaud dit Avocat of Fitzwilliam College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed [signature]

Date [date]

Contents

1 Introduction [14%] 9
1.1 Motivation . . . 9
1.2 Related Work . . . 10
1.3 Supervised learning and classification . . . 11

2 Preparation [26%] 13
2.1 Starting point . . . 13
2.2 Theoretical background . . . 14
2.2.1 Artificial neural networks . . . 14
2.2.2 Convolutional neural networks . . . 17
2.2.3 The overfitting problem . . . 19
2.3 Data source . . . 22
2.3.1 BraTS Challenge . . . 22
2.3.2 Data used in this project . . . 23

3 Implementation [40%] 25
3.1 Data pre-processing . . . 25
3.1.1 Patch extraction . . . 25
3.1.2 Data augmentation . . . 27
3.1.3 Normalisations . . . 28
3.2 Pereira model . . . 31
3.2.1 Architecture . . . 31
3.2.2 Implementation of the architecture in Keras . . . 32
3.2.3 Training . . . 33
3.3 My model . . . 34
3.3.1 Architecture . . . 34
3.3.2 Implementation . . . 35
3.3.3 Training . . . 36
3.4 Segmentation . . . 37
3.4.1 Pereira model . . . 37
3.4.2 My model . . . 38
3.5 Data post-processing . . . 39

4 Evaluation (+ Conclusion [20%]) 43
4.1 BraTS evaluation 43
4.1.1 Regions of evaluation 45
4.2 Evaluation of the model proposed by Pereira et al. 46
4.3 Evaluation of my model 48
4.4 Comparison 48

5 Conclusion 49
5.1 Summary of achievements 49
5.2 Future Work 49

Bibliography 49

List of Figures

2.1 Structure of a simple neural network with two hidden layers. 14
2.2 Plot of the activation functions. 16
2.3 Example computation done by a 3x3 filter in a convolutional layer 19
2.4 Example slices in the axial plane for the 4 different scan modalities 23
2.5 Slices of a T1 MRI scan annotated with the expert labelling. 24

3.1 Patch extraction process for a three dimensional array. 26
3.2 Example of the effects of winsorising and applying the N4ITK correction applied to scans for each of the four modalities. The first image in each row is the original scan, the middle image is obtained by winsorizing the original image and the last image is obtained by applying the N4ITK correction to the winsorised scan. From top to bottom, slice $z = 89$ is shown for patient 1 for the T1, T1c, T2 and Flair scans. 30
3.3 Convolutional network architecture proposed by Pereire et al. 32
3.4 Part of the output of the training script 34
3.5 Example segmentation computed using the model proposed by Pereira et al. 38
3.6 Effect of applying the post-processing step 41

4.1 Venn diagram showing the different areas and how the can be classified. . . 43
4.2 Example showing a segmentation for a slice in the axial plane. This image was reproduced from [2] 45
4.3 Online evaluation ranking. My submission was ranked 42nd TODO WRONG REDO SCREENSHOT FOR THE CORRECT SUBMISSION . . 48

Acknowledgements

Chapter 1

Introduction [14%]

1.1 Motivation

The glioma is most frequent type of brain tumours in adults. As their name suggest these arise from glial cell and infiltrate the surrounding tissue. Goodenberger and Jenkins [4] reported that gliomas make up to 30% of all brain and central nervous system tumours and 80% of all malignant brain tumours. These gliomas are further divided into low-grade and high-grade, depending on their pace of growth. While the low-grade gliomas come with a life expectancy of several years, the clinical population with the more aggressive form of the diseases, the high-grade glioma, have a median survival rate of two years or less [1]. For both groups, intensive neuroimaging protocols are used both before and after the treatment to evaluate the progression of the disease and the success of the treatment. It is therefore clear that image processing algorithms that can automatically analyse tumour scans would be of great value for improved diagnosis, treatment planning and evaluation of the tumour progression. Developing these automated brain tumour segmentation algorithms is technically challenging as lesion areas are only defined by changes in intensity relative to the surrounding normal tissue. Tumour size, extension and localisation also vary across patient, making it hard to incorporate and encode strong priors on shape and location which are often used in segmentations of anatomical structures. The problem of automatic brain tumour segmentation is therefore still an active research area. [2].

In recent years, convolutional neural networks have been shown to significantly outperform other methods in many areas such as Computer Vision (FaceNet[16]), image classification (AlexNet [12]), Natural language processing [18] and many others. The field of bioinformatics and medical imaging is no exception to this, in particular convolutional neural networks have been shown to perform as well as pervious state-of-the-art methods and even to outperform them on the problem of brain tumour segmentation [14] [9]. Thus convolutional neural networks have also become part of the state-of-the-art methods in brain tumour segmentation. The aim of my project is to understand and try to replicate one of the more recent techniques using convolutional neural networks applied to the problem of brain tumour segmentation. I chose to replicate a method proposed by Pereira et al.[14] in 2016 which is based on two-dimensional convolutional neural networks. In the second phase of my project, I experimented with different network architectures to explore what effect different changes would make to the performance of the network. In this dissertation I propose a different architecture.

I chose this project as it would allow me to explore the field of machine learning from a more practical point of view by working on a still open-ended but critical problem whilst having a clear starting point. The second reason is chose this project is that there is an open dataset freely provided by the BraTS Conference [2].

1.2 Related Work

Brain tumours segmentation methods can be divided into methods based on generative models and methods based on discriminative methods.

Generative methods rely on domain-specific prior knowledge, and combine that knowledge with the appearance of a new scan to detect anomalies. They therefore often generalise well to previously unseen images. However, encoding such prior knowledge is very difficult. For example, Prastawa et al. [15] used the ICBM brain atlas to detect abnormal regions. A post-processing step is then applied to ensure that these tumour regions have good spatial regularity.

On the other hand, discriminative method use very little prior knowledge on the brain's anatomy. After the extraction of low level image features using manually annotated images, discriminative models learn directly how to model the relationship between these features and the label of a given voxel. The features used vary across methods, for example Hamamci et al. [5] used raw input pixel and Kleesiek et al. [11] used local histograms. The methods usually train a classifier that relies on these had-designed features. These features are assumed to have a sufficiently high discriminative power so that the classifier can learn to separate the tissue into the appropriate classes. The problem with these hand-designed feature is that by their nature they are very generic, with no specific adaptation to the domain of brain tumours. Ideally, these low-level features should be composed into higher-level and more task-specific features. Convolutional neural networks are believed to work in this way, starting from the raw data input extracting increasingly more complex features using the features computed by the previous layer in the network.

Zikic et al. [20] first proposed a convolutional neural network with two convolutional layers followed by a fully-connected layer. The inputs to the network were patches of size 19×19 taken from slices of the scans. Thus, only a two-dimensional convolutional neural network is required which is much more efficient than a three-dimensional one and is less prone to overfitting. Continuing on this approach, Havaei et al. [6] used a novel double-pathway architecture and a cascade of two networks to rank 2$^{\text{nd}}$ on the BraTS2015 challenge. Novel in their approach was also the training of the network in two phases, first sampling patches from an equiprobable distribution before training only the fully-connected layers on data sampled using proportions near the originals. Pereira et al. [14] proposed a deeper two-dimensional convolutional neural network of 11 layers, that ranked 1st in the BraTS2013 and BraTS2015 challenges. As the aim of my project is to replicate their method, I will explain it in more details in the Preparation and Implementation chapters. Finally, it is worth noting that some of the more recent approaches have successfully overcome the difficulties involved in training three-dimensional neural networks. In particular the model proposed by Kamnitsasa et al. [10] used a three-dimensional convolutional neural network with residual connections [7] to take the first place in the BraTS2015 challenge.

1.3 Supervised learning and classification

Broadly speaking, we can divide machine learning into supervised and unsupervised learning. In unsupervised learning, the task is to learn a function to describe some hidden structure from unlabelled data. On the other hand, in supervised learning, the task is to infer a function $f : A \to B$ from a set of n labelled data points $\mathbb{X} = \{(\mathbf{a}_1, b_1), ..., (\mathbf{a}_n, b_n)\}$. Typically, if $A \subset \mathbb{R}^d$ and $b_i \in \mathbb{R}$ then the task is also called **regression** and if B is discrete, the task is called **classification**. In either case, we generally define an appropriate loss function $\mathcal{L} : B \times B \to \mathbb{R}$ and we aim to minimise the loss over some test dataset $\sum_{i=1}^{n} \mathcal{L}(f(\mathbf{a}_i), b_i)$ and hope that the learned function f will be able to generalise well on unseen data.

The brain tumour segmentation task can be seen as an example of supervised learning, more specifically of a classification problem. We are given a set of MRI scans segmented into one of five classes:

1. Non-tumour, which includes areas outside the brain and normal tissue inside the brain
2. Surrounding edema, which is swelling caused by excess fluid trapped in the brain's tissue
3. Necrotic tissue, which consists of dead cells
4. Non-enhancing tumour
5. Enhancing tumour

We can therefore reformulate the segmentation problem as the problem of learning a function f that can map each pixel in the input scan to one of the 5 classes. If we then apply this function f to every voxel, we have effectively created a segmentation.

In the Preparation chapter I will explain how this function is modelled by convolutional neural networks and how the data is structured. Then, in the Implementation chapter I will go over how the data is transformed into a format that can be used and how the convolutional neural network proposed by Pereira et al. [14] is implemented. I will also present the architecture of a model that my supervisor and I came up with. Then, I will explain in more details how the model is used to segment a scan. Finally, I will detail how the post-processing step based on connected components is implemented. In the evaluation chapter, I will go over how we can compare segmentations in a quantitative and meaningful way and will evaluate both models.

Chapter 2

Preparation [26%]

During the first phase of my project, the aim was to replicate the method used by Pereira et al [14], so it was crucial to first fully understand the steps taken in the paper to then be able to reimplement them. Unfortunately, the paper didn't include any source code which meant that if something wasn't fully explained in details, I would have to find out what was actually done. This turned out to be a problem for the pre-processing step as the proposed method uses a normalisation developed by Nyul [CITATION]. This normalisation requires human input, preferably from a domain expert, and I was therefore not able to use that normalisation method. The paper also proposed a second normalisation method, which used a combination of winsorizing and N4 normalization. This method performed slightly worse but had the advantage of being fully automated, which is why I chose to use this method instead.

2.1 Starting point

(I have seen such a section in some of the previous part II disserations, but not in all of them. I guess it would be good to include it as it puts the project in context with my previous knowledge and would show how much I have learned.)

The starting point for this project was mainly the part IB course 'Artificial Intelligence I'. In particular, neural networks, backpropagation and stochastic gradient descent were introduced, concepts also used in convolutional neural networks. Secondly, the course also introduced the general concept of Machine Learning and more specifically, formalised the task of supervised learning.

Second, I was able to use some of the material taught by the part II course 'Machine Learning and Bayesian Inference', especially the parts on evaluation of classifiers and on general techniques for machine learning.

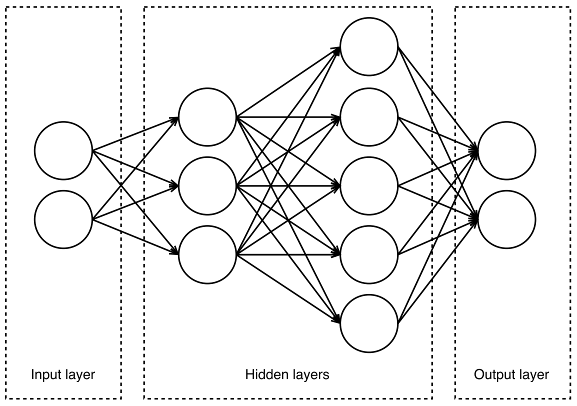


Figure 2.1: Structure of a simple neural network with two hidden layers.

The remaining theory was learned through self study at the start of the project, using as main resource the excellent Stanford course 'CS231n Convolutional Neural Networks for Visual Recognition'[1].

