Universitat Oberta de Catalunya (UOC)

Máster Universitario en Ciencia de Datos (*Data Science*)

TRABAJO FINAL DE MÁSTER

Área: 3 – Medicina

Reconstructing and Super–resolving Brain Magnetic Resonance Images Using Deep Learning

---

Autor: Francisco Enrique Lorente Banegas

Tutor: Baris Kanber

Profesor: Ferran Prados Carrasco

Barcelona, June 5, 2021

Créditos/Copyright

FICHA DEL TRABAJO FINAL

Título del trabajo:	Reconstructing and Super–resolving Brain Magnetic Resonance Images Using Deep Learning
Nombre del autor:	Francisco Enrique Lorente Banegas
Nombre del colaborador/a docente:	Baris Kanber
Nombre del PRA:	Ferran Prados Carrasco
Fecha de entrega (mm/aaaa):	06/2021
Titulación o programa:	Máster Universitario en Ciencia de Datos
Área del Trabajo Final:	TFM – Área 3
Idioma del trabajo:	Inglés
Palabras clave	Deep Learning · MRI · Inpainting · Super–resolution

Abstract

In the area of MRI analysis related to the identification of anomalies (lesions or artifacts) and the quality of anomalies inpainting, many studies focus in one of these two aspects, yet few include both. The efficiency of the training process (performance vs processing cost) is also often neglected, however, increased efficiency could lead to faster and more frequent trainings (e.g., by including new data) or allow its adequate execution with lower resource requirements such as low-end GPUs.

This project aims at analysing and choosing one of the existing models for image–related problems, adapting it to the challenge of lesion/artifacts detection and image inpainting. The training process is performed with different image sizes and various computation systems (CPU, integrated GPU, discrete GPU) enabling a comparison in terms of performance (accuracy of the reconstructed image) and computational cost/time.

Additional objectives of this project are the training and assessment of the model to address additional challenges such as super–resolution and noise reduction; as well as the impact in its performance when using data augmentation techniques.

Keywords: Deep Learning, MRI, Image Inpainting, Super–resolution, Efficiency

Contents

List of Figures

List of Tables

List of Code Listings

Chapter 1

Introduction

1.1 Problem statement

Medical imaging is an essential tool for diagnosis. In particular, magnetic resonance imaging (MRI), since it does not emit the ionizing radiation that can be found in X–ray imaging. The complexity of the acquisition and the amount of data it provides has triggered the creation of numerous pipelines to process and exploit those data. These pipelines however, are not always prepared to process incomplete or partially corrupt data (e.g. empty areas of the image, or artifacts) and might fail.

Reconstructing an incomplete or erroneous MRI is a twofold problem: firstly identifying what are the empty or corrupt areas in the image; secondly reconstructing these empty or corrupt areas. The latter should be of quality enough to be correctly processed by the above-mentioned pipelines; or even indistinguishable from real images. Although many of the studies in this field tackle these problems, they usually focus on one of them and overlook the computational cost of training, which could redirect the research attention to increase their efficiency and potentially extending its use in developing countries where access to state–of–the–art MRI scanners or high-end computer systems is limited.

1.2 Personal motivation

With a professional background developed around data, including data extraction, transformation and loading, reporting, modeling and architecture, never had the chance to deep dive into advanced analytics. That is why this Master, and more specifically this Master's Thesis, poses a great opportunity for further career evolution, helping to uncover the details of deep learning algorithms and how its use in medicine is improving human health.

1.3 Objectives

This project aims at analysing and choosing one of the existing models for image–related problems, adapting it to the challenge of lesion/artifacts detection and image inpainting. The training process is performed with different image sizes and various computation systems (CPU, integrated GPU, discrete GPU) enabling a comparison in terms of performance (accuracy of the reconstructed image) and computational cost/time.

1.3.1 Image inpainting

Use of images with random blank areas as input so the model can regenerate the complete image using the contextual information. These blank areas can be later on defined by a lesion/artifact detection model.

1.3.2 Lesion/artifact detection

Identification of lesions and artifacts by calculating the difference between the original image (containing artifacts or lesions) and the reconstructed image produced by the image inpainting process.

1.3.3 Performance and computational efficiency

Capture the training time of the different models under different configurations, such as:

- CPU (Intel i7 6 cores) with 16 GB memory
- Integrated GPU (Intel UHD 630) with system–shared memory
- Discrete GPU (AMD Radeon 5300M) with 4 GB dedicated memory

1.3.4 Other potential objectives

Additional objectives of this project are the training and assessment of the model to address additional challenges such as image resolution increase (super–resolution), noise reduction; and the impact in its performance when using data augmentation techniques.

1.4 Methodology

The research process, based on (15), starts with the **Literature review** which helps building the **Conceptual framework** and identifying the **Research questions**. These two activities

define the strategies to be followed, in this case, **Case study** for the analysis of the existing algorithms and **Design and creation** for the potential tuning of those in order to improve performance or efficiency. **Data generation** will be produced through the **Observations** of the models executions and finally, the **Data analysis** will be mostly **Quantitative** since data produced will be measurable and comparable.

The algorithms are be based on Keras (41), the high-level API of TensorFlow (8), and developed in Python 3.8. These algorithms will be trained and tested uisng the IXI Dataset (13), which contains approximately 600 MR images from healthy subjects in Neuroimaging Informatics Technology Initiative (NIFTI) (1) format. The system used for development, training and evaluation of the models is a 6-core 2.6 GHz Intel Core i7 with 16 GB of RAM memory, an Intel UHD Graphics 630 embedded graphics card with up to 1.5 GB of shared memory, and an AMD Radeon Pro 5300M graphics card with 4 GB of dedicated memory; running macOS Big Sur.

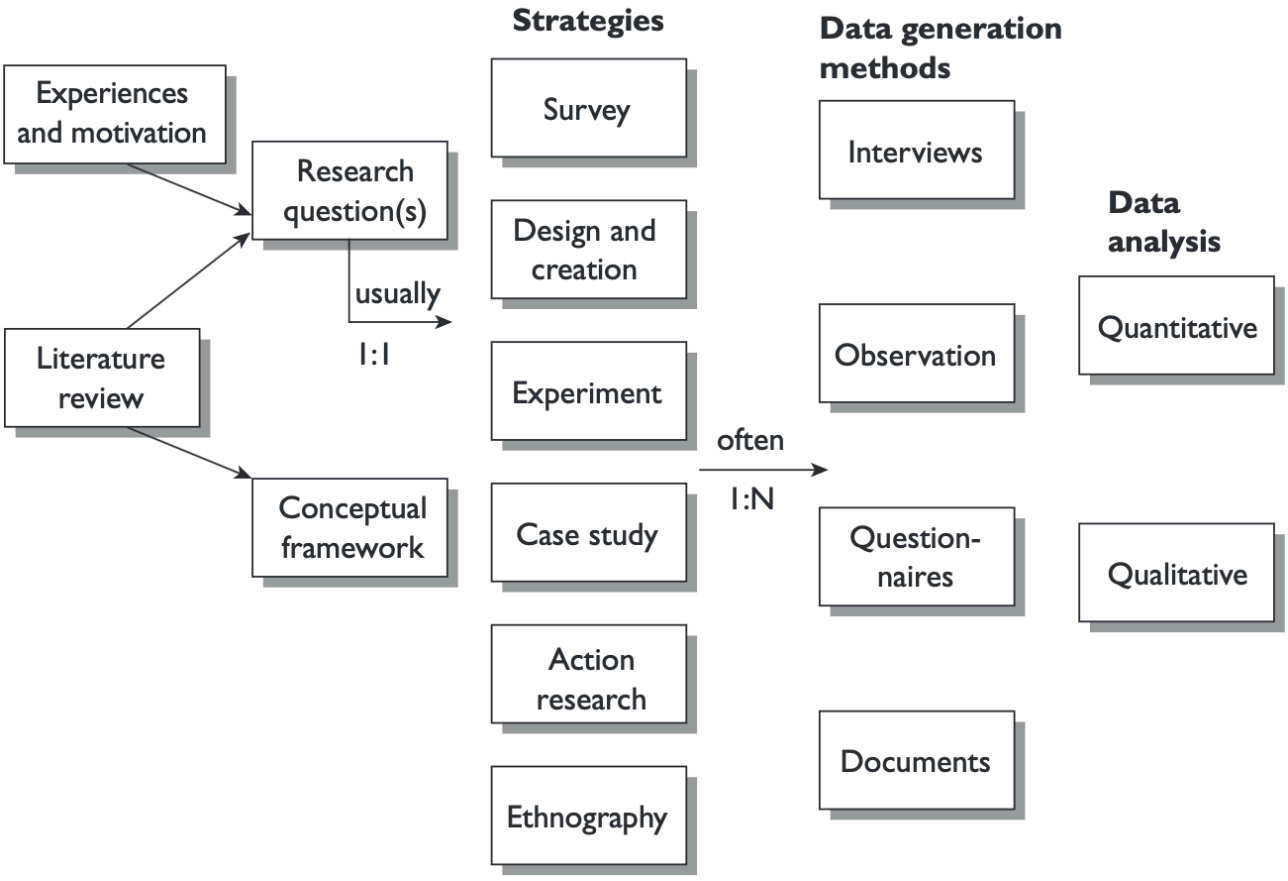

Figure 1.1: Model of the research process (15)

1.5 Plan

The table 1.1 summarises the high-level plan based on the objectives described in section 1.3.

Work areas	F	M	A	M	J
Problem statement					
Literature review					
Analysis					
Case studies					
Design and creation					
Reporting on results					
Master's Thesis writing					
Evaluation					

Table 1.1: Master's Thesis high-level plan

Chapter 2

State of the Art

The literature review carried out in this chapter is based on a selection of relevant publications from the recent years. These publications mainly tackle different approaches to perform lesion or anomaly detection and image inpainting, except one of those, consisting in a benchmark of different techniques for brain MRI reconstruction. The historical background provides a summary of the problems these works aimed at addressing; current techniques describe how they do it, with specific mentions to the main neural networks used and other peculiarities such as lesion detection, which is not always a trivial activity. Finally, some areas for further development are identified, which could be partially covered by this thesis.

2.1 Historical background

The use of MRI in disease diagnosis and treatment has increased over the years, as it does not emit the ionizing radiation that can be found, amongst others, in X-Ray imaging. The acquisition process however is sometimes challenging: depending on the area to be covered, it can take up to 90 minutes (4) in which the patient must remain still. This is not only uncomfortable, but impossible when the patient is a baby or has nervous condition such as Parkinson's disease; in these cases anaesthesia might be required.

Image inpainting might help improve the MRI process, by reducing the acquisition and reconstruction time (with the same image quality) and its robustness, by handling motion artifacts or noise, reducing the need of repeated scans. The computational cost of such processes was high and results were not always realistic, until deep learning techniques were used and efficiently implemented in GPUs.

Artifact or lesion detection in MRI has been and remains a challenging activity, relying first

on manual identification and annotation of such artifacts or lesions by human operators – more or less experienced depending on the specific task –, and later on supervised machine learning algorithms. The latter is also time-consuming: these algorithms need labeled data (e.g. MRI scans with anomalous areas identified) to learn. The increasing number of patients making use of MRI has not been followed by a sufficient amount of experts to process them, increasing the need for improving the performance of the process.

2.2 Current techniques

Most of the existing techniques either focus on image inpainting or in artifact/lesion detection, but few do on both, and for none of the analysed work learning time is an area of research. All of them use deep neural networks in form of autoencoders or GANs. In some cases, an additional layer for segmentation purposes is included. The basis of the work is usually similar: building a representation of a healthy (normal) brain (or any other body part) and then feed that representation with data not fitting it.

2.2.1 VAE and GAN

The algorithm proposed in (37), even though not related to brain MRI, followed the approach mentioned above, and it was the first to use GANs for anomaly detection and to include an anomaly score expressing the fit of an input image to the model of normal images. This algorithm was identified as competitor in (17) where the focus was on the investigation of variational autoencoders (VAE) and adversarial autoencoders (AAE) to imitate human's ability to detect abnormal-looking areas using prior information on healthy-looking tissues, as they were able to detect those areas accurately even without extensive training.

Both VAEs and GANs aim at learning the data distribution of the dataset used for training and then being able to generate new data. The main difference between them is that VAE learns the hidden representation of the data (latent space) with the encoder and its likelihood with the decoder; whilst GAN trains a generator to create data samples that a classifier (discriminator) cannot distinguish from real ones. Good results were produced with AAE beating VAE, even though the attempts to train the competing AnoGAN (37) were not successful (issues with gradient descent). Also, since the focus was on anomaly detection and not in image generation, the selected image size was only 32x32, clearly insufficient for image inpainting purposes.

Other work has followed both GAN and VAE approaches. Using MedGAN (10) – a framework for medical image translation tasks using a GAN –, ip-MedGAN (11) expands it to

perform inpainting tasks, where missing areas need to be realistic and fit in the overall picture. To achieve this, the initial architecture is expanded with a local discriminator network. The results improve pre-existing methods such as the original MedGAN, context encoders or GLCIC (Globally and Locally Consistent Image Completion). (46) also supports the high performance of GAN for the creation of realistic synthetic images –in this case, by adding noise–, so that the differences with a real ones are imperceptible.

On the VAE approach, (9) proposes extending it with transfer learning, storing the knowledge acquired with the training dataset and being able to apply it to a different problem (e.g. different demographics or different pre-processing techniques); and replacing KL–divergence with β–divergence as the latter deals better with outliers, increasing its robustness. Transfer learning allowed updating a pre-trained model with new data (without needing the data used to pre-train the model).

2.2.2 Lesion detection

It is worth mentioning that lesion detection is usually not a trivial task: although, as explained in (17), some anomalies are easily detectable, other require expertise in neuroanatomy, such as white matter hyperintensities (44), since they show as areas with increased intensity in MRIs, that are not always consistent in scans and there is high variability in the areas affected. For this specific problem, (12) propose an algorithm that accurately detects these areas by using T1, T2–weighted and FLAIR images of MRI and providing them as input in form of 3-dimension patches, where, the network has a softmax layer to segment between healthy and unhealthy tissue.

2.2.3 Performance metrics and computational cost

The main metrics used for measuring the performance of the models are the area under ROC curve (AUC) (6) in the case of anomaly detection, and Structural Similarity Index (SSIM) (43) and/or Peak Signal to Noise Ratio (PSNR) for image inpainting, amongst others.

Although some of the work analysed describes the training parameters of the model and the some of the components of the system used (normally the GPU) in general, it scarcely elaborates on the computational cost of training the models proposed. Only (11) mentions a training time of 24 hours with a specific configuration; and a reconstruction time of 8 seconds on CPU and 0.5 seconds on GPU. Benchmarks performed in (35) covers the reconstruction times of some algorithms (Zero–filled, KIKI–net, U–net, Cascade–net and PD–net) and their performance,

highlighting the improvement that deep learning –efficiently implemented in GPUs– represents compared to classic algorithms. However, no comparison of training times is presented.

2.3 Areas for further development

Some of the areas for development in the studies presented are partially covered by other studies, mainly the location of missing or anomalous areas to be inpainted in (11) are covered by (17), (33) or (37).

Other areas however, remain open: some of the methods for image inpainting, such as (33) have not been tested with other lesions than multiple sclerosis. Vascular lesions, brain tumors or even artifacts not related to any disease are examples of a potential extension. In (46), its potential use for data augmentation or image imputation has not been challenged. In (12), even though more specific to white matter hyperintensities, the model could be extended to identify abnormalities also the grey matter, which is relevant for improving the understanding of pathophysiology of many neurologic conditions (16).

Finally, as presented in 2.2.3, training time is not part of the work analysed (except (33)), although it could be relevant for some decision making: one example is assessing the cost/benefit of an algorithm depending on its training requirements and decide accordingly the training frequency with new data; another one is being able to perform such trainings in low-end computer systems: in developing countries, the availability of high-end ones might be limited.

Chapter 3

Theory

This chapter describes the theoretical basis of the work, starting from the most general concepts such as magnetic resonance imaging or image inpainting, denoising, super–resolution and its relationship with MRI. Then moves towards more data science–specific ones, such as machine learning or artificial neural networks. Finally the most specific elements are presented: U-net network, hyperparameters, performance metrics or data augmentation.

3.1 Magnetic resonance imaging

Magnetic resonance imaging (MRI) is a medical imaging method that allows the visualization of internal structures and some functions of the body, providing better results than X–rays or computed tomography and without using ionizing radiation (28).

It operates by means of a powerful magnetic field that aligns the nuclear magnetization of protons of hydrogen atoms in the body. Since the human body is made up of about 70 percent water, and water molecules are made up of two hydrogen atoms and one oxygen atom, when exposed to that powerful magnetic field, some of the protons in the nuclei of the hydrogen atoms align with the direction of the field. A radio frequency alters the alignment of the protons which is detected by the scanner. The scanner processes these changes and creates sets of cross–sectional images (slices). The design of the pulse sequences determines the image contrast, creating T1 or T2–weighted images (32).

The set of slices captured can be visualized in three different anatomical planes:

- **Axial**: a horizontal plane separating the volume in lower and upper parts
- **Sagittal**: a vertical plane separating the volume in left and right part

- **Coronal**: a vertical plane separating the volume in front and back part

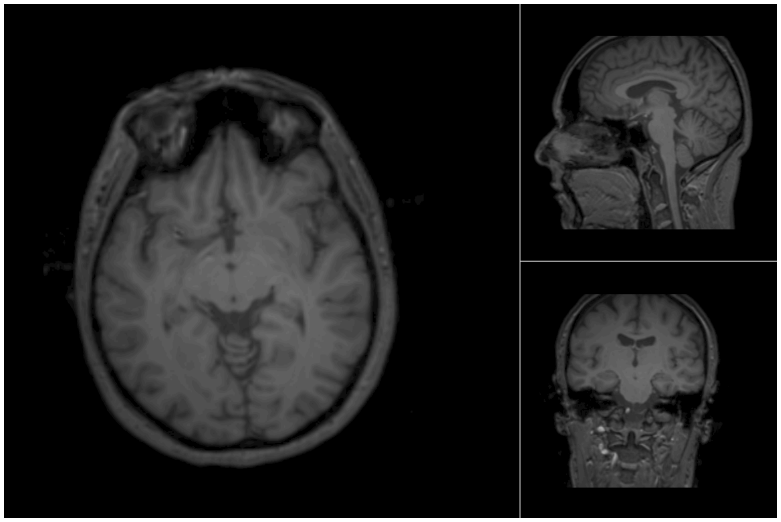

Figure 3.1: Anatomical planes: axial (left), sagittal (upper right), coronal (lower right). Picture extracted from (13) using (29)

3.2 Image inpainting

Image inpainting is the process of filling–in the missing or deteriorated areas of an image, with the aim of making the completed or restored image visually realistic and following the original context.

