MANCHESTER 1824
The University of Manchester

Evaluation of convolutional neural networks for the segmentation of stroke lesions

Third Year Individual Project – Final Report

April 2021

Faruk Fakih Safadi

10358028

Supervisor: Dr. Fumie Costen

School of Electrical and Electronic Engineering

Abstract

A stroke is a serious medical condition that affects the life of millions of people every year. The detection and segmentation of a stroke is a time-expensive process that requires an accurate assessment of brain tissue by the hands of an expert. For this reason, during the last decade, several automated stroke segmentation methods have been developed. The majority of these methods use machine learning and deep learning classifiers, as well as data pre-processing pipelines. This report aims to evaluate the feasibility of using a data pre-processing pipeline to train a convolutional neural network for the segmentation of stroke lesions. To accomplish this aim, two convolutional neural networks were trained. The only difference between these is that the dataset used to train one of the networks passed through a pre-processing pipeline whilst the other did not. The results obtained after evaluating both networks showed that for the scenario presented on this report, the convolutional neural network without a pre-processing pipeline had a better performance. However, due to specific problems on the pre-processing pipeline, it was concluded that further studies are required in order to arrive to a general conclusion on the feasibility of a pre-processing pipeline.

Contents

Total word count: 9777

List of figures

List of tables

1. Introduction

A stroke is a serious medical condition that occurs when blood flow to the brain is blocked, causing the death of brain cells [1]. There are two main types of stroke: ischemic, due to blockage of a vessel, and haemorrhagic, due to bleeding [1]. Both cause parts of the brain to stop functioning properly [1]. Around 87% of strokes are ischemic [2].

The traditional approach to identify/segment stroke-lesions is carried out by a radiologist and/or other specialists through the analysis of a magnetic resonance imaging (MRI) scan, or a computerized tomography (CT) scan of the brain. This can be considered as the 'gold standard' for stroke lesion segmentation. However, the drawback of this approach is that the procedure can take a considerable amount of time depending on factors such as the availability of the specialist and the stroke itself (e.g., size and location of the stroke). MRI scans can be classified into two types: T1 and T2 images [3]. The timing of radiofrequency pulse sequences used to make T1 images results in images which highlight fat tissue within the body [3]. On the other hand, the timing of radiofrequency pulse sequences used to make T2 images results in images which highlight fat and water within the body [3].

Over the past few years, technology has grown exponentially, and applications that were thought as impossible during the last decade such as self-driving cars, are now common in our daily life. Similarly, the same can be said about computer vision. Thanks to the quantity of data available nowadays, and the advances in machine learning and deep learning, it is now possible to build computer vision systems that can mimic or surpass human-level performance. As a result, many academics have conducted researches on how to develop state-of-the-art (SOTA) computer vision systems using deep learning approaches. These systems have been proved to be highly effective and are the most attractive for new researchers on the field of computer vision.

1.1. Background

During the last decade, different machine learning approaches have been developed in order to tackle the task of stroke segmentation [4]. For the development of such systems, researchers have implemented different data pre-processing pipelines that have been proved effective [5]. On the other hand, during recent years, the use of deep learning for medical image segmentation has increased drastically, more specifically, the use of Convolutional Neural Networks (CNNs) [6]. The main highlight of using CNNs is that given a dataset containing different images, a CNN can

automatically extract features that are then used for the segmentation, whereas for machine learning approaches, these complex features would have been needed to be crafted manually. CNN's can be used to perform an end-end approach (i.e. perform the segmentation without any data pre-processing steps at all). However, one of the drawbacks of CNNs is that they need a large amount of data to produce useful results.

1.2. Motivation

In 2015, stroke was the second most frequent cause of death after coronary artery disease, accounting to 6.3 million deaths (11% of the total) [7]. Furthermore, about half of the people who have had a stroke live less than a year [7]. In addition, disability affects 75% of stroke survivors, enough to decrease their ability to work [8]. Besides, while CNNs possess the ability of performing an end-to-end segmentation, researchers have not conducted in-depth studies regarding the benefits of using CNNs with a data pre-processing pipeline for the task of stroke segmentation.

Thus, the motivation for this project comes from the fact that if detected on its earliest phase (within three or four hours), the stroke may be treatable with a medication that can break down the clot [2]. Likewise, it is essential for a patient's recovery to constantly evaluate the evolution of the stroke during time. Unfortunately, the manual segmentation of stroke lesions is a labour and time intensive task. For this reason, it would be extremely beneficial to have access to a system that can accurately segment stroke lesions in a considerable amount of time without the necessity of an expert. Furthermore, such a system is likely to perform better with the implementation of an adequate pre-processing pipeline.

1.3. Aim and objectives

This report will aim to evaluate the feasibility of using a data pre-processing pipeline to train a convolutional neural network for the segmentation of stroke lesions. In order to achieve the aim of this project, several objectives were established:

- Collection of manually-segmented stroke lesions.
- Conduct research regarding the segmentation of medical images of brains with strokes.
- Evaluate different machine learning and deep learning approaches used for the segmentation of stroke lesions.
- Identify and evaluate different pre-processing steps performed by researchers on the topic of stroke segmentation.

- Develop and compare two convolutional neural networks, one without data pre-processing and another with data pre-processing.

1.4. Dataset

As mentioned, one of the objectives is the collection of manually-segmented stroke lesions. After thorough research and evaluation, it was decided that the dataset to be used in this report is the Anatomical Tracings of Lesions After Stroke (ATLAS) [9]. The ATLAS dataset is provided by the University of South California Stevens Neuroimaging and Informatics Institute. The ATLAS dataset contains 229 T1-weighted MRI scans with manually segmented lesions. All of the scans correspond to different individuals. In order to access the dataset, an online form must be filled and reviewed by the institute. For this reason, the ATLAS dataset would not be provided on this report.

2. Literature review

The purpose of this section is to introduce the background knowledge needed to understand neural networks, and to analyse and compare the different approaches performed by researchers on the topic of stroke segmentation. For more detailed information, refer to the individual papers.

2.1. Background knowledge

2.1.1. Artificial Neural Networks

"Artificial neural networks or simply referred as neural networks, are powerful connectionist systems vaguely inspired by biological neural networks." [10]. Neural networks are a collection of connected nodes called neurons. Neurons receive one or more signals, and their output is computed using a non-linear function applied to the sum of all the input signals. A neuron can be connected to other neurons, these connections are called edges. A neural network can be formed by a neuron that takes a single input and produces an output, or it can be formed by a collection of neurons that interact with each other. Neurons are organized into layers. All neural networks are comprised by an input layer, an output layer, and one or more intermediate layers, also known as hidden layers. In other words, it can be said that a neural network maps a set of inputs into a set of outputs using non-linear functions, e.g. calculate house prices given their size.

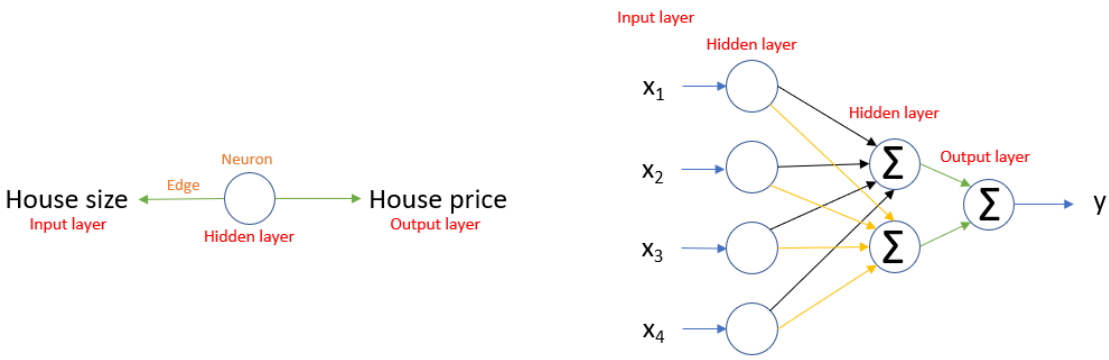

Figure 1. Example of a simple neural network

2.1.2. Weights & Bias

As observed in Figure 1, each neuron receives and outputs one or more connections. Each connection has an associated weight that represents the importance of that specific connection with respect to the overall neural network. The bias is a constant term used to add flexibility.

2.1.3. Activation Function

As mentioned, each neuron's output is determined using a non-linear function. These functions are referred to as activation functions. Neural networks are popular because of their capabilities to compute non-linear complex functions. It has been proved that by using non-linear activation functions, a neural network can approximate any continuous function, allowing for non-trivial problems to be solved. The following are some of the most used activation functions.

Table 1. Most-common activation functions

Name	f(x)	f′(x)	Range
Rectified Linear unit (ReLu)	$\begin{matrix} 0 \text{ for } x \leq 0 \\ x \text{ for } x > 0 \end{matrix} = \max(0, x)$	$\begin{matrix} 0 \text{ for } x < 0 \\ 1 \text{ for } x > 0 \\ \text{undefined for } x = 0^* \end{matrix}$	$(0, \infty)$
Sigmoid	$\frac{1}{1 + e^{-x}}$	$f(x)(1 - f(x))$	$(0,1)$
Hyperbolic tangent (tanh)	$\frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$	$1 - f(x)^2$	$(-1,1)$
Identity	x	1	$(-\infty, \infty)$

** In practice, 0 for x ≤ 0*

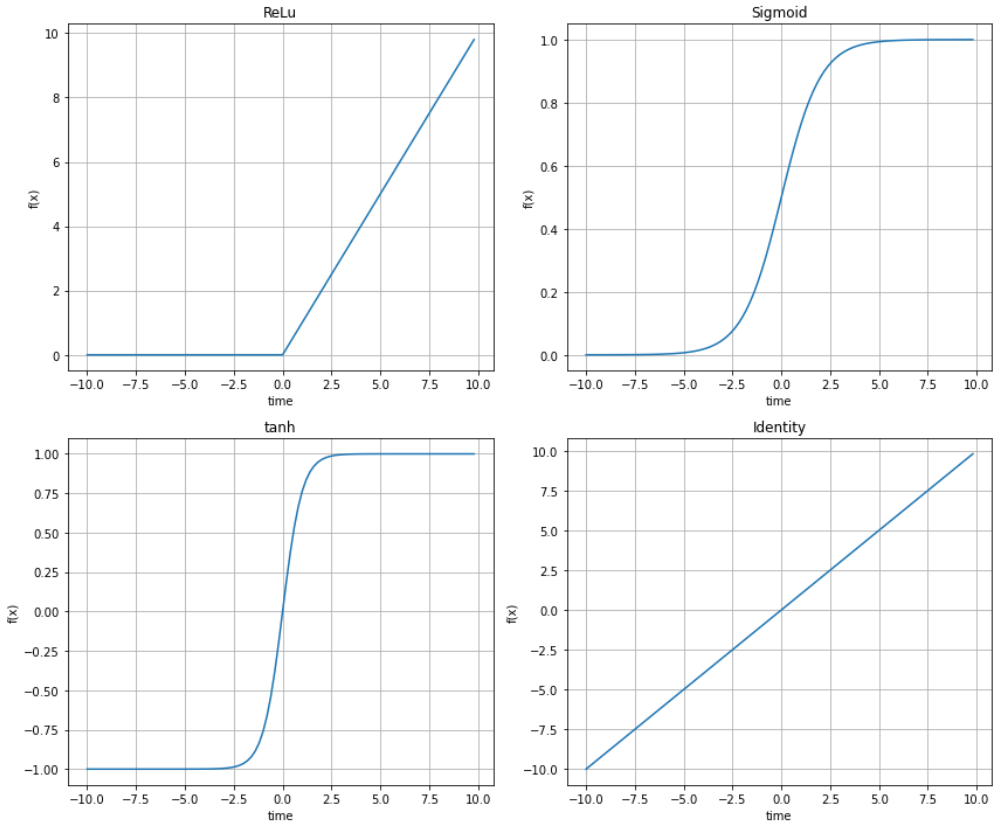

Figure 2. Plot of the most-common activation functions

After the weights and bias of each neuron have been calculated, a neuron's output/label is given by (1):

$$\text{output} = \text{activation function}\left(\sum(\text{weights} \times \text{inputs} + \text{bias})\right) \quad (1)$$

Equation 1 shows that the bias can determine if the output of a neuron is 'activated' or not. For example, consider that a neuron uses the ReLu as its activation function. If the bias is negative and small enough, the ReLu will output zero, essentially 'deactivating' the neuron.

2.1.4. Loss function

As mentioned, a neural network maps a set of inputs to a set of outputs. In order to get the desired outputs, the weights of the neural network must be adjusted or tuned. Generally, the total number of weights and biases of a neural network are referred to as parameters. Neural networks contain a high number of parameters, so it is difficult to calculate the exact weights required for a desired output. Instead, the performance of a network given a specific set of weights is measured using a loss function. The objective is to minimize the loss function by choosing/updating the appropriate weights of each connection. Table 2 shows the most commonly used loss functions:

Table 2. Most-common loss functions and their use

Loss function	Use	Equation
Mean-squared error	Linear regression	$\frac{1}{n} \times \sum_{i=1}^{n} (y_i - \tilde{y}_i)^2$
Cross entropy	Binary classification	$\frac{-1}{n} \times \sum_{i=1}^{n} y_i \times \log(\tilde{y}_i) + (1 - y_i) \times \log(1 - \tilde{y}_i)$
Dice loss	Image segmentation, information retrieval, etc.	$1 - \frac{TP}{TP + 0.5(FP + FN)}$
Focal loss	Class-imbalanced dataset	$-\alpha_t(1 - p_t)^{\gamma} \times \log(p_t)$

Where n is the number of training examples, $\tilde{y}$ is the truth label, y is the predicted label, TP are the true positives, FP are the false positives, FN are the false negatives, α_t and γ are scaling constants that deal with class imbalance, and p_t is the probability of prediction.

2.1.5. Class classification

Consider a scenario where the objective is to classify whether an animal is a cat or a dog. Each category can be referred to as a 'class.' Because there are two classes, this can be treated as a binary classification problem. For example, consider that if the animal is a cat, the neural network's output should be 0 (negative class), and if it is a dog, the output should be 1 (positive class). In this scenario, the following definitions can be introduced:

- True positives: An outcome where the model correctly predicts the positive class.
- False positives: An outcome where the model incorrectly predicts the positive class.
- False negatives: An outcome where the model incorrectly predicts the negative class.

2.1.6. Evaluation metrics

When/after training a neural network, its performance must be measured using reasonable evaluation metrics. Usually, the loss function and another metric (i.e. accuracy) are used to measure the model's performance. The metrics used depend on the loss function and the dataset. For example, if the dice loss is used, this suggests that the dataset is imbalanced (e.g. a dataset with 5000 images of cats and 100 of dogs) and a metric such as accuracy would not be ideal. The model may correctly predict all of the cat images while incorrectly predicting all of the dog images, giving an accuracy of 98% even though all the dog images were incorrectly predicted. Table 3 shows some of the most-common metrics used for the evaluation of neural networks:

Table 3. Most-common evaluation metrics and their use

Metric	Use	Equation	Range
Accuracy	Class-balanced dataset	$\frac{\text{number of correct predictions}}{\text{number of predictions}}$	0-1 or 0% to 100%
Precision	Positive predictions that were actually correct	$\frac{\text{TP}}{\text{TP} + \text{FP}}$	0-1 or 0% to 100%
Recall	Positive predictions that were predicted correctly	$\frac{\text{TP}}{\text{TP} + \text{FN}}$	0-1 or 0% to 100%
Dice coefficient	Class-imbalanced dataset	$2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}}$	0-1 or 0% to 100%

2.1.7. Training example

A training example is a pair consisting of an input and a truth output, e.g. (cat, 1) where cat is the input and 1 is the output, indicating that the image corresponds to a cat.

2.1.8. Dataset split

When training a neural network the dataset is usually split into three subsets known as training, validation and test sets. The data-distribution between these sets is determined based on the size of the original dataset. However, a common split is 80%, 10% and 10% for the training, validation and test sets, respectively. All of the subsets must come from the same distribution.

