DEEP LEARNING FOR CORPUS CALLOSUM SEGMENTATION IN BRAIN MAGNETIC RESONANCE IMAGES

Flávio Henrique Schuindt da Silva

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia de Sistemas e Computação.

Orientador: Ricardo Cordeiro de Farias

Rio de Janeiro
Março de 2018

DEEP LEARNING FOR CORPUS CALLOSUM SEGMENTATION IN BRAIN MAGNETIC RESONANCE IMAGES

Flávio Henrique Schuindt da Silva

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Examinada por:

Prof. Ricardo Cordeiro de Farias, PhD.

Prof. Felipe Maia Galvão França, PhD.

Prof. Marcelo Panaro de Moraes Zamith, D.Sc.

RIO DE JANEIRO, RJ – BRASIL

MARÇO DE 2018

Silva, Flávio Henrique Schuindt da

Deep Learning for Corpus Callosum Segmentation in Brain Magnetic Resonance Images / Flávio Henrique Schuindt da Silva. – Rio de Janeiro: UFRJ / COPPE, 2018.

XVI, 105 p.: il.; 29, 7cm.

Orientador: Ricardo Cordeiro de Farias

Dissertação (mestrado) – UFRJ / COPPE / Programa de Engenharia de Sistemas e Computação, 2018.

Referências Bibliográficas: p. 80 – 85.

1. Deep Learning. 2. Machine Learning. 3. Healthcare. 4. Brain. 5. MRI. 6. U-Net. 7. Image Segmentation. 8. Python. 9. Tensorflow. 10. Keras. I. Farias, Ricardo Cordeiro de. II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

Aos meus pais, Joaquim (in memoriam) e Elisabeth, que nos seus mais possíveis e impossíveis esforços, tentaram dar-me a melhor educação possível.

À minha amada, Yasmin, que na sua infinita bondade e paciência pela minha ausência nas horas despendidas nesse trabalho tornou-o possível.

Ao meu irmão, Pedro, por todos os momentos de diversão e distração proporcionados ao longo dessa jornada.

To my parents, Joaquim (in memoriam) and Elisabeth, who in their most possible and impossible efforts tried to give the best education possible.

To my beloved, Yasmin, who in his infinite goodness and patience for my absence in the hours spent in this work made it possible.

To my brother, Pedro, for all the moments of fun and distraction provided throughout this journey.

Agradecimentos

Primeiramente, aos meus pais e meu irmão, pois sem o apoio e suporte deles não seria possível esta jornada.

À minha querida e amada Yasmin, meu muito obrigado por toda paciência e por estar ao meu lado em todos esses momentos. Sem você, seria muito mais difícil.

Ao orientador e amigo, Prof. Ricardo Farias, que aceitou entrar nesta jornada e ajudou no que foi possível na remoção das pedras que apareceram pelo caminho. Obrigado pela cessão incrível da Lotus, sem a qual não seria possível treinar os modelos.

Ao MediaLab da Universidade Federal Fluminense (UFF), pela cessão de máquinas com GPU para treinamento.

Aos professores Andrew Ng, Andrej Karpathy, Fei-Fei Li, Justin Johson por produzirem conteúdos incríveis e deixá-los abertos ao mundo com intuito de disseminar conhecimento.

Thanks

First, to my parents and my brother, because without their support, this journey would not be possible.

To my dear and beloved Yasmin, my thanks for all the patience and for being by my side in all those moments. Without you, it would be much harder.

To my advisor and friend, Prof. Ricardo Farias, who accepted to enter this journey and helped in what was possible in this long path. Thanks for the incredible assignment of Lotus, without which it would not be possible to train the models

To MediaLab, from Universidade Federal Fluminense (UFF), for the assignment of machines with GPU for training.

To professors Andrew Ng, Andrej Karpathy, Fei-Fei Li, Justin Johson for producing amazing content and leaving them open to the world in order to disseminate knowledge.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

APRENDIZADO PROFUNDO PARA SEGMENTAÇÃO DO CORPO CALOSO EM IMAGENS DE RESSONÂNCIA MAGNÉTICA DO CÉREBRO

Flávio Henrique Schuindt da Silva

Março/2018

Orientador: Ricardo Cordeiro de Farias

Programa: Engenharia de Sistemas e Computação

Apresentamos neste trabalho um novo método para segmentar o Corpo Caloso em imagens de ressonância magnética (MRI) usando U-Net, uma rede puramente convolucional. Treinamos a U-Net usando dois datasets públicos e validamos o modelo treinado em um conjunto de teste também obtido a partir destes datasets públicos. Os resultados são obtidos realizando comparações usando o Índice de Similaridade Estrutural (SSIM) e o coeficiente Dice entre a imagem gabarito e a imagem gerada pelo modelo.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

DEEP LEARNING FOR CORPUS CALLOSUM SEGMENTATION IN BRAIN MAGNETIC RESONANCE IMAGES

Flávio Henrique Schuindt da Silva

March/2018

Advisor: Ricardo Cordeiro de Farias

Department: Systems Engineering and Computer Science

In this work we present a novel method to segment Corpus Callosum in Magnetic Resonance Images (MRI) using U-Net, a Fully Convolutional Neural Network. We trained the U-Net using two public datasets and evaluated the trained model in a test set also obtained from these two public datasets. Results are obtained making comparisons using the Structural Similarity Index (SSIM) and Dice Coefficient between the Ground Truth and the Predicted image.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The human brain is one of the most complex system that one can find in the nature. The average human brain has about 100 billion neurons and each of these neurons has in average connection with 10000 other neurons. It leads to about of 1000 trillion connections overall. Inside brain, there is some important structures. One of them is the Corpus Callosum (CC). The corpus callosum is a large bundle of nerve fibres that connects the left and right brain hemispheres. It's the largest collection of white matter within the brain, allowing communication between hemispheres.

Consisting of over 190 million axons, CC is hypothesized to play a fundamental role in integrating information and mediating complex behaviors. The lack of normal callosal development can lead to deficits in functional connectivity that are related to impairments in specific cognitive domains. Weakened integrity of the CC directly contributes to a decline in cognitive function in aging adults whereas increased callosal thickness in typical childhood development correlates with intelligence, processing speed and problem solving abilities. Last but not least, there is also a growing body of literature reporting that subtle structural changes in the corpus callosum may correlate with cognitive and behavioral deficits in neurodevelopmental disorders including autism schizo-phrenia and attention-deficit disorder [16]. The decreases in corpus callosum size in schizophrenia varys directly with length of illness, perhaps indicative of a progressive process [17]. CC also plays an important role in binocular vision, allowing the human being to perceive depth. Aside from diseases, in MR exams, there is one important landmark called ACPC line that acts as a convenient standard by the neuroimaging community,

and in most instances is the reference plane for axial imaging [18]. Corpus Callosum is the first reference to obtain ACPC. Figure 1.1 shows the CC and the ACPC.

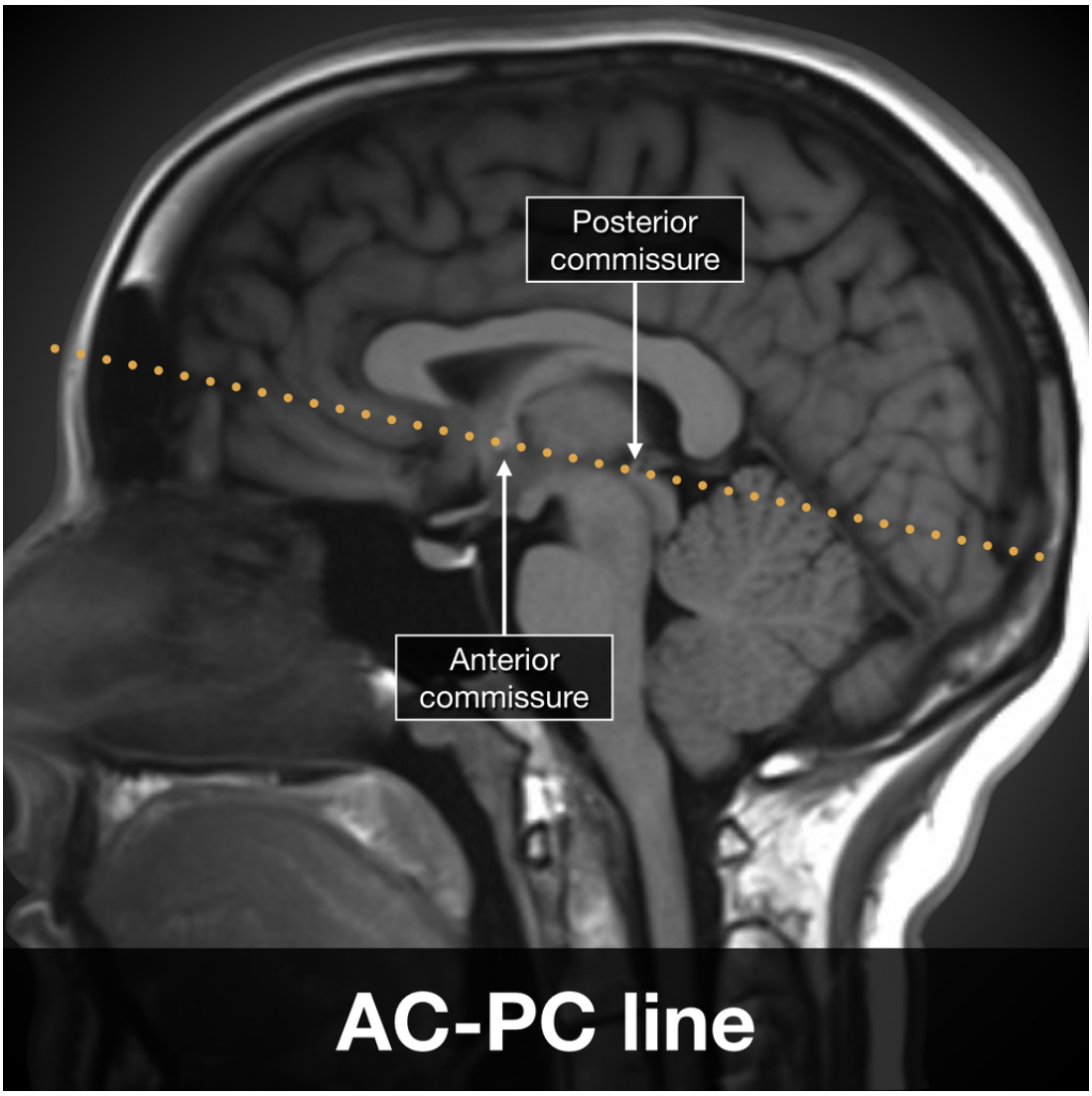


Figure 1.1: AC-PC Line.

Therefore, correct segmentation of the CC in Magnetic Resonance images (MRI) is very useful to diagnoses diseases, allowing doctors to measure the degree of injury in the area and make appropriate decisions quickly. Also, for MRI it plays an important role to detect the AC-PC landmark in the brain.

1.1 Motivation

Deep Learning has been overcoming traditional method in some image tasks like classification, localization, recognition, segmentation and detection. With advances in computational power and dataset extraction, the field has become extremely challenging. Specifically, there is a huge set of applications in the biomedical area, ranging from tissue segmentation (this work) to cancer detection and classification, for example. The main advantage of Deep Learning method is that, different from conventional methods, it can learn by itself the main features extractors presented in the input image, whereas conventional methods relies on handmade features extractors, which often are made by experts in the field and based on previous studies, requiring a significant domain expertise.

1.2 Objective

The objective of this work is to develop a novel and robust system to segment the Corpus Callosum in brain images. While this system is robust, it's trained, validated and tested with a set of only 2003 images, so another main objective is not only develop a robust system, but with restrictions on the dataset size. It avoids the human manual task of segmenting the corpus callosum, saving significant time from specialists, allowing them to concentrate in areas that requires their specific knowledge.

1.3 Methodology

We train a Fully Convolutional Neural Network (FCNN) to segment the Corpus Callosum. The FCNN is the U-Net, a convolutional neural network to biomedical image segmentation [10]. The training is done using two medical datasets annotated by specialists. The first one is the OASIS dataset [19] and the second one is the ABIDE dataset [20]. To avoid bias in the trained model, we split the total dataset images (OASIS+ABIDE) in three different sets: training, validation and test. The sets contain 70%, 10% and 20% of the images, respectively. We adjust the hyper-parameters of the neural network to different configurations, allowing us to get the

best model possible. All of these methods and configurations are detailed described in the methodology chapter.

1.4 Related Works

Image Segmentation is a process where the image is subdivided in its well defined regions. These regions are visually distinct and uniform with respect to some property such as grey level, texture or color and has a semantic meaning to the problem statement, e.g, one can segment the brain in its particular structures like Corpus Callosum, Hippocampus, Cerebellum, etc.

In the literature, many image segmentation techniques were tried to properly segment the corpus callosum in the brain. [21] proposed an unsupervised clustering using K-means that classify each image pixel into K different clusters. In [22] the authors first extract regions satisfying the statistical characteristics of gray level distributions of the corpus callosum, that have relatively high intensity values, and after that combines this result with a search for a region that has the shape of the corpus callosum. In [23] the authors use the watershed morphological approach with predefined marks to perform the segmentation. There are also some approaches using machine learning techniques. The work in [24] presents a machine learning approach for improving active shape model segmentation. It first extract local edge features using steerable filters and uses a machine learning classifier based on AdaBoost [25] to select a small number of critical features that can find optimal displacements for landmarks by searching along the direction perpendicular to each landmark. In [26] they use a training set to construct two models representing the objects-shape model and border appearance model. A two-step approach to image segmentation is reported. In the first step, an approximate location of the object of interest is determined. In the second step, accurate border segmentation is performed. Last, there are some initiatives using Atlas based segmentation [27]. The idea is to use a large dataset where important regions of interest in the brain were well delineated by experts radiologists. This large dataset gives a good understanding about pixels values intensity, edges, etc. With this information, one can get a new MRI image, that is not presented in the dataset, and match it with the Atlas

image and identify the regions of interest. In [28] it is constructed an atlas by clustering the patient images into subgroups of similar images and it is built a mean image from each cluster.

1.5 Deep Learning

As stated in previous section, some approaches relies on the use of Machine Learning based algorithms. Although machine learning algorithms are somehow great in some tasks, they are limited in their ability to process natural data in their raw form. For decades, constructing a pattern-recognition or machine-learning system required careful engineering and considerable domain expertise to design a feature extractor that transformed the raw data, such as the pixel values of an image, into a suitable internal representation or feature vector from which the learning subsystem, often a classifier, could detect or classify patterns from the input data [29]. Deep Learning comes to exactly avoid this feature engineering tasks to be developed by a human being. The idea is that you stack Convolutional Layers, one by one, and these layers are totally responsible for the discover of the representations needed to the detection or classification without any human intervention on the selection of the appropriate features. Deep Learning is a kind of Representational Learning in the sense that each layer is responsible to transform the input data, coming from the previous layer, to a new representation that is easier to be understood, depending on the problem: image classification, segmentation, edge detection, etc. Each layer is a non-linear module that transforms its input data into a representation at a higher and slightly more abstract level. An image, for example, comes in the form of an array of pixel values, and the learned features in the first layer of representation typically represent the presence or absence of edges at particular orientations and locations in the image. The second layer typically detects motifs by spotting particular arrangements of edges, regardless of small variations in the edge positions. The third layer may assemble motifs into larger combinations that correspond to parts of familiar objects, and subsequent layers would detect objects as combinations of these parts [29].

1.6 Dissertation Structure

This dissertation presents in the next two chapters the theoretical foundation. The other chapters are dedicated to Methodology, Results and Discussions, Conclusion and Future work, respectively.

In chapter 2, *Artificial Neural Networks*: we present the theoretical foundation to Neural Networks, beginning with a single Neuron, evolving to Multiple Layer Perceptrons (MLP) and ending the chapter with Convolutional Neural Networks. We also introduce some important Machine Learning concepts.

In chapter 3, *Modern History of Object Recognition*: we go deeper and explore important concepts to a better understanding of the segmentation task.

In chapter 4, *Methodology*: we present our methodology for training neural networks to segment the corpus callosum.

In chapter 5, *Results and Discussions*: the experimental essays, evaluation and discussion are presented.

In chapter 6, *Conclusion and Future Work*: we suggest some improvements to future works.

Chapter 2

Artificial Neural Networks

> The brain is like a muscle. When it is in use we feel very good. Understanding is joyous.
>
> Carl Sagan

One of the most impressive tasks that a human being can do is to recognize and label different objects in a image. We do it effortlessly. How we do it? The answer is that hundreds of million of years of evolution created a complex system in our brain called the virtual cortex. In each hemisphere of our brain, humans have a primary visual cortex, also known as V1, containing 140 million neurons, with tens of billions of connections among them. And yet human vision involves not just V1, but an entire series of visual cortices - V2, V3, V4, and V5 - doing progressively more complex image processing [1]

Consider, for example, a task of recognize handwritten digits like in figure 2.1.

Figure 2.1: Handwritten digits from MNIST dataset.

There are basically two main approaches to solve this task in a computer program. The first one is basically setup a set of rules and apply them, digit by digit. For example, we could say that a "4" has a vertical stroke, one horizontal and another one vertical with half size of the first one. Using this rule in a software, it

can identify digit by digit. The problem with this approach is that we need to write very complex algorithms that can handle all the possible variations, which is a very boring task.

The second approach, using Neural Networks, tries to work on the problem in a different way. Using a huge set of images known as training set, we guide the neural networks to learn how to identify each digit. With the training set size ranging from millions or billions of digits, the network can learn very well how to recognize each digit.

2.1 The Percetron

Before explaining the Neural Networks, we will introduce the most simple artificial neuron unit called Perceptron that was developed in the 1950s and 1960s. The perceptron is very simple and his work is only to calculate a mathematical function using the inputs and evaluate (activate or not) a final response. In the figure 2.2 we can see how it works.

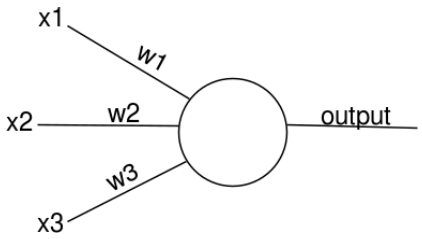


Figure 2.2: Perceptron model.

The perceptrons receives the input features x1, x2 and x3 as in Figure 2.2. A weight w1, w2 and w3 is associated to each input feature. The weights are real numbers that express how important is its associated input. In the end, the perceptron calculates the weighted sum of the input and evaluates the final response: If the weighted sum is higher than a threshold, the perceptron activates, output equal 1. Otherwise it's output is 0, not active. More formally:

$$output = \begin{cases} 0 & \text{if } \sum_j w_j x_j <= threshold \\ 1 & \text{otherwise} \end{cases} \tag{2.1}$$

The perceptron is a kind of unit that evaluates decisions by weighting up evidence. One great way to understand how it works is to make analogies. Suppose

that tomorrow an important music festival will be happening in your city and you would like to go, but you have some events to evaluate before decide in your final answer.

1. The weather is good or not

2. Your companion wants or not to go

3. International bands will play in the festival

We can represent these conditions by binary input variables x1, x2 and x3 respectively, i.e., x1 = 1 if the weather is good, x1 = 0 otherwise. Similary x2 = 1 if your companion wants to go with you and x3 = 1 if a international bands will play in the festival.

Now, we can model the perceptron to work the way we want: If an event has so much importance on the decision of going to the festival, we set a big weight for that. If not, we set a low weight. In the example above, let's set the value 4 to the threshold and let's assume that the third event has a great importance to you, i.e., no matter what happens (bad weather and without companion) if there are international bands, you are going anyway. In this case we set a weight equal to 5 to the third event, and weights 2 to the events 1 and 2. With these options, the perceptron model outputs 1 whenever international bands play in the festival. It doesn't matter if the weather is good or if your companion wants to go with you or not.

If one tries to model hard decisions with just one perceptron, probably it will fail. Hard decisions depends on multiple questions and answers. It's a level by level decision, where in the first levels the model tries to extract and understand basic features of the input and in the last levels more complex features are understood by the model. But if we stack many perceptrons in a layer by layer fashion, we can develop a robust model that can handle hard decisions. The proposed model is shown in Figure 2.3.

In this network, the first column of perceptrons on the left, is called the first layer of perceptrons, is responsible for making three very simple decisions, by weighing the input evidence. What about the perceptrons in the second layer? Each of those perceptrons is making a decision by weighing up the results from the first

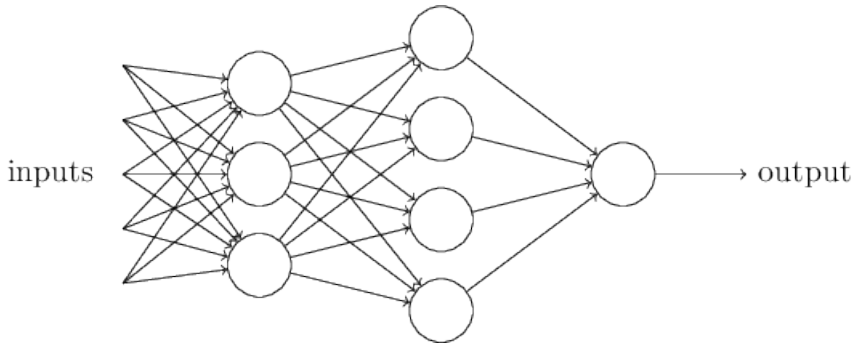


Figure 2.3: Many layers Perceptron model. [1]

layer of decision-making. This way a perceptron in the second layer can make a decisions at a more complex and abstract level than the perceptrons in the first layer. And even more complex decisions will be made by the perceptron in the third layer. Meaning that a many-layer network of perceptrons can perform sophisticated decision making [1].

Equation 2.1 is quite cumbersome with the threshold in the inequality. We will rearrange the equation to create a new term called *bias*. This term will control how easy is to the perceptron to output a 1. Also, we will get rid of the summation in the equation, changing it to a dot product. The new equation becomes 2.2.

$$output = \begin{cases} 0 & \text{if } w.x + b <= 0 \\ 1 & \text{otherwise} \end{cases} \tag{2.2}$$

The perceptron can also act as a NAND logical gate. Take a look at Figure 2.4.

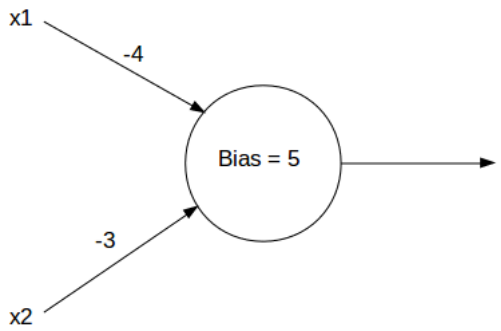


Figure 2.4: Perceptron can also works as a NAND logical gate.

The truth table for the perceptron when it receives "0" or "1" in its inputs is shown in the Table 2.1.

The truth table shows clearly that the percetron model reproduced faithfully the behaviour of the NAND gate. The good news is that NAND gates are universal in

X1	X2	Perceptron function evaluation	Output
0	0	-4*0 + -3*0 + 5	1
0	1	-4*0 + -3*1 + 5	1
1	0	-4*1 + -3*0 + 5	1
1	1	-4*1 + -3*1 + 5	0

Table 2.1: Truth table to perceptron as a NAND gate.

computation and then perceptrons can also be universal.

The computational universality of perceptrons is simultaneously reassuring and disappointing. It is reassuring because it tells us that networks of perceptrons can be as powerful as any other computing device. But it is also disappointing because it makes it seem that perceptrons are merely a new type of NAND gate. Which is hardly a big news [1]!

However, the situation is better than it seems. It turns out that we can devise learning algorithms which can automatically tune the weights and biases of a network of artificial neurons. This tuning happens in response to external stimuli, without direct intervention of the programmer. These learning algorithms enable us to use artificial neurons in a way which is radically different to conventional logic gates. Instead of explicitly laying out a circuit of NAND and other gates, our neural networks can simply learn to solve problems, sometimes problems where it would be extremely difficult to directly design a conventional circuit [1].

2.2 Sigmoid Neurons

There is a problem with perceptrons. It would be desired that small changes in the weights or bias to lead small changes on the output, but as will be seen later in this text, this can cause the network to generate completely different outputs. This desired behavior is very important for learning algorithms. This problem can be attacked by using Sigmoid neurons.

Like perceptrons, sigmoid neurons have inputs and weights, but its inputs can accept any value between "0" and "1". Also, they are modified so that small changes in their weights and bias cause only a small change in their output. Figure 2.2 that we use to represent perceptron can also be used to represent sigmoid neurons. They have the same components: Inputs, weights, output and a overall bias. The

difference here is that sigmoids neurons instead of only evaluating equation 2.2, it uses the result as the input for a new function called *sigmoid function*. The sigmoid function is defined as:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{2.3}$$

where z is:

$$z = \sum_j w_j x_j + b \tag{2.4}$$

The plot for the sigmoid function shown in the Figure 2.5.

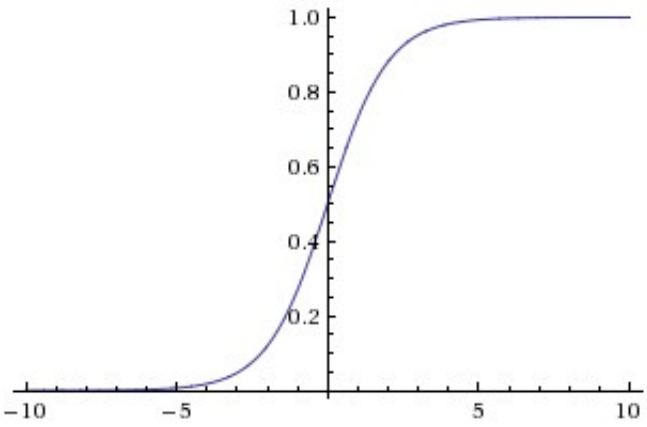


Figure 2.5: Sigmoid function [2].

From the plotting, one can see that when z assumes high negative or positive values, the sigmoid function behaves exactly as the perceptrons. The sigmoid function is like a smoothed version of a step function and this is the key fact that makes small changes in bias and weights to cause small changes in the output.

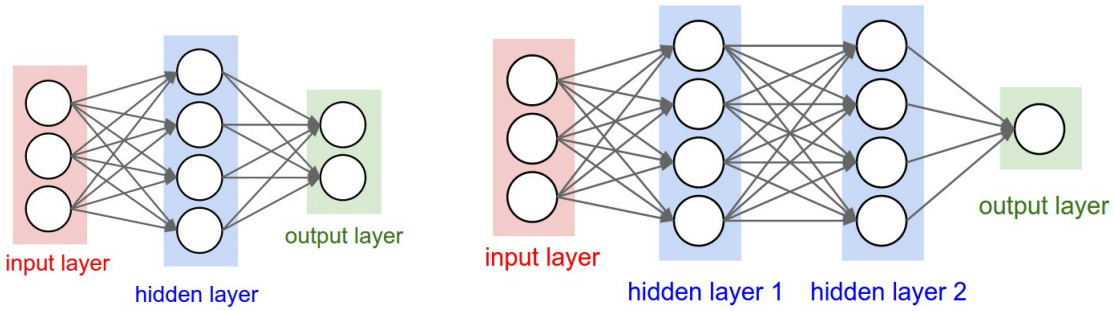
How should we interpret the output from a sigmoid neuron? Obviously, one big difference between perceptrons and sigmoid neurons is that sigmoid neurons do not output just 0s and 1s. They output any real number between 0 and 1, so values such as 0.173 and 0.689 are legitimate outputs. This can be useful, for example, if we want to use the output value to represent the average intensity of the pixels in an image input to a neural network. But sometimes it can be a nuisance. Suppose we

want the output from the network to indicate either "the input image is a 9" or "the input image is not a 9". Obviously, it would be easier to do this if the outputs were 0s and 1s, as for perceptrons. But in practice this behavior can be simulated by, for instance, stating that outputs smaller than 0.5 indicate a "not 9", and outputs greater or equal than 0.5 indicate "9" [1].