The project was almost entirely written in Python, except for a few bash scripts that I had to create as jobs for the Cambridge High Performance Cluster. I had only used Python before for very small, single file projects and had to learn some of the more advanced syntax. I also heavily used the Numpy library, which I hadn't used before. Thanks to the good documentation availalbe online that wasn't a problem.

I also used the Keras library, which makes it easy to create and train convolutional neural networks while leaving all important design and architectural decisions to the user. This was extremely helpful, as this project wouldn't have been possible (at least to such an extend) without it. Furthermore, as Keras is becoming the standard open-source library in deep learning research and applications, many online resources and posts could be found when needed.

2.2 Theoretical background

2.2.1 Artificial neural networks

To understand how convolutional neural networks work, it is important to be familiar with ordinary neural networks. These are made up from a sequence of layers of neurons, each neuron having a set of trainable weights that can be adjusted to change the overall function computed by the neural network. An example of the structure of such a neural network can be found in figure 2.1.

Each neuron in layer $n + 1$ is connected to every neuron in layer n and computes as

[1]`http://cs231n.github.io/`

an output

$$y = f_{act}((\sum_{i=1}^{n} y_i w_i) + b)$$

where f_{act} is a non-linear, differentiable activation function and y_i is the output of neuron i in the previous layer. A neuron is connected to every neuron in the previous layer, which is why this layer is also referred to as a fully connected layer.

Activation functions

The most common activation functions are the Sigmoid function,

$$S(x) = \frac{1}{1 + e^x}$$

the hyperbolic tangent

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

the rectifier

$$f(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{otherwise} \end{cases}$$

and the leaky rectifier, for some $0 < \alpha < 1$

$$f(x) = \begin{cases} \alpha x & \text{if } x < 0 \\ x & \text{otherwise} \end{cases}$$

This functions can be seen plotted in figure 2.2. Historically, the hyperbolic tangent function or the sigmoid function have been used as activation functions. However, as the magnitude of the gradients of those functions is always below 1, these activation function create a problem of vanishing gradients for deeper networks, as we have to multiply those gradients together for each layer. [CITATIONS!!!!] This is why it is nowadays preferred to use the rectifier or the leaky rectifier functions, especially with deeper architectures.

In a classification problem, the output layer will consist of K nodes, one for each class. Using a softmax activation function for the last layer, we can view the output of node k as the probability $P(y^{(i)} = k \mid \mathbf{x}^{(i)}; \theta)$ since the the output of each neuron will range between 0 and 1 and the sum of all outputs will be 1. The softmax activation computes for each output k

$$\sigma(\mathbf{x})_k = \frac{e^{\mathbf{x}_k}}{\sum_{j=1}^{K} e^{\mathbf{x}_j}} \tag{2.1}$$

where $\mathbf{x}$ is the vector consisting of all outputs from the previous layer.

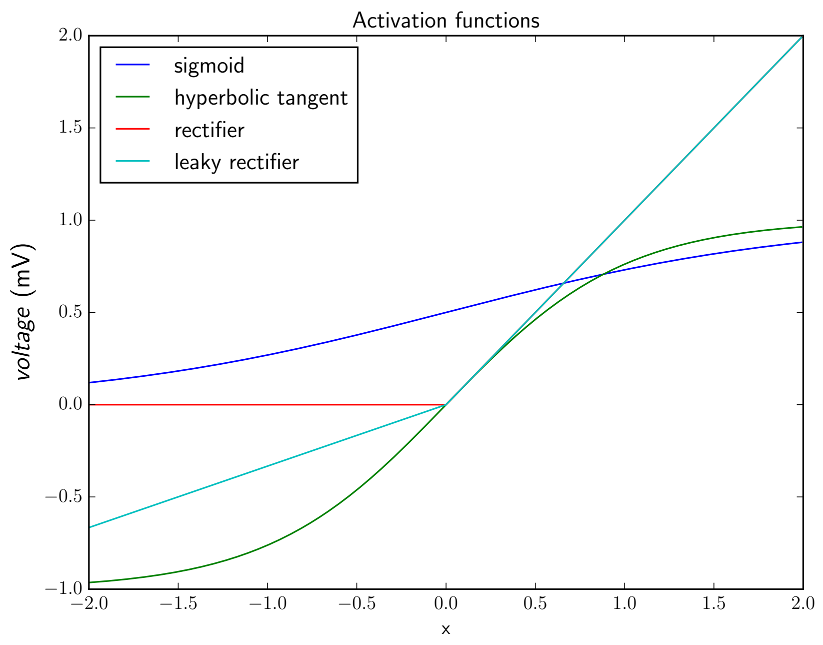


Figure 2.2: Plot of the activation functions.

Loss function

The next step is to compute how well our network approximates our training data with a loss function, to then decide how to change the weights of the network in order to minimise the loss. Since the aim of the network is to classify the central pixel(s) of the patch, we use the categorical cross-entropy loss function

$$\mathcal{L}(\theta) = \frac{1}{m}\Big[\sum_{i=1}^{m}\sum_{j=1}^{k} \mathbb{1}[y^{(i)} = k] \log(P(y^{(i)} = k \mid \mathbf{x}^{(i)}; \theta))\Big] \tag{2.2}$$

where $\mathbb{1}$ is the indicator function and $P(y^{(i)} = k \mid \mathbf{x}^{(i)}; \theta)$ is the probability computed by the neural network that input vector $\mathbf{x}$, in our case a patch, belongs to class k.

Optimisation

The next step is to calculate the gradient $\frac{d\mathcal{L}(\theta)}{d\theta}$, so that we can apply gradient descent and update our weight vector to

$$\theta = \theta - \epsilon\frac{d\mathcal{L}(\theta)}{d\theta} \tag{2.3}$$

where ϵ is the learning rate, a small positive value.

Since every basic operation used in the neural network is differentiable, the entire network will also be differentiable, which in turn makes it possible to calculate the gradient of a loss function with respect to the weights in the network. This process is called backpropagation.

The loss functions, as described in equation 2.2 is computed using the entire training data set $\mathbb{X} = \{(\mathbf{x}^{(1)}, y^{(1)}), ..., (\mathbf{x}^{(m)}, y^{(m)})\}$. In the case of deep learning where it is usual to have a very large training data set, this would be very memory costly and slow down the training phase unnecessarily. Therefore, stochastic gradient descent is used instead, where we process the training data sequentially in batches, each time computing the loss for that batch and updating the weights.

Momentum

A further optimisation used to speed up the training process is momentum update. Minimising the loss function can be interpreted as moving a small particle down a hilly terrain in the hyper-dimensional space defined by the loss function. Since the gradient is related to the force experienced by that particle, this suggest that the gradient should only influence the velocity vector and not the position directly. This leads to the velocity update

$$v = \mu v - \epsilon \frac{d\mathcal{L}(\theta)}{d\theta} \tag{2.4}$$

where μ is the momentum, typically set to 0.9. We then update our weights by simply adding the velocity to the current value.

$$\theta = \theta + v \tag{2.5}$$

Typically a slightly different version, called the Nesterov momentum, is used as it has been shown to work better in practice[CITATION].

2.2.2 Convolutional neural networks

Convolutional neural networks are different as they make the explicit assumption that the inputs will be images. This allows us to take advantage of some properties to make the forward function more effective and greatly reduce the number of weights in our network.

A typical convolutional network constists of three types of layers: **convolutional layers**, **pooling layers** and **fully-connected layers**.

Fully-connected layers

These are exactly the same as for ordinary neural networks.

Convolutional layers

Unlike a fully-connected layer, a convolutional layer is typically three-dimensional: widht, height and depth. The parameters of a layer are a set of learnable filters, each spatially small along the width and height but with the depth equal to the depth of input volume. The layer then computes a two-dimensional activation map by convolving the filter with the input and computing the dot product at each point. This means that the function learned by a filter has translational invariance, as the filter is convolved with the entire input and thus the same feature is detected independently of location. Each filter computes such a two-dimensional activation map that can then be stacked along the depth axis to produce the output volume.

The connectivity pattern is inspired by the organization of the animal visual cortex. Individual neurons respond to stimuli in a small region of space known as the **receptive field**. Every element in the output can then be interpreted as the output of a neuron whose receptive field is the width and height of the filter and who shares its weights with all its neighbours to the left and right spatially.

The size of the output volume is determined by three hyperparameters.

1. The **depth** corresponds to the number of filters in the layer and is therefore equal to the depth of the output volume.
2. The **stride** which determines by how many pixels we slide the filter. When the stride is greater than 1 the output width and height will be smaller than the input width and height.
3. The **padding** determines with how many zeros we pad the input width and height. This is particularly useful when we want to preserve the input dimensions.

The computation done by such a filter in shown in figure 2.3.

Pooling layers

The function of the pooling layer is to reduce the spatial size of the input layer in order to reduce the number of parameters and computation in the network. The most common pooling layer implementation is the **max-pooling** which just convolves a two-dimensional maximum operator of a given size, typically 2x2. The **stride** again determines the step size.

A convolutional neural network typically consists of a sequence of these layers, starting with pairs of convolutional neural networks and max pooling layers. The idea behind this is that each pair of layers can learn more and more abstract features using the features

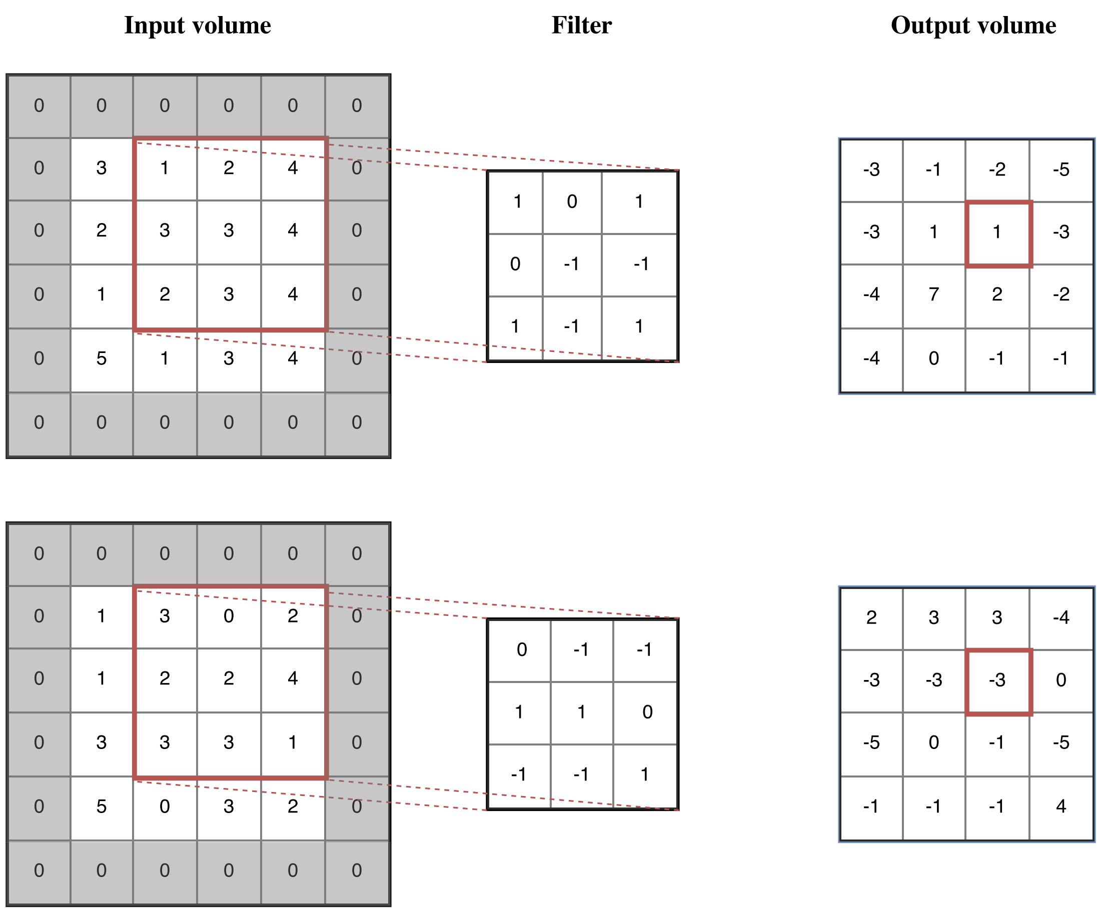


Figure 2.3: Example computation done by a 3x3 filter in a convolutional layer. Both the stride and the padding are 1, so as to keep the output dimensions identical. The output at depth 0 for the marked element is $1\cdot1+2\cdot0+4\cdot1+3\cdot0+3\cdot-1+4\cdot-1+2\cdot1+3\cdot-1+4\cdot1 = 1$ and similarly at depth 1 the output for that element is -3.

learned by the previous layer. For example, the first pair might learn to recognise edges, the second layer shapes, etc.. The last few layers of the network consist entirely of fully-connected layers. These learn how to classify the data using the features learned by the convolutional and max pooling layers.

