POLITECNICO DI TORINO

Corso di Laurea Magistrale in Ingegneria Matematica

Tesi di Laurea Magistrale

Representation learning and applications in retina imaging

POLITECNICO
DI TORINO

Thesis Advisors:
Demetrio Labate
Fabio Nicola

Candidate:
Mariachiara Mecati

Academic Year 2018-2019

Ai miei genitori
e ad Alessio,
che mi hanno sostenuta in ogni istante di questo percorso.

Al mio Professore Fabio Nicola,
sempre presente, che mi ha dato l'opportunità di vivere
questa indimenticabile esperienza di ricerca e di vita.

Contents

Introduction

My Master Thesis concerns methods for the segmentation of retina fundus images based on innovative techniques from deep learning.

Systematic diseases such as diabetic retinopathy, glaucoma and aged-related macular degeneration, are known to cause quantifiable changes in the morphology of the retinal microvasculature. This microvasculature is the only part of the human circulation that can be visualized non-invasively in vivo so that it can be readily photographed and processed with the tools of digital image analysis. As the treatment of serious pathologies such as diabetic retinopathy can be significantly improved with early detection, retinal image analysis has been the subject of extensive studies. To carry out this task successfully, one needs to quantify the morphological characteristics of the vascularization of the retina. However, manual extraction of this information is time-consuming, labor-intensive and requires trained personnel. For this reason, several methods have been proposed for the automated segmentation of the retinal microvasculature. Thanks to the advances in image processing and pattern recognition during the last decade, a remarkable progress is being made towards developing automated diagnostic systems for diabetic retinopathy and related conditions. Despite this progress, several challenges remain.

With recent remarkable advances in the field of neural networks and deep learning, several improved methods for segmentation have been introduced in biomedical imaging. Unlike classical model-based methods, neural networks require a training stage, hence there is the need of training data, specifically

images annotated by domain experts. Nonetheless, Convolutional Neural Networks (CNN) like U-net can be trained with a relatively small number of training examples. One main advantage of neural networks is that they offer the possibility to extract features from raw images avoiding the need of building hand-designed features. Their ability to discover spatial local correlations in the data at different scales and abstraction levels, allows them to learn a set of filters that are useful to correctly segment the data and, at the same time, to learn a representation of their morphological characteristics.
Since ocular fundus imaging is widely used to monitor the health status of the human eye and other pathologies (e.g., diabetic retinopathy) and several annotated databases are available, my investigation was focused on the development of a CNN for the segmentation of retina fundus images. For this goal, I adapted a U-net architecture consisting of two sections of convolutional filters; the first section is an encoder that is designed to find an efficient representation of the image in terms of high-dimensional feature vectors; the second section is a decoder that map the feature vectors of the encoder into an appropriate segmentation mask. By adapting existing results from the literature, I have also investigated the inclusion of an additional layer aimed at extracting an internal representation of the network encoding the morphological characteristics of the image.

The training and tuning of this CNN are the core of my Master Thesis, motivated by the goal to build an automated algorithm for the segmentation of retinal images with the ability to learn the morphology of the retinal microvasculature and to output feature vectors encoding this critical information.

Document structure

This Thesis is divided into four main sections.
The *first chapter* presents the "Background" and the motivations upon which I built my Thesis, as well as the subject of this research.
The *second chapter* gives an overview of deep learning and its crucial role in

the retina imaging area.
The *third chapter* deals an accurate explanation of all the critical concepts on which the code has been written: Neural Network and Convolutional Neural Network.
In the *fourth chapter* I will examine each script composing the code.
The *conclusions* summarize the main results, but also present a future possible research work.
Lastly, in the *appendix* the implemented scripts are available.

Chapter 1

Background

1.1 Motivation

A crucial element for the sense of sight is the retina, a layered tissue coating the interior of the eye responsible for the formation of images. By converting light into a neural signal that will be later processed in the brain visual cortex, the retinal tissue is classified as highly metabolically active because of its double blood supply which allows a direct non-invasive observation of the circulation system [3]. In particular, this microvasculature is the only part of the human circulation that can be visualized non-invasively in vivo, so that it can be readily photographed and processed with the tools of digital image analysis.

For this reason, knowing that systematic diseases such as diabetic retinopathy, glaucoma and aged-related macular degeneration cause quantifiable changes in the geometry of the retinal microvasculature and that the treatment of these serious pathologies can be significantly improved with early detection, retinal image analysis has been the subject of extensive studies. Multiple methods have been implemented with focus on the segmentation and delineation of the blood vessels: each approach attempts to recognize the vessel structure, fovea, macula and the optic disc, and to organize the fundus image according to a set of features [3].

Particularly, convolution with matched filters is an important method in this field, as these filters are approximations to the local profiles that vessels are expected to have and their convolution with the data outputs higher values at the locations where the similarity is higher. Even then, it is common that vessels show different types of cross section profiles, making the a priori design of filters a complex task. Therefore, despite the progress, significant challenges remain.

1.2 Anatomic structure of the retina

Retinal layer and structure

The retina is a sensitive tissue inside the eye, located in the inner surface of the posterior two-thirds at three-quarters of the eye on human individuals. So, the retina is a delicate, thin and transparent sheet of tissue derived from the neuroectoderm, in which a series of electrical and chemical events occur, triggered by the optical elements of the eye as they focus an image [5].
The retina comprises the sensory neurons and it is the beginning of the visual path way. Multiple neurons compose the neural retina (neuroretina): it is divided into nine layers and is a key component in the production and the transmission of electrical impulses. Then, the electrical signals generated from this chain of events are sent to the brain via nerve fibers, where they are interpreted as visual images.

The centre of the retina, known as macula, is responsible for the central vision (fine vision) and its center is designated as fovea (as we can observe in figure 1.1): this part allows the eye the greatest resolving power [5]. The living tissue of the retina can be captured using photography, laser polarimetry, fluorescein angiography or optical coherence tomography.

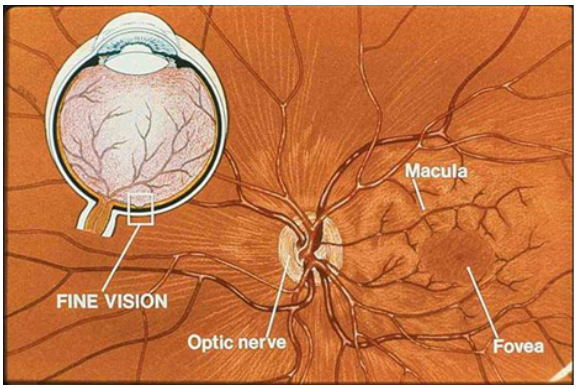

Figure 1.1: Retina and structures involved in fine vision [5].

Retinal vessels and blood supply

To understand functions and anatomy of the retina, a key concept is to study its vessels and the blood supply. As stated before, the analysis of these structures has a crucial importance to diagnose several diseases and malfunctions. Apart from the foveolar avascular zone, said FAZ, and the extreme retinal periphery (that can be supplied throughout the choroidal circulation by diffusion since they are extremely thin), the remaining part of the human retina is too thick to be supplied by either the retinal circultion alone or the choroidal circulation only [6]. Thus, the choroidal circulation supplies the outer retina, while the inner retina is supplied by the retinal circulation. The ophthalmic artery (the first branch of the carotid artery on each side) is the main supplier of blood to the retina, since both the choroidal and the retinal circulation has origin there.

Then there is the choroid, a vascular layer of the eye which contains connective tissues and lies between the retina and the sclera; this part receives till 80% of all the ocular blood. The remainder goes through the iris and the ciliary body for the 15%, while the last 5% goes to the retina.

Now, the choroidal circulation is fed by the ophthalmic artery by the lateral and medial posterior ciliary arteries, which give rise to one long and several short posterior ciliary arteries. All the blood in the capillaries of the choroid (the choroiocapillaris) is supplied by the short posterior arteries, which enter

the posterior globe close to the optic nerve [7], as we can see in figure 1.2.
Particulalrly conserning the retinal circulation, after entering the orbit the central retinal artery branches off the ophthalmic artery and enters the optic nerve behind the globe. Subsequently, the central artery emerges from within the optic nerve cup to give rise to the retinal and inferior circulation with its four main branches, the inferior and superior nasal and temporal retinal arteries. This circulation supplies blood to all the layers of the neuroretina with the exception of the photoreceptors layer (which is an avascular layer dependent on the choriocapillaris).
As last consideration, it is noticeable that the retina circulation has a recursive layout, as it is characterized by the temporal retinal vessel which curves around the fovea and the FAZ [8].

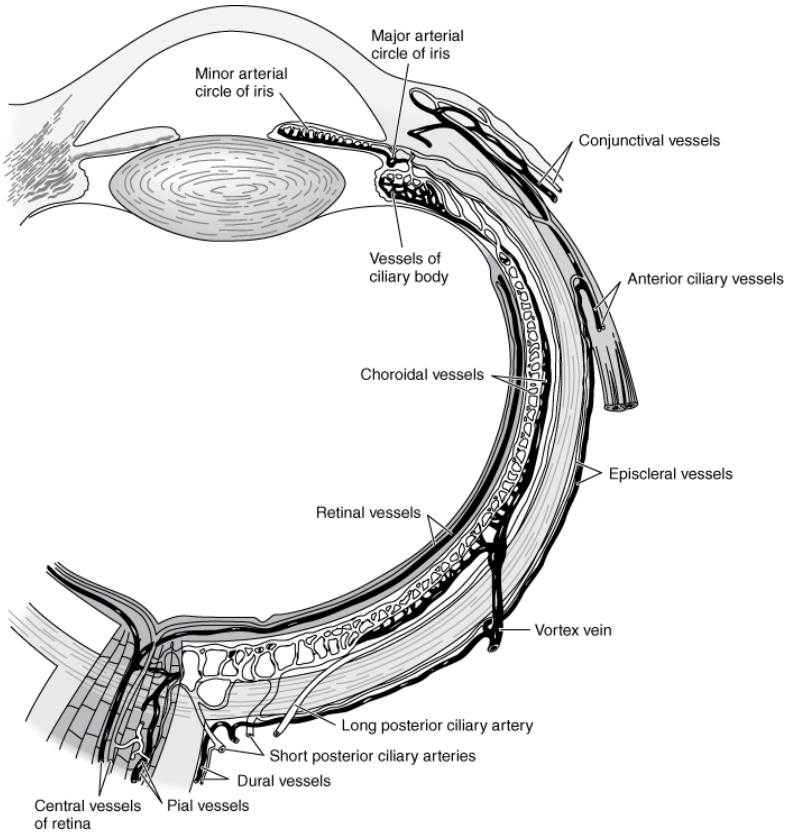

Figure 1.2: Choroidal circulation [7].

