Sebastian Borgeaud dit Avocat

Brain tumour segmentation using Convolutional Neural Networks

Computer Science Tripos – Part II

Fitzwilliam College

May 5, 2017

Proforma

Name:	**Sebastian Borgeaud dit Avocat**
College:	**Fitzwilliam College**
Project Title:	**Brain tumour segmentation using Convolutional Neural Networks**
Examination:	**Computer Science Tripos – Part II, June 2017**
Word Count:	**11,569**
Project Originator:	Duo Wang
Supervisor:	Dr. Mateja Jamnik & Duo Wang

Original aims of the project

The main success criteria for this project was to implement an algorithm based on a convolutional neural network that performs brain tumour segmentation. The second aim was to achieve similar results to those obtained by Pereira et al. [17], the current state-of-the-art, in particular obtaining 90% of the accuracy reported. This means achieving the Dice scores (0.79, 0.73, 0.67) for the regions 'complete' (including classes 2-4), core (classes 3-4) and enhancing (class 4) in the BraTS2013 challenge [2], respectively.

Completed work

The first goal has been achieved: Pereira's method was replicated to obtain a model that learns how to perform brain tumour segmentation. The dice scores obtained are (0.8149, 0.6704, 0.5921), performing better in the first region than Pereira's model but not achieving the 90% threshold for the two other regions. Then, a second model was designed using more recent techniques, achieving an extension goal.

Special difficulties

The main variant of Pereira's model used a semi-automatic normalisation which requires input from an expert. Therefore, a fully-automatic variant, also reported by Pereira et al. [17] in the same paper, was implemented.

[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

List of Figures

Acknowledgements

I would like to thank my supervisors, Dr. Mateja Jamnik and Duo Wang, for proposing this project to me, and for patiently providing assistance and answering my questions throughout the project. Moreover, I would also like to thank the Computational Biology group in the Computer Lab for giving me access to one of their GPUs. Finally I would like to thank my Director of Studies, Dr. Robert Harle, who read an early version of this dissertation and provided useful feedback.

Chapter 1

Introduction

1.1 Motivation

In recent years, convolutional neural networks have been shown to significantly outperform other methods in many areas of Computer Science such as Computer Vision (FaceNet[19]), image classification (AlexNet [13]), Natural language processing [23] and several 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 [17] [10].

Brain tumours may be primary or secondary. Primary brain tumours start in the brain while secondary (or metastatic) tumours spread to the brain from a different part of the body. Primary brain tumours are further divided into different types, of which the glioma is the most frequently occurring in adults. The glioma arises from glial cells and infiltrates the surrounding tissue. Goodenberger and Jenkins [5] 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 classified 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 disease – the high-grade glioma – have a median survival rate of two years or less [16]. 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. Image processing algorithms that can automatically analyse tumour scans would therefore be of great value for improved diagnosis, treatment planning and evaluation of the tumour progression. However, 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 patients, making it hard to incorporate and encode strong priors on shape and location which are often used in segmentations of anatomical structures [2]. The problem of automatic brain

tumour segmentation is therefore still an active research area.

The first aim of my project is to understand and 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.[17] in 2016 which is based on two-dimensional convolutional neural networks.

In the second phase of my project, I introduce a novel network architecture that achieves similar performance as my implementation of Pereira's model. My model uses a larger receptive field, but is still able to segment scans much faster.

1.2 Related Work

Brain tumour segmentation algorithms can be divided into generative and discriminative models.

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. [18] used a 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 methods 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: Hamamci et al. [6] used raw input pixels, for example, and Kleesiek et al. [12] used local histograms. The methods usually train a classifier that relies on these hand-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 such hand-designed features is that by their nature they are 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 do this to some extend; starting from the raw data input they extract increasingly more complex features using the features computed by the previous layer in the network.

Zikic et al. [25] first proposed a convolutional neural network with two convolutional layers followed by a fully-connected layer for brain tumour segmentation. 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. Building on this approach, Havaei et al. [7] used a novel double-pathway architecture and a cascade of two networks

to rank second 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. [17] proposed a deeper two-dimensional convolutional neural network of 11 layers, that ranked first in the BraTS2013 challenge. As the aim of my project is to replicate this method, the details are explained 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. [**?**] used a three-dimensional convolutional neural network with residual connections [8] to take the first place in the BraTS2015 challenge.

1.3 Supervised learning and classification

Machine learning tasks can be divided into supervised and unsupervised learning.

- In unsupervised learning, the task is to learn a function to describe some hidden structure from unlabelled data.
- In supervised learning, the task is to learn a relationship between data points from labelled data. More formally, the aim is to infer a function $f : A \to B$ from a set of n labelled data points $\mathbb{X} = \{(x^{(1)}, y^{(1)}), ..., (x^{(n)}, y^{(n)})\}$, with $\mathbb{X} \subseteq A \times B$. Typically, if $A \subset \mathbb{R}^d$ and $b_i \in \mathbb{R}$, then the task is also called **regression**. If $B = \{b_1, ..., b_k\}$ is discrete, the task is called **classification**.

In either case, one approach to solve the task is by defining an appropriate loss function $\mathcal{L} : B \times B \to \mathbb{R}$. The loss function is then minimised over some test dataset $\sum_{i=1}^{n} \mathcal{L}(f(\mathbf{a}_i), b_i)$ in the hope that the learned function f will be able to generalise well on unseen data.

The brain tumour segmentation is an instance of a supervised classification problem. The training data set $\mathbb{X}$ consists of MRI scans where each voxel is segmented into one of five classes:

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

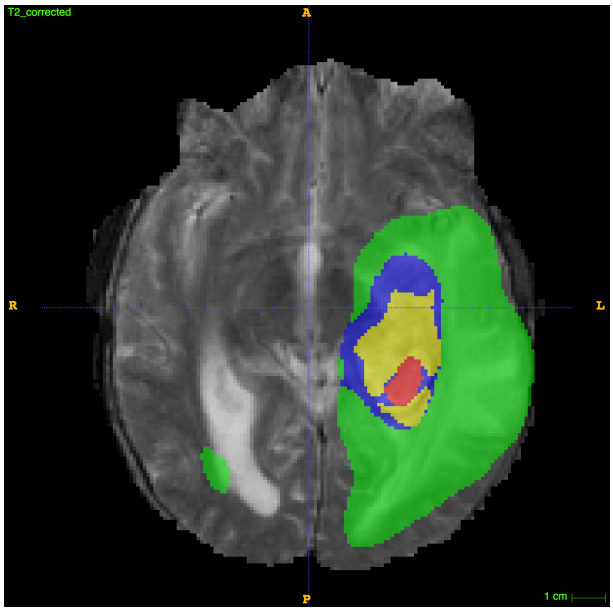

Figure 1.1: Axial plane slice of an MRI scans overlaid by a possible segmentation. The regions in red, green, blue and yellow correspond to classes 1, 2, 3 and 4 respectively.

An example of such a segmentation is shown in figure 1.1.

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 map this function f to every voxel in an image, we have effectively created a segmentation.

1.4 Dissertation outline

The rest of the dissertation is structured as follows: in the Preparation chapter I will explain how convolutional neural networks can be used to solve the supervised learning task. I will also give more details on how the data is structured and where it comes from. Then, in the Implementation chapter, I will go over how I transform the data into a format that can be used to train a convolutional neural network, and how I implemented the convolutional neural network proposed by Pereira et al. [17]. I will also present the architecture of my new model. Then, I will explain in more detail how the model is used to segment a scan efficiently. Finally, I will detail how I implement the post-processing step based on connected components. In the Evaluation chapter, I will go over how segmentations can be evaluated quantitatively and will evaluate both models.

Chapter 2

Preparation

2.1 Starting point

2.1.1 Theory

The starting point for this project was 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 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 those parts on the evaluation of classifiers and on general techniques for machine learning.

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

2.1.2 Programming languages & libraries

The project was almost entirely written in Python, except the computing jobs for the Cambridge High Performance Cluster, which are bash scripts. I had only used Python before for very small, single file projects and had to learn how Object Oriented Programming is handled in Python and some of the more advanced features. I also heavily used the Numpy[2] library, which I had not used before.

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

[2] `http://www.numpy.org/`

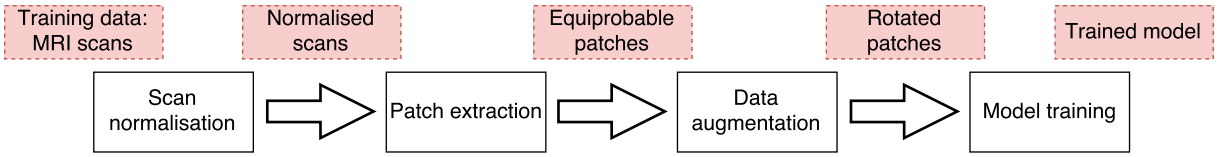

Figure 2.1: The process of training the convolutional neural network as a pipeline. The boxes in red describe the input and outputs for each stage.

To load in the scans, which are in MetaImage medical format ('.mha') I used the SimpleITK[3] library for python.

Finally, I used the Keras[4] library, which makes it easy to create and train convolutional neural networks while leaving all important design and architectural decisions to the user. As Keras is becoming the standard open-source library in deep learning research and applications, many online resources were available when needed.

2.1.3 Graphical Processing Unit (GPU)

To train the convolutional neural network in a reasonable amount of time, I needed a computer with an external GPU. Initially, I used the Wilkes Cluster[5] from the Cambridge University High Performance Computing services, which is equipped with NVIDIA K20 GPUs. From February 23rd 2017, I was able to use the TITAN X GPUs from the Computational Biology group in the Computer Lab. This reduced the training time for the model proposed by Pereira et al. from about 12 hours to 2–3 hours.

2.2 Software engineering

Training a model and using that model to segment new scans can conceptually be divided into different stages. These processes can therefore be modelled as two data pipelines, one for the training and one for the segmentation, where each stage takes as an input the output of the previous stage. The pipeline for training the model and the pipeline for segmenting an MRI scan are shown respectively in Figures 2.1 and 2.2.

I used a waterfall like approach for the implementation, building and testing the different stages independently. All stages except the scan normalisation and the post-processing are model dependent. Instead of reimplementing each stage twice, once for the model proposed by Pereira et al. and once for the model I propose, I implement

[3] `https://itk.org/Wiki/SimpleITK`
[4] `https://keras.io/`
[5] `https://www.hpc.cam.ac.uk/services/wilkes`

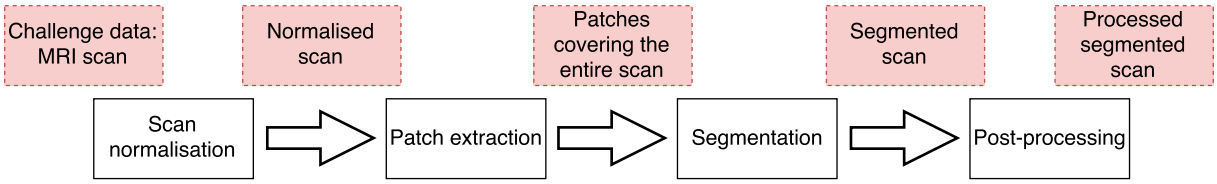

Figure 2.2: The process of segmenting an MRI scan as a pipeline. The boxes in red describe the input and outputs for each stage.

each stage only once in such a way that the model-dependent aspects can be specified by parameters. This flexibility is especially important as it made it easy to experiment many different model architectures.

It is important to note that division into sequential stages provides a nice way of modularising the code base, but it is mainly conceptual. In practice, the data cannot flow in this manner because of the large amount of it. For example, in the patch extraction phase for the segmentation, it would not be possible to load all patches in memory at once. Therefore the patches are extracted and segmented slice-by-slice. The implementation of these stages is described in the next chapter.

I used git as a version control system as I was already familiar with it. The repository is stored locally and in a private GitHub repository in the cloud. I pushed the contents of the local repository to GitHub after most commits, at least once a day. Furthermore, as an additional backup strategy I regularly backed up the local repository to my external hard disk.