2.3 Neural Network Architecture

In the last section, we talked about neuron units, perceptrons and sigmoid neurons, and how they work. We also made a brief introduction about the stack of neuron units in layers. This is the main idea of this section, how to stack neurons in layers to create Neural Networks!

Neural Networks are modeled as collections of neurons that are connected in an acyclic graph. In other words, the outputs of some neurons can become inputs to other neurons. Cycles are not allowed since that would imply an infinite loop in the forward pass of a network. Instead of an amorphous blobs of connected neurons, Neural Network models are often organized as distinct layers of neurons. For regular neural networks, the most common layer type is the fully-connected layer in which neurons in two adjacent layers are fully pairwise connected, but neurons belonging to the same layer, share no connection. Figure 2.6 shows two examples of Neural Network topologies, that use a stack of fully-connected layers [2].



(a) A 2-layer Neural Network. One hidden layer of 4 neurons, one output layer with 2 neurons, and three input units.

(b) A 3-layer neural network with three inputs, two hidden layers of 4 neurons each, and one output layer

Figure 2.6: Two neural networks with different topologies. Notice that in both cases there are connections, synapses, between neurons of different layers, but not within a layer [2].

There are some conventions that the reader should be aware of to properly understand Neural Networks architectures:

1. The first layer is called *input layer*, the last one is called *output layer* and all others are called *hidden layers*.

2. We don't count the input layer. In Figure 2.6, for example, the left image is a 2-layer network because we have one hidden layer and one output layer.

3. This multiple layer topology, for historical reason, is called multilayer perceptrons (MLP) sometimes, despite being made up of sigmoid neurons, not perceptrons. It can also be called sometimes using the term Artificial Neural Networks (ANN).

Generally, the design of input and output layers of the ANN should be very simple and straightforward. The cool and artistic work happens in the design of the hidden layers. How many hidden layers? How many neurons in each layer? How to model the activation functions to be used? These questions are asked every time by the ANN designer. This is what differentiates one Neural Network from another and can lead to very different results. We will talk more about specific architectures later.

2.4 The Logistic Regression Model

Logistic regression is a regression model when one tries to predict the value of a output variable y given a input variable x. It's used in binary classification problems, where y can only assume values 0 and 1, and plays an important rule as the foundation to the understanding of more complex concepts in deep neural networks. The logistic regression model is defined as: given the input x, we want to find the output $\hat{y}$ using the model parameters w and b, weight and bias, respectively. The best w and b will minimize the *loss function* L over a set of m training examples

$$min_{w,b} J(w, b) \tag{2.5}$$

where J(w, b) is:

$$J(w,b) = 1/m * \sum_{m} L(\hat{y}^i, y^i) \tag{2.6}$$

The loss function L measures how close the predicted result $\hat{y}$ is from the actual result y. One simple thing that we could do is to use the squared error as loss function. It could work, but in logistic regression we use the *log loss* function because it is convex, to be seen in section 2.6. It is defined as:

$$L(\hat{y}, y) = -(y\ log\hat{y} + (1-y)log(1-\hat{y})) \tag{2.7}$$

This makes sense because if we have $y = 1$, it means that $L(\hat{y}, y) = -log\hat{y}$, and since we are trying to minimize the loss function, we want $log\hat{y}$, i.e., $\hat{y}$ to be the largest possible value and, as mentioned before, this is a classification problem and the largest value that $\hat{y}$ can achieve is 1, which is the result we want to predict. The opposite is also true. If $y = 0$, it means that $L(\hat{y}, y) = -log(1-\hat{y})$. Since we are trying to minimize the loss function, we want $-log(1-\hat{y})$, i.e., $1-\hat{y}$ to be the lowest possible value and, as mentioned before, this is a classification problem and this value for $\hat{y}$ is 0, i.e, which is also the result we want to predict.

There are other types of loss function that we will investigate later, but for logistic regression, the most used is the *log loss* function.

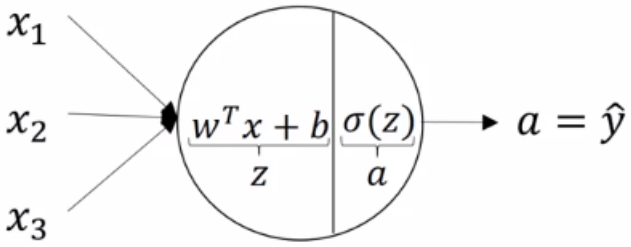
2.5 Feed-Forward Computation

Looking to Figure 2.6, one can see that there is no loop among the layers. For this is the reason those are called *feed-forward neural networks*. The data that came as the inputs go through the neural networks suffering modifications until the output at the end. In this section we will start to use some mathematical notation and formulas that will guide us to fully understand neural networks.

To understand the feed forward computation, we will first analyze what happens with a single neuron, after we will generalize for n neurons in a single layer l of the

neural network, and finally we will propagate of the calculations up to the output layer.

A neuron computes its output in two phases. First, it calculates the output z exactly as shown in equation 2.4. After, it uses the output z as an input to calculate a, i.e., its activation. Figure 2.7 shows these two phases.



$$z = w^T x + b$$

$$a = \sigma(z)$$

Figure 2.7: A single neuron feed-forward computation [3].

Now, let's generalize for an entire layer in the neural network. We will illustrate this example using the following neural network, Figure 2.8 [3].

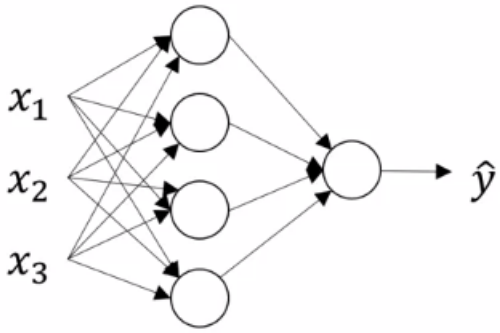


Figure 2.8: Neural network feed-forward computation [3].

In this network, we have three inputs, x_1, x_2 and x_3, first hidden layer has 4 neurons and the final output layer has one neuron. The feed-forward in the first neuron in the first hidden layer is:

$$z_1^1 = w^{1T}x + b_1^1 \tag{2.8}$$

$$a_1^1 = \sigma(z_1^1) \tag{2.9}$$

In Equation 2.8 and 2.9, the superscript indicates the layer number and the subscript the neuron position from top to bottom. In this case, for example, the equation above is calculating the output for the first neuron in the first hidden layer. In general, we have the output a for a neuron k in a layer l as follows:

$$z_k^l = w^{lT}x_l + b_k^l \tag{2.10}$$

$$a_k^l = \sigma(z_k^l) \tag{2.11}$$

And if we do the same thing for every neuron in a specific layer, we end up with a final matrix a^l that represents all the activations for all n neurons in a specific layer:

$$z^l = \begin{bmatrix} z_1^l \\ z_2^l \\ \vdots \\ z_n^l \end{bmatrix}$$

$$a^l = \begin{bmatrix} a_1^l \\ a_2^l \\ \vdots \\ a_n^l \end{bmatrix}$$

or:

$$z^l = a^l$$

The output a^l is feed-forward as input x to the next layer until we get the final result in the output layer. Specifically in the neural network from Figure 2.8 the intermediate and final results are:

$$z^1 = w^1 x + b^1 \tag{2.12}$$

$$a^1 = \sigma(z^1) \tag{2.13}$$

$$z^2 = w^2 a^1 + b^2 \tag{2.14}$$

$$a^2 = \sigma(z^2) \tag{2.15}$$

2.6 The Backpropagation Algorithm

From the previous section, we learned how to calculate the final output feed forwarding the data from the input until the last neuron in the output network layer. However, it's important to note that we are trying to *train* neural networks to achieve the best result for the task. We need a scheme that can read the output value and adjust every weight and bias of the network to achieve better results in the output. This is called the *backpropagation algorithm*.

Before studying this algorithm, we first need to understand the concept of *gradient descent*. We learned from Section 2.4 that we have to find parameters w and b that minimize the loss function. How do this by using the gradient descent.

The cost as a function of parameters W and b have a bowl shape as shown in Figure . It means that we are working with a convex function with global optima and this is the reason why the gradient descent works so well in this context [3].

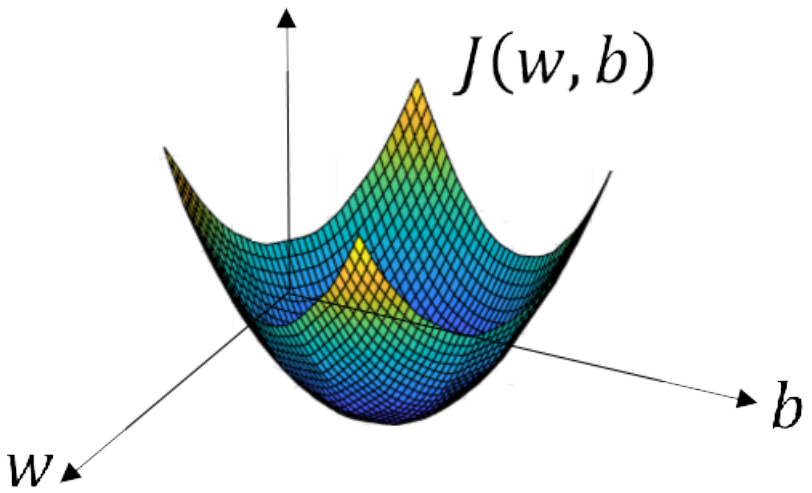


Figure 2.9: The cost as a function of parameters W and b [3].

The gradient descent algorithm is very simple, and can be written in *python* as follows:

```
1    def gradient_descent(learning_rate, w, b,
2                         num_epochs, training_set):
3        for i in range(num_epochs):
4            forward_result = forward_pass(training_set)
5            dw, db = backpropagation(forward_result)
6            w = w - learning_rate * dw
7            b = b - learning_rate * db
```

The algorithm only iterates a fixed number of *epochs* and in each iteration, using the forward and backpropagation algorithms, it calculates the derivative of the cost function in respect of w and b, dw and db respectively, and these values are scaled using a factor of *learning_rate* that controls how fast we are going towards the global optima point. The minus signal in the formula indicates that the gradient descent is trying to go in the opposite direction of the gradient, i.e, if the gradient is positive, and we saw previously that the function is convex, then the update must be negative to bring w and b to the global optima point. If it is negative, then the update must be positive.

In Deep Learning, we are dealing with a lot of data, in the order of ten of thousands or more. One flaw of the gradient descent algorithm is that if the size of the training set is huge it will take a lot of time to finish even the first epoch. It will be in practice impossible to find the global minima of the function in a reasonable time. There are two variations of the gradient descent algorithm that we can do to speedup the training in this case: (1) *mini-batch gradient descent* and (2) *stochastic gradient descent*. What we do in both algorithms is to divide the training set in K blocks called *batches*, of size B. Now a different block is used in each epoch of training. It means is that instead of updating weights and bias only once by epoch of the gradient descent algorithm, we will do K updates for the weight and bias per epoch. In particular, in stochastic gradient descent we have only one sample per batch, $B = 1$, and then $K = \text{m}$, where m is the number of training examples. Mini-batch gradient descent, in turn, allows us to set any value for B and then $K = m/B$. Generally, in Deep Learning, we avoid using the stochastic gradient

descent because it is much more difficult to converge: in one epoch you can have a good training sample and go one step towards the convergence. While in another epoch with a bad training sample, leads us away from convergence. Also, in terms of implementation you lose the advantage of vectorization, since there will be only one example in the batch. The best choice is to use the mini-batch gradient descent and choose a batch size of size 2^n, where $n = 1, 2, 3, 4...$ This choice makes the training step faster compared to gradient descent and with a better convergence compared to stochastic gradient descent. Using 2^n as batch size also helps the allocation in CPU/GPU memory, as the allocation systems in these devices generally work better with allocations multiples of 2. The mini-batch gradient descent algorithm is as follows:

```
1     def mini_batch_gradient_descent(learning_rate, w, b,
2                                     num_epochs, training_set, B):
3         K = len(training_set) / B
4         for i in range(num_epochs):
5             for j in range(K):
6                 current_batch = training_set[j*B:j*B+B]
7                 forward_result = forward_pass(current_batch)
8                 dw, db = backpropagation(forward_result)
9                 w = w - learning_rate * dw
10                b = b - learning_rate * db
```

Now that the gradient descent was introduced, lets understand how to calculate the derivatives. Again, we will first see how we do it using logistic regression and then we will generalize it. As shown in Figure 2.10, in logistic regression we do a forward computation, black arrows, afterwards we backpropagate, red arrows, the gradients and update the parameters w and b.

For the sake of simplicity, we will refer to da, dz, db and dw as the derivatives of the loss function with respect to the variables a, z, b and w, respectively. In logistic regression example above, the equations are:

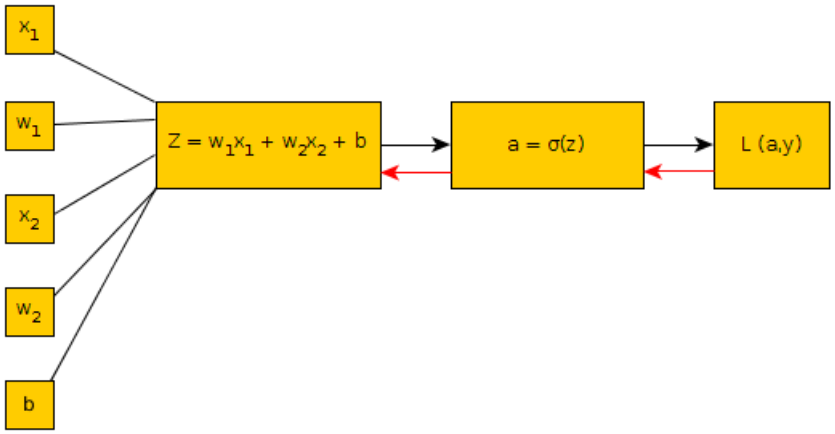


Figure 2.10: Forward and backpropagation in logistic regression.

$$da = \frac{dL(a,y)}{da} = \frac{-y}{a} + \frac{1-y}{1-a} \tag{2.16}$$

$$dz = \frac{dL(a,y)}{dz} = \frac{dL}{da} * \frac{da}{dz} = da * \frac{d\sigma}{dz} \tag{2.17}$$

$$dw_1 = \frac{dL(a,y)}{dw_1} = \frac{dL(a,y)}{dz} * \frac{dz}{dw_1} = dz * x_1 \tag{2.18}$$

$$dw_2 = \frac{dL(a,y)}{dw_2} = \frac{dL(a,y)}{dz} * \frac{dz}{dw_2} = dz * x_2 \tag{2.19}$$

$$db = \frac{dL(a,y)}{db} = \frac{dL(a,y)}{dz} * \frac{dz}{db} = dz * 1 = dz \tag{2.20}$$

$$\tag{2.21}$$

These obtained these equations for backpropagation using calculus concept of chain rule for derivatives. For example: To calculate dz we take "what is in front of parameter z in the sequence", in this example parameter a, and calculate the derivative of loss function in respect of this variable. Then we multiply this result by the derivative of a with respect of z since a depends on z. In other words, we are trying to measure the modifications that can occur in the output with a slight variation in a and z. All the equations above use this same chain rule principle.

Now, we will generalize the linear regression above to a neural network with two layers. From two layers and beyond the process is the same. The two layer neural network is shown in the Figure 2.11.

As usual, we start the process from the end to the begin. The superscript $[k]$ means the $k-th$ layer. In the example, starting from the end, second layer, the

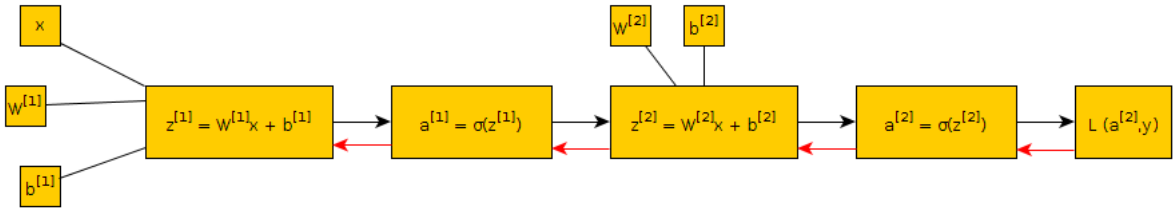


Figure 2.11: Forward and backpropagation in a 2-layer neural network.

equations are given by:

$$da^{[1]} = \frac{dL(a^{[2]}, y)}{da^{[1]}} = \frac{-y}{a^{[2]}} + \frac{1-y}{1-a^{[2]}} \tag{2.22}$$

$$dz^{[2]} = \frac{dL(a^{[2]}, y)}{dz^{[2]}} = \frac{dL}{da^{[2]}} * \frac{da^{[2]}}{dz^{[2]}} = da^{[2]} * \frac{d\sigma(z^{[2]})}{dz^{[2]}} = a^{[2]} - y \tag{2.23}$$

$$dw^{[2]} = \frac{dL(a^{[2]}, y)}{dw^{[2]}} = \frac{dL(a^{[2]}, y)}{dz^{[2]}} * \frac{dz^{[2]}}{dw^{[2]}} = dz^{[2]} * a^{[1]T} \tag{2.24}$$

$$db^{[2]} = \frac{dL(a^{[2]}, y)}{db^{[2]}} = \frac{dL(a^{[2]}, y)}{dz^{[2]}} * \frac{dz^{[2]}}{db^{[2]}} = dz^{[2]} * 1 = dz^{[2]} \tag{2.25}$$

Continuing to propagate the gradients until the input vector X, the equations for the first, and last propagation layer, become:

$$da^{[1]} = \frac{dL(a^{[2]}, y)}{da^{[1]}} = \frac{dL(a^{[2]}, y)}{dz^{[2]}} * \frac{dz^{[2]}}{da^{[1]}} = (a^{[2]} - y) * w^{[2]} \tag{2.26}$$

$$dz^{[1]} = \frac{dL(a^{[2]}, y)}{dz^{[1]}} = \frac{dL}{da^{[1]}} * \frac{da^{[1]}}{dz^{[1]}} = (a^{[2]} - y) * w^{[2]} * \frac{d\sigma(z^{[1]})}{dz^{[1]}} * z^{[2]} \tag{2.27}$$

$$dw^{[1]} = \frac{dL(a^{[2]}, y)}{dw^{[1]}} = \frac{dL(a^{[2]}, y)}{dz^{[1]}} * \frac{dz^{[1]}}{dw^{[1]}} = dz^{[1]} * X^T \tag{2.28}$$

$$db^{[1]} = \frac{dL(a^{[2]}, y)}{db^{[1]}} = \frac{dL(a^{[2]}, y)}{dz^{[1]}} * \frac{dz^{[1]}}{db^{[1]}} = dz^{[1]} * 1 = dz^{[1]} \tag{2.29}$$

With all the gradients at hand, one only needs to use the gradient descent to update the weights and biases of the neural network.

2.6.1 Faster Convergence: Input Normalization and Batch Normalization

Recall that the usage of mini-batch gradient descent turns the training reasonably faster. It is very simple, and it works like a brute force method, i.e, we are decreasing

the size of the training set for each gradient descent step. We can implement two data normalization that will help for a faster convergence of gradient descent: Input normalization and Batch normalization [30].

When training a Deep Neural Network, the input feature vector X can have many features. The problem here is that each feature can have different scales. This means that one feature could, for example, have values in range $[0, 1]$, but some other feature in X could have values in range $[0, 1000]$. This leads to a problem where one feature could have a variance much more larger than another and it will impact directly in the gradient descent derivatives and, as a consequence, in the training. The ideal solution would be to centralize all the features with zero mean and unit variance. We can achieve it by implementing *input normalization* as shown in equation 2.32.

$$\mu = \frac{\sum_{i=1}^{n} X^{(i)}}{n} \tag{2.30}$$

$$\sigma^2 = \frac{\sum_{i=1}^{n} (X^{(i)} - \mu) * *2}{n} \tag{2.31}$$

$$X = \frac{X - \mu}{\sigma^2} \tag{2.32}$$

This helps regarding the input features. But what about the activation of each neuron in each hidden layer? Can we also normalize it? That is the problem that Batch normalization tries to solve. For the sake of example, suppose that you are trying to train the neural network shown in 2.11 and you want to make the values of all neuron outputs in $Z^{[l]}$ in any hidden layer normalized, i.e., with the same mean and variance along all the training. For the sake of simplicity, let's think about $Z^{[l](i)}$, i.e., just one neuron output of the l-th layer. We start normalizing this neuron output exactly the same way we did in equation 2.32. Equation 2.35 shows the normalization of one neuron using batch normalization.

$$\mu = \frac{\sum_{i=1}^{n} Z^{[l](i)}}{n} \tag{2.33}$$

$$\sigma^2 = \frac{\sum_{i=1}^{n} (Z^{[l](i)} - \mu) * *2}{n} \tag{2.34}$$

$$Z_{norm}^{[l](i)} = \frac{Z^{[l](i)} - \mu}{\sqrt{\sigma^2 + \epsilon}} \tag{2.35}$$

Note that ϵ is just a parameter to control numerical stability, i.e., to avoid division by zero.

Now, remember that **we want a fixed mean and variance for the whole training**, i.e., Batch normalization has the objective to make the distribution of values fixed in a mean and variance no matter what the updates of gradient descent in the weights and bias of the network do. To do this, we introduce two new learnable parameters in the network, σ and β that will control this behaviour. The final normalized $Z^{[l](i)}$ is now known as $\tilde{Z}^{[l](i)}$ and is shown in equation 2.36.

$$\tilde{Z}^{[l](i)} = \sigma Z_{norm}^{[l](i)} + \beta \tag{2.36}$$

Finally, you just need to train your neural network exactly the same way we presented before using mini-batch gradient descent, but using $\tilde{Z}^{[l](i)}$ in forward propagation instead of $Z^{[l](i)}$ for each hidden layer.

2.7 Loss Functions

Until now, we presented in this text, as in Section 2.4, only the *log loss* function. There are a lot of other loss function and each one is suitable for a specific domain. In this section we will present some of these functions so that the reader can be aware of them. The notation in the section showing what is prediction and what is actual value is the same as used in 2.4.

Quadratic Loss Function

The quadratic loss function, also known as MSE function, is defined as:

$$L(\hat{y}, y) = \frac{1}{2n} \sum\nolimits_n |\hat{y} - y|^2 \tag{2.37}$$

Equation 2.37, it's very simple: it goes in each one of the n training examples and computes how far the prediction $\hat{y}$ is from the actual value y. This difference is augmented by a squared factor and summed up. At the end, it is averaged over the n training examples.

Although fast and simple, the MSE has a serious drawback from a training perspective, as explained next.

Let z and $\hat{y}$ be the same as defined in Figure 2.7. We can use the math chain of rule to differentiate Equation 2.37 with respect to the bias and weights:

$$\frac{dL(\hat{y}, y)}{dw} = (\hat{y} - y)\sigma^{'}(z)x \tag{2.38}$$

$$\frac{dL(\hat{y}, y)}{db} = (\hat{y} - y)\sigma^{'}(z) \tag{2.39}$$

Now imagine, as an example, a simple problem where the real output y is 0 and the neuron from Figure 2.7 starts to predict it as 1. The neuron will try to bring it down to zero, but it will be a very slow process because the term $\sigma^{'}(z)$ is very close to zero when the predicted output is close to 1 as shown in Figure 2.5. With this term close to zero, the weights and bias are updated very slow and the convergence for the true output is also very slow.

Cross-entropy Loss Function

In Section 2.4 we introduced the *log loss* function. Logistic regression is a problem where one tries to classify an input x in one of two different classes. However, the log loss function is just a special case of a more generic loss function known as *Cross-entropy* that deals with more than two classes as output, i.e, M classes.

$$L(\hat{y}, y) = \sum_{j=1}^{M} y_j \log \hat{y}_j + (1 - y_j) \log(1 - \hat{y}_j) \tag{2.40}$$

It is not so easy to see, in a first look, but the cross-entropy bupasses the slow learning curve of the MSE. It is easy to show that the derivative of log-loss function is what is shown in Equation 2.42. The term $\sigma^{'}(z)$ disappeared and now the updates of weights and bias are truly controlled by the error in the output, $\sigma(z) - y$.

$$\frac{dL(\hat{y}, y)}{dw} = x(\sigma(z) - y) \tag{2.41}$$

$$\frac{dL(\hat{y}, y)}{db} = \sigma(z) - y \tag{2.42}$$

Dice Coefficient Loss

The dice coefficient loss function, also known as **F1-Score**, **is the loss function that we use to classify pixel predictions of Corpus Callosum (CC) in the implementation of this work**. To understand it clearly, we need first to understand an important statistical concepts like $2x2$ contingency table, True Positive, True Negative, False Positive and False Negative. With that clear, we can show statistical metrics like Accuracy, Precision, Recall and, finally, the Dice Coefficient.

To illustrate all of the concepts, let's, for the sake of simplicity, imagine that we have a $3x3$ MRI image of a imaginary brain as shown in Figure 2.12. The white square is a "corpus callosum pixel" and the black square isn't.

Figure 2.12: A 3x3 MRI example image.

Now, again, for the sake of simplicity, suppose that we built a segmentation model and it predicted Figure 2.13 as the segmented image.

Looking at both images, the first question is *How far the segmented image is from the true image*? To answer this, let's first classify each pixel in the segmented image as TP, TN, FP and FN (True positive, True negative, False positive and False

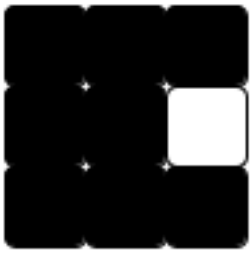


Figure 2.13: A $3x3$ MRI example image segmentation.

negative, respectively).

- TP: Pixel at position k in the true image is a corpus callosum pixel and pixel at the same position in predicted image is a corpus callosum pixel too;
- TN: Pixel at position k in the true image is not a corpus callosum pixel and pixel at the same position in predicted image is not a corpus callosum pixel too;
- FP: Pixel at position k in the true image is not a corpus callosum pixel and pixel at the same position in predicted image is a corpus callosum pixel;
- FN: Pixel at position k in the true image is a corpus callosum pixel and pixel at the same position in predicted image is not a corpus callosum pixel;

Table 2.2 is know as *contingency table* and shows these numbers for images of figure 2.12 and 2.13.

	Is CC Pixel (True Img.)	Is Not CC Pixel (True Img.)
Is CC Pixel (Predicted Img.)	0	1
Is Not CC Pixel (Predicted Img.)	1	7

Table 2.2: Contingency table.

The accuracy of the system is given by:

$$accuracy = \frac{TP + TN}{TP + FP + FN + TN} \tag{2.43}$$

Here is our first warning about using accuracy. We got a good accuracy ($\approx$ 77.7%) in the example, but the segmented image had the CC completely different

from the true image, i.e., all predictions in the CC area were wrong. It happens for accuracy whenever we have imbalanced data. In this example, for example, only one pixel for a total of nine is a CC pixel.