2.2.3 The overfitting problem

A common problem that arises with neural networks models is that of **overfitting**, that is when the model isn't able to generalise on previously unseen data and instead just memorises the training data. Due to the large number of weights in convolutional neural networks, these are particularly prone to this problem. Many techniques and heuristics have been developed in order to help reduce overfitting. In my project I have used three of them: **L2 weight normalisation**, **Dropout** and **Batch Normalisation**

L2 weight normalisation

The intuition behind this normalisation is that by preventing individual parameters to grow without bounds, the model must learn how to smoothly extract features of the data and this in turn will prevent the model from just memorising the data. This is achieved by adding a regularisation penalty to the loss function. There are many different regularisation costs one could use, the most common ones are the L2 and L1 normalisation. In my project, I used the L2 normalisation which computes the sum of the squares of every parameter:

$$R(\theta) = \sum_k \sum_l \theta_{k,l}^2 \tag{2.6}$$

where θ is the weight matrix that includes all weights, including biases. The loss function that we aim to minimise thus becomes

$$\mathcal{L}(\theta) = \mathcal{L}_{data}(\theta) + R(\theta) \tag{2.7}$$

where $\mathcal{L}_{data}(\theta)$ is the data loss, defined in equation 2.2.

Dropout

A more recent technique, developed for deep neural networks is Dropout [17]. The key idea is that the network randomly drops some neurons during the training phase. The computation done by a single neuron thus becomes

$$\begin{aligned} \mathbf{r} &\sim \text{Bernoulli}_n(p) \\ y &= f_{act}((\sum_{i=1}^{n} r_i y_i w_i) + b) \end{aligned} \tag{2.8}$$

where $\mathbf{r}$ is a vector of n independent Bernoulli random variables each with probability p of being one, that is $P(r_i = 1) = p$ and $P(r_i = 0) = 1 - p$. This is done at every layer and there amounts to sampling a sub-network from a larger network. At test time dropout is not applied. Thus, the network weights have to be scaled by a factor p to compensate for the extra inputs at each layer. The back-propagation algorithm remains unchanged for the sub-network and can therefore be applied in the learning phase.

It is common in practice to apply Dropout only to the fully connected layers and not to the convolutional layers, as it has been shown to produce better results. As I will show later, this is also what was done by Pereira et al. [14].

Batch Normalisation

The final regularisation technique used is Batch Normalisation [8]. As this technique is more recent than the paper published by Pereira et al., it was not used in their method.

However, in the second phase of my project I experimented with a different architecture in which I used Batch Normalisation.

The key idea is that each layer in the neural network experiences *internal covariance shift*, by which is meant the change in the distribution of the inputs due to the change in parameters of the network. This complicates the training of the neural network because each layer has to continuously adapt to this change in distribution. By fixing the distribution of these inputs the network is able to learn faster and is less prone to overfitting. Similarly to how the inputs to the network are often normalised to have mean 0 and variance 1, it would be beneficial to ensure that the input vector $\mathbf{x}$ to each layer has mean 0 and variance 1.

To make this efficient and differentiable, which is required for the minimisation of the loss function, each individual dimension of the input vector $\mathbf{x}^{(k)}$ is normalised using the mean and standard deviation of the training mini-batch. In the same way as the mini-batch is used as an approximation to calculate the gradient of the loss function on the entire training set, the mini-batch is used to approximate the mean and variance of the entire training set at different layers in the network. However, just normalising each input of a layer might restrict which functions the layer is able to represent. The output is therefore linearly scaled by γ and shifted by β, parameters that can be learned along with the weights. This ensures that the introduced transformation is able to represent the identity transformation, if that is the optimal thing to do. The Batch Normalisation transformation computes:

$$\begin{aligned}
\mu_{\mathbb{B}} &= \frac{1}{m}\sum_{i=1}^{m} x_i^{(k)} \\
\sigma_{\mathbb{B}}^2 &= \frac{1}{m}\sum_{i=1}^{m} (x_i^{(k)} - \mu_{\mathbb{B}})^2 \\
\hat{x}_i^{(k)} &= \frac{x_i^{(k)} - \mu_{\mathbb{B}}}{\sqrt{\sigma_{\mathbb{B}}^2 + \epsilon}} \\
\mathrm{BN}_{\gamma,\beta}(x_i^{(k)}) &= \gamma \hat{x}_i^{(k)} + \beta
\end{aligned} \tag{2.9}$$

where $\mathbb{B}$ is a minimatch of size m, $\mathbb{B} = \{\mathbf{x}_1, ..., \mathbf{x}_m\}$ and ϵ is a numerical constant added to increase numerical stability. In convolutional neural networks the Batch Normalisation transformation is applied just before the activation function. The output computed by each neuron therefore becomes

$$y = f_{act}(\mathrm{BN}_{\gamma,\beta}(\sum_{i=1}^{n} y_i w_i)) \tag{2.10}$$

where the Batch Normalisation transformation is applied to each dimension individually. Also notice that since the parameter β is added, the bias weight b is made redundant. The testing phase has to be modified accordingly, the details of which can be found in [8].

2.3 Data source

2.3.1 BraTS Challenge

The dataset I am using comes entirely from the BraTS [13] challenge. It is split into three sections:

1. The training dataset, which consists of 30 different patients and their ground truth marked by a human experts. The 30 patients are further divided into 20 high-grade glioma cases (HG) and 10 low-grade glioma cases (LG). The difference between these two types of brain tumours is their rate of growth, which is slower for the low-grade case.
2. The challenge set, which consists of 10 high-grade patients without the ground truth. These scans are meant to be segmented by the participants of the challenge, who can then submit their segmentation online.
3. The leaderboard set contains 25 patients, including both high-grade and low-grade gliomas. The ground truth labels are not available.

The challenge and leaderboard sets were both used to rank the participants of the original BraTS conference. After the conference, to create a benchmarking resource, an online platform was made available that automatically calculates the scores (Dice score, PPV and Sensitivity) of submitted segmentations. Part of the evaluation of my project will be comparing the results of my model with those made by other researchers using this online platform.

Each patient consists of 4 images taken using different MRI contrasts: T2 and FLAIR MRI which highlight differences in tissue water relaxational properties, T1 MRI which shows pathological intratumoral take-up of contrast agents and T1c MRI. Each of these modalities shows different type of biological information and may therefore be useful for creating different features during the classification of the tissues. Figure 2.5 shows the 4 different modalities for a slice in the axial plane for a patient. Figure **??** shows different slices of a T1 MRI scan annotated with the ground truth labelling created by a group of 4 experts.

To homogenise the data across the different scans, each patient's image volumes were co-registered to the T1c MRI scan, which has the highest spatial resolution in most cases. Then, all images were resampled to 1mm isotropic resolution in a standardised axial orientation with linear interpolation. Finally, all images were skull stripped to guarantee the anonymity of the patients.

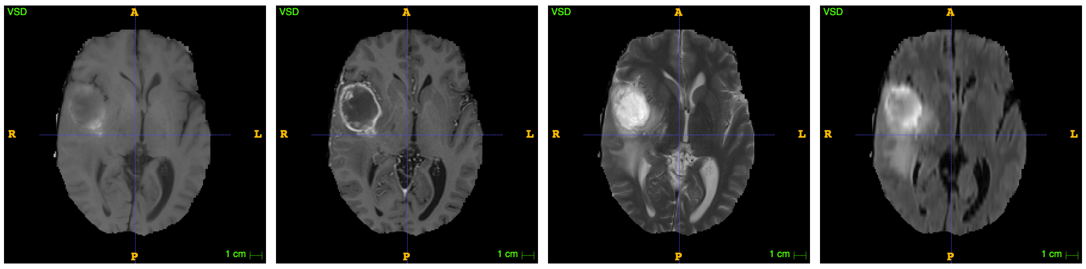


Figure 2.4: Example slices in the axial plane for the 4 different scan modalities. From left to right the modalities are T1, T1c, T2 and Flair. The scan is from patient 1, slice $z = 89$. The four modalities have some obvious differences which the convolutional neural network might be able to utilise to discriminate between tumour and non-tumour regions.

2.3.2 Data used in this project

For this project I have decided to focus only on the high-grade glioma cases. This decision was mainly made to keep the scope of this project reasonable. Secondly, the results of most research papers in this area are commonly reported in terms of high-grade glioma cases as it is considered to be harder than the low-grade cases. Lastly, the challenge dataset, which consists only of high-grade patients allowed me to compare easily the results of my models with those of other researchers. Thus only the 20 high-grade glioma cases of the training dataset were used as training data.

To provide some guidance on how well the network is performing during the training phase, I also used 10 high-grade glioma cases from the BraTS2015 dataset, which contains more patients prepared in a similar way. These 10 patients were used to provide the validation data.