This process has numerous applications such as the removal or replacement of unwanted objects, or the restoration of deteriorated images.

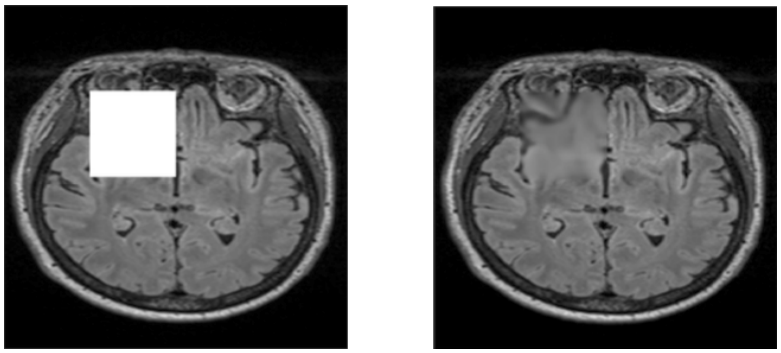

Figure 3.2: Image inpainting: original image (left) with missing area and completed (inpainted) image (right) (11).

3.3 Image denoising

Image denoising is the process of removing distortions or noise from an image. Noise is a random variation of brightness or color information in an image.

There are multiple types of noises such as Gaussian noise: a statistical noise having a probability density function equal to the Normal distribution; Salt and Pepper noise: an impulse noise with a fixed value; or Poisson noise: with a probability density function of a Poisson distribution. The following ones, however, are more specific to magnetic resonance imaging:

3.3.1 Gibb's ringing noise

Gibb's artifacts (or ringing) are the consequence of using Fourier transformations to reconstruct magnetic resonance signals into images (21). In magnetic resonance imaging, a finite number of frequencies can be used, hence the image must be reconstructed using only few armonics in its Fourier representation. These artifacts appear as multiple parallel lines immediately adjacent to high–contrast zones.

3.3.2 Ghosting noise

Ghosting is a type of noise appearing as repeated versions of the main image, blurred and shifted. They are usually caused by physical motion during image acquisition, including patient motion, blood flow or respiratory and cardiac effects (20).

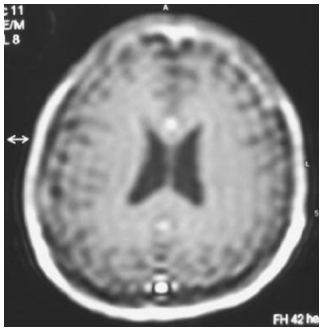

Figure 3.3: Gibb's ringing noise (21)

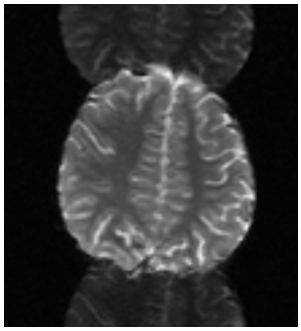

Figure 3.4: Ghosting noise (20)

3.4 Super–resolution

Image super–resolution is the task of increasing the resolution of an image. Amongst other, it is especially relevant for medical applications, e.g. by converting low–resolution MRI images

(that require less scan time) to high–resolution MRI versions.

Figure 3.5: Super–resolution example: low–resolution image (left), super–resolved image (right)

3.5 Machine learning and deep learning

According to IBM (19), *"Machine learning is a branch of artificial intelligence and computer science which focuses on the use of data and algorithms to imitate teh way that humans learn, gradually improving its accuracy"*. Using statistical methods, algorithms are trained to make classifications or predictions.

Deep learning can be considered as a sub–field of machine learning. The main difference relies in how each algorithm learns: in the case of deep learning, much of the feature extraction part of the work is automated, easing the use of large datasets for which human intervention would probably be too expensive. Finally, neural networks can be considered a sub–field of deep learning.

3.5.1 Artificial neural networks

Artificial neural networks (ANNs) are machine learning models inspired by the networks of biological neurons found in our brains. They can be considered the core of deep learning, as they are versatile, powerful and scalable, hence able to tackle large and complex machine learning tasks.

They were introduced in 1943 by the neurophysiologist Warren McCulloh and the mathematician Walter Pitts using a simplified computational model of how neurons might work together to perform complex operations.

The interest in neural networks has risen and fallen during the past years, where new approaches were explored, although the available data and computational power could not follow. In the recent days however, this is no longer the case: there is a huge amount of data ready to be used by neural networks and the computing power has increased exponentially over the years, especially with the use of GPU cards, which are extremely efficient in addressing these kind of problems.

In general, artificial neural networks are made of node layers. Each node layer contain an input layer, one or more hidden layers and an output layer. Each node is connected to another, and has a weight and threshold. When the output of an individual node is above the defined threshold, the node is activated, sending data to the next layer of the network; otherwise no data passes.

The neural network learns by processing tuples of data, consisting in an input and a known result, creating probability–weighted associations between the two and calculating the difference between the generated output and the expected output (error). The weights of the parameters are adjusted a number of times according to a learning rule with the objective of minimizing the error. The training can be concluded when it produces the expected results or when no further improvements are made.

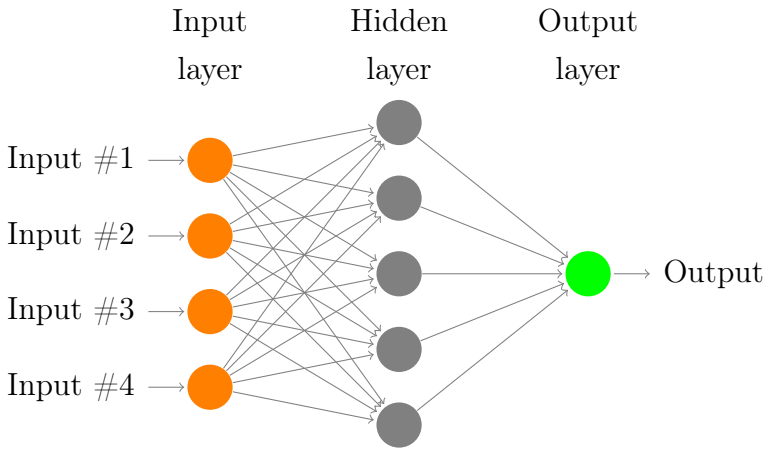

Figure 3.6: Example of artificial neural network

3.5.2 Convolutional neural networks

Convolutional neural networks (CNNs) emerged from the study of the brain's visual cortex, where many neurons react only to visual stimuli located in a limited region of the visual field (23), with specialized behavior: some react to horizontal lines, other to lines with different

orientations. Moreover, some have larger receptive fields, meaning that they react to more complex patterns that are combination of lower–level patterns. This approach is common in real images, making CNNs good for image recognition tasks.

A CNN is usually composed by three types of layers:

- **Convolutional layer**: neurons in the first convolutional layer are connected only to the pixels of the input image belonging to their receptive fields; subsequently, neurons in the second convolutional layer are connected only to neurons located within a small area of the first layer. In this manner, every new convolutional layer concentrates into higher–level features, starting with the lowest level in the first layer.

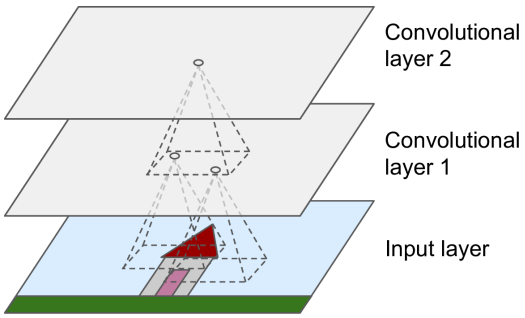

Figure 3.7: CNN layers with local receptive fields (23)

- **Pooling layer**: their objective is to downsample the input data (e.g. an image) to reduce the computational cost (processing time, memory usage) and the number of parameters (reducing the risk of overfitting).

- **Fully connected layer**: it connects with all the neurons in the preceding and succeeding layer as in regular fully connected neural networks. Its objective is to map the representation between the input and the output.

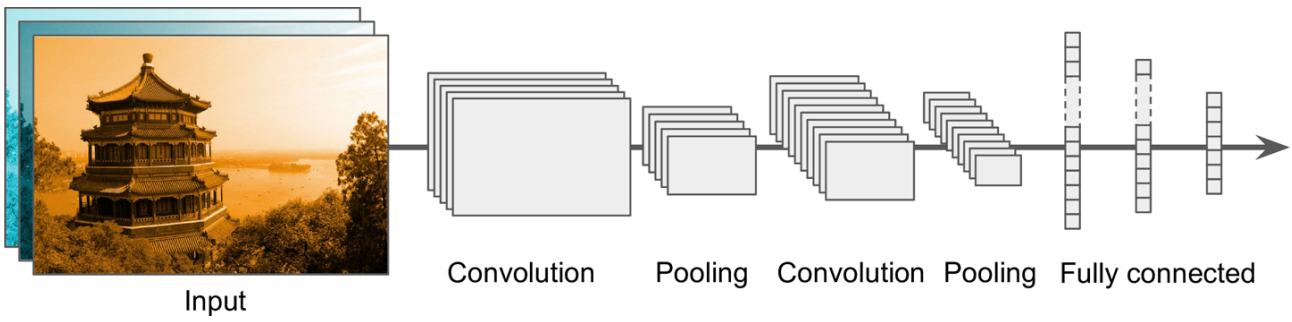

Figure 3.8: Typical CNN architecture (23)

3.5.3 U-net network

U–net (36) is a convolutional neural network, developed for biomedical image segmentation. Prior architectures use max–pooling or convolutions with stride > 1 to achieve successive abstractions that result in decreased resolution.

U–net addresses this challenge by, in the decoding phase, where the output size must match the input size, in addition to subsequently upsampling the different compressed layers, it feeds every upsampled layer with data from the corresponding layer in the downsizing sequence.

At each upsampling stage, the output from the previous layer is concatenated with its counterpart in the downsampling stage. The final output is a mask with the size of the original image.

Even though this network is intended to address segmentation problems, with a minor modification it can perform image translation/inpainting tasks, as it will be described in chapters 4 and 5.

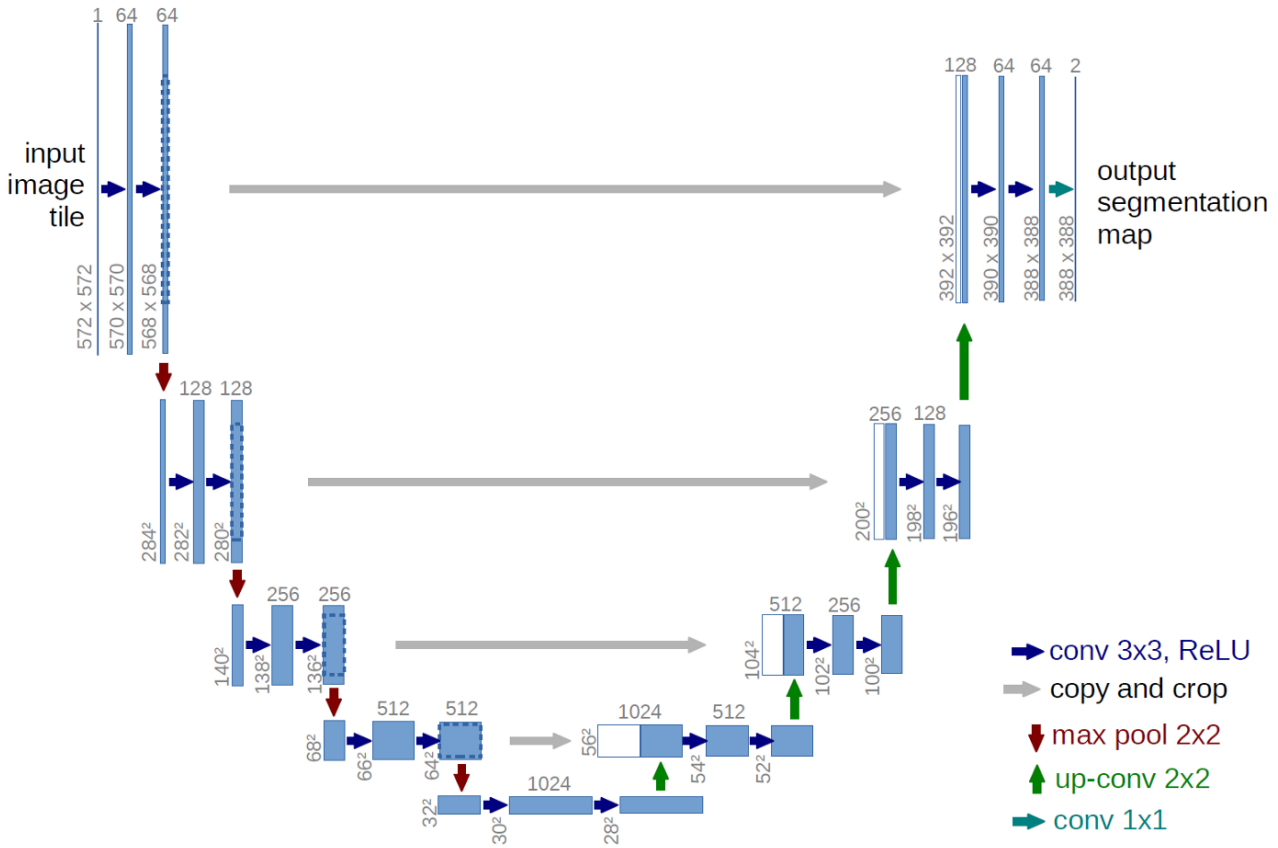

Figure 3.9: U-net architecture from (36)

3.5.4 Hyperparameters

Hyperparameters can be defined as the parameters that control the training process and that are set before the learning process begins; they do not change afterwards. Hereafter some of the most relevant:

3.5.4.1 Train–validation split ratio

In order to know how the model behaves with new data, it is mandatory to assess it against input data the model has not seen, and for which the expected outcome is known. Given an input dataset with the corresponding output, the usual practice is to split it into train and validation. The first is used to feed the model in the learning process; the second, to assess how well it generalises, hence, performs against unknown data.

The most used split is the one that assigns 80% of the data to the training set, and the remaining 20% to the validation set. If the dataset is small, any input data counts; that is why other approaches, like K-Fold cross-validation can be used. The details of this approach are out of the scope of this work.

3.5.4.2 Optimizer

The optimizer is the algorithm used by the neural network to change its weights in order to reduce the losses. Some of the most popular optimizers are SGD (Stochastic Gradient Descent), AdaGrad (Adaptive Gradients), RMSProp (Root Mean Square Propagation) and Adam (Adaptive moment).

It couldbe stated that each optimizer from the above improves the previous one: SGD uses one static learning rate for all the parameters during each training phase; AdaGrad uses a different learning rate for their parameters; RMSProp addresses the issue of low learning rate of AdaGrad by exponentially decaying them. Finally Adam, in addition to use momentum (past learning rates) for updating, it also uses past gradients, aiming at speeding up the learning.

Tests conducted to compare the performance of the optimizers show that Adam outperforms the rest (45).

3.5.4.3 Learning rate

Learning rate defines the magnitude of the weight updates in order to minimize the loss function of the network. When the learning rate is too low, the network will learn very slowly since the

changes in its weights are low as well. However, when the learning rate is too high, the network becomes unstable and is not able to progressively minimize the loss function, oscillating around higher and lower values. Learning rate is expressed as a real number in the range [0,1].

3.5.4.4 Batch size

Batch size defines the number of input items taken from the training dataset to train the network before updating its weights. As an example, if the training dataset contains 8000 samples and the batch size is 32, it will train the network with the the first set of 32 samples, update the weights, train with the second set of 32 samples, and so on for a total of 250 trainings (to cover the 8000 samples). This constitutes an epoch, further explained in 3.5.4.5.

Larger batch sizes usually mean lower training times, since more operations will run in parallel (if computational resources allow it). The tradeoff however is the risk of poor generalization: using smaller batch sizes converge faster to good solutions as the model can start learning from small sets of training data. Some tests performed show that smaller batch size increases accuracy and reduces loss (39).

3.5.4.5 Epochs

An epoch is one complete pass through the full training dataset. The number of epochs, along with the batch size, defines the number of times the weights in the network will be updated.

The number of epochs to choose depends on many factors such as the size of the training dataset, the time or computational resources available, or the improvements observed. For the latter, the early stopping technique stops the training after a defined number of epochs without performance increase.

3.5.4.6 Loss function

The loss function is used to drive the learning process of a neural network. The aim of the training is to minimize that function, so the network will try, after each iteration, to change the weights to achieve that.

There are widely used loss functions, suitable for different machine learning problems, namely regression (for continuous values prediction) and classification (for categorical values prediction):

- For regression problems, one of those functions is the mean squared error, which is the mean of the squared differences between the real value and the predicted value. Some variations of this function are the mean absolute error and the mean squared logarithmic error.

- For classification problems, these are the binary or categorical cross–entropy functions, which quantify the difference between two probability distributions: the distribution of true values and the distribution of predicted values. The formula below generalizes the cross–entropy function for a network based on multiple neurons and layers, where n is the number of training instances, d is the input, j is the number of neurons or layers, $c = c_1, c_2, ...$ are the real values and $y_1^L, y_2^L, ...$ are the values predicted by the output layer.

$$C = -\frac{1}{n} \sum_d \sum_j [c_j \ ln(y_j^L) + (1 - c_j) ln(1 - y_j^L)] \quad (22)$$

3.5.5 Performance metrics

The following section describes some of the metrics used for assessing the performance of a model involving image–related problems, namely the mean squared error, the peak signal to noise ratio and the structural similarity index.

3.5.5.1 Mean squared error

Continuing with the definition in 3.5.4.6, Mean squared error (MSE) is one of the most common metrics for image quality assessment, where values closer to zero mean better image quality. One problem with this metric is that it has a strong dependency on the image intensity scaling, so it is not suitable for images under different intensity scales.

