Sebastian Borgeaud dit Avocat

Brain tumour segmentation using Convolutional Neural Networks

Computer Science Tripos – Part II

Fitzwilliam College

March 26, 2017

Proforma

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

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

Declaration

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

Signed [signature]

Date [date]

Contents

1 Introduction [14%] 9
- 1.1 Motivation . . . 9
- 1.2 Related Work . . . 9
- 1.3 Supervised learning and Classification . . . 9

2 Preparation [26%] 11
- 2.1 Starting point . . . 11
- 2.2 Theoretical background . . . 12
  - 2.2.1 Artificial neural networks . . . 12
  - 2.2.2 Convolutional neural networks . . . 15
  - 2.2.3 The overfitting problem . . . 17
- 2.3 Data source . . . 18
  - 2.3.1 BraTS Challenge . . . 18
  - 2.3.2 Data used in this project . . . 20

3 Implementation [40%] 23
- 3.1 Data pre-processing . . . 23
  - 3.1.1 Patch extraction . . . 23
  - 3.1.2 Data augmentation . . . 26
  - 3.1.3 Normalisations . . . 26
- 3.2 Pereira model . . . 27
  - 3.2.1 Architecture . . . 27
  - 3.2.2 Implementation of the architecture in Keras . . . 29
  - 3.2.3 Training . . . 30
- 3.3 My model . . . 31
  - 3.3.1 Architecture . . . 31
  - 3.3.2 Implementation . . . 31
  - 3.3.3 Training . . . 31
- 3.4 Segmentation . . . 31
  - 3.4.1 Pereira model . . . 31
  - 3.4.2 My model . . . 31
- 3.5 Data post-processing . . . 31

4 Evaluation (+ Conclusion [20%]) 33
4.1 Metrics used for the evaluation . . . 33
4.1.1 Regions of evaluation . . . 33
4.1.2 Dice score . . . 33
4.1.3 Positive predictive value . . . 33
4.1.4 Sensitivity . . . 33
4.2 Evaluation of the model proposed by Pereira et al. . . . 33
4.3 Evaluation of my model . . . 33
4.4 Comparison . . . 33

5 Conclusion 35
5.1 Summary of achievements . . . 35
5.2 Future Work . . . 35

Bibliography 35

List of Figures

2.1 Structure of a simple neural network with two hidden layers. 12
2.2 Plot of the activation functions. 14
2.3 Example computation done by a 3x3 filter in a convolutional layer 17
2.4 Example slices in the axial plane for the 4 different scan modalities 19
2.5 Slices of a T1 MRI scan annotated with the expert labelling. 20

3.1 Convolutional network architecture proposed by Pereira et al. 23
3.2 Convolutional network architecture proposed by Pereire et al. 28
3.3 Part of the output of the training script 30

Acknowledgements

Chapter 1

Introduction [14%]

1.1 Motivation

- First the problem should be defined
- Motivation for choosing this problem: Openly available data, important problem (include count of people affected by brain tumours)
- Then motivate the choice of conv nets to tackle this problem.
- Also mention that this is an opportunity for me to explore the field of ML further and gain some practical experience working in that field.

1.2 Related Work

- Here I will introduce the previous work done. In particular, this should contain a short intro to the history of neural nets and conv nets. Then a brief history of the brain tumour segmentation problem.
- Next mention the main paper [2] and the BraTS challenge/conference.
- Mention more recent developments, such as the usage of ResNets.
- This paragraph should introduce the reader to the problem and to what has been done previously.

1.3 Supervised learning and Classification

1. A very brief intro to supervised learning is given in terms of data pairs $\mathbb{X}, \mathbb{Y}$ consisting of examples and labels.