2.3 Theoretical background

2.3.1 Artificial neural networks

To understand how convolutional neural networks work, we first need to be familiar with ordinary neural networks. These consist of a sequence of layers of neurons, in which each neuron has 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 2.3. Each neuron in layer $n+1$ is connected to every neuron in layer n and computes as 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 called a fully-connected layer.

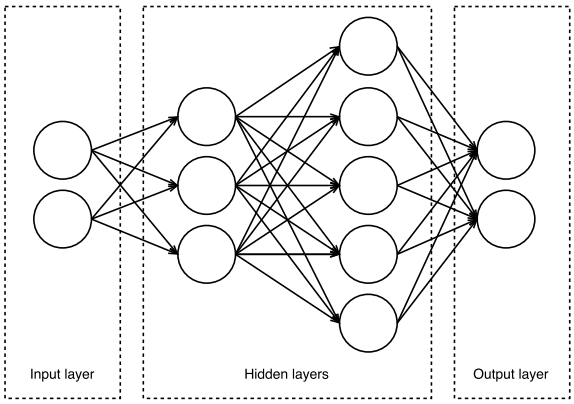

Figure 2.3: Structure of a simple neural network with two hidden layers.

Activation functions

The most common activation functions are the Sigmoid function:

$$S(x) = \frac{1}{1 + e^x} \tag{2.1}$$

the hyperbolic tangent:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.2}$$

the rectifier:

$$f(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{otherwise} \end{cases} \tag{2.3}$$

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

$$f(x) = \begin{cases} \alpha x & \text{if } x < 0 \\ x & \text{otherwise} \end{cases} \tag{2.4}$$

These functions are plotted in figure 2.4. Historically, the hyperbolic tangent function or the sigmoid function have been used as activation functions. However, the magnitude of the gradients of these two activation functions is always below 1. This causes a numerical issue for deeper networks, as the gradients for each layer are multiplied together during the backpropagation algorithm. This is known as the vanishing gradient problem [1] and is the reason 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 usually consist of K nodes, one for each class. Using a softmax activation function for the last layer, we can interpret the value at each node as the probability that the input belongs to this class. More formally, in a network with parameters θ, we can view the output of node k when presented with input $x^{(i)}$ as the probability $P(y^{(i)}_{\text{pred}} = k \mid x^{(i)}; \theta)$. The softmax activation computes for each output k

$$\sigma(\mathbf{x})_k = \frac{e^{\mathbf{x}_k}}{\sum_{j=1}^{K} e^{\mathbf{x}_j}} \tag{2.5}$$

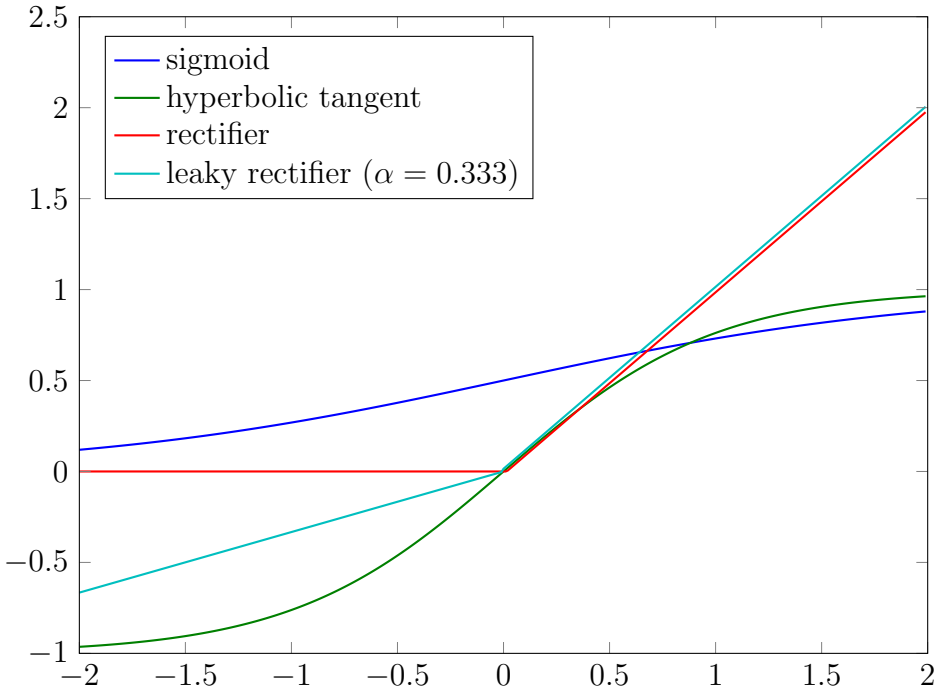

Figure 2.4: Plot of the activation functions.

where $\mathbf{x}$ is the vector consisting of all outputs from the previous layer. The output value of each neuron will range between 0 and 1 and the sum of all outputs will be 1, hence we can indeed interpret the output as a probability distribution.

Loss function

Now, that we have defined what the network computes for each input, we need to measure how well our network approximates our training data. A loss function is therefore introduced. Since the aim of the network is to classify the central pixel(s) of the patch, we use the categorical cross-entropy loss function

$$\mathcal{L}(\theta) = \frac{1}{m}\Big[\sum_{i=1}^{m}\sum_{j=1}^{k} \mathbb{1}[y^{(i)} = k]\log(P(y^{(i)}_{\text{pred}} = k \mid x^{(i)}; \theta))\Big] \tag{2.6}$$

where $\mathbb{1}$ is the indicator function and $P(y^{(i)}_{\text{pred}} = k \mid x^{(i)}; \theta)$ is the probability computed by the neural network with weights θ that input vector $x^{(i)}$ belongs to class k.

Optimisation

We now aim to minimise this loss function. This is done by using the stochastic gradient descent algorithm. First, we need to compute the gradient $\frac{d\mathcal{L}(\theta)}{d\theta}$, so that we can apply gradient descent and update our weight vector to

$$\theta = \theta - \epsilon\frac{d\mathcal{L}(\theta)}{d\theta} \tag{2.7}$$

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 differentiable loss function with respect to the weights in the network by repeatedly applying the chain rule. This process is called the backpropagation algorithm.

The loss functions, as described in equation 2.6 is computed using the entire training data set $\mathbb{X} = \{(x^{(1)}, y^{(1)}), ..., (x^{(m)}, y^{(m)})\}$. 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. A solution to this problem is to use stochastic gradient descent instead where the training data is split sequentially in batches. The loss and corresponding gradient updates are then computed individually for each batch.

Momentum

An 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 suggests that the gradient should only influence the velocity vector and not the position directly. This leads to the velocity update

$$v = \mu v - \epsilon \frac{d\mathcal{L}(\theta)}{d\theta} \tag{2.8}$$

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

$$\theta = \theta + v \tag{2.9}$$

Typically, a slightly different version called the Nesterov momentum, is used as it has been shown to work better in practice [22].

2.3.2 Convolutional neural networks

Convolutional neural networks make the explicit assumption that the inputs are images. This allows us to take advantage of some properties of images 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:

1. **Fully-connected layers**: These are identical to the layers in ordinary neural networks as introduced earlier.

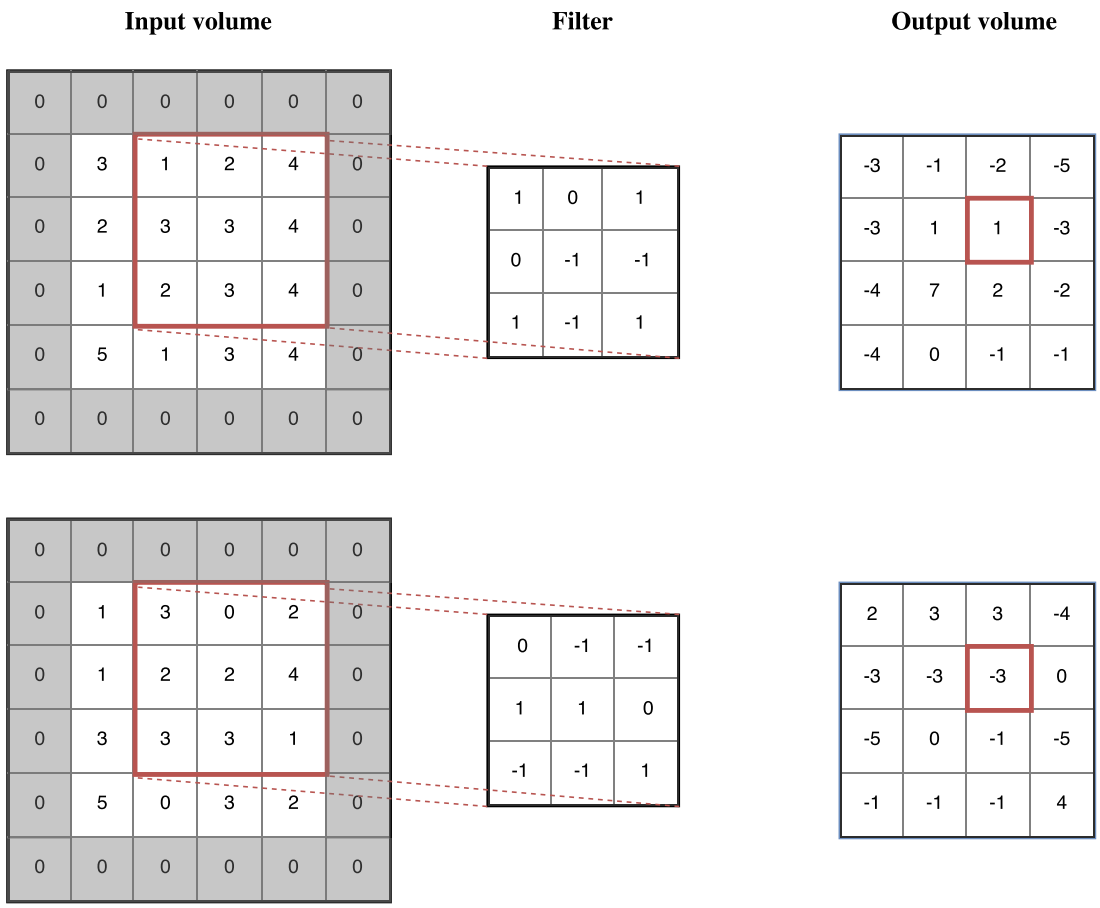

Figure 2.5: Example computation done by a 3x3 filter in a convolutional layer. Both the stride and the padding are 1, so as to keep the output dimensions identical. The output at depth 0 for the marked element is $1{\cdot}1+2{\cdot}0+4{\cdot}1+3{\cdot}0+3{\cdot}(-1)+4{\cdot}(-1)+2{\cdot}1+3{\cdot}(-1)+4{\cdot}1 = 1$ and similarly at depth 1 the output for that element is -3.

2. **Convolutional layers**: A convolutional layer is three-dimensional, consisting of a width, height and depth. The parameters of a layer define a set of learnable filters, each spatially small along the width and height of the layer but with the depth equal to the depth of the input volume. Each filter in the layer is convolved with the input and computes the dot product at each point. The result of this operation is a two-dimensional activation map, which has translational invariance, as the filter is convolved with the entire input and thus the same feature is detected independently of location. The computation done by such a filter in shown in figure 2.5. Finally, the two-dimensional activation maps are stacked along the depth axis to produce the output volume.

   This 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 value in an activation map 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 the following hyperparameters:

(a) The **depth** corresponds to the number of filters in the layer and is therefore equal to the depth of the output volume.

(b) The **stride** determines by how many pixels the filter is moved during each step of the convolution. When the stride is greater than 1 the output will be smaller than the input.

(c) The **padding** determines how many zeros are padded to the input width and height. This is particularly useful when we want to preserve the input dimensions.