Further, we can have two other numbers that measures this problem, Precision and Recall. The Precision tries to answer the question *From all the pixels that the model classified as CC, how many of them were really CC, i.e., the model took the right decision*?. Recall, in turn, is trying to answer the question *From all the pixels that were really CC in the true image, how many of them the model predicted right*?.

Recall and Precision are given by:

$$precision = \frac{TP}{TP + FP} \tag{2.44}$$

$$recall = \frac{TP}{TP + FN} \tag{2.45}$$

For the example, we have a precision and recall of 0%. That is much more consistent with the model predictions.

We can combine Precision and Recall in a harmonic average of both. This is known as the Dice Coefficient or F1-Score. It has the goal of giving equal importance to precision and recall and its score is always in the range $[0, 1]$, i.e., 1 means that the task was perfect, obtaining the correct result, and 0 means that we made all kinds of mistakes.

$$dice = \frac{2 * precision * recall}{precision + recall} \tag{2.46}$$

In resume, the Dice Coefficient is a measure of how well we are doing the comparison with the ground-truth. To use it as the loss functions, we must always instruct the gradient descent to go in the opposite direction of the dice. It means that if we have a dice coefficient of 1, the loss is -1 because we are making less mistakes. If the dice coefficient is near zero, we are making a lot of mistakes and then gradient descent should have a higher loss, near zero in this case.

2.8 More Activation Functions

In Section 2.2 we started describing the sigmoid function. Although it is very simple, easy to understand and to use, the sigmoid function has two major drawbacks: It can *saturate and kill the gradients*, and its output is not zero-centered. The first drawback is pretty clear when we look at Figure 2.14. Note that if we have activations near zero or one, near the "tails" of the sigmoid curve, the derivative of the activation function at these points will be almost zero or zero. As we use the derivative of activation function in the backpropagation algorithm, this zeroed value will be propagate through the network and the gradient will be entirely killed for the neurons that receive this gradient. This is known as the *vanishing gradient* problem [31]. The second drawback has less severe consequences, but it is still a problem because if its output is not zero-centered it means that the activations will be particularly in the range $[0, 1]$ for the sigmoid neuron. Now, see in Equation 2.49 what happens for a single neuron in the backpropagation:

$$z = \sum_i w_i x_i + b \tag{2.47}$$

$$\frac{df}{dw_i} = x_i \tag{2.48}$$

$$\frac{dL}{dw_i} = \frac{dL}{df}\frac{df}{dw_i} = \frac{dL}{df}x_i \tag{2.49}$$

As x_i is always positive, then the gradient will depend exclusively on $\frac{dL}{df}$. It will make the whole gradient to be either all positive or all negative, introducing an undesirable zig-zagging effect in the gradient descent algorithm while updating. This has less severe consequences because the gradients are summed up across mini-batches of data in each training epoch and then the weights will have variable signs.

One way to solve this second drawback is to make the sigmoid zero-centered. This is what we do by introducing the *tanh non-linearity* activation function. The tanh takes a real number and squashes it into range $[-1, 1]$. As this range now can have values positives or negatives, the zig-zagging effect disappears. However, we still have problems of gradient being killed with tanh as shown in Figure 2.15.

We still have to consider ReLU [32]. It has become very popular in the last

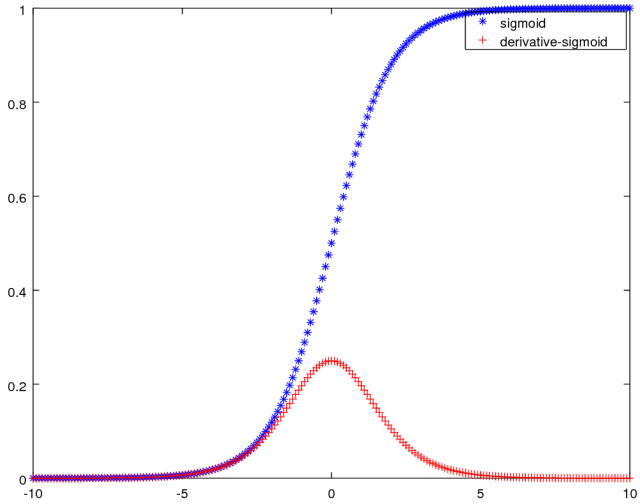


Figure 2.14: Sigmoid and its derivative.

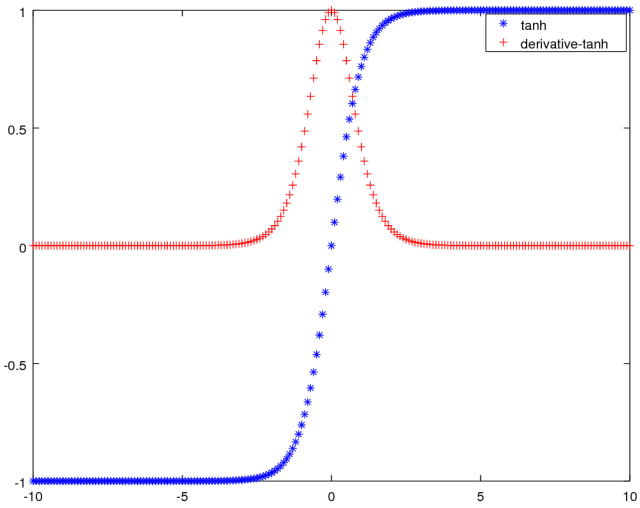


Figure 2.15: Tanh and its derivative.

few years. It computes the function $f(x) = max(0, x)$ as shown in Figure 2.16. ReLU has two advantages over tanh and sigmoid. First, it is simpler to calculate: no exponentials, no divisions. Just a threshold. Second, it was found to greatly accelerate approximately by a factor of $6x$ the convergence of stochastic gradient descent compared to the sigmoid/tanh functions. It is argued that this is due to its linear, non-saturating form [33]. However, it has the same problem of not being centered similar to sigmoid function and it can also suffer from the vanishing gradient problem because as shown in Figure 2.16, the derivative value of ReLU for $x < 0$ is always zero. It's important to note that, unlike sigmoid, this problem has lesser

chance to happen because the gradient is only zero on half of the curve.

One thing that can be done to prevent the vanish gradient problem in ReLU is to implement a new function called Leaky ReLU [34]. The Leaky ReLU is defined by the function $f(x) = max(0.1x, x)$. What is does is that in the first half of the ReLU curve it adds a small positive slope to prevent the vanishing gradient problem.

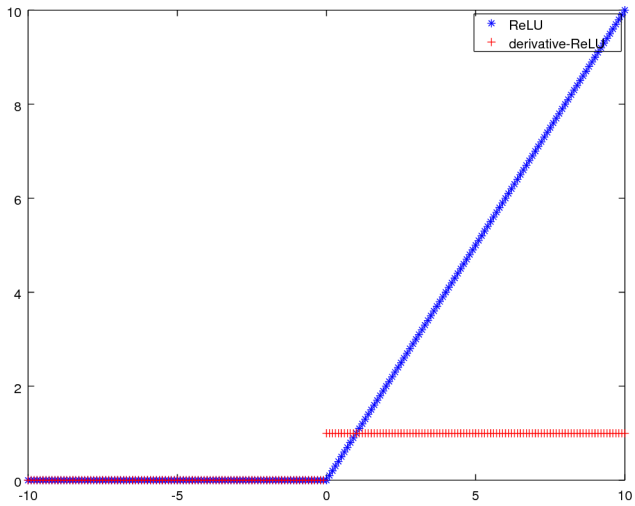


Figure 2.16: ReLU and its derivative.

2.9 Improving the way Neural Networks Learn

In the previous sections, we explained the foundations of how neural networks work. In this section we will go further, showing how can be improved the learning of the neural networks using hyperparameters tuning, data setup and a faster optimization algorithm.

2.9.1 Machine Learning Basics: Training, Validation and Test Datasets

The applied Machine Learning is a highly interactive process as shown in the Figure 2.17. It starts with the idea, goes to coding and ends with the experiments. It is repeated until you find a good model.

A deep learning model depends on a large number of parameters: number of layers, number units in each hidden layer, activation functions, learning rates, etc.

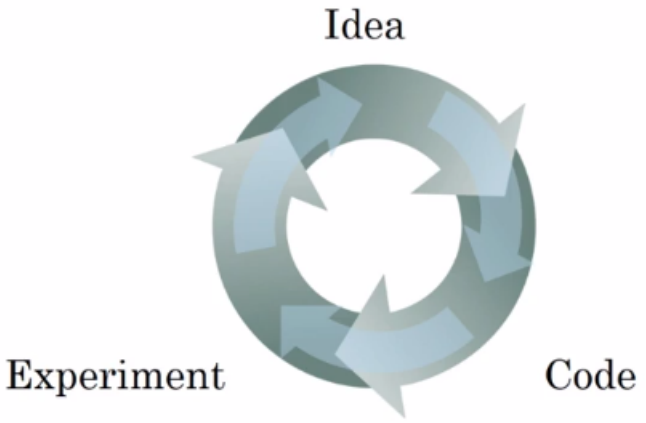


Figure 2.17: The iterative Machine Learning process. [4]

Even very experienced deep learning experts find it very hard to correctly guess the best choice of hyperparameters at a first glance. If one tries to repeat the cycle depicted in Figure 2.17 indefinitely, by randomly changing the hyperparameters, the learning process does not converge. That is the reason that we must split our dataset in three sub-sets: training, validation and test. This will allow us to find the best model.

The training sub-set is used to train the models. The validation sub-set is used to evaluate the models, each one having different hyperparameters. The goal of the validation set is just select the best model, i.e. the model that produces the results closer to the expected ones. Finally, the test sub-set is the fireproof that will show how well the best model works on the real world. The importance of the test sub-set is that it gives an *unbiased* estimation of your model performance. It's not absolutely required to have the test sub-set, but if your dataset is large enough to allow you to separate a part of it to used as such, it is always a good idea to do it.

In the past, before the "big data" era, it was a common practice to split your dataset in 60% for training, 20% for validating and 20% for testing. Nowadays with huge datasets at hand, for instance one million data, one can split it into 98/1/1% for training, validation and test, respectively, which will give you enough amount of data for each of the three steps [4].

When you train a neural network, you must pay attention two important measurements: *bias* and *variance*. The bias measures how well your model fit the training data, while the variance measures how well your model can generalize in

the validation set. To illustrate these concepts, let's imagine that we are trying to do a cat classifier, i.e., a system that classifies an image as *cat* or *not cat* and assume that the human error in classification of cat images is almost 0%. This is called *optimal bayes error*, i.e, this is our error rate baseline and we can't be better than this. Now, we can measure bias and variance in four different cases:

1. 1% training set error and 9% validation set error → low bias and high variance: Model is not good because is not generalizing well.

2. 10% training set error and 9% validation set error → high bias and low variance: Even though the model has low variance, it is not good because it is not fitting the training data.

3. 20% training set error and 12% validation set error → high bias and high variance: Model is not good because is not fitting the training data and it is not generalizing well. Worst case!

4. 1% training set error and 2% validation set error → low bias and low variance. This is the best model! The model is fitting the training data and it is also generalizing well on validation dataset.

It is important to note that we always have to pay attention to the baseline error. If, for example, the baseline human error were about 9% then the second case above would be considered a good model.

A general framework for Machine Learning is:

1. Train the model. If it has high bias, then you try a few things: A bigger network, train it longer or try a new neural network architecture.

2. Repeat step 1 above until you are comfortable with a good bias value.

3. Once you solved the bias problem, you check for the variance. If the model has high variance, you can try to add more data. Use regularization, we will talk more about it in the next section, or try a new neural network architecture.

4. Repeat step 3 above until you are comfortable with a good variance value.

2.9.2 Regularization

Sometimes, your model overfits. Overfitting happens when your model has a low error in training set, but a relative high error in validation data, i.e., the model is only memorizing the examples, but no generalizing well for new examples. It is the case 1 presented before.

A simple solution to prevent overfitting is to add more data. If there is no more data available another solution is to apply *regularization*. Let's try to understand how to apply *regularization* by using logistic regression, introduced in Section 2.4. Basically the two most common types of regularizers are L1-norm 2.51 and L2-norm 2.53. By applying these regularizers to Equation 2.6, we obtain equations 2.50 and 2.52:

$$J(w,b) = 1/m * \sum_{m} L(\hat{y}^i, y^i) + 1/m * \lambda * ||w||_1^1 \tag{2.50}$$

Where L1 regularization term is defined as:

$$||w||_1^1 = \sum_{j} w_j \tag{2.51}$$

$$J(w,b) = 1/m * \sum_{m} L(\hat{y}^i, y^i) + 1/2m * \lambda * ||w||_2^2 \tag{2.52}$$

Where L2 regularization is known as squared euclidean norm and is defined as:

$$||w||_2^2 = \sum_{j} w_j^2 = w^T w \tag{2.53}$$

The parameter λ is a hyperparameter known as *regularization parameter* and is obtained using cross validation in the training set as explained previously in Subsection 2.9.1.

The reader could also argue that we are not adding any regularization term in

bias parameter b. We don't regularize parameter b because it is a single number while w is a much higher dimension parameter vector and presents higher variance problem [4].

Besides L1 and L2 regularization, one very creative and very functional technique is *dropout* [35]. Imagine, for example, that you trained a Neural Network like the one from Figure 2.6(b) and it overfits. Dropout means that for every training example that is fed through out the network, we are going to randomly select, with probability p, in each layer some neurons to literally vanish. It means that their output will be set to zero. This will have two consequences that will prevent overfitting: (1) By removing some neurons, our network become simpler, with less parameters, or less likely to overfit; and (2) the preserved neurons become **much more specialized** in the task, once they can not rely on a neighbor neuron, since this neuron could have been dropped out from the process. This is a process called *weight spread out* and it means that the neuron that was not dropped out will try to spread the weights among its neighbors neurons. The weights spreading will have the effect of shrinking the squared norm of the weights and it is very similar to what happens in L2 regularization.

Another regularization technique is *early stopping*. In this technique, we keep measuring the training and validation set error of the model iteration by iteration. When these two curves start diverging, it is time to stop the training and save the model. The downside of this technique is that by early stopping the training you can simultaneously affect the bias and variance of your model, meaning that they are coupled. It is against the orthogonalization principle of machine learning that states that you should be able to change one parameter and see only the particular effect of on the model without coupling it with other parts of your model [36].

Recall from section 2.6.1 that Batch Normalization scales each mini batch by the mean/variance computed on only that mini batch. Well, as we are not using the full distribution, i.e., the whole training set and only a batch, this mean and variance is a estimate over the entire training set and then adds some noise to the values of $\tilde{Z}^{[l](i)}$ in this particular mini batch. This is a similar process that happens with dropout. In dropout, we are introducing a multiplicative noise by multiplying some neurons outputs by zero. In Batch normalization, the noise is introduced by

μ and σ^2. This noise, like dropout, has a regularization effect [30].

Remember from the beginning of this section that we said that a simpler approach is add more data. If it is difficult to get new data, one can synthesize new data from the data at hand, known as *data augmentation*. It can be done by applying rotation, translation, shearing, zooming, etc, to the images that you have, generating new ones.

2.10 Convolutional Neural Networks

From section 2.3 to 2.6, we established the Neural Networks foundations, from its conception to how we train it. We started explained what is a neuron, how we stack many of them in a layer, how we create many layers and how we train of all this using the backpropagation algorithm. In this section we are going to explain what is a Convolutional Neural Network (CNN).

Convolutional Neural Networks are very similar to ordinary Neural Networks from the previous sections: they are made up of neurons that have adjustable weights and biases. Each neuron receives some inputs, performs a dot product and optionally follows them with a non-linearity. The whole network still expresses a single differentiable score function: from the raw image pixels on one end to class scores at the other. And they still have a loss function (e.g. SVM/Softmax) on the last (fully-connected) layer and all the tips/tricks we developed for learning regular Neural Networks still apply [2].

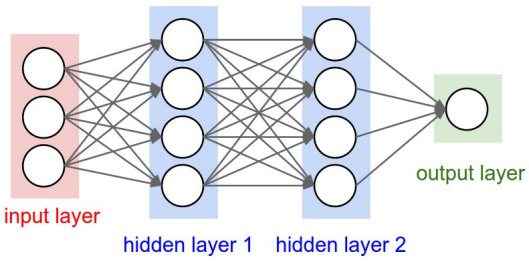
If they are similar and every technique that we learned for Neural Networks can be used in CNN, why do we need to learn one more type of network? A simple assumption made for the CNN is that the inputs **are always images**. This assumption allows the application of contractions on the images, through out the neural network, making the computation faster, since it reduces the number of parameters in the network.

2.10.1 CNN Architecture

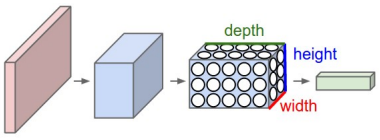
Recall from the Neural Networks, that NNs receives an input, for instance an image, the process is a sequence of activations that happens from a neuron to all neurons in

the next hidden layer. Each neuron of a hidden layer is fully connected to all neurons in the previous hidden layer. The problem with this approach is the big number of *weights* that we would have in the neural network. To work on $256x256$ images as input, for instance, each neuron in the hidden layers will have $64K$ connections. If we consider a bigger image, like $1Kx1K$, we would have $1M$ connections. Such architecture may not scale satisfactory and great probability of overfitting.

Convolutional Neural Networks take advantage of the fact that the input consists of images and they constrain the architecture in a more sensible way. In particular, unlike a regular Neural Network, the layers of a ConvNet have neurons arranged in 3 dimensions: width, height, depth. Note that the word depth here refers to the third dimension of an activation volume, not to the depth of a full Neural Network, which can refer to the total number of layers in a network [2]. This arrangement makes neurons in one layer connect only with a small region of the layer before it, instead of all neurons. Every layer in the ConvNet has a simple contract: It transforms a 3D input volume to another different output volume. In the figure below, one can see the difference between Neural Networks and Convolutional Neural Networks.



(a) A 3-layer neural network with three inputs, two hidden layers of 4 neurons each and one output layer



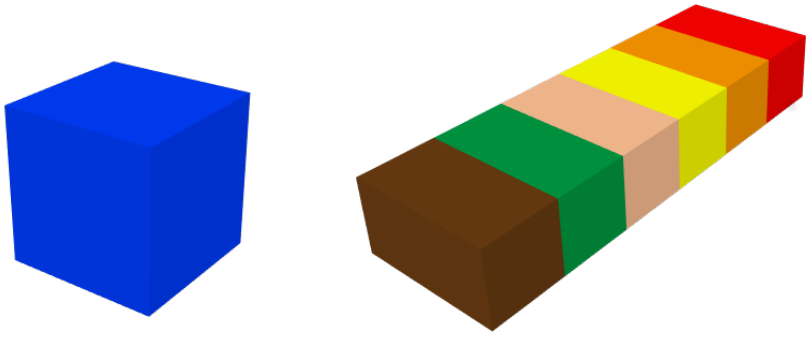
(b) A convolutional neural network with layers arranged as volumes.

Figure 2.18: In the Neural Network (a), all neurons of one layer are fully connected to all neurons from the previous layer. In the ConvNet (b), the neurons are arranged as 3D volumes [2].

2.10.2 The spacial arrangement of CNN

Recall from section 2.10.1 that each layer in the CNN receives in its input a volume. Let's now describe how we can "walk" in a input volume and produce a different spatially arranged output volume. We need to understand three hyperparameters that control the size of the output volume: Depth, Stride and Padding. For the

sake of example, let's assume that our input volume is a single channel image with dimensions 32x32x1 and that we try to apply in this volume six filters of dimensions 3x3x1 each one as as shown in Figure 2.19.



(a) Input volume. (b) Filter volume.

Figure 2.19: The input volume, the image with dimension 32x32x1 to be covolved, is represented by a single cube in (a). In (b), there are 6 filters, each one having dimensions 3x3x1 and being represented by a different color.

The first hyperparameter is *Depth*. Depth, as the name says, is just the depth of the input/filter volume, i.e., the number of filters that we are applying in the input volume. In this example, the depth of the filter volume is six.

Now, imagine that we get only the first filter, the red one, to do convolutions of this filter through the input volume. To do this, we place the red filter in the top left corner of the input volume and make the following question to ourselves: *Walking pixel by pixel horizontally, from left to right, how many different positions the filter can occupy?* In this example, it can occupy exactly different 30 positions without extrapolate the input volume size in the same dimension (Remember: Input volume has size of 32 in the same dimension!). To analyze the size vertically, i.e. from top to bottom, we do the same logic. As the input volume has also size of 32 from top to bottom then the output dimension in this case will also be 30. An important point here is, as we are doing convolutions, each filter in the filter volume **needs to have** the same depth of the input volume, one in this case (otherwise the convolution math would not work). So, in the end, the red filter produced a filtered image of size 30x30x1. Doing this for all the filters in the filter volume we would have in the end a final output volume of 6x30x30x1. The step size that we choose to walk through the pixels is known as *Stride*. The stride controls the size of the output volume spatially. If we had chosen a filter size of 5x5x1 and a stride of 3,

for example, we would have a volume of 16x16x1 in the end. That's approximately half of the output when we were using Stride of one.

The third hyperparameter is *padding*. Imagine that we will use a filter size of 5x5x1 and striding of 1. In this case, we would have, thinking only in the red filter, a filtered image of size 28x28x1. But now, imagine that we would like to preserve the image input size, 32x32 in this example. What we can do is add two borders of zero around the input volume, making it of size 36x36x1. If we do the exactly same math as explained in the striding, then we will have a final output volume of size 6x32x32x1, i.e., each slice of it containing a filtered image of the same dimension as the input.

More general, we can define Equation 2.54 as a general equation to obtain spatial dimensions to the output volume O when applying convolutions for a image of size WxW, a filter size of size FxF, a Padding of size P and using Stride S. It's important to keep in mind while observing this equation that, in Deep Learning, is very common to use squared images, that's the reason that we only use W. Also, whenever we apply padding, we apply it in all the borders of the input image.

$$O = (W - F + 2P)/S + 1 \tag{2.54}$$

An important property of the CNN is that it does *parameter sharing*. It means that each slice of the filter volume has the same weights and only one bias. To make it more general, if we have a input volume of size WxWx1 and K filters with dimensions FxFxN, then each filter will have FxF unique weights and one bias that will be *shared* through the whole convolution process for that particular filter. In the end, we will have a total of KxFxF different weights and K biases. It reduces drastically the number of weights and biases in the neural network, making it faster to be trained.

2.10.3 CNN Layers

A CNN layer can be classified as one of the following four types: *Convolutional Layer* (ConvLayer), *ReLU Layer*, *Pooling Layer*, or *Fully-Connected Layer*. The

convolutional and fully-connected layers have parameters that are properly adjusted in the training phase using the same gradient descent and backpropagation algorithm that we described before. The pooling layer does not need parameters and always implement a fixed function, i.e., a function that will receive an input and apply a mathematical operation without any dynamic learned parameters in this function.

Convolutional Layer

To understand the convolutional layer, let's make an analogy using car headlights. Imagine that you are driving your car at night in Los Angeles and you see a big picture in a huge building wall. You point your car headlight to the picture to try to figure out what it is. Since the picture is big and your headlights have limited range, you can only highlight a small part of the picture. Also the light illuminates better the center than the borders. On this analogy, consider your car headlight as a *convolutional filter* and the difference in illumination as the *weights* of this filter. The region that the headlights is illuminating in the picture is known as *receptive field*. The convolution operation is the operation that happens with the interaction between the filter weights and the values of the picture pixels. We perform a element-wise multiplication of the filter weights and pixels values, summing up the multiplications, what results in a number. For the sake of simplicity, assume that in this analogy the car is illuminating a $3x3$ area in the wall. The filter is shown in Figure 2.20 for the car analogy.

Continuing with this analogy, the visualization of a small part of the image is not enough to figure out what the picture is about. We need to *stride* our filter through out the whole wall to highlight important features in other parts of the picture, as shown in Figure 2.21.

After the filter is convolved through out the entire image, we end up with an *activation map* or *feature map*. This map highlights the important features that the filter extracted from the image. However, it may not be, and it is almost always the case, sufficient to have only one activation map. We need to have a *volume* of activation maps. Recall the CNN API described before. In this analogy, another activation map would come from a friend's car headlights with a different headlights, meaning different filter weights. Consider the same illumination area size of $3x3$, as

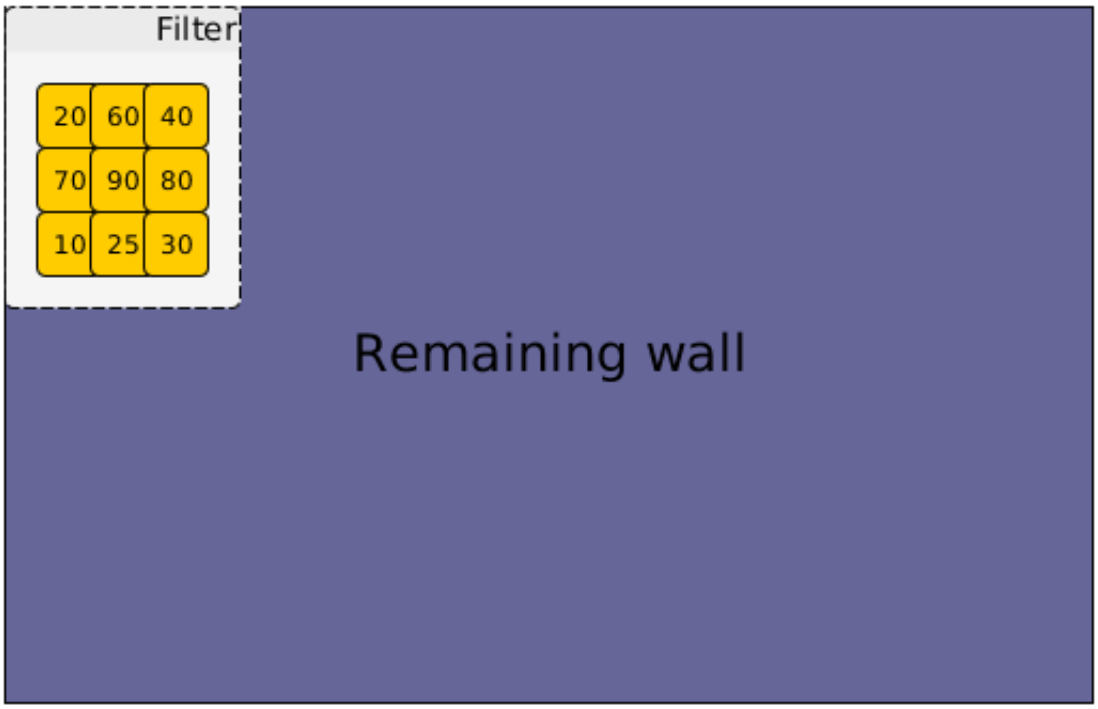


Figure 2.20: Convolutional filter is the yellow box and the numbers inside it are the *weights* of the filter. The weights highlight elements from the center to the borders. That is not always the case when we are working with CNNs, tough.

mentioned above, but with different weights for the convolution.