2. Then introduce classification

3. Finally show how this segmentation problem can be reduced to a classification problem.

Chapter 2

Preparation [26%]

During the first phase of my project, the aim was to replicate the method used by Pereira et al [2], so it was crucial to first fully understand the steps taken in the paper to then be able to reimplement them. Unfortunately, the paper didn't include any source code which meant that if something wasn't fully explained in details, I would have to find out what was actually done. This turned out to be a problem for the pre-processing step as the proposed method uses a normalisation developed by Nyul [CITATION]. This normalisation requires human input, preferably from a domain expert, and I was therefore not able to use that normalisation method. The paper also proposed a second normalisation method, which used a combination of winsorizing and N4 normalization. This method performed slightly worse but had the advantage of being fully automated, which is why I chose to use this method.

2.1 Starting point

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

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

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

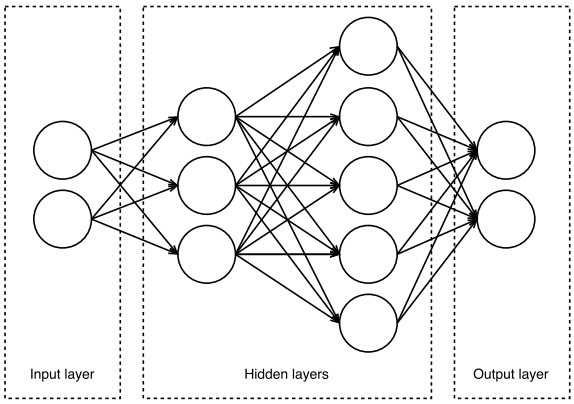

Figure 2.1: Structure of a simple neural network with two hidden layers.

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

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

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

2.2 Theoretical background

2.2.1 Artificial neural networks

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

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

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

an output

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

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

Activation functions

The most common activation functions are the Sigmoid function,

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

the hyperbolic tangent

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

the rectifier

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

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

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

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

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

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

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

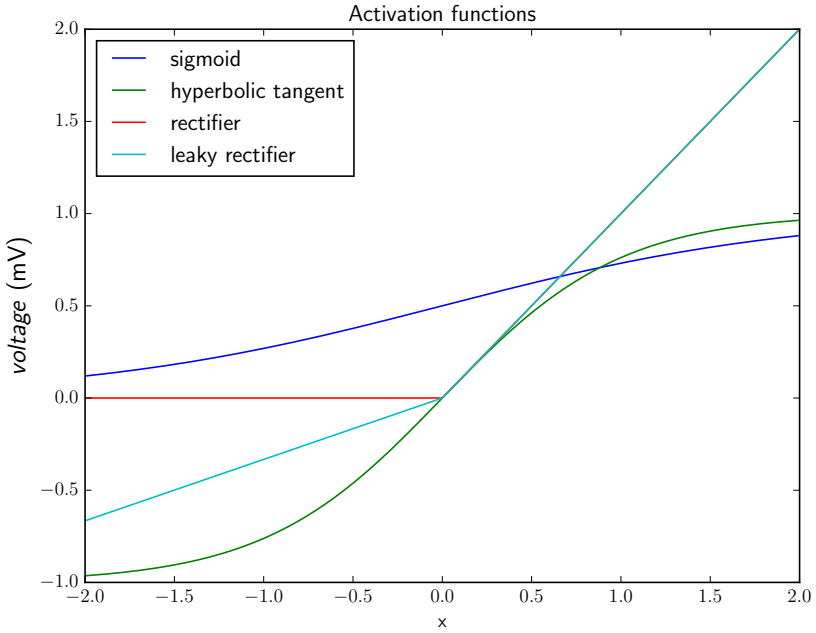

Figure 2.2: Plot of the activation functions.

Loss function

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

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

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

Optimisation

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

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

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

Since every basic operation used in the neural network is differentiable, the entire network will also be differentiable, which in turn makes it possible to calculate the gradient

of a loss function with respect to the weights in the network. This process is called backpropagation.

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

Momentum

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

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

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

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

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

Further optimizations

Should I explain adam optimisation??

2.2.2 Convolutional neural networks

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

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

Fully-connected layers

These are exactly the same as for ordinary neural networks.