3.5.5.2 Peak signal to noise ratio

Peak signal to noise ratio (PSNR) indicates the ratio between the maximum possible signal power and the power of the distorting noise affecting the quality of its representation. It avoids the issue of the intensity scaling MSE suffers, by scaling the MSE according to the image range. It is usually expressed as a logarithmic quantity using the decibel scale.

3.5.5.3 Structural similarity index

The structural similarity index (SSIM) (43) is a well-known quality metric used to measure the similarity between two images. One of its benefits is that takes advantage of known characteristics of the human visual system, defining the image distortion as a combination of loss

of correlation, luminance distortion and contrast distortion. As shown in Figure 3.10, mean squared error is not always sufficient as a performance metric for image–related problems.

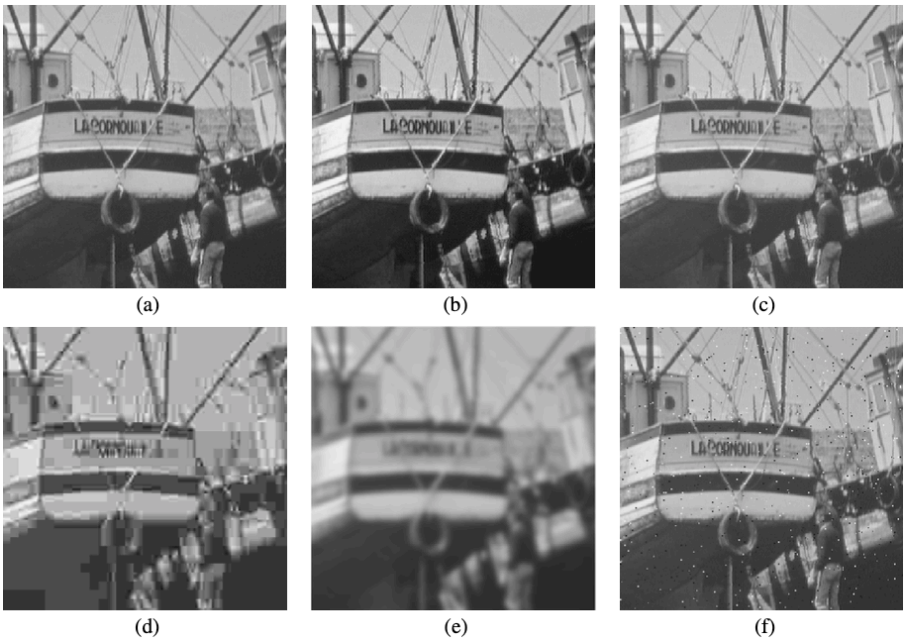

Figure 3.10: Distortion examples from (43) where SSIM drops from (b)=0.92 to (f)=0.77, although MSE remains unchanged at 210, being (a) the original image

3.5.6 Data augmentation

Data augmentation is a technique used to artificially expand labeled training datasets, creating modified versions of the existing ones. It is closely related to neural networks as this kind of models usually require a high volume of data which is not always available.

Any kind of data (text, audio, images) can be augmented. Focusing on image data augmentation, the main changes that can be applied are:

- **Geometric transformations**: flip, crop, rotate or translate are some examples
- **Color space transformations**: changing the intensity of any of the RGB color channels
- **Kernel filters**: blurring or sharpening
- **Random erasing**: removing parts of the original image
- **Mixing images**: mixing two different images or the same one in different positions.

The objectives of data augmentation are to increase the generalization power (prevent overfitting) and enhance the model performance.

Chapter 4

Method

This chapter describes how the theory presented in chapter 3 is applied setting the basis for the several experiments elaborated in chapter 5. Hardware and software used, along with all the data preparation and model configuration activities are described and justified, the latter when considered relevant.

4.1 Hardware and software

4.1.1 Hardware

4.1.1.1 System

The system used to run all the experiments is an Apple MacBook Pro with the following specifications:

- **Architecture**: x86–64
- **Processor**: Intel Mobile Core i7 "Coffee Lake" (i7-9750H) with 6 cores, base speed 2.6 GHz, turbo boost 4.5 GHz, L1 cache 32KB x 6, L2 cache 256KB x 6, L3 cache 12MB.
- **Memory**: 16GB DDR4 SDRAM, speed 2.66 GHz.

4.1.1.2 Execution engine: AMD Radeon Pro 5300M GPU

The Radeon Pro 5300M is a professional mobile graphics chip by AMD.

- **Architecture**: RDNA 1.0
- **Processor**: 1280 shading units, 80 texture mapping units, 32 ROPs and 20 compute units; L2 cache 2MB.

- **Memory**: 4GB GDDR6, speed 1.5GHz, 12Gbps effective
- **Theoretical performance**: 40 GPixel/s, 100 GTexel/s, 6400 TFLOPS (FP16), 3200 TFLOPS (FP32) 200 GFLOPS (FP64).

4.1.1.3 Execution engine: Intel UHD Graphics 630 GPU

The UHD Graphics 630 is an integrated graphics solution by Intel.

- **Architecture**: Generation 9.5
- **Processor**: 184 shading units, 23 texture mapping units, 3 ROPs and 23 compute units.
- **Memory**: Up to 1.5GB of system memory.
- **Theoretical performance**: 3.45 GPixel/s, 26.45 GTexel/s, 846.4 GFLOPS (FP16), 423.2 GFLOPS (FP32) 105.8 GFLOPS (FP64).

4.1.1.4 Execution engine: Intel i7 6–core CPU

See 4.1.1.1 for details.

4.1.2 Software

4.1.2.1 Operating system

MacOS BigSur version 11.4. At the time of writing, it is the latest operating system released by Apple.

4.1.2.2 Programming language

The programming language used in this work is Python version 3.8.5 64-bit. Python is an interpreted object-oriented programming language. It combines dynamic typing and dynamic binding and contains built-in data structures. Its syntax is focused in readability, making the code easier to understand and to maintain (2).

4.1.2.3 Development environment

The development environments chosen are the Jupyter Notebook under Microsoft Visual Studio Code and Anaconda. The Jupyter Notebook (40) is an open–source web application that allows the creation and sharing of documents containing live code, equations, visualizations and narrative text in an interactive manner. It supports, amongst other Python, R, Julia or

Scala programming languages.

Microsoft Visual Studio Code (3) is a source code editor that runs as a desktop application and is available for Windows MacOS and Linux. It has a rich ecosystem of extensions for different languages such as C++, C# or Python. The version used in this work is 1.56.2. The Jupyter Extension for Visual Studio Code (7) enables Jupyter notebook edition in Visual Studio Code.

Anaconda (5) is an open-source distribution of programming languages, mainly Python and R. It is oriented to data science–related developments. The distribution includes hundreds of packages and a package manager with thousands of open–source additional libraries. Jupyter notebook is provided and runs in any modern web browser.

4.1.2.4 Libraries

General functions are provided by the common libraries used in Python for data science–related projects. Some examples are:

- **Numpy** (24) : numerical computing
- **Pandas** (34) : data manipulation
- **Matplotlib** (25) : data visualization
- **Scikit-image** (42) and **Pillow** (18) : image manipulation

MRI data are accessible using Nibabel (14). Nibabel provides read/write access to some common medical and neuroimaging file formats, including NIFTI (1).

Deep learning functions are provided by Keras (41). It is a deep learning API written in Python, running on top of the machine learning platform TensorFlow (8), as a high–level API. It is an approachable, highly-productive interface for solving machine learning problems, with a focus on deep learning. It provides essential abstractions and building blocks for developing and shipping machine learning solutions.

4.1.2.5 Tensor compiler

PlaidML is an advanced and portable tensor compiler for enabling deep learning on laptops or embedded devices (amongst other), where the available computing hardware is not well supported or the available software stack contains questionable license restrictions. PlaidML

sits underneath common machine learning frameworks, enabling users to access any hardware supported by PlaidML (30).

PlaidML is, at the time of writing, the only tensor compiler that allows the execution using the GPU engines described above. It supports the Metal API (26) as default configuration, but also gives the opportunity to try the OpenCL API (27), although as an experimental configuration. Considering the overall stability on the supported platforms, the experimental configuration is not recommended on production–systems. In addition to the native support to GPUs, PlaidML enables the use of the CPU for tensor–related tasks. The version used in this work is 0.7.0.

4.2 General methodology

The starting point for the work described in this report is in (38), where a basic convolutional autoencoder is presented. That article describes how to manipulate MRI in NIFTI format, to feed them to a simple model and to extract the result of the encoding–decoding process. A stepped approach has been followed, first developing and testing the basic workflow:

Data ingestion → *Model training* → *Performance analysis*

Then adding complexity to it, by manipulating data –adding patches and noise, reducing the resolution–, using different resolutions for the input images, using different subsets of each complete MRI scan, increasing the complexity of the model and testing with different execution engines. The sections below are presented in the same sequence as the execution of the code.

4.3 Environment preparation

PlaidML provides the `plaidml-setup` command–line tool to select the compute engine to use in tensor–related activities with Keras. In the code, several adjustments must be made in order to make Python use the PlaidML compiler:

```
1 import os
2 os.environ["RUNFILES_DIR"] = "/usr/local/share/plaidml"
3 os.environ["PLAIDML_NATIVE_PATH"] = "/usr/local/lib/libplaidml.dylib"
4 import plaidml.keras
5 plaidml.keras.install_backend()
6 os.environ["KERAS_BACKEND"] = "plaidml.keras.backend"
```

Listing 4.1: PlaidML libraries loading

The rest of the libraries used are the ones described in 4.1.2.4.

4.4 Data preparation

4.4.1 Input dataset

The input dataset is the IXI dataset (13), consisting of 581 T1–weighted MRI from normal, healthy subjects, collected at three different hospitals in London:

- Hammersmith Hospital using a Philips 3T system
- Guy's Hospital using a Philips 1.5T system
- Institute of Psychiatry using a GE 1.5T system

The MRI are in Neuroimaging Informatics Technology Initiative (NIFTI) format, with a typical matrix size of 256 x 256 x 150 voxels, with a resolution of 0.938 x 0.938 x 1.2 mm (X, Y, Z axes).

4.4.2 Train/validation split

The input dataset is split between training and validation, allocating approximately 80% to the training subset, and the remaining 20% to the validation subset. The granularity of the dataset for the split is the MRI: this ensures that after shuffling the data, slices belonging to the same MRI (subject) are not in both training and validation subsets as this could artificially facilitate the work of the network.

More specifically, for the full dataset, out of the 581 MRI, 465 (80.03%) are assigned to the training subset and the remaining 116 (19.97%) to the validation subset.

4.4.3 Anatomical plane

The anatomical plane mainly used in this work is the **axial**, providing 256 slices of 256x150 pixels. Sagittal and coronal planes can be obtained by pivoting the slices' axis.

4.4.4 Input resolution

Due to time/performance limitations, the slices of each MRI are downsized. The main experiments are performed using slices of 64 x 64 pixels and 32 x 32 pixels. This represents a data reduction of 89.3% and 97.3% respectively. Few experiments are however also observed using 128 x 128 pixel slices.

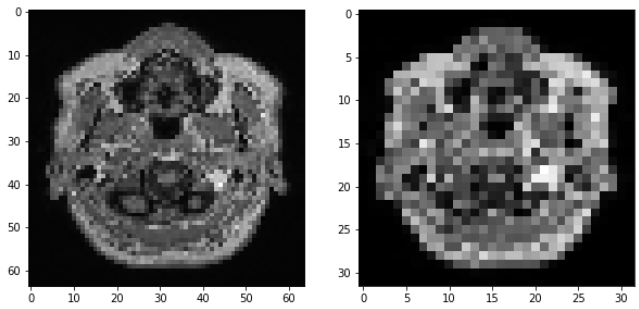

Figure 4.1: 64 x 64 pixel slice (left) and 32 x 32 pixel slice (right)

4.4.5 Number of slices

Using the axial plane, two ranges of slices for each MRI have been provided as input data:

51 central slices [102,153]: this subset of slices (approx. 20%) around the center ensures low variability between the MRI of the different subjects, as shown in figure 4.2.

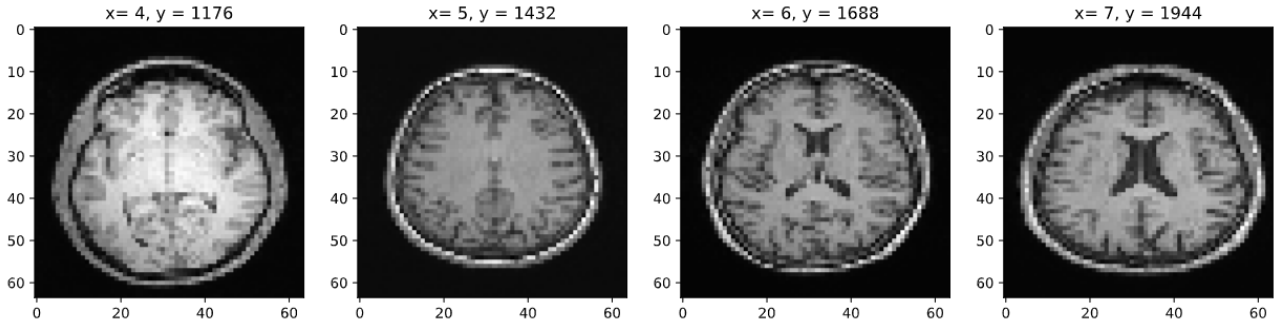

Figure 4.2: Slice #153 for 4 different MRI

```
1    ...
2    a = nib.load( input[f])
3    a = a.get_fdata()
4    a = a[:,53:204,:]
5    ...
```

Listing 4.2: Selection of the 51 central slices from each MRI

151 central slices [52,203]: this subset of slices (approx. 59%) around the center increases the size of the input dataset, and it could be seen as a form of data augmentation. However it

introduces more variability –the difference in head's anatomy of the subjects or their position are more evident (see figure 4.5)–, forcing the neural network to learn additional shapes.

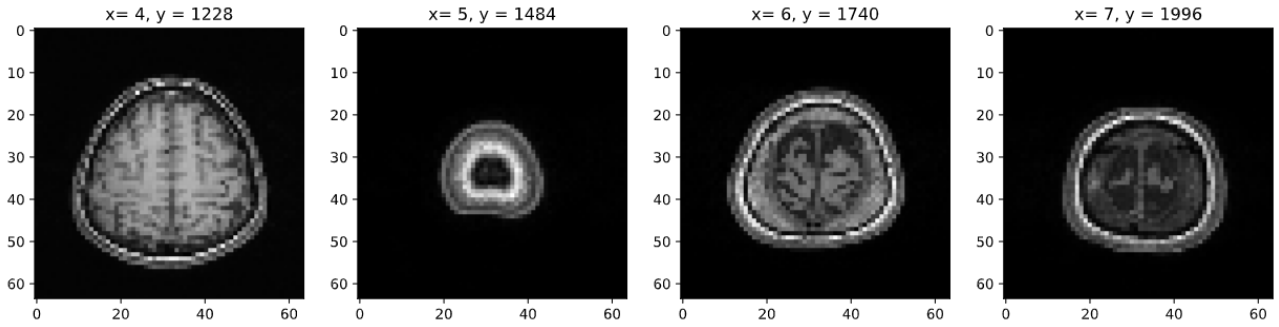

Figure 4.3: Last slice in the [52,203] range for 4 different MRI (same MRI than in figure 4.2)

The complete set of slices has been discarded, since these eccentric slices adds even more variability to the dataset, leaving it unclear whether that variability is worth the additional information they might provide.

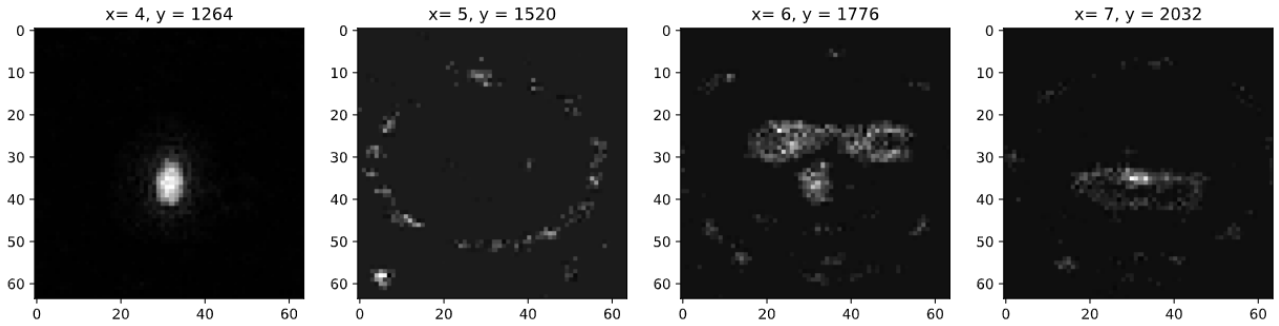

Figure 4.4: Slice #240 (out of 256) for 4 different MRI (same MRI than in figure 4.2)

4.4.6 Data pre–processing

Based on the parameters described above (anatomical plane, resolution and number of slices), the training and validation datasets are processed as follows:

1. The slices of each NIFTI file are extracted according to the range defined for the workflow (e.g. [102,153])

2. Each slice is resized to the size defined for the workflow and added to a list of pictures, later converted to an array

3. Each slice in the array is normalized using the min–max normalization technique

4. Each set of slices is extracted from each NIFTI file and resized to the size defined for the workflow (e.g. 64 x 64 pixel), then added to an array of pictures.

4.4.7 Patch generation

A mean to train the network to identify anomalous areas in a slice of a MRI is to add artificially–generated anomalies (patches) to the input (anomaly–free) data. The patches are white lines of random length, position, thickness and opacity. These are added to the pictures in this fashion:

1. An empty mask is generated with the size of the input picture

2. Random values are generated for the position of the line (x and y start and end coordinates), around the center of the picture (to avoid creating patches in blank areas), random values are also generated for the thickness (up to 1/8 of the picture height) and opacity of the line

3. The line with the parameters calculated above is added to the empty mask

4. If the intensity of the patch is higher than the maximum intensity of the input picture, the intensity of the patch is reduced to the maximum intensity of the picture, to avoid darkening the input image (since the dynamic range of the resulting picture would be increased by the patch)

5. The mask containing the patch is added to the input picture

6. The function returns the masked picture as well as the coordinates and thickness of the patch: this will allow applying the metrics of the model to the patched area.