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 layers and max–pooling layers. The idea is that each one of these pairs of convolutional and pooling layers can learn more and more abstract features using the features 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.

2.3.3 The overfitting problem

A common problem that arises with neural networks models is **overfitting**. This occurs when the model is not able to generalise on previously unseen data and instead just memorises the training data. Due to the large number of weights in deep convolutional neural networks, these are particularly prone to overfitting. Many techniques and heuristics have been developed in order to help prevent this; in my project I have used three of them: **L2 weight regularisation**, **Dropout** and **Batch Normalisation**, which I discuss below.

L2 weight regularisation

The L2 regularisation prevents individual parameters to grow without bounds, instead making the network use all its parameters, which has the effect of preventing the model from just memorising the data[6]. This is achieved by adding a regularisation penalty to the loss function, which is the sum of the squares of every parameter:

$$R(\theta) = \sum_k \sum_l \theta_{k,l}^2 \tag{2.10}$$

[6] `http://cs231n.github.io/neural-networks-2/#reg`

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

$$\mathcal{L}(\theta) = \mathcal{L}_{data}(\theta) + R(\theta) \tag{2.11}$$

where $\mathcal{L}_{data}(\theta)$ is the data loss, defined in equation 2.6.

Dropout

A more recent technique, developed for deep neural networks is Dropout [21]. When Dropout is used, the network randomly drops some neurons during the training phase. This forces the network to distribute the computation on all neurons evenly, which prevents the network from overfitting. The computation done by a single neuron thus becomes

$$\begin{aligned} \mathbf{r} &\sim \text{Bernoulli}_n(p) \\ y &= f_{act}((\sum_{i=1}^{n} r_i y_i w_i) + b) \end{aligned} \tag{2.12}$$

where $\mathbf{r}$ is a vector of n independent Bernoulli random variables each with parameter p, that is $P(r_i = 1) = p$ and $P(r_i = 0) = 1 - p$ for each r_i. This is done at every layer and therefore 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.

In practice, it is common 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. [17].

Batch Normalisation

The final regularisation technique used is Batch Normalisation [9]. 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 to prevent overfitting.

The important realisation is that the distribution of the inputs to each layer changes as the weights of the network are changed. This phenomenon is called *internal covariance shift*. The training of the neural network is slowed down by this because each layer has to continuously adapt to this change in the distribution of its inputs. 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:

$$\begin{aligned} \mu_{\mathbb{B}} &= \frac{1}{m}\sum_{i=1}^{m} x_i^{(k)} \\ \sigma_{\mathbb{B}}^2 &= \frac{1}{m}\sum_{i=1}^{m}(x_i^{(k)} - \mu_{\mathbb{B}})^2 \\ \hat{x}_i^{(k)} &= \frac{x_i^{(k)} - \mu_{\mathbb{B}}}{\sqrt{\sigma_{\mathbb{B}}^2 + \epsilon}} \\ \mathrm{BN}_{\gamma,\beta}(x_i^{(k)}) &= \gamma\hat{x}_i^{(k)} + \beta \end{aligned} \tag{2.13}$$

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

$$y = f_{act}(\mathrm{BN}_{\gamma,\beta}(\sum_{i=1}^{n} y_i w_i)) \tag{2.14}$$

where the Batch Normalisation transformation is applied to each dimension individually. 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 [9].

2.4 Data source

2.4.1 BraTS Challenge

For my project, I use the dataset provided by BraTS2013[14] challenge. It is split into three sections:

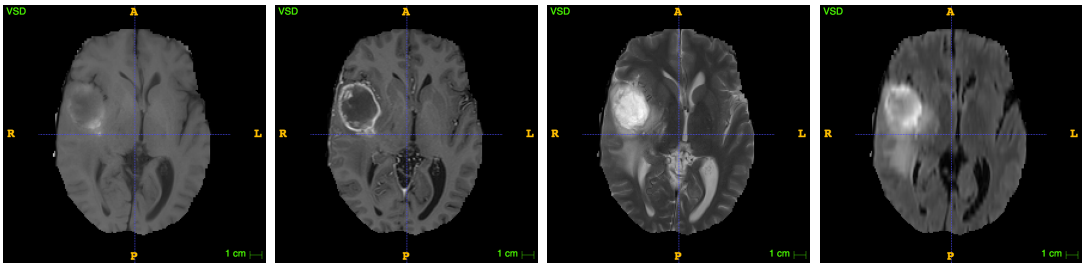

Figure 2.6: Example slices in the axial plane for the 4 different scan modalities. From left to right the modalities are T1, T1c, T2 and Flair. The scan is from patient 1, slice $z = 89$. The four modalities have some obvious differences which the convolutional neural network might be able to utilise to discriminate between tumour and non-tumour regions.

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 ground truth annotation. 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 publicly available.

The challenge and leaderboard sets are 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.

Each patient consists of 4 images taken using different MRI contrasts: T2 and FLAIR MRI which highlight differences in tissue water relaxational properties, and T1 MRI and T1c MRI which show pathological intratumoural take-up of contrast agents. 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. The highlight differences for the 4 modalities can be seen in figure 2.6, which shows the same axial plane slice for a patient for each modality.

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

The training dataset has been manually segmented by 4 different human experts into the 5 classes described in the introduction. These segmentations are then merged together

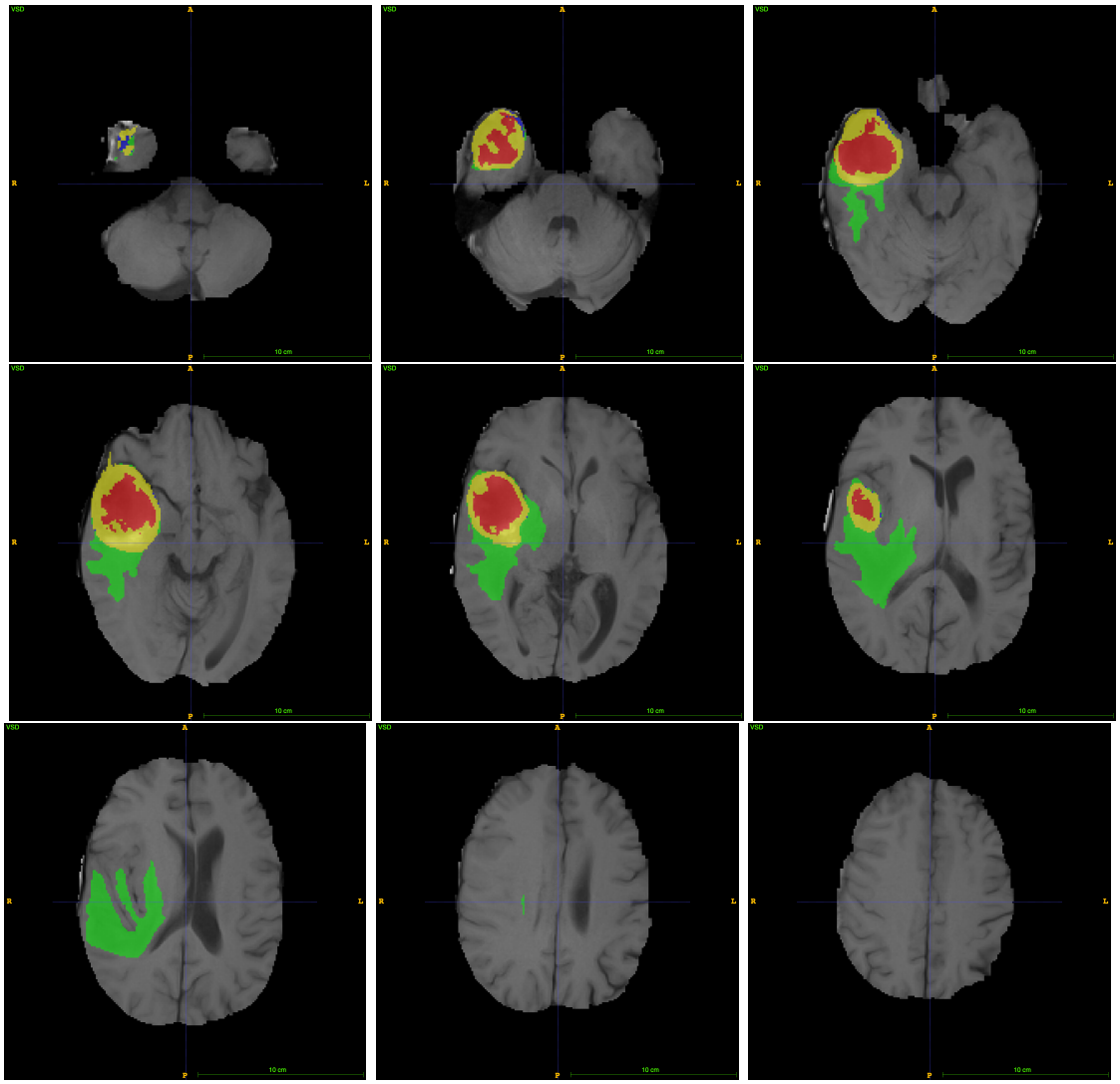

Figure 2.7: Axial plane slices of a T1 MRI scan annotated by the expert labelling. The slices were taken from patient 1, using $z \in \{49, 59, 69, 79, 89, 99, 109, 119, 129\}$. The enhancing tumour region is in yellow, the non-enhancing tumour in blue, the edema in green and the necrotic core in red, corresponding to classes 4, 3, 2 and 1 respectively.

to provide a single ground truth segmentation. As an example of what such a segmentation looks like I have included figure 2.7 which shows different slices of a T1 MRI scan annotated with the ground truth.

2.4.2 High grade gliomas

For this project I have decided to focus only on the high-grade glioma cases. I made this choice for the following reasons:

1. To keep the scope of the project within that of a Part II project. Pereira et al. propose a different model for the low-grade glioma case which I would have had to

replicate as well, possibly doubling the amount of work needed during the training phase.

2. 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.

3. 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. Accordingly, I only use the 20 high-grade glioma cases of the training dataset to train the convolutional neural networks.

Chapter 3

Implementation

This chapter is divided into two parts. The first part describes the implementation for the training of a model. The second part describes how a trained model is used to segment MRI scans.

3.1 Training the model

3.1.1 Scan normalisations

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 in which the images are made available. 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 $(z_{\text{size}} \times y_{\text{size}} \times x_{\text{size}} \times 4)$.

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 shows how the sizes are distributed among patients. Because of this, the data for the patients cannot be aggregated into a single 5-dimensional numpy array. Instead, I aggregate those arrays into a single python list containing all the training data.

The model proposed by Pereira et al. [17] uses the Nyul normalisation [15]. This normalisation requires human input, preferably from a domain expert. In order to make the process fully automatic, I use another method, also reported in Pereira's paper that I describe next.

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 $(z \times y \times z)$

Winsorising

The first normalisation that is applied to entire scans 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 1^{st} percentile are set to the value of the 1^{st} percentile and the value above the 99^{th} percentile are set to the value of the 99^{th} percentile. Note that this process is different from trimming as the values are not discarded but just clipped.

N4 Bias field correction

The second normalisation applied to scans is the N4 bias field correction [24]. MRI scanner can create a bias in which the intensity of the same tissue varies across the produced scan, making it hard for automated segmentation algorithms to recognise that it is in fact the same tissue. N4 is a variant of the popular nonparametric nonuniform intensity normalisation N3 which aims at eliminating this bias introduced by MRI scanners.

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 3.2 lists the parameters I used to perform the normalisation. I first used the default parameters and later contacted Pereira who kindly agreed to share the parameters he used in his published method, which are reported here. I used the implementation provided by the 'Nipype'[1] and the 'Advanced Normalization Tools'[2].

Figure 3.1 shows the effects that winsorising and then applying the N4 bias field correction has on scans for each of the four modalities T1, T1c, T2 and Flair.