1.3 *Fundus* Image

The ocular *fundus* imaging plays a key role in monitoring the health status of the human eye. In general, the assessment and classification of ocular alterations can be easily made thanks to a large number of imaging modalities. Particularly, the color *fundus* photography is one the most important and older techniques used to obtain these images. It requires a fundus camera, which is a complex optical system comprising a specialized low power microscope with an attached camera, capable of simultaneously illuminating and imaging the retina. Overall, this system is designed to image the interior surface of the eye, which includes the retina, posterior pole, optic disc and macula [4]. Initially born as a film-based imaging, fundus photography has been a crucial tool for the early developments in diagnoses and pathology studies. Later, with the advent of digital imaging, the use of digital fundus imaging allowed to reach higher resolution, easier manipulation, processing, tracing of irregularities and faster transmission of information.

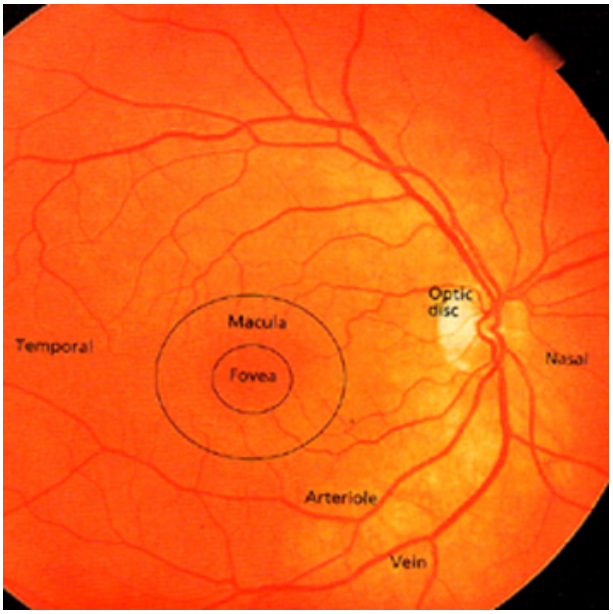

Figure 1.3: Retinal circulation illustrated through Fundus Fluorescein Angiography [8].

1.4 DRIVE Dataset

My study is built upon the *Fundus* image database DRIVE, which consists of images obtained from a diabetic retinopathy screening program in The Netherlands, in which the screening population were composed of 400 diabetic subjects between 25 and 90 years of age. Forty photographs have been randomly selected: 33 belong to health individuals and 7 show signs of mild early diabetic retinopathy [3]. Each image was acquired using a Canon CR5 non-mydriatic 3CCD camera with a 45-degree field of view (FOV) using 8 bits per colour plane at 768 by 584 pixels, and has been cropped around the FOV, which is circular with a diameter of approximately 540 pixels, so that the final dimension of each image is 584×565 pixels.
For each image it is provided a mask image delineating the FOV, so that the database is divided into two sets of images: a *training set* and a *testing set*, each one composed of 20 images.
In addition, for each training image is available a manual segmentation of the vasculature (figure 1.4), while for the test dataset two segmentations are provided, where the first one is used as gold standard and the other one can be used to compare the segmentation performed by an independent human observer with the computer generated segmentation.

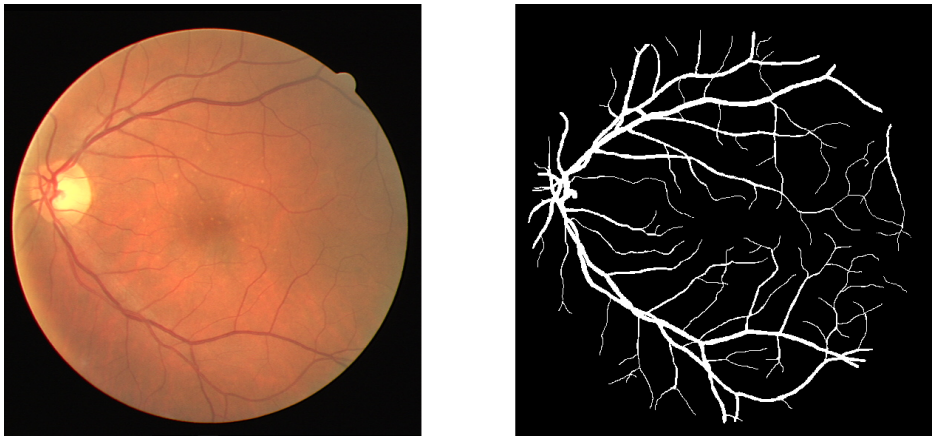

Figure 1.4: An original DRIVE image from the training set on the left and its manual segmentation on the right.

Chapter 2

Retina imaging and Deep learning

2.1 Context

As already mentioned in the previous chapter, the vascular network of the human retina is a crucial diagnostic factor in ophthalmology. An automatic analysis and detection of the retinal vasculature can be widely useful for the implementation of screening programs for the diagnostic of several diseases, computer assisted surgery and biometric identification [9]. The retinal vasculature is composed of arteries and veins which appear as elongated features in the retinal image. Neverthless, the segmentation of the vascular network in fundus imaging is a nontrivial task owing to the variable size of the vessels, relative low contrast and potential presence of pathologies such as haemorrhages, microaneurysms, bright and dark lesions and cotton wool spots. The vessels are locally linear and their intensity varies gradually along the length, but the shape, size and local grey level of blood vessels can widely vary, and some background features may show similar attributes. In addition, vessel crossing and branching can further complicate this task [3].
Notice that, in simple terms, while the purpose of *Image Classification* is to assign a label to an image from a fixed set of categories, the *Segmentation* task aims to identify a precise pattern in the image.

In general, the algorithms for the segmentation of blood vessels in medical

images are divided into two groups. One consists of the rule-based methods, i.e., *unsupervised methods*: vessel tracking, matched filter responses, multi-scale approaches and morphology-based techniques. The other group consists of *supervised methods* (which require manually labelled images for training) and includes pattern identification, classification and recent developments in deep learning. Particularly, compared to the unsupervised learning techniques, the implementation of supervised methodologies have shown an important improvement on specificity and accuracy. Moreover, even though the application of these techniques are still conditioned to the existence of a ground truth, when labelled data are available, deep learning has been further improving the performance of computers in many daily tasks.

2.2 Performance indicator

To analyse and evaluate outcomes and performances of the segmentation, it is important to clearly understand which variables are involved to define the performance, the way they influence the results and the information that can be deducted [3].
The result in the retinal vessel segmentation process is a $pixel-based\ classification$ outcome, so that each pixel can be classified either as vessel or surrounding tissue. As a consequence, each pixel belongs to one of the following categories: *true positive* (TP) when the pixel is identified as a vessel in both the segmentation output and ground truth; *true negative* (TN) when the pixel is identified as non-vessel in both the segmentation output and ground truth; $false\ positive$ (FP) if the pixel is identified as vessel in the segmentation output but as non-vessel in the ground truth; $false\ negative$ (FN) if the pixel is identified as nonvessel in the segmentation output but as vessel in the ground truth. Thanks to these indicators, it is easy to calculate some metrics of performance. Particularly, the *true positive rate* (TPR) is the fraction of vessel pixels which are correctly identified as belonging to vessels; similarly, the $false\ positive\ rate$ (FPR) is the ratio of non-vessel pixels

which were erroneously classified as belonging to vessels. The ratio of the total number of correctly classified pixels (that is the sum of true positive and true negative) to the total number of pixels in the image, is defined as *Accuracy*.

	Vessel present	**Vessel Absent**
Vessel detected	TP	FP
Vessel not detected	FN	TN

Figure 2.1: Vessel classification [3].

2.3 Deep learning overview

As already mentioned, *vessel segmentation* is a key step for different medical applications, as it is widely used to monitor disease progression and evaluate various ophthalmologic diseases. However, manual vessel segmentation executed by trained specialists is a repetitive and time-consuming task; so that during the last decade, thanks to the advances in image processing and pattern recognition, a remarkable progress is being made towards developing automated diagnostic systems for diabetic retinopathy and related conditions. With the more recent advances in the field of neural networks and deep learning, multiple methods have been implemented with focus on the segmentation and delineation of the blood vessels [10].
Especially, deep learning methods such as *Convolutional Neural Networks* (CNN) have recently become a new trend in the Computer Vision area, as they allow to extract features from raw images, avoiding the use of hand-designed features [11]. In fact, their ability to find strong spatially local correlations in the data at different abstraction levels, allows them to learn a set of filters that are useful to correctly segment the data given a labeled training set. Then, a deep-learning network trained on labeled data can be subsequently applied to unstructured data: higher performance can be

achieved giving the network access to much more input than traditional machine-learning approaches, since the more data a net can train on, the more accurate it is likely to be.
Therefore, the process of *feature extraction* is the crucial concept of a convolutional neural network: with local receptive fields, neurons in a CNN can detect elementary visual features such as oriented edges, end-points and corners, that are then combined by successive layers in order to capture higher-order features [12].
In conclusion, the different types of networks as well as the different types of available data, allows the implementation of a new and improved solution, which can be compared with the previous works submitted in this area.

Chapter 3

Neural Networks and Convolutional Neural Networks

Convolutional Neural Networks for semantic segmentation are being used in a large range of research fields, especially in the field of classification and segmentation of medical images. In this particular research area, the use of Convolutional Neural Networks for a pixel-wise classification has shown a solid set of interesting results in the detection and classification of a variety of anatomic structures such as tumors, multiple vessels and brain lesions.

We can summarize the key steps of an *Image Classification* task as follows:

- *Input*. It consists of a set of images, each labeled with one of the different classes; we define this data as *training set*.
- *Learning*. We can refer to this step as *training a classifier*, or *learning a model*: here the *training set* is used to learn what every one of the classes looks like.
- *Evaluation*. In this final step, we evaluate the quality of the classifier by testing it on a new set of images that it has never seen before, called *testing set*; therefore, we will then compare the labels predicted by the classifier to the true labels of these images, named *ground truth*.

A powerful approach to Image Classification that we will naturally extend to entire Neural Networks and Convolutional Neural Networks, is based on two major concepts: a *score function* that maps the raw data to class scores, as well as a *loss function* that quantifies the agreement between the predicted scores and the ground truth labels. This will eventually give rise to an *optimization problem* in which the loss function has to be minimized with respect to the parameters of the score function [14].

3.1 Basic concepts

3.1.1 Score function

To define the score function that maps the pixel values of an image to confidence scores for each class, we assume a training dataset of images $x_i \in \mathbb{R}^D$, $i = 1, ..., N$, each associated with a label $y_i \in \{1, ..., K\}$. So, given N examples (each with a dimensionality D) and K distinct categories, the score function is defined as $f : \mathbb{R}^D \to \mathbb{R}^K$, which maps the raw image pixels to class scores [14].