4.4.8 Resolution reduction

Another expected capability of the model is super–resolution, i.e. increasing the resolution of the input images. In the case of this work, to double it: from 32 x 32 pixel to 64 x 64 pixel and from 16 x 16 pixel to 32 x 32 pixel.

4.4.9 Gibb's ringing noise generation

Based on the work done in (31), a function has been created to simulate Gibb's ringing noise. The purpose is to perform data augmentation and to assess the capacity of the model to correct this type of noise. It works as follows::

1. A 2D Fourier transform is applied to the input image

2. The resulting k–space is reduced by a defined value, removing the higher frequencies

3. The noisy image is reconstructed by applying the inverse 2D Fourier transform.

4.4.10 Ghosting noise generation

Following the same approach described in 4.4.9, and for the same purposes (data augmentation and noise reduction) ghosting noise can be added by scaling odd or even lines in the k–space of the input picture.

4.4.11 Data augmentation

As part of the experiments implemented, the techniques presented in sections 4.4.9 and 4.4.10 have been used for data augmentation purposes. More precisely, the resulting augmented dataset is three times the original one, since every slice has three variations: with reduced resolution, with Gibb's ringing noise and with ghosting noise.

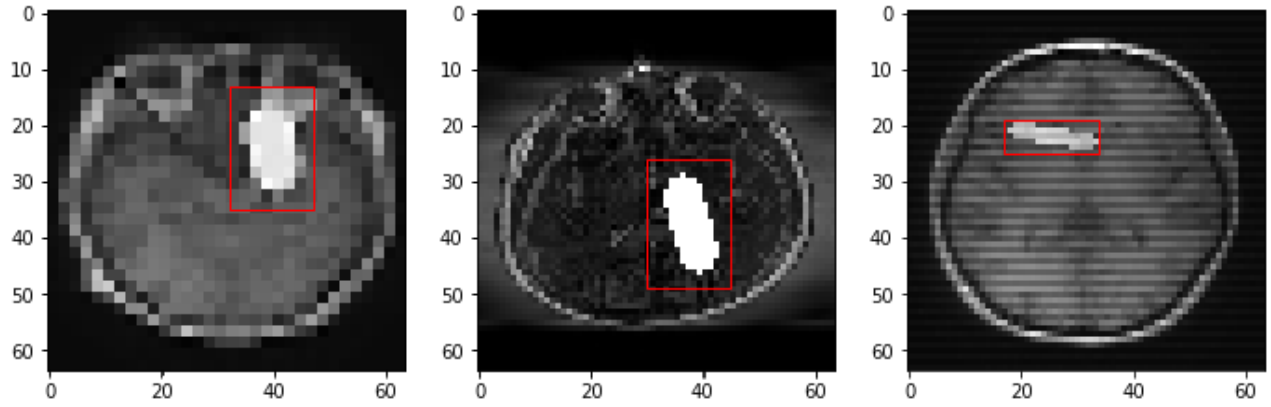

Figure 4.5: Application of patches (all the pictures), resolution reduction (left), Gibb's ringing noise (center) and ghosting noise (right)

When using the above–mentioned data augmentation techniques, the ground truth data (the original, unaltered images) are also triplicated in the same sequence as the augmented images.

4.5 Training the model

This section describes the network used and its related parameters:

4.5.1 U–net network

The network proposed is based on the U–net network, adapting its input layer to the size of the pictures and adjusting the last layer to the specific problem of this work: instead of using a softmax activation (which is intended for segmentation problems), the last layer uses the ReLU activation function. The resulting network has in total 38 layers and more than 31 million parameters.

4.5.2 Model parameters

The following parameters have been used to train and assess the network:

4.5.2.1 Loss function

The loss function used is the mean squared error, since it is one of the most used functions for regression problems. It is however known that this function might not be optimal for image quality assessment. The intention in this work was to use a custom function, such as the inverse of the structural similarity index, but unfortunately PlaidML (the tensor compiler) in the latest version available at the time of writing, does not support custom functions in the training process.

The approach followed to overcome this limitation is to calculate the relevant metrics after each epoch and store all the models generated, to later keep only the most performant. More details on this approach are provided in 4.5.3.1.

4.5.2.2 Optimizer and learning rate

Following the findings described in 3.5.4.2, Adam is the optimizer chosen for the different experiments. Its learning rate is 0.0001. This value is also widely used and shows stable while fast improvements.

4.5.2.3 Batch size

As shown in 3.5.4.4, a bigger batch size can be leveraged by the execution engine, especially the GPU; however it can become a bottleneck and increase the overall training time.

The batch sizes have varied, depending on the specific experiment, namely: the size of the dataset, input images and the execution engine used (discrete GPU, embedded GPU or CPU). Since the difference in performa nce between the embedded GPU and the CPU used is not remarkable, the same batch sizes have been used for both.

- **Discrete GPU**: Batch size between 48 and 6.
- **Embedded GPU / CPU**: Batch size between 1 and 2.

It is worth noting that, in the case of the discrete GPU used, bigger batch size does not directly mean better performance. Moreover, the best results are achieved using different batch sizes than the usual ones (e.g. 32 or 64), working better with batch sizes that are multiple of 6 and power of 2. As an example, a batch size of 24 outperforms 16 or 32. Acknowledging that fact, for 64 x 64 pixels problems, the sweet spot is in the range [6,24]; for 32 x 32 pixels problems, in [12,48].

4.5.2.4 Epochs

The number of epochs the model has been trained depended on the performance increase observed over the epochs. The main experiments have run during up to 50 epochs, whilst other only 10 or 20, as the most important improvements happen after the first ones, with marginal improvement afterwards.

4.5.2.5 Performance metrics

The performance metrics used are the ones introduced in 3.5.5: mean squared error (MSE), peak signal to noise ratio (PSNR) and the structural similarity index (SSIM). Since only MSE is included as a standard loss/accuracy function by Keras, and PlaidML does not support custom functions, these are calculated as callbacks after each epoch.

The metrics indicated above are calculated for the validation data, both for the complete slice –to test the whole reconstruction performance– and for the rectangular area covering each patch –to test the inpainting performance–. The location of the patches is provided by the patch generation function.

```
1 ...
2 psn=[]
3 mse=[]
4 ssi=[]
5
```

```
# For each slice in the validation dataset
for i in  range( len(self.pred)):
    # store the original (ground truth) image
    i1 = test_original[i, ..., 0].reshape(img_size*img_size)
    # predict the input validation image
    i2 = self.pred[i, ..., 0].reshape(img_size*img_size)
    # calculate the PSNR and add it to the psn temp array
    psn.append(cv2.PSNR(np.float32(i1), np.float32(i2), 1))
    # calculate the MSE and add it to the mse temp array
    mse.append(np.square(np.subtract(i1, i2).mean()))
    # calculate the SSIM and add it to the ssi temp array
    ssi.append(ssim(i1, i2, data_range=i2. max() - i2. min()))

# After calculating the metrics for each image,
# store the averagein the final arrays.
psnr_result.append(np.average(psn))
mse_result.append(np.average(mse))
ssim_result.append(np.average(ssi))
...
```

Listing 4.3: Performance metrics calculation

4.5.3 Model configuration

This section completes the second step of the workflow from 4.2 (Model training), and consists in setting the configuration of the model –based on parameters presented previously–, control its behavior (using callbacks and checkpoints), and finally training the model.

4.5.3.1 Callbacks and checkpoints

Callbacks are special functions that can be instantiated before or after a specific event has happened. That feature is highly valuable since it helps overcoming the limitations of the PlaidML library related to the use of custom performance metrics, and allows a close follow–up of the learning process. The latter is especially relevant in the cases where training time is long.

In the scope of this work, callbacks have been implemented for the following events:

- **On train begin**: to show few samples of the input data, highlighting the patched areas.
- **On epoch end**: to calculate and record the performance metrics and show the reconstruction result of the sample images

- **On train end**: to show the final reconstruction result for the sample images, the expected result (ground truth) and the related heat maps highlighting the differences.

A custom variable controls the frequency of the `on_epoch_end` function (similar to the parameter `period` in the `ModelCheckpoint` function.

Due to the time required for training the model, especially in the most complex cases, these callbacks store the information calculated and shown also in the filesystem. This is useful in case of anomalous behavior of the system (e.g. unexpected reboot) to preserve the information.

Checkpoints also store the model after each epoch (or another defined frequency). In this case, all models are preserved, since keeping only the best model according to the standard metrics available cannot guarantee that model would be the best according to the custom metrics (SSIM, PSNR).

4.5.3.2 Compilation and training

All the parameters described above are used to configure, compile and train the model:

```
# Load the model
model = Unet_model()

# Compile the model
model.compile(optimizer = Adam(lr = 1e-4), loss = 'mean_squared_error')

# Compilation summary
model.summary()

# Storage of the model
model_checkpoint = ModelCheckpoint(
                    filepath='unet_augmented_16_amd_top_50_{epoch:02d}.h5',
                    monitor='loss', verbose=1,
                    save_best_only=False, period=1)

# Training the model
history = model.fit(train_masked,
                    train_original,
                    validation_data=(test_masked, test_original),
                    epochs=20, batch_size=6, verbose=1,
                    callbacks=[model_checkpoint,show_result_Callback()])
```

Listing 4.4: Model compilation and training

Chapter 5

Experiments and results

In this chapter, the theory and method described in chapters 3 and 4 are put into practice. The main scenario, based on the axial plane of the MRI,is the most thorough one. It is based on the main challenges to address (reconstruction, inpainting, super–resolution) considering the resources limitations of the main execution engine used (discrete GPU, 4.1.1.2) and time available.

The low–requirements scenario aims at showing the practical applicability of this model in low–end execution engines, such as the integrated GPU (4.1.1.3) and the CPU used (4.1.1.1).

The other scenarios described include complementary experiments that can help extrapolate the results obtained in the default and low–requirement scenarios to other cases not tested so thoroughly; or to refute such extrapolation. These include the use of a reduced dataset, higher resolution images (128 x 128 pixels) and reconstruction of other planes.

5.1 Main scenario: 64 x 64 pixel slices

5.1.1 581 samples (original dataset)

5.1.1.1 51 central slices

The main default scenario uses the original data set, with the 51 central slices of each MRI, and produces good results for the slices: SSIM ≈ 0.85; and acceptable for the patches: ≈ 0.57. The performance of the model is already visible after the first epoch for the slices: SSIM ≈ 0.74, although the patches require more time to learn. Figure 5.2 shows that the bigger improvements for the slices take place during the first 10 epochs, after which the change $epoch/epoch - 1 < 0.5\%$ in almost all the cases. The learning process of the patches show more unstability, oscillating around the 2% in many $epoch/epoch - 1$ periods.

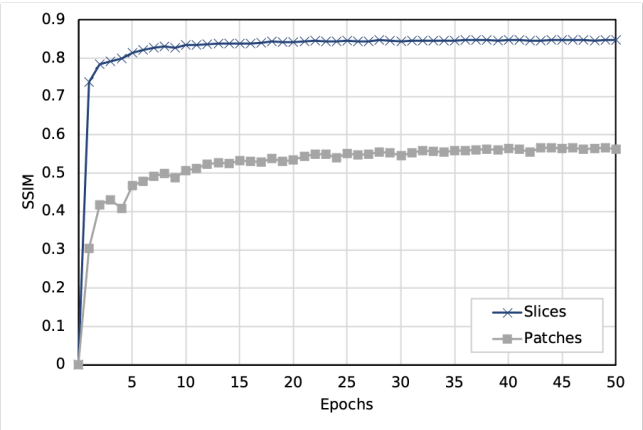

Figure 5.1: SSIM

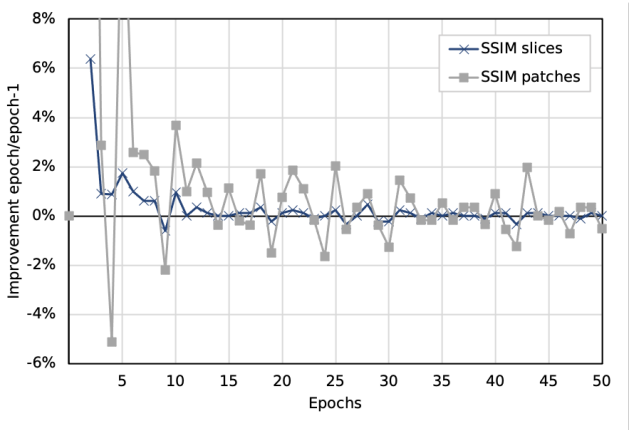

Figure 5.2: SSIM change epoch/epoch–1

With regards to the computational cost, and as expected, the discrete GPU significantly outperforms the integrated GPU and the CPU, and even the CPU is slightly faster than the integrated GPU in this experiment. As a summary, the discrete GPU completes the training of the model (for 50 epochs) in 11.4 hours, 72x faster than the integrated GPU (822 hours) and 64x faster than the CPU (730 hours). It is worth mentioning that in order to assess the computational time required for the integrated GPU and the CPU, these have run for 1 or few epochs, extrapolating the result to the epochs run in the complete training.

In case of time constraints (e.g. 24 hours), only one epoch would complete for the integrated GPU and the CPU, delivering, as mentioned above, a reasonable result for the slices, but probably insufficient for the patches.

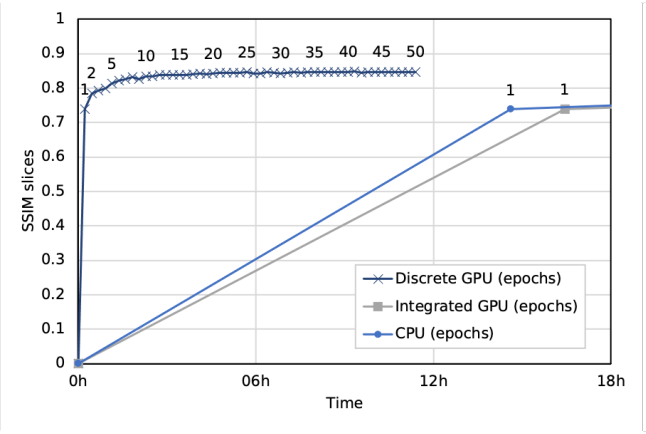

Figure 5.3: SSIM slices over time

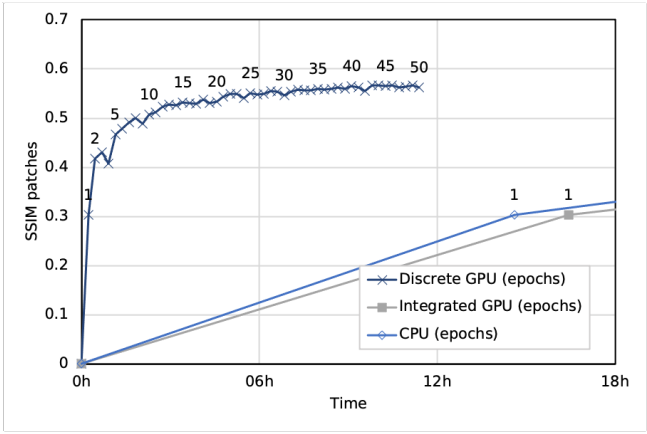

Figure 5.4: SSIM patches over time

The figures below show how the model performs in the cases highlighted: after 1 epoch and after 50 (complete training). As explained previously, the complete training has only been performed using the discrete GPU (for time limitation reasons), however the results in terms of performance metrics (mainly SSIM, but also PSNR and MSE) are applicable to any execution engine.

To illustrate the quality of the model, 5 samples of input slices are provided (highlighting the patched areas), with their corresponding reconstructed results, the original slices (ground truth) and the difference between them.

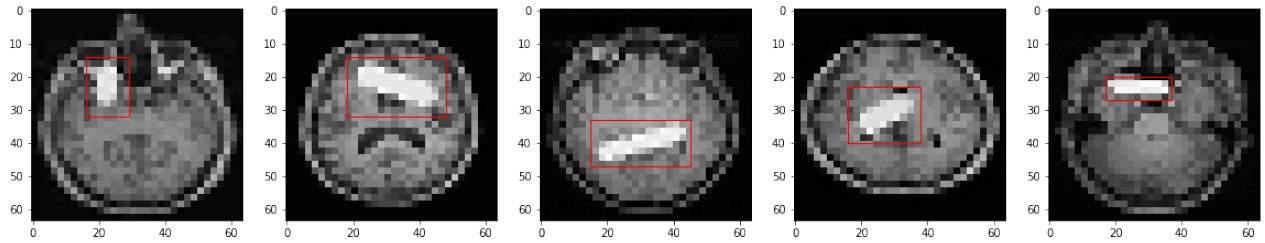

Figure 5.5: Input slices with the patched area highlighted

The patches are generated randomly around the center of the image, with different sizes and opacity. The images are provided in low resolution so the model can do both the inpainting and super–resolution tasks.

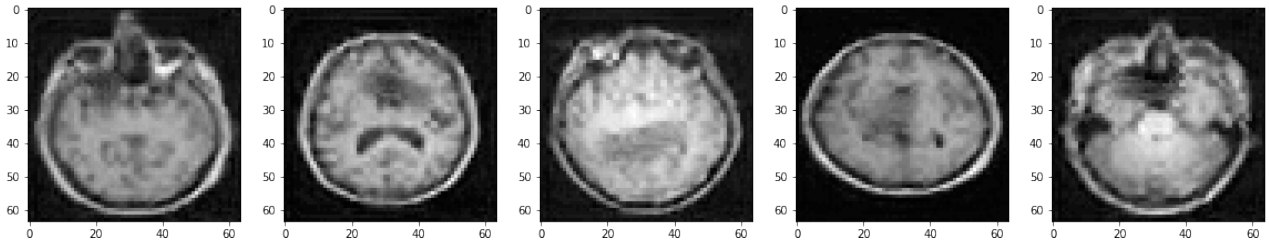

Figure 5.6: Output after 1 epoch

The result after one epoch (above) is generally goood although inpainted (patched) areas are easily identifiable since they appear blurrier than the rest of the image and with little context. After 50 epochs however (below), the inpainted areas are harder to distinguish.