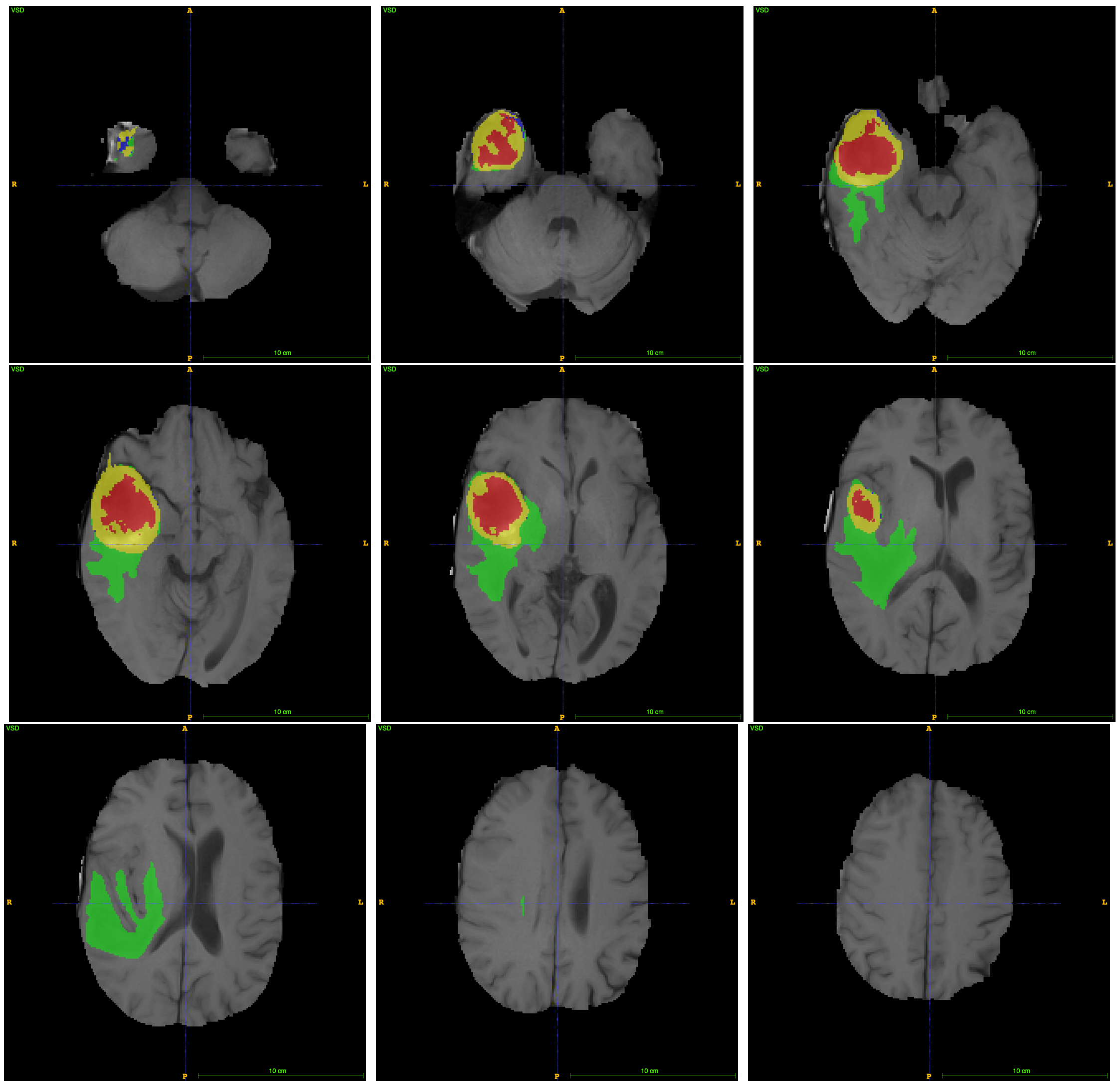


Figure 2.5: Axial plane slices of a T1 MRI scan annotated by the expert labelling. The slices were taken from patient 1, using $z \in \{49, 59, 69, 79, 89, 99, 109, 119, 129\}$. The enhancing tumour region is in yellow, the non-enhancing tumour in blue, the edema in green and the necrotic core in red, corresponding to classes 4, 3, 2 and 1 respectively.

Chapter 3

Implementation [40%]

3.1 Data pre-processing

The first step is to read in the scans, which is done using the SimpleITK library that has inbuilt support for the MetaImage medical ('.mha') format. For each patient the 4 scans (T1, T1c, T2, Flair) are read in as numpy arrays and stacked along a new dimension, resulting in a 4-dimensional array for each patient of shape $(z_{\text{size}} \times y_{\text{size}} \times x_{\text{size}} \times 4)$. I then aggregate those arrays into a single conveniently-formatted list containing all the training data.

3.1.1 Patch extraction

Each input to the neural network is a three-dimensional array, of shape $33 \times 33 \times 4$, since it consists of a two-dimensional patch of 33×33 voxels for each one of the 4 scan modalities. The two-dimensional patches are taken along the x–y axis, refer to as the axial plane in anatomy. Figure 3.1 gives an overview of this process for the simplified case with a single modality, meaning that the image array is three-dimensional and the resulting patch is two-dimensional.

As explained in the previous chapter the data consists of 20 different patients, each consisting of 4 different MRI scans, taken with different modalities. Unfortunately, the size of the different scans varies from patient to patient. Table 3.1.1 shows how the sizes are distributed among patients. Since we are working with individual patches, this discrepancy in scans sizes between patients has no impact, as long as each voxel has the same spatial resolution. This can be seen as one of the advantages of a model based on patches compared to a model based on slices or even entire MRI scans, as it offers more flexibility on the size of the MRI scans.

The dataset used is also extremely unbalanced, which would cause issues if the model

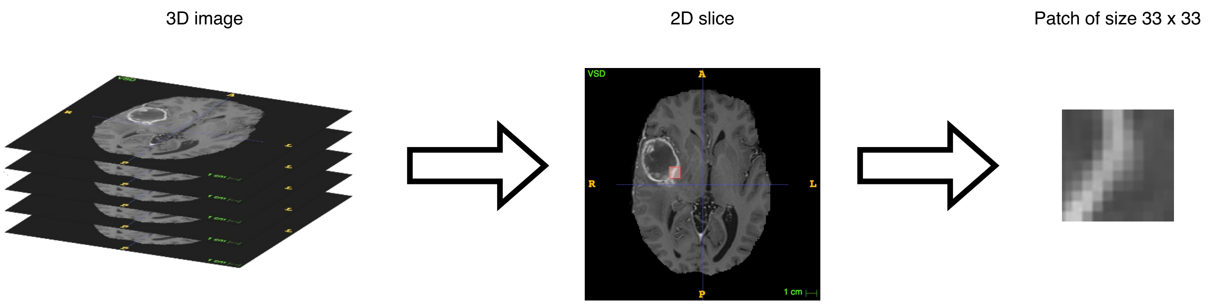


Figure 3.1: Patch extraction process for a three dimensional array. The actual image array is four dimensional, the 4th dimension being the different modalities.

Size	Number of scans
$176 \times 216 \times 160$	8
$176 \times 216 \times 176$	6
$230 \times 230 \times 162$	2
$176 \times 236 \times 216$	1
$165 \times 230 \times 230$	1
$240 \times 240 \times 168$	1
$220 \times 220 \times 168$	1

Table 3.1: Scan sizes in voxels for the 20 different patients, dimensions are ordered as $(z \times y \times z)$

was training using patches chosen uniformly at random from the scans. Table 3.1.1 shows the proportion of data available in each class. Because the training data for the convolutional neural network should be balanced, we need to extract the same number of patches for each class. Note that this further implies that the maximum number of patches we can extract is 5 times the number of voxels in the least represented class, which is the non-enhancing tumour class with 184,436 labeled voxels, for a maximum of 922,130 of patches.

A further question is how we should deal with a voxel that is to close to the edge of the scan to be able to extract an entire patch around it. I could either ignore the patch

Class	Tissue type	Number of labeled voxels	frequency
0	Non tumour	138 958 832	98.23%
1	Necrosis	282 936	0.20%
2	Edema	1 466 271	1.36%
3	Non-enhancing tumour	184 436	0.13%
4	Enhancing tumour	560 777	0.34%

Table 3.2: Class frequencies in the BraTS2013 HG dataset. The normal tissue (class 0) is highly overrepresented, which leads to issues when training the convolutional neural network. We therefore have to balance the dataset when extracting the patches.

or pad the scan such that each labelled voxel is surrounded by enough voxels. The first option is not only easier, but also helps with the balanced classes issue as most of the voxels close to an edge are from class 0. This is because the brains are centered within the scans. Table 3.1.1 shows the distribution of classes for "valid" voxels, .i.e those at least more than half the patch length away from the x or y edges.

Class	Tissue type	Labelled voxels	Frequency	Ignored voxels
0	Non tumour	94 773 429	97.45%	44 185 403
1	Necrosis	281 831	0.29%	1105
2	Edema	1 453 205	1.49%	13 066
3	Non-enhancing tumour	183 396	0.19%	1040
4	Enhancing tumour	558 320	0.57%	2457

Table 3.3: Class frequencies in the BraTS2013 HG dataset for valid voxels only, that is, those voxels it is possible to extract a patch of size 33×33 around. As most of the ignored voxels are in class 0, we can safely ignore them.

The patch extraction can now be performed. First I created a list for each class containing the positions of the valid voxels for that class. Note that at this point it would be unfeasible to store the patches, as it would require to store over a 100 billion 32 bit floats for class 0 alone, or over 400 GB. To get the indices of the valid voxels, I used numpy's argwhere function on the ground truth arrays. The argwhere function takes in an array and a predicate and returns the indices of those elements in the array satisfying the predicate. Here, the predicate used is just equality checking between the voxel label and the class number. This is done for every patient. Since the scan number must also be included in the indices, I prepended the scan number along the first axis, before aggregating the results for each patient into a list. The final result is a list for each class that contains tuples of the form (patient number, z, y, x). The second step consists of removing those voxels that are too close to an x-axis or y-axis edge. Only indices (patient, z, y, x) where the x and y values are within the allowed ranges of $[16, \text{size}-(16+1)]$ are kept. Again this can be done using numpy and filtering based on column values. Algorithm 1 summarises the steps taken during the patch extraction.

3.1.2 Data augmentation

To increase the amount of data available, some data augmentation techniques can be used. In typical applications of convolutional neural networks for image processing and computer vision tasks, translation and rotations can be used. Since the data consists entirely of two-dimensional patches, translation is useless as it would just result in a different patch, with a possibly different label. However, using rotations of the patches might give some performance improvements. Pereira et al. proposes different techniques in [14]:

Algorithm 1 Patch extraction

```
1: for class $k$ in $[0, 1, 2, 3, 4]$ do
2:     valid_indices[$k$] $\leftarrow$ []
3:     for each index, label in labels do
4:         possible_indices $\leftarrow$ *numpy*.argwhere($label == k$)
5:         patient_indices $\leftarrow$ *numpy*.full(*len*(possible_indices), index)
6:         possible_indices $\leftarrow$ *numpy*.append(patient_indices, possible_indices, axis = 1)
7:
8:         possible_indices $\leftarrow$ possible_indices[possible_indices$[:, 2] - 16 \geq 0$]
9:         possible_indices $\leftarrow$ possible_indices[possible_indices$[:, 2] + (16 + 1) <$ label.y_dimension]
10:        possible_indices $\leftarrow$ possible_indices[possible_indices$[:, 3] - 16 \geq 0$]
11:        possible_indices $\leftarrow$ possible_indices[possible_indices$[:, 3] + (16 + 1) <$ label.x_dimension]
12:
13:        valid_indices[$k$] $\leftarrow$ possible_indices
14:    end for
15: end for
```

1. No rotations
2. Rotations of 0, 90, 180 and 270 degrees.
3. Rotations of multiples (???)

Using rotations of multiples of 90 degrees performed the best, which is why I have decided to perform those. I also have investigate the effect on the results of using no rotations at all, the results are reported in the evaluation chapter (MAYBE).

Once again, Keras is extremely useful as it allows the creation of "ImageDataGenerators" which given two numpy arrays of the training data and labels, generates batches of augmented data in real-time during the training phase of the convolutional neural network. In particular it is possible to randomly flip the images vertically and horizontally, which was used to 'rotate' the images.

Note that, as opposed to the patch extraction and the normalisation phases, the data augmentation is only done for the training data and not for the validation or test data.

3.1.3 Normalisations

Winsorising

The first normalisation that is applied is called 'winsorising'. The aim is to limit the values of extremes, in order to reduce the effect that outliers may have. This is also known as ‘clipping’ in digital signal processing. I used a 98% winsorisation, meaning that the data values below the 1st percentile are set to the value of the 1st percentile and the value above the 99th percentile are set to the value of the 99th percentile. Note that this process is different from trimming as the values are not discarded but just modified.

N4 Bias field correction

The second normalisation applied is the N4 bias field correction [19]. N4 is a variant of the popular nonparametric nonuniform intensity normalisation N3. This family of corrections aim at eliminating the bias introduced by MRI scanners. This biases can be difficult for automated segmentation algorithms to handle because the intensity of the same tissue can vary across a scan.

As this correction is rather complicated and not the main point of my project, I will not go into more details here. However, for completeness, table 3.1.3 lists the parameters I used to perform the normalisation. I first used the default parameters and later emailed the author of [14] who kindly agreed to share the parameters he used in his published method. These are the parameters reported here. I used the implementation provided by the ‘Nipype’[1] and the ‘Advanced Normalization Tools’[2].

Parameter	**Value**
n_iterations	[20,20,20,10]
dimension	3
bspline_fitting_distance	200
shrink_factor	2
convergence_threshold	0

Table 3.4: Parameters used to perform the N4ITK correction.

Figure 3.1.3 shows the effects that winsorising and the N4 bias field correction have on scans for each of the four modalities T1, T1c, T2 and Flair.