The simplest possible case is given by the *Linear classifier*:

$$f(x_i, W, b) = Wx_i + b$$

where we assume that the image x_i has all of its pixels flattened out to a single column vector $[D \times 1]$. The parameters of the function are the matrix W, of size $[K \times D]$, and the vector b, of size $[K \times 1]$: the parameters in W are called *weights*, while b is the *bias* vector (it influences the output scores without interacting with the actual data x_i). Therefore, we can notice that:

- the single matrix multiplication Wx_i evaluates separate classifiers in parallel, one for each class, where each classifier is a row of W;
- the input data (x_i, y_i) are given and fixed, whereas we have control over the setting of the parameters W, b. Our purpose is to set these in such way that the computed scores match the ground truth labels across

the whole training set. Intuitively, we wish that the correct class has a score that is higher than the scores of incorrect classes;

- this method takes an advantage from the fact that the training data is used to learn the parameters W, b, but once the learning is complete, the entire training set can be discarded, so that we only keep the learned parameters. In fact, a new test image can be simply forwarded through the function and classified with respect to the computed scores;
- lastly, classifying the test image involves a single matrix multiplication and an addition, which are two simple and efficient operations in computational terms.

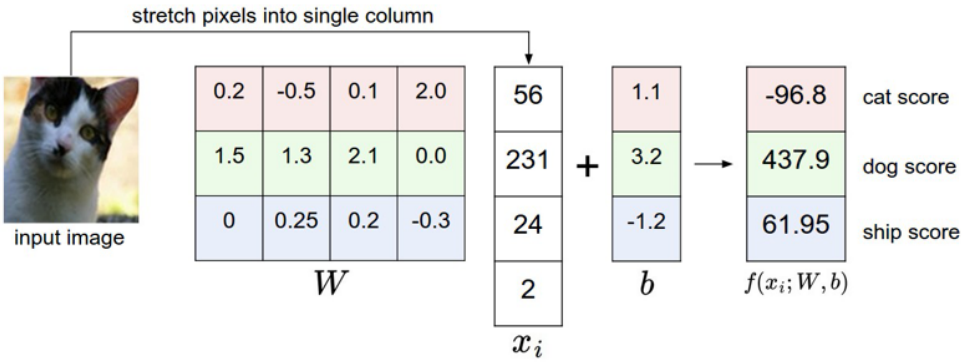

Figure 3.1: A simple example of mapping an image to class scores [14]: suppose the image to have only 4 pixels and assume to have 3 classes (cat, dog, ship). We flatten out the image pixels to a column and perform matrix multiplication to get the scores for each class. As we can deduct from the final prediction, this set of weights W is not good at all, since the weights assign the cat image a very low cat score, erroneously.

Notice now that keeping track of two sets of parameters (biases b and weights W) separately is a little cumbersome. As we can see in figure 3.2, common trick is to combine the two sets of parameters into a single matrix that holds both of them: it is sufficient to extend the vector x_i with one additional dimension that always holds the constant 1 (a default bias dimension).

With this extra dimension, the new score function will simplify to a single matrix multiply, i.e.:

$$f(x_i, W) = Wx_i.$$

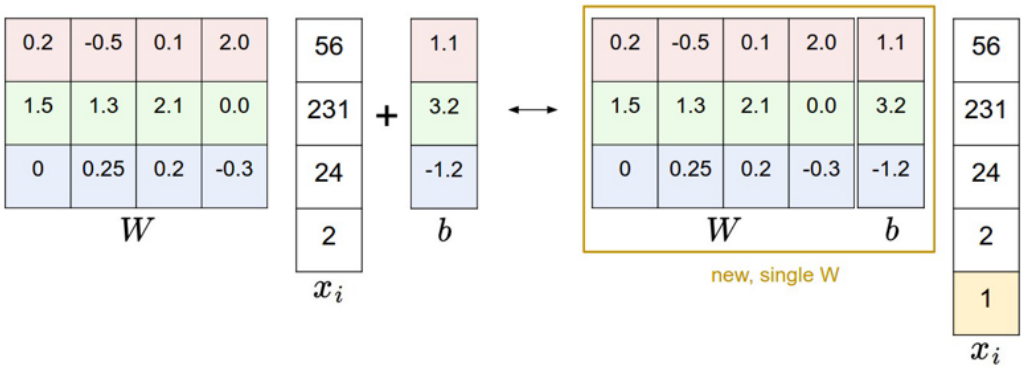

Figure 3.2: According to the *bias trick* [14], doing a matrix multiplication and then adding a bias vector (on the left) is equivalent to adding a dimension with a constant of 1 to all input vectors and extending the weight matrix by the "bias column" bias (on the right). In this way, we only have to learn the matrix of weights.

3.1.2 Loss function

As already seen in the previous section, we do not have control over the data x_i, y_i (which are given and fixed), but we have control over the weights. Thus, we want to set them so that the predicted class scores are consistent with the ground truth labels in the training data.

One of te most commonly classifier is the $Softmax\ classifier$, which gives normalized class probabilities as output and we can think of it as a generalization to multiple classes of the binary Logistic Regression classifier [14]. In the Softmax classifier, the function $f(x_i, W) = Wx_i$ stays unchanged, but if we now interpret these scores as the unnormalized log probabilities for

each class, then we can define the so called $cross-entropy\ loss\ function$:

$$L_i = -\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}\right) \text{ or equivalently } L_i = -f_i + \log\sum_j e^{f_j}$$

where f_j indicates the j-th element of the vector of class scores f.
So, the $Softmax\ function$ is defined as

$$f_j(z) = \frac{e^{z_j}}{\sum_k e^{z_k}},$$

which takes a vector of arbitrary real-valued scores (in $\mathbb{Z}$) and maps it to a vector of values between zero and one which sum to one.

3.1.3 Optimization

As the loss function quantifies the quality of any particular set of weights W, the goal of optimization is to find W such that *minimizes* the loss function. For this purpose, we can search for a direction in the weight-space that would improve our weights, giving a lower loss: thinking of W as a weight vector, we can compute the best direction along which we should change this weight vector, which is mathematically guaranteed to be the direction of the steepest descend (in the limit as the step size goes towards zero).
In simple terms, in one-dimensional functions the slope is the instantaneous rate of change of the function at any point; so the gradient is a generalization of slope for functions which take a vector of numbers (instead of a single number) and we can consider it as a vector of slopes (derivatives) for each dimension in the input space. The mathematical expression for the derivative of a 1-D function with respect to its input is given by:

$$\frac{df(x)}{dx} = lim_{h\to 0}\frac{f(x+h)-f(x)}{h}$$

Then, when the functions of interest do not take a single number but a vector of numbers, we refer to derivates of the function with respect to each variable as *partial derivatives* and the *gradient* is the vector ∇f of partial derivatives in each dimension.

The procedure of repeatedly evaluating the gradient and performing a parameter update is named *Gradient Descent*, that is currently by far the most established way of optimizing Neural Network loss functions (with suitable smoothness properties, e.g. differentiable).

Particularly, *Stochastic Gradient Descent* (SGD) is an iterative method for optimizing the loss function. It is called stochastic as it uses randomly selected (or shuffled) samples to evaluate the gradients. Therefore, SGD is a sort of stochastic approximation of the common gradient descent optimization.

In large-scale applications, the training data can have on order of millions of samples, thus it is wasteful to compute the loss function over the entire training set in order to perform one single parameter update. A common approach to address this matter is to compute the gradient over batches of training data, which are then used to perform a parameter update. The reason for which this strategy works well is that the samples in the training data are correlated. Hence, in practice, much faster convergence can be achieved by evaluating the mini-batch gradients to perform more frequent parameter updates [14]. The *batch size* is a hyperparameter usually based on memory constraints, or set to some value, e.g. 64, 128 or 256; in general we use powers of 2 because many operation implementations work faster if their inputs are sized in powers of 2.

3.1.4 Backpropagation

Backpropagation is a strategy to compute gradients of expressions through recursive application of chain rule [14].
In Neural Networks, given a loss function $f(x)$ where x is a vector of inputs (which consist of the training data and the Neural Network weights), we are interested in computing the gradient of f at x (i.e. $\nabla f(x)$). Since we do not have control over the training data but over the weights, we compute the gradient only for the parameters (e.g. W, b) so that we can use it to perform

a parameter update.

To understand this mechanism, we can look at the example in figure 3.3, where the real-valued "circuit" shows the visual representation of the computation. The forward pass computes values from inputs to output (in green), whereas the backward pass performs backpropagation which starts at the end and recursively applies the chain rule to compute the gradients (in red) all the way to the inputs of the circuit. Hence, the gradients can be thought of as flowing backwards through the circuit.

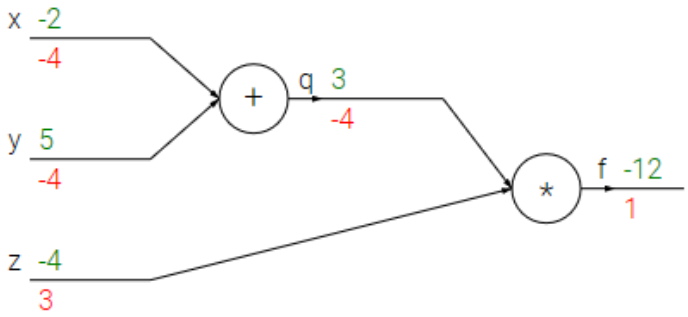

Figure 3.3: Intuitive understanding of *backpropagation* through a simple visual representation of the computation [14].

Now, observe that backpropagation is a local process: each gate in a "circuit" diagram gets some inputs and can right away compute both its output value and the local gradient of its inputs with respect to its output value. Moreover, we notice that every gate executes this computation completely independently, without being aware of any of the details of the full circuit that they are embedded in. However, once the forward pass ends, during backpropagation process the gate will eventually learn about the gradient of its output value on the final output of the entire circuit. The *chain rule* says that the gate is allowed to take that gradient and multiply it into every gradient it normally computes for all of its inputs. Therefore, we can think of backpropagation as gates communicating to each other (through the gradient signal) whether they want their outputs to increase or decrease (and

how strongly), in order to make the final output value higher.

Looking at the example above, the add gate received inputs $[-2, 5]$ and computed output 3. Since this gate computes an addition operation, its local gradient for both of its inputs is $+1$, then the rest of the circuit computed the final value, that is -12. During the backward pass in which the chain rule is applied recursively backwards through the circuit, this add gate (which represents an input to the multiply gate) learns that the gradient for its output was -4. Supposing the circuit to "want" to output a higher value (which helps with intuition), then we can think of the circuit as "wanting" the output of the add gate to be lower (owing to negative sign) and with a force of 4. Now, to continue the recurrence and to chain the gradient, the add gate takes that gradient and multiplies it to all of the local gradients for its inputs, i.e.: it makes the gradient on both x and y equal to $1 * (-4) = -4$. Notice that whether x, y were to decrease (responding to their negative gradient) then the add gate's output would decrease, which in turn makes the multiply gate's output increase, as desired.