- Training set: The purpose of this subset is to use each of its training examples to train a neural network. The neural network is trained by iterating over the training set multiple times. Iterations through the training set are known as epochs.
- Validation set: The purpose of this subset is to tune the hyperparameters of a neural network and avoid possible overfitting problems.
- Test set: As its name says, the purpose of this subset is to test the performance of the network to unseen data. Even though the truth label of each training example on this subset is known, only the input is fed to the trained neural network and its predicted output is compared with the truth label.

2.1.9. Learning

Equation (1) shows that the output of a neuron is determined by its weights. Similarly, it can be said that each prediction of a neural network depends on all the individual weights of each neuron. The process of constantly adjusting the weights of a neural network by iterating the training set is known as learning. Backpropagation is the most-common method for the weight's update. Backpropagation computes the gradient of the loss function with respect to the network's weight and adjusts them accordingly. The process of backpropagation is complicated and involves the use of derivatives and the chain rule.

2.1.10. Learning rate

When training a neural network, the objective is to constantly adjust the neuron's weights until the loss function is minimized. The learning rate determines the step size of this adjustment. If the learning rate is small, the weights will be adjusted at a small pace, and vice versa.

2.1.11. Hyperparameters

The hyperparameters of a neural network are a set of variables that can be configured or tuned, e.g. learning rate, number of hidden units, number of layers, number of epochs, and activation functions.

2.1.12. Underfitting and overfitting

A neural network can map a set of inputs to a set of outputs. This mapping is done by approximating/fitting a function/polynomial. Because the objective of a neural network is to predict an outcome for unseen data, the fit of the function can become a problem. If a neural network cannot fit the training set's polynomial, it will fail to predict examples from all the data subsets. This is known as underfitting. On the other hand, if a neural network perfectly fits the training set's polynomial (this polynomial can be of high order, e.g. 80^{th} order), it will correctly predict the training set, but will fail for unseen data. This phenomenon is known as overfitting.

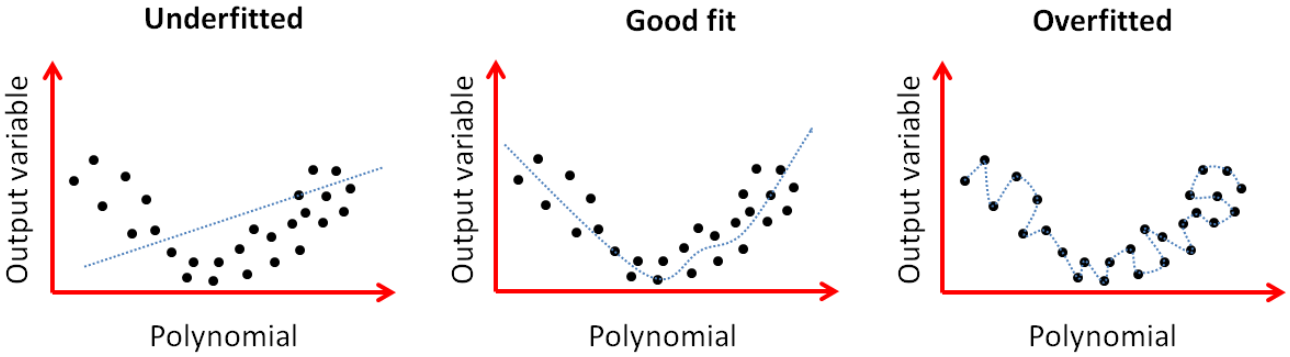

Figure 3. Example of underfit, overfit and 'good' fit

2.1.13. Convolutional neural networks

Convolutional neural networks are a type of deep neural networks. When dealing with 2-dimensional (2D) or 3-dimensional (3D) data such as images and volumes, a CNN is superior to a classic neural network. As its name says, in a CNN, the input matrices are convolved with filters. These filters can capture features that a classic neural network would not be able to, e.g. vertical and horizontal edges. However, the main feature of a CNN is that instead of using predetermined filters, a CNN learns its own filters and extracts different features from it.

2.1.14. Filters

Filters (also known as kernels) are matrices that when convolved with an image, can apply/extract features such as edge detection, blurring, sharpening, etc.

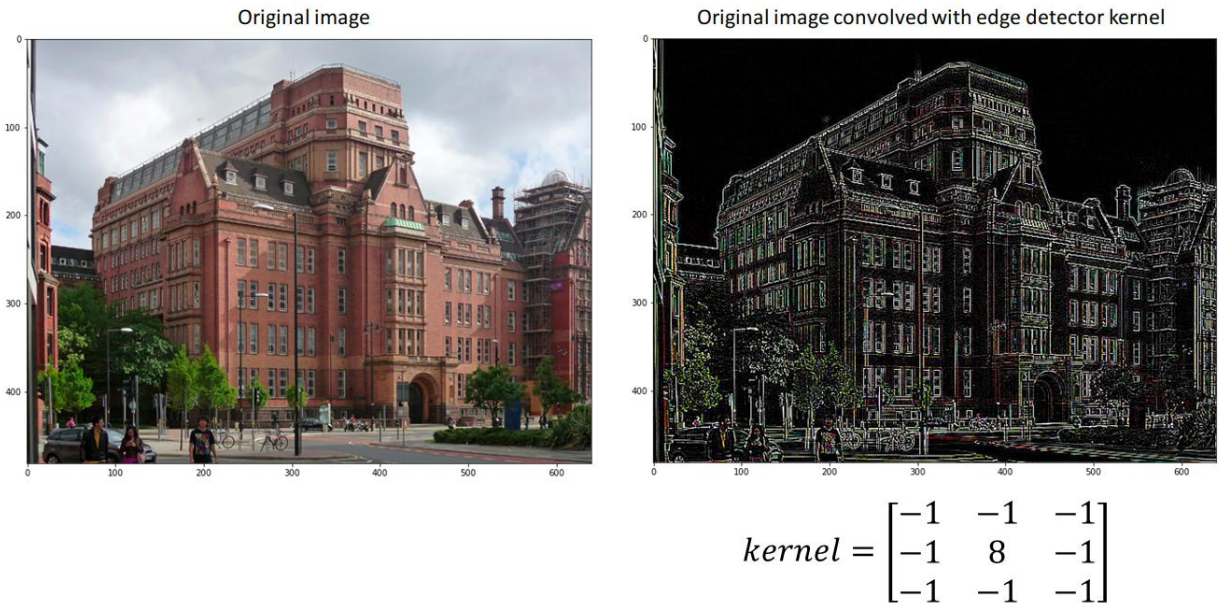

Figure 4. Image of Sackville street building convolved with an edge detector filter

2.1.15. Convolution

A convolution is a linear mathematical operator. It is represented by the symbol '*'. Generally, it is defined as the integral of the product of two functions after one has been reversed and time shifted. A convolution can be continuous or discrete. When working with discrete data (images, audio, etc.), the discrete convolution is used. In the context of CNNs, 2D or 3D convolutions are commonly used, with the first mentioned being the most popular option. A common representation of the 2D convolution operation between an input matrix and a filter is to visualize the filter 'sliding' over the input data performing element-wise multiplication and additions. Figure 5 illustrates a 2D convolution.

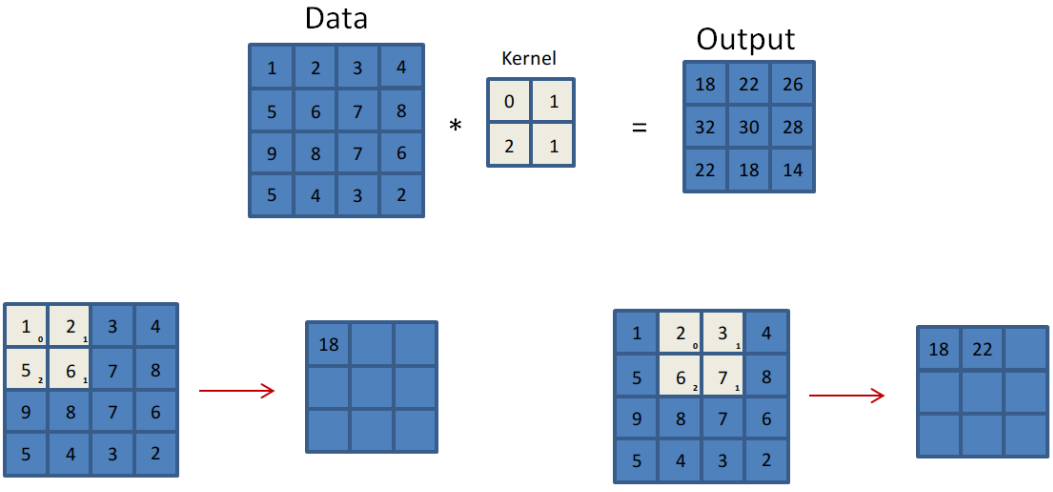

Figure 5. Example of a 2D Convolution

2.1.16. Stride

The stride defines how many pixels the kernel will move. On Figure 5, the stride is one, as the kernel is moving one pixel horizontally and vertically. The row-column stride can differ, but it is often symmetrical.

2.1.17. Valid and same padding

A 'valid' convolution is a convolution where no padding is involved, e.g. the convolution observed on Figure 5. However, sometimes it is desired for the output to maintain the same shape as the input data, especially for deep CNNs with many convolutional layers. This can be achieved using a 'same' convolution. A same convolution uses zero-padding to ensure that the shape of the input and the output do not differ after the convolution operation.

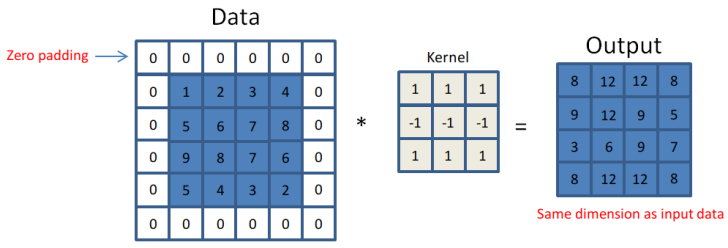

Figure 6. Example of a same convolution

2.1.18. Layers of a convolutional neural network

A CNN is comprised by multiple layers that perform different operations. Some of these layers are the following:

- Convolutional layer: Performs the convolution operation.
- Pooling layer: The objective of this layer is to down-sample an input image while conserving most of its information. The most common pooling methods are maximum pooling and average pooling.

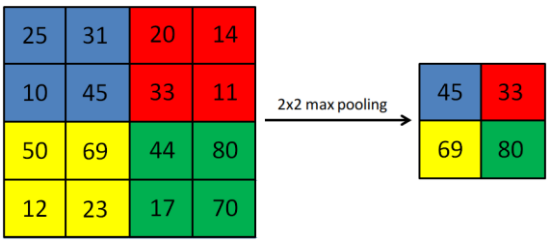

Figure 7. Example of 2x2 maximum pooling

- Up-sampling layer: In contrast with a pooling layer, the purpose of the up-sampling layer is to up-sample an image. This layer is commonly used in image segmentation as it allows recovering the original dimension of an image.
- Concatenate layer: As its name says, the purpose of this layer is to concatenate the results of different layers. This layer is not commonly used, but it is present on some SOTA architectures [6].
- Dropout layer: Dropout is a regularization technique. The objective of regularization is to avoid overfitting. Dropout works by deactivating random neurons during each epoch. For example, a dropout layer with a 'drop' parameter of 0.5 indicates that the neurons connected to that layer have a 50% chance of being deactivated.
- Fully connected network: After the input data has passed through the CNN, the features have been already extracted. Next, the extracted features are used as input to a deep neural network that carries out the prediction/classification of each pixel. However, this can also be done using a 1x1 convolutional layer with the sigmoid activation function.

2.1.19. Kernel initializer

It was previously mentioned that a CNN calculates its own kernel values and extracts features from it. However, the kernel values must be previously initialized. If the kernel values are 'incorrectly' initialized, two problems known as vanishing gradient and exploding gradient may occur. A popular initialization to avoid the mentioned problems is the He initialization [11].

2.2. Classical machine learning approaches

Before the deep learning era, classical machine learning approaches were the preferred option for the task of medical image segmentation, especially stroke segmentation. Nevertheless, machine learning approaches are still widely used as there are some benefits attached to them, e.g. training speed.

In [12], Maier et al performed an evaluation and comparison between nine classification methods for the task of acute stroke segmentation. Out of the nine classifiers trained, eight were machine learning classifiers; the other was a neural network. For the evaluation of the classifiers, different metrics were used, but the authors emphasise on the dice coefficient (the higher the dice coefficient, the better the classifier). The dataset used to train the classifiers consisted of 37 MRI brain scans. Table 4 shows the most-popular classifiers evaluated by the authors.

Table 4. Best classifiers used on Maier et al [12]

Classifier	Dice coefficient	Train time
5 Nearest Neighbour	0.58 ± 0.18	5 seconds
Extra Trees	0.64 ± 0.19	3 minutes
Gaussian Naïve Bayes	0.48 ± 0.22	1 second
Random Decision Forest (RDF)	0.67 ± 0.18	6 minutes
Convolutional Neural Network	0.67 ± 0.18	2 hours

As observed on Table 4, the RDF classifier seems to perform the best, taking into account the dice coefficient and the training time. The authors agreed that the RDF classifier was indeed the best choice. However, throughout the paper, the authors constantly mentioned that one of the main drawbacks of ML classifiers is the need of hand-tailored features that are time-expensive to create. On the other hand, the authors recognized that the CNN's performance was head to head to the RDF's and mention that whilst the CNN required a longer training time and higher computational costs, its highly configurable hyperparameters may bare for high potential if tuned properly.

2.3. Modern deep learning approaches

Due to the quantity of data available nowadays, the popularity of deep learning systems has increased during the last decade, especially in the field of medical imaging segmentation where the acquisition of data is not an easy task. Hence, it is not surprising that many researches have opted to use CNNs for the segmentation of stroke lesions.

Generally, deep learning classifiers require a huge amount of data. However, in [6], Ronneberger et al introduced 'U-Net', an award-winning CNN for biomedical image segmentation that relies on data augmentation techniques, allowing for excellent results using a small dataset. The network's architecture consists of a down-sampling and an up-sampling path, together with concatenations. The CNN was evaluated three times, using three different datasets with an average of 28 training example pairs containing the 2D images and the manually segmented labels. Overall, the network's performance is exceptional, achieving SOTA results. Furthermore, the authors mention that the network can be further applicable to different medical image segmentation problems.

As a consequence of the astonishing results produced by the U-Net model, many researchers have developed new models using the U-Net as a reference. For example, in [13], Zhou et al introduced 'D-Unet', a CNN used for the segmentation of chronic stroke lesions. The main highlight of the D-Unet model is that it combines the use of 2D and 3D data for the segmentation process. When using 3D data, the features obtained by the filters during the convolution process are superior than when using 2D data, mainly because the brain scans were originally volumetric data, not 2D data. In addition, the authors introduce an 'enhanced mixing loss' (EML) that combines the dice loss and focal loss to increase the convergence speed of the classifier. The network was trained using the ATLAS dataset, achieving a dice coefficient of 0.5349 after 150 epochs, the highest obtained up to this date. The EML is defined as follows:

$$\mathrm{EML} = \frac{1}{\mathrm{N}} \times \text{focal loss} + \log \text{dice loss} \tag{2}$$

Where N is the size of the dataset used to train the network.

On the other hand, different authors have decided to develop classifiers that are not based on the U-Net architecture. For instance, in [14], Qi et al introduce 'X-Net', a stroke lesion segmentation method based on Depth-wise Separable Convolutions (DSC) and long-range dependencies. The main highlight of X-Net, is the reduced number of parameters when compared with models such as the U-Net, allowing for lower memory requirements and faster train times. Similarly, the authors introduce a Feature Similarity Module, (FSM) that is used to capture the long-range spatial contextual information, helping the classifier when identifying strokes of different shape and sizes. Furthermore, the authors designed the 'X-block'. The X-block contains the basic operations present on the classical U-Net, but the convolutions are replaced with DSC operations, and batch normalization block are added in between convolutions. The X-Net was trained using a 2D normalized version of the ATLAS dataset, achieving a dice coefficient of 0.4867 after 100 epochs, not far from SOTA results.

2.4. Data pre-processing

A topic of high relevance is the pre-processing of data used to train both machine learning and deep learning classifiers. Generally, when dealing with medical image segmentation, data pre-processing pipelines are used. However, because machine learning models use hand-tailored features instead of extracting their own, the pre-processing of the dataset is more important for such classifiers.