Linear scaling

Then, each scan with different modality is individually scaled into the range $[0, 1]$. This is done to guarantee that the absolute values of the different scans are within the same range, making it possibly easier for the convolutional neural network to generalise the features it learns onto new data. In general to linearly scale a data point x into the range $[a, b]$, the formula described in equation 3.1 is applied, where x_{min} and x_{max} are the minimum and maximum value across the dataset to scale.

$$x' = a + (b - a)\frac{x - x_{min}}{x_{max} - x_{min}} \tag{3.1}$$

[1] `http://nipype.readthedocs.io/en/latest/`

[2] urlhttp://stnava.github.io/ANTs/

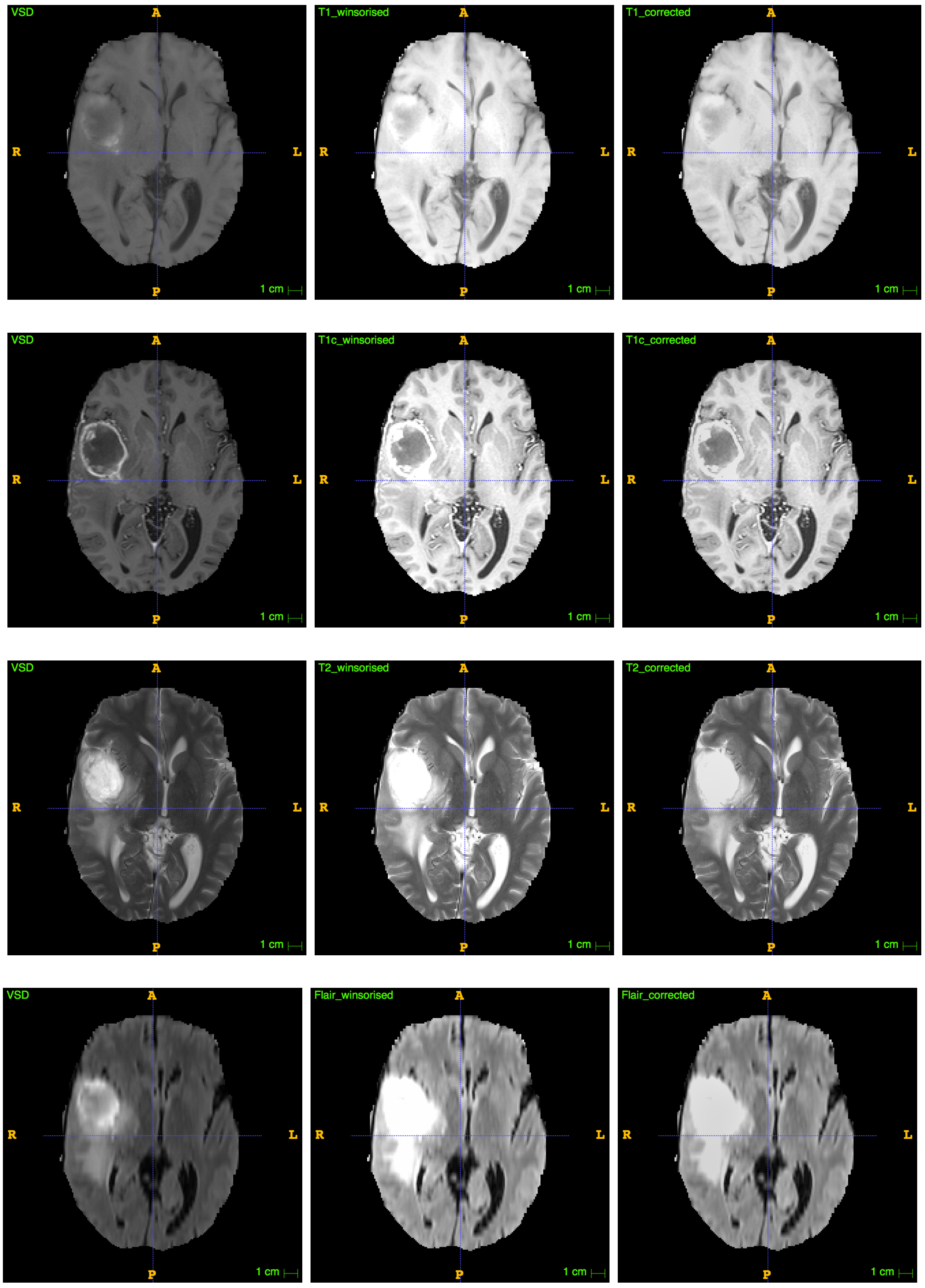


Figure 3.2: Example of the effects of winsorising and applying the N4ITK correction applied to scans for each of the four modalities. The first image in each row is the original scan, the middle image is obtained by winsorizing the original image and the last image is obtained by applying the N4ITK correction to the winsorised scan. From top to bottom, slice $z = 89$ is shown for patient 1 for the T1, T1c, T2 and Flair scans.

Since, I am scaling to the range $[0, 1]$ the formula becomes

$$x' = \frac{x - x_{min}}{x_{max} - x_{min}} \tag{3.2}$$

Using numpy's 'min' and 'max' function, we can perform this computation in a single line by using the fact that addition, subtraction and division is done element-wise.

Mean and Variance standardisation

The last step is to standardise the mean and variance for each scan. This is done by first subtracting the mean across all scan values and then dividing by the standard deviation.

$$x' = \frac{x - \mu}{\sigma} \tag{3.3}$$

As $\mathbb{E}[aX] = a\mathbb{E}[X]$ and $\text{Var}[cX] = c^2\text{Var}[X]$ it is easy to see that after this transformation the new mean will be 0 and the standard deviation will be 1. Also note that as the true mean and standard deviation are not known, the sample mean and sample standard deviation must be used.

Similarly as for the linear scaling, I used numpy's 'mean' and 'std' functions and the fact that basic operations are done element-wise to implement this normalisation.

3.2 Pereira model

3.2.1 Architecture

The model proposed by Pereira et al. [14] consists of 11 layers, shown in figure 3.3. The first three layers are convolutional layers with filter size 3×3, stride 1×1 and width 64. Three layers are used consecutively because the combination of them effectively has a receptive field of size 7×7 while having fewer parameters then a single 7×7 convolutional layer would have, making the network less prone to overfitting. [CITATION 36 in pereira]. The next layer is a max pooling layer of size 3×3 and stride 2×2. This is an unusual design as the size is larger than the stride. This is done because although pooling can be positive to eliminate unwanted details and achieve invariance, it can also eliminate some of the important details. By keeping the size of the layer larger than the stride, the pooling will overlap which will allow the network to keep more information about location. Next, three convolutional layers are applied, this time doubling the width to 128 filters. Again a max pooling layer with the same hyper-parameters as the previous one is applied, before adding three fully-connected layers of size 256, 256 and 5 respectively. Table 3.2.1 shows the number of weights in each layer.

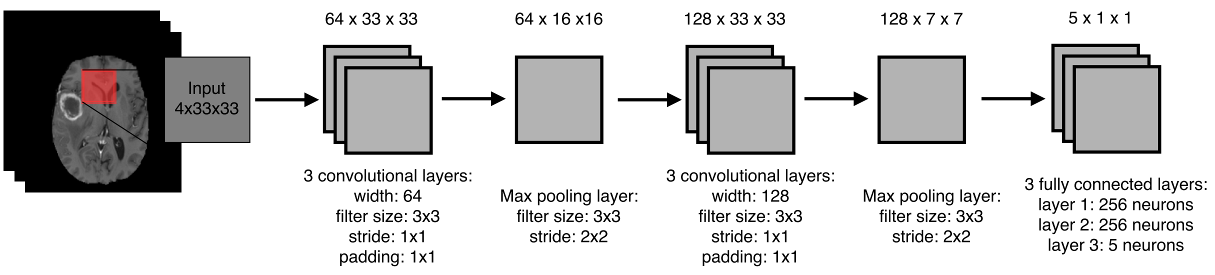


Figure 3.3: Convolutional network architecture proposed by Pereire et al.

Layer type	**Output shape**	**# Parameters**
Conv	$64 \times 33 \times 33$	2368
Conv	$64 \times 33 \times 33$	36 928
Conv	$64 \times 33 \times 33$	36 928
Max-Pool	$64 \times 16 \times 16$	0
Conv	$128 \times 16 \times 16$	73 856
Conv	$128 \times 16 \times 16$	147 584
Conv	$128 \times 16 \times 16$	147 584
Max-Pool	$128 \times 7 \times 7$	0
FC	$256 \times 1 \times 1$	1 605 888
FC	$256 \times 1 \times 1$	65 792
FC	$5 \times 1 \times 1$	1285
		2 118 213

Table 3.5: Summary of the architecture proposed by Pereira, including the number of parameters in each layer. The network has a total of 2,118,213 trainable parameters.

The initialisation of the weights in the convolutional layers was done using the Xavier initialisation [3]. The biases were initialised to the fixed constant value 0.1, except for the last layer.

The activation function used throughout the network is the leaky rectifier, with $\alpha = 0.333$. However, using a leaky rectifier over a rectifier has very little effect on the final results, as will be shown in the next chapter. ???

To avoid overfitting Dropout [] is used in the fully-connected layers, with probability $p = 0.1$ of removing a node from the network.

3.2.2 Implementation of the architecture in Keras

Since the architecture is entirely sequential, I used the keras 'sequential' model to implement it. The 'sequential' model makes it possible to create networks by adding layers to it sequentially, as the name suggests. The layers I had to use had available implementations, and therefore building the network itself was relatively straightforward. First, an instance a of model is created:

```
model = Sequential()
```

Then, layers can be added by calling the ‘add’ member function of the model. For example to add a convolutional layer, we first create an instance of the ‘Convolution2D’ class and then add it to the model:

```
conv = Convolution2D(nb_filters , size[0], size[1], border_mode='same',
  init='glorot_normal')
model.add(conv)
```

Note that the library makes it easy to specify how the weights should be initialised, by setting the ‘init’ argument. The ‘border_mode’ determines if padding should be added and how much. In our case we want the output of the layer to have the same size as the input. Max pooling layers and fully-connected layers can be added similarly. In Keras, activation functions are treated in the same way as layers. Hence, to add a leaky rectifier, as called LReLU, we add an instance of the ‘LeakyReLU‘ class to the model:

```
alpha = 0.333
LReLU = LeakyReLU(alpha)
model.add(LReLU)
```

Adding Dropout is again done similarly:

```
p = 0.1
model.add(Dropout(p))
```

As the last step before being able to train the model, we need to specify which loss function and which algorithm should be used to respectively specify and minimise the loss function. Again, keras makes this very simple:

```
sgd = SGD(lr = 3e-5, decay =0.0, momentum = 0.9, nesterov = True)
model.compile(optimizer=sgd, loss='categorical_crossentropy', metrics=['
  accuracy']
```

3.2.3 Training

The paper specifies that the model was trained over 20 epochs, each training the network on 450,000 patches (90,000 per class selected at random as explained previously). The optimisation algorithm used is a standard stochastic gradient descent algorithm with Nesterov momentum ($\mu = 0.9$). The learning rate should be linearly decreased from 3×10^{-5} to 3×10^{-7}. Keras doesn't implement this functionality, so it has to be done manually. This can be done using algoritm 2