Lastly, we observe that the operation computed at each gate is relatively arbitrary (any kind of differentiable function can act as a gate); moreover, caching forward pass variables is very useful in order to make them available during backpropagation to compute the backward pass.

3.2 Neural Network

Originally, the area of Neural Networks has been inspired by the goal of modeling biological neural systems, then it has diverged and become a matter of engineering, achieving good results in Machine Learning tasks.

3.2.1 Inspiration from biology

The fundamental computational unit of the brain is the neuron: around 86 billion neurons can be found in the human nervous system, connected to each other through synapses [14].

The figure 3.4 shows a *biological neuron*: every neuron receives input signals from its dendrites and produces output signals along its axon. The axon eventually branches out and connects the neuron via synapses to dendrites of other neurons.

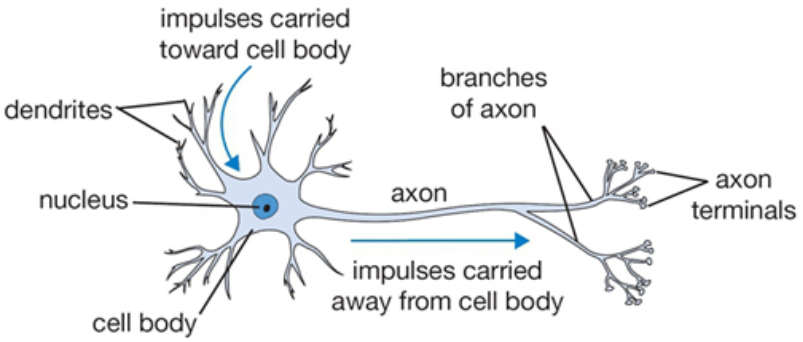

Figure 3.4: Simplified biological neuron.

In the *mathematical model* that we can visualize in figure 3.5, the signals that travel along the axons (e.g. x_0) interact multiplicatively (e.g. $w_0 x_0$) with the dendrites of the other neuron according to the synaptic strength at that synapse (e.g. w_0). In particular, the synaptic strengths (the weights w) are learnable and control the strength of influence (and its direction, which is excitory if the weight is positive, or inhibitory if the weight is negative) of one neuron on another. Then, the dendrites carry the signal to the cell body, where they all get summed: if the final sum is above a certain threshold, the neuron can fire sending a spike along its axon. In the computational model, we assume that only the frequency of the firing do communicates information, while the precise timings of the spikes do not matter. According to this interpretation, we model the firing rate of the neuron with an *activation function* f, which represents the frequency of the spikes along the axon.

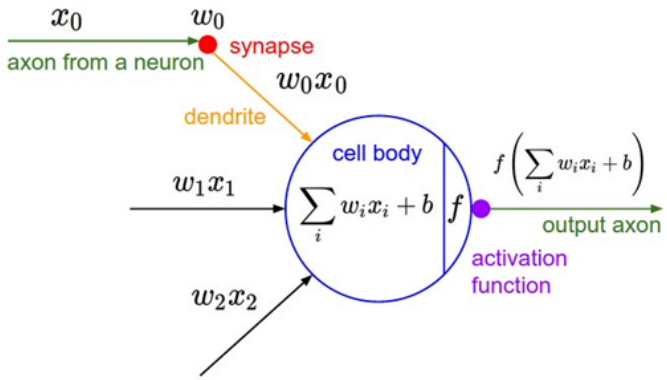

Figure 3.5: Simplified mathematical model.

3.2.2 Activation functions

Binary Softmax classifier

Following the interpretation explained above, a neuron has the capacity to "like" (activation near one) or "dislike" (activation near zero) certain regions of its input space. Hence, we can think of a single neuron as a classifier, with an appropriate loss function on its output. Considering a $Binary\ Softmax\ classifier$, we can interpret $\sigma(\sum_i w_i x_i + b)$ to be the probability of one of the classes $P(y_i = 1 \mid x_i; w)$, so the probability of the other class is $P(y_i = 0 \mid x_i; w) = 1 - P(y_i = 1 \mid x_i; w)$, since they must sum to one. In this way we can define the cross-entropy loss as already seen in the subsection 3.1.2 and then optimize it, obtaining a binary Softmax classifier (also known as logistic regression).

Rectified Linear Unit

Every activation function, also said $non - linearity$, takes a single number and performs a certain predefined mathematical operation on it. One of the most common activation functions is the $Rectified\ Linear\ Unit$, or ReLU, which computes the function $f(x) = max(0, x)$, so that the activation is simply thresholded at zero.

There are some advantages and disadvantages to using the ReLUs:

(+) it greatly accelerates the convergence of stochastic gradient descent compared to other activation functions such as Sigmoid or Tanh;

(+) it can be implemented by simply thresholding scores at zero;

(−) the ReLU units can be fragile during training and can "die": for instance, a large gradient flowing through a ReLU neuron could cause the weights to update in such a way that the neuron will never activate on any datapoint again, but then the gradient flowing through the unit will forever be zero from that point on. This means that the ReLU units can irreversibly die during training [14].

3.2.3 Neural Network architecture

Neural Networks are modeled as collections of neurons that are connected in an acyclic graph, so that the outputs of some neurons can become inputs to other neurons. Cycles are not allowed as this would imply an infinite loop in the forward pass of a network. Neural Network models are organized into distinct *layers* of neurons: for regular Neural Networks, the most common layer is the $fully\ connected$ layer in which neurons between two adjacent layers are fully pairwise connected, while neurons within a single layer share no connections (which is the typical structure of a complete bipartite graph).

Observe that when we say $N - layer\ Neural\ Network$, we do not count the input layer, so that a single-layer Neural Network describes a network with no hidden layers (i.e., input directly mapped to output).

Moreover, unlike all the others layers in a Neural Network, the *output layer* neurons most commonly do not have an activation function, or we can think of them as having a linear identity activation function. This is due to the fact that the last output layer is usually taken to represent the class scores.

Neural Networks size

As regards the size of Neural Networks, the most used metric is the number of parameters. Looking at the two examples in figure 3.6:

- the network on the left has $4+2=6$ neurons (not counting the inputs), $[3\times 4]+[4\times 2]=20$ weights and $4+2=6$ biases, for a total of 26 learnable parameters;
- the network on the right has $4+4+1=9$ neurons, $[3\times 4]+[4\times 4]+[4\times 1]=12+16+4=32$ weights and $4+4+1=9$ biases, for a total of 41 learnable parameters.

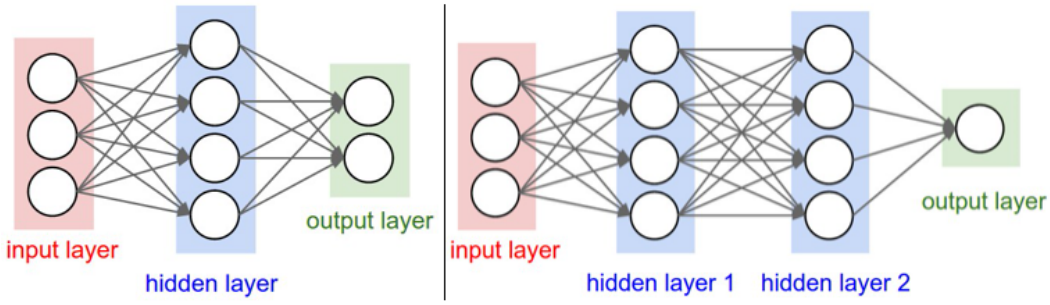

Figure 3.6: On the left, a 2−layer Neural Network (one hidden layer of 4 neurons and one output layer with 2 neurons) with three inputs. On the right, a 3−layer Neural Network (two hidden layers of 4 neurons each and one output layer with 1 neuron) with three inputs.

Number of layers

First of all, notice that as we augment the size and number of layers in a Neural Network, the capacity of the network increases, i.e.: the space of representable functions grows since the neurons can collaborate to express many different functions.
Smaller networks are harder to train with local methods such as Gradient Descent: their loss functions have relatively few local minima, but many

of these minima are easier to converge to and they have high loss. On the contrary, bigger Neural Networks contain much more local minima, but these minima are much better in terms of loss. Since Neural Networks are non-convex, it is hard to study these properties mathematically, anyway some attempts to understand these objective functions have been made: from the literature we find that, if we train a small network, then the final loss can display a good amount of variance - in some cases it converges to a good place, but sometimes it "gets trapped" in one of the bad minima. Instead, if we train a large network, we will find many different solutions, but the variance in the final achieved loss will be much smaller, which means that all solutions are about equally as good and rely less on the luck of random initialization [14].

Weight Initialization

After constructing the network, we have to initialize its parameters: one method is to calibrate the variances with $1/\sqrt{n}$. In fact, if we randomly initialize neurons, the distribution of the outputs has a variance that grows with the number of inputs. So we can normalize the variance of each neuron's output to 1 by scaling its weight vector by the square root of its number of inputs, in order to ensure that all neurons in the network initially have approximately the same output distribution and empirically improve the rate of convergence [14].

Regularization

Overfitting occurs when a model with high capacity fits the noise in the data, instead of the underlying relationship; hence several ways of controlling the capacity of Neural Networks to prevent overfitting have been studied. *Dropout* is an extremely effective and simple regularization technique: during training, dropout is implemented by only keeping a neuron active with some probability p (a hyperparameter) or setting it to zero otherwise. As showed in figure 3.7, we can interpret Dropout as sampling a Neural Network within

the full Neural Network and updating just the parameters of the sampled network based on the input data.

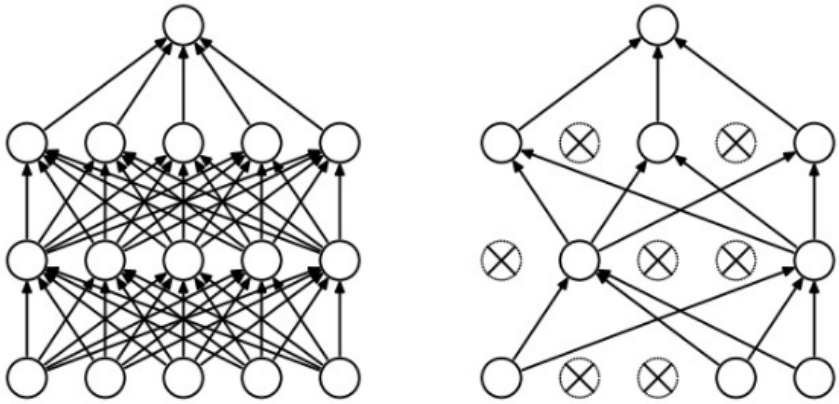

Figure 3.7: A standard Neural Network on the left, and one after applying Dropout on the right.