In [12], Maier et al explains that the dataset passed through a pre-processing pipeline defined on [15]. On [15], Maier et al use a dataset of 37 MRI brain scans and define a pre-processing pipeline with the objective to facilitate the classification of the classifier. The first step of the pipeline is the resampling of the scans to a common voxel spacing. Secondly, the brain images were registered to account for movement and other small errors during the acquisition of the scan. Next, the skull was stripped out of the scans to extract the brain. Subsequently, the bias field was corrected to increase homogeneity between tissue types. Lastly, an intensity standardisation algorithm was applied to remove intensity scale differences. Figure 8 is illustrated on Maier et al [15], and shows the pre-processing steps performed by the authors.

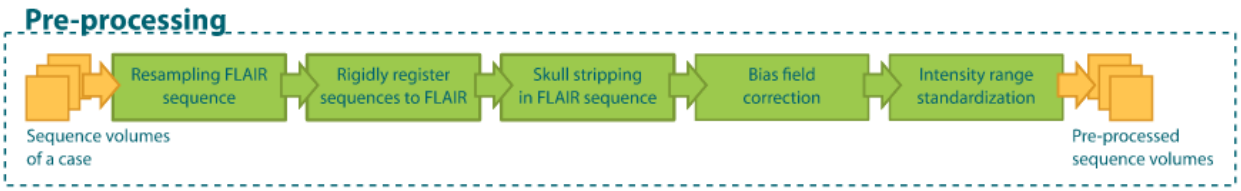

Figure 8. Pre-processing pipeline presented on Maier et al [15]

The classifier trained on [15] is an Extra Tree Forest classifier and its performance was evaluated using the dice coefficient, obtaining an average value of 0.65. Unfortunately, the authors do not train a classifier without the pre-processing pipeline, so it is difficult to further evaluate the effectiveness of this step. However, judging by the dice coefficient achieved and the justified decisions for the implementation of each pre-processing step, it is clear that the classifier performed well.

In contrast with [12] and [15], on [6] and [14], the authors trained a CNN without any data pre-processing steps, showing that an end-to-end training is possible for the networks. The decision of not using a pre-processing pipeline seems to be logical because the dataset did not consist of complex scan such as an MRI scan, instead, microscopic images were used. Similarly, in [13], the only pre-processing step was the resizing of the input images, demonstrating that an end-to-end approach is also possible with this architecture.

Lastly, it is worth mentioning that on [15], the authors mentioned that in some cases, imperfections present on the images caused by the skull-stripping algorithm caused a negative impact on the model's performance.

2.5. Observations

After evaluating the work of different researchers and academics, the following observations could be made:

- Classical machine learning classifiers are faster to train than deep learning classifiers, but they require complex hand-crafted features and data pre-processing pipelines. On the other hand, a neural network is capable of providing an end-to-end approach.
- Even though the U-Net model was not designed specifically for the task of stroke segmentation, a vast majority of the researchers on the topic use its architecture as a base/reference model; while others use it compare their results.
- Researchers perform different pre-processing steps and justify their decision, but do not focus on the benefits obtained by using a pre-processing pipeline, i.e. they do not compare the performance between a classifier that uses a pre-processing pipeline and one that does not.
- Brain scans are typically on the form of MRI scans, known for being volumetric data. For this reason, some researchers decide to train their classifiers using 3D data or 2D data, and sometimes, a combination of both. While the use of 3D data has its advantages over 2D data, an important drawback is that the computational costs to train such classifiers are high.
- Pre-processing steps such as skull-stripping may negatively impact a model's performance.

3. Project Methodology

It is important to mention that all the procedure described on this section was performed on Python using the deep learning framework Keras [16]. For further details regarding the code, refer to Appendix 1.

3.1. Dataset analysis

The ATLAS dataset contains 229 segmented T1-weighted MRI brain scans. Each scan is given in the format of nii files, the common format for volumetric medical data. For the manipulation of the nii files, the libraries NiBabel [17] and Numpy [18] were used. The scans are volumes of shape 197x233x189 and they all have one or more stroke lesions. The first two dimensions indicate the height and width of the scans, while the last corresponds to the depth. In order to visualize the data in a clear way, 2D images were generated by slicing the scans on the z-axis, resulting in images of shape 197x233.

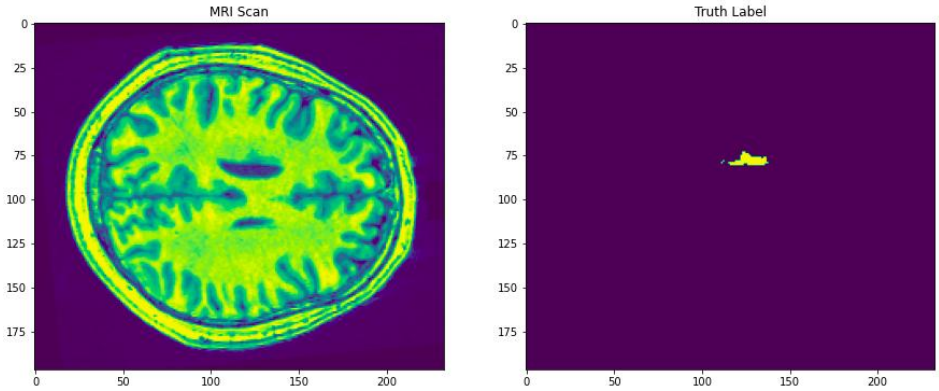

Figure 9. Slice of a brain's MRI Scan and segmented stroke lesion

3.2. Data Pre-processing

The ATLAS dataset does not contain any type of pre-processing. This can be proven by observing Figure 9 and noticing that the skull has not been stripped out from the brain. As mentioned in section 2, deep learning systems are able to provide an end-to-end approach. However, as mentioned on section 2.4, researchers have proved that pre-processing steps such as registration, skull stripping and bias-field correction are useful for the stroke segmentation task. For this reason, this report will evaluate the performance of a deep learning system by training and comparing two different models, one without any data pre-processing step, and another where registration, skull stripping and bias-field correction have been applied. The following are definitions of the applied pre-processing steps:

3.2.1. Image registration

“Image registration is a method used to align multiple images to ensure the spatial correspondence of anatomy across different images.” [19]. Brains are unique, so it is difficult to compare one brain with another. The purpose of the registration process is to be able to make comparisons of medical images across different subjects. To perform the registration process, a template must be provided. The template used on this report is one of the most-used templates, the MNI152. The MNI152 is an ‘average’ brain and it was created using MRI scans of 152 individuals. This pre-processing step was carried out using the FSL library command, FLIRT [20] [21], (v6.0).

3.2.2. Skull stripping

Skull stripping is the removal of non-cerebral tissue. This pre-processing step was carried out using the FSL command, Brain Extraction Tool (BET) [22], (v6.0, frac=0.5).

3.2.3. Bias-field correction (BFC)

“Bias field signal is a low-frequency and very smooth signal that corrupts MRI images specially those produced by old MRI machines.” [23]. “Image processing algorithms such as segmentation, texture analysis or classification that use the grey level values of image pixels will not produce satisfactory results.” [23]. The purpose of this process is to remove the bias-field signal, allowing for a better segmentation. This process was carried out using the ANTs library command, N4BiasFieldCorrection [24].

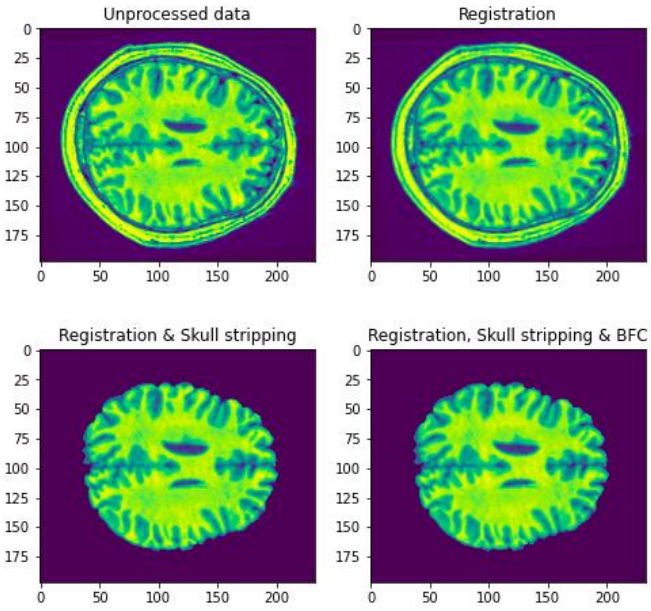

Figure 10. Pre-processing steps applied

3.3. Neural Network Implementation

After providing background information of neural networks and analysing the previous work of different researchers, the next step is to analyse possible problems and implement a model.

3.3.1. Dataset split

After the pre-processing of the data, the next step was to split the dataset into the training, validation and test subsets. There are a total of 229 brain scans on the ATLAS dataset. The distribution between the subsets was decided to be of 80%, 10% and 10% for the training, validation and test sets, respectively. Because 229 is an odd number, the split is not perfect, resulting in 185, 23 and 21 brain scans for the training, validation and test sets, respectively. This split was carried out for both of the datasets, the original dataset, and the pre-processed dataset.

3.3.2. Use of 3D or 2D data

As different papers show, it is possible to train a CNN using either 2D or 3D data, and in some cases, a combination of both. However, the use of 3D data involves high computational costs. Unfortunately, the Random Access Memory (RAM) and video-card memory requirements required for loading and processing a dataset of volumetric data could not be met. Hence, it was decided that instead, 2D data would be used to train the model. Each volume is of size 197x233x189, and by slicing each scan on the z-axis, each volume can be represented by 189 2D images. If this is applied to each scan, there will be 229x189 = 43281 2D brain slices, each of size 197x233.

3.3.3. Reference model

The selected model for this report uses the U-Net model as a reference. Figure 11 illustrates the original U-net model. The original U-net model can be understood as follows:

- The model takes a 2D image of size 572x572 as input. On Figure 11, the numbers above each box represents the number of filters while the numbers below represents the image size.
- The model has two paths, a down-sampling path and an up-sampling path, as well as a concatenation operation that connects both paths.

Down-sampling path (left-hand side): Performs two 3x3 2D convolutions with a ReLu activation function, followed by a 2x2 max pooling. The down-sampling process is performed five times, the number of filters starts at 64 and ends at 1024, while the image size starts at 572x572 and finishes

at 28x28. The purpose of the down-sampling path is to capture spatial relationships while reducing the image size (otherwise, it would not fit on GPU). The spatial relationships are captured using filters and it is common to increase their number according to the network's depth. Every time the filter's number is increased, the more complex are the features captured. For example, with 64 filters, the model can detect basic patterns such as edges, while with 512 filters the model can detect features such as circles, rectangles, etc.

Up-sampling path (right-hand side): After the image has been down-sampled, the next step is to concatenate the data from the down-sampling path to the data from the up-sampling path and perform 2x2 convolutions while reducing the number of filters. It is important to notice that even though the number of filters is being reduced on every stage of the up-sampling path, the network is still capturing important features. When concatenating the values from the down-sampling path, it is necessary to crop the image due to loss of border-pixels on each convolution.

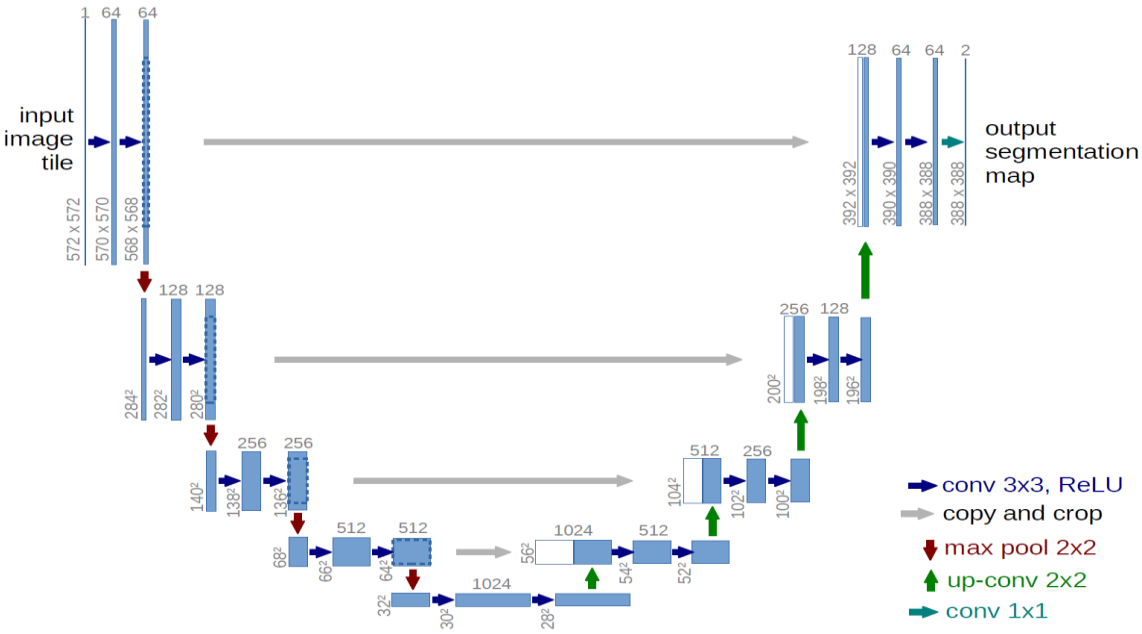

Figure 11. Original U-net [6]

Finally, after the down-sampling and up-sampling paths have been processed and concatenated, a 1x1 convolution is used to classify each pixel to a certain class. If a pixel is classified as the positive class, it indicates that the pixel is part of the predicted segmentation. On the other hand, if a pixel is classified as the negative class, it indicates that the pixel is not part of the segmentation.

3.3.4. Selected model

Now that a good understanding of the U-net has been provided, it is possible to introduce the

model used on this report. Figure 12 illustrated the model that will be used on this report. For visualization purposes, not all of the image sizes after the convolutions were shown. The model is based on the D-Unet and the model in [25]. Next, the model can be related to the ATLAS dataset and address possible problems and improvements:

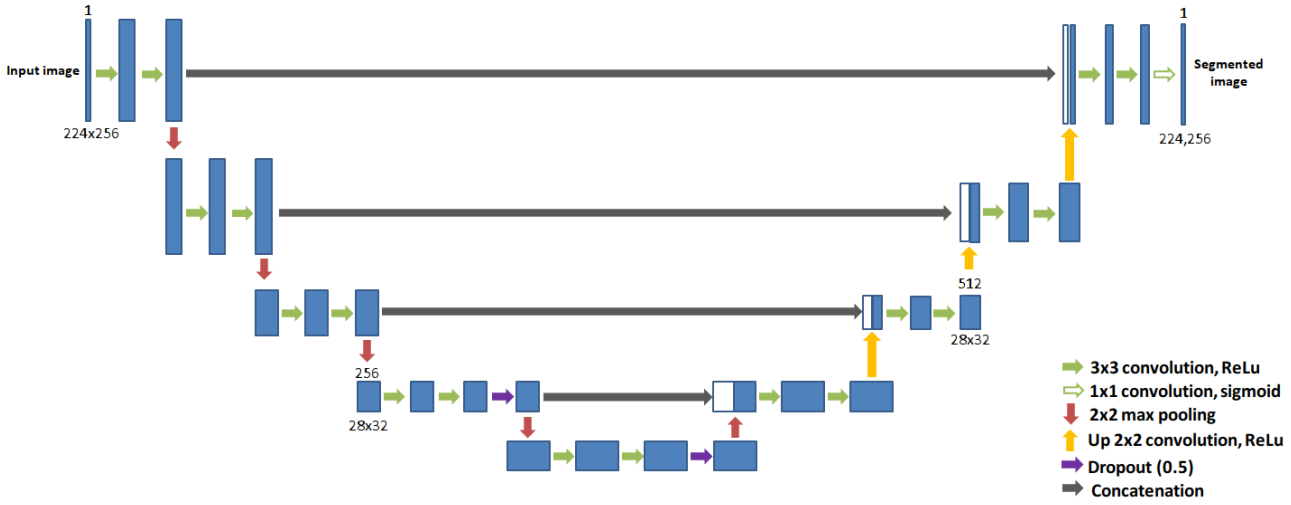

Figure 12. Developed model

3.3.4.1. Use of same padding

The original U-net model uses valid padding for the convolutions, causing the output and input data shape of each convolutional layer to differ. Because of the use of valid padding, it is necessary to crop the data before the concatenations. Moreover, the use of valid padding may cause a significant data-loss on the border pixels after several convolutions. For the mentioned reasons, the use of same padding is a more popular choice nowadays, and was the option used on this report.