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

[2] `http://stnava.github.io/ANTs/`

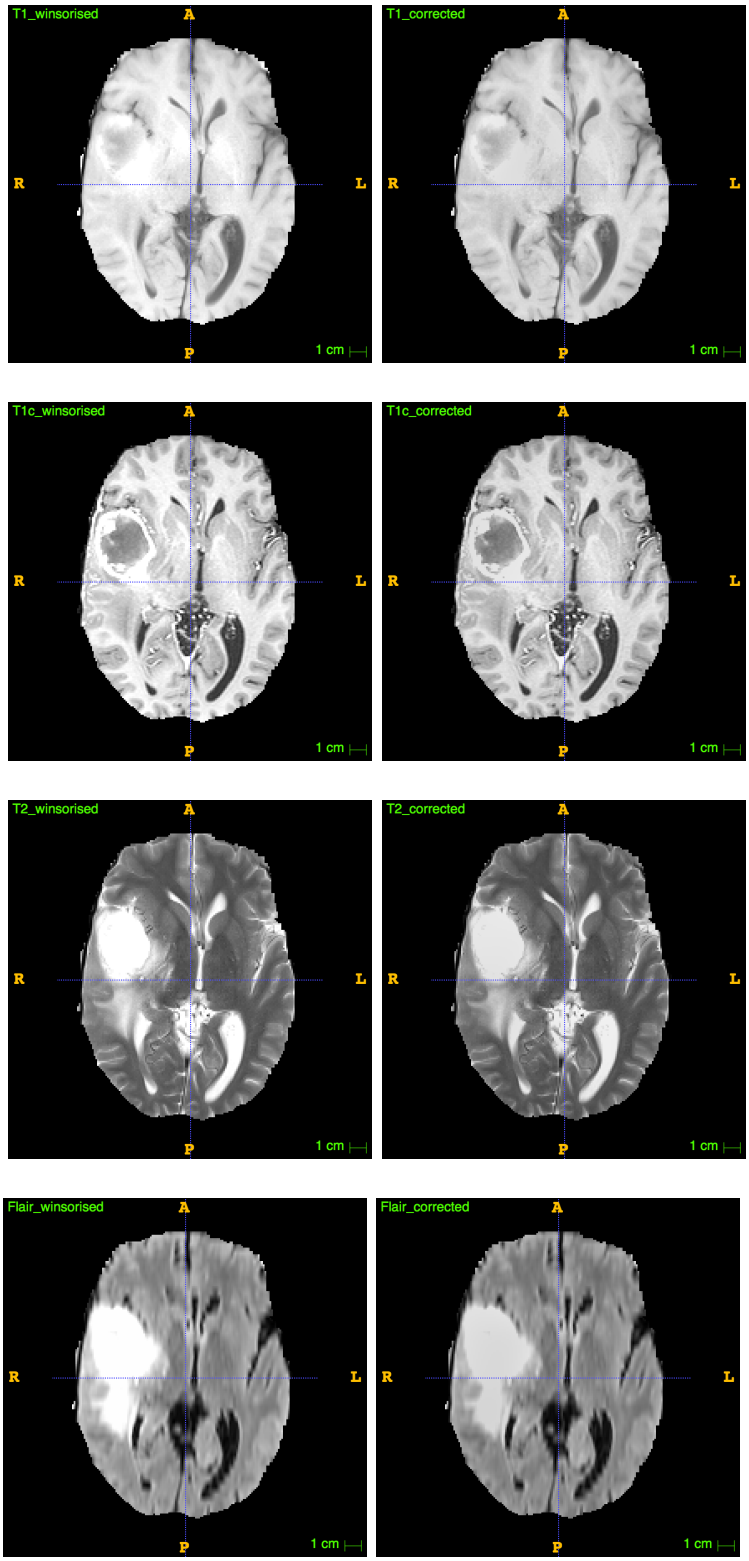

Figure 3.1: Example of the effects of winsorising and applying the N4ITK correction applied to scans for each of the four modalities. The first image in each row is obtained by winsorizing the original image and the last image is obtained by applying the N4ITK correction to the winsorised scan. From top to bottom, slice $z = 89$ is shown for patient 1 for the T1, T1c, T2 and Flair scans.

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

Table 3.2: Parameters used to perform the N4ITK correction.

Mean and Variance standardisation

The last step is to standardise the mean and variance for each scan along each slice. This is done by first subtracting the mean value of the voxels in the slice and then dividing by the standard deviation of those values.

$$x' = \frac{x - \mu}{\sigma} \tag{3.1}$$

As $\mathbb{E}[aX] = a\mathbb{E}[X]$ and $\mathrm{Var}[cX] = c^2\mathrm{Var}[X]$ one can show 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.

I use numpy's `mean` and `std` functions and the fact that basic operations on numpy arrays are done element-wise to implement this normalisation.

3.1.2 Patch extraction

Each input to the convolutional neural network is a three-dimensional array, of shape $\mathsf{height} \times \mathsf{width} \times 4$, since it consists of a two-dimensional patch of $\mathsf{width} \times \mathsf{height}$ voxels for each one of the 4 scan modalities. The two-dimensional patches are taken along the x–y axis, also called the axial plane in anatomy. Figure 3.2 gives a diagrammatic overview of the transformation for the simplified case with a single modality, meaning that the image array is three-dimensional and the resulting patch is two-dimensional, instead of four and three-dimensional respectively.

There are two issues that need to be resolved before extracting the patches:

1. The training data has to be balanced, that is the same number of examples for each class should be included in the training data. This is to ensure that the convolutional model is able to generalise well. However, the BraTS dataset is extremely unbalanced with over 98% of the voxels belonging to class 0, as shown in table 3.3. Therefore, simply randomly picking patches from the dataset does not work and a more complex method has to be used. Also note that a fixed balanced dataset would limit the maximum number of patches we can extract to 5 times the number

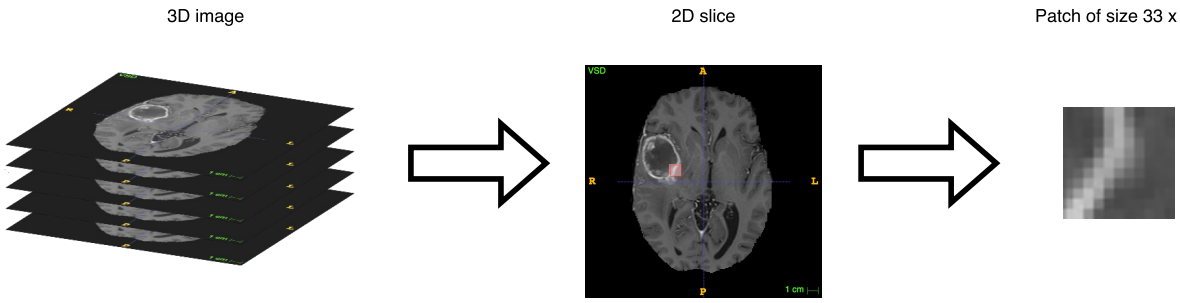

Figure 3.2: Patch extraction process for a three dimensional array. The actual image array is four dimensional, the 4th dimension being the different modalities. Here width = height = 33 and therefore corresponds to the patch size proposed by pereira et al.

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 3.3: 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.

of voxels in the least represented class. This is the non-enhancing tumour class (class 3) with 184,436 labeled voxels, hence a maximum of 922,130 patches. To increase the number of different patches the network sees during training for the overrepresented classes while keeping the data balanced, I randomly resample at each epoch the same number of patches for each class from all possible patches for that class.

2. It is not possible to extract an entire patch around the voxels too close to the edge of the scan. To solve this issue there are two options: I could either ignore the patch entirely or pad the scan such that each labelled voxel is surrounded by enough voxels. Only the first option is considered as it helps with the balanced classes issue as most of the voxels close to an edge are from class 0. Furthermore, as the brains are centred within the MRI scans, voxels close to an edge are surrounded only by voxels of value 0, making the patch identical to many other patches close to an edge. Table 3.4 shows the distribution of classes for "valid" voxels, that is those at least more than half the patch length away from the x or y edges for the Pereira model with width = height = 33. Indeed, most ignored voxels come from class 0.

In personal correspondance, Pereira mentioned that he used two further heuristics for extracting patches from class 0 that were not mentioned in his paper, which I also implemented:

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 3.4: 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.

1. Half of the patches for class 0 are selected close to the tumour, where close means that the central voxel is within the bounding box of the diseases tissue. This helps the network to learn how to draw the boundaries between class 0 and the other classes.

2. The other half of the training patches for class 0 are spaced apart with a distance of at least 3 voxels in the z, y and x directions. This spreads the input data to the entire scan.

Patch extraction algorithm

To make the resampling of the patches at each epoch more efficiently, I first create a list (`valid_positions`) containing for each class the positions of the valid patches (really the position of the central voxel) for that class. Note that at this point it would be unfeasible to store all possible 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. I use equality checking between the voxel label and the class number. This is done for every patient and returns an array consisting of the (z, y, x) coordinates of valid patches for every patient. Since the patient number must also be included in the indices, I prepend the patient number along the first axis, resulting for each patient in an array consisting of the $(\text{patient}, z, y, x)$ coordinates and aggregate the results for each patient into a single list. The second step consists of removing those voxels that are too close to an x-axis or y-axis edge. Only indices $(\text{patient}, z, y, x)$ where the x and y values are within the allowed ranges defined by half the size of the patch: $[\lfloor \frac{\mathsf{width}}{2} \rfloor, \mathsf{width}_{\text{scan}} - \lceil \frac{\mathsf{width}}{2} \rceil]$ and respectively for the height 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.

At the beginning of each training epoch, I then randomly choose the same number of positions for each class from `valid_positions` using numpy's `random.choice` function.

Algorithm 1 Patch extraction

for class k in $[0, 1, 2, 3, 4]$ **do**
    valid_positions$[k] \leftarrow []$
    for each index **in** len(labels) **do**
        label $\leftarrow$ labels[index]
        possible_indices $\leftarrow$ *numpy*.argwhere(label $== k$)
        patient_indices $\leftarrow$ *numpy*.full(*len*(possible_indices), index)
        possible_indices $\leftarrow$ *numpy*.append(patient_indices, possible_indices, axis = 1)

        possible_indices $\leftarrow$ possible_indices[possible_indices$[:, 2] \geq \lfloor \frac{\text{height}}{2} \rfloor$]
        possible_indices $\leftarrow$ possible_indices[possible_indices$[:, 2] \leq \text{height}_{\text{label}} - \lceil \frac{\text{height}}{2} \rceil$]
        possible_indices $\leftarrow$ possible_indices[possible_indices$[:, 3] \geq \lfloor \frac{\text{width}}{2} \rfloor$]
        possible_indices $\leftarrow$ possible_indices[possible_indices$[:, 3] \leq \text{width}_{\text{label}} - \lceil \frac{\text{width}}{2} \rceil$]

        valid_indices$[k] \leftarrow$ possible_indices
    end for
end for

For each position I extract the corresponding patch and label which are returned as training data.

3.1.3 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 are used. Since the data consists entirely of two-dimensional patches, translation cannot be used as it would result in a different patch, with a possibly different label. However, using rotations of the patches might give some performance improvements. Pereira et al. propose the following variants in [17]:

1. No rotations
2. Rotations of 90, 180 and 270 degrees.
3. Uniformly sample three rotations from an array of equally spaced angles. The angle step proposed is $\frac{1}{16} \times 90° = 6°$

Pereira et al. reported that using rotations of multiples of 90 degrees performed the best, which is why I decided to implement it. I perform the rotations with a custom Keras 'data generator'. Such a data generator receives as an input a batch of data and returns another batch of data. Here, the generator randomly rotates each training patch in the batch using the numpy `rot90` function which rotates arrays by steps of 90 degrees.