Loss Function

In a supervised learning problem, the data loss measures the compatibility between a prediction (e.g. the class scores in classification) and the ground truth label. The data loss takes the form of an average over the data losses for every individual example, i.e.: $L = \frac{1}{N}\sum_i L_i$ where N is the number of training data. If we abbreviate $f = f(x_i, W)$ to be the activations of the output layer in a Neural Network, in order to solve a Classification problem we assume a dataset of examples and a single correct label (out of a fixed set) for each example. As already showed in the subsection 3.1.2, a common choice is the $Softmax\ classifier$, which takes advantage of the following *cross entropy loss*:

$$L_i = -\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}\right)$$

3.2.4 Learning and Evaluation

After defining the *static* parts of a Neural Networks, such as network connectivity, data and loss function, we can treat the *dynamics* part of a network, that is the process of learning the parameters and finding good hyperparameters [14].

One useful quantity to monitor during training of a Neural Network is the *Loss*, which is evaluated on the individual batches during the forward pass. The figure 3.8 shows the loss over time. Firstly, notice that the x-axis of the plot is in units of epochs, which measure how many times every example has been seen during training in expectation (e.g. one epoch means that every example has been seen only once); it is better to track epochs rather than iterations, since the number of iterations depends on the arbitrary setting of the batch size.

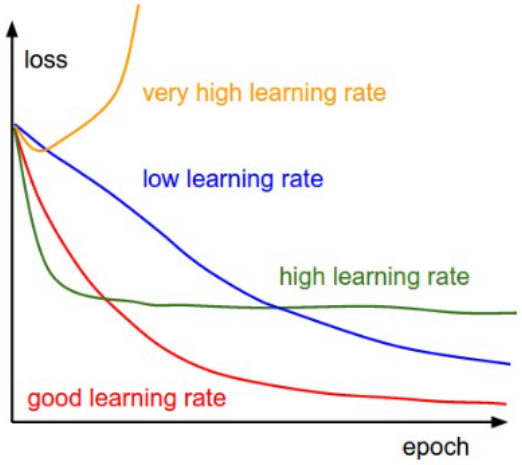

Figure 3.8: Effects of different learning rates.

In addition, we observe that the shape reveals the goodness of the *learning rate*. In fact, with low learning rates the improvements are linear, then they start to look more exponential as the learning rates increase. Higher learning rates will decay the loss faster, but they get stuck at worse values of loss (green line), since there is too much "energy" in the optimization and the parameters are unable to settle in a nice spot in the optimization field.

Lastly, looking at figure 3.9, the amount of "wiggles" in the loss is related to the *batch size*: when the batch size is 1, wiggles will be relatively high [14]; if the batch size is the full dataset, wiggles will be minimal because every gradient update should improve the loss function monotonically (unless the learning rate is set too high, as observed above).

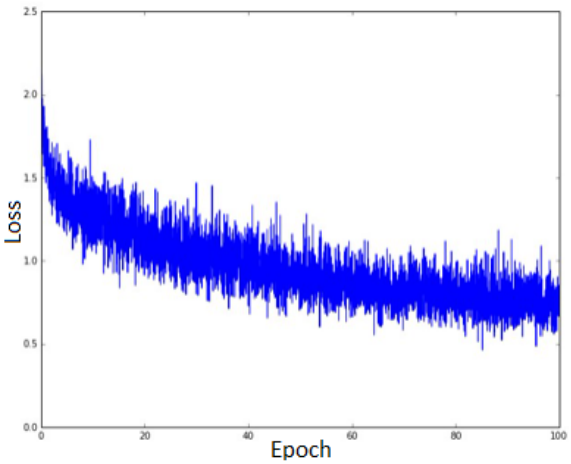

Figure 3.9: Example of a typical trend of a loss function over time, during the training.

Parameter updates

Once the analytic gradient is computed with backpropagation, the gradients are used to perform a parameter update: one of the approaches for performing the update, is the *SGD*. The simplest way to update is to change the parameters along the negative gradient direction, as the gradient indicates the direction of increase, but our goal is to minimize the loss function. Therefore, assuming a vector of parameters x and the gradient dx, the simplest update has the following form: $x = x - learning\_rate * dx$ where $learning\_rate$ is a hyperparameter, i.e., a fixed constant. When evaluated on the full dataset and if the learning rate is low enough, this is guaranteed to make non-negative progress on the loss function, as we desired.

3.3 Convolutional Neural Network

Convolutional Neural Networks, commonly said **CNN**, are similar to ordinary Neural Networks, so they are made up of neurons that have learnable weights and biases; each neuron receives some inputs, performs a dot product and possibly follows it with an non-linearity. Even in the case of CNN, the whole network still expresses a single differentiable *Score function*: from the raw image pixels on one end to class scores at the other end. CNN still have a *Loss function* (e.g. Softmax) on the last (fully connected) layer, and all the tricks and tips we developed for learning regular Neural Networks are still worth.
For the so called "*CNN architectures*" we make the explicit assumption that the inputs are *images*, allowing us to deduct certain properties into the architecture [14].

3.3.1 CNN architecture

As we explained in the previous section, a Neural Network receives a single vector as input and transforms it through a series of hidden layers. Each hidden layer consists of a set of neurons where each neuron is fully connected to all neurons in the previous layer, but neurons in a single layer are completely independently and do not share any connections. The last fully connected layer is the so called "output layer" and represents the class scores in classification settings.

Now, considering Convolutional Neural Networks, these take advantage of the fact that the input is made up of images and they constrain the architecture in a more sensible way [14]. Particularly, the layers of a CNN have neurons arranged in *three dimensions*: width, height, depth. We observe that in this contest *width* and *height* would be the dimensions of the image, while *depth* refers to the third dimension of an activation volume, not to the depth of a full Neural Network (which is given by the total number of layers in the network). As we will see, neurons in a layer are only connected to a

small region of the layer before it, instead of all of the neurons as occurs in a fully connected manner. In addition, by the end of the CNN architecture, the full image will be reduced into a single vector of class scores arranged along the depth dimension.

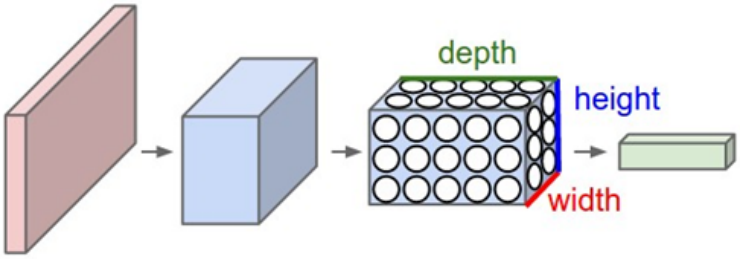

Figure 3.10: A simple example of CNN, where the red input layer holds the image (its width and height are the dimensions of the image, while its depth represents the 3 channels, red, green and blue); then each layer of a CNN transforms the 3D input volume to a 3D output volume of neuron activations.

Thus, a common CNN is a sequence of layers where each layer transforms one volume of activations to another through a differentiable function [14]. We will stack these layers in order to give rise to a complete CNN architecture; according to the leterature in this area, the following CNN structure illustrates a very common layer pattern:

- *Input layer* - it holds the raw pixel values of the image; this layer should be divisible by 2 many times, hence common input layer sizes include numbers such as 64, 384, or 512;

- *Conv layer* - the Convolutional layer computes the output of neurons that are connected to local regions in the input, each computing a dot product between their weights and the small region they are connected to in the input volume;

- *ReLU layer* - the Rectified Linear Unit applies an elementwise activation function, such as the thresholding at zero, $max(0, x)$; so we

can interpret this activation function as a layer that applies a simple non-linearity;

- *Pool layer* - the Pooling layer performs a downsampling operation along the spatial dimensions (width, height);
- *FC layer* - the Fully Connected layer computes the class scores; hence, each of its numbers correspond to a class score. As occurs in ordinary Neural Networks, each neuron in this layer is connected to all the numbers in the previous volume.

This pattern is repeated until the image has been merged spatially to a small size, then the last fully connected layer holds the output (class scores); i.e., following a pattern like that, a CNN transforms the original image layer by layer from the original pixel values to the final class scores [14].

Note that some layers contain parameters and others do not: for instance, *ReLU* and *Pool* layers will implement a fixed function. On the other hand, the *Conv* and *FC* layers perform transformations that are a function of both the activations in the input volume and the parameters, i.e. weights and biases of the neurons, which will be trained with gradient descent so that the class scores that the CNN computes are consistent with the labels in the training set for each image.

Convolutional Layer

The Convolutional layer (we can say *Conv* layer) is the core building block of a Convolutional Neural Network. The Conv layer's parameters consist of a set of learnable filters: each filter is small spatially, along width and height, but it extends through the full depth of the input volume, as we can observe in figure 3.11. For instance, a common filter on a first layer of a CNN may have size $5 \times 5 \times 3$ (i.e. 5 pixels width and height, and 3 pixels depth which represent the three color channels). During the forward pass, we slide (or convolve) every filter across the width and height of the input volume and compute dot products between the entries of the filter and the input

at any position: since we slide the filter over the width and height of the input volume, we will obtain a $2-$*dimensional activation map* that gives the responses of that filter at every spatial position. Intuitively, the network will learn filters which activate when they "see" some type of visual feature we are interested in, such as an edge of some orientation or a blotch of some color on the first layer, or some precise patterns on higher-level layers of the network [14].
In this way, we have now an entire set of filters in each Conv layer, each of them producing a separate 2-dimensional activation map: thus, we will stack these activation maps along the depth dimension in order to produce the output volume.

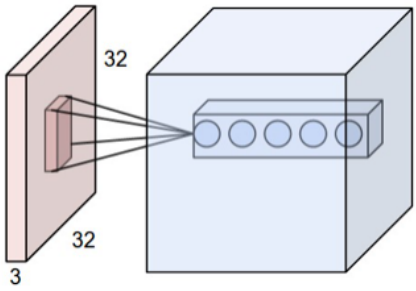

Figure 3.11: An example of input volume in red and of a volume of neurons in the first Conv layer: every neuron in the Conv layer is connected only to a local region in the input volume spatially, but to the full depth; so there are multiple neurons (here 5) along the depth, all looking at the same region in the input volume.

According to a *biological interpretation*, we can think of every entry in the 3D output volume as an output of a neuron that looks at only a small region in the input and shares parameters with all neurons to the left and right spatially (as these numbers all result from applying the same filter).