3.3.4.2. Input size

When working with U-net-derived models, it is always important to pay attention to the size of the input images. Even though the valid padding was substituted to same padding, to avoid cropping the data, the input image size must be divisible by 32. The original model takes as input a square image of size 572x572, but rectangular images are also allowed. Because the images on the ATLAS dataset are of size 197x233, the images can either be cropped or zero-padded to the nearest number divisible by 32. Cropping an image can cause some of the data to be lost, so zero-padding is a better option. Hence, the original images were zero-padded to obtain an input size of 224x256.

3.3.4.3. Dropout

CNNs are known for their high number of parameters. By having a high number of filters, the model becomes extremely complex and there is a high risk of overfitting. Hence, in order to avoid a possible overfitting problem, two dropout layers with a drop parameter of 0.5 were used.

3.3.4.4. Loss function

When deciding which loss function to use, the most important factor to keep in mind is the dataset. Even though each scan present on the ATLAS dataset contains one or more stroke-lesions, this is not the case when the dataset is split into 2D images. Out of the 189 slices per brain-scan, the vast majority of the slices do not contain a stroke-lesion. Hence, the 'new' dataset can be considered as a class-imbalanced dataset. The original U-net model uses a cross-entropy loss function, a loss function that is not ideal for class-imbalanced datasets. On the other hand, loss functions such as focal loss and dice loss are ideal for class-imbalanced datasets. The main difference between the cross-entropy and focal loss functions is that the first mentioned requires a high level of confidence before making a prediction, whereas the last mentioned allows for more 'freedom' before making a prediction. Moreover, as mentioned on [13], Zhou et al developed the EML (2). The EML combines the focal and dice losses, producing SOTA results [13]. Hence, it was decided for this report's model to use the EML.

3.3.4.5. Learning rate

The learning rate is an important hyperparameter that is usually tuned after trial and error. There are many algorithms and techniques used to tune this hyperparameter [26]. Unfortunately, the computational costs required to train the model are extremely high, and there is was time and resources to tune this hyperparameter. However, a common choice of learning rate is between the range of 1^{-3} and 1^{-6} [13] [14]. If there is no confidence when choosing the learning rate, the wisest choice is to select a value small enough, as the worst-case scenario would be an increase of the training time. For the mentioned reasons, the learning rate used on this model was set to 1^{-4}.

3.3.4.6. Optimizer

It was previously mentioned that for the learning process of the network, the weights are constantly updated using gradient-based methods. The most common algorithm used is called gradient descent. Gradient descent is an optimization algorithm where the objective is to minimize

a target function. If the training set is large, gradient descent may take a long time as for every weight's update, the whole training set needs to be loaded into memory. On the other hand, a similar algorithm called stochastic gradient descent (SGD) performs the weight's update by loading a small and random subset of the training set called batch size. In cases where the training set is large, SGD reduces the computational burden, achieving faster weight updates in trade for a lower convergence rate than gradient descent [27]. Because the training set used on this report is large (43281 2D images), the SGD with a batch size of 8 was used for the training of the models.

3.3.4.7. Evaluation metric

The choice of an evaluation metric depends on the loss function and the dataset**.** Table 4 shows the most-commonly used evaluation metrics and their use. Even though the precision and recall are extremely important parameters when dealing with class-imbalanced datasets, both must be taken into account. For this reason, the dice coefficient, a metric that evaluates both the precision and recall, was used to evaluate the performance of the models.

3.3.4.8. Number of epochs and training time

As mentioned, one of the objectives was to train two models, one without any pre-processing steps apart from the 2D slicing and another with the pre-processing pipeline developed on section 3.2. Each model has a total of 31,031,685 parameters (weights + biases) and both were trained for 25 epochs. For better results, a higher number of epochs i.e. more than 50 would have been preferred, but the computational resources required for this were not available.

3.4. Computational resources available

For the training of the models used on this report, an Hp OMEN 15 laptop with the following specifications was used:

- RAM: 8 Gigabytes.
- Processor: Intel core i7 2.2GHz, 8^{th} generation.
- Graphic card: Geforce GTX 1050.
- Operating system: Ubuntu 18.04.

4. Results

Following the training of both models, the next step was to measure the model's performance on all the data subsets. After each epoch, the loss and dice coefficients of the training and validation set were calculated; whilst the dice coefficient for the test set was calculated after completing the training of the models.

4.1. Model without pre-processing

Figure 13 shows the results obtained during the training process of the model.

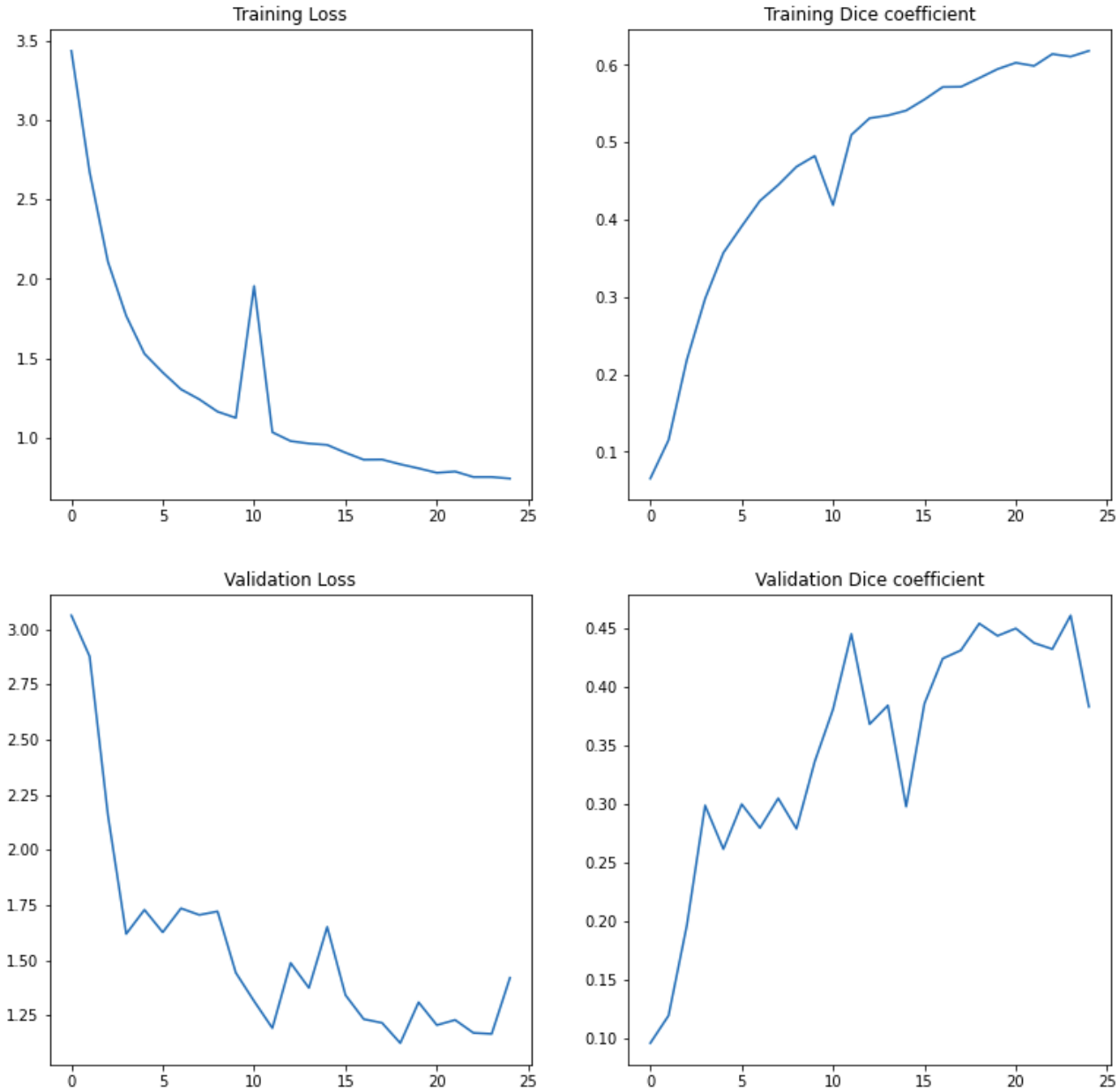

Figure 13. Training process for the non-pre-processed model

After the 25 epochs, the performance of the model on the test set could now be measured. The dice coefficient for the test set was measured to be 0.4143, a score not far from SOTA performance [13]. The following figures aim to illustrate examples of good, average and bad predictions performed by the model.

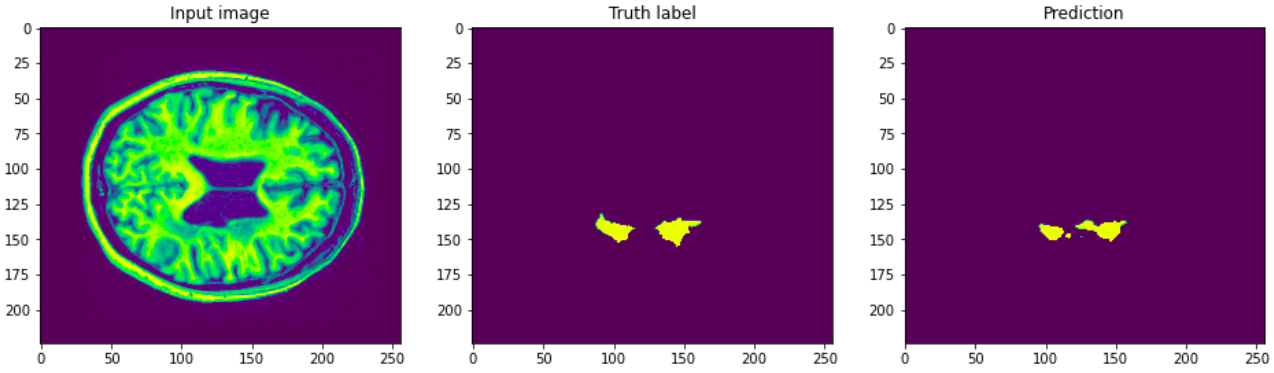

Figure 14. Good prediction of the non-pre-processed model

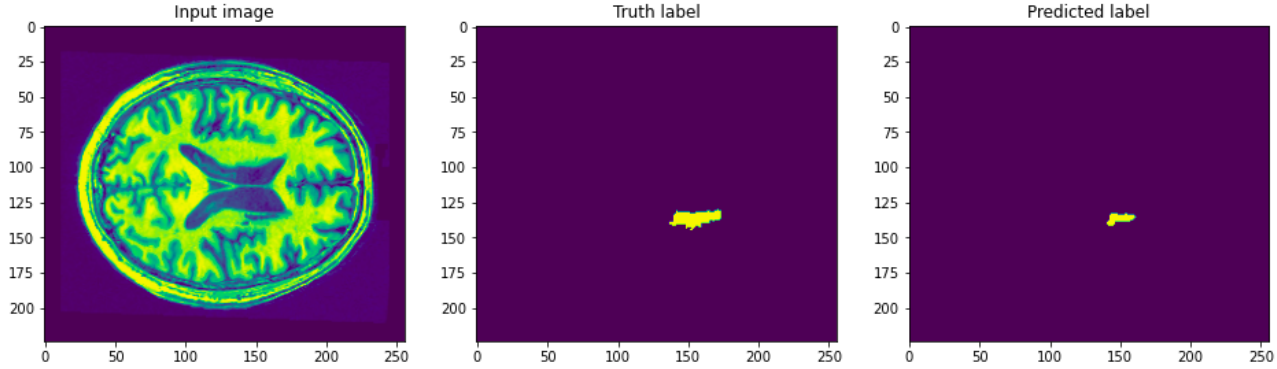

Figure 15. Average prediction of the non-pre-processed model

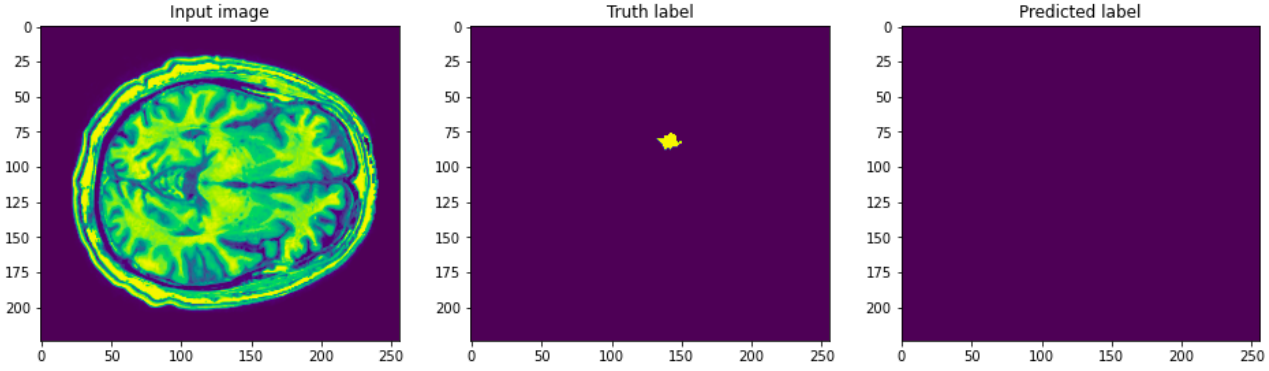

Figure 16. Bad prediction of the non-pre-processed model

4.2. Pre-processed model

Identical the normal model, the pre-processed model was also trained for 25 epochs using the same evaluation metrics as the normal model. Figure 17 illustrates the results obtained after the training of the pre-processed model.

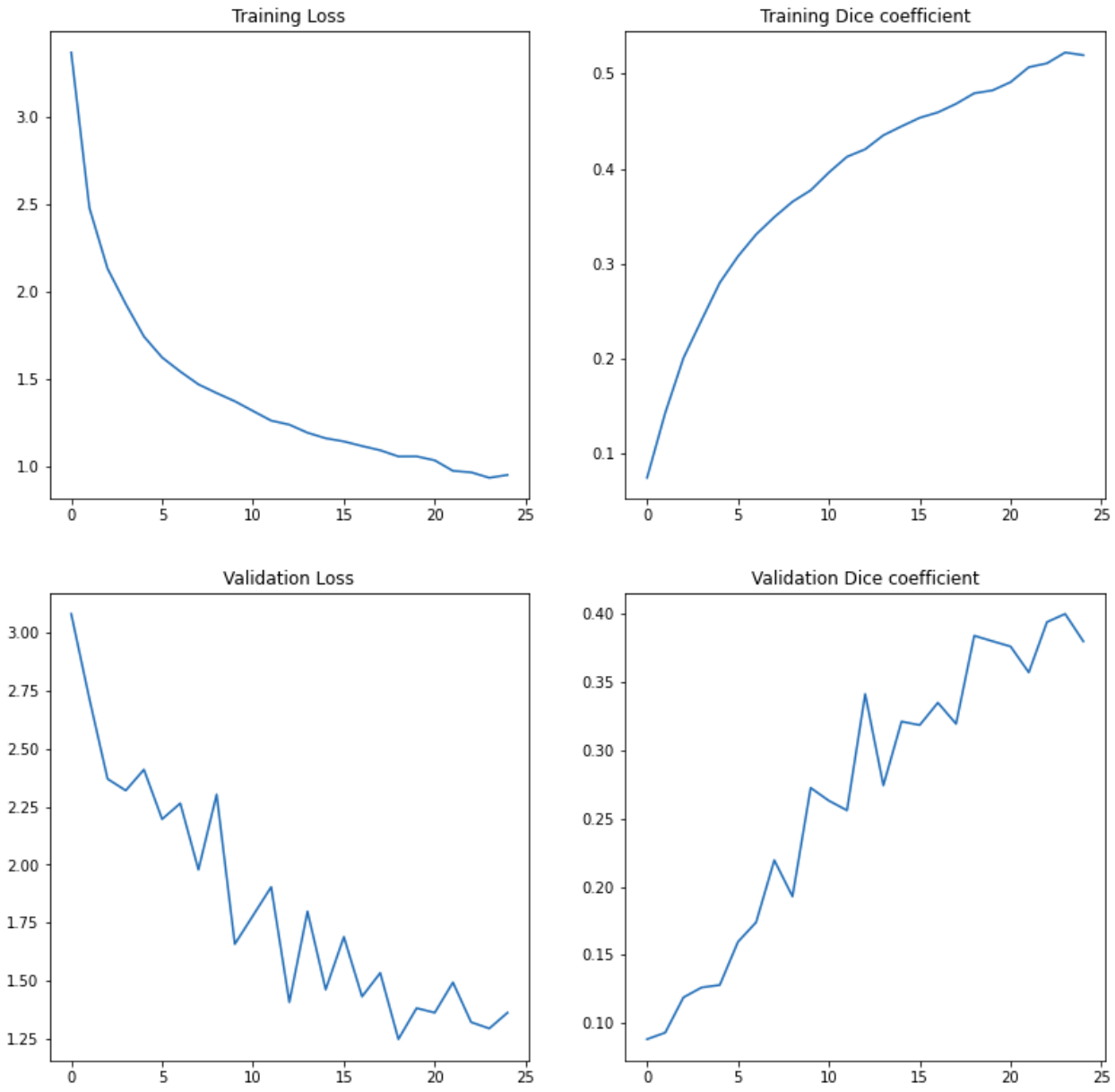

Figure 17. Training process of the pre-processed model

After the training of the pre-processed model, its performance to unseen data could now be measured. The dice coefficient of the test set for this model was measured to be 0.3635. The following figures aim to illustrate examples of good, average and bad predictions performed by the model.