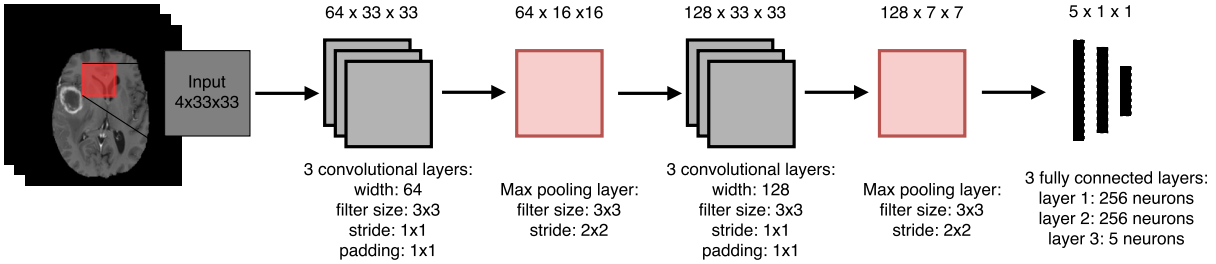

Figure 3.3: Convolutional network architecture proposed by Pereire et al.

3.1.4 Training: Pereira model

Architecture

The model proposed by Pereira et al. [17] 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. Using three layers of size 3×3 consecutively effectively has a receptive field of size 7×7, but has fewer parameters then a single 7×7 convolutional layer would have, reducing the overall number of parameters and therefore making the network less prone to overfitting [20]. The next layer is a max pooling layer of size 3×3 and stride 2×2. This is an unusual design choice for convolutional networks as the size is typically smaller than the stride. Pereira et al. motivate this choice by arguing that 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 more consecutive 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 3.1.4 shows the model architecture in more details including the number of weights in each layer.

The weights in the convolutional layers are initiliased using Xavier normal initialisation [4]. The biases are initialised to the constant value 0.1, except for the last layer.

The activation function used throughout the network is the leaky rectifier, with $\alpha = 0.333$.

Of the three techniques presented for avoiding overfitting in the preparation chapter, Pereira uses Dropout [21] in the fully-connected layers, with probability $p = 0.1$ of removing a node from the network.

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.

Implementation of the architecture in Keras

The network proposed by Pereira et al. is sequential, that is the layers are stacked linearly. Using the Keras `Sequential` model is therefore the simplest way to implement it. The `Sequential` model makes it possible to create networks by adding layers to it sequentially. The three types of layers used in the network have available Keras implementations, and therefore building the network itself was relatively straightforward. First, an instance of a sequential model is instantiated:

```
model = Sequential()
```

Then, layers can be added by calling the `add` function on 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, width, height, border_mode='same', init=
  'glorot_normal')
model.add(conv)
```

The library makes it easy for the user to specify how the weights should be initialised by setting the `init` argument. The `border_mode` argument determines the size of the padding that is added to input before applying the convolutional layer. In our case the output of the layer should have the same size as the input, which is specified by setting the argument to 'same'. Max pooling layers and fully-connected layers can be added similarly using instances of `MaxPooling2D` and `Dense` classes respectively.

Activation functions are added in the same way as layers. Hence, to add a leaky rectifier, also 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 similarly:

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

Finally, before training the model, we need to specify which loss function and which algorithm should be used to respectively specify and minimise the loss function. In Keras this is called 'compiling' the model:

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

Training

Pereira's paper specifies that the model was trained over 20 epochs, each training the network on 450,000 patches, that is 90,000 patches per class. The optimisation algorithm used is a standard stochastic gradient descent algorithm with Nesterov momentum, with constant momentum $\mu = 0.9$. The learning rate is linearly decreased from 3×10^{-5} to 3×10^{-7}. As Keras only an implementation for exponential learning rate decay has, I use algoritm 2 to manually specify the learning rate at each epoch.

Algorithm 2 Model training with linear learning rate decay

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

At the end of each epoch the loss function is evaluated on the validation data and the accuracy (equation 3.2) is computed. The loss and accuracy are then reported to the standard output. Figure 3.4 shows part the output of training a network for a few epochs.

$$\text{accuracy} = \frac{1}{m}\Big[\sum_{i=1}^{m} \mathbb{1}[y^{(i)} = \arg\max_{k} P(y^{(i)} = k)]\Big] \tag{3.2}$$

After training the model for 20 epochs, I save the model with its weights in a directory specified as an argument to the main python script, so that I can reload the model for the segmentation of MRI scans.

In figure 3.5 the validation loss and validation accuracy are plotted for the 20 epochs. Both seem to converge over this period. The validation accuracy is higher than the training

```
Epoch 9/20
Extracting 225000 training_samples
[(45000, 4, 33, 33), (45000, 4, 33, 33), (45000, 4, 33, 33), (45000, 4, 33, 33), (45000, 4, 33, 33)]
Input data shape (225000, 4, 33, 33) (225000, 5)
Validation data shape (500, 4, 33, 33) (500, 5)
lr: 1.7494736312073655e-05
Epoch 1/1
225000/225000 [==============================] - 86s - loss: 0.2867 - acc: 0.8900 - val_loss: 0.2770 - val_acc: 0.8840
Extracting 225000 training_samples
[(45000, 4, 33, 33), (45000, 4, 33, 33), (45000, 4, 33, 33), (45000, 4, 33, 33), (45000, 4, 33, 33)]
Epoch 1/1
225000/225000 [==============================] - 86s - loss: 0.2837 - acc: 0.8902 - val_loss: 0.2706 - val_acc: 0.8880
Epoch 10/20
Extracting 225000 training_samples
[(45000, 4, 33, 33), (45000, 4, 33, 33), (45000, 4, 33, 33), (45000, 4, 33, 33), (45000, 4, 33, 33)]
Input data shape (225000, 4, 33, 33) (225000, 5)
Validation data shape (500, 4, 33, 33) (500, 5)
lr: 1.5931578673189506e-05
Epoch 1/1
225000/225000 [==============================] - 86s - loss: 0.2797 - acc: 0.8921 - val_loss: 0.2671 - val_acc: 0.8820
Extracting 225000 training_samples
[(45000, 4, 33, 33), (45000, 4, 33, 33), (45000, 4, 33, 33), (45000, 4, 33, 33), (45000, 4, 33, 33)]
Epoch 1/1
225000/225000 [==============================] - 85s - loss: 0.2746 - acc: 0.8943 - val_loss: 0.2697 - val_acc: 0.8840
```

Figure 3.4: Part of the of the training script. The output is shown for epochs 9 and 10. Both the validation loss and the validation accuracy are reported, as well as how long it took to train for this epoch. Because all 450,000 patches cannot be held at once in memory, the epoch is divided into two training phases, each on 225,000 patches.

accuracy because of Dropout: during validation no neurons are dropped, meaning that the entire network is used which is therefore able to classify the data more accurately.

3.1.5 Training: My model

In the second phase of my project I designed a convolutional neural network to perform the segmentations. In this subsection I justify the design choices I made and explain the modifications necessary for its implementation.

Architecture

The design of the architecture for my model is based on the following points:

1. I increase the size of the input patches, almost doubling it to 64×64. Using this bigger patch, the model should be able to incorporate more contextual information from voxels further away in the slice for the classification. Figure 3.6 shows the difference in patch size.

2. To speed up the segmentation and to prevent an explosion of the number of weights, instead of only classifying the central voxel, the network classifies the central 8×8 voxels of a patch. The output of the network is therefore no longer a 5×1 array

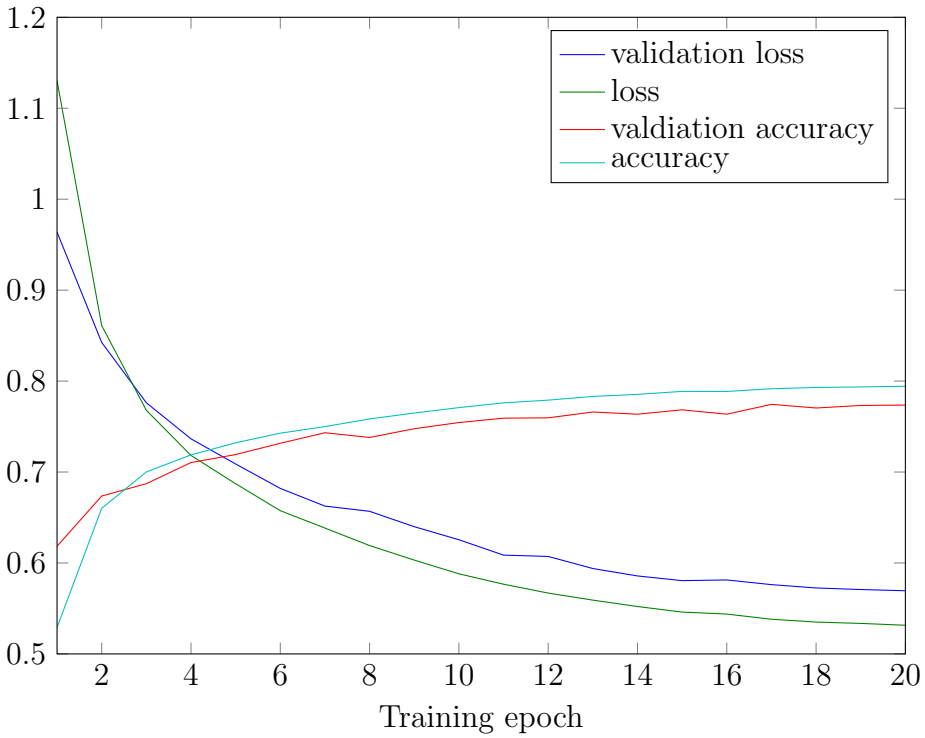

Figure 3.5: Validation loss and accuracy during training for the model proposed by Pereira et al. Both seem to converge over this period.

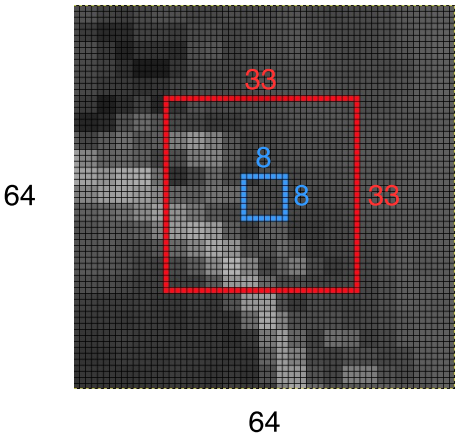

Figure 3.6: Example patch for my model compared to a patch for the model proposed by Pereira et al. [17]. The patch for the Pereira model is drawn in red. The central blue square is the region classified by my model, of size 8×8.

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' of the input.

3. I 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 [9].

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, that is, there are only convolutional layers and no fully-connected layers. This choice is justified by 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 faster to train and can predict outputs much faster, due to the fact that it mainly applies convolutions, which are faster then the operation computed by a fully-connected layer. 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 question is how using a larger output and a fully-convolutional network will affect the performance of the model. This will be discussed in the Evaluation chapter.

My model contains 13 layers, shown in table 3.6. 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 more 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 class probability distribution for each one of the central 64 voxels. The kernel weights are initialised using a Xavier normal heuristic [4]. The biases are all initialised to 0.

Implementation

The implementation of my model is similar to how I implemented the model proposed by Pereira et al. This is the main advantage of using a library such as Keras. However, there is a subtlety that arises from the fact 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

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
		2 485 253

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

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(l)))
model.add(BatchNormalization(axis=axis, trainable=trainable))
model.add(LeakyReLU(alpha))
```

Training

I train my model over 40 epochs using 100,000 patches per epoch. To minimise the loss function I use the Adaptive Moment Estimation method (adam) [11] which is a stochastic gradient descent method that computes adaptive learning rates for each parameter. This method is implemented in Keras and can therefore be specified when compiling the model as follows:

```
adam = Adam()
  model.compile(optimizer=adam, loss='categorical_crossentropy', metrics
  =['accuracy'])
```

3.2 Segmenting MRI scans

The first stage, normalising the scans, is identical to the first stage for training the model and is independent of the model used to segment the scans. The patch extraction is however different, as the patches have to be extracted such that every voxel in the input scan can be classified by the model. As the implementation of this is more complicated for my model, I describe the implementation of the patch extraction and the segmentation stages independently for the two models.