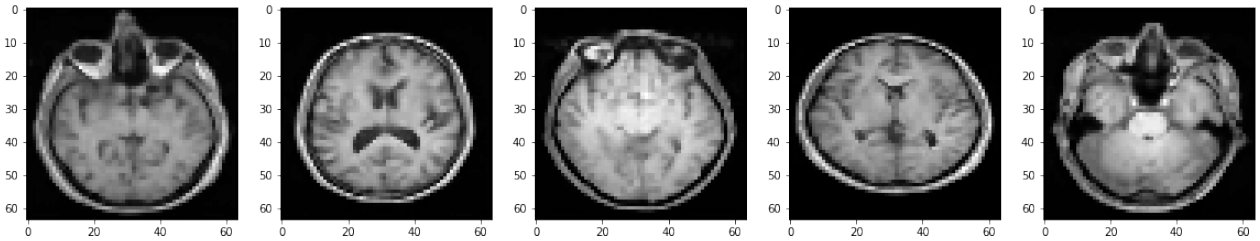

Figure 5.7: Output after 50 epochs

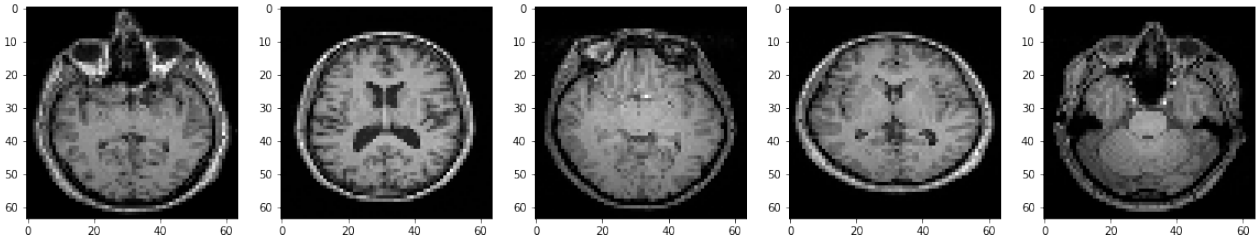

Figure 5.8: Original slices (ground truth)

Although the original images (ground truth) contain more details, the result of super–resolution and patch inpainting provide a good level of similarity, supported by the metrics used and the samples visualized: the difference between the original and reconstructed images are pixel–wise below 5%.

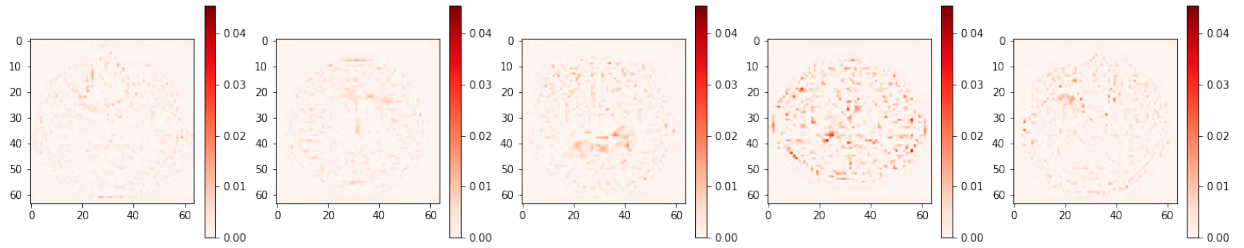

Figure 5.9: Difference between reconstructed slices and ground truth

5.1.1.2 151 central slices

As foreseen in 4.4.5, the additional eccentric slices cannot be used as a data augmentation technique, since they add more variability hence more use cases the model has to learn. This is clearly shown in figure 5.10, where the best SSIM is low compared to previous cases and the model does not improve over time.

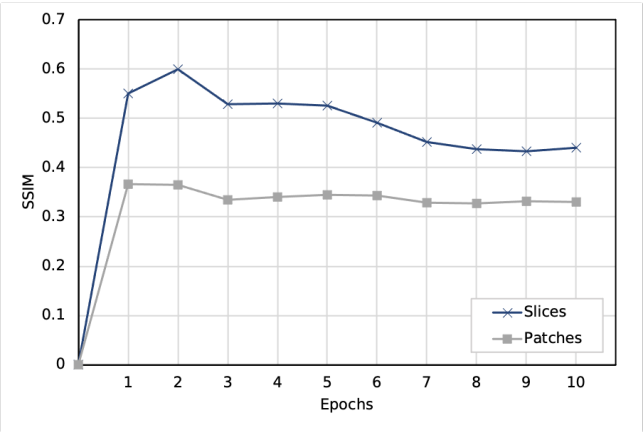

Figure 5.10: SSIM

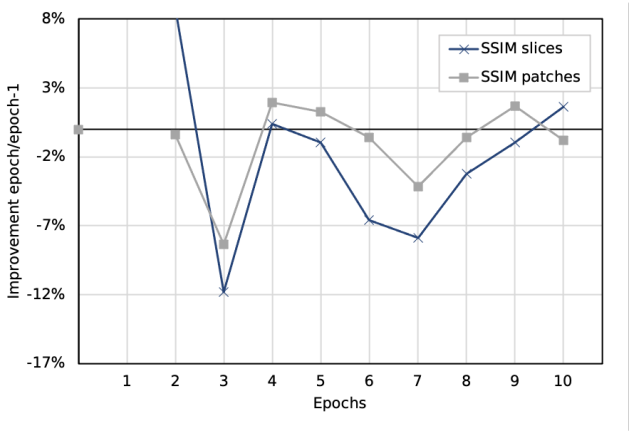

Figure 5.11: SSIM change epoch/epoch–1

From the shown slices in figure 5.12, only the first one is correctly reconstructed. With regards to the computational cost, the tests performed are not relevant due to the poor performance of the model.

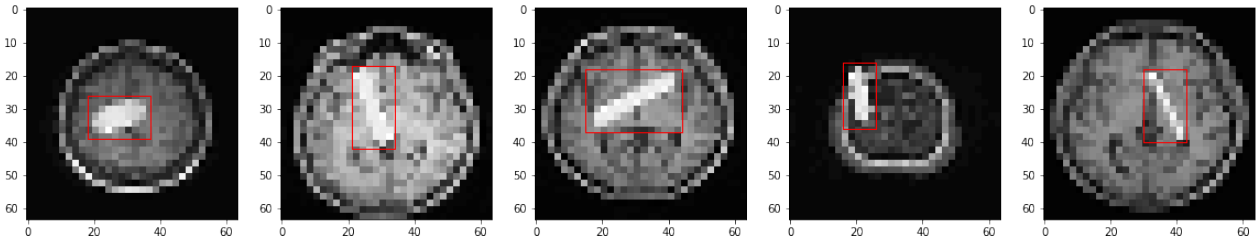

Figure 5.12: Input slices with the patched area highlighted

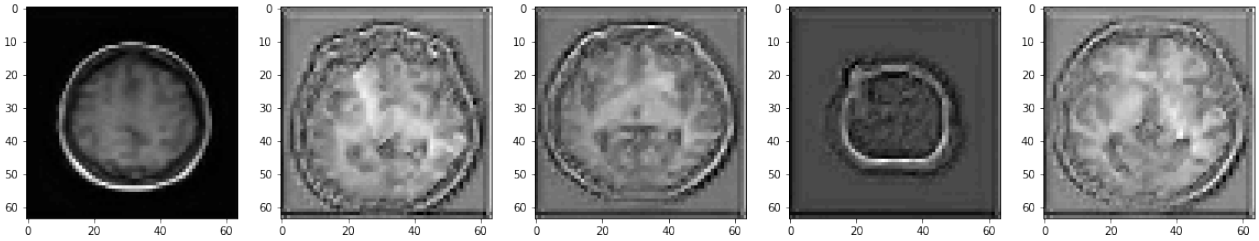

Figure 5.13: Output after 1 epoch

5.1.2 1743 samples (augmented dataset)

5.1.2.1 51 central slices

Creating additional versions of the original slices using the Gibb's ringging and ghosting noise as a data augmentation techniques has proven being a good approach, not only to increase the performance of the model (SSIM slices ≈ 0.91 and SSIM patches ≈ 0.65), but also to remove these two types of additional artifacts. Although executed during 50 epochs, the biggest improvements happened during the first 10: Figure 5.15 shows an increase close to zero in the case of slices, and around 1% (but more unstable) in the case of the patches after 10 epochs.

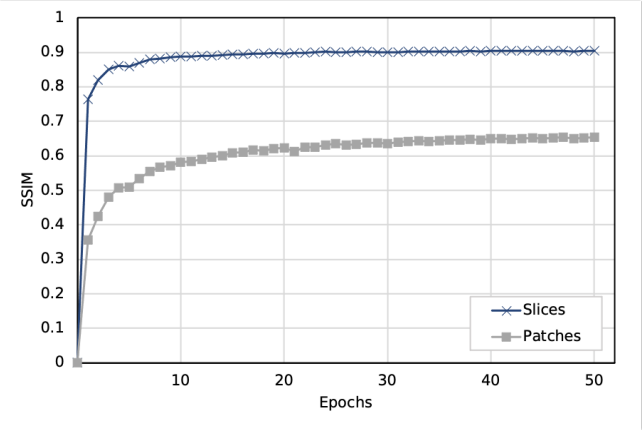

Figure 5.14: SSIM

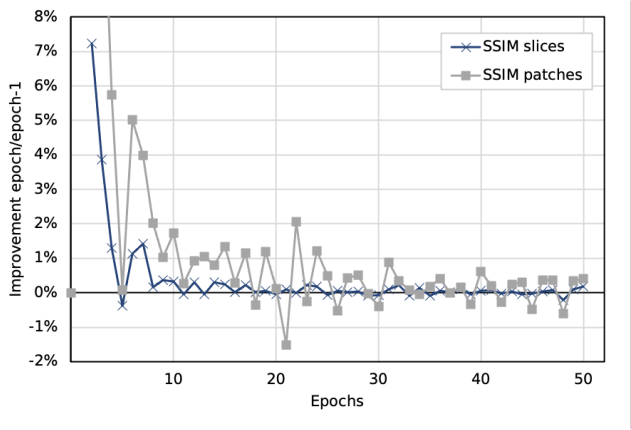

Figure 5.15: SSIM change epoch/epoch–1

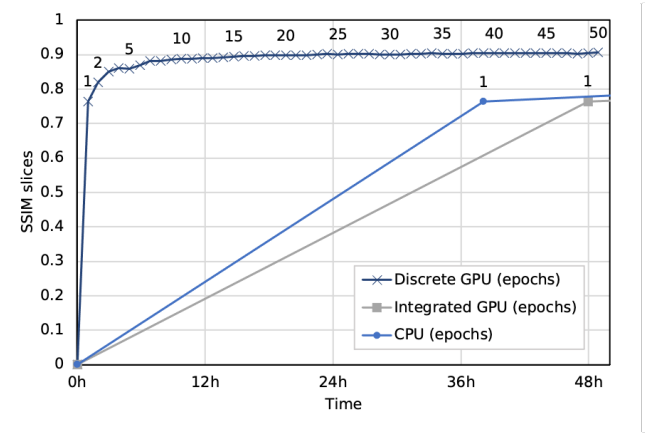

Figure 5.16: SSIM slices over time

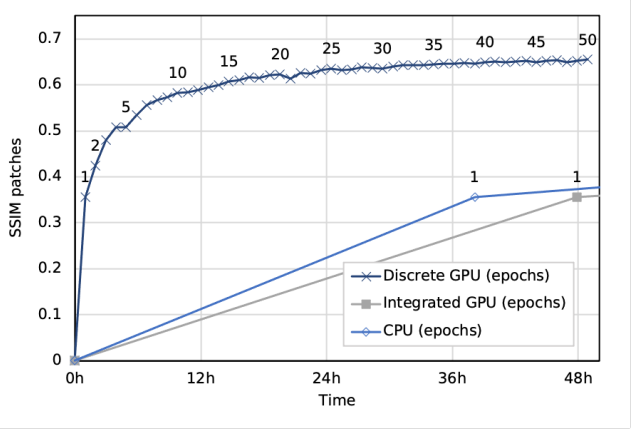

Figure 5.17: SSIM patches over time

In terms of computational cost, the discrete GPU performs 49x faster than the integrated GPU and 39x faster than the CPU. For the latter, despite the performance gap, after 24–28

hours of execution, a SSIM of 0.76 for the slices can be achieved. This could be considered a decent result and it is in the boundaries of a reasonable training time. Unfortunately one epoch is not sufficient to achieve an accurate inpainting: the SSIM obtained is 0.36, compared to 0.64 achieved by the discrete GPU after the same training time.

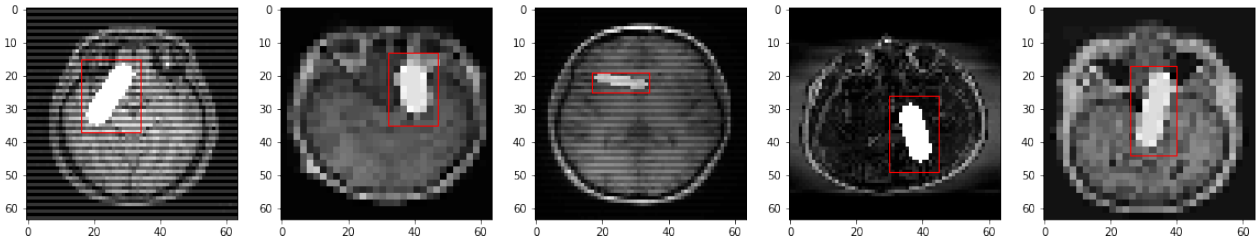

Figure 5.18: Input slices with the patched area highlighted

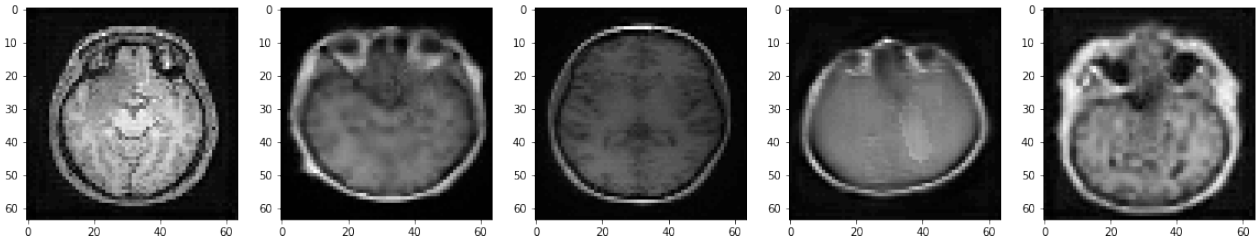

Figure 5.19: Output after 1 epoch

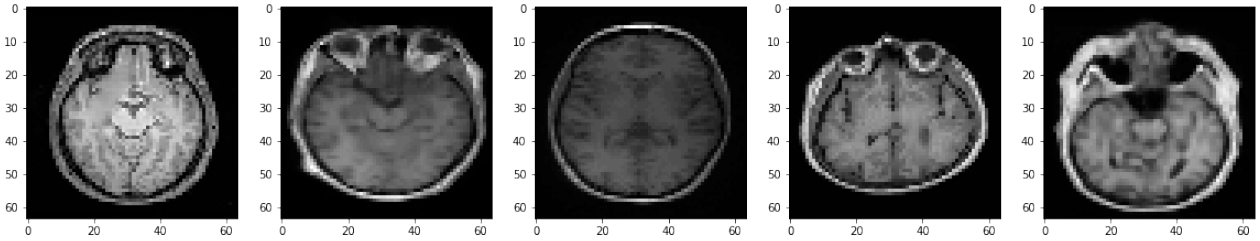

Figure 5.20: Output after 50 epochs

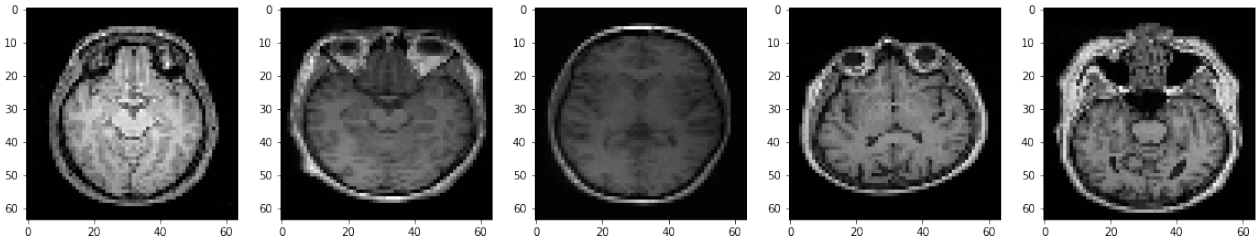

Figure 5.21: Original slices (ground truth)

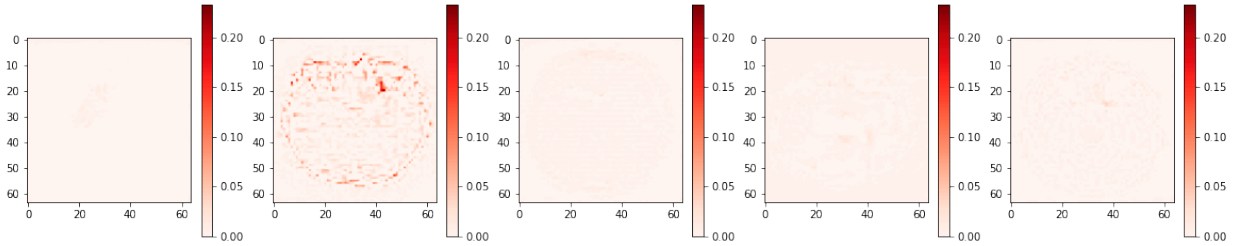

Figure 5.22: Difference between reconstructed slices and ground truth

The second sample above shows an area of up to 20% difference between the original and the reconstructed image, corresponding to inpainting work, where the model shows the complexity of performing such tasks in complex areas.

5.1.2.2 151 central slices

As an additional benefit of those described in 5.1.2.1, creating additional versions of the original slices using the Gibb's ringging and Ghosting noise as a data augmentation techniques now allows the model to produce reasonable results also with more eccentric slices that introduce high variability: SSIM of slices is now ≈ 0.77 and SSIM of patches ≈ 0.53 both in epoch 5.

Observing the decreasing performance for the slices and the slow increase for the patches in figure 5.45, 10 epochs seemed sufficient to observe the behavior of the model.