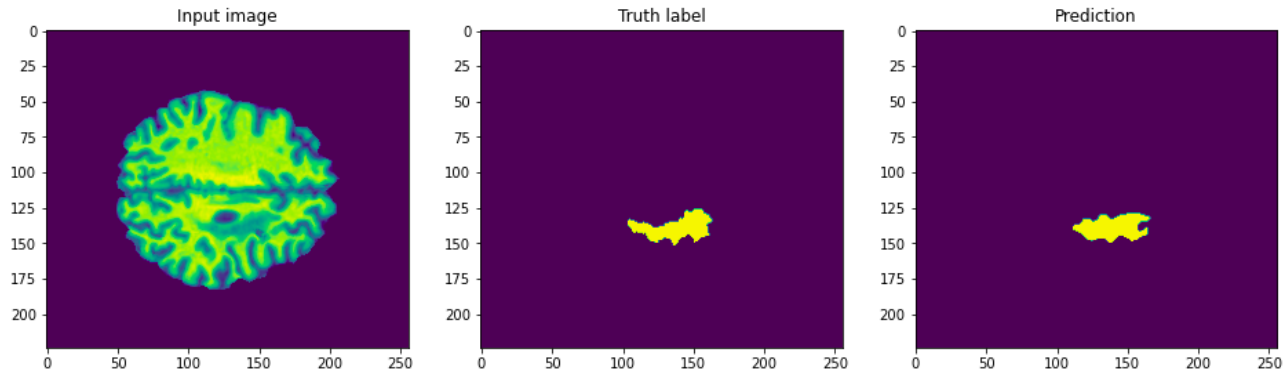

Figure 18. Good prediction of the pre-processed model

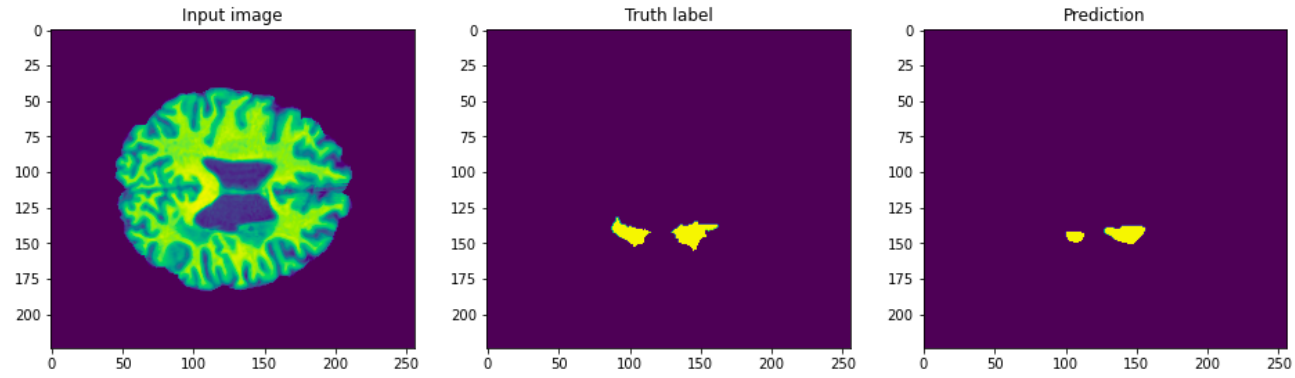

Figure 19. Average prediction of the pre-processed model

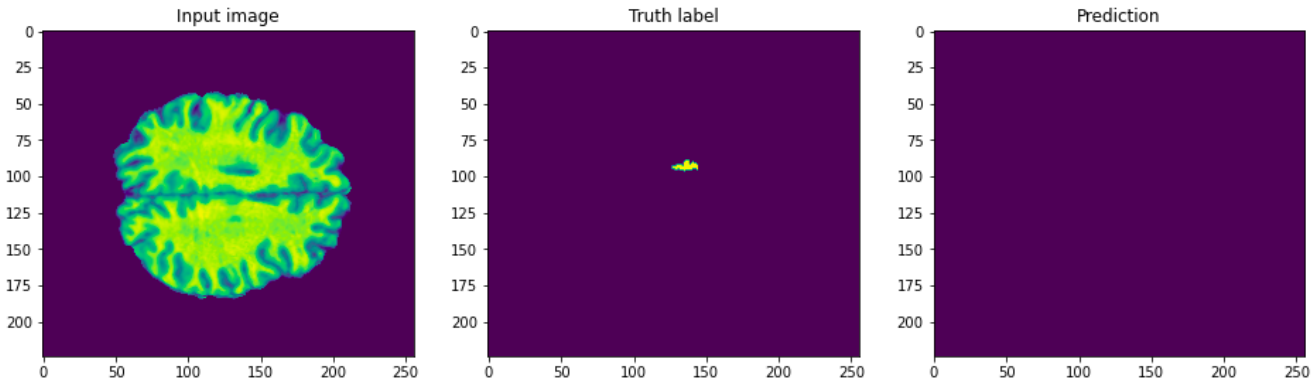

Figure 20. Bad prediction of the pre-processed model

The predictions presented above were manually selected and are just a small portion of the total test set. During the evaluation of the model's performance on the test data, an interesting and important phenomenon was observed. For the majority of the images containing a stroke lesion near to the skull, some/all of the stroke-lesion was stripped out of the brain scan. Figure 21 illustrates an example of the mentioned phenomenon.

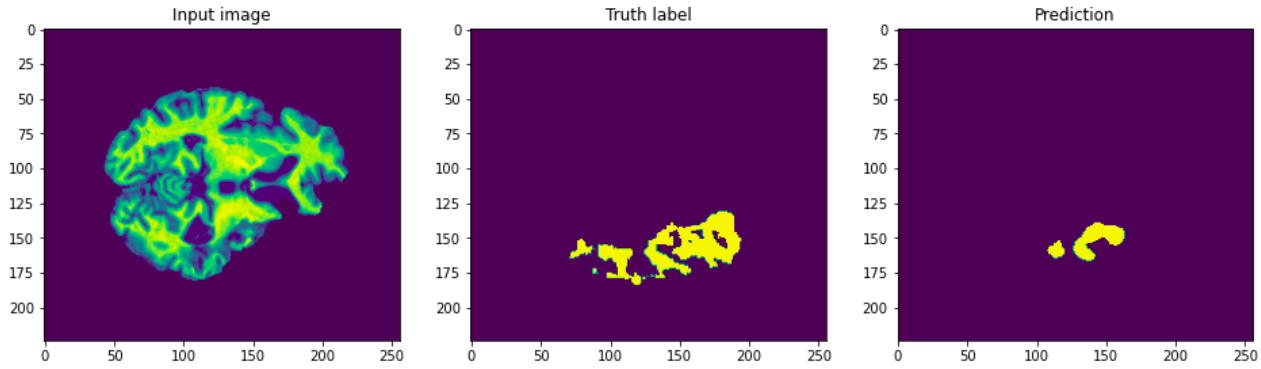

Figure 21. Stripped stroke lesion

4.3. Training and pre-processing time

As mentioned on section 3.4, each model was trained for 25 epochs. The average time took for each epoch to complete was approximately three hours, with a total of 75 hours required to train each model. In addition, before the training of the pre-processed model, the raw data had to go through the pre-processing pipeline. Table 5 shows the time taken for each pre-processing step to complete.

Table 5. Time taken for each pre-processing step to complete

Pre-processing step	Average time taken to process one scan
Registration	20 seconds
Skull-stripping	10 seconds
Bias field correction	3 minutes
Total pre-processing pipeline	3 minutes and 30 seconds

Furthermore, the time taken to predict and render a single brain slice is approximately 2 seconds. Besides, it is worth mentioning that the model's training, evaluation and prediction time, as well as the time required for the pre-process of the data is dependent on the computational resources available.

5. Discussion

The purpose of this section is to interpret and analyse the results obtained after evaluating the performance of both models.

5.1. Comparison of results

The plots obtained on section 4 agree with the expected behaviour. During each epoch, the model's weights are adjusted and the loss and dice coefficients are evaluated. For the training subset, it was expected for the loss and dice coefficients to decrease and increase at a constant rate, respectively. This behaviour was expected because the model's weights are improving during each epoch. On the other hand, while the model's weights are improving during each epoch, the model does not see the validation data during the training stage, it only evaluates the performance on the validation subset at the end of every epoch. Because the validation data is not seen during the training stage, the loss and dice coefficient plots are noisier than for the training set, but follow the same trend as the weights are still improving during each epoch. Furthermore, the 'spike' observed for the training loss and dice coefficient on Figure 13 is completely normal and it is caused by the randomness present on the SGD algorithm.

Table 6 shows the performance of different CNNs on the ATLAS dataset, as well as the number of epochs used to train the models.

Table 6. Comparison of performance between different models

Classifier	Dice coefficient	Number of epochs
D-Unet	0.5349	150
X-Net	0.4867	100
Trained model without pre-processing	0.4143	25
Trained model with pre-processing	0.3635	25

As observed, the D-Unet model provides the best performance on the ATLAS dataset, while the trained model with pre-processing is the worst. However, a comparison between the models trained on this report and external models is not suitable because of the large difference between

the numbers of epochs, and most importantly, it is not within the objectives of this report. Instead, a much fairer comparison, and within the objectives of this report, is to compare the results of the trained models, i.e. the model without pre-processing, and the model with pre-processing.

Clearly, the model without the data pre-processing is performing better on the test set as it has a higher dice coefficient. However, this does not strictly mean that a pre-processing pipeline has a negative effect on a model's performance as there is always the possibility that something went wrong along the pre-processing pipeline.

5.2. Problems along the data pre-processing pipeline

As mentioned in section 3.2, the pre-processing pipeline implemented for both models consisted of the following: Registration – skull stripping – bias field correction. By observing Figure 21 and analysing the pre-processing steps, it is clear that the removal of stroke tissue was caused by imperfections on the skull-stripping algorithm as it is the only pre-processing step that involves the removal of tissue. In fact, as stated on section 2.4, the authors mentioned that the skull-stripping algorithm had imperfections that had a negative impact on the model's performance, agreeing with the results obtained on this report. The skull stripping algorithm used on this report was the BET provided by the FSL library. The BET has different tuneable parameters, being the 'frac' parameter the most important. The frac parameter is used to set the fractional intensity threshold which determines where the edge of the final segmented brain is located. The default value is 0.5, and the valid range is between 0-1. If a small frac value is used, the segmented brain will be large. On the other hand, if a large frac value is used, the segmented brain will be small. In other words, the frac parameter can be thought of as a sensitivity parameter, if it is large, the sensitivity of the skull-stripping algorithm will be low, possibly leading to the removal of non-skull tissue, and vice-versa. For this report, the BET algorithm was applied to the ATLAS dataset using the default frac value (0.5). Hence, it was feasible to arrive to two possible and related causes of the errors experienced by the skull-stripping algorithm, and they are the following:

- On MRI scans, stroke lesions are commonly 'darker' than healthy brain tissue due to the presence of blood clots. Thus, the problem arises when the stroke-lesion is present near the skull-brain boundary region. The BET was not created for the skull-stripping of stroke-affected brains, so it is difficult for the tool to correctly extract the brain when there is a stroke-lesion near the skull. This factor, combined with the fact that the frac parameter

was not tuned, were likely the cause of the imperfections present on the skull-stripping algorithm.

Furthermore, in general, neural networks tend to perform better on datasets that come from the same distribution. For this report, even though the whole ATLAS dataset comes from the same distribution, i.e. all of the scans were performed following the same guidelines; the imperfections present on the skull-stripping algorithm had an impact on the distribution of the dataset. For the model without pre-processing, the model was trained and evaluated using an input – output pair where the input consisted of a brain scan containing one or more stroke lesions, and the output being the hand-labelled lesions. On the other hand, while this was also true for the pre-processed model, the input – output pairs seen by the model did not follow the same distribution. First, there is the scenario on where the input – output pair fed to the model corresponded to a correctly skull-stripped scan and a hand-labelled lesion, respectively. Second, there is the scenario on where the input – output pair fed to the model corresponded to an incorrectly skull-stripped scan and a hand-labelled lesion, respectively. In the first mentioned scenario, the model's weights are adjusted based on the fact that there is a lesion present on the input because that is what the hand-labelled lesion indicates. However, on the second mentioned scenario, the model's weights are also updated, although in this case, the input does not contain a stroke-lesion because of the errors on the skull-stripping algorithm, but the hand-labelled lesion still indicates that there is a stroke-lesion present. This phenomenon can be observed on Figure 21; the hand-labelled lesion indicates that the segmented tissue corresponds to a stroke-lesion even though some of that space is empty. As a result of having two different scenarios, the model is trying to predict the correct outcome for both scenarios, so the model's weights are adjusted according to both cases. Nevertheless, as observed on section 4.2, the model is able to learn despite of the errors caused by the skull-stripping algorithm. Comparing Figure 13 and Figure 17, it is clear that the pre-processed model is learning, but because of the mentioned situation, it is doing so at a smaller pace than the model without pre-processing.

5.3. Training and pre-processing time

While it was mentioned that both models are identical and were trained by the same number of epochs, before making a prediction, the pre-processed model requires for the data to go through the pre-processing pipeline. Table 6 shows that the time taken for a scan to go through the pre-processing pipeline is around 3 minutes and 30 seconds. On the other hand, the model without

pre-processing does not require any pre-processing steps, and can segment a brain slice in around 2 seconds. Clearly, based on the dice coefficients and prediction times of both models, the model without pre-processing is a better option than the model with pre-processing. However, the results were obtained after 25 epochs, as Table 5 shows, a small number when compared to SOTA models. Besides, the performance of the model with pre-processing was affected by the problems present on the skull-stripping algorithm. For these reasons, even though the model without pre-processing had a better performance than the model with pre-processing, a further training of the models is required to decide whether it is worth to wait the time taken to pre-process a scan or not.

5.4. Limitations

CNNs are known for their huge number of parameters, resulting in high computational costs. As mentioned before, the models trained on this report consist of 31,031,685 parameters. As a consequence, the computational resources available were not enough to achieve the wanted results. While each model was trained for 25 epochs, an ideal number would have been above 50, but this was not possible due to the available resources and the time constraints. Because of the computational limitations, several attempts were made to train the models on cloud services such as Microsoft Azure and Google colab. However, the dataset size was greater than 50 Gigabytes, and the majority of the cloud services could not cope with such file sizes. Furthermore, the option to rent a private virtual server such as DigitalOcean was also evaluated, but the allocated budget was not enough to cover the costs for this.