3.2.1 Pereira model

For the model proposed by Pereira et al. [17], 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 axial 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. Normalise the image as explained in section 3.1.1. Thus, the image is first winsorised, then the N4ITK correction is applied to it, and finally for each modality the mean and variance are normalised.
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 $z = z_i$ in the axial plane, extract all patches and store them in a list. Note that this has to be done for each slice sequentially as it would not be possible to store all patches for the entire image simultaneously in memory. 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 makes 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 $y \cdot x$, which is the $z = z_i$ slice in the axial plane of the segmentation.

Figure 3.7 shows the resulting segmentation for the first challenge patient.

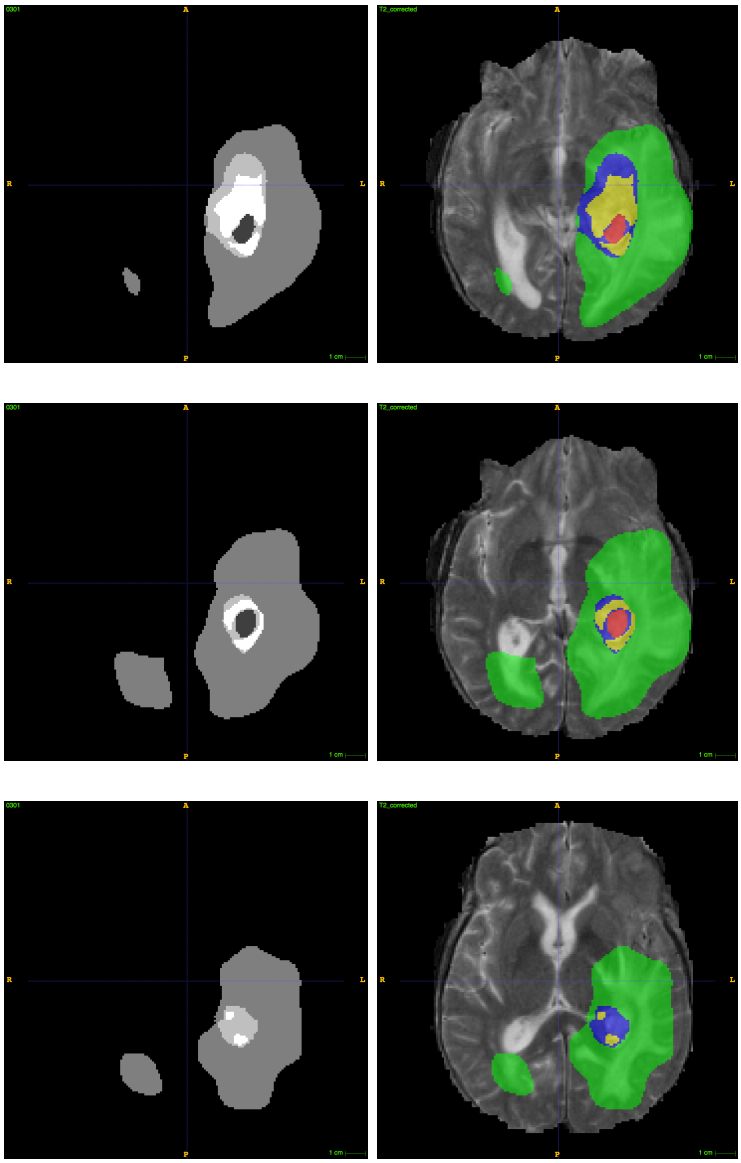

Figure 3.7: Left column: The segmentation for patient 1 in the challenge dataset for axial plane slices for $z \in \{66, 71, 76\}$. Right column: The same segementation slices overlaid on the T2 scan.

3.2.2 My model

As my model classifies the central 8×8 voxels for each input patch, the segmentation is slightly more complicated. The padding has to be done differently, as it depends on the size of the image. 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 end of the x and y axes by an extra 8 voxels. Thus, the model is padded by $\frac{64-8}{2} = 28$ zeros in front of the x and y axis and the end is padded with $\frac{64-8}{2} + 8 = 36$ zeros. Then, for each slice in the image, the steps computed are:

1. Extract all patches of size $64 \times 64 \times 4$ for the entire slice, so that the center 8×8 squares of each patch do not overlap but cover the entire input image. This can be done after padding by two nested loops with step increments of 8.

2. Iterate over the patches in batches, such that each batch fits into the memory of the GPU and use the convolutional neural network to predict the class for the central $8 \cdot 8 = 64$ voxels of each patch. These are returned in a one-dimensional array.

3. Concatenate all the predictions for all batches into a single one-dimensional array of length $64 \cdot \#patches$. This one-dimensional array has to be transformed back into a two-dimensional array such that each voxel is mapped back into its original position in the input image.

4. Calculate the number of patches that fit into the slice as follows:

$$\begin{aligned} \text{height}_p &\leftarrow \left\lceil \frac{\text{image\_height}}{8} \right\rceil \\ \text{width}_p &\leftarrow \left\lceil \frac{\text{image\_width}}{8} \right\rceil \end{aligned} \tag{3.3}$$

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 3-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, 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. 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.

3.3 Segmentation 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, based on the assumption that the diseased tissue be connected into a single component of relatively large size. Pereira et al [17] proposed to remove all connected components (continuous regions of diseased tissue) of less than 10,000 voxels in volumes.

I implement this heuristic using a depth first search on the graph represented by a scan, where each voxel is a vertex and shares 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. This array is used to mark which nodes have already been visited and processed. As only the diseased tissue is considered, i.e. classes 1–4, I mark all voxels of class 0 as processed. 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. Take an voxel that has not been visited, mark it as processed and add its coordinates to C.
3. Initialise a queue containing all neighbours of the voxel that have not yet been visited and are segmented as diseased tissue. Mark those neighbours as visited.
4. Then, repeat while the queue is not empty:
   (a) Pop the first element, mark it as processed, and add its coordinates to the connected component C.
   (b) Find its neighbours that have not been visited or processed. Mark those as visited 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 3.8 shows the effect of applying this post-processing step to the last challenge scan.

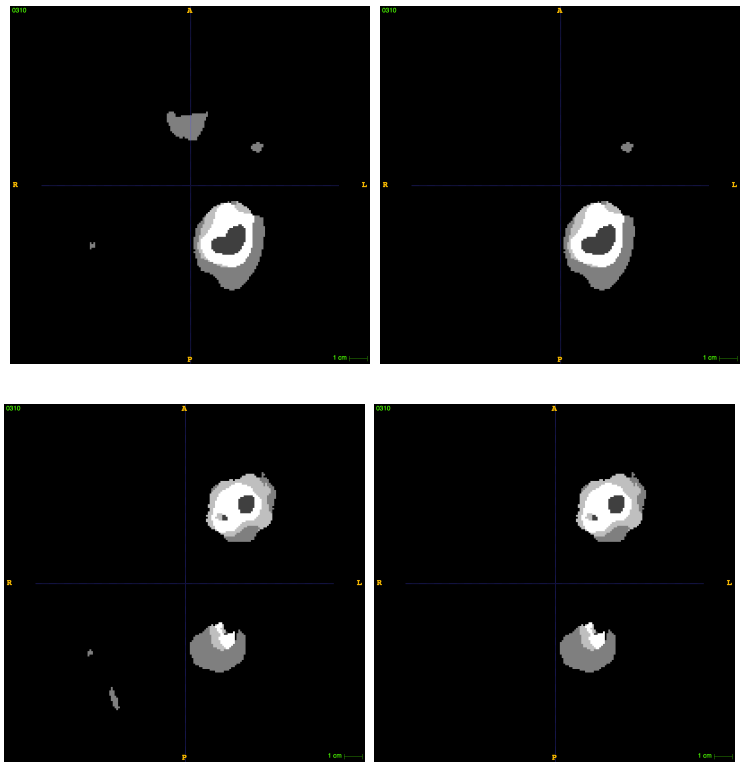

Figure 3.8: Effect of applying the post-processing step to the 10^{th} challenge scan. Axial plane slices $z \in \{77, 87\}$ are shown side-by-side (on the left before the post-processing step and on the right with the post-processing step).

Chapter 4

Evaluation

4.1 Unit Tests

To verify that the stages behave as expected, I implement some unit tests using the Python `unittest`[1] module as it does not require any additional libraries. Furthermore, the module was inspired by JUnit, which I had previously used during other projects.

For the normalisation stage, I used the ANTs neuroimaging tools and scipy to perform the N4 correction and the winsorising step respectively, thus I only tested the mean and standard deviation normalisation. I tested the patch extraction phase with small 4-dimensional arrays, to check that the correct patches are returned and that the patch labels are balanced. The data augmentation stage does not need to be unit tested as it is implemented with a Keras 'data generator'. I tested the segmentation stage similarly, using small arrays and a simple classifying function to mock the behaviour of the convolutional neural network. Finally, I also tested the post-processing stage for different inputs with known outputs.

As an example, I include two of the tests for the post-processing step:

```
class TestRemoveConnectedComponents(unittest.TestCase):

    def test_no_components(self):
        image = np.full(shape=(3,3,3), fill_value=0)
        image2 = np.copy(image)
        remove_connected_components(image2, 1, verbose=False)
        self.assertTrue(np.array_equal(image, image2))

    def test_remove_one_component(self):
        image = np.array([[[0,0,0],[0,0,0],[0,0,0]],
                          [[0,0,0],[0,1,0],[0,0,0]],
                          [[0,0,0],[0,0,0],[0,0,0]]])
        remove_connected_components(image, 2, verbose=False)
```

[1] `https://docs.python.org/3/library/unittest.html`

```
14          self.assertTrue(np.array_equal(image, np.full(image.shape, 0)))
```

Figure 4.1 shows the output of running the unit tests. All unit tests pass, indicating that the tested stages behave as expected.

```
test_extract_training_patches (test.testDataExtractorWithLargerArray.TestDataExtractorWithLargerArrayInput) ... ok
test_find_training_patches_close_to_tumour (test.testDataExtractorWithLargerArray.TestDataExtractorWithLargerArrayInput) ... ok
test_find_valid_patches_coordinates (test.testDataExtractorWithLargerArray.TestDataExtractorWithLargerArrayInput) ... ok
test_find_valid_patches_coordinates_with_large_patches (test.testDataExtractorWithLargerArray.TestDataExtractorWithLargerArrayInput) ... ok
test_extract_training_patches (test.testDataExtractorWithSimpleArrayInput.TestDataExtractorWithSimpleArrayInput) ... ok
test_find_training_patches_close_to_tumour (test.testDataExtractorWithSimpleArrayInput.TestDataExtractorWithSimpleArrayInput) ... ok
test_find_valid_patches_coordinates (test.testDataExtractorWithSimpleArrayInput.TestDataExtractorWithSimpleArrayInput) ... ok
test_find_valid_patches_coordinates_with_large_patches (test.testDataExtractorWithSimpleArrayInput.TestDataExtractorWithSimpleArrayInput) ... ok
test_extract_patches (test.testFCNSegment.TestFCNSegment) ... ok
test_pad (test.testFCNSegment.TestFCNSegment) ... ok
test_segment_to_constant (test.testFCNSegment.TestFCNSegment) ... ok
test_normalize_channel (test.testNormalization.TestNormalization) ... ok
test_normalize_scans (test.testNormalization.TestNormalization) ... ok
test_diagonals (test.testRemoveConnectedComponents.TestRemoveConnectedComponents) ... ok
test_no_components (test.testRemoveConnectedComponents.TestRemoveConnectedComponents) ... ok
test_remove_large_3d_component (test.testRemoveConnectedComponents.TestRemoveConnectedComponents) ... ok
test_remove_multiple_components (test.testRemoveConnectedComponents.TestRemoveConnectedComponents) ... ok
test_remove_one_component (test.testRemoveConnectedComponents.TestRemoveConnectedComponents) ... ok
test_threshold (test.testRemoveConnectedComponents.TestRemoveConnectedComponents) ... ok
test_pad (test.testSegment.TestSegment) ... ok
test_segment_to_constant (test.testSegment.TestSegment) ... ok

----------------------------------------------------------------------
Ran 21 tests in 0.141s

OK
```

Figure 4.1: Output of running the unit tests. All tests pass successfully.