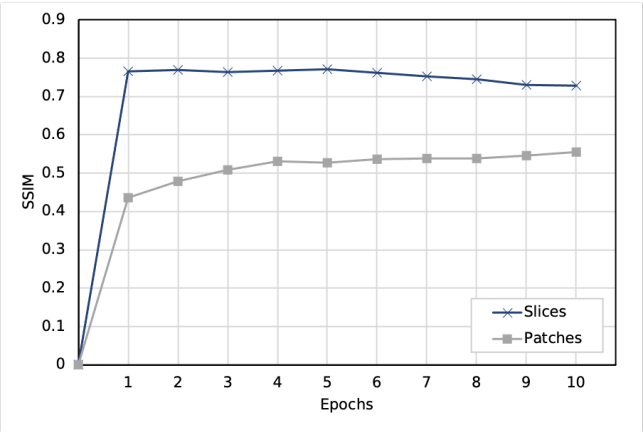

Figure 5.23: SSIM

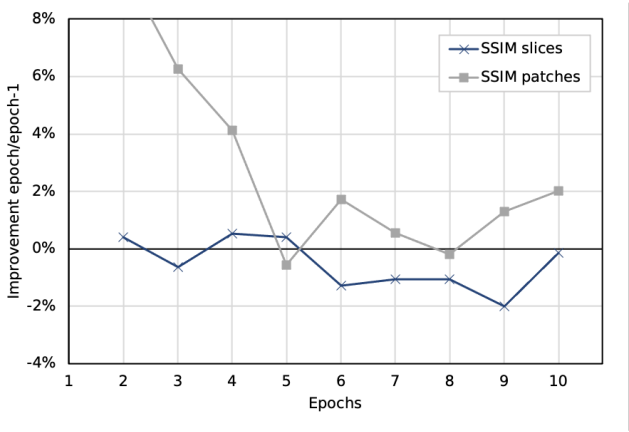

Figure 5.24: SSIM change epoch/epoch–1

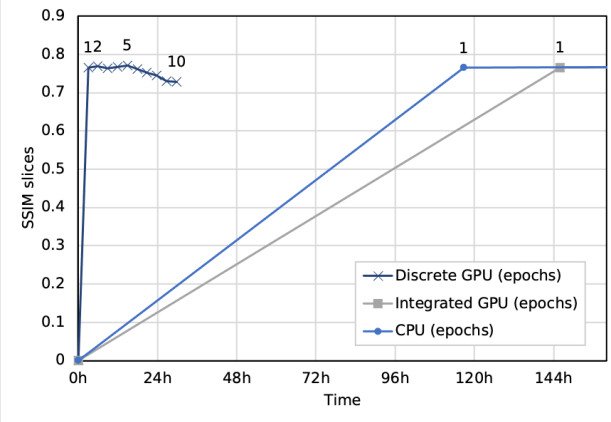

Figure 5.25: SSIM slices over time

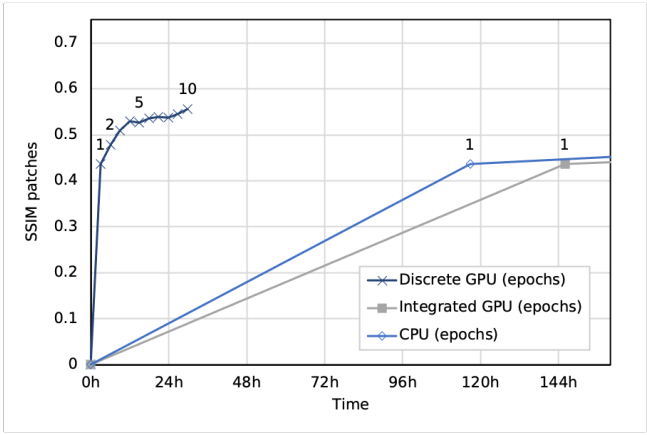

Figure 5.26: SSIM patches over time

In terms of computational cost, the discrete GPU performs 50x faster than the integrated GPU and 40x faster than the CPU. Having the CPU offering even higher speed than the integrated GPU highlights the limitations of such component for this kind of deep learning problems. Regardless, requiring 146 hours (integrated GPU) or 117 hours (CPU) to complete one epoch is likely to be below the minimum requirements of a real use case scenario.

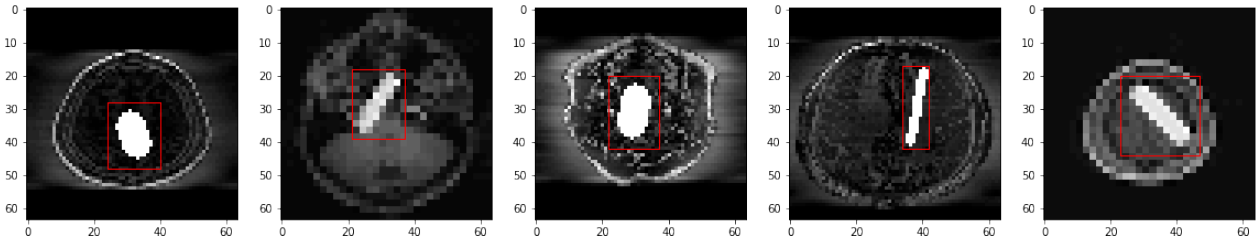

Figure 5.27: Input slices with the patched area highlighted

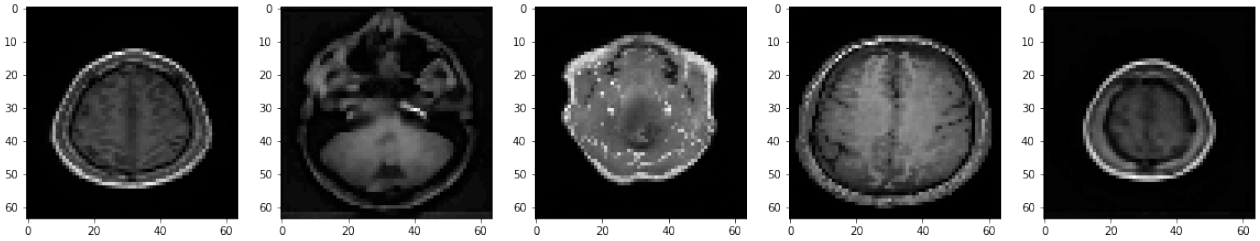

Figure 5.28: Output after 50 epochs

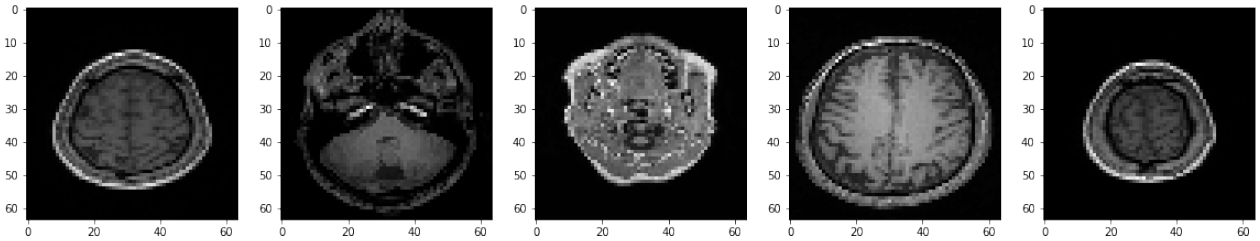

Figure 5.29: Original slices (ground truth)

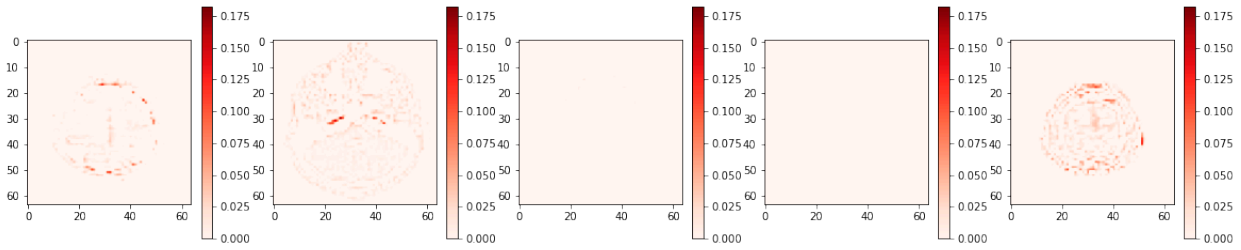

Figure 5.30: Difference between reconstructed slices and ground truth

5.2 Low–requirements scenario: 32 x 32 pixel slices

The experimentations carried out in section 5.1, when applied to 32 x 32 pixel slices, produce variable results in terms of metrics performance depending on the specific experiment, but with a substantial reduction in the training times. Below the most relevant ones:

5.2.1 581 samples (original dataset)

5.2.1.1 51 central slices

When using input slices of 32 x 32 pixels, the results are poor (as opposite as when using slices of 64 x 64 pixels): SSIM of ≈ 0.66 for slices and ≈ 0.4 for patches, highlighting the amount of information that is lost when downsizing images from 64 x 64 to 32 x 32 pixels and the need of additional training data to reach acceptable levels of quality, as described in 5.2.2. Training time in this case, although low (approximately 3.5 hours per epoch in the integrated GPU and the CPU), is not relevant due to the performance obtained. As an example, only the third slice in figure 5.32 seems of good visual quality.

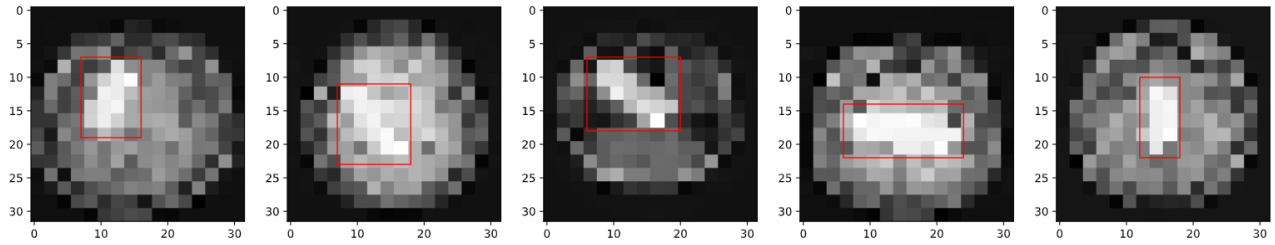

Figure 5.31: Input slices with the patched area highlighted

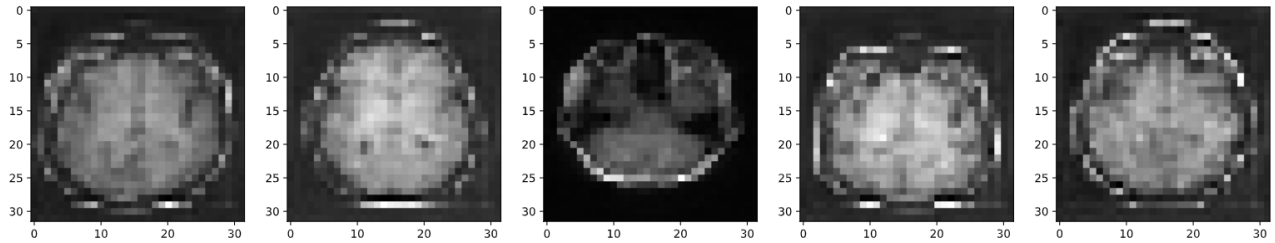

Figure 5.32: Output after 100 epochs

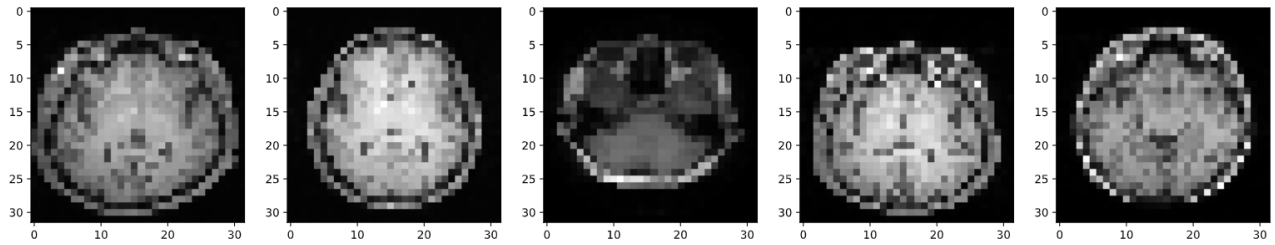

Figure 5.33: Original slices (ground truth)

5.2.2 1743 samples (augmented dataset)

The use of data augmentation addresses the additional issues caused by using low–resolution images and enables the model to handle noise, as shown below:

5.2.2.1 51 central slices

Compared to the maximum SSIM obtained with the original dataset (≈ 0.66 for slices, ≈ 0.4 for patches), the use of augmented data improves this result already after the first epoch: ≈ 0.81 for slices and ≈ 0.45 for patches, and within a reasonable time when executed on the integrated GPU or CPU (13 and 15 hours). The maximum SSIM obtained is ≈ 0.88 for the slices and ≈ 0.65 for the patches.

The behavior observed with regards to the time required to complete an epoch (or the whole training), is that the integrated GPU outperforms the CPU when the input image size is 32 x 32 pixels; and the opposite if the input image size is 64 x 64 pixels.

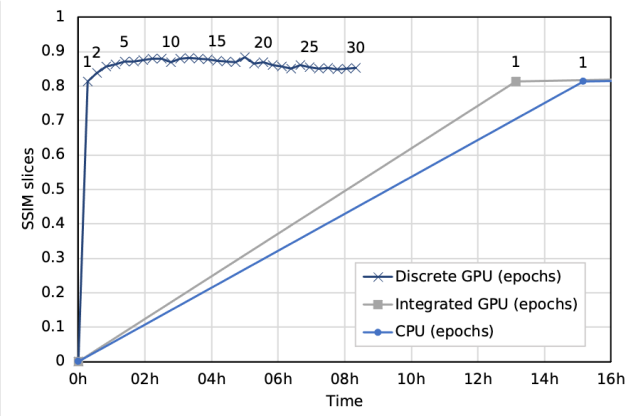

Figure 5.34: SSIM slices over time

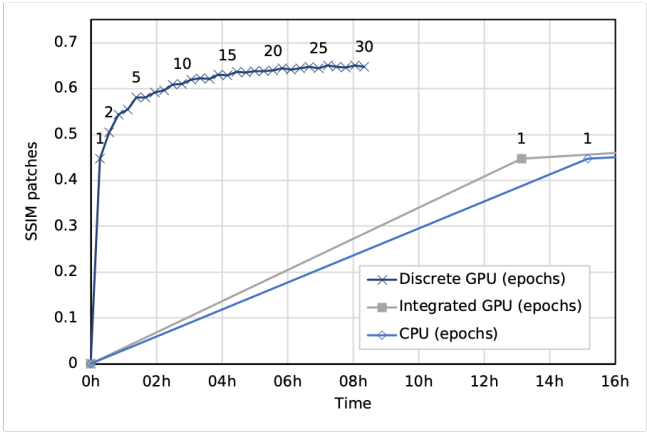

Figure 5.35: SSIM patches over time

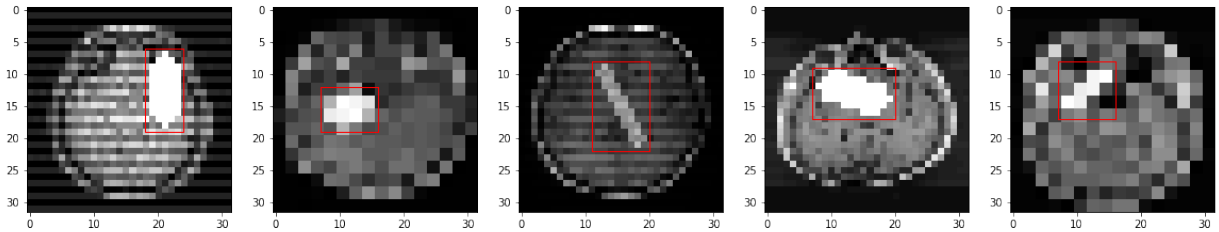

Figure 5.36: Input slices with the patched area highlighted

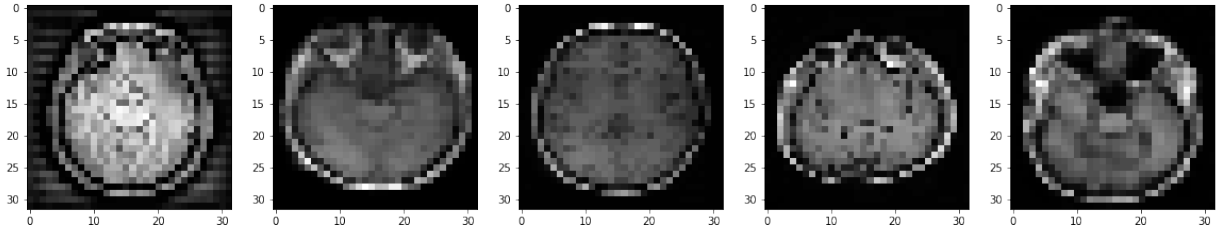

Figure 5.37: Output after 30 epochs

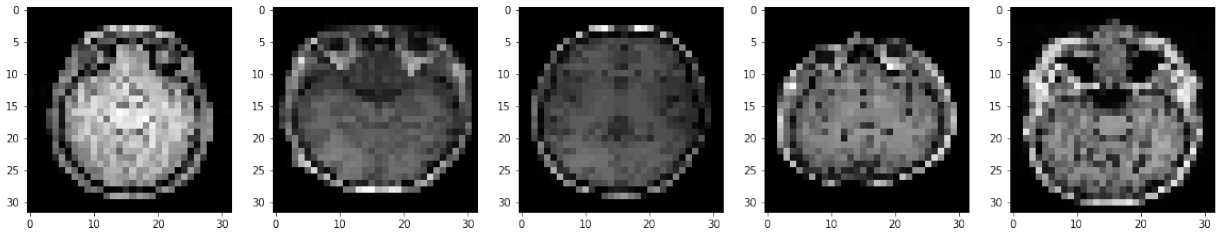

Figure 5.38: Original slices (ground truth)

From the samples shown, only the first one contains noise surrounding the head, although the rest of the image is well reconstructed. This represent a non–negligible improvement compared to the original dataset without data augmentation, as illustrated in figure 5.32.

5.2.2.2 151 central slices

The augmented dataset not only overcomes the reconstruction challenge for the 51 central slices –as described above–, but also addresses the complexity introduced by the use of 151 central slices. Although the time required by the integrated GPU and the CPU to complete one epoch is high (39 and 45 hours), it already delivers acceptable results: SSIM slices ≈ 0.79 and SSIM

patches ≈ 0.47. The slices only require 6 epochs to obtain the maximum performance (SSIM ≈ 0.85), whilst the patches require 13 (SSIM ≈ 0.61).