Similarly, it is common for CNN-based research paper to be developed in groups. The task of developing and evaluating new network architectures requires a huge amount of budget and time. For this reason, even though features that were implemented and selected such as dropout, EML, learning rate, optimizer, etc., were justified according to previous research, there was not enough time and budget to properly tune and test them. For example, it was mentioned that the learning rate was selected according to the common choices used by researchers on the topic. However, while this is a justified action, it was not the preferred option. Instead, approaches were models with different learning rates, loss functions, optimizers, etc., were trained and compared would have been a much better option.

6. Conclusion

This report aimed to evaluate the feasibility of using a data pre-processing pipeline to train a convolutional neural network for the segmentation of stroke lesions. Based on the results obtained after training two different convolutional neural network models, it can be concluded that the model without data pre-processing achieved a better performance than the model with data pre-processing. However, this was a direct consequence of the errors present on the skull-stripping algorithm and the model's small number of epochs. Hence, even though the model without data pre-processing achieved better results than the model with data pre-processing, the scenario presented on this report was extremely specific and further studies are necessary to arrive to a general conclusion on the feasibility of using a data pre-processing pipeline.

Furthermore, while this report thoroughly explained the pre-processing steps that were performed on the data, as well as their justification, it also raises the question of whether 'error-prone' pre-processing steps such as skull-stripping are worth implementing. In addition, as mentioned in section 2.5, researchers did not quantify the performance benefit obtained by implementing a pre-processing pipeline. On the other hand, in this report, the results of the two convolutional neural networks were quantified, carefully analysed and compared, helping to seal the existent gap on the topic.

Finally, as previously discussed, several limitations were faced during the completion of this report. Constraints such as time, budget and computational resources had a huge impact on the outcome of this report. Ideally, the two convolutional neural networks would have been trained for 100 or more epochs. When planning the methodology for this report, the objective was to develop many pre-processing pipelines with different steps and select the best possible combination. The same applies to the model's hyperparameters. Hyperparameters such as the learning rate, number of epochs, loss function and optimizer are generally defined after trial and error. Nevertheless, despite of the limitations present on this report, the results and knowledge obtained were useful and satisfactory.

6.1. Future work

While the results obtained on this report allowed for interesting discussion, further studies and testing are needed to reach a general conclusion on the feasibility of a data pre-processing pipeline for the task of stroke segmentation. The following are some of the recommendations that researchers on the topic could evaluate.

6.1.1. Use of batch normalization layers

As mentioned on section 3.3, CNNs are known for their high number of parameters. The deeper a network is, and the bigger the number of filters, the more parameters the network will have. The network trained on this model had more than 30 million parameters, a high number even for a CNN. A problem that arises with having a high number of parameters is not only the increase in training time, but also the risk of overfitting. By having a high number of parameters, the polynomial representing the training set becomes extremely complex and specific. Hence, there is the risk for the model's weights to be adjusted in a manner such that they perfectly fit the training set. An example of this can be seen on the right-hand plot observed on Figure 3. Fortunately enough; researchers have proved that the use of dropout layers help to tackle the problem of overfitting. However, during recent years, the use of batch normalization layers has become a popular option amongst researchers for different reasons. For example, on [28], Garbin et al mention that the use of batch normalization layers helps to increase the accuracy of a network in exchange to a small increase on training time. Furthermore, the authors also pointed out that in some cases, the use of batch normalization layers introduced regularization effects, deeming dropout layers as unnecessary. After testing different models, the authors conclude that batch normalization layers should be the preferred option as they obtained the best performance with only a small increase on training time. In addition, networks such as the D-Unet and the X-Net use batch normalization layers instead of dropout layers.

6.1.2. Use of 3D data

Throughout the completion of this report, it was mentioned that the ATLAS dataset consists of MRI brain scans. MRI brain scans are known for being 3D data. Because of the computational resources available for this report, the scans had to be sliced on the z-axis, converting the 3D scans into 2D images. However, a problem that arises by slicing the volumes into 2D images is the loss of information and spatial relationships originally present on the scans. For this reason, researchers have evaluated the use of 3D data for the segmentation of stroke lesions, and even a combination of both 2D and 3D data. For example, as mentioned on section 2.3, in [13], Zhou et al acknowledge that the features extracted using 3D data facilitated the ability of the network to identify small lesions. The structure of the D-Unet is similar to the classical U-Net, but the main highlight is that it uses both 2D and 3D convolutions for the down-sampling path of the network. Furthermore, after performing the 3D convolutions, the authors performed a dimensionality reduction technique in order to 'squeeze' the 3D data and combine it with the 2D data.

6.1.3. Use of X-blocks

Undoubtedly, the U-Net is the most-common architecture used for the segmentation of medical images. Even though the original U-Net is not the preferred option nowadays, it is the base model for the majority of the networks dealing with medical image segmentation. However, there are exceptions to this, and different researchers have tried to create models that are not based on the U-Net. For example, as mentioned on section 2.3, in [14], Qi et al introduced the concept of the X-Net. The X-block is a combination of batch normalization layers, ReLu activation functions, 1x1 convolutional layers and depth-wise separable convolution layers. By using the X-block, the authors effectively reduced the number of parameters to 15.1 million, a significant smaller number when compared to the 31 million of parameters of this report's model. Hence, by reducing the number of parameters, the training time is also reduced, allowing for the model to be trained by a higher number of epochs.

6.1.4. Implementation of new and/or different pre-processing steps

The segmentation of stroke lesions is a popular topic. Hence, every year multiple researchers produce papers that introduce new techniques and advances for the task of stroke lesions segmentation. In this report, the pre-processing pipeline applied to the data was based on the evaluation of previous literature. For this reason, even though the objective was to achieve the best results, it is possible for a different pre-processing pipeline to produce much better results. Similarly, it was observed that pre-processing steps such as skull-stripping can produce errors that may have a negative impact on the model's performance. Thus, in this case, a removal of error-prone pre-processing steps such as skull-stripping may be beneficial for the network's performance.

6.1.5. Hyperparameters tuning

When training a neural network, it is common to train a model multiple times using different set of hyperparameters. The purpose of using different hyperparameters is to find the combination that performs the best on the test set. Ideally, the hyperparameters should always be tuned when training neural networks. However, the process of tuning the hyperparameters is an extremely time-consuming process. An example of hyperparameters tuning could be the selection of different learning rates and number of epochs.

7. References

[1]"Stroke | NHLBI, NIH", Nhlbi.nih.gov. [Online]. Available: https://www.nhlbi.nih.gov/health-topics/stroke. [Accessed 16 October 2020].

[2]G. Donnan, M. Fisher, M. Macleod and S. Davis, "Stroke", The Lancet, vol. 371, no. 9624, pp. 1612-1623, 2008. Available: 10.1016/s0140-6736(08)60694-7 [Accessed 18 October 2020].

[3]D. Lloyd-Jones, "MRI interpretation - T1 v T2 images", Radiologymasterclass.co.uk, 2017. [Online]. Available: https://www.radiologymasterclass.co.uk/tutorials/mri/t1_and_t2_images. [Accessed 22 November 2020].

[4]K. Ito, H. Kim and S. Liew, "A comparison of automated lesion segmentation approaches for chronic stroke T1-weighted MRI data", Human Brain Mapping, vol. 40, no. 16, pp. 4669-4685, 2019. Available: 10.1002/hbm.24729 [Accessed 23 November 2020].

[5]O. Maier, C. Schröder, N. Forkert, T. Martinetz and H. Handels, "Correction: Classifiers for Ischemic Stroke Lesion Segmentation: A Comparison Study", PLOS ONE, vol. 11, no. 2, p. e0149828, 2016. Available: 10.1371/journal.pone.0149828 [Accessed 25 November 2020].

[6]O. Ronneberger, P. Fischer and T. Brox, "U-Net: Convolutional Networks for Biomedical Image Segmentation", arXiv.org, 2020. [Online]. Available: https://arxiv.org/abs/1505.04597. [Accessed 25 November 2020].

[7]H. Wang et al., "Global, regional, and national life expectancy, all-cause mortality, and cause-specific mortality for 249 causes of death, 1980–2015: a systematic analysis for the Global Burden of Disease Study 2015", The Lancet, vol. 388, no. 10053, pp. 1459-1544, 2016. Available: 10.1016/s0140-6736(16)31012-1 [Accessed 27 November 2020].

[8]C. Coffey and J. Cummings, The American psychiatric press textbook of geriatric neuropsychiatry, 2nd ed. Washington: American Psychiatric Press, 2000.

[9]S. Liew et al., "A large, open source dataset of stroke anatomical brain images and manual lesion segmentations", 2017. Available: 10.1101/179614 [Accessed 03 January 2021].

[10]Y. Chen, Y. Lin, C. Kung, M. Chung and I. Yen, "Design and Implementation of Cloud Analytics-Assisted Smart Power Meters Considering Advanced Artificial Intelligence as Edge Analytics in

Demand-Side Management for Smart Homes", Sensors, vol. 19, no. 9, p. 2047, 2019. Available: 10.3390/s19092047 [Accessed 14 February 2021].

[11]K. He, X. Zhang, S. Ren and J. Sun, "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification", 2015 IEEE International Conference on Computer Vision (ICCV), 2015. Available: 10.1109/iccv.2015.123 [Accessed 16 February 2021].

[12]O. Maier, C. Schröder, N. Forkert, T. Martinetz and H. Handels, "Correction: Classifiers for Ischemic Stroke Lesion Segmentation: A Comparison Study", PLOS ONE, vol. 11, no. 2, p. e0149828, 2016. Available: 10.1371/journal.pone.0149828 [Accessed 18 February 2021].

[13]Y. Zhou, W. Huang, P. Dong, Y. Xia and S. Wang, "D-UNet: a dimension-fusion U shape network for chronic stroke lesion segmentation", IEEE/ACM Transactions on Computational Biology and Bioinformatics, pp. 1-1, 2019. Available: 10.1109/tcbb.2019.2939522 [Accessed 20 February 2021].

[14]K. Qi et al., "X-Net: Brain Stroke Lesion Segmentation Based on Depthwise Separable Convolution and Long-Range Dependencies", Lecture Notes in Computer Science, pp. 247-255, 2019. Available: 10.1007/978-3-030-32248-9_28 [Accessed 6 March 2021].

[15]O. Maier, M. Wilms, J. von der Gablentz, U. Krämer, T. Münte and H. Handels, "Extra Tree forests for sub-acute ischemic stroke lesion segmentation in MR sequences", Journal of Neuroscience Methods, vol. 240, pp. 89-100, 2015. Available: 10.1016/j.jneumeth.2014.11.011 [Accessed 8 March 2021].

[16] F. Chollet et al., “Keras”, 2015. Available: https://github.com/fchollet/keras. [Accessed 10 March 2021].

[17]M. Brett et al., nipy/nibabel: 3.2.1. Zenodo, 2020.

[18]C. Harris et al., "Array programming with NumPy", Nature, vol. 585, no. 7825, pp. 357-362, 2020. Available: 10.1038/s41586-020-2649-2 [Accessed 15 March 2021].

[19]X. Zhang et al., "Linear Registration of Brain MRI Using Knowledge-Based Multiple Intermediator Libraries", Frontiers in Neuroscience, vol. 13, 2019. Available: 10.3389/fnins.2019.00909 [Accessed 15 March 2021].

[20]M. Jenkinson and S. Smith, "A global optimisation method for robust affine registration of brain images", Medical Image Analysis, vol. 5, no. 2, pp. 143-156, 2001. Available: 10.1016/s1361-

8415(01)00036-6 [Accessed 15 March 2021].

[21]M. Jenkinson, P. Bannister, M. Brady and S. Smith, "Improved Optimization for the Robust and Accurate Linear Registration and Motion Correction of Brain Images", NeuroImage, vol. 17, no. 2, pp. 825-841, 2002. Available: 10.1006/nimg.2002.1132 [Accessed 15 March 2021].

[22]S. Smith, "Fast robust automated brain extraction", Human Brain Mapping, vol. 17, no. 3, pp. 143-155, 2002. Available: 10.1002/hbm.10062 [Accessed 15 March 2021].

[23]J. Juntu, J. Sijbers, D. Dyck and J. Gielen, "Bias Field Correction for MRI Images", Advances in Soft Computing, pp. 543-551, 2005. Available: 10.1007/3-540-32390-2_64 [Accessed 15 March 2021].

[24]B. Avants, H. Johnson and N. Tustison, "Neuroinformatics and the The Insight ToolKit", Frontiers in Neuroinformatics, vol. 9, 2015. Available: 10.3389/fninf.2015.00005 [Accessed 01 April 2021].

[25]Zhixuhao, 2019. Available: https://github.com/zhixuhao/unet [Accessed 03 April 2021]

[26]M. D. Zeiler, “ADADELTA: An Adaptive Learning Rate Method,” arXiv:1212.5701 [cs.LG], 2012 [Accessed 10 April 2021]

[27]S. Sra, S. Nowozin and S. Wright, Optimization for machine learning. Cambridge, Mass.: MIT Press, 2012, pp. 351-367.

[28]C. Garbin, X. Zhu and O. Marques, "Dropout vs. batch normalization: an empirical study of their impact to deep learning", Multimedia Tools and Applications, vol. 79, no. 19-20, pp. 12777-12815, 2020. Available: 10.1007/s11042-019-08453-9 [Accessed 20 April 2021].

8. Appendices

8.1. Appendix 1

The information contain on this appendix corresponds to the code used to develop and train the neural networks presented on this report. The code was written on Python 3 using the deep learning framework Keras.