ReLU Layer - Rectified Linear Units)

After each convolutional layer, it is convention to apply a nonlinear layer, or activation layer. The purpose of this layer is to introduce nonlinearity to a system that has basically been computed by linear operations, for the convolutional layers, or just element wise multiplications and summations. In the past, nonlinear functions like *tanh* and *sigmoid* were used, but researchers found out that ReLU layers work better because the network is able to train much faster because of the computational efficiency, and without causing significant loss of accuracy. Also, it helps to alleviate the vanishing gradient problem, that makes the lower layers, of the network, to get trained very slowly, since the gradient decreases exponentially from layer to layer. The ReLU layer applies the function $f(x) = max(0, x)$ to all the values of the input volume. Basically, this layer changes all negative activations to 0. This layer increases the nonlinear properties of the model and the overall network without affecting the receptive fields of the Convolutional Layer [7].

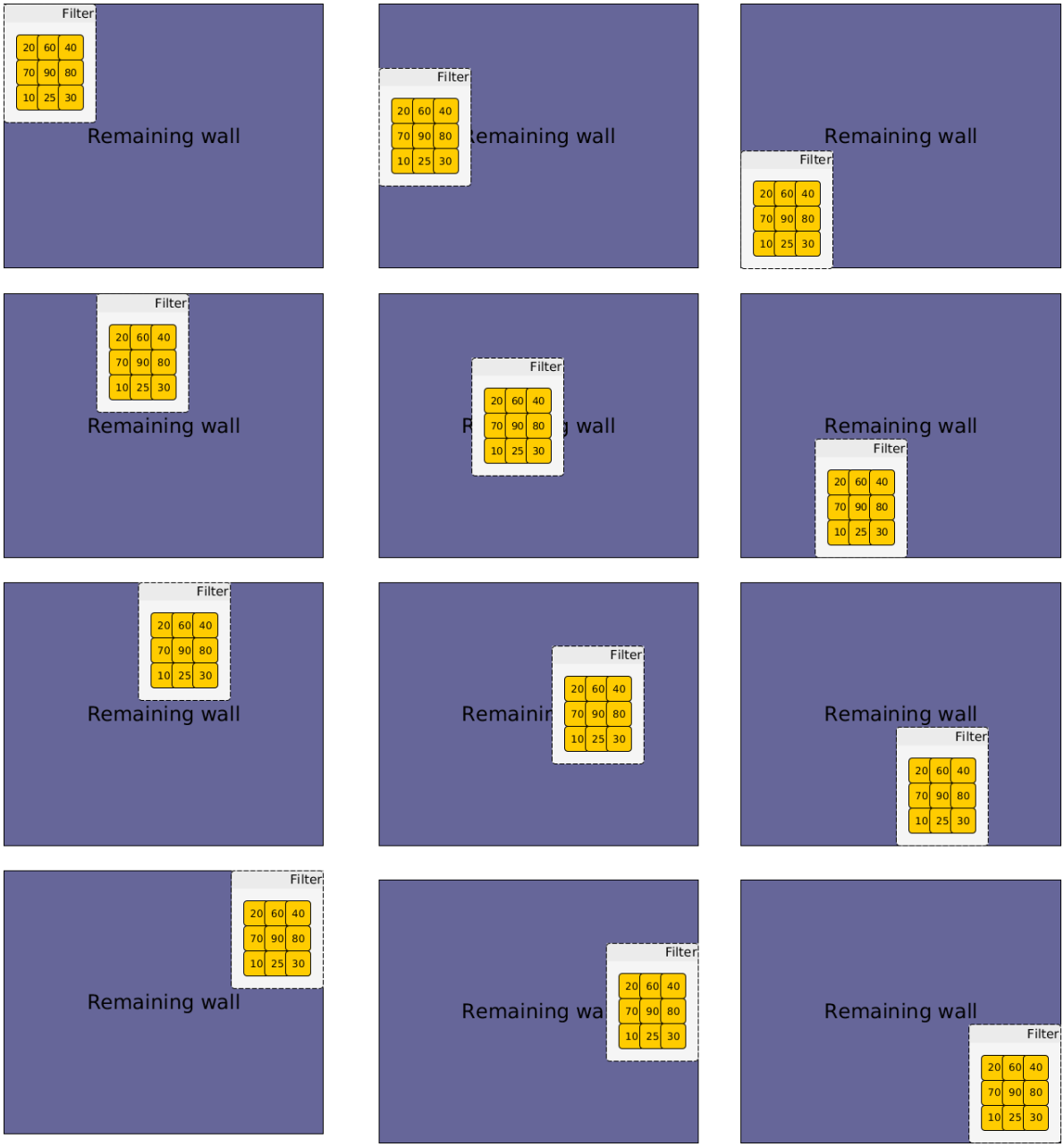


Figure 2.21: Filter striding. Columns represent the filter striding horizontally.

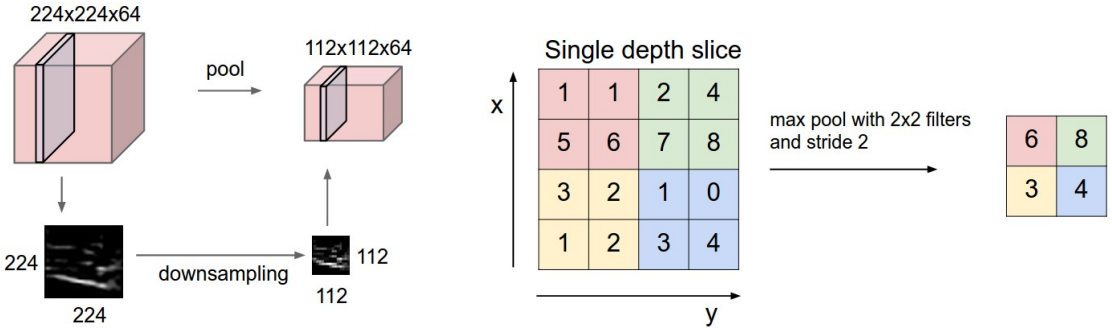
Pooling Layer

The pooling layer, also known as downsampling layer, has the function to progressively reduce the spatial size of the image [2]. Think about the pooling layer exact as we humans do while trying, for example, to identify another human being. We don't need the full representation of the human face to identify it. A small (*downsample*) is sufficient. It works because our brain identify elements in the face related with other elements, e.g., the mouth is below the nose. And that is essentially what

we have in the filter response explained in subsection 2.10.3. The filter learns and highlights the important features, afterwards we can compress, downsample, the information and move forward.

The *Pooling Layer* serves for two purposes: it reduces the computational cost and helps preventing *overfitting*. You can use a number of pooling variants, like Max Pooling, Average Pooling, L2-Norm Pooling, etc. It is important to mention here, that no matter which variant you use, all of them will select a block in the activation map and apply an operation, Max, Average, L2-Norm, etc, that will reduce the spacial dimensionality.

Figure 2.22 shows an input volume being downsampled using a Max Pooling layer.



(a) The input volume of size $224x224x64$ is pooled by a filter size $2x2$, stride 2 to an output volume of size $112x112x64$.

(b) The most commonly downsampling operation used is *max pooling*. It is shown here with a stride of 2. That is, each max is taken over 4 numbers, a $2x2$ square).

Figure 2.22: Pooling operation [2].

As we will see, in section 3.4, the max pooling has a interesting property that allows the network to achieve translation invariance over small spatial shifts in the input image.

Fully-connected layer

The Fully-connected layer in CNN works the same way as the hidden layers commented in section 2.3, i.e., all the neurons in this layer are fully-connected with the neurons in the previous layer. Generally, this layer is used in CNNs as the last layer and has the function to classify the final pixel in some class. Figure 2.23 shows a Convolutional Neural Network that attempts to classify an image in one of ten different classes using the well-known dataset CIFAR-10 [6]. Note that the Fully-

Connected layer is at the end with the intention of classifying the final result. The reason for being at the end is obvious: If we implement it in every layer, we would end up with the same excessive neuron connections as in the Neural Networks.

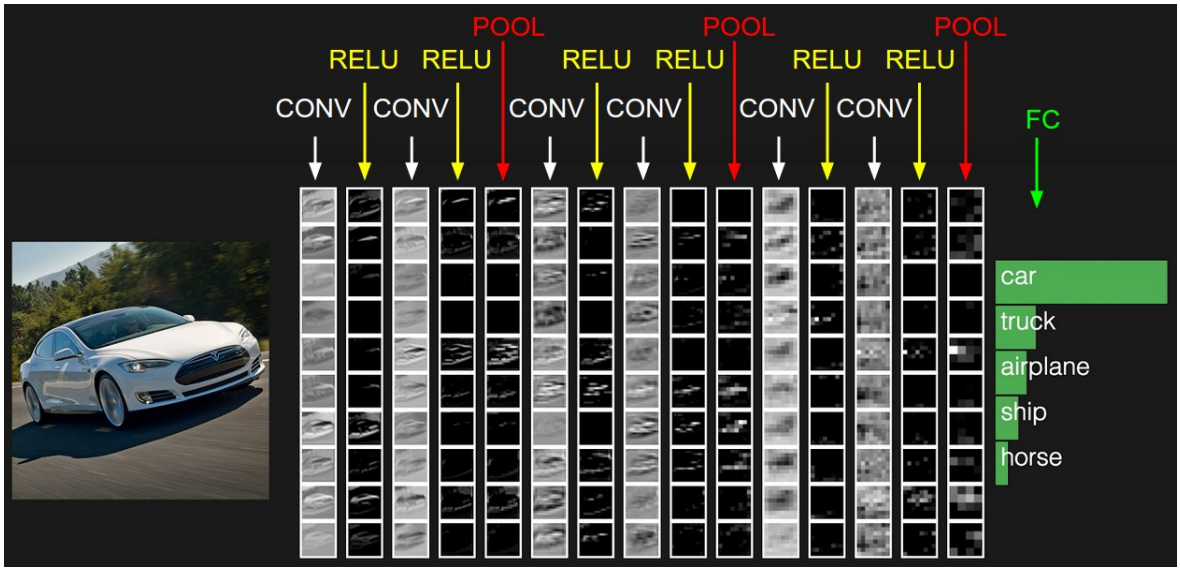


Figure 2.23: A CNN based on VGG Net [5] architecture using CIFAR-10 dataset [6]. Although CIFAR-10 has 10 classes for each image, for simplicity, this example shows only the top five classes [2].

Chapter 3

The Modern History of Object Recognition

> What I cannot create, I do not understand.
>
> Richard Feynman (1988)

The human visual system of recognition is pretty much fantastic. Humans can classify, localize, recognize, detect and segment practically all objects present in one image or scene. But how can the computer do something similar? In this chapter, we will describe the challenges in computer vision briefly and we will concentrate on the topic that this work is trying to solve: *segmentation*. We will talk about the Fully Convolutional Networks for Semantic Segmentation [37] that will be the basis to understand two important architectures that we use here: U-Net [10] and SegNet [11].

3.1 Challenges in Computer Vision

We can divide the tasks that one can do in image using computer vision essentially in four different types: Image classification, Object localization, Object recognition and Segmentation. Let's discuss what means each of these tasks.

3.1.1 Image Classification

Image classification is a task where the computer, i.e., an algorithm, tries to classify an image based on the **dominant object** inside it. This is essentially the case from Figure 2.23. In this example, the figure is about a car in a road with some trees and the sky in the background. What is the dominant image in the image? Clearly the dominant object is the car and that is the reason that the CNN in this example outputs "car".

3.1.2 Object Localization

Object localization goes further and not just classifying the dominant object but also surround it with one bounding box in the image. It's important to have in mind that the bounding box can contain part of other objects, i.e., it's not the perfect localization, but based on the dominant object exactly as define in the Image classification task. Figure 3.1 shows the difference between object classification and object localization.

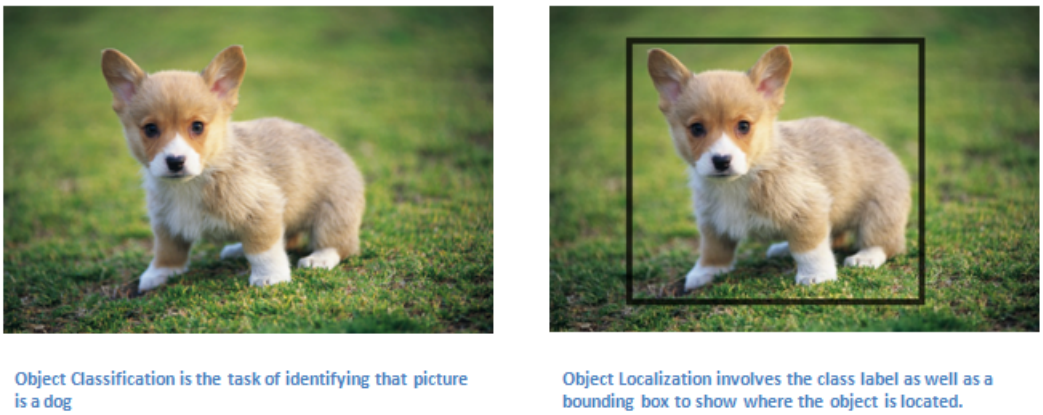


Figure 3.1: Classification and Localization of an object. [7]

3.1.3 Object recognition

The object recognition task is more extensive and, unlike localization that only surrounds the dominant object with a bounding box, in this task we are trying to classify and surround **all** objects in the image, not only the dominant one. Figure 3.2 shows the objects being recognized in a scene.

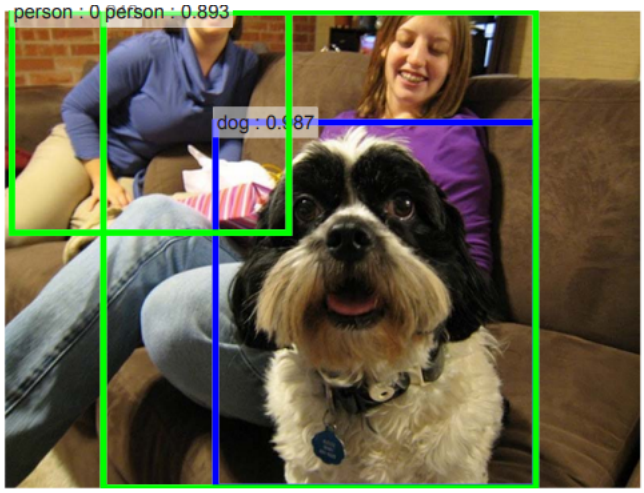


Figure 3.2: Object recognition [8].

3.1.4 Segmentation

Neither tasks mentioned tried to delimit perfectly an object in the scene. The closer that was achieved was to draw bounding boxes surrounding the objects. The image segmentation task tries to address this problem by making contours in the objects. The goal is to have these contours the best possible. Figure 3.3 shows objects being segmented. The segmentation task can also be divide in *semantic segmentation* and *instance segmentation*. Semantic segmentation doesn't care about objects of the same class, i.e., three dogs in a scene, for example, will be segmented and painted with the same color. It does not distinguish between elements of the same class. Instance segmentation, however, paint objects of the same class with different colors, i.e., three dogs in a scene, for example, will be segmented and painted with different colors because essentially they are different *instances*.

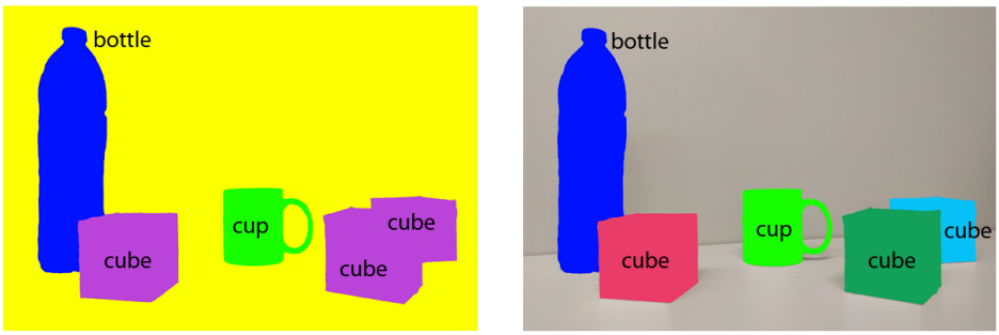


Figure 3.3: Image segmentation. On the left, semantic segmentation with all *cubes* belonging to the same class. On the right, instance segmentation is performed and each cube is belonging to a different class [9].

3.2 Fully Convolutional Networks for Semantic Segmentation

Recall from section 2.10 that Convolutional Neural Networks tries to solve a classification problem by using a series of convolutions, poolings and at the end a uses a fully connected layer. Is it possible to apply the same methods and concepts that we learn for CNNs to solve the semantic segmentation problem? It seems that a lot of the concepts can be utilized here!

Let's start with a first intuition: Instead of using a CNN to classify the whole image, what can be done is to divide the input image in patches of equal size and to use the CNN to classify the *pixel at the center* of the patch as shown in Figure 3.4.

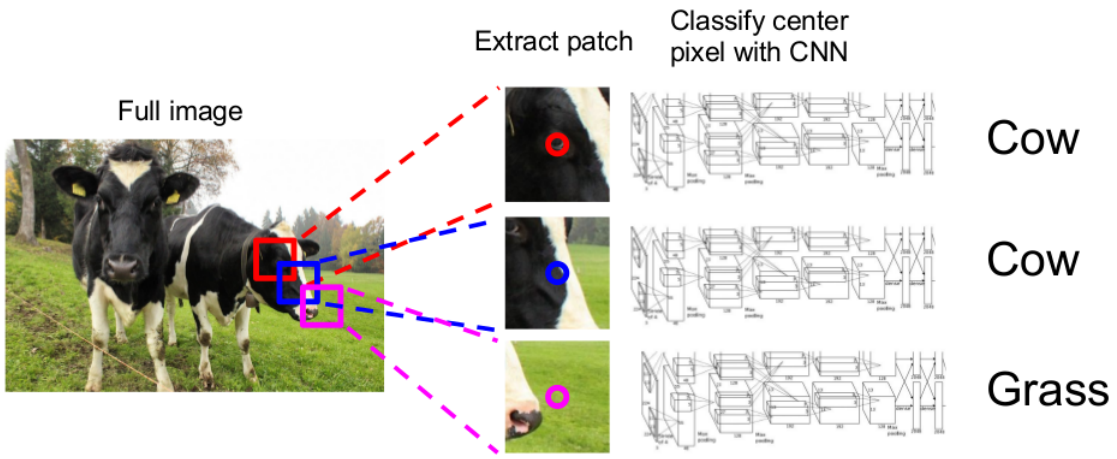


Figure 3.4: Image segmentation using patches. [2]

Although this is a creative and a simple approach, it is extremely inefficient since it is not reusing shared features from overlapping patches due to the fact that we are training the patches separately [2].

The next step is to implement what we call a *Fully Convolutional Neural Network* [37]. The idea is that you receive a image of size HxW and you pass this image through a deep network that has only convolutional layers, i.e., no fully-connected layers at the end. The only concern here is that your convolutional layers must be configured, striding and pooling, in a such a way that they preserve the size of the input. This network will produce in the output a *classification volume* of size $HxWxC$, where C is the available classes. With the classification volume at hand, you just take the *argmax* for every pixel, and this will be your final classification at

the pixel level. To train this network, you can take the average cross entropy loss through all pixels and then backpropagate the same way as we saw in section 2.6. Figure 3.5 shows a fully convolutional network.

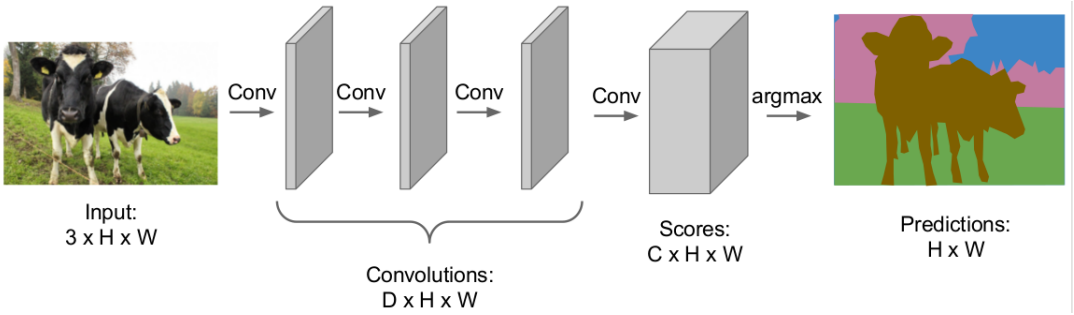


Figure 3.5: Fully convolutional neural network architecture. [2]

This approach is much better than the first one but we are still having a problem: Note that we are not using any *pooling* layer. The pooling layer, as explained in Subsection 2.10.3, has the function to downsampling the input, decreasing its sizing. This makes the training faster. This makes this configuration very computationally expensive. We need to figure out a way to make training simpler and faster.

There is a way of achieving this goal, and it is presented in Fully Convolutional Networks for Semantic Segmentation Section [37]. The trick is to separate the convolutional layers in two symmetric groups: downsampling and upsampling. The first one will run exactly as we already saw in the CNN Section 2.10, i.e., it will apply convolutional filters and reduce the spatial size of the image through pooling layers. The upsampling group will get the reduced images and increase the spatial size of the image output until it has the same size of the input. This improves our training by reducing size of the image in the downsampling layer instead of working with the original size in every layer as before. Figure 3.6 shows this architecture.

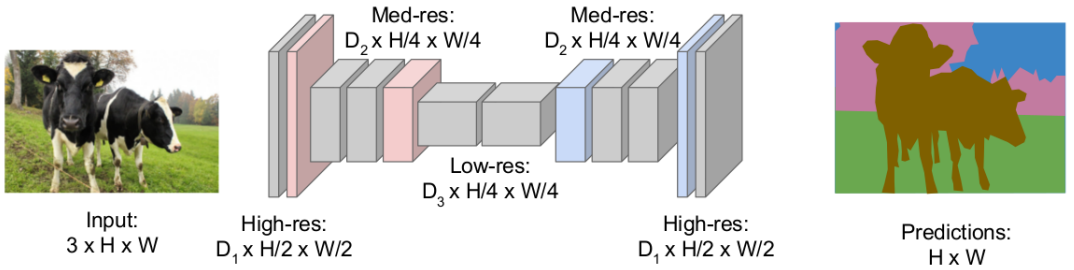


Figure 3.6: Fully Convolutional Neural Network architecture with upsampling [2].

Like we explained in subsection 2.10.3 that there are some different functions that

can choosen to be used in a pooling operation. This is also true for the "unpooling" in the upsampling layers. Figure 3.7 shows two examples.

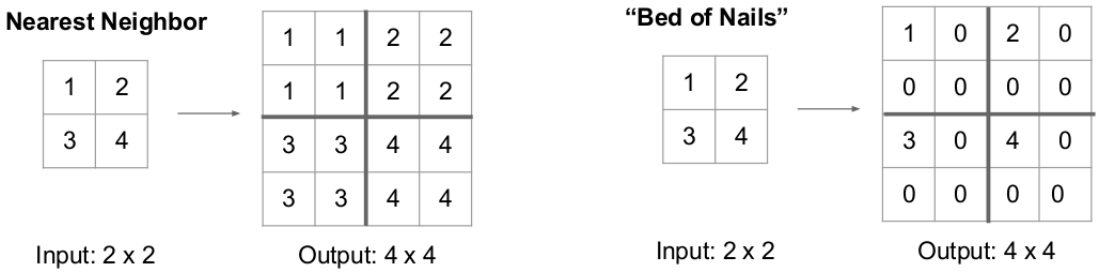


Figure 3.7: Some examples of unpooling operation. [2]

In the nearest neighbor approach, we just repeat the value in the receptive field in the output. In "Bed of Nails" approach, the value in the receptive field of the filter are allocated in the left corner of the expanded output and all the other positions are filled with zeros.

There is another important unpooling operation called *max unpooling*. In this operation, we are taking advantage of the symmetry of the downsampling and upsampling layers and keeping in every downsampling operation the pixel position that was took as the maximum in the receptive field. With that at hand, when we apply the unpooling operation in the respective symmetric upsampling layer, we use these pixel positions from the downsampling layer. Figure 3.8 shows this operation.

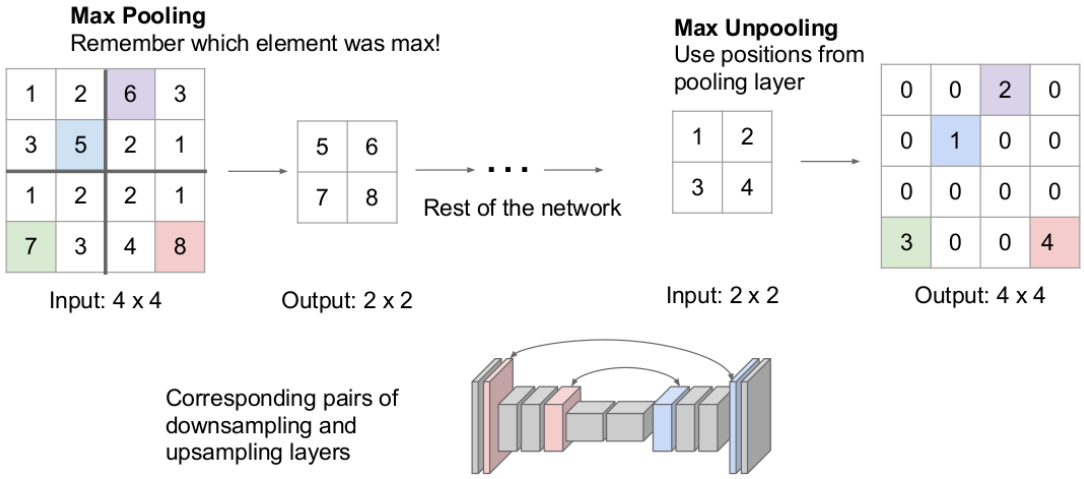


Figure 3.8: The max unpooling operation [2].

Max unpooling operation has one advantage over Nearest neighbor and Bed of nails. When we are talking about semantic segmentation we want that our pixel class prediction to be as perfect as possible. The problem is that when you do, for

example, Max pooling in the downsampling layers, you are loosing spatial information in some sense. But when you keep the index in the downsampling layer that generate the max pixel value and you use this information to do the max unpooling operation in the corresponding symmetric upsampling layer you are preserving the spacial information that was lost during downsampling due to max pooling operation.

The upsampling operations, just described, are fixed functions. They don't describe how we actually do the upsampling. For this we need to do an operation called *transposed convolution*. This operation is a normal convolution but it has an opposite interpretation from the stride convolution that we saw in section 2.10. Recall from the stride convolution that we slide the filter through out the input and for every filter position on the image we make a dot product between the filter and the input. The stride gives the ratio between movement in input and output, i.e., filter moves *strides* pixels in the input for every pixel in the output. Now, in transpose convolution the stride gives the ratio between movement in output and input., i.e., filter moves *strides* pixels in the output for every pixel in the input. Another point to note is that in transpose convolutional the input will give weights for the filter, i.e., for every value in the filter we get the corresponding pixel in the input and apply it as a weight in the filter instead of making inner products between the filter and the input. Figure 3.9 shows how it works.