In the case of CNN, since we deal with high-dimensional inputs such as images, it is unobtainable to connect neurons to all neurons in the previous volume; so, it is much more convinient to connect every neuron to a local region of the input volume. The spatial extent of this connectivity is a

hyperparameter said *receptive field* of the neuron, or *filter size*. First, we observe that the extent of the connectivity along the depth axis is always equal to the depth of the input volume. Hence, it is important to emphasize that the connections are local in space (along width and height), but always full along the entire depth of the input volume.
The formula for calculating the *spatial size of the output volume* is given by

$$\frac{W - F + 2P}{S} + 1$$

where W is the input volume size, F is the receptive field size (or filter size) of the Conv layer neurons, S is the stride with which they are applied, and P is the amount of zero padding used on the border.
In particular, we can summarize the three crucial hyperparameters which control the size of the output volume as follow:

- the depth of the output volume corresponds to the *number of filters* we would like to use, each learning to search for something different in the input. For instance, if the first Convolutional layer takes as input a raw image, then different neurons along the depth dimension might activate in presence of various oriented edges or blobs of color. So we can define as *depth column* a set of neurons that are all looking at the same region of the input (see figure 3.11);

- the *stride* with which we slide the filter, allow us to produce smaller output volumes spatially: if the stride is 1, then we move the filters one pixel at a time; if the stride is 2, then the filters jump 2 pixels at a time as we slide them around;

- *zero padding* allow us to control the spatial size of the output volumes, by padding the input volume with zeros around the border. In general, it is very common to use zero-padding in order to ensure that the input volume and the output volume will have the same size spatially (where "size spatially" means width and height). For example, if the stride is $S = 1$, we can set zero padding to be $P = \frac{F-1}{2}$.

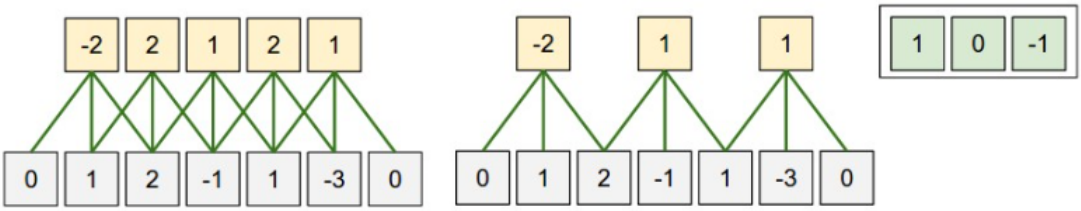

Figure 3.12: A simple example with only one spatial dimension (along the x-axis), input size W=5, one neuron with a receptive field size of F=3, and zero padding of P=1. The biases are zero and the neuron weights are $[1, 0, -1]$ (shown on the right), which are shared across all yellow neurons. On the left, the neuron strided across the input in stride of S=1, so that the output has a size $(5 - 3 + 2)/1 + 1 = 5$. On the right, the stride is S $= 2$, giving an output of size $(5 - 3 + 2)/2 + 1 = 3$. Observe that stride S $= 3$ could not be used, as it would not fit neatly across the volume, in fact $5 - 3 + 2 = 4$ is not divisible by 3.

Notice that the spatial arrangement hyperparameters have mutual *con − straints*, thus we have to size CNN in an appropriate way so that all the dimensions "work well". We can alleviate this matter through the use of zero-padding and some design guidelines.

Lastly, *parameter sharing scheme* is used in Convolutional Layers to control the number of parameters. Looking at real-world examples, it turns out that we often have a very high number of parameters, but we can dramatically reduce this number by making one reasonable assumption: if one feature is useful to compute at some spatial position (x, y), then it is useful to compute also at a different position (x_2, y_2). That is, denoting a single 2-dimensional slice of depth as a *depth slice*, we will constrain the neurons in each depth slice to use the same weights and bias. Thanks to this parameter sharing scheme, the Conv layer would now have a much smaller and unique set of weights (one for each depth slice). In other words, all neurons in each depth slice will now be using the same parameters. Practically, during backpropagation, each neuron in the volume computes the gradient for its

weights, then these gradients will be added up across each depth slice and only update a single set of weights per slice.
Observe that if all neurons in a single depth slice are using the same weight vector, then the forward pass of the Conv layer can in each depth slice be computed as a convolution of the neuron's weights with the input volume, for this reason we refer to this operation as *Convolutional Layer* and we commonly refer to the sets of weights as a filter (or a kernel), that is convolved with the input.

To sum up, we can schematize the action of the Conv layer as follow [14]:

- it takes as input a volume of size $W_1 \times H_1 \times D_1$;
- it requires to set four hyperparameters:
  - the number of filters K
  - the spatial extent of each filter F
  - the stride S
  - the zero padding "size" P
- it outputs a volume of size $W_2 \times H_2 \times D_2$, where:
  - $W_2 = \frac{W_1 - F + 2P}{S} + 1$
  - $H_2 = \frac{H_1 - F + 2P}{S} + 1$
    (that means that width and height are computed equally by symmetry);
  - $D_2 = K$
- thanks to the parameter sharing strategy, it introduces $F \cdot F \cdot D_1$ weights per filter, for a total of $(F \cdot F \cdot D_1) \cdot K$ weights and K biases;
- looking at the output volume, the d-th depth slice (which has a size of $W_2 \times H_2$) is the result of performing a valid convolution of the d-th filter over the input volume with a stride of S, and then offset by d-th bias.

In general, Conv layers should be using small filters such as 3×3 or at most 5×5, with a stride of $S = 1$, and padding the input volume with zeros in such way that the Conv layer preserves the spatial dimensions of the input. For instance, if $F = 3$, then using $P = 1$ will retain the original size of the input; generally, with $P = \frac{F-1}{2}$ the input size remains unchanged.

Pooling Layer

This type of layers is built upon lower level and more complex information. In Convolutional Neural Networks, it is used to make the detection of certain features in the input invariant to scale and orientation changes. It helps overfitting by providing an abstracted form of the representation, and reduces the computational cost by reducing the number of parameters to learn. In particular, the pooling operation consists in placing windows in each feature map and keeping one value per window, so that the resulting feature maps are sub-sampled. Thus, pooling enables to move from high resolution data to lower resolution information [3].

Between successive Conv layers in a CNN architecture, it is common to periodically insert a Pooling layer, which progressively reduces the spatial size of the representation in order to reduce the amount of parameters and computation in the network, thus to also control overfitting. This layer operates independently on every depth slice of the input and resizes it spatially by using the MAX operation, giving rise to *max pooling*.

We can schematize the action of the Pooling Layer as follow [14]:

- it takes as input a volume of size $W_1 \times H_1 \times D_1$;
- it requires two hyperparameters:
  - the spatial extent of each filter F
  - the stride S
- it outputs a volume of size $W_2 \times H_2 \times D_2$ where:
  - $W_2 = \frac{W_1 - F}{S} + 1$

- $H_2 = \frac{H_1 - F}{S} + 1$
  (that means that width and height are computed equally by symmetry);
- $D_2 = D_1$

- it does not introduce new parameters, since it computes a fixed function of the input

Notice that for Pooling layers it is not common to pad the input using zero-padding and, in practice, there are only two common variations of the max pooling layer: one with $F = 3,\ S = 2$ (also known as overlapping pooling) and one with $F = 2,\ S = 2$. The most common form is the second one (see the example in figure 3.13), with filters of size 2×2 applied with a stride of 2, which downsamples each depth slice in the input by 2 along both width and height, discarding exactly 75% of the activations in the input volume. In this case, each MAX operation takes a max over 4 numbers (i.e., little 2×2 region in some depth slice), while the depth dimension remains unchanged.

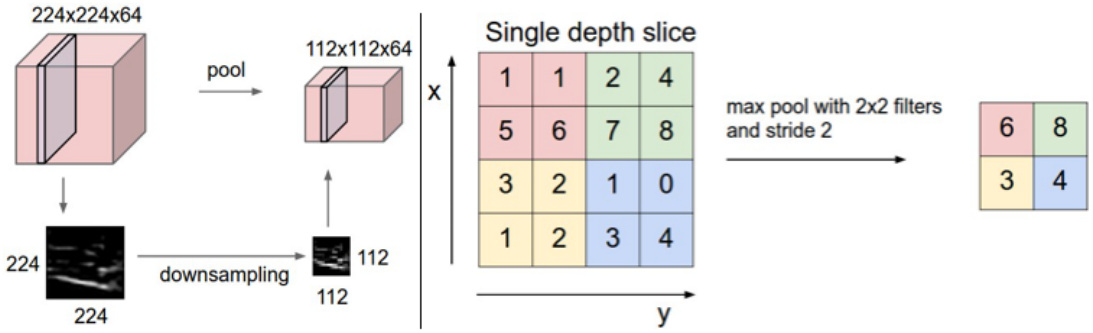

Figure 3.13: Here a Pool layer. On the left, an example of input volume of size $[224 \times 224 \times 64]$, which is pooled with filter size 2 and stride 2 into an output volume of size $[112 \times 112 \times 64]$, so the volume depth is preserved. On the right, a simple representation of max pooling with a stride of 2 and filters 2×2, so that each max is taken over little 2×2 squares.

Fully Connected Layer

In a Fully Connected layer, neurons have full connections to all activations in the previous layer, as occurs in regular Neural Networks, thus their activations can be computed with a matrix multiplication followed by a bias offset. See subsection 3.2.3.

3.3.2 Computational considerations

When constructing CNN architectures, we must be aware of its memory bottleneck. Many of the modern GPUs have a limit of 3/4/6 GB memory, with the best GPUs having about 12GB of memory [14]. So we have to keep track of three major sources of memory:

- First, from the intermediate volume sizes. These are the raw number of activations at each layer of the CNN and their gradients (of the same size). In general, most of the activations are on the first Conv layers and are needed for backpropagation; but a smart implementation that runs a CNN only at test time, could reduce this by a huge amount, by only storing the current activations at any layer and discarding the previous activations on layers below.

- Secondly, from the parameter sizes. These numbers hold the network parameters and their gradients during backpropagation, and commonly also a step cache if the optimization is using momentum. Hence, the memory to store the parameter vector must usually be multiplied by a factor of at least 3.

- Lastly, each CNN implementation has to maintain miscellaneous memory, such as the image data batches, their augmented versions, and so on.

Once we have a rough estimate of the total number of values, this number should be converted to size in GB, in order to check the amount of memory we need. If the network does not fit, a common way to "make it fit" is to

decrease the batch size, as most of the memory is usually consumed by the activations.

Chapter 4

The Code

The code has been written in **Python**.