8.1.1. Code for model without pre-processing

Main code:

```
# Import essential libraries

import tensorflow as tf
import nipype
import nipype.interfaces.fsl as fsl
from nipype.interfaces.ants.segmentation import N4BiasFieldCorrection
import nibabel as nib
import matplotlib.pyplot as plt
import os
import numpy as np
from shutil import copyfile, copy
import time
from sklearn.model_selection import train_test_split
import re
import random
import keras

# Import created functions

from ProcessData import process_data
from TrainGen import *
from ValGen import *
from TestGen import *
from SaveSlice import saveSlices
from Stats import *

# Visualize data

scan =
nib.load('/home/faruk/Desktop/BrainSeg/Dataset/ATLAS_R1.1/Site1/031768/t01/0
31768_t1w_deface_stx.nii.gz')
label =
nib.load('/home/faruk/Desktop/BrainSeg/Dataset/ATLAS_R1.1/Site1/031768/t01/0
31768_LesionSmooth_stx.nii.gz')

scan_data = scan.get_fdata()
label_data = label.get_fdata()

print("The shape of the brain scan is:",scan_data.shape)
print("The shape of the segmented label is:",label_data.shape)
```

```
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(10,10))
ax[0].imshow(scan_data[:,:,100]) #row=0, col=0
ax[0].title.set_text('100th slice of a brain scan')
ax[1].imshow(label_data[:,:,100]) #row=0, col=0
ax[1].title.set_text('Truth label')
plt.show()


# Move the data and combine the labels

# The scans contained on the ATLAS dataset contain one or more stroke-
lesions. The segmentation for these...
# lesions are contained on different files. The purpose of the following
function is to combine all of the...
# segmentation labels into one file, and move all of the data to a different
folder.

#!! ONLY RUN ONCE !!
process_data()
#!! ONLY RUN ONCE !!


# Training-validation-test split

images = [file for root,dirs,files in
os.walk('/home/faruk/Desktop/BrainSeg/Dataset/Data/') for file in files]
labels = [file for root,dirs,files in
os.walk('/home/faruk/Desktop/BrainSeg/Dataset/Labels/') for file in files]

images.sort()
labels.sort()

X_train, X_val, y_train, y_val = train_test_split(images, labels,
test_size=0.1, random_state=1)
X_train, X_test, y_train, y_test = train_test_split(X_train, y_train,
test_size=0.1, random_state=1)


# 2D slicing

# The ATLAS dataset consists of volumes. The code presented below, simply
slices the scans present on each...
# data subset on the z-axis, transforming the volumes into 2D images. !!
ONLY RUN ONCE !!

inp = '/home/faruk/Desktop/BrainSeg/Dataset/Data/'
out = '/home/faruk/Desktop/BrainSeg/Dataset/Train/'
for file in X_train:
    saveSlices(inp,file,out)
    time.sleep(0.05)

in_l = '/home/faruk/Desktop/BrainSeg/Dataset/Labels/'
out_l = '/home/faruk/Desktop/BrainSeg/Dataset/Train_labels/'
for file in y_train:
    saveSlices(in_l,file,out_l)
    time.sleep(0.05)

in_vd = '/home/faruk/Desktop/BrainSeg/Dataset/Data/'
out_vd = '/home/faruk/Desktop/BrainSeg/Dataset/Val/'
for file in X_val:
```

```
    saveSlices(in_vd,file,out_vd)
    time.sleep(0.05)

in_vl = '/home/faruk/Desktop/BrainSeg/Dataset/Labels/'
out_vl = '/home/faruk/Desktop/BrainSeg/Dataset/Val_labels/'
for file in y_val:
    saveSlices(in_vl,file,out_vl)
    time.sleep(0.05)

in_td = '/home/faruk/Desktop/BrainSeg/Dataset/Data/'
out_td = '/home/faruk/Desktop/BrainSeg/Dataset/Test/'
for file in X_test:
    saveSlices(in_td,file,out_td)
    time.sleep(0.05)

in_tl = '/home/faruk/Desktop/BrainSeg/Dataset/Labels/'
out_tl = '/home/faruk/Desktop/BrainSeg/Dataset/Test_labels/'
for file in y_test:
    saveSlices(in_tl,file,out_tl)
    time.sleep(0.05)


# Generators

# In order to feed the scans to the neural network, they must be first
loaded into memory. While Keras has...
# pre-made tools for this, the 2D images were stored as .npy arrays, a
format that the pre-built functions of...
# Keras does not support. Furthermore, before feeding the images to the CNN,
the input scan and the label are...
# zero-padded.

train_files = os.listdir('/home/faruk/Desktop/BrainSeg/Dataset/Train/')
val_files = os.listdir('/home/faruk/Desktop/BrainSeg/Dataset/Val/')
test_files = os.listdir('/home/faruk/Desktop/BrainSeg/Dataset/Test/')

train = [item for item in train_files]
val = [item for item in val_files]
test = [item for item in test_files]

train_generator = TrainGen(train)
val_generator = ValGen(val)


# Model

IMAGE_HEIGHT = 224
IMAGE_WIDTH = 256
CHANNELS = 1 # One channel because the image is a grayscale image.
batch_size = 8

### Steps per epoch ###
spe = len(train)//batch_size
val_spe = len(val)//batch_size
test_spe = len(test)//batch_size

### Necessary libraries ###

from keras.models import *
from keras.layers import *
```

```
from keras.optimizers import *
from keras.callbacks import ModelCheckpoint

def unet_modified(input_size = (IMAGE_HEIGHT,IMAGE_WIDTH,CHANNELS)):
    inputs = Input(input_size)
    conv1 = Conv2D(64, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(inputs)
    conv1 = Conv2D(64, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(conv1)
    pool1 = MaxPooling2D(pool_size=(2, 2))(conv1)
    conv2 = Conv2D(128, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(pool1)
    conv2 = Conv2D(128, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(conv2)
    pool2 = MaxPooling2D(pool_size=(2, 2))(conv2)
    conv3 = Conv2D(256, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(pool2)
    conv3 = Conv2D(256, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(conv3)
    pool3 = MaxPooling2D(pool_size=(2, 2))(conv3)
    conv4 = Conv2D(512, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(pool3)
    conv4 = Conv2D(512, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(conv4)
    drop4 = Dropout(0.5)(conv4)
    pool4 = MaxPooling2D(pool_size=(2, 2))(drop4)

    conv5 = Conv2D(1024, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(pool4)
    conv5 = Conv2D(1024, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(conv5)
    drop5 = Dropout(0.5)(conv5)

    up6 = Conv2D(512, 2, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(UpSampling2D(size = (2,2))(drop5))
    merge6 = concatenate([drop4,up6], axis = 3)
    conv6 = Conv2D(512, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(merge6)
    conv6 = Conv2D(512, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(conv6)

    up7 = Conv2D(256, 2, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(UpSampling2D(size = (2,2))(conv6))
    merge7 = concatenate([conv3,up7], axis = 3)
    conv7 = Conv2D(256, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(merge7)
    conv7 = Conv2D(256, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(conv7)

    up8 = Conv2D(128, 2, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(UpSampling2D(size = (2,2))(conv7))
    merge8 = concatenate([conv2,up8], axis = 3)
    conv8 = Conv2D(128, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(merge8)
    conv8 = Conv2D(128, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(conv8)

    up9 = Conv2D(64, 2, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(UpSampling2D(size = (2,2))(conv8))
    merge9 = concatenate([conv1,up9], axis = 3)
```

```
    conv9 = Conv2D(64, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(merge9)
    conv9 = Conv2D(64, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(conv9)
    conv9 = Conv2D(2, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(conv9)
    conv10 = Conv2D(1, 1, activation = 'sigmoid')(conv9)

    model = Model(inputs=inputs, outputs=conv10)

    return model

model = unet_modified()


# The following function shows a summary of the models, including the size
of the image after each layer.

model.summary()


# Compile model

# The following code configures the optimizer of the model, the learning
rate, the loss function and the...
# metrics used to evaluate the model's performance. Furthermore, it
configures a callback that is used to save...
# the model's weights at the end of each epoch.

model.compile(optimizer=SGD(lr=1e-4), loss=EML, metrics=[dice_coef])
output_path = '/home/faruk/Desktop/BrainSeg/Weights/'
checkpointer = ModelCheckpoint(output_path + '20-02-{epoch:02d}-
{dice_coef:.2f}-{val_dice_coef:.2f}.hdf5', verbose=1, save_best_only=False,
save_freq='epoch')
callbacks_list = [checkpointer]


# Train model

# When executed, the code below will start to train the model and save the
data on the History variable.

History=model.fit_generator(generator=train_generator,
validation_data=val_generator, callbacks=callbacks_list, epochs=2,
steps_per_epoch=spe, validation_steps = val_spe, shuffle=False, verbose=1)

# After each epoch, the data saved into the History variable must be stored.

Dict = {'loss' :
[3.4339,2.6715,2.1084,1.7681,1.5294,1.4120,1.3056,1.2428,1.1653,1.1262,1.955
2,1.0357,0.9802,0.9648,0.9562,0.9072,0.8631,0.8642,0.8344,0.8088,0.7808,0.78
89,0.7542,0.7542,0.7446], 'dice_coef' :
[0.0654,0.1152,0.2190,0.2980,0.3573,0.3915,0.4245,0.4450,0.4686,0.4825,0.419
0,0.5097,0.5313,0.5349,0.5412,0.5555,0.5716,0.5719,0.5832,0.5948,0.6030,0.59
89,0.6143,0.6109,0.6184], 'val_loss' :
[3.0621,2.8765,2.1585,1.6189,1.7283,1.6264,1.7349,1.7051,1.7209,1.4431,1.314
5,1.1922,1.4874,1.3743,1.6508,1.3418,1.2328,1.2160,1.1242,1.3093,1.2059,1.22
91,1.1707,1.1660,1.4203], 'val_dice_coef' :
[0.0960,0.1195,0.1964,0.2986,0.2614,0.2997,0.2794,0.3047,0.2787,0.3361,0.380
```

```
6,0.4448,0.3679,0.3839,0.2976,0.3854,0.4238,0.4309,0.4537,0.4432,0.4495,0.43
71,0.4319,0.4606,0.3827]}

fig, ax = plt.subplots(nrows=2, ncols=2, figsize=(13,13))
ax[0,0].plot(Dict['loss'])
ax[0,1].plot(Dict['dice_coef'])
ax[1,0].plot(Dict['val_loss'])
ax[1,1].plot(Dict['val_dice_coef'])

ax[0,0].title.set_text('Training Loss')
ax[0,1].title.set_text('Training Dice coefficient')
ax[1,0].title.set_text('Validation Loss')
ax[1,1].title.set_text('Validation Dice coefficient')


# Predictions

model.load_weights('/home/faruk/Desktop/BrainSeg/Weights/20-02-03-0.62-
0.38.hdf5')
test_gen = TestGen(test)
predictions = model.evaluate_generator(test_gen, steps=test_spe, verbose=1)


# Making a single prediction

model.load_weights('/home/faruk/Desktop/BrainSeg/Weights/20-02-03-0.62-
0.38.hdf5') # Load model weights.
test_gen = TestGen(test) # Configure the test generator.

# The code below feeds a new image into the generator and predicts the
segmentation. If the prediction is...
# less than 0.5, the pixel value is set to 0, if it is greater or equal than
0.5, it is set to 1. Re-run the...
# cell to load a predict a new image. Bear in mind that most of the images
are blank due to class-imbalance.

img,label = next(test_gen)
pred = model.predict(img)[0,:,:,:]
pred = pred >= 0.5

fig, ax = plt.subplots(nrows=1, ncols=3, figsize=(16,16))
ax[0].imshow(img[0,:,:,:])
ax[1].imshow(label[0,:,:,:])
ax[2].imshow(pred)
ax[0].title.set_text('Input image')
ax[1].title.set_text('Truth label')
ax[2].title.set_text('Predicted label')
```

process_data function:

```
import numpy as np
import nibabel as nib
import os
import re
import time
from shutil import copyfile, copy
```

```
def process_data():

    regex = re.compile(r'\d+')

    for root, dirs, files in
os.walk('/home/faruk/Desktop/BrainSeg/Dataset/ATLAS_R1.1/'):
        dirs.sort()
        files.sort()
        for file in files:
            if 'deface' in file:
                if os.path.basename(root) == 't01':
                    a = regex.findall(file)
                    copy(os.path.join(root,file) ,
os.path.join('/home/faruk/Desktop/BrainSeg/Dataset/Data/',
't01_'+a[0]+'_deface.nii.gz'))
                else:
                    a = regex.findall(file)
                    copy(os.path.join(root,file),
os.path.join('/home/faruk/Desktop/BrainSeg/Dataset/Data/',
't02_'+a[0]+'_deface.nii.gz'))

                print('Training Example processed and stored')
                time.sleep(0.2)


    for root, dirs, files in
os.walk('/home/faruk/Desktop/BrainSeg/Dataset/ATLAS_R1.1/'):
        added = 0
        dirs.sort()
        files.sort()
        for file in files:
            if 'Lesion' in file:
                label = nib.load(os.path.join(root,file)).get_data()
                added += label

        if os.path.basename(root) == 't01':
            a = regex.findall(file)
            proc = nib.Nifti1Image(added, np.eye(4))
            nib.save(proc,
os.path.join('/home/faruk/Desktop/BrainSeg/Dataset/Labels/',
't01_'+a[0]+'_label.nii.gz'))

        elif os.path.basename(root) == 't02':
            a = regex.findall(file)
            proc = nib.Nifti1Image(added, np.eye(4))
            nib.save(proc,
os.path.join('/home/faruk/Desktop/BrainSeg/Dataset/Labels/',
't02_'+a[0]+'_label.nii.gz'))

        print('Labels combined and stored')
        time.sleep(0.2)
```

Training generator:

```
import numpy as np
import nibabel as nib
```

```
import os

def weirddivision(n,d):
    return np.array(n)/np.array(d) if d else 0

def TrainGen(file_list, batch_size=8):
    while True:
        np.random.shuffle(file_list)
        for start in range(0,len(file_list),batch_size):
            train_loc = '/home/faruk/Desktop/BrainSeg/Dataset/Train/'
            label_loc = '/home/faruk/Desktop/BrainSeg/Dataset/Train_labels/'
            X = np.empty((batch_size,224,256,1))
            y = np.empty((batch_size,224,256,1))

            end = min(start + batch_size, len(file_list))
            ids_batch = file_list[start:end]

            for i,ID in enumerate(ids_batch):
                x_file_path = os.path.join(train_loc, ID)
                y_file_path = os.path.join(label_loc, ID)

                img = np.load(x_file_path)
                img = np.pad(img, pad_width=((14,13),(12,11)),
mode='constant')
                img = np.expand_dims(img,-1)
                img = weirddivision(img, img.max())

                label = np.load(y_file_path)
                label = np.pad(label, pad_width=((14,13),(12,11)),
mode='constant')
                label = np.expand_dims(label,-1)
                label = weirddivision(label, label.max())


                X[i,] = img
                y[i,] = label

            yield X,y
```

Validation generator:

```
import numpy as np
import nibabel as nib
import os

def weirddivision(n,d):
    return np.array(n)/np.array(d) if d else 0

def ValGen(file_list, batch_size=8):
    while True:
        np.random.shuffle(file_list)
        for start in range(0,len(file_list),batch_size):
            train_loc = '/home/faruk/Desktop/BrainSeg/Dataset/Val/'
            label_loc = '/home/faruk/Desktop/BrainSeg/Dataset/Val_labels/'
```

```
            X = np.empty((batch_size,224,256,1))
            y = np.empty((batch_size,224,256,1))

            end = min(start + batch_size, len(file_list))
            ids_batch = file_list[start:end]

            for i,ID in enumerate(ids_batch):
                x_file_path = os.path.join(train_loc, ID)
                y_file_path = os.path.join(label_loc, ID)

                img = np.load(x_file_path)
                img = np.pad(img, pad_width=((14,13),(12,11)),
mode='constant')
                img = np.expand_dims(img,-1)
                img = weirddivision(img, img.max())

                label = np.load(y_file_path)
                label = np.pad(label, pad_width=((14,13),(12,11)),
mode='constant')
                label = np.expand_dims(label,-1)
                label = weirddivision(label, label.max())


                X[i,] = img
                y[i,] = label

            yield X,y
```

Test generator:

```
import numpy as np
import nibabel as nib
import os

def weirddivision(n,d):
    return np.array(n)/np.array(d) if d else 0

def TestGen(file_list, batch_size=8):
    while True:
        np.random.shuffle(file_list)
        for start in range(0,len(file_list),batch_size):
            train_loc = '/home/faruk/Desktop/BrainSeg/Dataset/Test/'
            label_loc = '/home/faruk/Desktop/BrainSeg/Dataset/Test_labels/'
            X = np.empty((batch_size,224,256,1))
            y = np.empty((batch_size,224,256,1))

            end = min(start + batch_size, len(file_list))
            ids_batch = file_list[start:end]

            for i,ID in enumerate(ids_batch):
                x_file_path = os.path.join(train_loc, ID)
                y_file_path = os.path.join(label_loc, ID)

                img = np.load(x_file_path)
                img = np.pad(img, pad_width=((14,13),(12,11)),
mode='constant')
                img = np.expand_dims(img,-1)
```

```
                img = weirddivision(img, img.max())

                label = np.load(y_file_path)
                label = np.pad(label, pad_width=((14,13),(12,11)),
mode='constant')
                label = np.expand_dims(label,-1)
                label = weirddivision(label, label.max())


                X[i,] = img
                y[i,] = label

            yield X,y
```

saveSlices function:

```
import numpy as np
import re
import nibabel as nib
import os
import time

def saveSlices(dpath,file,outpath):
    regex = re.compile(r'\d+')
    a = regex.findall(file)
    inp = nib.load(os.path.join(dpath,file)).get_fdata()
    (dimx,dimy,dimz) = inp.shape
    for i in range(0,dimz):
        np.save(outpath+'t'+a[0]+'_'+a[1]+'_'+str(i), inp[:,:,i])
        time.sleep(0.02)
        print('Slice saved')
```