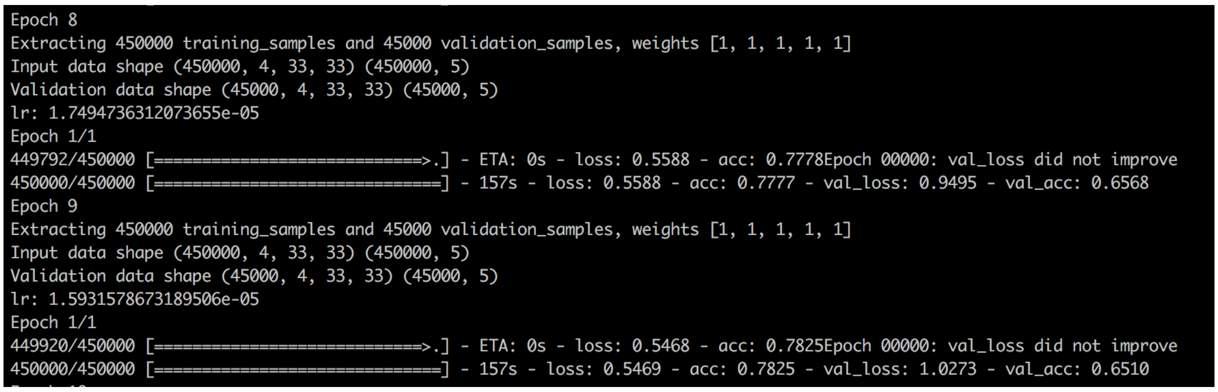
At the end of each epoch the loss function is evaluated on the validation dat and the accuracy (equation 3.4) is computed. The loss and accuracy are then reported to the

Algorithm 2 Training the model with linearly decaying learning rate

```
1: for $i$ from 0 to nb_epochs $-$ 1 do
2:     learning_rate $\leftarrow$ start_rate $+ i \times \frac{\text{end\_rate} - \text{start\_rate}}{\text{nb\_epochs} - 1}$
3:     train_model(nb_epochs = 1, learning_rate)
4: end for
```

standard output. Figure 3.4 shows part the output of training a network for a few epochs.

$$\text{accuracy} = \frac{1}{m}\Big[\sum_{i=1}^{m} \mathbb{1}[y^{(i)} = \arg\max_k P(y^{(i)} = k)]\Big] \tag{3.4}$$



```
Epoch 8
Extracting 450000 training_samples and 45000 validation_samples, weights [1, 1, 1, 1, 1]
Input data shape (450000, 4, 33, 33) (450000, 5)
Validation data shape (45000, 4, 33, 33) (45000, 5)
lr: 1.7494736312073655e-05
Epoch 1/1
449792/450000 [============================>.] - ETA: 0s - loss: 0.5588 - acc: 0.7778Epoch 00000: val_loss did not improve
450000/450000 [==============================] - 157s - loss: 0.5588 - acc: 0.7777 - val_loss: 0.9495 - val_acc: 0.6568
Epoch 9
Extracting 450000 training_samples and 45000 validation_samples, weights [1, 1, 1, 1, 1]
Input data shape (450000, 4, 33, 33) (450000, 5)
Validation data shape (45000, 4, 33, 33) (45000, 5)
lr: 1.5931578673189506e-05
Epoch 1/1
449920/450000 [============================>.] - ETA: 0s - loss: 0.5468 - acc: 0.7825Epoch 00000: val_loss did not improve
450000/450000 [==============================] - 157s - loss: 0.5469 - acc: 0.7825 - val_loss: 1.0273 - val_acc: 0.6510
```

Figure 3.4: Part of the of the training script. The output is shown for epochs 8 and 9. Both the validation loss and the validation accuracy are reported, as well as how long it took to train and validate for this epoch.

After training the model for 20 epochs, the model is saved in a directory specified as an argument to the main python script. Furthermore, to investigate how training the network for longer affects the results, each time the validation loss reaches a new minimum, the model at that point is also saved.

TODO: ADD GRAPH OF TRAINING/VALIDATION ACCURACY

3.3 My model

3.3.1 Architecture

My model differs in a few important ways from the model proposed by Pereira et al. [14].

1. I have decided to increase the size of the input patches, almost doubling it to 64×64. This is because a patch size of only 33×33 seems very small and could potentially miss important contextual clues from voxels slight further away in the slice.

2. The output of the network is no longer a 5×1 array of the probabilities for each class. Instead, the output is now a 5×64 array that outputs the class probabilities for each voxel in the central 8×8 'subpatch' from the input.

3. I have decided to use Batch Normalisation instead of Dropout to regularise the network as it has been shown to work better and significantly speed-up the training process [8].

4. I also added a L2 regularisation penalty to the loss function as a further method to decrease the amount of overfitting done by my model.

5. The network is fully convolutional, meaning that there are only convolutional layers and no fully-connected layers. I made this decision for the following reasons:

   (a) Because no fully-connected layers are used, which are usually those with the highest number of parameters, a fully convolutional neural network will have fewer parameters. It will therefore be less prone to overfitting.

   (b) The fully convolutional neural network is much faster to train and can predict outputs much faster, due to the fact that it mainly applies convolutions, which can be parallelised easily on a GPU. This combined with the fact that the output is an 8×8 patch, will greatly decrease the time necessary to segment a entire scan.

My model should therefore be able to predict classes much faster, using a larger context. The main question now becomes how well it is able to learn the data and to generalise on the challenge data. This will be discussed in the Evaluation chapter.

The model contains 13 layers, shown in table 3.3.1. All convolutional layers have filter size 3×3 and stride 1×1. The width of the filters is increased as the output shape decreases to allow the network to model complex features. Max pooling layers are applied every 2–3 convolutional layers, to reduce the spatial input for the following layers. It is important to note that there are no fully-connected layers. Instead convolutional layers and max pooling layers are applied until we have achieved the correct dimension of $5 \times 8 \times 8$. At this point, we have to flatten the input into a single two-dimensional array of size 5×64, onto which we can apply a softmax activation to obtain normalised probabilities for each one of the central 64 voxels.

3.3.2 Implementation

I implemented my model using Keras and it is therefore very similarl to how I implemented the model proposed by Pereira et al. However, there is a subtlety that arises from the fact

Layer type	Output shape	# Parameters
Conv	$64 \times 64 \times 64$	2368
Conv	$64 \times 64 \times 64$	36 928
Max-Pool	$64 \times 32 \times 32$	0
Conv	$128 \times 32 \times 32$	73 856
Conv	$128 \times 32 \times 32$	147 584
Conv	$128 \times 32 \times 32$	147 584
Max-Pool	$128 \times 16 \times 16$	0
Conv	$256 \times 16 \times 16$	295 168
Conv	$256 \times 16 \times 16$	590 080
Max-Pool	$256 \times 8 \times 8$	0
Conv	$256 \times 8 \times 8$	590 080
Conv	$256 \times 8 \times 8$	590 080
Conv	$5 \times 8 \times 8$	11 525
		2 118 213

Table 3.6: Summary of my convolutional neural network architecture, including the number of parameters in each layer.

that the network is fully-convolutional. The output of the last layer is a three-dimensional array of size $5 \times 8 \times 8$ but the softmax activation can only take as an argument two-dimensional arrays. Therefore, I have to reshape the array into a 64×5 array before applying the softmax activation. Keras makes this operation easy with the ‘Reshape’ and ‘Permute’ layers:

```
model.add(Reshape((5, 64)))
model.add(Permute((2,1)))
model.add(Activation('softmax'))
```

Similarly, using Batch Normalisation is easy in Keras, as it has a ‘BatchNormalization’ layer built in its library. A typical convolutional layer with Batch Normalisation is implemented as follows:

```
model.add(Convolution2D(256, 3, 3, border_mode='same', init=init,
  W_regularizer=l2(1)))
model.add(BatchNormalization(axis=axis, trainable=trainable))
model.add(LeakyReLU(alpha))
```

3.3.3 Training

TODO: choose a training method that works fairly well...

3.4 Segmentation

After the model has been trained, the next step consists of segmenting entire scans to either validate the model, inspect how it behaves or to submit the segmentations to the online platform for evaluation. This means that the input scan has to be transformed into a set of patches that can then be evaluated by the convolutional neural network. The result of classifying each of those patches then has to be put back into a single three-dimensional scan, which will be the segmentation.

3.4.1 Pereira model

For the model proposed by Pereira et al. [14], the convolutional neural network can be modelled as a function classifying the central voxel of a patch of size $33 \times 33 \times 4$. In this case, the segmentation of a patient is equivalent to convolving the convolutional neural network on each of the two-dimensional slices of the input images. The images must however be zero-padded around the x and y edges so as to keep the dimensions of the segmentation the same as the dimensions of the input images. Hence the steps are:

1. Read in the scan images as a single 4-dimensional array of size $z \times y \times x \times 4$.
2. Pre-process the image as explained in section 3.1, except for the data augmentation step. Thus, the image is first winsorised, then the N4ITK correction is applied to it, and finally each modality within the image is linearly scaled and its mean and variance are standardised.
3. Pad the x and y dimensions with zeros. Since we want the convolution operation to return an image of identical size, we need to pad with exactly half of the width of the filter, that is 16 voxels. The array has now size $z \times (y + 32) \times (x + 32) \times 4$.
4. For each slice in the axial plane, that is for each slice in the z axis, extract all patches into a list. Note that this has to be done for each slice sequentially as storing all patches for the entire image at once would not be possible. For each slice there will be $y \cdot x$ patches of size $33 \times 33 \times 4$.
5. Evaluate the convolutional neural network on each patch. Evaluating these patches in batches using a GPU can make this step significantly faster. The convolutional neural network will return a probability distribution over the 5 classes for each patch. The most likely class is chosen as the class for that patch. Record the classes in a second list.
6. Transform the list of classes back into a two-dimensional array of size $y \cdot x$. This slice corresponds to the z.

Figure 3.5 shows the segmentation of the first challenge patient using the above described method. The segmentation of a patient takes about 40 minutes using the GPUs provided by the Cambridge High Performance Computing Cluster.

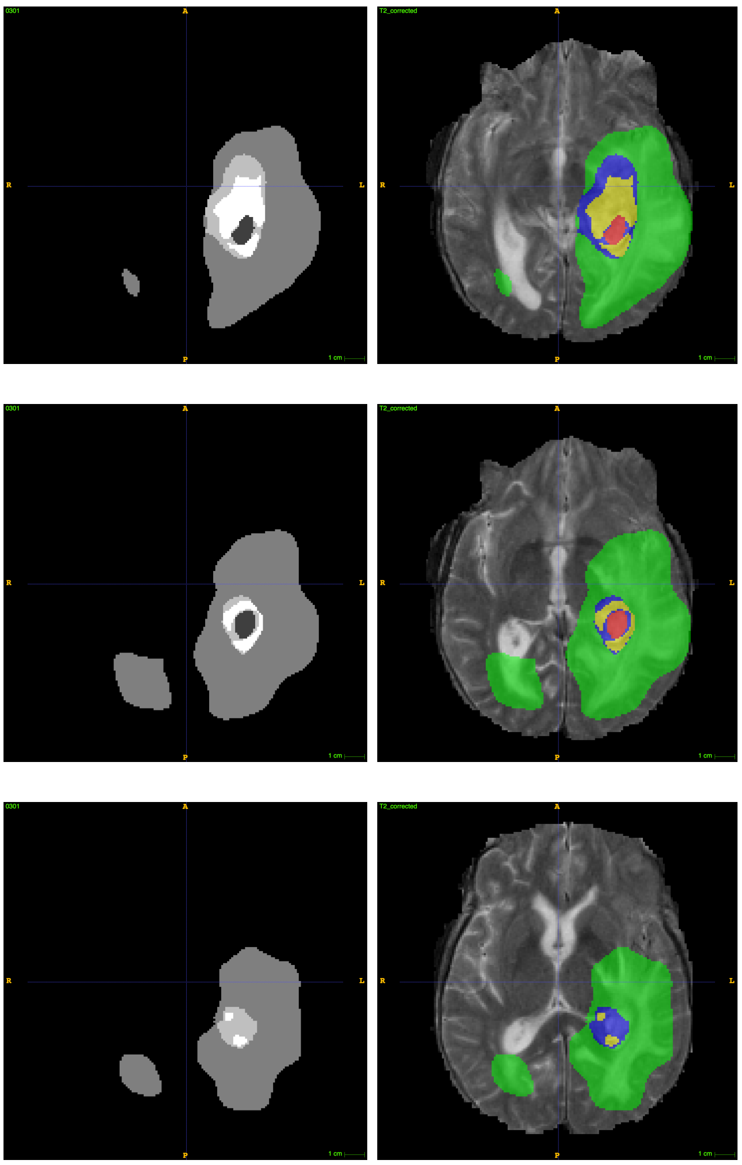


Figure 3.5: The segmentation alone is shown on the left and the segmentation overlaid on the T2 scan is shown on the right. Axial plane slices for $z \in \{66, 71, 76\}$ are shown.