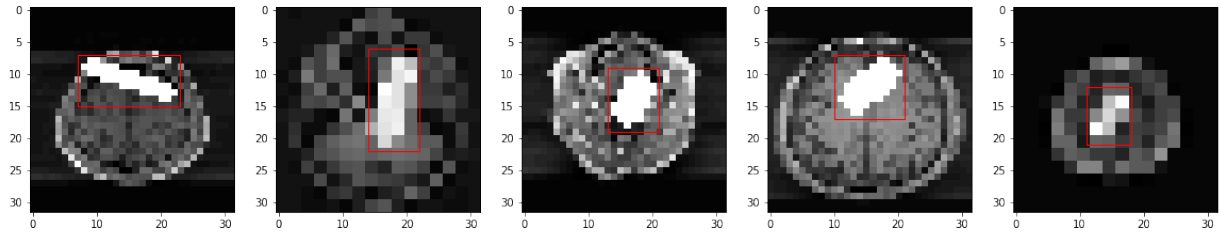

Figure 5.39: Input slices with the patched area highlighted

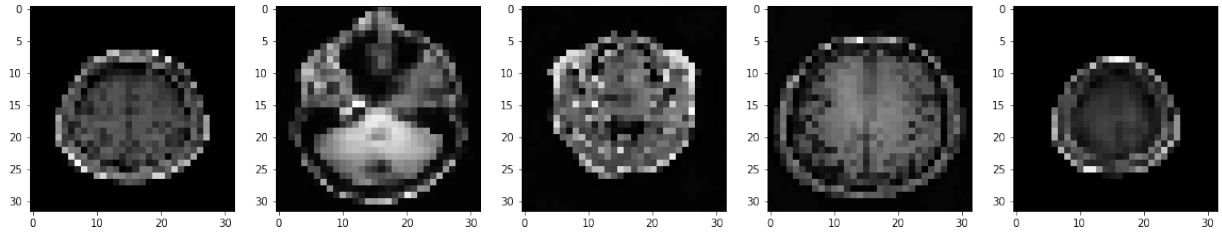

Figure 5.40: Output after 13 epochs

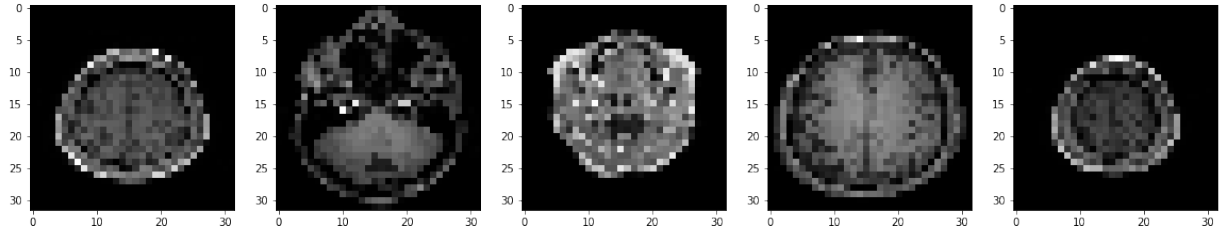

Figure 5.41: Original slices (ground truth)

5.3 Other scenarios

5.3.1 100 samples (reduced dataset)

This experiment proves that even with a limited amount of samples, and without applying any data augmentation technique, good results can be obtained at least for the overall slices. It requires however a considerable amount of epochs, especially if compared with the previously

presented scenarios. This allows nevertheless the integrated GPU or the CPU to achieve the training in reasonable time: for 64 x 64 pixel slices, one epoch requires around 3 hours to complete; SSIM slices ≈ 0.7 is reached after 3 epochs. Unfortunately SSIM patches requires many more iterations, and even then only a value of ≈ 0.47 can be obtained.

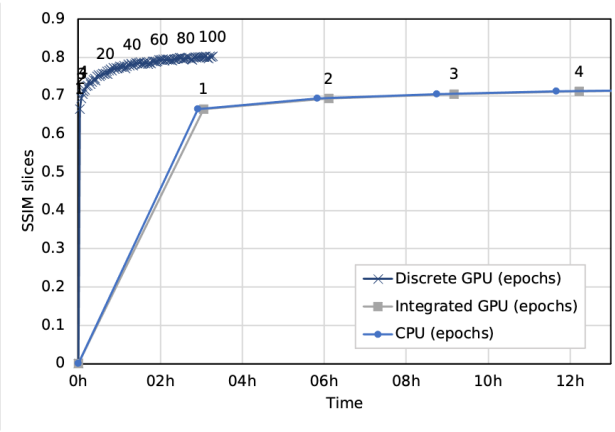

Figure 5.42: SSIM slices over time

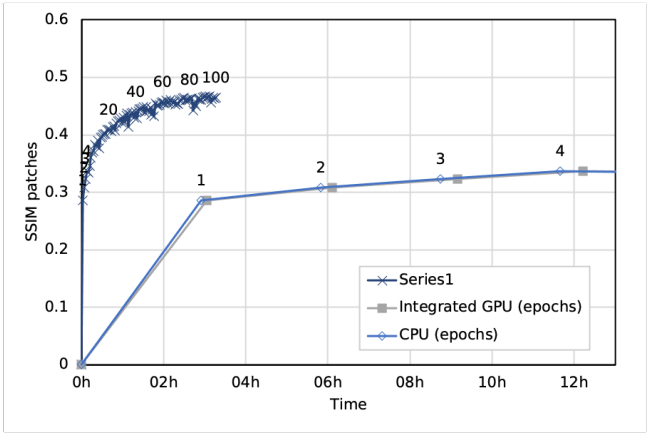

Figure 5.43: SSIM patches over time

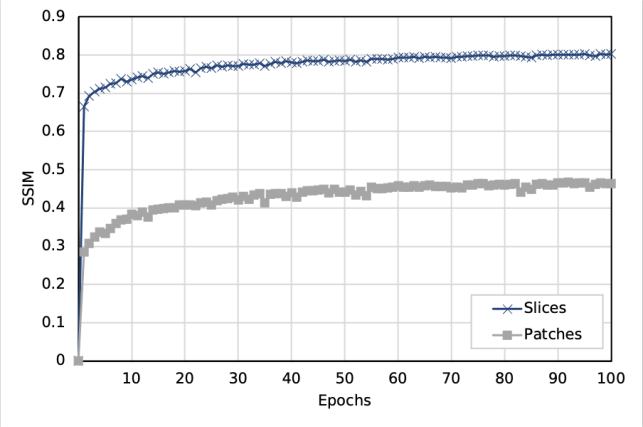

Figure 5.44: SSIM

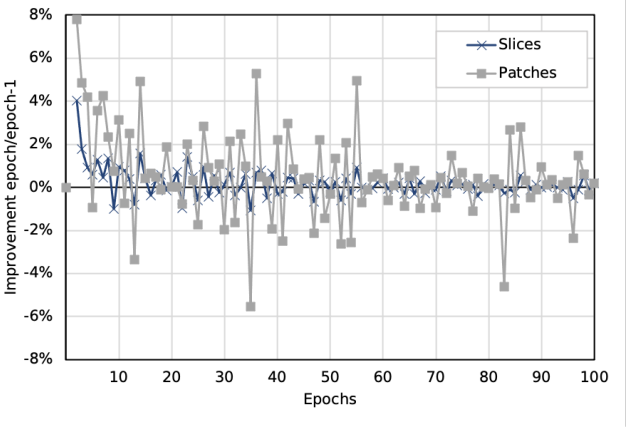

Figure 5.45: SSIM change epoch/epoch–1

This is an example where the use of the CPU or the integrated GPU provides good results although with limitations. The figures below show the result after approximately 12 hours of training with the integrated GPU or the CPU (4 epochs, figure 5.47) and with 3 hours on the discrete GPU (100 epochs, figure 5.48).

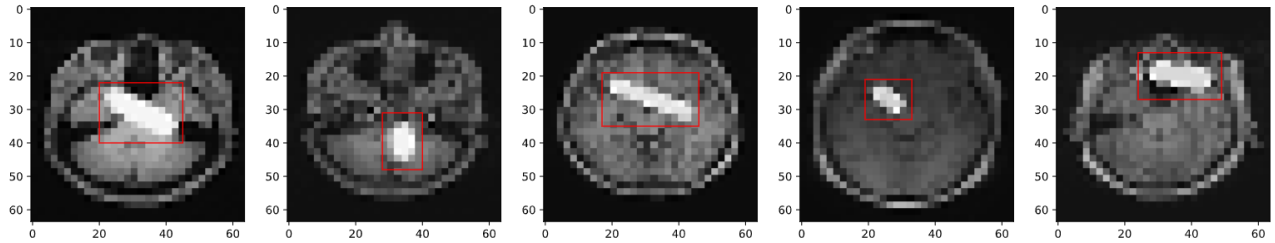

Figure 5.46: Input slices with the patched area highlighted

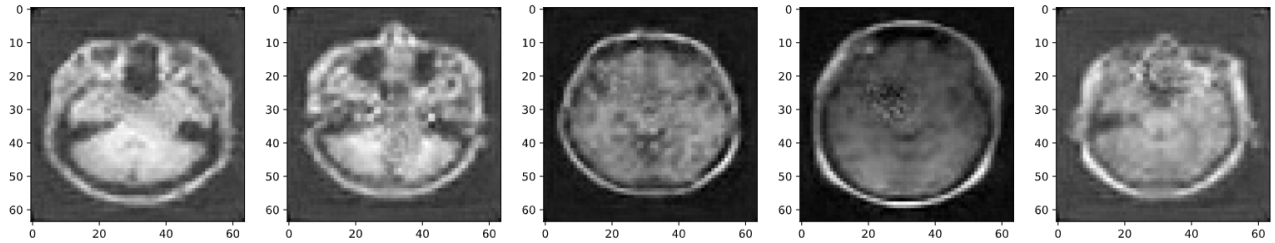

Figure 5.47: Output after 4 epochs

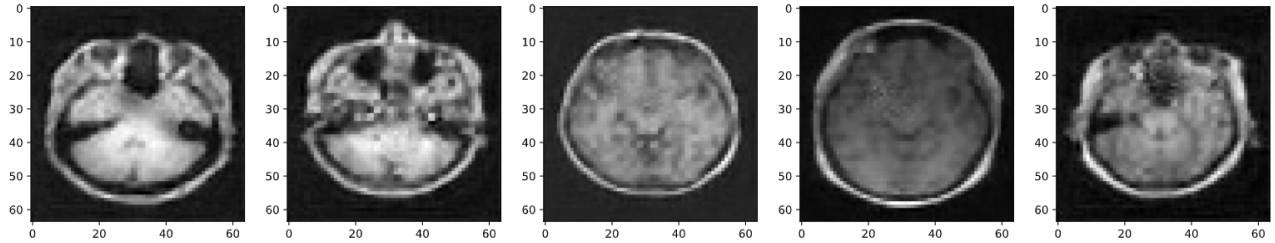

Figure 5.48: Output after 100 epochs

When using input slices of 32 x 32 pixels, the results are poor as described in 5.2.1.1: SSIM of $\approx$ 0.58 for slices and $\approx$ 0.36 for patches. Training time is also low in this case, (approximately 37 minutes per epoch in the integrated GPU and the CPU), but not relevant due to the performance obtained.

5.3.2 128 x 128 pixel slices

As proven when testing the model with input images of 32 x 32 pixels, downsizing input images heavily reduces the amount of information available to the the model, and must be compensated with additional input data to achieve comparable results. In the same manner, having

input images twice the size (128 x 128 pixels) multiplies the amount of information available for the same number of MRI or slices. This allows the model to produce better results with less slices and epochs. The SSIM achieved for the slices is ≈ 0.89 and for the patches ≈ 0.55. Since it has used the reduced dataset, with the original or the augmented dataset , even better results could be achieved, especially with regards to inpainting.

This comes, however, with a cost: training time is 5 times longer with the discrete GPU, which could limit its application with extensive datasets. Considering the performance shown by the integrated GPU and the CPU, the use of this image size can be discarded.

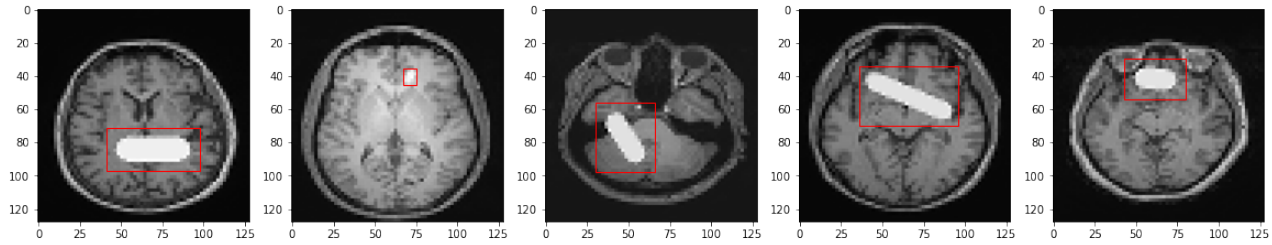

Figure 5.49: Input slices with the patched area highlighted

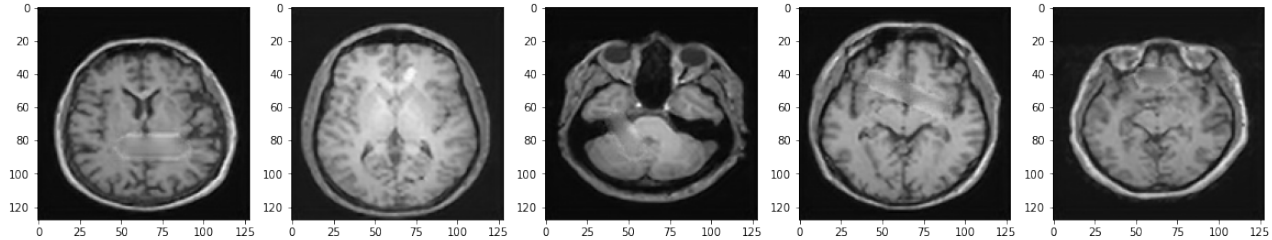

Figure 5.50: Output after 20 epochs

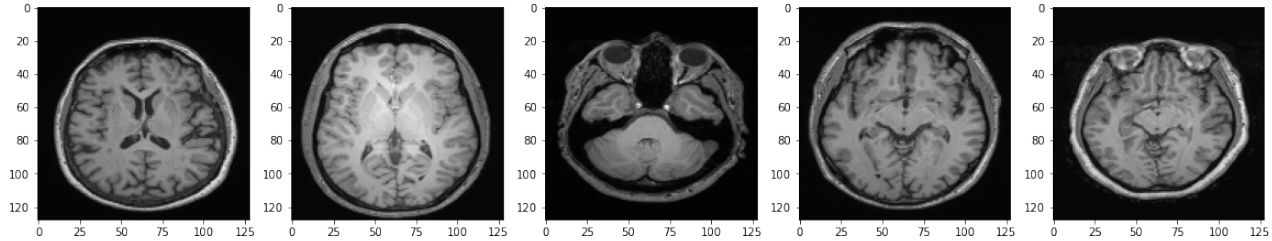

Figure 5.51: Original slices (ground truth)

5.3.3 Sagittal and coronal axes

The experiments done at 64 x 64 pixels resolution, with the reduced dataset (only 100 MRI) and using the 51 central slices, show a level of performance slightly lower than the observed in 5.3.1 but comparable: in the case of the sagittal axis, the SSIM for slices ≈ 0.74 of and SSIM for patches of ≈ 0.43 after 50 epochs.

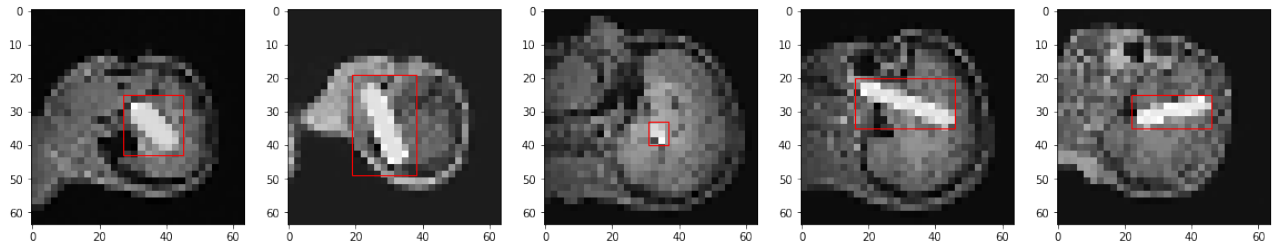

Figure 5.52: Input slices with the patched area highlighted (sagittal axis)

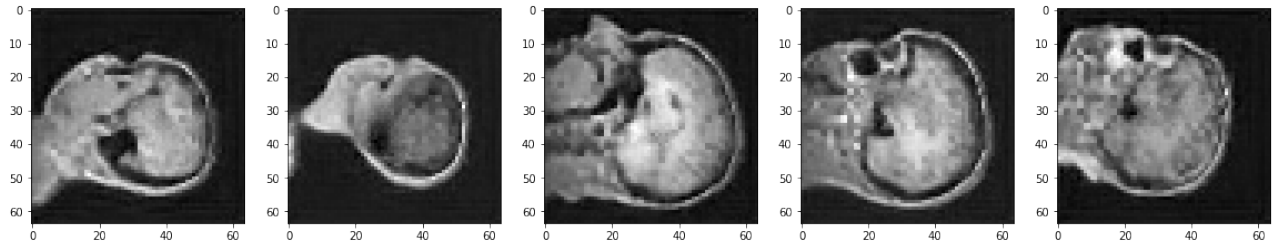

Figure 5.53: Output after 50 epochs (sagittal axis)

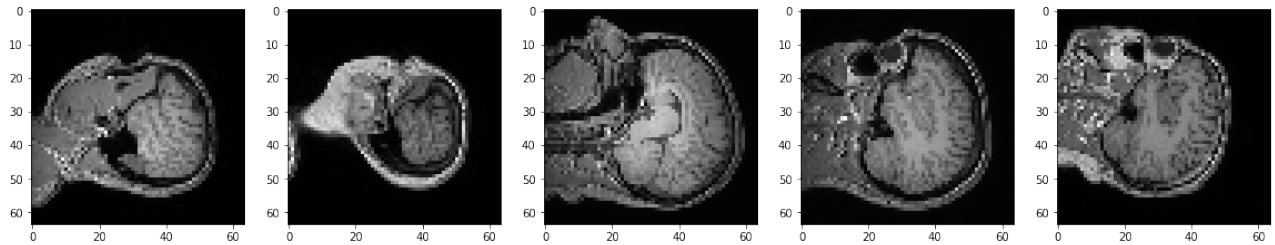

Figure 5.54: Original slices (ground truth – sagittal axis)

The reconstruction of the coronal axis delivers a SSIM for slices ≈ 0.7 and SSIM for patches ≈ 0.4 after 50 epochs.