Convolutional layers

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

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

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

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

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

Pooling layers

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

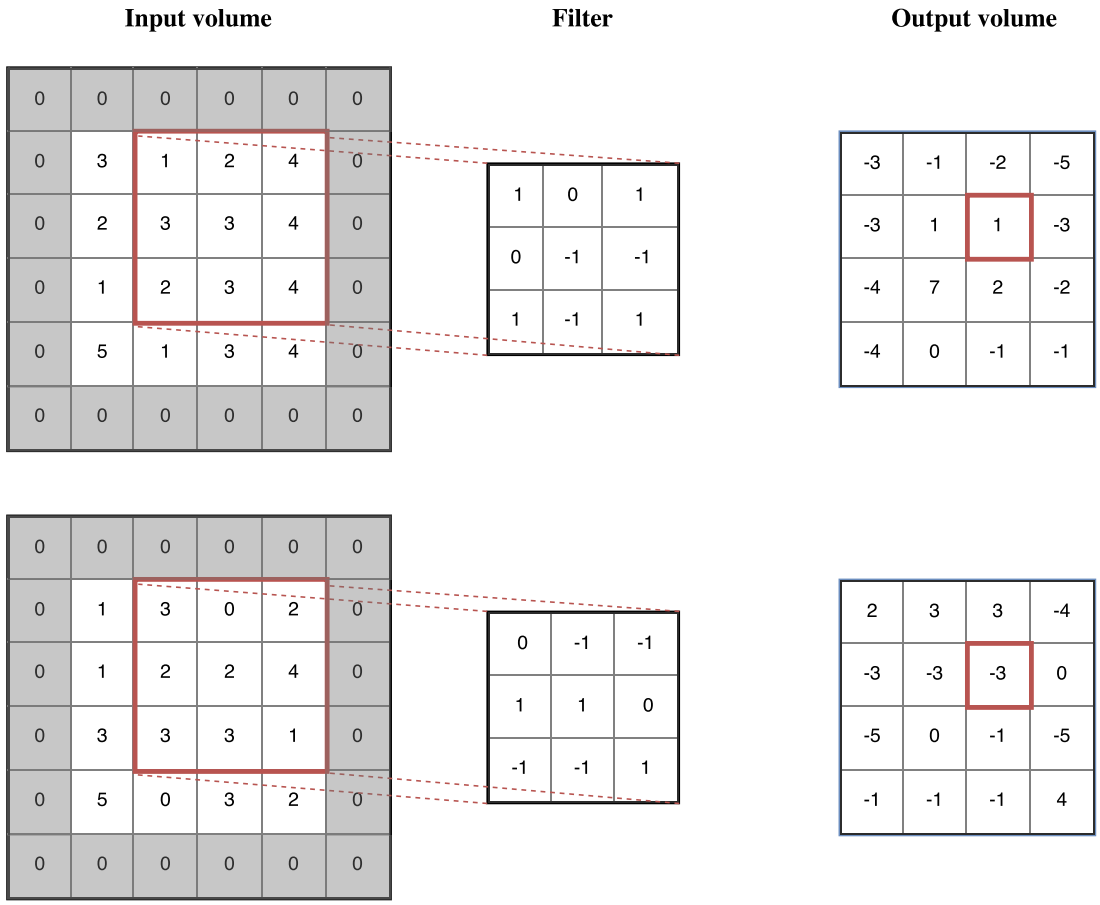

Figure 2.3: Example computation done by a 3x3 filter in a convolutional layer. Both the stride and the padding are 1, so as to keep the output dimensions identical. The output at depth 0 for the marked element is $1 \cdot 1 + 2 \cdot 0 + 4 \cdot 1 + 3 \cdot 0 + 3 \cdot -1 + 4 \cdot -1 + 2 \cdot 1 + 3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

determines the step size.

A convolutional neural network typically consists of a sequence of these layers, starting with pairs of convolutional neural networks and max pooling layers. The idea behind this is that each pair of layers can learn more and more abstract features using the features learned by the previous layer. For example, the first pair might learn to recognise edges, the second layer shapes, etc.. The last few layers of the network consist entirely of fully-connected layers. These learn how to classify the data using the features learned by the convolutional and max pooling layers.