Stats functions:

```
from keras import backend as K
import numpy as np
import tensorflow as tf

def TP(y_true, y_pred):
    y_true_f = K.flatten(y_true)
    y_pred_f = K.flatten(y_pred)
    true_positives = K.sum(K.round(K.clip(y_true_f * y_pred_f, 0, 1)))
    return true_positives


def FP(y_true, y_pred):
    y_true_f = K.flatten(y_true)
    y_pred_f = K.flatten(y_pred)
    y_pred_f01 = K.round(K.clip(y_pred_f, 0, 1))
    tp_f01 = K.round(K.clip(y_true_f * y_pred_f, 0, 1))
    false_positives = K.sum(K.round(K.clip(y_pred_f01 - tp_f01, 0, 1)))
```

```
    return false_positives


def TN(y_true, y_pred):
    y_true_f = K.flatten(y_true)
    y_pred_f = K.flatten(y_pred)
    y_pred_f01 = K.round(K.clip(y_pred_f, 0, 1))
    all_one = K.ones_like(y_pred_f01)
    y_pred_f_1 = -1 * (y_pred_f01 - all_one)
    y_true_f_1 = -1 * (y_true_f - all_one)
    true_negatives = K.sum(K.round(K.clip(y_true_f_1 + y_pred_f_1, 0, 1)))
    return true_negatives


def FN(y_true, y_pred):
    y_true_f = K.flatten(y_true)
    y_pred_f = K.flatten(y_pred)
    # y_pred_f01 = keras.round(keras.clip(y_pred_f, 0, 1))
    tp_f01 = K.round(K.clip(y_true_f * y_pred_f, 0, 1))
    false_negatives = K.sum(K.round(K.clip(y_true_f - tp_f01, 0, 1)))
    return false_negatives


def recall(y_true, y_pred):
    tp = TP(y_true, y_pred)
    fn = FN(y_true, y_pred)
    return tp / (tp + fn)


def precision(y_true, y_pred):
    tp = TP(y_true, y_pred)
    fp = FP(y_true, y_pred)
    return tp / (tp + fp)


def dice_coef(y_true, y_pred):
    smooth = 1.
    y_true_f = K.flatten(y_true)
    y_pred_f = K.flatten(y_pred)
    intersection = K.sum(y_true_f * y_pred_f)
    return (2. * intersection + smooth) / (K.sum(y_true_f * y_true_f) +
K.sum(y_pred_f * y_pred_f) + smooth)

def EML(y_true, y_pred):
    gamma = 1.1
    alpha = 0.48
    smooth = 1.
    y_true = K.flatten(y_true)
    y_pred = K.flatten(y_pred)
    intersection = K.sum(y_true*y_pred)
    dice_loss = (2.*intersection +
smooth)/(K.sum(y_true*y_true)+K.sum(y_pred * y_pred)+smooth)
    y_pred = K.clip(y_pred, K.epsilon(),None)
    pt_1 = tf.where(tf.equal(y_true, 1),y_pred,tf.ones_like(y_pred))
    pt_0 = tf.where(tf.equal(y_true, 0),y_pred,tf.zeros_like(y_pred))
    focal_loss = -K.mean(alpha*K.pow(1. -pt_1, gamma)*K.log(pt_1),axis=-1)\
                 -K.mean((1-alpha)*K.pow(pt_0,gamma)*K.log(1. -
pt_0),axis=-1)
    return focal_loss - K.log(dice_loss)
```

8.1.2. Code for model with data pre-processing

The functions used on both models are identical (Generators, stats, etc.), the only difference being the path of the folders. Hence, the only piece of code that will be provided for the pre-processed model is the main code, i.e. excluding the functions.

Main code:

```
# Import essential libraries

import tensorflow as tf
import nipype
import nipype.interfaces.fsl as fsl
from nipype.interfaces.ants.segmentation import N4BiasFieldCorrection
import nibabel as nib
import matplotlib.pyplot as plt
import os
import numpy as np
import time
from sklearn.model_selection import train_test_split
import re
import random
import keras

# Import created functions

from ProcessData import process_data
from TrainGenProc import *
from ValGenProc import *
from SaveSlice import saveSlices
from Stats import *
from TestGenProc import *


# Data pre-processing

# Perform Registration using FLIRT and standard_MNI152 as reference image

regex = re.compile(r'\d+')

for root, dirs, files in
os.walk('/home/faruk/Desktop/BrainSeg/Dataset/Data/'):
        files.sort()
        for file in files:
            a = regex.findall(file)
            myflirt =
fsl.FLIRT(in_file=os.path.join('/home/faruk/Desktop/BrainSeg/Dataset/Data/',
file),
reference='/home/faruk/Desktop/BrainSeg/Dataset/ATLAS_R1.1/standard_mni152.n
ii.gz',
out_file=os.path.join('/home/faruk/Desktop/BrainSeg/Dataset/Processed/FLIRT/
'+'t'+a[0]+'_'+a[1]+'_FLIRT.nii.gz'), bins=256, cost='corratio',
searchr_x=[0,0], searchr_y=[0,0], searchr_z=[0,0], dof=12,
interp='trilinear')
            result = myflirt.run()
```

```
            print('Training Example processed and stored')
            time.sleep(0.2)


# Perform Skull stripping using BET

regex = re.compile(r'\d+')

for root, dirs, files in
os.walk('/home/faruk/Desktop/BrainSeg/Dataset/Processed/FLIRT'):
        files.sort()
        for file in files:
            a = regex.findall(file)
            mybet =
fsl.BET(in_file=os.path.join('/home/faruk/Desktop/BrainSeg/Dataset/Processed
/FLIRT',file),
out_file=os.path.join('/home/faruk/Desktop/BrainSeg/Dataset/Processed/BET/'+
't'+a[0]+'_'+a[1]+'_BET.nii.gz'), frac=0.5)
            result = mybet.run()
            print('Training Example processed and stored')
            time.sleep(0.2)


# Bias Field Correction

regex = re.compile(r'\d+')

for root, dirs, files in
os.walk('/home/faruk/Desktop/BrainSeg/Dataset/Processed/BET'):
        files.sort()
        for file in files:
            a = regex.findall(file)
            n4 = N4BiasFieldCorrection()
            n4.inputs.input_image =
os.path.join('/home/faruk/Desktop/BrainSeg/Dataset/Processed/BET',file)
            n4.inputs.output_image =
os.path.join('/home/faruk/Desktop/BrainSeg/Dataset/Processed/N4/'+'t'+a[0]+'
_'+a[1]+'_N4.nii.gz')
            n4.inputs.dimension = 3

            n4.inputs.n_iterations = [100, 100, 60, 40]
            n4.inputs.shrink_factor = 3
            n4.inputs.convergence_threshold = 1e-4
            n4.inputs.bspline_fitting_distance = 300
            result = n4.run()
            print('Training Example processed and stored')
            time.sleep(0.2)


# Training-validation-test split

images = [file for root,dirs,files in
os.walk('/home/faruk/Desktop/BrainSeg/Dataset/Processed/N4/') for file in
files]
labels = [file for root,dirs,files in
os.walk('/home/faruk/Desktop/BrainSeg/Dataset/Processed/Labels/') for file in
files]

images.sort()
```

```
labels.sort()

X_train, X_val, y_train, y_val = train_test_split(images, labels,
test_size=0.1, random_state=1)
X_train, X_test, y_train, y_test = train_test_split(X_train, y_train,
test_size=0.1, random_state=1)


# 2D slicing

# The ATLAS dataset consists of volumes. The code presented below, simply
slices the scans present on each...
# data subset on the z-axis, transforming the volumes into 2D images. !!
ONLY RUN ONCE !!

inp = '/home/faruk/Desktop/BrainSeg/Dataset/Processed/N4/'
out = '/home/faruk/Desktop/BrainSeg/Dataset/Processed/Train/'
for file in X_train:
    saveSlices(inp,file,out)
    time.sleep(0.05)

in_l = '/home/faruk/Desktop/BrainSeg/Dataset/Processed/Labels/'
out_l = '/home/faruk/Desktop/BrainSeg/Dataset/Processed/Train_labels/'
for file in y_train:
    saveSlices(in_l,file,out_l)
    time.sleep(0.05)

in_vd = '/home/faruk/Desktop/BrainSeg/Dataset/Processed/N4/'
out_vd = '/home/faruk/Desktop/BrainSeg/Dataset/Processed/Val/'
for file in X_val:
    saveSlices(in_vd,file,out_vd)
    time.sleep(0.05)

in_vl = '/home/faruk/Desktop/BrainSeg/Dataset/Processed/Labels/'
out_vl = '/home/faruk/Desktop/BrainSeg/Dataset/Processed/Val_labels/'
for file in y_val:
    saveSlices(in_vl,file,out_vl)
    time.sleep(0.05)

in_td = '/home/faruk/Desktop/BrainSeg/Dataset/Processed/N4/'
out_td = '/home/faruk/Desktop/BrainSeg/Dataset/Processed/Test/'
for file in X_test:
    saveSlices(in_td,file,out_td)
    time.sleep(0.05)

in_tl = '/home/faruk/Desktop/BrainSeg/Dataset/Processed/Labels/'
out_tl = '/home/faruk/Desktop/BrainSeg/Dataset/Processed/Test_labels/'
for file in y_test:
    saveSlices(in_tl,file,out_tl)
    time.sleep(0.05)


# Generators

# In order to feed the scans to the neural network, they must be first
loaded into memory. While Keras has...
# pre-made tools for this, the 2D images were stored as .npy arrays, a
format that the pre-built functions of...
# Keras does not support. Furthermore, before feeding the images to the CNN,
the input scan and the label are...
```

```
# zero-padded.

train_files =
os.listdir('/home/faruk/Desktop/BrainSeg/Dataset/Processed/Train/')
val_files =
os.listdir('/home/faruk/Desktop/BrainSeg/Dataset/Processed/Val/')
test_files =
os.listdir('/home/faruk/Desktop/BrainSeg/Dataset/Processed/Test/')

train = [item for item in train_files]
val = [item for item in val_files]
test = [item for item in test_files]

train_generator = TrainGen(train)
val_generator = ValGen(val)


# Model

IMAGE_HEIGHT = 224
IMAGE_WIDTH = 256
CHANNELS = 1
batch_size = 8
spe = len(train)//batch_size
val_spe = len(val)//batch_size
test_spe = len(test)//batch_size

from keras.models import *
from keras.layers import *
from keras.optimizers import *
from keras.callbacks import ModelCheckpoint

def unet_modified(input_size = (IMAGE_HEIGHT,IMAGE_WIDTH,CHANNELS)):
    inputs = Input(input_size)
    conv1 = Conv2D(64, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(inputs)
    conv1 = Conv2D(64, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(conv1)
    pool1 = MaxPooling2D(pool_size=(2, 2))(conv1)
    conv2 = Conv2D(128, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(pool1)
    conv2 = Conv2D(128, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(conv2)
    pool2 = MaxPooling2D(pool_size=(2, 2))(conv2)
    conv3 = Conv2D(256, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(pool2)
    conv3 = Conv2D(256, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(conv3)
    pool3 = MaxPooling2D(pool_size=(2, 2))(conv3)
    conv4 = Conv2D(512, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(pool3)
    conv4 = Conv2D(512, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(conv4)
    drop4 = Dropout(0.5)(conv4)
    pool4 = MaxPooling2D(pool_size=(2, 2))(drop4)

    conv5 = Conv2D(1024, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(pool4)
    conv5 = Conv2D(1024, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(conv5)
```

```
    drop5 = Dropout(0.5)(conv5)

    up6 = Conv2D(512, 2, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(UpSampling2D(size = (2,2))(drop5))
    merge6 = concatenate([drop4,up6], axis = 3)
    conv6 = Conv2D(512, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(merge6)
    conv6 = Conv2D(512, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(conv6)

    up7 = Conv2D(256, 2, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(UpSampling2D(size = (2,2))(conv6))
    merge7 = concatenate([conv3,up7], axis = 3)
    conv7 = Conv2D(256, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(merge7)
    conv7 = Conv2D(256, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(conv7)

    up8 = Conv2D(128, 2, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(UpSampling2D(size = (2,2))(conv7))
    merge8 = concatenate([conv2,up8], axis = 3)
    conv8 = Conv2D(128, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(merge8)
    conv8 = Conv2D(128, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(conv8)

    up9 = Conv2D(64, 2, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(UpSampling2D(size = (2,2))(conv8))
    merge9 = concatenate([conv1,up9], axis = 3)
    conv9 = Conv2D(64, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(merge9)
    conv9 = Conv2D(64, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(conv9)
    conv9 = Conv2D(2, 3, activation = 'relu', padding = 'same',
kernel_initializer = 'he_normal')(conv9)
    conv10 = Conv2D(1, 1, activation = 'sigmoid')(conv9)

    model = Model(inputs=inputs, outputs=conv10)

    return model

model = unet_modified()

# The following function shows a summary of the models, including the size
of the image after each layer.

model.summary()


# Compile model

# The following code configures the optimizer of the model, the learning
rate, the loss function and the...
# metrics used to evaluate the model's performance. Furthermore, it
configures a callback that is used to save...
# the model's weights at the end of each epoch.

model.compile(optimizer=SGD(lr=1e-4), loss=EML, metrics=[dice_coef])
output_path = '/home/faruk/Desktop/BrainSeg/Dataset/Processed/Weights/'
```

```
checkpointer = ModelCheckpoint(output_path + '20-04-{epoch:02d}-
{dice_coef:.2f}-{val_dice_coef:.2f}.hdf5', verbose=1, save_best_only=False,
save_freq='epoch')
callbacks_list = [checkpointer]


# Train model

# When executed, the code below will start to train the model and save the
data on the History variable.

History=model.fit_generator(generator=train_generator,
validation_data=val_generator, callbacks=callbacks_list, epochs=2,
steps_per_epoch=spe, validation_steps = val_spe, shuffle=False, verbose=1)

Dict = {'loss' :
[3.3674,2.4782,2.1303,1.9281,1.7422,1.6221,1.5415,1.4681,1.4183,1.3716,1.315
8,1.2607,1.2375,1.1914,1.1596,1.1415,1.1153,1.0910,1.0556,1.0561,1.0333,0.97
32,0.9649,0.9336,0.9497], 'dice_coef' :
[0.0750,0.1433,0.2009,0.2409,0.2802,0.3079,0.3310,0.3492,0.3654,0.3774,0.396
1,0.4127,0.4206,0.4353,0.4447,0.4537,0.4594,0.4684,0.4795,0.4825,0.4912,0.50
69,0.5110,0.5224,0.5196], 'val_loss' :
[3.0805,2.7149,2.3701,2.3200,2.4102,2.1967,2.2649,1.9792,2.3031,1.6587,1.781
5,1.9047,1.4084,1.7995,1.4633,1.6900,1.4329,1.5346,1.2494,1.3830,1.3635,1.49
39,1.3226,1.2955,1.3633], 'val_dice_coef' :
[0.0883,0.0931,0.1189,0.1262,0.1280,0.1596,0.1740,0.2195,0.1929,0.2725,0.263
1,0.2560,0.3413,0.2743,0.3211,0.3185,0.3349,0.3194,0.3840,0.3800,0.3761,0.35
71,0.3940,0.4,0.3799]}
fig, ax = plt.subplots(nrows=2, ncols=2, figsize=(13,13))
ax[0,0].plot(Dict['loss'])
ax[0,1].plot(Dict['dice_coef'])
ax[1,0].plot(Dict['val_loss'])
ax[1,1].plot(Dict['val_dice_coef'])

ax[0,0].title.set_text('Training Loss')
ax[0,1].title.set_text('Training Dice coefficient')
ax[1,0].title.set_text('Validation Loss')
ax[1,1].title.set_text('Validation Dice coefficient')


# Predictions

model.load_weights('/home/faruk/Desktop/BrainSeg/Dataset/Processed/Weights/2
0-04-02-0.52-0.38.hdf5')
test_gen = TestGen(test)
predictions = model.evaluate_generator(test_gen, steps=test_spe, verbose=1)


# Make a single prediction

model.load_weights('/home/faruk/Desktop/BrainSeg/Dataset/Processed/Weights/2
0-04-02-0.52-0.38.hdf5') # Load model weights.
test_gen = TestGen(test) # Configure the test generator.

# The code below feeds a new image into the generator and predicts the
segmentation. If the prediction is...
# less than 0.5, the pixel value is set to 0, if it is greater or equal than
0.5, it is set to 1. Re-run the...
# cell to load a predict a new image. Bear in mind that most of the images
are blank due to class-imbalance.
```

```
img,label = next(test_gen)
pred = model.predict(img)[0,:,:,:]
pred = pred >= 0.5

fig, ax = plt.subplots(nrows=1, ncols=3, figsize=(16,16))
ax[0].imshow(img[0,:,:,:])
ax[1].imshow(label[0,:,:,:])
ax[2].imshow(pred)
ax[0].title.set_text('Input image')
ax[1].title.set_text('Truth label')
ax[2].title.set_text('Prediction')
```