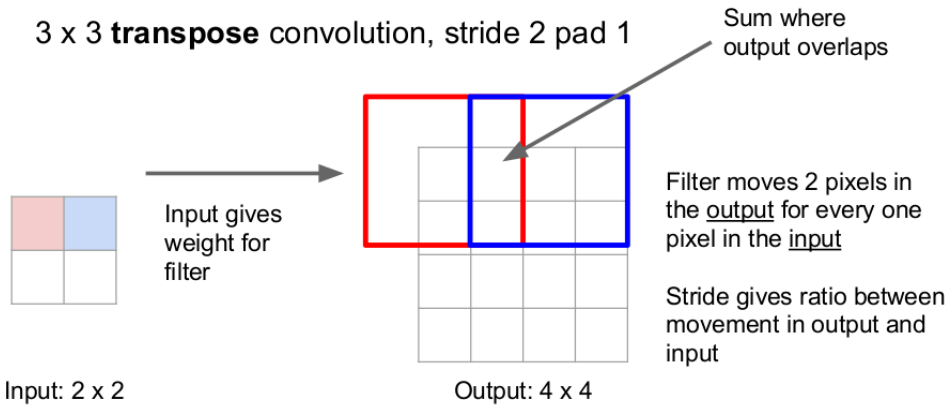


Figure 3.9: Transpose convolution [2].

3.3 U-Net

The typical use of convolutional networks is on classification tasks, where the output for an image is a single class label. However, in many visual tasks, especially in biomedical image processing, the desired output should include localization, i.e., a class label is supposed to be assigned to each pixel. Moreover, thousands of training images are usually very difficult to be obtained in the biomedical field [10].

As mentioned in the 3.2, a simple idea to segment an image is to divide it into patches and classify the patch's central pixel. Ciresan et al. [38] approach does exactly this. With this architecture, they won the EM segmentation challenge at ISBI 2012 by a large margin.

Although this is very impressive, their architecture has two major drawbacks: First, it is quite slow because the network must be run separately for each patch, and there is a lot of redundancy due to overlapping patches. Second, there is a trade-off between localization accuracy and the use of context. Larger patches require more max-pooling layers that reduce the localization accuracy, while small patches allow the network to see only little context. [10]

The purpose of U-Net is that it modifies the network architecture proposed by Long et al. [37] and extend it to work with very few images and more precision segmentations. The main idea of U-Net is that it implements two paths: contraction and expansion path. The contraction path works the same way as regular CNNs, applying downsampling successively, pooling operation. The expansion path uses upsampling operators as described in the previous sections.

In order to localize, high resolution features from the contracting path are combined with the upsampled output. A successive convolution layer can then learn to assemble a more precise output based on this information. One important modification introduced by U-Net is that in the upsampling part there is also a large number of feature channels, which allow the network to propagate context information to higher resolution layers. As a consequence, the expansive path is more or less symmetric to the contracting path, and yields a u-shaped architecture [10]. Figure 3.10 shows the U-Net architecture.

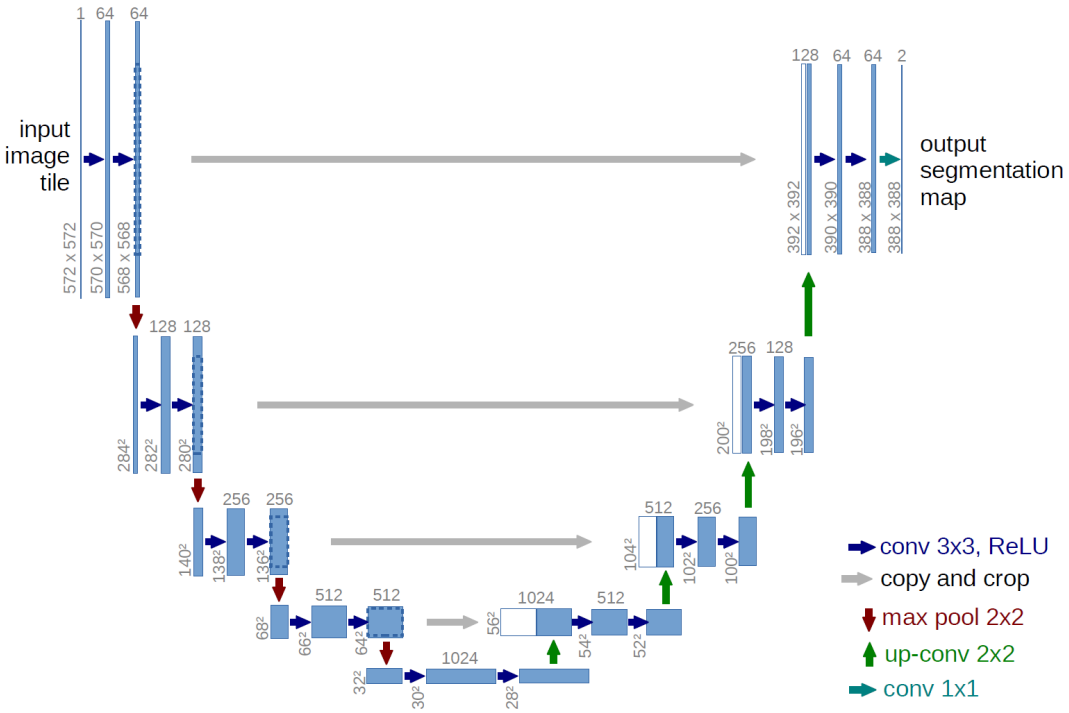


Figure 3.10: U-Net architecture [10].

3.4 SegNet

SegNet is a deep convolutional encoder-decoder architecture for image segmentation. As we can be seing in Figure 3.11, in terms of architecture it is a little bit similar with U-Net: It has a encoder, contraction path equivalent for U-Net, and a decoder, expansion path in U-Net. However, a key and important difference among them is that U-Net combines the high resolution features from the contracting path with the upsampled feature map by copying *the entire map*. SegNet, in turn, just stores the max pooling indices in each encoder feature map and use it to perform non-linear upsampling of the input feature maps. In terms of computational memory, this is much more effective. According to [11], reusing max-pooling indices in the decoder has three main advantages: (i) it improves boundary delineation, (ii) reduces the number of parameters enabling end-to-end training, and (iii) this form of upsampling can be incorporated into any other encoder-decoder architecture with minor modifications. However, although having a higher memory cost and inference time, architectures that uses the full feature map from the encoder into the decoder layer still have better performance [11].

The idea of reusing the max pooling indices in SegNet was inspired from an

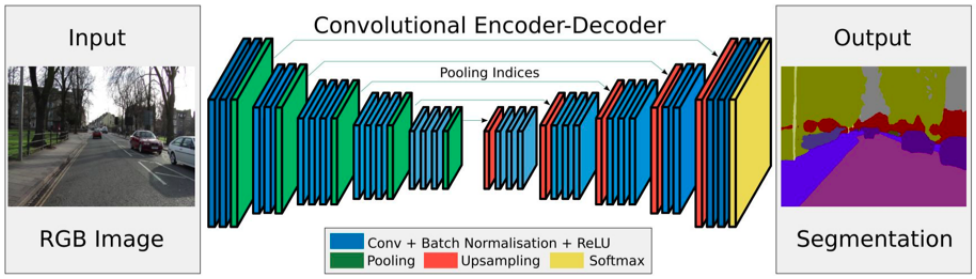


Figure 3.11: SegNet architecture [11].

architecture designed for unsupervised feature learning [12] shown in Figure 3.12.

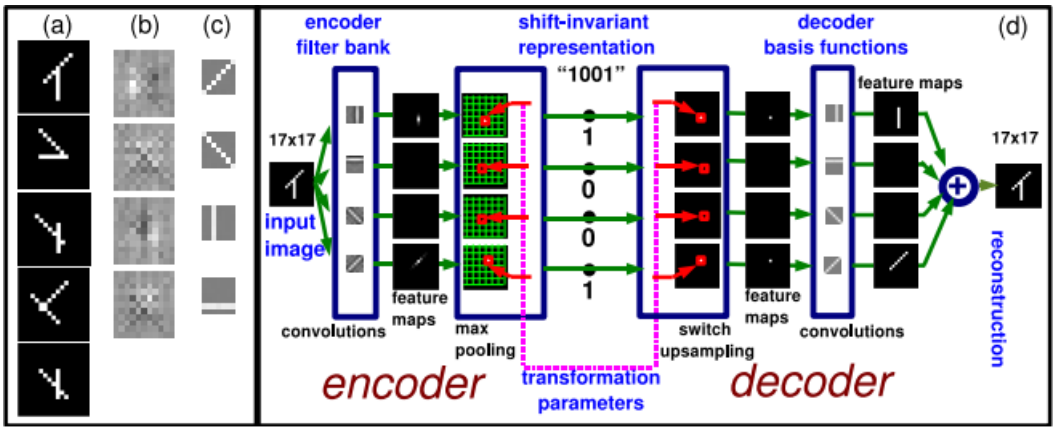


Figure 3.12: Convolutional auto-encoder [12].

Note how the max pooling operation in this architecture produces a shift-invariant representation of the learned features. To understand this, imagine that you have the feature map F and you want to apply a $2x2$ max pooling window on it. The max pooling operation here is just extracting from the feature map F the highest value from the $2x2$ window. We can see in M that the resulting operation of the max pooling. But it seems that we can *encode* this learned representation: Instead of using the magnitude of the pixels in M, we create a new coded representation C setting just the *index* of the highest pixel value in the feature map to 1 and all others to 0. We can say, for example, that for input feature map F we have the representation code 1001. Now, if you move the feature map F a little bit to the top, bottom, left or right, your final code will still be 1001 because it is still the max value in the window. This way the network is *coding* the features, retaining *what* is important and being shift-invariant.

$$F = \begin{bmatrix} 2 & 3 & 0 & 0 \\ 5 & 1 & 0 & 0 \\ 0 & 0 & 3 & 8 \\ 0 & 0 & 7 & 2 \end{bmatrix}$$

$$M = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 8 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$C = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

This idea, that comes from the max pooling operation, is used both on U-Net and Segnet.

Chapter 4

Methodology

> The purpose of computing is insight, not numbers.
>
> R. Hamming.

In this chapter it is described the methodology we use in this work. We will describe the datasets used, pre-processing steps applied, and the dataset split and augmentation. Afterwards, we describe the training phase and its *hyperparameters*, the tools and the network architecture. Finally, we show the technique that we used to compare the result images against the ground-truth.

4.1 Understanding the data

Before going into the details of the methodology, we need to understand the dataset images that we use to train the deep neural network. In the coming subsections we show how they are structured and how we can augument it to make the trained model more robust.

4.1.1 The Datasets

In Magnetic Resonance Image, MRI, there are three different types of images: Axial, Sagittal and Coronal, shown in Figure 4.1.

As defined in [13]:

- Axial plane: Transverse images represent "slices" of the body;

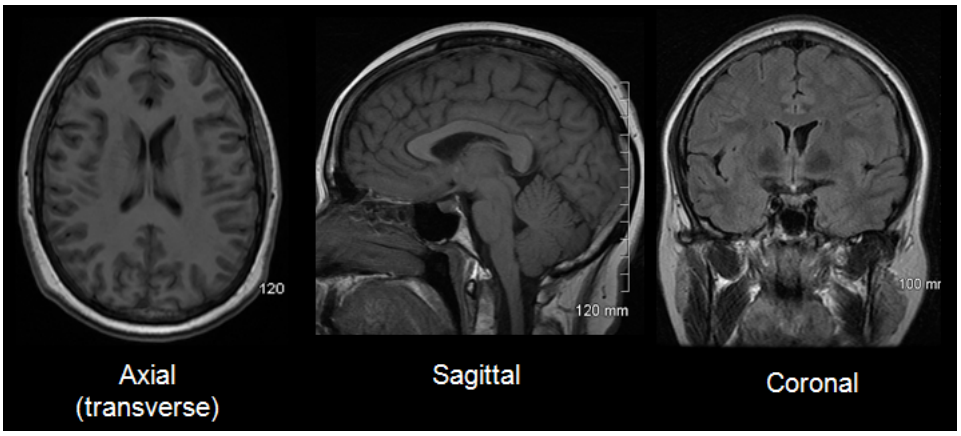


Figure 4.1: MRI Planes. [13]

- Sagittal plane: Images taken perpendicular to the axial plane which separate the left and right sides, Lateral view;
- Coronal plane: Images taken perpendicular to the sagittal plane which separate the front from the back, Frontal view.

From Figure 4.1 it can be seen that the corpus callosum are completely observable in the sagittal plan, although part of appears also in the other view planes. For this reason, in this work we are only training, consequently evaluating the model, using sagittal images.

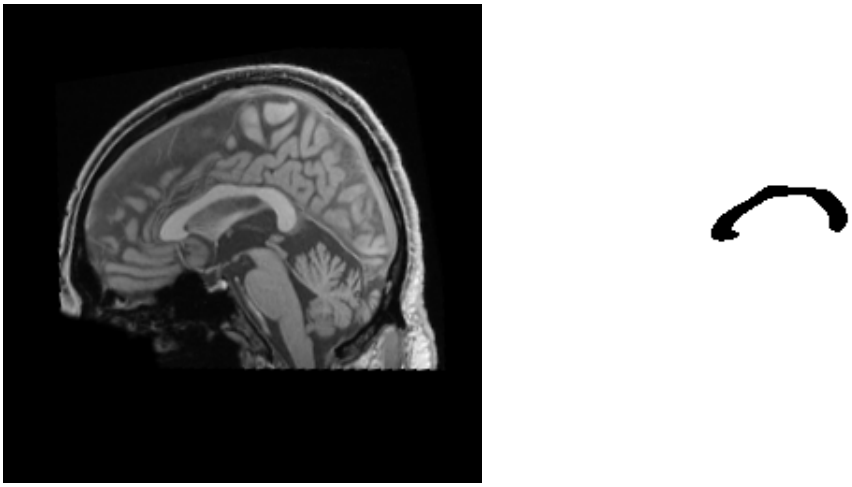
Two different datasets are used: OASIS and ABIDE. The Open Access Series of Imaging Studies, OASIS, is a series of magnetic resonance imaging data set publicly available for studies and analysis. The initial data set consists of a cross-sectional collection of 416 subjects aged between 18 to 96 years old. A hundred of them are older than 60 years, which have been clinically diagnosed with very mild to moderate Alzheimer's disease. The subjects are all right-handed and include both men and women. Each subject includes, three or four individuals T1-weighted magnetic resonance imaging scans were obtained in single imaging sessions. Multiple within-session acquisitions provide extremely high contrast-to-noise ratio, making the data amenable to a wide range of analytic approaches including automated computational analysis [39]. In this work, we selected only a subset containing 1806 sagittal images, 903 original images and 903 ground truth images, from the original dataset.

The Autism Brain Imaging Data Exchange, ABIDE, has aggregated functional and structural brain imaging data collected from laboratories around the world to

accelerate our understanding of the neural bases of autism. With the ultimate goal of facilitating discovery science and comparisons across samples, the ABIDE initiative now includes two large-scale collections called ABIDE I and ABIDE II. Each collection was created by the aggregation of datasets independently collected across more than 24 international brain imaging laboratories and have been made available to investigators throughout the world, consistent with open science principles, such as those at the core of the International Neuroimaging Data-sharing Initiative [14]. The original ABIDE dataset has all images in *nii* format. A python program was written to convert the original format into *tiff* files, generating a total of 2200 sagittal images, 1100 original images and 1100 ground truth images. Using these two datasets, we have a reasonable database, containing images of normal subjects and individuals with autism, generating a heterogeneous and not only strict database.

4.1.2 Data Pre-Processing

As described in the previous section, we have a total of 2003 images pairs containing in each pair one image the whole brain and in the other the ground truth. An example pair is shown in Figure 4.2.



(a) Whole brain training image from ABIDE dataset.

(b) Corpus Callosum ground truth extracted from image (a) from ABIDE dataset.

Figure 4.2: MRI sagittal image and ground truth from ABIDE dataset.[14]

Although the dataset is ready to be used, there are some pre-processing steps that one must follow to get the images ready train the neural network. In this work,

the following steps are taken as data preparation:

- Image conversion to gray scale;
- Image resizing to $128x128$;
- Feature scaling, z-score.

The image conversion to gray scale is a simple way to make the training computationally less expensive. Instead of having the three RGB channels, we use only one channel for each pixel value ranging from 0 to 255. Afterwards, we resize the image to a smaller resolution $128x128$ with the same goal: make the training faster.

In general, it is not safe to feed into a neural network data that contains relatively large values or data which are heterogeneous, for example, data where one feature ranges between [0,1] and another one that ranges between [100-200]. This could trigger large gradient updates that will prevent the network from converging [40]. This is avoided by applying a feature scaling in the dataset. The feature scaling, or z-score, is calculated as the *mean* (μ) and *standard deviation* (σ) from the *training set*, which are used to scale each dataset:

$$X_{train} = \frac{X_{train} - \mu_{train}}{\sigma_{train}} \tag{4.1}$$

$$X_{val} = \frac{X_{val} - \mu_{train}}{\sigma_{train}} \tag{4.2}$$

$$X_{test} = \frac{X_{test} - \mu_{train}}{\sigma_{train}} \tag{4.3}$$

An important note, that always causes confusion, is that we should use only the mean and standard deviation from the training data and apply the equations above on all separate datasets. The reason is that, by machine learning principles, your test set should be reasonably representative of the training set, otherwise you would be comparing apples with oranges, which makes no sense. With this assumption, using only the training set mean and standard deviation, we avoid that sampling errors in the test set negatively bias the predictions [41].

4.1.3 Data Augmentation

The aim of the deep learning model is to be comprehensive enough to deal with whatever image that is used in the system. That brings up an important question: *How to asure that the system will be robust enough if we only have 2003 images available for training it?* The answers for this question is *data augmentation* and it helps to avoid overfitting, as described in subsection 2.9.2, by generating more training data from the existing ones using a number of random transformations. The idea is that in the training step, your model will not see exactly the same picture twice, exposing the model to more aspects of the data, leading to a better generalization [40].

In this work, we use two affine transformations to augment our data: *translations* and *rotations*. An affine transformation is a function between two affine spaces that preserves points, straight lines and planes. It means that the rotations and translations that we apply to the original images do not change the image structure, i.e., a brain MRI will keep showing an image of a brain, a little displaced and/or rotated, but still a brain. Table 4.1 shows the rotations and translations that were used in this work.

Total	Total Training	Translations	Rotations	Aug. Factor	Total Final
2003	1402	[(-1, -1),(-1, 0),(0, -1),(0, 0)]	[-1, 0]	8	11216

Table 4.1: Data augmentation table.

The understanding of this table is simple. We start with the original set of 2003 images. We use the first 1402 images pairs as training images. We explain the dataset splitting in the next sections. For each image, we apply a translation pair (x, y) on each pixels, and apply a rotation angle a, in degrees. As we have four translation pairs and two angles, we are augmenting the total training set size in a factor of 8, Augmentation factor, generating 9814 new images, resulting in a final training set of size 11216.

4.1.4 Dataset Split

We need to separate our dataset in training, validation and test sets. From the original 2003 images pairs, we divide it into 70%, 10% and 20% for training, validation

and test, respectively. As described in section 4.1.3, we augment only the training set. Validation and test sets are not modified. Table 4.2 shows the final dataset size for each one.

	Training	Validation	Test
Dataset #1 (Initial)	1402	200	401
Dataset #2 (After augmentation)	11216	200	401

Table 4.2: Training, Validation and Test sets sizes.

4.1.5 Training

With the dataset ready to be used, the next step is to actually train the Neural Network. In this work, we train the U-Net [10] model with a slight change in the cost function from the original paper. In the original paper, the authors use a Weighted cross-entropy [42] as the loss function. We change it to use the Dice loss function defined in 2.7 because when the level of pixel imbalance increases, Dice has better results than Weighted cross-entropy [42, 43].

hyperparameters-note As we will see in the next chapter when we present the results, each one is obtained training the network with a different set of hyperparameters. Essentially, the hyperparameters that we change for each test are:

- *boolean* : *use_dropout* - To add, or not, dropout as regularizer.
- *boolean* : *use_batch_normalization* - To add, or not, batch normalization as regularizer.
- *int* : *dropout_rate* - If true, it indicates the probability of dropping out a neuron.
- *int* : *number_of_epochs* - Number of epochs for training.
- *int* : *batch_size* - Size of mini batch.
- *int* : *kernel_size* - The size of the convolutional filters.
- *int* : *initial_volume_size* - The initial amount of filters to start the training.

We train the neural network using mini-batch gradient descent with Adam optimizer [44] and a batch size of 32. In each training epoch, we shuffle the training data what ensures that bias in the presentation order does not adversely affect the final result [45]. We use the deep learning framework TensorFlow [46] and the high-level API [47] to implement the training, obtain the model and evaluate it on test set.

In Table 4.3 it can be seen the trained model, layer by layer, i.e., the number of parameters in the layer, type of the layer, and what is the output volume generated by the layer. In Figure 4.3, we show the architecture of the trained model. It's important to note that, as we stated in Section 4.1.5, the model is trained with a lot of different configurations to compare the results. In Table 4.3, we present the model with the last test configuration, and it does not alter the understanding of the whole model, since the architecture is the same for all tests, and just some *hyperparameters* are changed.

4.1.6 Model Evaluation

After training, we have the model ready for evaluation on our test set. Each evaluation on the test set will produce a result image that needs to be compared with the *ground truth*. In this work, the model evaluation is done using the Structural Similarity (SSIM) index [15].

One of the simplest and easiest way to compare two images is using the minimum square error (MSE), Equation 2.37. It is appealing because they are simple to calculate, have clear physical meanings, and are mathematically convenient in the context of optimization [15]. However, in the comparison of two images, there is a strong drawback: It does not indicates the *structure* of the images when the error is being penalized. Figure 4.4 shows a classical example of this problem: all of the images have the same MSE error even though the human vision system can perceive huge differences among their qualities. The SSIM index comes in place to solve exactly this problem. In this same figure, note that SSIM is very different between images and also gives higher value to the images that are visually better.

The luminance of the surface of an object being observed is the product of the illumination and the reflectance, but the structures of the objects in the scene are independent of the illumination and contrast [15]. The general SSIM framework can

be seen in Figure 4.5.

The SSIM between two images x and y is composed essentially by three components: a luminance comparison function $l(x, y)$, a contrast comparison function $c(x, y)$ and a structural comparison function $s(x, y)$. The luminance μ_x is obtained by taking the mean intensity among the N pixels of the image, shown in Equation 4.4. Following the framework, we use the standard deviation as a estimator of image contrast σ_x, shown in Equation 4.5. The structure comparison is made in the normalized signals with normalization being copmuted as defined in Equation 2.32. With all these variables, the SSIM components can be defined for luminance and contrast, as in Equations 4.6 and 4.7, respectively. For the structure component, as we normalize it, the correlation between $(\frac{x-\mu_x}{\sigma_x})$ and $(\frac{y-\mu_y}{\sigma_y})$ is defined in Equation 4.8. Equation 4.9 uses this correlation to obtain the structural comparison function. Finally, we can define SSIM as a linear combination of $l(x, y)$, $c(x, y)$ and $s(x, y)$ as shown in Equation 4.10. In the original paper, the authors set α, β and γ to 1, we did the same. The SSIM index can be written as in Equation 4.11. They calculate the SSIM index using M patches of size $11x11$ and afterwards they average the these local SSIM. Equation 4.12 shows the mean SSIM index, MSSIM. As a side note, in all equations the parameters C_x are added to asure numerical stability, i.e., avoid division by zero.

$$\mu_x = \frac{\sum_{i=1}^{n} x_i}{N} \tag{4.4}$$

$$\sigma_x = \frac{\sum_{i=1}^{n} (x_i - \mu_x)^2}{N-1} \tag{4.5}$$

$$l(x,y) = \frac{2\mu_x\mu_y + C_1}{\mu_x^2 + \mu_y^2 + C_1} \tag{4.6}$$

$$c(x,y) = \frac{2\sigma_x\sigma_y + C_2}{\sigma_x^2 + \sigma_y^2 + C_2} \tag{4.7}$$

$$\sigma_{xy} = \frac{\sum_{i=1}^{n} (x_i - \mu_x)(y_i - \mu_y)}{N-1} \tag{4.8}$$

$$s(x,y) = \frac{\sigma_{xy} + C_3}{\sigma_x + \sigma_y + C_3} \tag{4.9}$$

$$SSIM(x,y) = [l(x,y)]^\alpha \cdot [c(x,y)]^\beta \cdot [s(x,y)]^\gamma \tag{4.10}$$

$$SSIM(x,y) = \frac{(2\mu_x\mu_y + C_1)(2\sigma_{xy} + C_2)}{(\mu_x^2 + \mu_y^2 + C_1)(\sigma_x^2 + \sigma_y^2 + C_2)} \tag{4.11}$$

$$MSSIM(X,Y) = \frac{\sum_{i=1}^{M} SSIM(x_j, y_j)}{M} \tag{4.12}$$

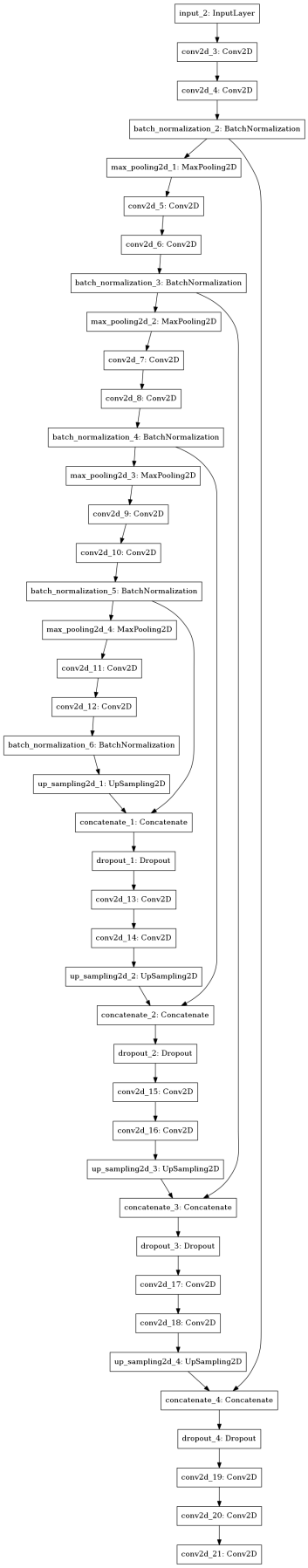


Figure 4.3: Trained Model Architecture.