2.2.3 The overfitting problem

A common problem to arise in neural networks is that of **overfitting**, that is when the model isn't able to generalise on previously unseen data and instead just memorises the

training data. Due to the large number of weights in convolutional neural networks, these are particularly prone to this problem. Many techniques and heuristics have been developed in order to help reduce overfitting. In particular, I used three of them: **L2 weight normalisation**, **Dropout** and **Batch Normalisation**

L2 weight normalisation

The intuition behind this normalisation is that by preventing individual parameters to grow without bounds, the model must learn how to smoothly extract features of the data and this in turn will prevent the model from just memorising the data. This is achieved by adding a regularisation penalty to the loss function. The L2 normalisation computes the sum of the squares of every parameter:

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

Hence, the new loss function is

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

where $\mathcal{L}_{data}(\theta)$ is the data loss, defined in equation 2.2. Although I have only used the L2 normalisation (explain why?), it is important to realise that different regularisation penalties could be used, which lead to different regularisations.

Dropout

TODO

Batch Normalisation

TODO

2.3 Data source

2.3.1 BraTS Challenge

The dataset I am using comes entirely from the BraTS [1] challenge. It is split into three sections:

1. The training dataset, which consists of 30 different patients and their ground truth marked by a human experts. The 30 patients are further divided into 20 high-grade

glioma cases (HG) and 10 low-grade glioma cases (LG). The difference between these two types of brain tumours is their rate of growth, which is slower for the low-grade case.

2. The challenge set, which consists of 10 high-grade patients without the ground truth. These scans are meant to be segmented by the participants of the challenge, who can then submit their segmentation online.

3. The leaderboard set contains 25 patients, including both high-grade and low-grade gliomas. The ground truth labels are not available.

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

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

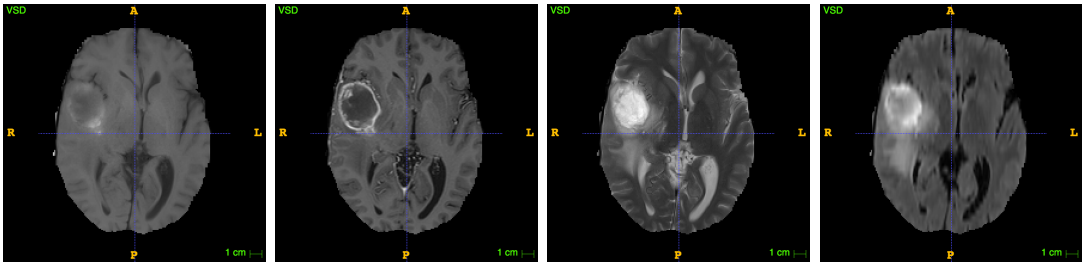

Figure 2.4: Example slices in the axial plane for the 4 different scan modalities. From left to right the modalities are T1, T1c, T2 and Flair. The scan is from patient 1, slice $z = 89$. The four modalities have some obvious differences which the convolutional neural network might be able to utilise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

slices of a T1 MRI scan annotated with the ground truth labelling created by a group of 4 experts.