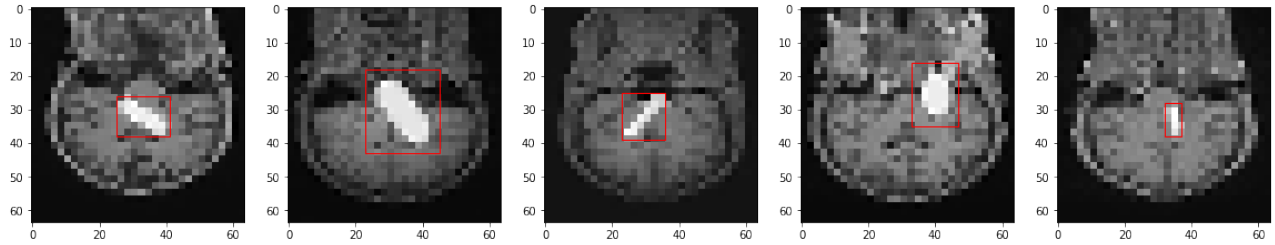

Figure 5.55: Input slices with the patched area highlighted (coronal axis)

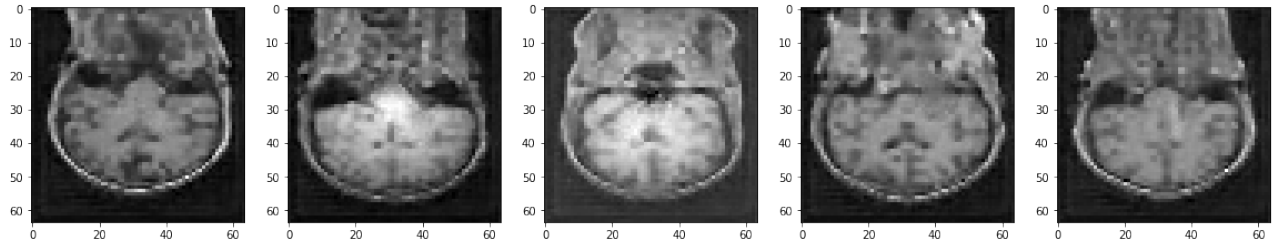

Figure 5.56: Output after 50 epochs (coronal axis)

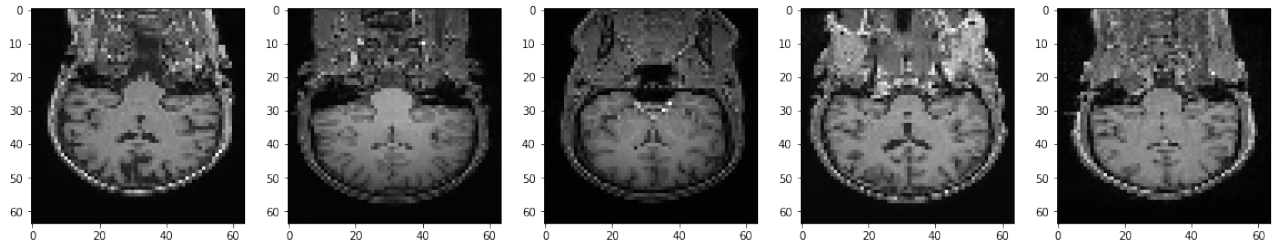

Figure 5.57: Original slices (ground truth – coronal axis)

With the additional tests just described, the model has proven to be suitable for different kind of problems related to brain MRI.

5.4 Metrics summary

The tables below summarise the experiments performed, highlighting (where applicable) the best results for each test case. As a reminder, the original dataset contains 581 MRI scans, the augmented dataset contains 1743 MRI scans and the reduced dataset contains 100 MRI scans.

The number of epochs varies depending on the objective of the experiment and the improvement trend observed, but it is sufficient to allow the comparison between the different cases.

dataset	# slices	SSIM slices	SSIM patches	PSNR slices	PSNR patches	MSE slices	MSE patches	h/epoch D. GPU	h/epoch I. GPU	h/epoch CPU
original	50	0.85	0.57	46.2	42.6	2.3×10^{-8}	5.6×10^{-6}	0.23	16.43	14.61
original	150	0.6	0.37	41.2	41.3	1.7×10^{-5}	2.5×10^{-5}	1.08	na	na
augm.	51	0.91	0.65	50.01	43.66	8.2×10^{-7}	9×10^{-6}	0.98	47.91	38.13
augm.	151	0.77	0.56	41.2	40.9	9.9×10^{-5}	9×10^{-5}	2.97	145.97	116.77

Table 5.1: Metrics for the main scenario (64 x 64 pixel images)

dataset	# slices	SSIM slices	SSIM patches	PSNR slices	PSNR patches	MSE slices	MSE patches	h/epoch D. GPU	h/epoch I. GPU	h/epoch CPU
original	51	0.66	0.4	38.8	38.6	2.4×10^{-6}	2.7×10^{-5}	0.14	3.37	3.87
original	151	0.58	0.34	39.74	40.91	1.8×10^{-5}	1.9×10^{-5}	0.26	na	na
augm.	51	0.88	0.65	44.4	41.3	8.3×10^{-6}	1.6×10^{-5}	0.28	13.15	15.17
augm.	151	0.85	0.61	44.4	43.8	1.3×10^{-5}	1.2×10^{-5}	0.58	38.92	44.91

Table 5.2: Metrics for the low–requirements scenario (32 x 32 pixel images)

image size	# slices	SSIM slices	SSIM patches	PSNR slices	PSNR patches	MSE slices	MSE patches	h/epoch D. GPU	h/epoch I. GPU	h/epoch CPU
128×128	51	0.89	0.55	48.4	42.8	4.7×10^{-7}	1.3×10^{-5}	0.14	na	na
64×64	51	0.8	0.47	44	40.32	6.3×10^{-6}	7.5×10^{-5}	0.03	3.06	2.94
32×32	51	0.59	0.36	39.8	37.93	2.7×10^{-6}	6.8×10^{-5}	0.02	0.62	0.67

Table 5.3: Metrics for the reduced dataset scenario (100 MRI scans)

Chapter 6

Conclusion and future work

This work analysed in depth the U-net network and adapted it to the needs of image reconstruction tasks. It proved to be a versatile architecture, since it has been able to address different challenges such as image inpainting, super–resolution, Gibb's ringing and ghosting noises and the three axes (or anatomical planes) by only adapting the training data, i.e. using data augmentation techniques.

Numerous tests were performed to assess the time required under the different configurations (discrete GPU, integrated GPU, CPU), and based on the outcome, acknowledge the limitations of some of those configurations, making them not suitable for all the problems, especially those involving high volume of training data or high–resolution images.

Areas for future work are using the model with MRI containing real lesions and artifacts, and assess its capacity to identify them; the combination of additional issues, such as super–resolution and artificial noise; the reconstruction of 3D volumes instead of 2D slices; or the use of other neural network architectures, such as evolutions of U-net or GANs.

Bibliography

[1] Nifti-1 data format - neuroimaging informatics technology initiative. URL: https://nifti.nimh.nih.gov/nifti-1.

[2] What is python? executive summary. URL: https://www.python.org/doc/essays/blurb/.

[3] Documentation for visual studio code, Apr 2016. URL: https://code.visualstudio.com/docs.

[4] Mri scan - how it's performed, Aug 2018. URL: https://www.nhs.uk/conditions/mri-scan/what-happens/.

[5] Anaconda software distribution, 2020. URL: https://docs.anaconda.com/.

[6] Classification: Roc curve and auc | machine learning crash course, Feb 2020. URL: https://developers.google.com/machine-learning/crash-course/classification/roc-and-auc.

[7] Jupyter - visual studio marketplace, Nov 2020. URL: https://marketplace.visualstudio.com/items?itemName=ms-toolsai.jupyter.

[8] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org. URL: https://www.tensorflow.org/.

[9] H. Akrami, A. A. Joshi, J. Li, S. Aydore, and R. M. Leahy. Brain lesion detection using a robust variational autoencoder and transfer learning. In *2020 IEEE 17th International Symposium on Biomedical Imaging (ISBI)*, pages 786–790, April 2020. doi:10.1109/ISBI45749.2020.9098405.

[10] Karim Armanious, Chenming Jiang, Marc Fischer, Thomas Küstner, Konstantin Nikolaou, Sergios Gatidis, and Bin Yang. Medgan: Medical image translation using gans, 2019. arXiv:1806.06397.

[11] Karim Armanious, Youssef Mecky, Sergios Gatidis, and Bin Yang. Adversarial inpainting of medical image modalities. *ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, May 2019. URL: http://dx.doi.org/10.1109/ICASSP.2019.8682677, doi:10.1109/icassp.2019.8682677.

[12] Hans E. Atlason, Askell Love, Sigurdur Sigurdsson, Vilmundur Gudnason, and Lotta M. Ellingsen. Unsupervised brain lesion segmentation from mri using a convolutional autoencoder. *Medical Imaging 2019: Image Processing*, Mar 2019. URL: http://dx.doi.org/10.1117/12.2512953, doi:10.1117/12.2512953.

[13] Imperial College London Biomedical Image Analysis Group. Ixi dataset. URL: http://brain-development.org/ixi-dataset/.

[14] Matthew Brett, Christopher J. Markiewicz, Michael Hanke, Marc-Alexandre Côté, Ben Cipollini, Paul McCarthy, Dorota Jarecka, Christopher P. Cheng, Yaroslav O. Halchenko, Michiel Cottaar, Satrajit Ghosh, Eric Larson, Demian Wassermann, Stephan Gerhard, Gregory R. Lee, Hao-Ting Wang, Erik Kastman, Jakub Kaczmarzyk, Roberto Guidotti, Or Duek, Ariel Rokem, Cindee Madison, Félix C. Morency, Brendan Moloney, Mathias Goncalves, Ross Markello, Cameron Riddell, Christopher Burns, Jarrod Millman, Alexandre Gramfort, Jaakko Leppäkangas, Anibal Sólon, Jasper J.F. van den Bosch, Robert D. Vincent, Henry Braun, Krish Subramaniam, Krzysztof J. Gorgolewski, Pradeep Reddy Raamana, B. Nolan Nichols, Eric M. Baker, Soichi Hayashi, Basile Pinsard, Christian Haselgrove, Mark Hymers, Oscar Esteban, Serge Koudoro, Nikolaas N. Oosterhof, Bago Amirbekian, Ian Nimmo-Smith, Ly Nguyen, Samir Reddigari, Samuel St-Jean, Egor Panfilov, Eleftherios Garyfallidis, Gael Varoquaux, Jon Haitz Legarreta, Kevin S. Hahn, Oliver P. Hinds, Bennet Fauber, Jean-Baptiste Poline, Jon Stutters, Kesshi Jordan, Matthew Cieslak, Miguel Estevan Moreno, Valentin Haenel, Yannick Schwartz, Zvi Baratz, Benjamin C Darwin, Bertrand Thirion, Dimitri Papadopoulos Orfanos, Fernando Pérez-García, Igor Solovey, Ivan Gonzalez, Jath Palasubramaniam, Justin Lecher, Katrin Leinweber, Konstantinos Raktivan, Peter Fischer, Philippe Gervais, Syam Gadde,

Thomas Ballinger, Thomas Roos, Venkateswara Reddy Reddam, and freec84. nipy/nibabel: 3.1.0, April 2020. URL: https://doi.org/10.5281/zenodo.3757992, doi:10.5281/zenodo.3757992.

[15] Oates Briony J. *Researching Information Systems and Computing.* SAGE Publications Ltd, 2005. URL: https://ezproxy.biblioteca-uoc.idm.oclc.org/login?url=http://search.ebscohost.com/login.aspx?direct=true&db=nlebk&AN=1099431&lang=es&site=ehost-live&scope=site.

[16] Mara Cercignani, Matilde Inglese, Malgorzata Siger-Zajdel, and Massimo Filippi. Segmenting brain white matter, gray matter and cerebro-spinal fluid using diffusion tensor-mri derived indices. *Magnetic Resonance Imaging*, 19(9):1167–1172, 2001. URL: https://www.sciencedirect.com/science/article/pii/S0730725X0100457X, doi:https://doi.org/10.1016/S0730-725X(01)00457-X.

[17] Xiaoran Chen and Ender Konukoglu. Unsupervised detection of lesions in brain MRI using constrained adversarial auto-encoders. *CoRR*, abs/1806.04972, 2018. URL: http://arxiv.org/abs/1806.04972, arXiv:1806.04972.

[18] Alex Clark. Pillow (pil fork) documentation, 2015. URL: https://buildmedia.readthedocs.org/media/pdf/pillow/latest/pillow.pdf.

[19] IBM Education. What is machine learning?, Jul 2020. URL: https://www.ibm.com/cloud/learn/machine-learning.

[20] AD Elster. Ghosting. URL: http://www.mriquestions.com/ghosting.html.

[21] AD Elster. Gibbs artifact. URL: http://www.mriquestions.com/gibbs-artifact.html.

[22] Jordi Girones, Jordi Casas, Julia Minguillon, and Ramón Caihuelas. *Mineria de datos, modelos y algoritmos.* Editorial UOC, 2017.

[23] Aurélien Géron. *Hands-on machine learning with Scikit-Learn and TensorFlow : concepts, tools, and techniques to build intelligent systems.* O'Reilly Media, Sebastopol, CA, 2017.

[24] Charles R. Harris, K. Jarrod Millman, Stéfan J van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585:357–362, 2020. doi:10.1038/s41586-020-2649-2.

[25] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007. doi:10.1109/MCSE.2007.55.

[26] Apple Inc. Metal. URL: https://developer.apple.com/metal/.

[27] Apple Inc. Opencl for macos. URL: https://developer.apple.com/opencl/.

[28] B.Sc. Ingram, April D. Magnetic resonance imaging. *Salem Press Encyclopedia of Science*, 2018. URL: http://search.ebscohost.com/login.aspx?direct=true&db=ers&AN=89250510&authtype=sso&custid=uoc&site=eds-live&scope=site&custid=uoc&authtype=ip,sso.

[29] Research Imaging Institute. Mango, Mar 2019.

[30] Intel. URL: https://plaidml.github.io/plaidml/.

[31] Saad Jbabdi. URL: https://users.fmrib.ox.ac.uk/~saad/ONBI/ONBI-Fourier_Practical_python.html.

[32] Graham Lloyd-Jones. Mri interpretation, Sep 2017. URL: https://www.radiologymasterclass.co.uk/tutorials/mri/t1_and_t2_images.

[33] José V. Manjón, José E. Romero, Roberto Vivo-Hernando, Gregorio Rubio, Fernando Aparici, Maria de la Iglesia-Vaya, Thomas Tourdias, and Pierrick Coupé. Blind mri brain lesion inpainting using deep learning. In Ninon Burgos, David Svoboda, Jelmer M. Wolterink, and Can Zhao, editors, *Simulation and Synthesis in Medical Imaging*, pages 41–49, Cham, 2020. Springer International Publishing.

[34] Wes McKinney et al. Data structures for statistical computing in python. In *Proceedings of the 9th Python in Science Conference*, volume 445, pages 51–56. Austin, TX, 2010.

[35] Zaccharie Ramzi, Philippe Ciuciu, and Jean-Luc Starck. Benchmarking mri reconstruction neural networks on large public datasets. *Applied Sciences*, 10(5), 2020. URL: https://www.mdpi.com/2076-3417/10/5/1816, doi:10.3390/app10051816.

[36] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In Nassir Navab, Joachim Hornegger, William M. Wells, and Alejandro F. Frangi, editors, *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*, pages 234–241, Cham, 2015. Springer International Publishing.

[37] Thomas Schlegl, Philipp Seeböck, Sebastian M. Waldstein, Ursula Schmidt-Erfurth, and Georg Langs. Unsupervised anomaly detection with generative adversarial networks to guide marker discovery, 2017. arXiv:1703.05921.

[38] Aditya Sharma. Reconstructing brain mri images, Jun 2018. URL: https://www.datacamp.com/community/tutorials/reconstructing-brain-images-deep-learning.

[39] Kevin Shen. Effect of batch size on training dynamics, Jun 2018. URL: https://medium.com/mini-distill/effect-of-batch-size-on-training-dynamics-21c14f7a716e.

[40] Jupyter Team. The jupyter notebook, May 2020. URL: https://jupyter-notebook.readthedocs.io/en/latest/.

[41] Keras Team. Keras documentation: About keras. URL: https://keras.io/about/.

[42] Stefan Van der Walt, Johannes L Schönberger, Juan Nunez-Iglesias, François Boulogne, Joshua D Warner, Neil Yager, Emmanuelle Gouillart, and Tony Yu. scikit-image: image processing in python. *PeerJ*, 2:e453, 2014.

[43] Zhou Wang, A.C. Bovik, H.R. Sheikh, and E.P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13(4):600–612, 2004. doi:10.1109/TIP.2003.819861.

[44] Muñoz-Maniega S. Wardlaw JM, Valdés Hernández MC. What are white matter hyperintensities made of? relevance to vascular cognitive impairment. *Journal of the American Heart Association*, 2015. doi:10.1161/JAHA.114.001140.

[45] Rick Wierenga. An empirical comparison of optimizers for machine learning models, Apr 2021. URL: https://heartbeat.fritz.ai/an-empirical-comparison-of-optimizers-for-machine-learning-models-b86f29957050.

[46] Lianrui Zuo, Blake E. Dewey, Aaron Carass, Yufan He, Muhan Shao, Jacob C. Reinhold, and Jerry L. Prince. Synthesizing realistic brain mr images with noise control. In Ninon Burgos, David Svoboda, Jelmer M. Wolterink, and Can Zhao, editors, *Simulation and Synthesis in Medical Imaging*, pages 21–31, Cham, 2020. Springer International Publishing.