Layer (type)	Output shape	Param #	Connected to
input_2 (InputLayer)	(32, 128, 128, 1)	0	N/A
conv2d_3 (Conv2D)	(32, 128, 128, 64)	1664	input_2[0][0]
conv2d_4 (Conv2D)	(32, 128, 128, 64)	102464	conv2d_3[0][0]
batch_norm_2 (BatchNorm)	(32, 128, 128, 64)	256	conv2d_4[0][0]
max_pooling2d_1 (MaxPooling2D)	(32, 64, 64, 64)	0	batch_norm_2[0][0]
conv2d_5 (Conv2D)	(32, 64, 64, 128)	204928	max_pooling2d_1[0][0]
conv2d_6 (Conv2D)	(32, 64, 64, 128)	409728	conv2d_5[0][0]
batch_norm_3 (BatchNorm)	(32, 64, 64, 128)	512	conv2d_6[0][0]
max_pooling2d_2 (MaxPooling2D)	(32, 32, 32, 128)	0	batch_norm_3[0][0]
conv2d_7 (Conv2D)	(32, 32, 32, 256)	819456	max_pooling2d_2[0][0]
conv2d_8 (Conv2D)	(32, 32, 32, 256)	1638656	conv2d_7[0][0]
batch_norm_4 (BatchNorm)	(32, 32, 32, 256)	1024	conv2d_8[0][0]
max_pooling2d_3 (MaxPooling2D)	(32, 16, 16, 256)	0	batch_norm_4[0][0]
conv2d_9 (Conv2D)	(32, 16, 16, 512)	3277312	max_pooling2d_3[0][0]
conv2d_10 (Conv2D)	(32, 16, 16, 512)	6554112	conv2d_9[0][0]
batch_norm_5 (BatchNorm)	(32, 16, 16, 512)	1024	conv2d_10[0][0]
max_pooling2d_4 (MaxPooling2D)	(32, 8, 8, 512)	0	batch_norm_5[0][0]
conv2d_11 (Conv2D)	(32, 8, 8, 1024)	13108224	max_pooling2d_4[0][0]
conv2d_12 (Conv2D)	(32, 8, 8, 1024)	26215424	conv2d_11[0][0]
batch_norm_6 (BatchNorm)	(32, 8, 8, 1024)	4096	conv2d_12[0][0]
up_sampling2d_1 (UpSampling2D)	(32, 16, 16, 1024)	0	batch_norm_6[0][0]
concat_1 (Concatenate)	(32, 16, 16, 1536)	0	up_sampling2d_1[0][0] batch_norm_5[0][0]
dropout_1 (Dropout)	(32, 16, 16, 1536)	0	concat_1[0][0]
conv2d_13 (Conv2D)	(32, 16, 16, 512)	19661312	dropout_1[0][0]
conv2d_14 (Conv2D)	(32, 16, 16, 512)	6554112	conv2d_13[0][0]
up_sampling2d_2 (UpSampling2D)	(32, 32, 32, 512)	0	conv2d_14[0][0]
concat_2 (Concatenate)	(32, 32, 32, 768)	0	up_sampling2d_2[0][0] batch_norm_4[0][0]
dropout_2 (Dropout)	(32, 32, 32, 768)	0	concat_2[0][0]
conv2d_15 (Conv2D)	(32, 32, 32, 256)	4915456	dropout_2[0][0]
conv2d_16 (Conv2D)	(32, 32, 32, 256)	1638656	conv2d_15[0][0]
up_sampling2d_3 (UpSampling2D)	(32, 64, 64, 256)	0	conv2d_16[0][0]
concat_3 (Concatenate)	(32, 64, 64, 384)	0	up_sampling2d_3[0][0] batch_norm_3[0][0]
dropout_3 (Dropout)	(32, 64, 64, 384)	0	concat_3[0][0]
conv2d_17 (Conv2D)	(32, 64, 64, 128)	1228928	dropout_3[0][0]
conv2d_18 (Conv2D)	(32, 64, 64, 128)	409728	conv2d_17[0][0]
up_sampling2d_4 (UpSampling2D)	(32, 128, 128, 128)	0	conv2d_18[0][0]
concat_4 (Concatenate)	(32, 128, 128, 192)	0	up_sampling2d_4[0][0] batch_norm_2[0][0]
dropout_4 (Dropout)	(32, 128, 128, 192)	0	concat_4[0][0]
conv2d_19 (Conv2D)	(32, 128, 128, 64)	307264	dropout_4[0][0]
conv2d_20 (Conv2D)	(32, 128, 128, 64)	102464	conv2d_19[0][0]
conv2d_21 (Conv2D)	(32, 128, 128, 1)	65	conv2d_20[0][0]

Table 4.3: Trained model. Total params: $87,157,889$, where $87,153,921$ are trainable and $3,968$ are non-trainable.

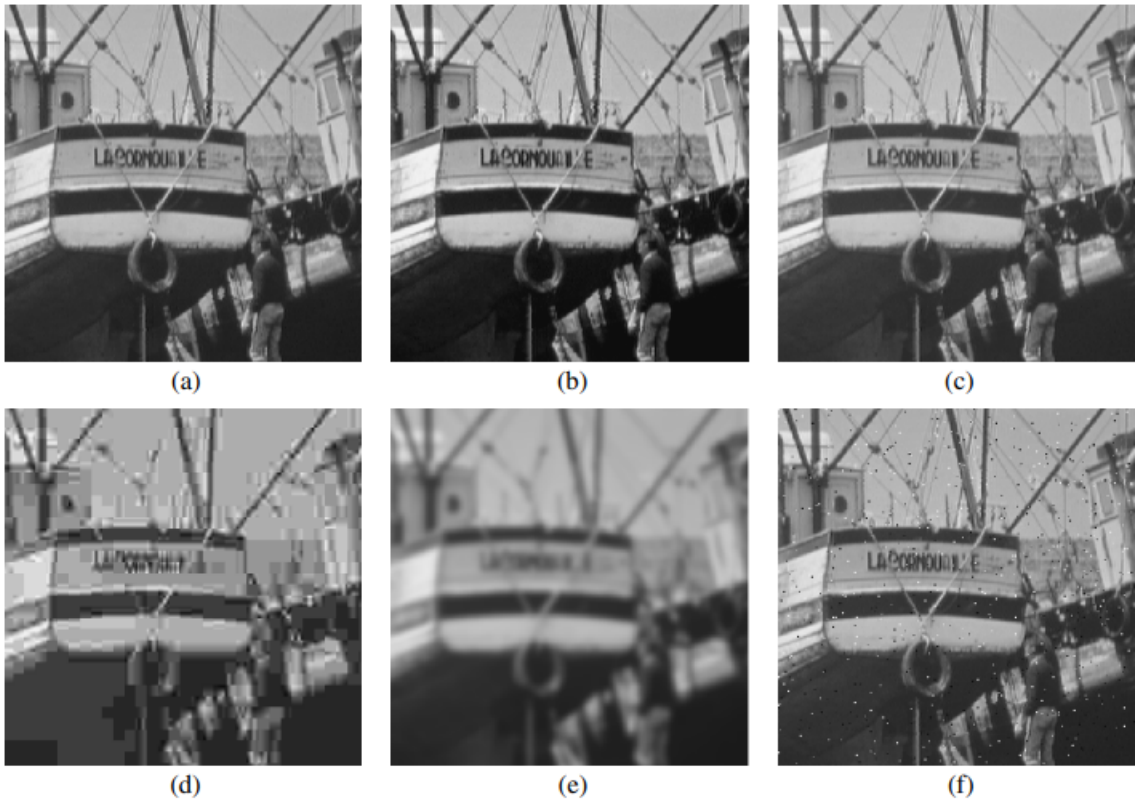


Figure 4.4: Comparison of "Boat" images with different types of distortions, all presenting MSE = 210. (a) Original image, 8 bits/pixel; cropped from 512x512 to 256x256. (b) Contrast-stretched image, MSSIM = 0.9168. (c) Mean-shifted image, MSSIM = 0.9900. (d) JPEG compressed image, MSSIM = 0.6949. (e) Blurred image, MSSIM = 0.7052. (f) Salt-pepper impulsive noise contaminated image, MSSIM = 0.7748 [15].

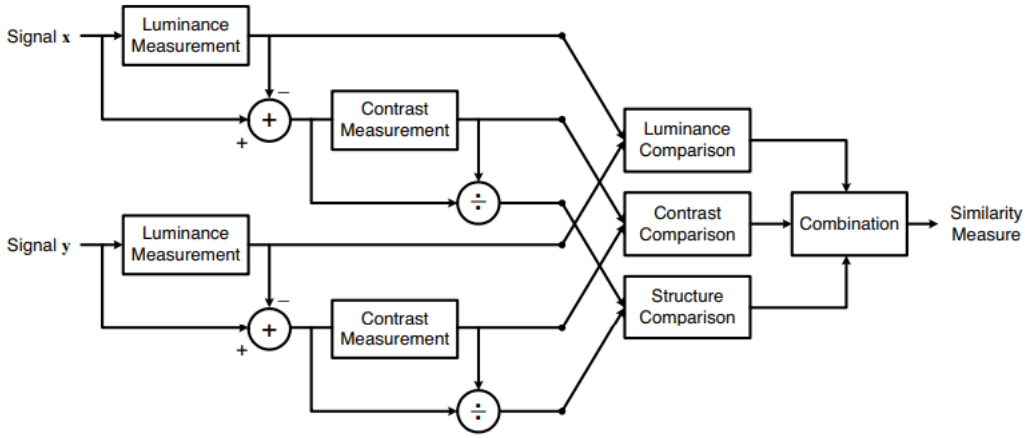


Figure 4.5: SSIM Framework. [15]

Chapter 5

Results and Discussions

> Information is the resolution of uncertainty.
>
> Claude Shannon (1948)

In this chapter we will describe our tests results. Each test tries to achieve a better model, i.e., a model that performs better on the test set, than the previous one, by changing essentially the regularization parameters, number of epochs, batch size, filter size, and initial quantity of filters. At the end of the chapter, we make a general discussion of all tests.

5.1 The Tests

Table 5.1 resumes all the tests. We conducted four different tests, each one with a small variation of one or more *hyperparameters*. The datasets described in this table are the same as those described in Table 4.2.

Test	Dataset	Epochs	BatchSize	FilterSize	Filters#	Dropout	BatchNorm
1	1	1000	32	$3x3$	32	X	X
2	2	100	32	$3x3$	32	X	X
3	2	740	32	$3x3$	32	✓	✓
4	2	200	32	$5x5$	64	✓	✓

Table 5.1: Tests configurations.

As described in Section 2.7, we used the dice coefficient as the loss function. Each test shows dice coefficient loss curves in the training and validation sets.

5.1.1 Test 1: Getting a baseline

We start with the most basic test. In this test, we don't make any data augmentation in the training set and we do not apply any regularization nor Batch normalization. The objective in this test is to determine a baseline for all other tests. Figure 5.1 shows the loss obtained for this test.

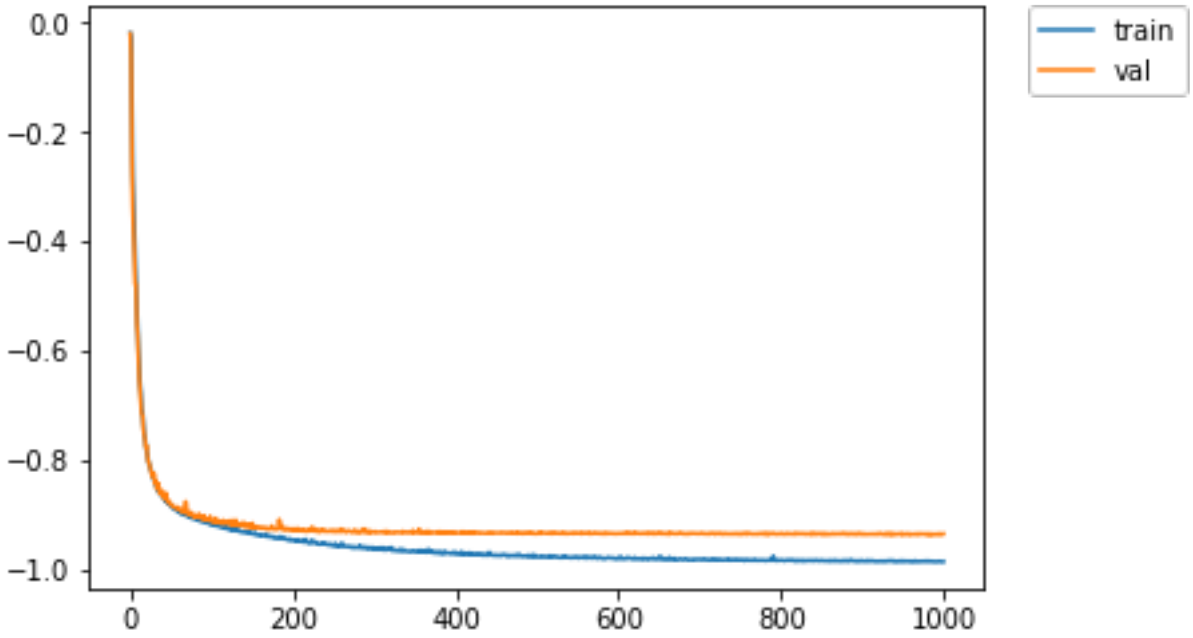


Figure 5.1: Test 1 - Dice loss - 1000 epochs.

5.1.2 Test 2: The Effect of Data Augmentation

With the baseline set, we want to see the effect of the data augmentation. For this we use the same configuration as for test in Section 5.1.1, but we decrease the number of epochs and use data augmentation. Figure 5.2 shows the loss for this test. We did not observe any reasonable improvement over test 5.1.1. It's important to note here that we train only with 100 epochs because we increased the dataset by a factor of 9 compared with test 5.1.1 and it would be computationally expensive training 1000 epochs. Moreover, it's pretty clear from the loss curve that we obtain a good convergence even before the first 100 epochs.

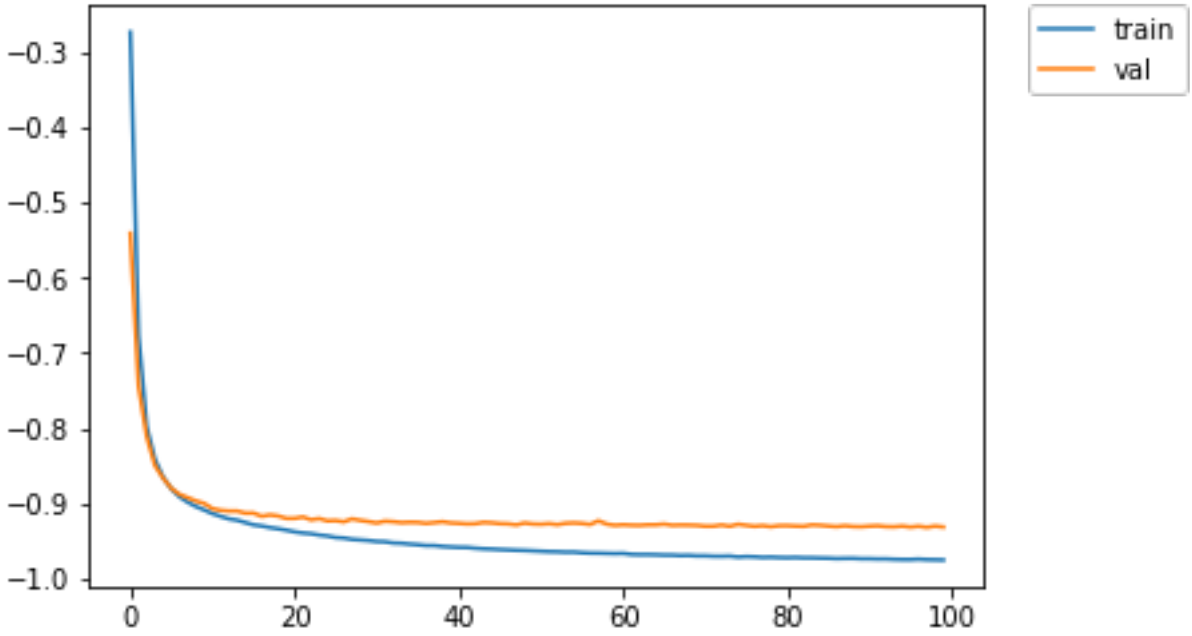


Figure 5.2: Test 2 - Dice loss - Data Augment; 100 Epochs.

5.1.3 Test 3: Adding Dropout and Batch Normalization

Until now, no test used any kind of regularizer, like Dropout, and they were trained without Batch normalization. If we look at the loss curve for all the tests, they do not appear to be overfitting, but by introducing regularizer we obtain a better model, since the validation loss curve keeps closer to the training loss curve. Figure 5.3 shows the loss curve for this test and we can see that the general validation loss is smaller than the previous one. By being closer to the training curve, it means that the model is predicting better on the validation set.

5.1.4 Test 4: More and Bigger Filters per Layer

All the configurations so far started with 32 filters of size $3x3$ each. In this test, we want to see how the model behaves when we increase the *capacity*, i.e., the number of parameters. To achieve this, we increase the filter size to $5x5$, starting with 64 instead of 32. For a similar reason that the one explained in test in Section 5.1.2, in this test, again, we decrease the number of epochs because the model has much more parameters and it is much more computationally expensive to train. Figure 5.4 shows the loss curve for this test. We can see clearly that with a higher capacity the model starts to be a little bit unstable compared to others, although this instability does

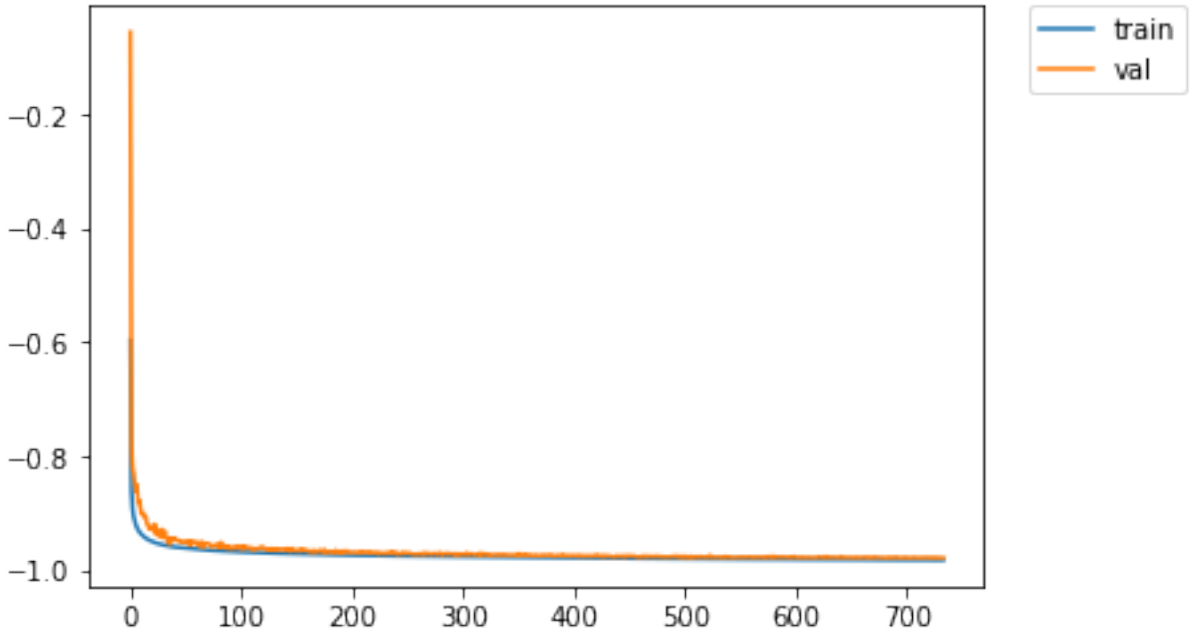


Figure 5.3: Test 3 - Dice loss - With Droupout and Normalization.

not cause an overfitting. Moreover, with more parameters, training and inference start to be really computationally expensive.

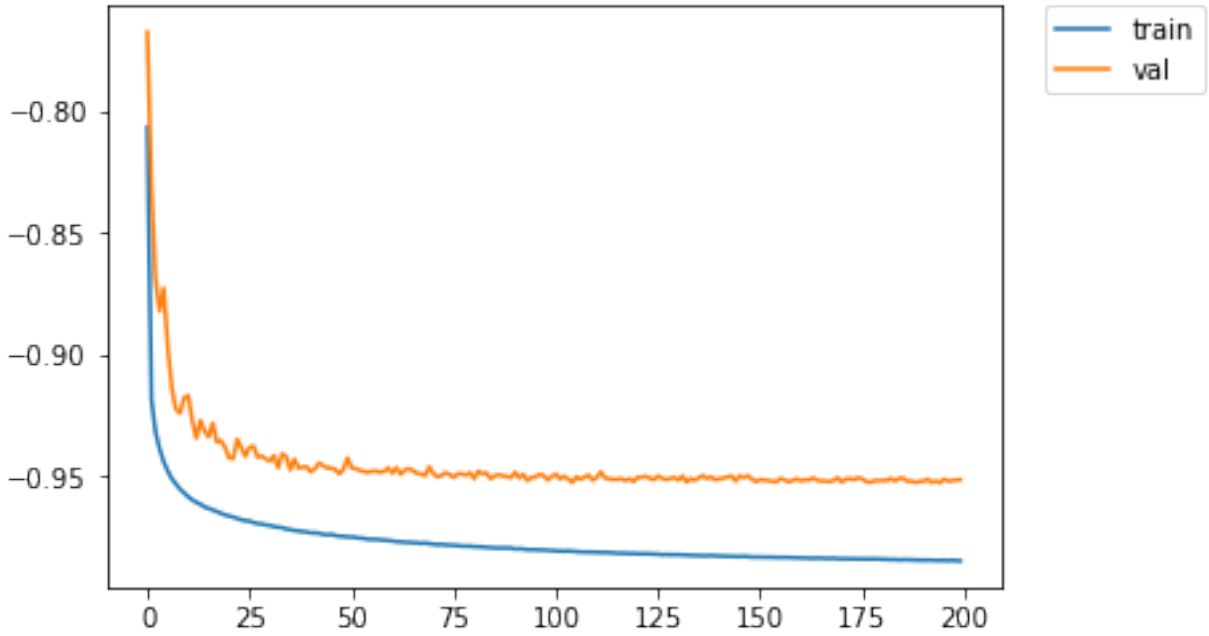


Figure 5.4: Test 4 - Dice loss - Bigger and More Filters.

5.2 Results

For each trained model, we measure the accuracy of the model with the test set. To do this, we use the opposite of the loss function, $-dice\_loss$, to obtain the F1-Score. Table 5.2 shows the F1-Score averaged over all images of the test set and also the training timing and inference timing.

Test	Average F1-Score (%)	TrainingTime($hours$)	InferenceTime(s)
1	93.76	1.0	0.2
2	93.38	8.3	0.2
3	95.10	61.7	0.2
4	95.18	88.8	2

Table 5.2: Average F1-score of segmentation, training time and inference time.

Looking at the F1-score in Table 5.2, the models 5.1.3 and 5.1.4 seems to perform better. Actually, they don't have much difference in the results compared each other. However, we can not look only to the overall averaged F1-Score. We need somehow to make a more significant statistical analysis. To do this, we use a Box Plot to evaluate median, percentiles and outliers. Figure 5.5 shows the results comparing all the models using SSIM index and Figure 5.6 using the Dice coefficient. Figures 5.7 and 5.8 show the model from test 5.1.3 being applied on 16 unseen images from test set along with its respective Ground Truth segmentation and Predicted segmentation results. Figures 5.9 and 5.10 show the model from test 5.1.3 being applied on 14 general images found on Internet. These images are not in OASIS neither in ABIDE dataset.

5.3 Discussions

We obtained an overall score of 95.10% in the test set for the model obtained in test 5.1.3. Compared to models obtained in tests 5.1.1 and 5.1.2, it is far away better than these. Compared to model 5.1.4, they have almost the same performance. We choose model 5.1.3 because it has less outliers compared to model 5.1.4, its training and inference time are computationally cheaper, and also because the real test on images obtained from Internet it performed better.

It is clear when we test with images that are in a different distribution from the

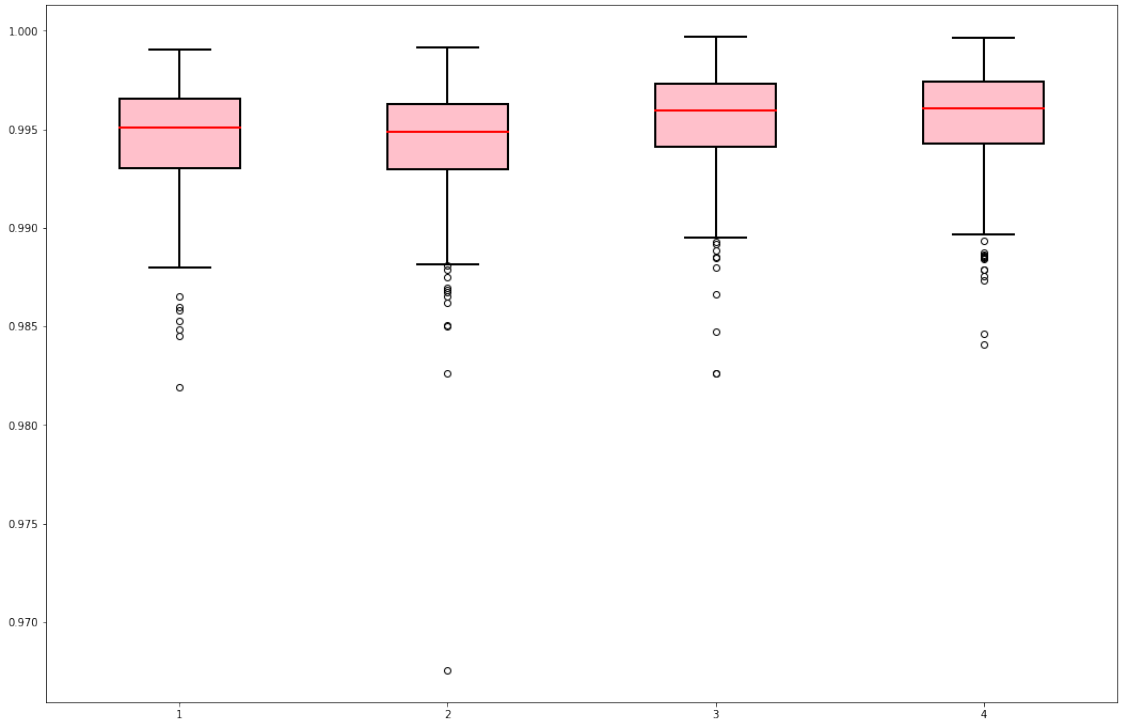


Figure 5.5: Box plot for all trained models using SSIM index.

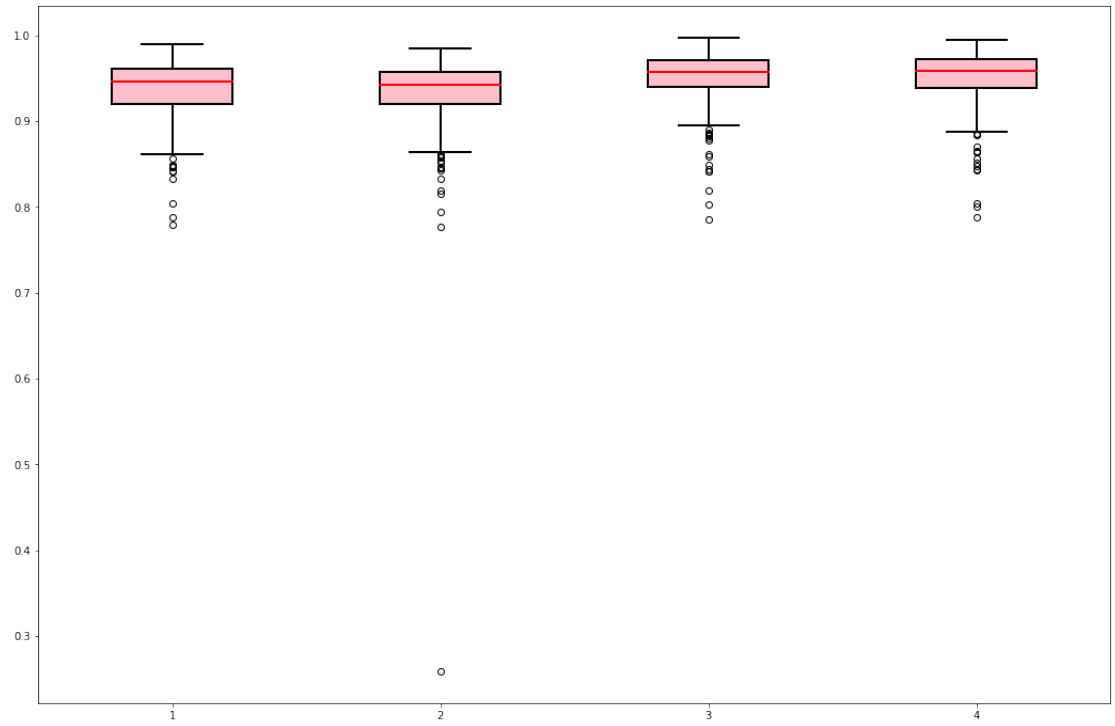


Figure 5.6: Box plot for all trained models using Dice coefficient.

dataset, i.e., images that not belong to dataset OASIS neither ABIDE, we get some reasonable segmentation, even though the model makes more mistakes. To address this drawback, we would have to make our dataset more generic, adding images of

other distributions and augmenting it with other types of image transformations such as shear, zoom and elastic deformations.