Python is an object-oriented and high-level programming language, with dynamic semantics. These features allow Python to express powerful ideas in few lines of code, while being very readable at the same time. This is possible also thanks to the large set of libraries (i.e., sets of routines and written functions that carry out a specific task) holded by Python; then, these libraries can be recalled according to the issue.
The combination of shorter development times, flexibility and consistent syntax, make Python suitable for the development of sophisticated forecasting models, production systems and machine learning [16][18].

To develop this code, I particularly took advantage of the *PyTorch* library, a popular Deep Learning library created by Facebook: this library aims to give users a fast and flexible modeling experience, spreading PyTorch to the Deep Learning community. Even if PyTorch is less mature than Tensorflow (a very common library in deep learning areas), its community is growing rapidly since it is easier to learn and use [17].

In this chapter I will examine each script composing the code, starting with the data preprocessing tecnques; then I will provide a detailed description of the Convolutional Neural Network I implemented; lastly, I will describe the evaluation part.

4.1 Preprocessing

The input data have a crucial role in the deep learning based approaches, as well as in every machine learning algorithm in general. Since the training process benefits from an adequate preprocessing, in this section I will describe the approach I used to generate the preprocessed patches of retinal images. The following preprocessing techniques has been applied to both the training and testing datasets of the DRIVE database.

So, the image preprocessing method used on this study starts with the conversion from the original RGB image into a black and white one, through the function `rgb2gray` which converts the image to a single channel. Then I applied the `dataset_normalized` function, that is a Gray-scale normalization owing to the fact that as a result of the acquisition process, retinal fundus images are often non-uniformly illuminated and show local contrast and luminosity variability. So I adjusted luminosity and contrast of the images by subtracting the global mean and diving by the standard deviation, as follows:

$$x' = \frac{x - \bar{x}}{\sigma}$$

In the segmentation of retinal vessels is important to detect also smaller vessels, such as terminal vessels and small ramifications of the vascular tree. As they frequently exhibit a low contrast to the background, through the `clahe_equalized` function I performed a contrast limited adaptive histogram equalization operating in 8×8 square tiles (so the image has been divided into small blocks of size 8×8) in order to enhance the contrast between these small vessels and the background. In this way I approximated the histogram of the intensities inside each square tile to a given histogram: even though this approach does not improve meaningfully the contrast of large structures, it makes the more subtle features more pronounced [3].
To further improve the delineation of smaller vessels, I performed a gamma correction applying the function `adjust_gamma`: this tecnique builds a lookup table mapping the pixel values in the range $[0,\ 255]$ to their adjusted gamma

values.

Now, in order to take advantage of deep learning strategies to segment the vasculature, we need to extract small *patches* from retinal images, as the DRIVE dataset (like many other public databases) does not contain many images. Particularly, in this project I considered square patches, as the majority of the literature suggests, in order to achieve a better coherence between the data and the network. Thus, images are then divided into overlapping patches that are subsequently used as input for the Neural Network: from each image of the *training dataset*, 2000 square patches of size 48×48 have been extracted from the FOV region (checking this constraint through the function `is_patch_inside_FOV`), whereas I extracted 11445 square patches from each image of the *testing dataset*.

4.2 Utils_dataset

In this short script I defined the `class Retina`, that is crucial in order to store and load the extracted patches from the images of both the training dataset and the testing dataset. In addition, this script contains the `Scale` function, which simply scales the input so that its values range in $[-1, 1]$.

4.3 Model

The goal of this script is to train an *Encoder−Enhanced Fully Convolutional Neural Network*, to segment the retinal vasculature and to implicitly learn a compact representation of such vasculature simultaneously.

The general methodology is an element-wise classification where the CNN is trained patch by patch and the output is the probability of the center pixel being in a certain class (we consider a two class classification, as each pixel may be vessel or non vessel). This approach use a large number of small square patches from a preprocessed training set of images as input of the Neural Network. So the network is trained with said data and then tested

with patches from the test dataset.

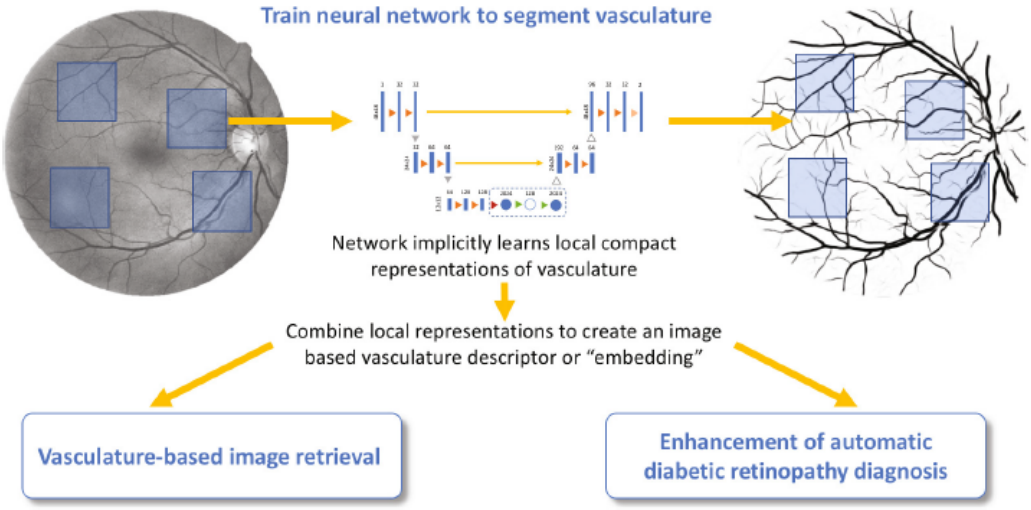

Figure 4.1: Diagram of this study, where I trained an Encoder-Enhanced Fully Convolutional Neural Network to segment the retinal vasculature and learn a compact representation of such vasculature at the same time [1].

In a *patch segmentation approach*, the CNN predicts the probability of each pixel in the patch to belong to a vessel. The **U-net** architecture is an $encoder - decoder$ type network for image segmentation, whose name derives from its unique shape. In this architecture, the feature maps of decreasing resolution in the downsampling step are fed into the convolutional layers in the upsampling step (skip-layer connections). The contracting path is an encoder which captures contextual information by successively reducing the resolution of feature maps, while the connection path combines feature maps from the contracting and expanding paths to fuse local information with accurate localization [2].
Now, looking at figure 4.2, we can observe that the **contracting path** follows the typical architecture of a convolutional network, so it consists of the repeated application of two 3×3 *convolutions*, each followed by a rectifier linear unit *ReLU* and a 2×2 *max pooling* operation with a stride of 2 for downsampling, which reduce the spatial resolution and increase the number

of filters.

In contrast, every step in the **expansive path** consists of an $up-sampling$ of the feature map, followed by a *concatenation* with the correspondingly cropped feature map from the contracting path and two 3×3 "$up-convolutions$" (each followed by a $ReLU$), which decrease the number of filters and increase image resolution until reaching the final layer.

Moreover, this implementation takes advantage of *dropout* regularization layers with 0.2 ratio between the two convolutional layers at each resolution level.

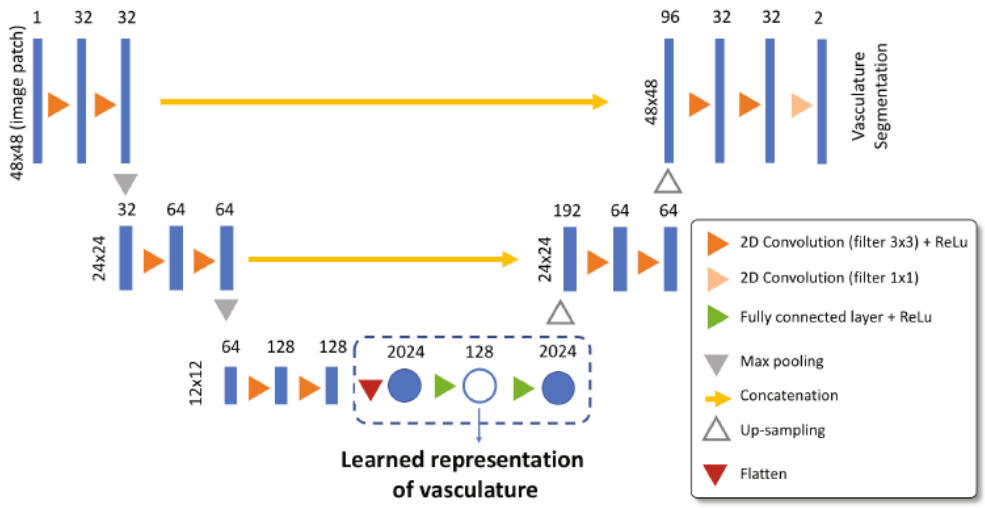

Figure 4.2: Here the U-shaped Neural Network architecture, where the dashed box represents the encoding layers, each blue box represents a 3D dimensional tensor, whose dimensions are written on the left and on the top of the box, showing the first two dimensions (image x-y) and last dimension (number of filters) respectively. Lastly, each round box represents a vector (with its size) and the arrows represent operations [1].

Lastly, the *final layer* is a convolutional layer from which the final segmentation results are obtained; in particular, as the final goal is to obtain a binary classification (whether a pixel is vessel or non vessel), in this output

layer I applied the activation function $Softmax$, given by

$$f(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

The softmax function calculates the probabilities distribution of the event over n different events, i.e.: this function will calculate the probabilities of each pixel of being a vessel or background. The main advantage of using Softmax is the output probabilities range, that is between 0 and 1, so that the sum of all the probabilities will be equal to 1.
As already mention, notice that the activation function in the hidden layers of this CNN is the standard $ReLU$ function, which is given by the equation $f(x) = x^+ = max(0; x)$. This activation function allows to greatly accelerate the convergence of stochastic gradient descent and is very efficient in terms of computation.

The main changes over a typical U-Net architecture are the **encoding layers** allowing the creation of the vasculature embedding: in fact, I relaxed the fully convolutional network approach and added three $fully\ connected$ non-convolutional layers on the saddle point of the network, where the imaging scale is lowest and the number of filters is highest. Thus, these fully connected layers are included in the U-shaped network and act as an implicit encoder to compress the relevant information flowing through the convolutional filters [1].

To train this implementation, I took advantage of the preprocessed images derived from the preprocessing (explained in 3.1); in fact, the data transformation used in combination with an activation function, is critical to the training performance and to the final results.

4.4 Main

The *Main* script contains the main flow of operations in order to perform the segmentation task, based on a powerful deep learning tecnique as Convolutionl Neural Networks. Referring to the summarized explanation at the

beginnig of chapter 3, we can identify in this scripts all the key steps of the *Image Classification* task, starting from the loading of the *input data*, then passing through the *training* part in which we wish the Neural Network to learn patterns in the input data (*training set*), and finally performing the *testing* step in order to evaluate how well the Neural Network can predict patterns in a new set of images (*testing set*).