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

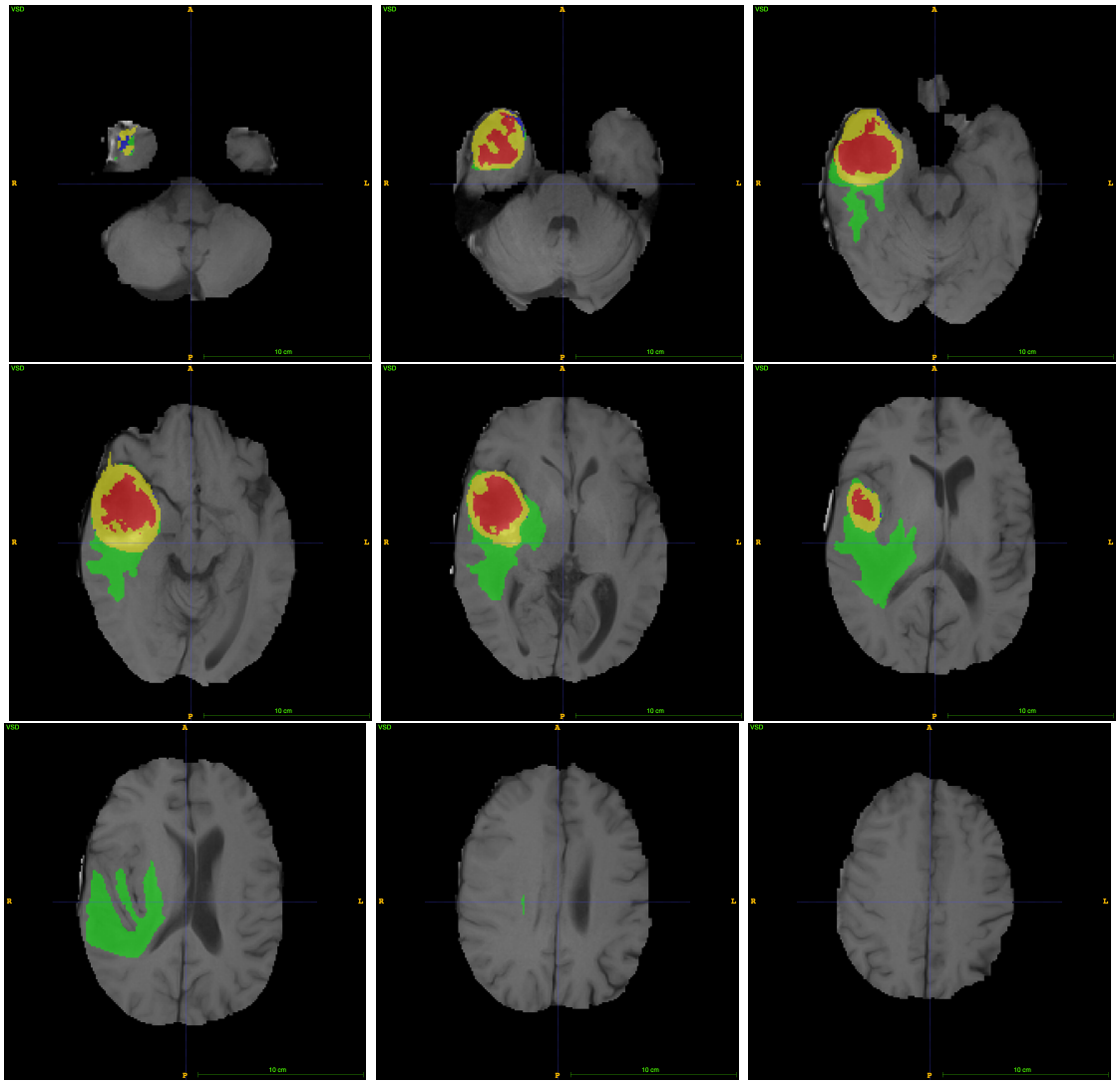

Figure 2.5: Axial plane slices of a T1 MRI scan annotated by the expert labelling. The slices were taken from patient 1, using $z \in \{49, 59, 69, 79, 89, 99, 109, 119, 129\}$. The enhancing tumour region is in yellow, the non-enhancing tumour in blue, the edema in green and the necrotic core in red,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

2.3.2 Data used in this project

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

To provide some guidance on how well the network is performing during the training phase, I also used 10 high-grade glioma cases from the BraTS2015 dataset, which contains

more patients prepared in a similar way. These 10 patients were used to provide the validation data.

Chapter 3

Implementation [40%]

3.1 Data pre-processing

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

3.1.1 Patch extraction

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

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

Figure 3.1: Convolutional network architecture proposed by Pereira et al.