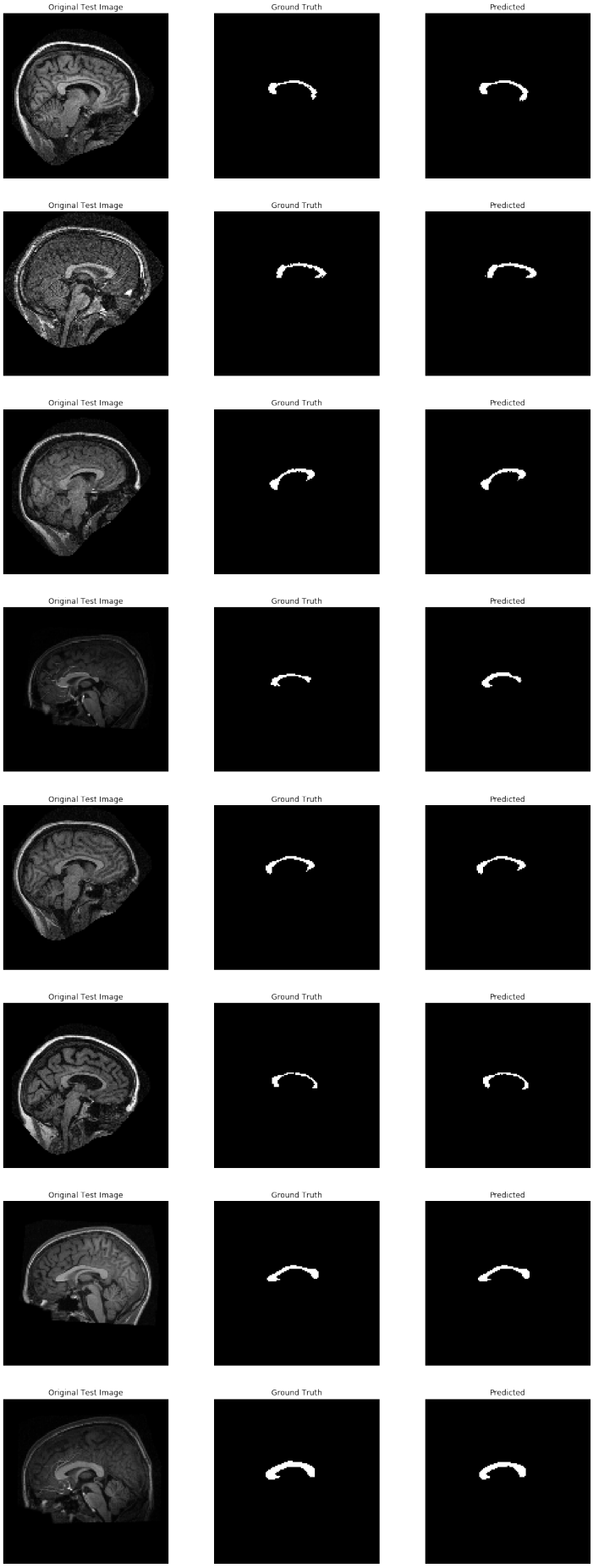


Figure 5.7: Final results for test set. First 8 images.

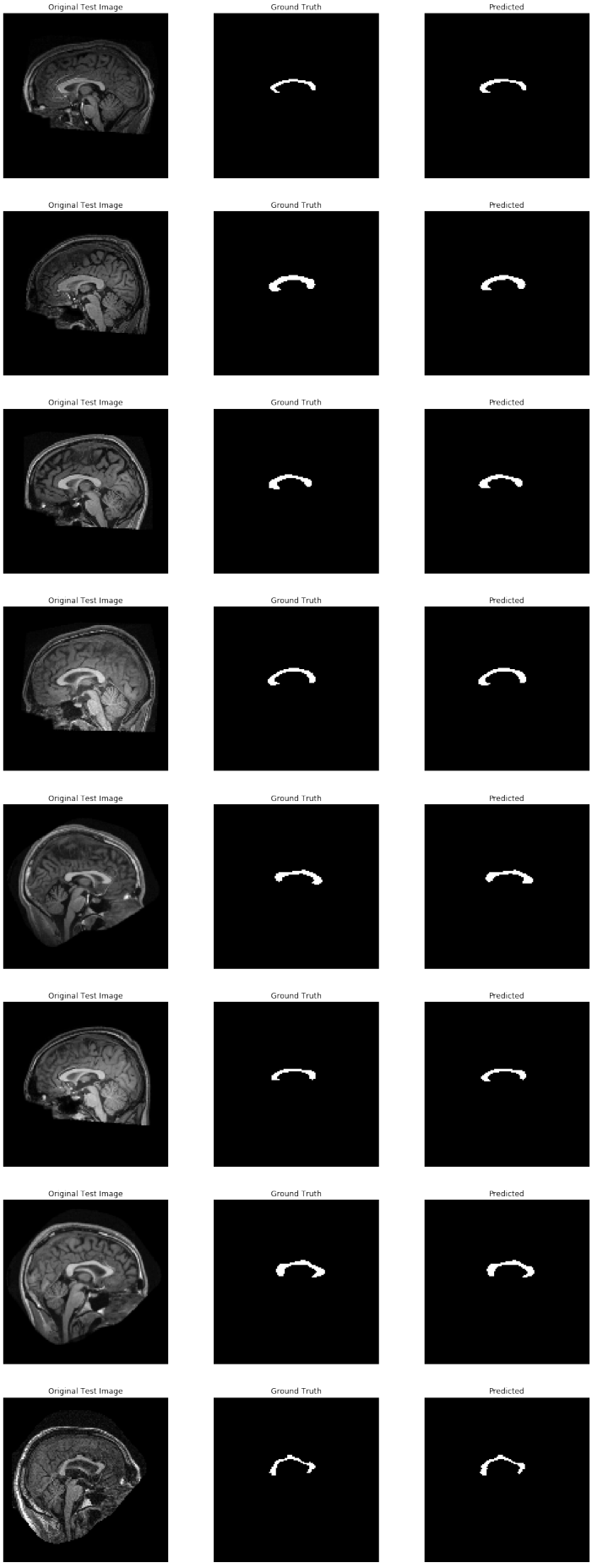


Figure 5.8: Final results for test set. Another 8 images.

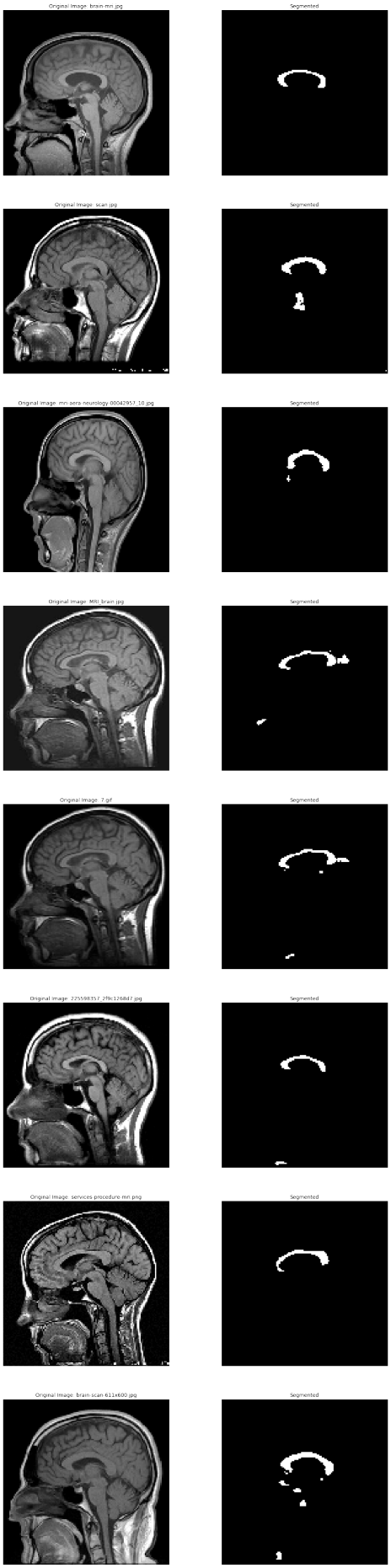


Figure 5.9: Final results for images obtained on the Internet. First 8 images.

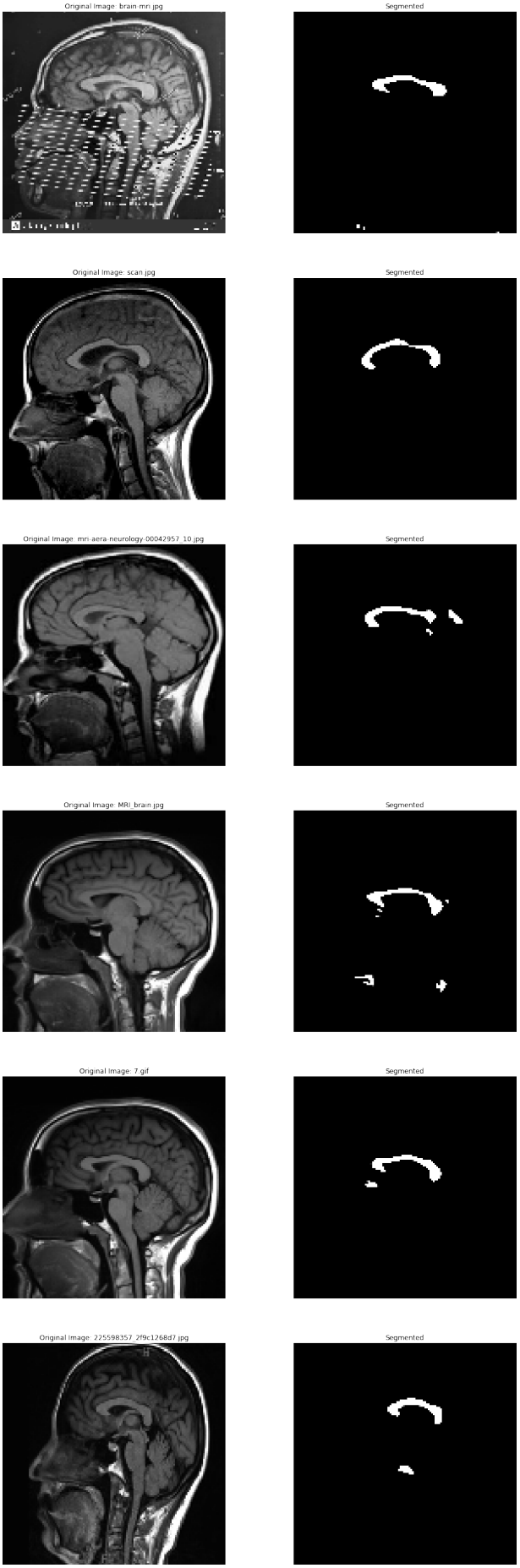


Figure 5.10: Final results for images obtained on the Internet. Another 6 images.

Chapter 6

Conclusion and Future Work

> For millions of years, mankind lived just like the animals. Then something happened which unleashed the power of our imagination. We learned to talk.
>
> Stephen Hawking (1994)

6.1 Conclusion

We successfully built a system that obtained an overall 95.10% F1-Score with only 2003 annotated images in the dataset and without using any pre-processing steps to help the network to segment. Another key point is that the best model took about 61 hours to be trained in a NVIDIA® Tesla K80 GPU. Although this is a long time for to train the model, the inference takes only 0.2 s per image in a regular Intel® Core™ i7-7500U CPU. It makes the system feasible to be used in a real application. Unfortunately, it was not possible to compare the U-Net with the SegNet due to lack of time. This is an issue we intend to address in the near future. Also, there is a segmentation tool for Corpus Callosum developed by [48], which we were not able to properly run in our environment. That could be a good candidate to test the developed system against a traditional segmentation method.

6.2 Future Work

There are a lot of opportunities to enhance this system. Regarding datasets, we can increase the dataset size if we obtain new sources of data with annotated images. These should be new datasets different from OASIS and ABIDE. Also, we can experiment new data augmentation techniques like *shear*, *zoom* and *elastic deformation*. It can help in the robustness of the network for unseen data.

To improve results, we can compare the developed system with SegNet and other segmentation networks. Also, a comparison with traditional techniques should also be good.

To improve training, there are two works [34, 49] which shows that Parametric ReLU and Randomized ReLU consistently outperform the original ReLU. We can add this techniques in our system too and compare the results. To get a speedup in training, a compression in the input images using Principal Component Analysis, PCA, could be applied. It transforms the input image to a new representation with less components. Another idea is that our current architecture is based on U-Net and it copies the whole features maps from the contraction path to their correspondent pair in the expansion path. Looking at the SegNet, it only copies the *max pooling* indexes and it is much faster than copying the whole feature map. These two ideas could be combined: In the first concatenations, we are talking about small feature maps, so we can keep the whole feature map as in the current implementation, but as we go on the expansion path, we could copy just the max pooling indexes, making computation faster.

Figure 6.1: Deusa Minerva.

Bibliography

[1] NIELSEN, M. A. *Neural Networks and Deep Learning*. Determination Press, 2015.

[2] LEE, F., Y. S. J. J. "CS231n Convolutional Neural Networks for Visual Recognition". 2017. Available in: <http://cs231n.github.io/>.

[3] NG, A., K. K. M. Y. B. "Neural Networks and Deep Learning". 2017. Available in: <https://www.coursera.org/learn/neural-networks-deep-learning>.

[4] NG, A., K. K. M. Y. B. "Improving Deep Neural Networks: Hyperparameter tuning, Regularization and Optimization". 2017. Available in: <https://www.coursera.org/learn/deep-neural-network>.

[5] SIMONYAN, K., ZISSERMAN, A. "Very Deep Convolutional Networks for Large-Scale Image Recognition", *CoRR*, v. abs/1409.1556, 2014.

[6] KRIZHEVSKY, A. "CIFAR-10 and CIFAR-100 datasets". 2010. Available in: <https://www.cs.toronto.edu/~kriz/>.

[7] DESHPANDE, A. "A Beginner's Guide To Understanding Convolutional Neural Networks". 2016. Available in: <https://adeshpande3.github.io/adeshpande3.github.io/A-Beginner's-Guide-To-Understanding-Convolutional-Neural-Networks/>.

[8] REN, S., HE, K., GIRSHICK, R. B., etãl. "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks", *CoRR*, v. abs/1506.01497, 2015. Available in: <http://arxiv.org/abs/1506.01497>.

[9] GARCIA-GARCIA, A., ORTS-ESCOLANO, S., OPREA, S., etãl. "A Review on Deep Learning Techniques Applied to Semantic Segmentation", *CoRR*, v. abs/1704.06857, 2017. Available in: <http://arxiv.org/abs/1704.06857>.

[10] RONNEBERGER, O., FISCHER, P., BROX, T. "U-Net: Convolutional Networks for Biomedical Image Segmentation", *CoRR*, v. abs/1505.04597, 2015. Available in: <http://arxiv.org/abs/1505.04597>.

[11] BADRINARAYANAN, V., KENDALL, A., CIPOLLA, R. "SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation", *CoRR*, v. abs/1511.00561, 2015. Available in: <http://arxiv.org/abs/1511.00561>.

[12] RANZATO, M., HUANG, F. J., BOUREAU, Y. L., etãl. "Unsupervised Learning of Invariant Feature Hierarchies with Applications to Object Recognition". In: *2007 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1–8, June 2007. doi: 10.1109/CVPR.2007.383157.

[13] KENNEDY, T. "Neuroradiology". 2018. Available in: <https://sites.google.com/a/wisc.edu/neuroradiology/home>.

[14] INDI. "ABIDE". 2018. Available in: <http://fcon_1000.projects.nitrc.org/indi/abide/>.

[15] WANG, Z., BOVIK, A. C., SHEIKH, H. R., etãl. "Image quality assessment: from error visibility to structural similarity", *IEEE Transactions on Image Processing*, v. 13, n. 4, pp. 600–612, April 2004. ISSN: 1057-7149. doi: 10.1109/TIP.2003.819861.

[16] HINKLEY, L. B. N., MARCO, E. J., FINDLAY, A. M., etãl. "The Role of Corpus Callosum Development in Functional Connectivity and Cognitive Processing", *PLOS ONE*, v. 7, n. 8, pp. 1–17, 08 2012. doi: 10.1371/journal.pone.0039804. Available in: <https://doi.org/10.1371/journal.pone.0039804>.

[17] DOWNHILL, J. E. J., BUCHSBAUM, M. S., WEI, T., etãl. "Shape and size of the corpus callosum in schizophrenia and schizotypal personality disorder", v. 42, n. 3, pp. 123–208, May 2000. ISSN: 0920-9964.

[18] TATCO, V., G. F. E. A. "AC-PC line". 2013. Available in: <https://radiopaedia.org/articles/ac-pc-line-1>.

[19] MARCUS, D. S., WANG, T. H., PARKER, J., etãl. "Open Access Series of Imaging Studies (OASIS): cross-sectional MRI data in young, middle aged, nondemented, and demented older adults", *Journal of cognitive neuroscience*, v. 19, n. 9, pp. 1498–1507, 2007.

[20] KUCHARSKY HIESS, R., ALTER, R., SOJOUDI, S., etãl. "Corpus Callosum Area and Brain Volume in Autism Spectrum Disorder: Quantitative Analysis of Structural MRI from the ABIDE Database", *Journal of Autism and Developmental Disorders*, v. 45, n. 10, pp. 3107–3114, Oct 2015. ISSN: 1573-3432. doi: 10.1007/s10803-015-2468-8. Available in: <https://doi.org/10.1007/s10803-015-2468-8>.

[21] BHALERAO, G. V., SAMPATHILA, N. "K-means clustering approach for segmentation of corpus callosum from brain magnetic resonance images". In: *International Conference on Circuits, Communication, Control and Computing*, pp. 434–437, Nov 2014. doi: 10.1109/CIMCA.2014.7057839.

[22] LEE, C., HUH, S., KETTER, T., etãl. "Automated Segmentation of the Corpus Callosum in Midsagittal Brain Magnetic Resonance Images", *Optical Engineering*, v. 39, n. 4, pp. 924–935, 2000. doi: 10.1117/1.602449. Available in: <http://bigwww.epfl.ch/publications/lee0001.html, http://bigwww.epfl.ch/publications/lee0001.ps>.

[23] FREITAS, P., RITTNER, L., APPENZELLER, S., etãl. "Watershed-Based Segmentation of the Midsagittal Section of the Corpus Callosum in Diffusion MRI". In: *2011 24th SIBGRAPI Conference on Graphics, Patterns and Images*, pp. 274–280, Aug 2011. doi: 10.1109/SIBGRAPI.2011.46.

[24] LI, S., ZHU, L., JIANG, T. "Active shape model segmentation using local edge structures and AdaBoost". In: *Miar*, v. 3150, pp. 121–128. Springer, 2004.

[25] FREUND, Y., SCHAPIRE, R. E. "A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting", *Journal of Computer and System Sciences*, v. 55, n. 1, pp. 119 – 139, 1997. ISSN: 0022-0000. doi: https://doi.org/10.1006/jcss.1997.1504. Available in: <http://www.sciencedirect.com/science/article/pii/S002200009791504X>.

[26] BREJL, M., SONKA, M. "Object localization and border detection criteria design in edge-based image segmentation: automated learning from examples", *IEEE Transactions on Medical Imaging*, v. 19, n. 10, pp. 973–985, Oct 2000. ISSN: 0278-0062. doi: 10.1109/42.887613.

[27] CABEZAS, M., OLIVER, A., LLADÓ, X., etãl. "A Review of Atlas-based Segmentation for Magnetic Resonance Brain Images", *Comput. Methods Prog. Biomed.*, v. 104, n. 3, pp. e158–e177, dez. 2011. ISSN: 0169-2607. doi: 10.1016/j.cmpb.2011.07.015. Available in: <http://dx.doi.org/10.1016/j.cmpb.2011.07.015>.

[28] MEYER, A. "Multi-atlas Based Segmentation of Corpus Callosum on MRIs of Multiple Sclerosis Patients". In: Jiang, X., Hornegger, J., Koch, R. (Eds.), *Pattern Recognition: 36th German Conference, GCPR 2014, Münster, Germany, September 2-5, 2014, Proceedings*, pp. 729–735, Cham, Springer International Publishing, 2014. ISBN: 978-3-319-11752-2. doi: 10.1007/978-3-319-11752-2_61. Available in: <https://doi.org/10.1007/978-3-319-11752-2_61>.

[29] LECUN, Y., BENGIO, Y., HINTON, G. "Deep learning", *Nature*, v. 521, n. 7553, pp. 436–444, 2015.

[30] IOFFE, S., SZEGEDY, C. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift", *CoRR*, v. abs/1502.03167, 2015. Available in: <http://arxiv.org/abs/1502.03167>.

[31] BENGIO, Y., SIMARD, P., FRASCONI, P. "Learning long-term dependencies with gradient descent is difficult", *IEEE Transactions on Neural Networks*, v. 5, n. 2, pp. 157–166, Mar 1994. ISSN: 1045-9227. doi: 10.1109/72.279181.

[32] NAIR, V., HINTON, G. E. "Rectified Linear Units Improve Restricted Boltzmann Machines". In: *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, pp. 807–814, USA, 2010. Omnipress. ISBN: 978-1-60558-907-7. Available in: <http://dl.acm.org/citation.cfm?id=3104322.3104425>.

[33] KRIZHEVSKY, A., SUTSKEVER, I., HINTON, G. E. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, pp. 1097–1105, USA, 2012. Curran Associates Inc. Available in: <http://dl.acm.org/citation.cfm?id=2999134.2999257>.

[34] XU, B., WANG, N., CHEN, T., etãl. "Empirical Evaluation of Rectified Activations in Convolutional Network", *CoRR*, v. abs/1505.00853, 2015. Available in: <http://arxiv.org/abs/1505.00853>.

[35] SRIVASTAVA, N., HINTON, G., KRIZHEVSKY, A., etãl. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting", *Journal of Machine Learning Research*, v. 15, pp. 1929–1958, 2014. Available in: <http://jmlr.org/papers/v15/srivastava14a.html>.

[36] GEMAN, S., BIENENSTOCK, E., DOURSAT, R. "Neural Networks and the Bias/Variance Dilemma", *Neural Comput.*, v. 4, n. 1, pp. 1–58, jan. 1992. ISSN: 0899-7667. doi: 10.1162/neco.1992.4.1.1. Available in: <http://dx.doi.org/10.1162/neco.1992.4.1.1>.

[37] SHELHAMER, E., LONG, J., DARRELL, T. "Fully Convolutional Networks for Semantic Segmentation", *IEEE Trans. Pattern Anal. Mach. Intell.*, v. 39, n. 4, pp. 640–651, abr. 2017. ISSN: 0162-8828. doi: 10.1109/TPAMI.2016.2572683. Available in: <https://doi.org/10.1109/TPAMI.2016.2572683>.

[38] CIREŞAN, D. C., GIUSTI, A., GAMBARDELLA, L. M., etãl. "Deep Neural Networks Segment Neuronal Membranes in Electron Microscopy Images". In: *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'12, pp. 2843–2851, USA, 2012. Curran Associates Inc. Available in: <http://dl.acm.org/citation.cfm?id=2999325.2999452>.

[39] MARCUS, D. S., WANG, T. H., PARKER, J., etãl. "Open Access Series of Imaging Studies (OASIS): Cross-sectional MRI Data in Young, Middle Aged, Nondemented, and Demented Older Adults", *Journal of Cognitive Neuroscience*, v. 19, n. 9, pp. 1498–1507, 2007. doi: 10.1162/jocn.2007.19.9.1498. Available in: <https://doi.org/10.1162/jocn.2007.19.9.1498>.

[40] CHOLLET, F. *Deep Learning with Python*. Manning, 2017.

[41] BROADHURST, D. "If I used data normalization (x-mean(x))/std(x) for training data, would I use train Mean and Standard Deviation to normalize test data?" 2013. Available in: <https://www.researchgate.net/post/If_I_used_data_normalization_x-meanx_stdx_for_training_data_would_I_use_train_Mean_and_Standard_Deviation_to_normalize_test_data>.

[42] SUDRE, C. H., LI, W., VERCAUTEREN, T., etãl. "Generalised Dice overlap as a deep learning loss function for highly unbalanced segmentations", *CoRR*, v. abs/1707.03237, 2017. Available in: <http://arxiv.org/abs/1707.03237>.

[43] DROZDZAL, M., VORONTSOV, E., CHARTRAND, G., etãl. "The Importance of Skip Connections in Biomedical Image Segmentation", *CoRR*, v. abs/1608.04117, 2016. Available in: <http://arxiv.org/abs/1608.04117>.

[44] KINGMA, D. P., BA, J. "Adam: A Method for Stochastic Optimization", *CoRR*, v. abs/1412.6980, 2014. Available in: <http://arxiv.org/abs/1412.6980>.

[45] MORISHITA, M., ODA, Y., NEUBIG, G., etãl. "An Empirical Study of Mini-Batch Creation Strategies for Neural Machine Translation", *CoRR*, v. abs/1706.05765, 2017. Available in: <http://arxiv.org/abs/1706.05765>.

[46] ABADI, M., AGARWAL, A., BARHAM, P., etãl. "TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems". 2015. Available in: <https://www.tensorflow.org/>. Software available from tensorflow.org.

[47] CHOLLET, F., OTHERS. "Keras". https://github.com/keras-team/keras, 2015.

[48] VACHET, C., YVERNAULT, B., BHATT, K., etãl. "Automatic corpus callosum segmentation using a deformable active Fourier contour model". In: *Medical Imaging 2012: Biomedical Applications in Molecular, Structural, and Functional Imaging*, v. 8317, 2012. ISBN: 9780819489661. doi: 10.1117/12.911504.

[49] HE, K., ZHANG, X., REN, S., etãl. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification", *CoRR*, v. abs/1502.01852, 2015. Available in: <http://arxiv.org/abs/1502.01852>.