3.4.2 My model

The segmentation algorithm for my model is different from the one I implemented for the Pereira model as my model classifies a square of 8×8 voxels at a time. First, we have to pad the image using a slightly different formula, depending on the length of the image in the x and y axis. This is because we are effectively convolving the model on the slices with a stride of 8×8 and thus the last application of the model in a row or column might not fit if the width or height of the slice is not a multiple of 8. To fix this I pad the model by $\frac{64-8}{2} = 28$ at in front of the x and y axis and an extra 8 voxels at the end of the x and y axes so pad the end with $\frac{64-8}{2} + 8 = 36$. The convolution is then implemented by two nested for loops, with increments of 8 after each step. In more detail, the steps involved in the segmentation algorithm applied to every slice are:

1. Extract the patches of size $64 \times 64 \times 4$ for the entire slice. This can be done after padding by two nested for loops with step increments of 8.
2. For each patch, use the convolutional neural network to predict the class for its central 64 voxels. This are returned in a one-dimensional array of length 64.
3. Concatenate in order the predictions for all patches into a single one-dimensional array of length $64 \cdot \#patches$.
4. Calculate the number of patches that fit into the slice as follows:

$$\begin{aligned} \text{height}_p &\leftarrow \left\lceil \frac{\text{image\_height}}{8} \right\rceil \\ \text{width}_p &\leftarrow \left\lceil \frac{\text{image\_width}}{8} \right\rceil \end{aligned} \tag{3.5}$$

5. Reshape the voxels into a single 4-dimensional array of size $(\text{height}_p \times \text{width}_p \times 8 \times 8)$
6. Concatenate the array along its first axis. This can be done using the ‘concatenate’ function provided by numpy. The result is a 4-dimensional array of size $(\text{height}_p \times 8 \cdot \text{width}_p \times 8)$. This will concatenate individual patches along rows together.
7. Similarly, concatenate the new array along its first axis. This will concatenate the columns along the rows, resulting into a single 2-dimensional array of size $(8 \cdot \text{height}_p \times 8 \cdot \text{width}_p)$.
8. Finally, remove the extra voxels that might have been added by the extra padding needed. This can be done using numpy to trim the array to the image width and height. The result will be a 2-dimensional array which is the segmented slice.

3.5 Data post-processing

After the new patient has been segmented, a further simple heuristic is used to remove small volumes of data labeled as non normal tissue. This assumption is made on the basis that the tumour or tumours will be connected into components of relatively large size. Pereira et al [14] proposed to remove all connected components of less than 10,000 voxels in volumes.

I implemented this heuristic using a depth first search on the graph represented by a scan, where each voxel is a vertex and share an edge with its 6 immediate neighbours (or less if it is on an edge). First I create a second 3-dimensional array of the same size as the scan and mark all voxels with a 1 for which the corresponding voxel in the segmentation has value 0. This will allow us to remember which voxels have already been visited. Then the algorithm iterates the following steps until all voxels have been visited:

1. Initialise an empty list, to represent the next connected component C.
2. Remove a previously not visited voxel, mark it as visited, add its coordinates to C.
3. Initialise a queue containing all non visited neighbours of the voxel that are not segmented as normal tissue.
4. Then, repeat while the queue is not empty:
   (a) Pop the first element, mark it as visited, and add its coordinates to the connected component C.
   (b) Find its neighbours that are unprocessed and have not yet been added to the queue and push those to the end of the queue. Repeat step 4.
5. Once the queue is empty, add the connected component C to a list containing all disjoint connected components.

Finally, I iterate over the list of connected components, finding those which have less voxels than the threshold size and setting the segmentation of those voxels to 0. Figure 3.6 shows the effect of applying this post-processing step to the last challenge scan.

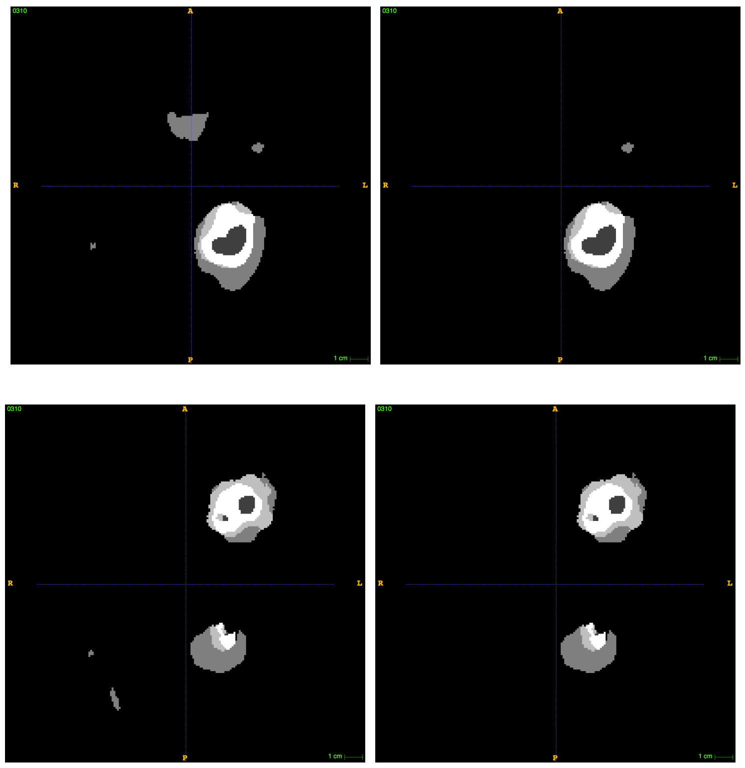


Figure 3.6: Effect of applying the post-processing step to the 10th challenge scan. Axial plane slices $z \in \{77, 87\}$ are shown side-by-side (on the left before the post-processing step and on the right with the post-processing step).

Chapter 4

Evaluation (+ Conclusion [20%])

4.1 BraTS evaluation

The BraTS2013[2] evaluates the segmentation using three different metrics: **dice score**, **sensitivity** (also referred to as recall) and **positive predictive value** (also referred to as precision). These metrics are best visualised using the Venn diagram shown in figure 4.1. Note in particular that the more standard metric of accuracy is not used. This is because the data is highly unbalanced, as shown previously, meaning that a model labelling everything with class 0 would reach an accuracy of 99% or more, making it very hard to compare different models.

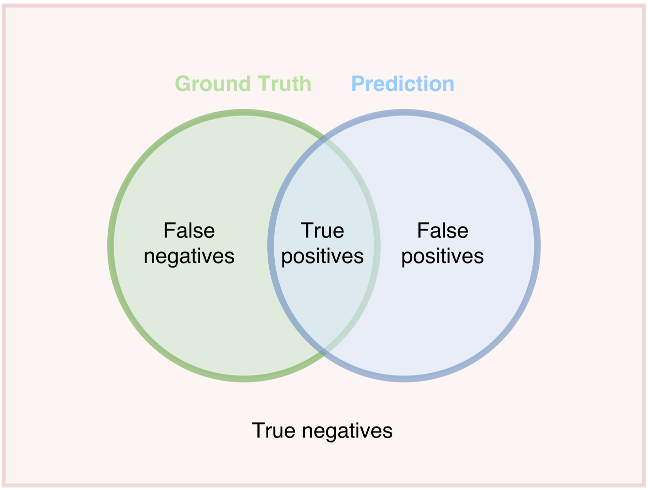


Figure 4.1: Venn diagram showing the different areas and how the can be classified.

Positive predictive value

The positive predictive value is defined as

$$\text{PPV} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{4.1}$$

where TP and FP are the numbers of true and false positives respectively. In the Venn diagram 4.1, this corresponds to the size of the green and blue area relative to the size of the entire blue area. Since the denominator TP + FP is the total number of positively classified elements, the positive predictive value gives an indication as to how accurately the model can predict a positive class. In our case of brain tumour segmentations, the positive predictive value measure how confidently the model is predicting tumours, with a high positive predictive value meaning that most of tumours labelled by the model really tumours are.

Sensitivity

The sensitivity is defined as

$$\text{Sensitivity} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{4.2}$$

where FN is the number of false negative classifications. The sensitivity measures the size of the blue and green region relative to the size of the entire green region. As the denominator TP + FN equals the green area, i.e. the total number of positively labelled elements in the ground truth, the sensitivity measures how well a model is able to recognise positively labelled examples. In the case of brain tumour segmentations, the sensitivity measures how good a model is at recognising a tumour. A high sensitivity would indicate that the model is able to recognise a high fraction of the positively labelled voxels.

There is a trade-off between the sensitivity and the positive predictive value. A model classifying everything as negative would reach perfect positive predictive as there would be no false positives but would have a sensitivity of 0. Similarly, a model classifying everything as positive would reach perfect sensitivity as there would be no false negatives but would have a positive predictive value of 0. For brain tumours, a high sensitivity is arguably more important as missing a tumour can have deadly consequences. However, this has a trade-off with the utility of a model, since a low positive predictive value also means that we can only have little confidence in the positive predictions made by it.

Dice score

The dice score takes into account both the false negatives and the false positives and thus give a metric which is to some extend independent of that trade-off. The dice score is defined as

$$\begin{aligned}\text{Dice Score} &= \frac{|\text{Ground truth} \cap \text{Prediction}|}{\frac{1}{2} \cdot (|\text{Ground truth}| + |\text{Prediction}|)} \\ &= \frac{2 \cdot \text{TP}}{\text{FP} + 2 \cdot \text{TP} + \text{FN}}\end{aligned} \tag{4.3}$$

The dice score normalises the number of true positive classification to the average size of the two segmented areas.

Figure 4.1 shows a fictive example for a segmented slice in the axial plane. The metrics for this segmentation are

$$\begin{aligned}\text{PPV} &= \frac{|P_1 \cup T_1|}{|P_1|} \\ \text{Sensitivity} &= \frac{|P_1 \cup T_1|}{|T_1|} \\ \text{Dice score} &= \frac{|P_1 \cup T_1|}{\frac{1}{2} \cdot (|P_1| + |T_1|)}\end{aligned}$$

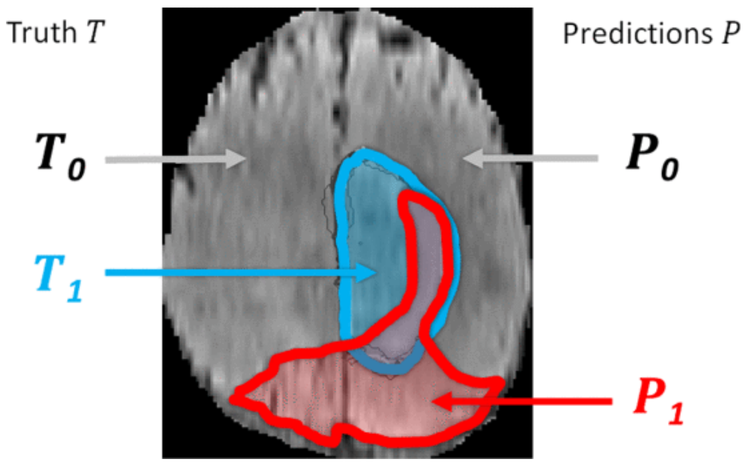


Figure 4.2: Example showing a segmentation for a slice in the axial plane. This image was reproduced from [2]

4.1.1 Regions of evaluation

As stated above, the metrics are defined only for binary classification into positive and negative classes. However, our segmentations are classified into 5 different classes (0–4). Therefore, different 'regions' define which classes are counted as positive and which ones are negative. For the BraTS challenge there are three regions:

1. **Complete**: All classes 1–4 are defined as positive, only 0 as negative. The metrics for the complete scores therefore evaluate how well a model is able to discern diseased tissues from normal tissues.
2. **Core**: Classes 1,3 and 4 are defined as positive, 0 and 2 as negative. In this region, the edema class is no longer considered as positive.
3. **Enhancing** Only class 4, the enhacing tumour class is defined as positive. The metrics for this region measure how well the model can segment the enhancing tumour class only.