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 $(z \times y \times z)$

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

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

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

A further question is how we should deal with a voxel that is to close to the edge of the scan to be able to extract an entire patch around it. I could either ignore the patch or pad the scan such that each labelled voxel is surrounded by enough voxels. The first option is not only easier, but also helps with the balanced classes issue as most of the voxels close to an edge are from class 0. This is because the brains are centered within the scans. Table 3.1.1 shows the distribution of classes for "valid" voxels, .i.e those at least more than half the patch length away from the x or y edges.

The patch extraction can now be performed. First I created a list for each class containing the positions of the valid voxels for that class. Note that at this point it would be unfeasible to store the patches, as it would require to store over a 100 billion 32 bit floats for class 0 alone, or over 400 GB. To get the indices of the valid voxels, I used numpy's argwhere function on the ground truth arrays. The argwhere function

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

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

takes in an array and a predicate and returns the indices of those elements in the array satisfying the predicate. Here, the predicate used is just equality checking between the voxel label and the class number. This is done for every patient. Since the scan number must also be included in the indices, I prepended the scan number along the first axis, before aggregating the results for each patient into a list. The final result is a list for each class that contains tuples of the form (patient number, z, y, x). The second step consists of removing those voxels that are too close to an x-axis or y-axis edge. Only indices (patient, z, y, x) where the x and y values are within the allowed ranges of $[16, \text{size}-(16+1)]$ are kept. Again this can be done using numpy and filtering based on column values. Algorithm 1 summarises the steps taken during the patch extraction.

Algorithm 1 Patch extraction

for class k in $[0, 1, 2, 3, 4]$ **do**
    valid_indices[k] $\leftarrow$ []
    for each index, label **in** labels **do**
        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] $- 16 \geq 0$]
        possible_indices $\leftarrow$ possible_indices[possible_indices[:, 2] $+ (16 + 1) <$ label.y_dimension]
        possible_indices $\leftarrow$ possible_indices[possible_indices[:, 3] $- 16 \geq 0$]
        possible_indices $\leftarrow$ possible_indices[possible_indices[:, 3] $+ (16 + 1) <$ label.x_dimension]

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

3.1.2 Data augmentation

To increase the amount of data available, some data augmentation techniques can be used. In typical applications of convolutional neural networks for image processing and computer vision tasks, translation and rotations can be used. Since the data consists entirely of two-dimensional patches, translation is useless as it would just result in a different patch, with a possibly different label. However, using rotations of the patches might give some performance improvements. Pereira et al. proposes different techniques in [2]:

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

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

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

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

3.1.3 Normalisations

This section should explain what the next 3 normalisations do and how they are implemented, i.e. using python's numpy.

Winsorising

The first normalisation that is applied is called 'winsorising'. The aim is to limit the values of extremes, in order to reduce the effect that outliers may have. This is also known as 'clipping' in digital signal processing. I used a 98% winsorisation, meaning that the data values below the 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 modified.

N4ITK

This section might be slightly longer. It should definitely include a picture of a comparison of some slices before and after the normalisation. I am not sure if I should explain what the normalisation actually does as it is quite complicated.

Linear scaling

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

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

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

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

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

Mean and Variance standardisation

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

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

As $E[aX] = aE[X]$ and $Var[cX] = c^2Var[X]$ it is easy to see that after this transformation the 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 are used.

3.2 Pereira model

3.2.1 Architecture

The model proposed by Pereira et al. [2] consists of 11 layers, shown in figure 3.2. The