Appendix A

Preparing the Input Dataset

In this appendix we show the code for preparing the input dataset. It is divided in two parts:

1. **Create numpy arrays from raw images**: Get all images from disk and save them as a binary object in pickle format.
2. **Separate train, val and test data and save them on disk**: Split the dataset in train, validation and test.

generate-dataset-thesis

March 21, 2018

In [ ]:
```
import numpy as np
import os
import joblib
import math
from PIL import Image
```

0.0.1 Create numpy arrays from raw images

In [ ]:
```
datasets = [
    'datasets/abide_imgs',
    'datasets/oasis_imgs'
]

img_size = (128, 128)
Xall, Yall = np.array([]), np.array([])
number_of_images_total = 0

for dataset in datasets:
    print("Reading data for dataset {}".format(dataset))
    total_images_for_dataset = 0
    dataset_folder = os.path.join('.', dataset)
    dataset_files = sorted(os.listdir(dataset_folder))
    dataset_size = len(dataset_files)
    for i in range(0, dataset_size, 2):
        number_of_images_total += 1
        total_images_for_dataset += 1
        full_image = dataset_files[i]
        segmented_image = dataset_files[i+1]
        if 'abide' in dataset_folder:
            full_image, segmented_image = segmented_image, full_image
        filename = os.path.splitext(full_image)[0]

        # create np array image of full image
        tiff_file_path = os.path.join(dataset_folder, full_image)
        tiff_image = Image.open(tiff_file_path, 'r').convert('L').resize(img_size)
        full_image = np.array(tiff_image)
        Xall = np.append(Xall, full_image)
```

```
            # create np array image of segmented image
            tiff_file_path = os.path.join(dataset_folder, segmented_image)
            tiff_image = Image.open(tiff_file_path, 'r').convert('L').resize(img_size)
            segmented_image = np.array(tiff_image)
            segmented_image[segmented_image != 255] = 1.0
            segmented_image[segmented_image == 255] = 0.0
            Yall = np.append(Yall, segmented_image)

            if number_of_images_total % 1000 == 0:
                print("{} / {} processed!".format(
                    total_images_for_dataset, dataset_size // 2))
        print("Dataset {} finished!".format(dataset))

    Xall = Xall.reshape(number_of_images_total, *img_size, 1)
    Yall = Yall.reshape(number_of_images_total, *img_size, 1)

    print("Generated dataset shapes. input: {} ; output: {}".format(
        Xall.shape, Yall.shape))

    joblib.dump((Xall, Yall), 'datasets/dataset-1/all.pkl')
```

0.0.2 Separate train, val and test data and save them on disk

```
In [ ]: Xall, Yall = joblib.load('datasets/dataset-1/all.pkl')
        print(Xall.shape)
        print(Yall.shape)

        training_percentage = 0.7
        validation_percentage = 0.1

        training_set_index = math.floor(Xall.shape[0]*training_percentage)
        validation_set_index = math.floor(
            Xall.shape[0]*validation_percentage) + training_set_index

        # shuffling before training-validation-test slicing
        ids = np.arange(Xall.shape[0])
        np.random.shuffle(ids) # shuffle images to avoid bias in training
        Xall, Yall = Xall[ids], Yall[ids]

        print(Xall.shape)
        print(Yall.shape)

        Xte, yte = Xall[validation_set_index:,:],
                   Yall[validation_set_index:] # X and y for testing
        # test set is saved on disk.
        #It should NOT be modified. All model evaluations MUST target the same test set.
        joblib.dump((Xte,
```

```
            yte,
            {
               'test_percentage': 1 - training_percentage - validation_percentage
            }
            ), 'datasets/dataset-1/test.pkl')

# X and y for training and validation
X_remaining, y_remaining = Xall[:validation_set_index,:],
                           Yall[:validation_set_index]
# test and val set are saved on disk.
#It can be loaded after and be shuffled, cross validated, etc.
config = {
            'train_percentage': training_percentage,
            'training_set_index': training_set_index,
            'val_percentage': validation_percentage,
            'validation_set_index': validation_set_index
         }
joblib.dump((X_remaining, y_remaining, config), 'datasets/dataset-1/train-and-val.pkl')
```

Appendix B

Train and Results

In this appendix we show the training code and results. It loads the datasets that we generated in appendix A, configure the U-Net model, train it with the datasets and generate a trained model. Finally, use the generated model to evaluate test images.

train-thesis

March 21, 2018

```
In [ ]: import joblib
        import numpy as np
        import os
        import tensorflow as tf
        import math
        import csv

        from keras.models import Model
        from keras.layers import Input, merge, Conv2D,
                                 MaxPooling2D, UpSampling2D, concatenate,
                                 Dropout
        from keras.layers.normalization import BatchNormalization
        from keras.optimizers import Adam
        from keras.callbacks import ModelCheckpoint, LearningRateScheduler,
                                    TensorBoard, CSVLogger
        from keras.preprocessing.image import ImageDataGenerator
        from keras.utils import plot_model
        from keras import backend as K
        from PIL import Image
        from skimage.measure import compare_ssim as ssim
        from sklearn.preprocessing import StandardScaler
        from sklearn.decomposition import PCA
        import matplotlib.pyplot as plt # to plot images
        %matplotlib inline
```

0.0.1 Consts

```
In [ ]: train_and_val_dataset_file = 'datasets/dataset-1/train-and-val.pkl'
        test_dataset_file = 'datasets/dataset-1/test.pkl'
        saved_model_filename = "datasets/dataset-1/test-3.hdf5"
        csv_logger_training = "datasets/dataset-1/test-3.csv"
```

0.0.2 Load datasets

```
In [ ]: X_remaining, Y_remaining, remaining_dataset_desc = joblib.load(
                                                            train_and_val_dataset_file
                                                            )

        # X and y for test
```

```
Xte, yte, test_dataset_desc = joblib.load(test_dataset_file)
training_set_index = remaining_dataset_desc['training_set_index']
validation_set_index = remaining_dataset_desc['validation_set_index']

# X and y for training
Xtr, ytr = X_remaining[:training_set_index,:],
           Y_remaining[:training_set_index]
# X and y for validation
Xva, yva = X_remaining[training_set_index:validation_set_index,:],
           Y_remaining[training_set_index:validation_set_index]

print(Xtr.shape)
print(Xva.shape)
print(Xte.shape)
print(ytr.shape)
print(yva.shape)
print(yte.shape)
```

0.0.3 Data augmentation

```
In [ ]: txtyrange = range(-1, 1, 1) # translation range for x and y directions
        loat = [ (tx, ty) for tx in txtyrange
                          for ty in txtyrange
               ] # list of accepted translations
        loaa = list(range(-1, 1, 1))
        foia = len(loat) * len(loaa) # factor of image augmentation
        print(foia)

        total_imgs = Xtr.shape[0]
        increment = 0

        print(total_imgs*foia)

        for i in range(total_imgs):
            x = Xtr[i]
            y = ytr[i]
            for (tx, ty) in loat:
                input_array = x.reshape(x.shape[0], x.shape[1])
                output_array = y.reshape(y.shape[0], y.shape[1])

                input_image = Image.fromarray(input_array)
                input_image = input_image.transform(
                                  input_image.size, Image.AFFINE, (1, 0, tx, 0, 1, ty)
                              ) # translated full image

                output_image = Image.fromarray(output_array)
                output_image = output_image.transform(
                                  output_image.size, Image.AFFINE, (1, 0, tx, 0, 1, ty)
```

```
                    ) # translated full image

            for a in loaa:
                increment += 1

                if increment % 1000 == 0:
                    print("Processed {}/{}".format(increment, total_imgs*foia))

                # rotated trcimg
                input_image = input_image.rotate(a, resample=Image.BICUBIC)
                input_array_augmented = np.array(input_image) # array with pixel values
                Xtr = np.append(Xtr, input_array_augmented).reshape(
                        total_imgs+increment, x.shape[0], x.shape[1], x.shape[2]
                        )
                 # rotated trcimg
                output_image = output_image.rotate(a, resample=Image.BICUBIC)
                output_array_augmented = np.array(output_image) # array with pixel values
                ytr = np.append(ytr, output_array_augmented).reshape(
                                total_imgs+increment, y.shape[0], y.shape[1], y.shape[2]
                                )
```

0.0.4 Pre processing

In [ ]:
```
# Preprocessing in the training set (mean and sd) and apply it to all sets

full_image_mean_value = Xtr.mean() # mean-value for each pixel of all full images
full_image_sd = Xtr.std() # standard deviation for each pixel of all full images

Xtr = (Xtr - full_image_mean_value) / full_image_sd
Xva = (Xva - full_image_mean_value) / full_image_sd
Xte = (Xte - full_image_mean_value) / full_image_sd
```

0.0.5 Pre-configurations

In [ ]:
```
K.set_image_data_format('channels_last')  # TF dimension
_, *input_image_shape, _ = Xtr.shape
input_image_shape = tuple(input_image_shape)
print(input_image_shape)

smooth = 1.

use_dropout = True
use_regularizers = True
dropout_rate = 0.5
number_of_epochs = 1000
batch_size = 32
kernel_size = (5, 5)
initial_volume_size = 64
```

0.0.6 Define Unet model

```
In [ ]: # Define loss function
        def dice_coef_per_image_in_batch(y_true, y_pred):
            y_true_f = K.batch_flatten(y_true)
            y_pred_f = K.batch_flatten(y_pred)
            intersection = 2. * K.sum(y_true_f * y_pred_f, axis=1, keepdims=True) + smooth
            union = K.sum(y_true_f, axis=1, keepdims=True) + K.sum(
                                                                y_pred_f,
                                                                axis=1,
                                                                keepdims=True) + smooth
            return K.mean(intersection / union)

        def dice_coef_loss(y_true, y_pred):
            return -dice_coef_per_image_in_batch(y_true, y_pred)

        def dice_coef_accur(y_true, y_pred):
            return dice_coef_per_image_in_batch(y_true, y_pred)


        def setup_regularizers(conv_layer):
            return BatchNormalization()(conv_layer) if use_regularizers else conv_layer

        def setup_dropout(conv_layer):
            return Dropout(dropout_rate)(conv_layer) if use_dropout else conv_layer

        # Define model
        inputs = Input((*input_image_shape, 1))
        conv1 = Conv2D(initial_volume_size,
                       kernel_size, activation='relu',
                       padding='same')(inputs)
        conv1 = Conv2D(initial_volume_size,
                       kernel_size, activation='relu',
                       padding='same')(conv1)
        conv1 = setup_regularizers(conv1)
        pool1 = MaxPooling2D(pool_size=(2, 2))(conv1)

        conv2 = Conv2D(initial_volume_size*2,
                       kernel_size, activation='relu',
                       padding='same')(pool1)
        conv2 = Conv2D(initial_volume_size*2,
                       kernel_size,
                       activation='relu',
                       padding='same')(conv2)
        conv2 = setup_regularizers(conv2)
        pool2 = MaxPooling2D(pool_size=(2, 2))(conv2)

        conv3 = Conv2D(initial_volume_size*4,
```

```
                kernel_size,
                activation='relu',
                padding='same')(pool2)
conv3 = Conv2D(initial_volume_size*4,
                kernel_size,
                activation='relu',
                padding='same')(conv3)
conv3 = setup_regularizers(conv3)
pool3 = MaxPooling2D(pool_size=(2, 2))(conv3)

conv4 = Conv2D(initial_volume_size*8,
                kernel_size,
                activation='relu',
                padding='same')(pool3)
conv4 = Conv2D(initial_volume_size*8,
                kernel_size, activation='relu',
                padding='same')(conv4)
conv4 = setup_regularizers(conv4)
pool4 = MaxPooling2D(pool_size=(2, 2))(conv4)

conv5 = Conv2D(initial_volume_size*16,
                kernel_size,
                activation='relu',
                padding='same')(pool4)
conv5 = Conv2D(initial_volume_size*16,
                kernel_size,
                activation='relu',
                padding='same')(conv5)
conv5 = setup_regularizers(conv5)

up6 = concatenate([UpSampling2D(size=(2, 2))(conv5), conv4], axis=3)
up6 = setup_dropout(up6)
conv6 = Conv2D(initial_volume_size*8,
                kernel_size,
                activation='relu',
                padding='same')(up6)
conv6 = Conv2D(initial_volume_size*8,
                kernel_size,
                activation='relu',
                padding='same')(conv6)

up7 = concatenate([UpSampling2D(size=(2, 2))(conv6), conv3], axis=3)
up7 = setup_dropout(up7)
conv7 = Conv2D(initial_volume_size*4,
                kernel_size,
                activation='relu',
                padding='same')(up7)
conv7 = Conv2D(initial_volume_size*4,
```

```
                    kernel_size,
                    activation='relu',
                    padding='same')(conv7)

    up8 = concatenate([UpSampling2D(size=(2, 2))(conv7), conv2], axis=3)
    up8 = setup_dropout(up8)
    conv8 = Conv2D(initial_volume_size*2,
                    kernel_size,
                    activation='relu',
                    padding='same')(up8)
    conv8 = Conv2D(initial_volume_size*2,
                    kernel_size,
                    activation='relu',
                    padding='same')(conv8)

    up9 = concatenate([UpSampling2D(size=(2, 2))(conv8), conv1], axis=3)
    up9 = setup_dropout(up9)
    conv9 = Conv2D(initial_volume_size,
                    kernel_size,
                    activation='relu',
                    padding='same')(up9)
    conv9 = Conv2D(initial_volume_size,
                    kernel_size,
                    activation='relu',
                    padding='same')(conv9)

    conv10 = Conv2D(1, (1, 1), activation='sigmoid')(conv9)

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

    model.compile(optimizer=Adam(lr=1e-5),
                  loss=dice_coef_loss,
                  metrics=[dice_coef_accur])
    print("Size of the CNN: %s" % model.count_params())
```

```
In [ ]: print(model.summary())
```

0.0.7 Train model

```
In [ ]: # Define callbacks
    model_checkpoint = ModelCheckpoint(
                                       saved_model_filename,
                                       monitor='val_dice_coef_accur',
                                       save_best_only=True, verbose=1
                                      )
    csv_logger = CSVLogger(csv_logger_training, append=True, separator=';')

    # Train
```

```
        history = model.fit(Xtr,
                            ytr,
                            batch_size=batch_size,
                            epochs=number_of_epochs,
                            verbose=2,
                            shuffle=True,
                      callbacks=[model_checkpoint, csv_logger], validation_data=(Xva, yva))
```

0.0.8 Show model metrics

```
In [ ]: x = history.history['dice_coef_accur']
        y = history.history['val_dice_coef_accur']
        plt.plot(x, label='train')
        plt.plot(y, label = 'val')
        plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
        plt.show()

        x = history.history['loss']
        y = history.history['val_loss']
        plt.plot(x, label='train')
        plt.plot(y, label='val')
        plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
        plt.show()
```

0.0.9 Evaluate the model

```
In [ ]: test_loss, accuracy_test = model.evaluate(Xte, yte, verbose=0)
        print("Training Accuracy Mean: "+str(np.array(data['dice_coef_accur']).mean()))
        print("Validation Accuracy Mean: "+str(np.array(data['val_dice_coef_accur']).mean()))
        print("Test Accuracy Mean: "+str(accuracy_test))
```

0.0.10 Predict masks using the trained model

```
In [ ]: model.load_weights("datasets/dataset-1/test-6-new-tentative-105-0.9527.hdf5")
        print(model.metrics_names)
        test_loss, accuracy_test = model.evaluate(Xte, yte, verbose=0)
        print("Test Accuracy Mean: "+str(accuracy_test))
        imgs_mask_test = model.predict(Xte, verbose=1)
```

0.0.11 Show results

```
In [ ]: ncols = 3 # number of columns in final grid of images
        nrows = 30 # looking at all images takes some time
        _, axes = plt.subplots(nrows, ncols, figsize=(17, 17*nrows/ncols))
        for axis in axes.flatten():
            axis.set_axis_off()
            axis.set_aspect('equal')
```

```
for k in range(0, nrows):
    im_test_original = Xte[k].reshape(*input_image_shape)
    im_result = imgs_mask_test[k].reshape(*input_image_shape)
    im_ground_truth = yte[k].reshape(*input_image_shape)

    axes[k, 0].set_title("Original Test Image")
    axes[k, 0].imshow(im_test_original, cmap='gray')

    axes[k, 1].set_title("Ground Truth")
    axes[k, 1].imshow(im_ground_truth, cmap='gray')

    axes[k, 2].set_title("Predicted")
    axes[k, 2].imshow(im_result, cmap='gray')
```

0.0.12 Compare images quality

In [ ]:
```
def mse(ground_truth, predicted):
    _, width, height, _ = ground_truth.shape
    return np.sum(( (predicted - ground_truth) ** 2), axis=(1,2,3)) / (width * height)

result = mse(imgs_mask_test, yte)

objects = tuple([x for x in range(yte.shape[0])])
y_pos = np.arange(len(objects))

plt.figure(figsize=(10, 10))
plt.plot(y_pos, result, 'ro')
#plt.bar(y_pos, result)
plt.ylabel('Error')
plt.xlabel('Image')
plt.title('MSE Error Per Image')

plt.show()

number_of_images, width, height, _ = yte.shape
objects = []
result = []
for i in range(number_of_images):
    objects.append(i)
    ground_truth = yte[i].astype('float32').reshape(yte.shape[1:3])
    predicted = imgs_mask_test[i].reshape(imgs_mask_test.shape[1:3])
    result.append(ssim(ground_truth, predicted))

objects = tuple(objects)
y_pos = np.arange(len(objects))

plt.figure(figsize=(10, 10))
plt.plot(y_pos, result, 'ro')
```

```
#plt.bar(y_pos, result)
plt.ylabel('SSIM')
plt.xlabel('Image')
plt.title('SSIM Per Image')

plt.show()
```

0.0.13 Compare image quality results of all tests

```
In [ ]: def _setup_regularizers(conv_layer, use_regularizers):
            return BatchNormalization()(conv_layer) if use_regularizers else conv_layer

        def _setup_dropout(conv_layer, use_dropout):
            return Dropout(dropout_rate)(conv_layer) if use_dropout else conv_layer

        def get_model(kernel_size, use_regularizer, use_dropout, initial_volume_size):
            # Define model
            inputs = Input((*input_image_shape, 1))
            conv1 = Conv2D(initial_volume_size,
                           kernel_size,
                           activation='relu',
                           padding='same')(inputs)
            conv1 = Conv2D(initial_volume_size,
                           kernel_size,
                           activation='relu',
                           padding='same')(conv1)
            conv1 = _setup_regularizers(conv1, use_regularizer)
            pool1 = MaxPooling2D(pool_size=(2, 2))(conv1)

            conv2 = Conv2D(initial_volume_size*2,
                           kernel_size,
                           activation='relu',
                           padding='same')(pool1)
            conv2 = Conv2D(initial_volume_size*2,
                           kernel_size,
                           activation='relu',
                           padding='same')(conv2)
            conv2 = _setup_regularizers(conv2, use_regularizer)
            pool2 = MaxPooling2D(pool_size=(2, 2))(conv2)

            conv3 = Conv2D(initial_volume_size*4,
                           kernel_size,
                           activation='relu',
                           padding='same')(pool2)
            conv3 = Conv2D(initial_volume_size*4,
                           kernel_size,
                           activation='relu',
                           padding='same')(conv3)
```

```
conv3 = _setup_regularizers(conv3, use_regularizer)
pool3 = MaxPooling2D(pool_size=(2, 2))(conv3)

conv4 = Conv2D(initial_volume_size*8,
               kernel_size,
               activation='relu',
               padding='same')(pool3)
conv4 = Conv2D(initial_volume_size*8,
               kernel_size,
               activation='relu',
               padding='same')(conv4)
conv4 = _setup_regularizers(conv4, use_regularizer)
pool4 = MaxPooling2D(pool_size=(2, 2))(conv4)

conv5 = Conv2D(initial_volume_size*16,
               kernel_size,
               activation='relu',
               padding='same')(pool4)
conv5 = Conv2D(initial_volume_size*16,
               kernel_size,
               activation='relu',
               padding='same')(conv5)
conv5 = _setup_regularizers(conv5, use_regularizer)

up6 = concatenate([UpSampling2D(size=(2, 2))(conv5), conv4], axis=3)
up6 = _setup_dropout(up6, use_dropout)
conv6 = Conv2D(initial_volume_size*8,
               kernel_size,
               activation='relu',
               padding='same')(up6)
conv6 = Conv2D(initial_volume_size*8,
               kernel_size,
               activation='relu',
               padding='same')(conv6)

up7 = concatenate([UpSampling2D(size=(2, 2))(conv6), conv3], axis=3)
up7 = _setup_dropout(up7, use_dropout)
conv7 = Conv2D(initial_volume_size*4,
               kernel_size,
               activation='relu',
               padding='same')(up7)
conv7 = Conv2D(initial_volume_size*4,
               kernel_size,
               activation='relu',
               padding='same')(conv7)

up8 = concatenate([UpSampling2D(size=(2, 2))(conv7), conv2], axis=3)
up8 = _setup_dropout(up8, use_dropout)
```

```
            conv8 = Conv2D(initial_volume_size*2,
                           kernel_size,
                           activation='relu',
                           padding='same')(up8)
            conv8 = Conv2D(initial_volume_size*2,
                           kernel_size,
                           activation='relu',
                           padding='same')(conv8)

            up9 = concatenate([UpSampling2D(size=(2, 2))(conv8), conv1], axis=3)
            up9 = _setup_dropout(up9, use_dropout)
            conv9 = Conv2D(initial_volume_size,
                           kernel_size,
                           activation='relu',
                           padding='same')(up9)
            conv9 = Conv2D(initial_volume_size,
                           kernel_size,
                           activation='relu',
                           padding='same')(conv9)

            conv10 = Conv2D(1, (1, 1), activation='sigmoid')(conv9)

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

            model.compile(optimizer=Adam(lr=1e-5),
                          loss=dice_coef_loss,
                          metrics=[dice_coef_accur])
            return model


In [ ]: trained_models = [
            (
                "datasets/dataset-1/test-2-996--0.9361-0.9361.hdf5",
                get_model((3,3), False, False, 32)),
            (
                "datasets/dataset-1/test-3-97-0.9321.hdf5",
                get_model((3,3), False, False, 32)),
            (
                "datasets/dataset-1/test-5-new-tentative-682-0.9783.hdf5",
                get_model((3,3), True, True, 32)),
            (
                "datasets/dataset-1/test-6-new-tentative-105-0.9527.hdf5",
                get_model((5,5), True, True, 64))
        ]

        def dice_coef_numpy(y_true, y_pred):
            y_true_f = y_true.flatten()
            y_pred_f = y_pred.flatten()
```

```
    intersection = 2 * np.sum(y_true_f * y_pred_f) + smooth
    union = np.sum(y_true_f) + np.sum(y_pred_f) + smooth
    return np.mean(intersection / union)

def _get_predicitions_for_model(trained_model):
    weights, model = trained_model
    model.load_weights(weights)
    imgs_mask_test = model.predict(Xte, verbose=1)
    return imgs_mask_test

def get_box_data_for_each_model(comparison_function):
    data = []
    for model_prediction in models_predictions:
        result = []
        for i in range(number_of_images):
            ground_truth = yte[i].astype('float32').reshape(yte.shape[1:3])
            predicted = model_prediction[i].reshape(imgs_mask_test.shape[1:3])
            measure = comparison_function(ground_truth, predicted)
            result.append(measure)
        data.append(result)
    return data

def _create_box_plot(data, fig_identifier, subplot_index):
    # Create a figure instance
    fig = plt.figure(fig_identifier, figsize=(18, 12))

    # Create an axes instance
    ax = fig.add_subplot(subplot_index)

    # Create the boxplot
    bp = ax.boxplot(data, patch_artist=True)

    for box in bp['boxes']:
        # change outline color
        box.set( color='#000000', linewidth=2)
        # change fill color
        box.set( facecolor = 'pink' )

    ## change color and linewidth of the whiskers
    for whisker in bp['whiskers']:
        whisker.set(color='#000000', linewidth=2)

    ## change color and linewidth of the caps
    for cap in bp['caps']:
        cap.set(color='#000000', linewidth=2)

    ## change color and linewidth of the medians
    for median in bp['medians']:
```

```
            median.set(color='#ff0000', linewidth=2)

        ## change the style of fliers and their fill
        for flier in bp['fliers']:
            flier.set(marker='o', color='green')


    models_predictions = []
    for trained_model in trained_models:
        models_predictions.append(_get_predicitions_for_model(trained_model))
```

In [ ]:
```
data = get_box_data_for_each_model(dice_coef_numpy)
_create_box_plot(data, 'dice', 111)

plt.show()
```

In [ ]:
```
data = get_box_data_for_each_model(ssim)
_create_box_plot(data, 'ssim', 111)

plt.show()
```

0.0.14 Save predicted images to disk

In [ ]:
```
total_images, width, height, _ = imgs_mask_test.shape

for i in range(total_images):
    I = imgs_mask_test[i].reshape(width, height)
    I8 = (((I - I.min()) / (I.max() - I.min())) * 255.9).astype(np.uint8)
    img = Image.fromarray(I8)
    img.save(result_imgs_folder.format(i, 'predicted'))

    I = yte[i].reshape(width, height)
    I8 = (((I - I.min()) / (I.max() - I.min())) * 255.9).astype(np.uint8)
    img = Image.fromarray(I8)
    img.save(result_imgs_folder.format(i, 'gt'))
```

0.0.15 Segmenting random images from Internet

In [ ]:
```
path = "datasets/dataset-1/tests-random-images"
images_in_disk = os.listdir(path)
img_size = (128, 128)
images = np.array([])
number_of_images_total = 0

for f in images_in_disk[8:]:
    input_image_location = os.path.join(path,f)
    if os.path.isfile(input_image_location):
        input_image = Image
                        .open(input_image_location, 'r')
```

```
                                .convert('L')
                                .resize(img_size)
                input_image_array = np.array(input_image)

                images = np.append(images, input_image_array)
                number_of_images_total += 1

        images = images.reshape(number_of_images_total, *img_size, 1)
        images = (images - full_image_mean_value) / full_image_sd

        segmented_images = model.predict(images, verbose=1)

        ncols = 2 # number of columns in final grid of images
        nrows = number_of_images_total # looking at all images takes some time
        _, axes = plt.subplots(nrows, ncols, figsize=(17, 17*nrows/ncols))
        for axis in axes.flatten():
            axis.set_axis_off()
            axis.set_aspect('equal')

        for k in range(0, nrows):
            im_original = images[k].reshape(*img_size)
            im_result = segmented_images[k].reshape(*img_size)

            axes[k, 0].set_title("Original Image: {}".format(images_in_disk[k]))
            axes[k, 0].imshow(im_original, cmap='gray')

            axes[k, 1].set_title("Segmented")
            axes[k, 1].imshow(im_result, cmap='gray')
```

0.0.16 Fun Experiment: Looking at the layers!

```
In [ ]: def _get_convolutions(layer_name):
            output = [layer.output for layer in model.layers if
                      layer.name == layer_name or layer_name is None][0]
            inputs = [K.learning_phase()] + model.inputs
            _convout1_f = K.function(inputs, [output])
            def convout1_f(X):
                # The [0] is to disable the training phase flag
                return _convout1_f([0] + [X])

            convolutions = convout1_f(Xte[0:1])
            convolutions = np.squeeze(convolutions)
            return convolutions


        def layer_to_visualize(layer_name, save_plot=False):
            convolutions = _get_convolutions(layer_name)
            #print ('Shape of conv:', convolutions.shape)
```

```
            n = convolutions.shape[0]

            from math import sqrt
            ncols = nrows = int(sqrt(convolutions.shape[2])) # board
            fig, axes = plt.subplots(nrows, ncols, figsize=(80, 80*nrows/ncols))
            for axis in axes.flatten():
                axis.set_axis_off()
                axis.set_aspect('equal')

            for k in range(0, nrows):
                for j in range(0, ncols):
                    axes[k, j].imshow(convolutions[:,:,k*nrows+j], cmap='gray')
            if save_plot:
                fig.savefig(layer_name)

        layers_name = [layer.name for layer in model.layers]
        print(layers_name)
```

In [ ]:
```
layer_to_visualize(layer_name='conv2d_15', save_plot=False)
```