4.2 Evaluation of the model proposed by Pereira et al.

In table 4.2 are shown the Dice score values I obtained for each of the 10 challenge scans by replicating the method described by Pereira et al. [14]. Note that for patient 7, the dice scores for the core and enhancing regions are particularly low, almost 0. For this patient, the diseased tissue has been segmented correctly but the model wasn't able to distinguish the different classes within the diseased tissue and classified almost everything as either normal tissue (class 0) or edema (class 2). Because class 2 is not defined as positive in the core and enhancing regions, the scores for those regions is very close to 0.

Patient	**Dice score**		
	Complete	Core	Enhancing
1	0.722	0.779	0.561
2	0.718	0.669	0.627
3	0.825	0.693	0.320
4	0.741	0.447	0.200
5	0.725	0.676	0.580
6	0.741	0.449	0.502
7	0.767	0.064	0.043
8	0.750	0.840	0.710
9	0.823	0.842	0.808
10	0.788	0.795	0.841

Table 4.1: Dice scores obtained for the 10 challenge patients using the method proposed by Pereira et al. These results were produced by the online evaluation platform as the ground truth labelling is not publicly available.

Using these values I was able to calculate the average and the sample standard deviation, reported in Table 4.2.

Region	**Average dice score**
Complete	0.7600 ± 0.0398
Core	0.6253 ± 0.2433
Enhancing	0.519 ± 0.2602

Table 4.2: Dice score mean and standard deviation for the three regions obtained for the 10 challenge patients using the method proposed by Pereira et al.

The mean scores for the core region and the enhancing region are significantly lower than the mean score for the complete region and have a much higher standard deviation. This can be explained partially by the outlier values obtained from patient 7. However, even if the results from patient 7 were ignored, the average results are still significantly below those report by Pereira et al. [14] of (0.8, 0.78, 0.73). As I had exactly followed the method described in the paper published by Pereira et al., I emailed the author to try to understand this discrepancy. He answered that there were two differences between our methods that could potentially explain this difference in results. The first difference were the parameters used in the N4 correction, which had not be included in the paper. The author was kind enough to share those with me. However, changing the parameters used for the N4 correction did not have an impact on the results. The second difference was how the patches are selected. The author of the paper wrote in his email that he only selected patches at least 3 voxels apart and made sure to include patches of normal tissue close to a tumour edge. As this was not mentioned in the paper either, I did not have the time to verify what impact this difference had on the results.

Table 4.2 shows the sensitivity and the positive predictive values obtained for the 10 challenge patients. The means and standard deviations for the sensitivity and positive predictive value are reported in tables 4.2 and **??** respectively.

Patient	**Sensitivity**			**Positive predictive value**		
	Complete	Core	Enhancing	Complete	Core	Enhancing
1	0.960	0.781	0.407	0.578	0.778	0.902
2	0.978	0.523	0.481	0.567	0.929	0.903
3	0.822	0.532	0.197	0.828	0.992	0.856
4	0.653	0.311	0.115	0.857	0.796	0.757
5	0.758	0.526	0.495	0.694	0.945	0.702
6	0.775	0.293	0.340	0.709	0.963	0.962
7	0.887	0.033	0.022	0.676	0.975	0.995
8	0.959	0.765	0.723	0.616	0.930	0.698
9	0.934	0.887	0.843	0.736	0.801	0.777
10	0.969	0.976	0.863	0.663	0.670	0.820

Table 4.3: Sensitivity and positive predictive value obtained for the 10 challenge patients using the method proposed by Pereira et al. These results were produced by the online evaluation platform as the ground truth labelling is not publicly available.

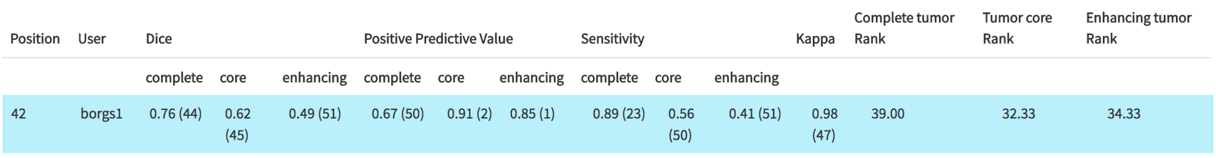
Region	Average sensitivity mean
Complete	0.8696 ± 0.1120
Core	0.5627 ± 0.2955
Enhancing	0.4485 ± 0.2936

Table 4.4: Sensitivity mean and standard deviation for the three regions obtained for the 10 challenge patients using the method proposed by Pereira et al.

Region	Average positive predictive value
Complete	0.6926 ± 0.0963
Core	0.8779 ± 0.1081
Enhancing	0.8371 ± 0.1040

Table 4.5: Positive predictive vale mean and standard deviation for the three regions obtained for the 10 challenge patients using the method proposed by Pereira et al.

Finally, the online evaluation platform also ranks the different contestants according to their scores. Figure 4.2 shows a screenshot of part of the ranking table that includes my submission. Note that my submission ranked 2nd and 1st for the positive predictive value on the core and enhancing regions respectively, but ranked very low for the sensitivity on those regions. This suggests that my model finds tumour with very high reliability but doesn't recognise all tumours and shows that there is indeed a trade-off between positive predictive value and sensitivity.



Position	User	Dice			Positive Predictive Value			Sensitivity			Kappa	Complete tumor Rank	Tumor core Rank	Enhancing tumor Rank
		complete	core	enhancing	complete	core	enhancing	complete	core	enhancing				
42	borgs1	0.76 (44)	0.62 (45)	0.49 (51)	0.67 (50)	0.91 (2)	0.85 (1)	0.89 (23)	0.56 (50)	0.41 (51)	0.98 (47)	39.00	32.33	34.33

Figure 4.3: Online evaluation ranking. My submission was ranked 42nd TODO WRONG REDO SCREENSHOT FOR THE CORRECT SUBMISSION

4.3 Evaluation of my model

4.4 Comparison

Chapter 5

Conclusion

5.1 Summary of achievements

5.2 Future Work

I hope that this rough guide to writing a dissertation is LaTeX has been helpful and saved you time.

Bibliography

[1]

[2] Pavel Dvorak and Bjoern Menze. Structured prediction with convolutional neural networks for multimodal brain tumor segmentation. *Proceedings of the Multimodal Brain Tumor Image Segmentation Challenge*, pages 13–24, 2015.

[3] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS10). Society for Artificial Intelligence and Statistics*, 2010.

[4] McKinsey L. Goodenberger and Robert B. Jenkins. Genetics of adult glioma. *Cancer Genetics*, 205(12):613–621, 2017/03/29.

[5] A. Hamamci, N. Kucuk, K. Karaman, K. Engin, and G. Unal. Tumor-cut: Segmentation of brain tumors on contrast enhanced mr images for radiosurgery applications. *IEEE Transactions on Medical Imaging*, 31(3):790–804, March 2012.

[6] Mohammad Havaei, Axel Davy, David Warde-Farley, Antoine Biard, Aaron C. Courville, Yoshua Bengio, Chris Pal, Pierre-Marc Jodoin, and Hugo Larochelle. Brain tumor segmentation with deep neural networks. *CoRR*, abs/1505.03540, 2015.

[7] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[8] S. Ioffe and C. Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *ArXiv e-prints*, February 2015.

[9] Konstantinos Kamnitsas, Christian Ledig, Virginia F. J. Newcombe, Joanna P. Simpson, Andrew D. Kane, David K. Menon, Daniel Rueckert, and Ben Glocker. Efficient multi-scale 3d CNN with fully connected CRF for accurate brain lesion segmentation. *CoRR*, abs/1603.05959, 2016.

[10] Konstantinos Kamnitsas, Christian Ledig, Virginia F.J. Newcombe, Joanna P. Simpson, Andrew D. Kane, David K. Menon, Daniel Rueckert, and Ben Glocker. Efficient multi-scale 3d {CNN} with fully connected {CRF} for accurate brain lesion segmentation. *Medical Image Analysis*, 36:61 – 78, 2017.

[11] J. Kleesiek, A. Biller, G. Urban, Ullrich Köthe, M. Bendszus, and Fred A. Hamprecht. ilastik for multi-modal brain tumor segmentation. In *MICCAI BraTS (Brain Tumor Segmentation) Challenge. Proceedings, 3rdplace*, pages 12–17, 2014.

[12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[13] Bjoern Menze, Andras Jakab, Stefan Bauer, Jayashree Kalpathy-Cramer, Keyvan Farahani, Justin Kirby, Yuliya Burren, Nicole Porz, Johannes Slotboom, Roland Wiest, Levente Lanczi, Elisabeth Gerstner, Marc-Andre Weber, Tal Arbel, Brian Avants, Nicholas Ayache, Patricia Buendia, Louis Collins, Nicolas Cordier, Jason Corso, Antonio Criminisi, Tilak Das, Hervé Delingette, Cagatay Demiralp, Christopher Durst, Michel Dojat, Senan Doyle, Joana Festa, Florence Forbes, Ezequiel Geremia, Ben Glocker, Polina Golland, Xiaotao Guo, Andac Hamamci, Khan Iftekharuddin, Raj Jena, Nigel John, Ender Konukoglu, Danial Lashkari, Jose Antonio Mariz, Raphael Meier, Sergio Pereira, Doina Precup, S. J. Price, Tammy Riklin-Raviv, Syed Reza, Michael Ryan, Lawrence Schwartz, Hoo-Chang Shin, Jamie Shotton, Carlos Silva, Nuno Sousa, Nagesh Subbanna, Gabor Szekely, Thomas Taylor, Owen Thomas, Nicholas Tustison, Gozde Unal, Flor Vasseur, Max Wintermark, Dong Hye Ye, Liang Zhao, Binsheng Zhao, Darko Zikic, Marcel Prastawa, Mauricio Reyes, and Koen Van Leemput. The Multimodal Brain Tumor Image Segmentation Benchmark (BRATS). *IEEE Transactions on Medical Imaging*, page 33, 2014.

[14] Sergio Pereira, Adriano Pinto, Victor Alves, and Carlos A. Silva. Brain tumor segmentation using convolutional brain tumor segmentation using convolutional neural networks in mri images. *IEEE Transactions on medical imaging*, 35(5):1240–251, May 2016.

[15] Marcel Prastawa, Elizabeth Bullitt, Sean Ho, and Guido Gerig. A brain tumor segmentation framework based on outlier detection. *Medical Image Analysis*, 8(3):275 – 283, 2004. Medical Image Computing and Computer-Assisted Intervention - {MICCAI} 2003.

[16] Florian Schroff, Dmitry Kalenichenko, and James Philbin. Facenet: A unified embedding for face recognition and clustering. *CoRR*, abs/1503.03832, 2015.

[17] Srivastava, Hinton, Krizhevsky, Sutskever, and Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, pages 1929–1958, 2014.

[18] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In *Proceedings of the 27th International Conference on Neural Information Processing Systems*, NIPS'14, pages 3104–3112, Cambridge, MA, USA, 2014. MIT Press.

[19] N. J. Tustison, B. B. Avants, P. A. Cook, Y. Zheng, A. Egan, P. A. Yushkevich, and J. C. Gee. N4itk: Improved n3 bias correction. *IEEE Transactions on Medical Imaging*, 29(6):1310–1320, June 2010.

[20] Darko Zikic, Yani Ioannou, Antonio Criminisi, and Matthew Brown. Segmentation of brain tumor tissues with convolutional neural networks. In *MICCAI workshop on Multimodal Brain Tumor Segmentation Challenge (BRATS)*. Springer, October 2014.