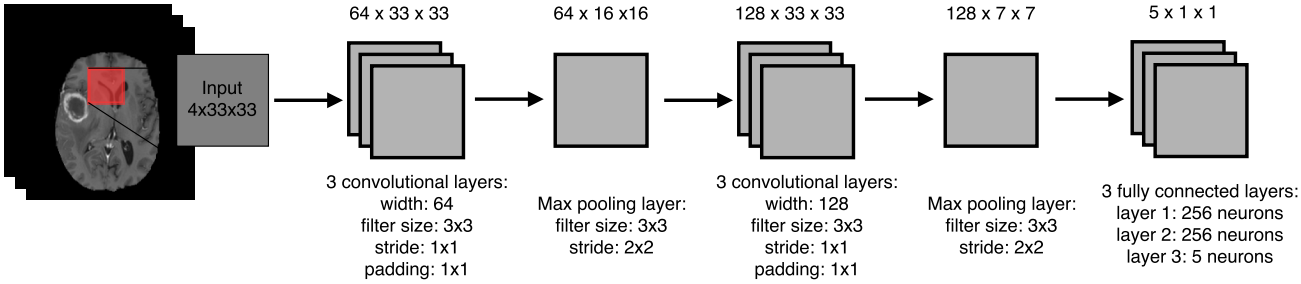

Figure 3.2: Convolutional network architecture proposed by Pereire et al.

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

Table 3.4: 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.

first three layers are convolutional layers with filter size 3x3, stride 1x1 and width 64. Three layers are used consecutively because the combination of them effectively has a receptive field of size 7x7 while having fewer parameters then a single 7x7 convolutional layer would have, making the network less prone to overfitting. [CITATION 36 in pereira]. The next layer is a max pooling layer of size 3x3 and stride 2x2. This is an unusual design as the size is larger than the stride. This is done because although pooling can be positive to eliminate unwanted details and achieve invariance, it can also eliminate some of the important details. By keeping the size of the layer larger than the stride, the pooling will overlap which will allow the network to keep more information about location. Next, three convolutional layers are applied, this time doubling the width to 128 filters. Again a max pooling layer with the same parameters as the previous one is applied, before adding three fully connected layers of size 256, 256 and 5 respectively. Table 3.2.1 shows the number of weights in each layer.

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

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

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

3.2.2 Implementation of the architecture in Keras

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

```
model = Sequential()
```

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

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

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

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

Adding Dropout is again done similarly:

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

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

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

3.2.3 Training

The paper specifies that the model was trained over 20 epochs, passing each patch through the network once per epoch. The optimisation algorithm used is a standard stochastic gradient descent algorithm with Nesterov momentum ($\mu = 0.9$). The learning rate should be linearly decreased from 3×10^{-5} to 3×10^{-7}. Keras doesn't implement this functionality, so it has to be done manually. This can be done using algoritm 2

Algorithm 2 Training the model with linearly decaying learning rate

1: **for** i from 0 to nb_epochs $-$ 1 **do**
2: learning_rate $\leftarrow$ start_rate $+ i \times \dfrac{\text{end\_rate} - \text{start\_rate}}{\text{nb\_epochs} - 1}$
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 dat and the accuracy (equation 3.4) is computed. The loss and accuracy are then reported to the standard output. Figure 3.3 shows part the output of training a network for a few epochs.

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

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

Figure 3.3: Part of the of the training script. The output is shown for epochs 8 and 9. Both the validation loss and the validation accuracy are reported, as well as how long it took to train and validate for this epoch.

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

TODO: ADD GRAPH OF TRAINING/VALIDATION ACCURACY

3.3 My model

3.3.1 Architecture

3.3.2 Implementation

3.3.3 Training

3.4 Segmentation

3.4.1 Pereira model

1. Explain how a scan is segemented
   (a) Normalising identically to the training data
   (b) Adding 0 padding to the image to be able to classify all voxels
   (c) 3 Convolve the model on the new bigger image.

3.4.2 My model

1. Similarly here, but show how it is more complicated as the model now predicts a single block of 64x1 voxels, which is really an 8x8 block and say that we want to use as large batch sizes as possible to accelerate the process.

3.5 Data post-processing

1. Explain how a connected components analysis is made to remove the small regions wrongly classified as a tumour region. Show an example of before and after.

Chapter 4

Evaluation (+ Conclusion [20%])

4.1 Metrics used for the evaluation

4.1.1 Regions of evaluation

4.1.2 Dice score

4.1.3 Positive predictive value

4.1.4 Sensitivity

4.2 Evaluation of the model proposed by Pereira et al.

4.3 Evaluation of my model

4.4 Comparison

Chapter 5

Conclusion

5.1 Summary of achievements

5.2 Future Work

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

Bibliography

[1] 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.

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