4.2 BraTS evaluation

4.2.1 Evaluation metrics

The BraTS2013 challenge [2] evaluates the segmentation using three different metrics: **dice score**, **sensitivity** (or recall) and **positive predictive value** (or precision). These metrics are best visualised using the Venn diagram shown in figure 4.2.1. Note that accuracy is not used. This is because the data is highly unbalanced, as shown previously, meaning that a model trivially labelling everything with class 0 would reach an accuracy of 99% or more, making it very hard to compare different models.

Positive predictive value

The positive predictive value is defined as

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

where TP and FP are the numbers of true and false positives respectively. In the Venn diagram in figure 4.2.1, 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

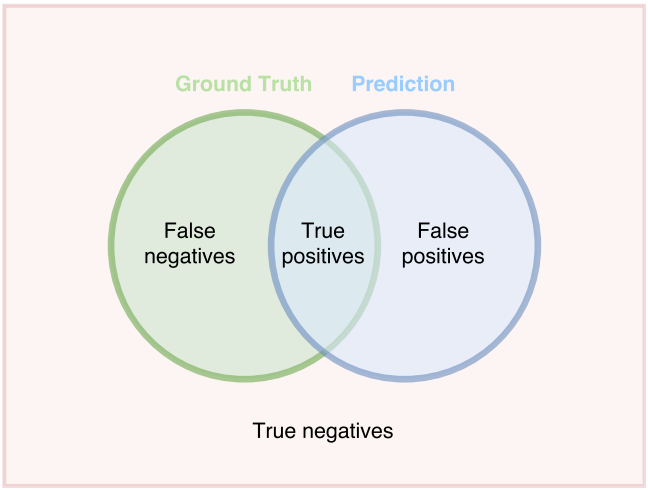

Figure 4.2: Venn diagram showing the different areas and how the can be classified.

positively classified elements, the positive predictive value gives an indication as to how accurately the model can predict a positive class. In the case of brain tumour segmentations, the positive predictive value measures how confidently the model is predicting tumours. Thus, a high positive predictive value means those regions labelled as tumours 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 and TP is the number of true positives. The sensitivity measures the size of the blue and green region relative to the size of the entire green region. As the denominator $\text{TP} + \text{FN}$ equals the green area, that is 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.

Trade-off between positive predictive value and sensitivity

There is a trade-off between the sensitivity and the positive predictive value. A model classifying everything as negative would reach perfect positive predictive value 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 because 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. Thus, the Dice score is used to mitigate this trade-off, as explained next.

Dice score

The dice score is defined in terms of both the false negatives and the false positives and thus defines a metric which tries to take this trade-off into account. 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 classifications to the average size of the two segmented areas.

Figure 4.2.1 shows an example for a segmented slice in the axial plane. The metrics for this segmentation are

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

4.2.2 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 as 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.

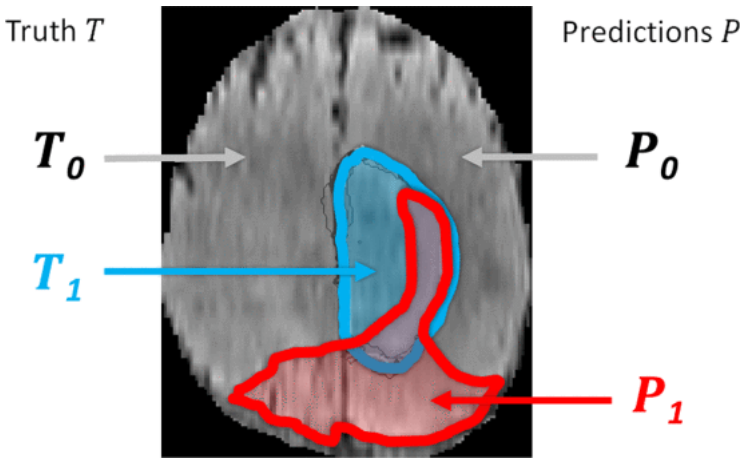

Figure 4.3: Example showing a segmentation for a slice in the axial plane. This image was reproduced from [2]

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.

4.3 Evaluation of the model proposed by Pereira et al.

4.3.1 BraTS evaluation

Dice score

Table 4.1 shows the Dice scores I obtained for each of the 10 challenge scans using the method proposed by Pereira et al. [17]. In figure 4.4 the mean and standard deviation are plotted for the three regions along with the minimum and maximum values.

The mean scores for the 'Core' and 'Enhancing' region are significantly lower than the mean score for the 'Complete' region and have a much higher standard deviation. This can partially be explained by the outlier values obtained from patient 7. For this patient, the diseased tissue has been segmented correctly, but the model was not able to distinguish the different classes within the diseased tissue, and it 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 are very close to 0.

However, even if the results from patient 7 were ignored, the average results for regions 2 and 3 are still below those reported by Pereira et al. [17] of (0.8, 0.78, 0.73). As I

Patient	**Dice score**		
	Complete	Core	Enhancing
1	0.813	0.843	0.472
2	0.796	0.700	0.619
3	0.808	0.765	0.345
4	0.723	0.426	0.178
5	0.750	0.676	0.550
6	0.776	0.510	0.518
7	0.818	0.035	0.034
8	0.829	0.878	0.700
9	0.881	0.883	0.818
10	0.852	0.840	0.840
mean	0.805 ± 0.047	0.656 ± 0.267	0.508 ± 0.262

Table 4.1: Dice scores obtained for the 10 challenge patients using my implementation of 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.

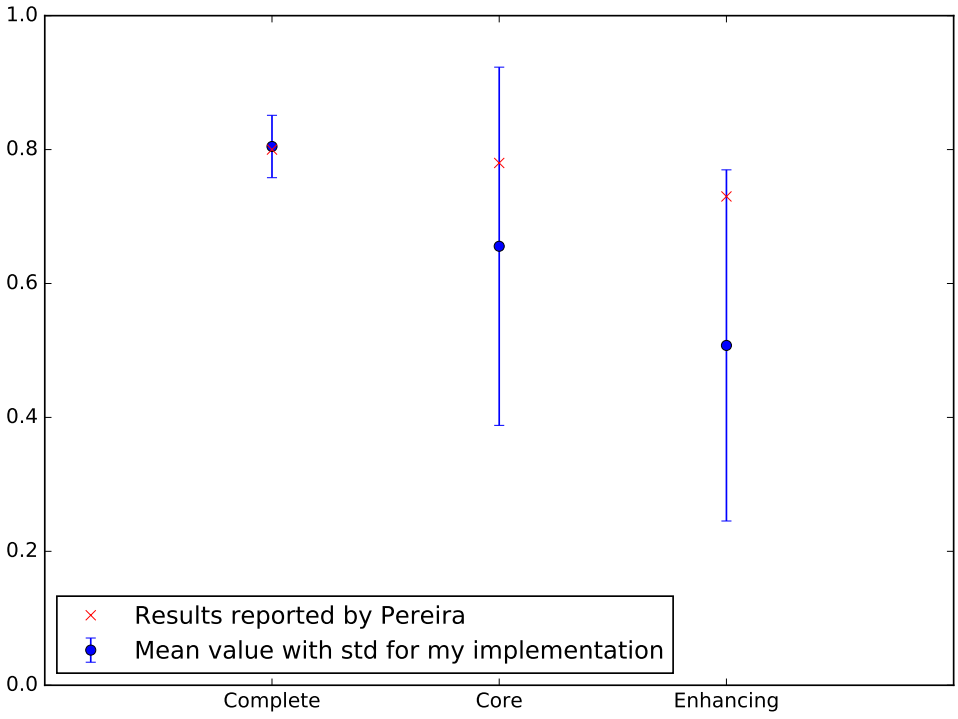

Figure 4.4: Box plot showing the dice scores obtained by my implementation of the model proposed by Pereira et al.

		Region		
		Complete	Core	Enhancing
Sensitivity	My implementation	0.816 ± 0.135	0.588 ± 0.304	0.429 ± 0.289
	Pereira	0.92	0.74	0.77
PPV	My implementation	0.820 ± 0.092	0.910 ± 0.074	0.861 ± 0.096
	Pereira	0.75	0.86	0.71

Table 4.2: Sensitivity and positive predictive value for the three regions obtained on the challenge dataset using my implementation of the method proposed by Pereira et al. The results reported by Pereira are also shown for comparison. My implementation achieves higher values for the positive predictive value but lower values for the sensitivity.

exactly followed the method described in the paper published by Pereira et al., I emailed the author asking for a possible explanation for this difference. He answered that (at the time) there were two differences between his and my method, which he did not report in his paper. The first difference was the parameters used in the N4 correction, which he did not include in his paper. However, changing the parameters I use to those used in his proposed model 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 used a heuristic to avoid selecting patches that were too close to each other (less than 3 voxels apart) and made sure to include patches of normal tissue close to the tumour. I also implemented these heuristics as described in section 3.1.2, which improved the results slightly but not to a level similar to those reported by the author. This discrepancy is hard to explain, but it is likely due to the fact that the heuristics I implemented are only approximations to those used (but not reported) by Pereira.

Sensitivity and positive predictive value

The means and standard deviations for the sensitivity and positive predictive value for my implementation of the model proposed by Pereira are reported in table 4.2. My implementation obtained higher values for the positive predictive value but lower scores for the sensitivity. This can be explained by the trade-off that exists between these two metrics, explained previously.

BraTS ranking

Finally, I also submitted my segmentations to the online evaluation platform which ranks the different contestants according to their scores. Figure 4.3.1 shows a screenshot of part of the ranking table that shows the entry for my submission. Note that my submission ranked 2^{nd} and 1^{st} 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

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				
41	dvorp1	0.83 (39)	0.70 (39)	0.61 (48)	0.86 (18)	0.80 (26)	0.82 (4)	0.81 (44)	0.66 (44)	0.56 (49)	0.99 (36)	33.67	36.33	33.67
42	borgs1	0.80 (43)	0.66 (46)	0.51 (53)	0.82 (32)	0.91 (2)	0.86 (1)	0.82 (42)	0.59 (50)	0.43 (53)	0.99 (43)	39.00	32.67	35.67
43	watti1	0.75 (47)	0.70 (38)	0.67 (38)	0.66 (52)	0.67 (47)	0.61 (49)	0.90 (18)	0.76 (22)	0.78 (12)	0.98 (49)	39.00	35.67	33.00

Figure 4.5: Online evaluation ranking. My submission was ranked 42nd.

		Predicted				
		0	1	2	3	4
Ground truth	0	**99.94**	0.001	0.052	0.002	0.001
	1	**42.287**	25.361	5.059	17.710	9.582
	2	**55.743**	1.508	38.762	3.065	0.921
	3	**51.163**	8.576	23.191	10.702	6.367
	4	25.067	0.891	3.926	6.107	**64.0078**

Table 4.3: Confusion matrix for the model proposed by Pereira for 11 scans taken from the BraTS 2015 dataset. The percentage of correctly predicted voxels for each class is shown.

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.

4.3.2 Confusion matrix

To get more insights into how the model behaves, I decided to segment 11 scans from the 2015 dataset, normalised identically. The dice scores obtained for these 11 scans are 0.681, 0.643 and 0.673 for the 'Complete', 'Core' and 'Enhancing' regions respectively. The ground truth for these scans is put together slightly differently as for the 2013 dataset, which explains why the dice scores is lower for the 'Complete' region. However, the scans are still similar enough for these scores and the confusion matrix to be relevant. The confusion matrix is shown in table 4.3. As expected from the dice scores, the model is doing well for classes 0 and 4, predicting over 64% of the voxels in class 4 correctly. For classes 1, 2 and 3 the model is doing rather poorly, reaching no more than 38% of correctly classified voxels and predicting a majority of voxels as being from class 0. The especially low result for class 3, with only about 11% of correctly classified voxels can be explained by the fact that there is less training data available for this class. Furthermore, the boundaries between class 2, 3 and 4 are somewhat subjective, meaning that different interpretations of the tissues possible are.

Variant	Dice score		
	Complete	Core	Enhancing
Adam + ReLU	0.796 ± 0.054	0.630 ± 0.279	0.486 ± 0.265
Adam optimiser	0.815 ± 0.069	0.670 ± 0.233	0.592 ± 0.217

Table 4.4: Dice scores obtained using variants of the model proposed by Pereira et al.

4.3.3 Effect of various components

Finally, I investigate what impact different components and design choices for the convolutional neural networks have on the final dice score on the challenge dataset. I investigate the following design choices:

1. Using the linear rectifier (ReLU) activation function 2.3 instead of the Leaky linear rectifier.
2. Using the Adaptive Moment Estimation method (adam) [11] to update the learning rates automatically, instead of linearly decaying the learning rate.

In table 4.4 the results obtained using these variants are shown. The variant using the adam optimisation performs better than my implementation of Pereira's model which uses linearly decaying learning rates. This suggests that my implementation of the model proposed by Pereira might not have fully converged after 20 epochs and would need further training and learning rate adaptation to reach the same convergence as the model using the adam optimisation.

4.4 Evaluation of my model

4.4.1 BraTS evaluation

Table 4.5 shows the results I obtained using my fully-convolutional model on the 10 images included in the challenge dataset. The results for patient number 7 are again considerably lower than those obtained on the other patients in the 'Core' and 'Enhancing' regions. This is hard to explain, as Pereira reported no such discrepancy. A possible explanation, is that the images for patient 7 are slightly rotated in the x-y plane, as shown in figure 4.4.1.

4.4.2 Confusion matrix

The confusion matrix obtained for my model using 11 of the BraTS2015 patients is shown in table 4.7. Interstingly, although my model performs worse on the 2013 challenge dataset

Patient	**Dice score**		
	Complete	Core	Enhancing
1	0.856	0.796	0.665
2	0.843	0.622	0.670
3	0.744	0.297	0.368
4	0.379	0.156	0.150
5	0.812	0.689	0.665
6	0.772	0.381	0.493
7	0.593	0.015	0.015
8	0.877	0.705	0.675
9	0.725	0.857	0.781
10	0.898	0.905	0.838
mean	0.750 ± 0.158	0.542 ± 0.310	0.532 ± 0.273

Table 4.5: Dice scores obtained for the 10 challenge patients using my model.

Region	**Mean sensitivity**	**Mean positive predictive value**
Complete	0.661 ± 0.209	0.933 ± 0.042
Core	0.453 ± 0.318	0.945 ± 0.038
Enhancing	0.467 ± 0.297	0.840 ± 0.101

Table 4.6: Sensitivity and positive predictive value for the three regions obtained on challenge dataset using my model.

than the model proposed by Pereira et al., it performs much better on this subset of the 2015 training dataset, obtaining average dice scores of (0.806,0.776,0.819). The lowest value is obtained for class 3 for the same reason. However, the value has more than tripled from 11% to 38% of voxels from class 3 being correctly classified.

		Predicted				
		0	1	2	3	4
Ground truth	0	**99.78**	0.02	0.16	0.02	0.02
	1	0.74	**42.66**	20.58	26.91	9.11
	2	28.79	4.13	**57.97**	7.54	1.57
	3	5.87	10.49	30.81	**37.98**	14.84
	4	4.44	1.62	4.77	8.52	**80.64**

Table 4.7: Confusion matrix obtained with my model for 11 scans taken from the BraTS 2015 dataset. The percentage of correctly predicted voxels for each class is shown.

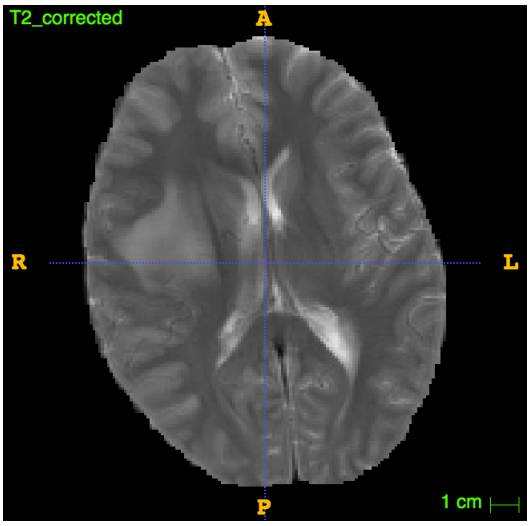

Figure 4.6: Slice taken from the T2 scan for patient 7 of the challenge dataset. The image is slightly rotated or skewed in the x-y plane, which is a possible explanation for the lower results in the 'Core' and 'Enhancing' regions for that patient.

4.5 Comparison

4.5.1 Performance

Table 4.8 shows the results I obtained using my implementations along to those reported in Pereira's paper on the BraTS2013 Challenge dataset. My implementations of the model proposed by Pereira et al. has the highest dice scores for the 'Complete' region. However, the Dice scores for the 'Core' and the 'Enhancing' regions are lower than those reported by Pereira, as explained previously. My fully-convolutional architecture has lower Dice scores but obtains the highest positive predictive value on all 3 regions. This suggest that this model could further be improved by attempting to trade-off some positive predictive value for sensitivity to obtain higher dice scores.

Method	Dice			PPV			Sensitivity		
	Complete	Core	Enhancing	Complete	Core	Enhancing	Complete	Core	Enhancing
My FCN model	0.75	0.54	0.53	**0.93**	**0.95**	**0.84**	0.66	0.45	0.47
My Pereira impl.	0.81	0.66	0.51	0.82	0.91	0.86	0.82	0.59	0.43
Variant with Adam	**0.82**	0.67	0.53	0.89	0.87	0.80	0.78	0.61	0.54
Reported by Pereira	0.80	**0.78**	**0.73**	0.75	0.86	0.71	**0.92**	**0.74**	**0.77**

Table 4.8: Comparison of model performances on the Brats2013 Challenge dataset.

The average Dice score over the 11 evaluated scans of the 2015 dataset is shown in table 4.9. The fully-convolutional model performed much better than my implementation of the model proposed by Pereira. This can be explained by the fact that the 2015 dataset is normalised differently and that the fully-convolutional model has a much larger receptive field.

Method	Dice		
	Complete	Core	Enhancing
My FCN model	0.806	0.776	0.819
My Pereira impl.	0.681	0.643	0.673

Table 4.9: Dice scores obtained on the 2015 dataset for my FCN model and my implementation of the model proposed by Pereira.

4.5.2 Segmentation speed

Finally, I compared how fast the two implementations can segment a scan of $155 \times 240 \times 240$ voxels by taking the average time needed to segment 11 scans of the 2015 dataset. As expected, the fully-convolutional model is much faster, achieving a speedup of a factor of 7.25 , as shown in table 4.10.

Method	Time to segment a scan (in seconds)
My FCN model	110.47
My Pereira impl.	802.82

Table 4.10: Comparison of the time required for segmenting a scan of $155 \times 240 \times 240$ voxels for my implementation of the model proposed by Pereira and my fully-convolutional model.

Chapter 5

Conclusion

5.1 Summary of achievements

This project surveyed the mathematical foundations of deep learning with convolutional neural networks and how these models can be applied to the problem of brain tumour segmentation. I implemented two different convolutional neural networks. First, I replicated the model proposed by Pereira et al. [17]. This gave me an introduction to this field with some certainty that the model would work. I then designed a second model using techniques more recently developed for the design of convolutional neural networks. The model I built is performing similarly, but has the significant advantage of being able to segment a scan up to 7 times faster. This speed-up is achieved by removing all fully-connected layers from the network. This also has the effect of reducing the number of parameters, which makes my convolutional neural networks able to consider an input patch with 4 times more voxels while keeping the number of parameters similar. Both models were trained on the data made available through the BraTS2013 challenge.

I evaluated both models using the standard metrics for this segmentation task: Dice score, positive predictive value and sensitivity. I also compared the performance of my models to the performance of models published by researchers in the field using the BraTS challenge leaderboard. Using the data from the BraTS 2015 dataset, I was further able to get more insight in how well the different models perform by examining the confusion matrix for this dataset.

Both models approximated the performance obtained by recent state-of-the-art methods for the 'Complete' region, but not for the 'Core' or 'Enhancing' regions. Especially noticeable is the difference in the Dice score for the 'Enhancing' tumour region between my implementation of the model proposed by Pereira et al. and their published results. After emailing the author, I was able to confirm that the only difference in the two implementations was the heuristic for selecting which patches to train the network on, which was not specified in the published paper by Pereira. Hence, if I had to do it again, I would

probably choose to replicate a more detailed paper or a paper for which the source code is available. This difference in results also shows the importance of the quality of the input data for deep learning methods, which often is more important than the architectural details of the network. Interestingly, increasing the receptive field of the neural network to a larger patch surrounding the voxels to be classified did not significantly increase the performance of the network. This indicates that the class of a voxel is highly dependent on the surrounding voxels, but not on those voxels slightly further apart.

5.2 Future Work

A likely future direction is to use three-dimensional patches instead of being restricted to two-dimensional patches. Incorporating the third dimension is challenging in convolutional neural networks due to the exponential growth in parameters associated with it. However, some very recent approaches have overcome these difficulties and use three-dimensional patches to achieve state-of-the-art results [**?**].

A further likely future direction concerns the fact that convolutional neural networks are not able to give any measure of uncertainty for their predictions, which might be very important for life-or-death cases such as brain tumours. Some recent work by Gal and Ghahramani showed how it is possible to get such uncertainty bounds in convolutional neural networks using Dropout [3]. It would be interesting to research how this new knowledge could be incorporated into models for brain tumour segmentation.

Bibliography

[1] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166, Mar 1994.

[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] Yarin Gal and Zoubin Ghahramani. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In *International Conference on Machine Learning*, pages 1050–1059, 2016.

[4] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Aistats*, volume 9, pages 249–256, 2010.

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

[6] 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.

[7] Mohammad Havaei, Axel Davy, David Warde-Farley, Antoine Biard, Aaron Courville, Yoshua Bengio, Chris Pal, Pierre-Marc Jodoin, and Hugo Larochelle. Brain tumor segmentation with deep neural networks. *Medical image analysis*, 35:18–31, 2017.

[8] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

[9] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[10] Konstantinos Kamnitsas, Christian Ledig, Virginia FJ 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] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[12] 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.

[13] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[14] 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.

[15] L. G. Nyul, J. K. Udupa, and Xuan Zhang. New variants of a method of mri scale standardization. *IEEE Transactions on Medical Imaging*, 19(2):143–150, Feb 2000.

[16] Hiroko Ohgaki and Paul Kleihues. Population-based studies on incidence, survival rates, and genetic alterations in astrocytic and oligodendroglial gliomas. *Journal of Neuropathology & Experimental Neurology*, 64(6):479, 2005.

[17] Sérgio Pereira, Adriano Pinto, Victor Alves, and Carlos A Silva. Brain tumor segmentation using convolutional neural networks in mri images. *IEEE transactions on medical imaging*, 35(5):1240–1251, 2016.

[18] 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.

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

[20] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[21] 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.

[22] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ICML'13, pages III–1139–III–1147. JMLR.org, 2013.

[23] 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.

[24] 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.

[25] Darko Zikic, Yani Ioannou, Matthew Brown, and Antonio Criminisi. Segmentation of brain tumor tissues with convolutional neural networks. *Proceedings MICCAI-BRATS*, pages 36–39, 2014.

Appendix A

Project Proposal