In this applications in retina imaging, the training procedure ran for 150 *epochs* in less than one day, using a *batch size* of 256 patches. So I trained the network end-to-end applying a stochastic gradient descent (*SGD*) optimizer with a *learning rate* of 0.01, a *decay* of 1*e*-6 and a *momentum* of 0.3. Then I considered the *binary cross entropy* as loss function, which is computed uniquely between the target vessel segmentation and final layer. The result is then back-propagated using SGD to all layers, including the fully connected layers in the *encoding layers*. Once trained, the network receives an image patch as input and outputs a *vessel segmentation map* at the final layer.

To sum up, I trained the network on the DRIVE dataset using a randomly sampled set of 40000 patches extracted from the 20 images in the training set. Then, I performed the testing part on 228900 patches extracted (in an ordered and overlapped way) from the 20 images in the testing set.

4.5 Evaluation

The last script has been implemented in order to perform the *evaluation* task, to evaluate the quality of the "trained model" by testing it on a new set of images, called *testing set*.

As already showed in section 2.2, the result in the retinal vessel segmentation process is a pixel-based classification outcome, so that each pixel can be classified either as vessel or surrounding tissue. Hence, in order to segment the "reconstructed" images, I applied a *threshold* to the scores predicted by the model, assigning 1 to all the scores above the threshold, and 0 otherwise.

Therefore, in this script I compared the labels predicted by the model to

the *ground truth*, i.e., the true labels of these new images. Then, I compute the *accuracy* score, which is given by the ratio between the total number of correctly classified pixels (the sum of true positive and true negative) and the total number of pixels in the image (see section 2.2).

Figure 4.3: A *test image* from the DRIVE dataset.

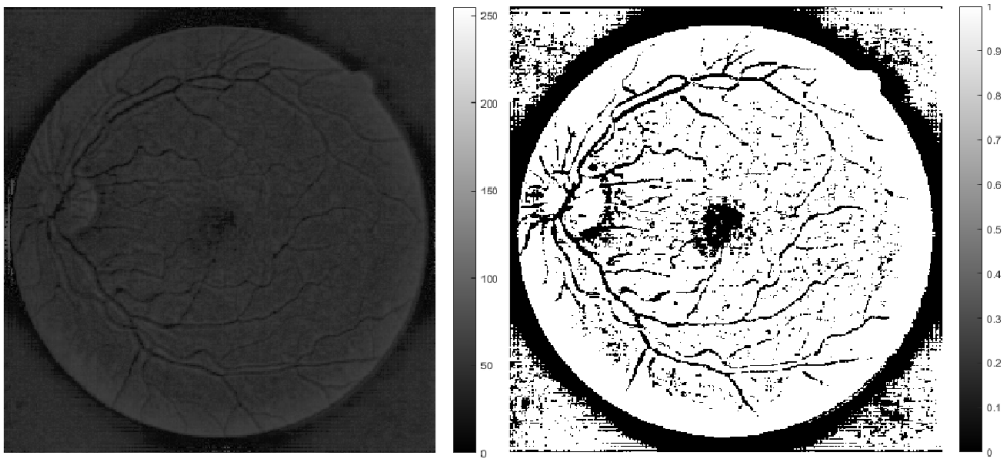

Figure 4.4: On the left, the "*reconstructed*" image correspondig to figure 4.3, before thresholging; on the right, the final *segmented* image.

Chapter 5

Conclusions and Future work

5.1 Conclusions

The focus of my research thesis was to analyse methods for the segmentation of retina fundus images based on innovative techniques from deep learning, as well as build an algorithm able to segmented fundus images from the DRIVE Dataset. Therefore, I have introduced an approach to learn vasculature embeddings from vessel segmentation data without the need of defining vasculature morphology variables a priori.

To understand how this process works and which aspects can influence its performance and results, I implemented, tested and analyzed a methodology based on Convolutional Neural Network.
The results have shown that the use of a more complex U-net architecture yields better results in the patch segmentation task, when compared to a simpler architecture without enhanced encoding layers.
Taking into account that complex networks need considerably more time to run, the training process can be further improved by using dropout between the fully connected layers of the network and by normalizing the network input data. Several attempts have shown that these two mechanisms induced lower training times and larger accuracies on the test set.
By this study, it came out that also some details and parameters such as the

learning rate and the optimizer can influence final results.
Moreover, I improved the results by adopting preprocessed images derived from different preprocessing operations, as some non-uniformity of the RGB image are eliminated, as well as a higher number of thinner vessels and other details of the microvascularization of retinal tissue become more apparent after correcting some nonuniformities of the images and enhancing small structures.

The results obtained are encouraging, even if further work is needed to test repeatability and validity as candidate image based biomarker, as well as direct comparison with alternative methods to characterize vasculature. Another interesting aspect of this work is the versatility of the U-net implementation: in fact, the methodology developed is not inherently specific to retina fundus images, so that other imaging modalities could be explored in future work.

5.2 Acknowledgements and Future work

In first place I would like to thank Professor Fabio Nicola, my thesis advisor, for the guidance and the opportunity he gave me to spend three months in Houston, where I developed this research Thesis under the direction of my second advisor Dr. Demetrio Labate, at the Department of Mathematics of the University of Houston.
To Dr. Demetrio Labate, the greatest thanks for all help and support during this journey, and also for his proposal on the following future work: we would like to develop a new research that will build upon the work I started during the last year and will bring together methods from time-frequency analysis and data science applied to a very important application from medical imaging, in additional collaboration with The University of Texas Medical Branch.

Neurodegenerative disorders such as Alzheimer's disease (AD) are a huge human and economic burden on our society, and early detection of the dis-

ease is critical for the success of any therapy. The goal of this research is to develop an innovative quantitative method for the discovery of image-based biomarkers of neurodegenerative disorders using Optical Coherence Tomography Angiography (OCTA), a new noninvasive imaging technique of the retina [13].

As an extension of the central nervous system, the retina shares many features with the brain, in terms of response to injury, immunology and physiological characteristics. Emerging evidence indicates that neurodegenerative processes associated with AD induce changes in the microvascularization of retinal tissue and this suggests the potential for utilizing OCTA to detect retinal neurovascular biomarkers of AD. However, progress in discovering robust biomarkers of AD has been slow, due to the complexity of the images, the difficulty in defining variables a priori of neurovascular dysfunction and developing quantitative methods specific to these variables. To address this challenge, we propose to integrate methods from time-frequency analysis and representation learning: the main novelty of this approach is to quantify morphology and structural changes of retinal vascularization by leveraging the internal representation of a new encoderenhanced convolutional neural network, trained with images annotated by domain experts. One major novelty is the development of an enhanced encoder section in the network consisting of convolutional filters that are constrained by imposing geometric conditions based the theory of sparse multiscale representations and are trained to learn the fundamental morphological characteristics of the retinal vascularization. In the pilot study I conducted on fundus retina images, we demonstrated the feasibility of this method, so we are confident that the application of this approach to OCTA images will effectively transfer neurovascular structural characteristics learned by the network into embedding vectors encoding critical information for biomarkers of neurodegeneration. Since OCTA is a non-invasive imaging technique, the ability to detect such biomarkers would make it very a promising method for largescale population screening and early detection of AD.

Appendix A

Python Scripts

A.1 Preprocessing

```
import os
import argparse
from tqdm import tqdm
from glob import glob
from sys import argv
import matplotlib.pyplot as plt

import cv2
import numpy as np
from PIL import Image


FLAGS = None


def is_patch_inside_FOV(x, y, img_w, img_h, patch_size):
    ''' check if the patch is fully contained in the FOV '''

    x_ = x - int(img_w/2)  # origin (0,0) shifted to image center
    y_ = y - int(img_h/2)  # origin (0,0) shifted to image center
    R_inside = 270 - int(patch_size * np.sqrt(2.0) / 2.0)
    # radius is 270 (from DRIVE db docs),
    # minus the patch diagonal (assumed it is a square
    # this is the limit to contain the full patch in the FOV)
    radius = np.sqrt((x_*x_)+(y_*y_))
    if radius < R_inside:
        return True
    else:
        return False


def zero_padding(array, patch_size, stride):
    p1 = (array.shape[0]-patch_size) % stride
    p2 = (array.shape[1]-patch_size) % stride
    if p1 != 0:
        array = np.lib.pad(array, ((0, stride-p1), (0, 0), (0, 0)),
                                    mode='constant', constant_values=0)
    if p2 != 0:
        array = np.lib.pad(array, ((0, 0), (0, stride-p2), (0, 0)),
                                    mode='constant', constant_values=0)
    return array
```

```
def extract_ordered_overlap(array, patch_size, stride):
    ''' extract patch from each image in array '''
    w, h, c = array.shape[0], array.shape[1], array.shape[2]
    assert((h-patch_size) % stride == 0 and (w-patch_size) % stride == 0)
    # print("Number of patches along h : " +str(((h-patch_size)//stride+1)))
    # print("Number of patches along w : " +str(((w-patch_size)//stride+1)))
    M, N = len(range((h-patch_size)//stride+1)),
                                        len(range((w-patch_size)//stride+1))

    N_patch = M * N
    patch = np.zeros((N_patch, patch_size, patch_size, c), dtype=np.uint8)
    k = 0
    for j in range((w-patch_size)//stride+1):
        for i in range((h-patch_size)//stride+1):
            patch[k] = array[j*stride:(j*stride)+patch_size,
                                                i*stride:(i*stride)+patch_size]
            k += 1
    return patch


def recompone_overlap(preds, img_w, img_h, stride):
    # the reference array is the original test images
    assert (len(preds.shape)==4)  #4D arrays
    assert (preds.shape[1]==1 or preds.shape[1]==3)# check the channel is 1 or 3
    patch_size = preds.shape[2]   # == preds.shape[3]
    N_patches_w = (img_w-patch_size)//stride+1
    N_patches_h = (img_h-patch_size)//stride+1
    N_patches_img = N_patches_w * N_patches_h
    N_image = int(len(preds) / N_patches_img)
    print('\nN_image: ', N_image)
    print("N_patches_w: " +str(N_patches_w))
    print("N_patches_h: " +str(N_patches_h))
    print("N_patches_img: " +str(N_patches_img))
    print("According to the dimension inserted, there are "+str(N_image)+
                      " full images (of " +str(img_w)+"x" +str(img_h) +" each)")

    # initialize to zero mega array with sum of Probabilities
    full_prob = np.zeros((N_image, preds.shape[1],img_w,img_h),
                                                        dtype=np.float32)
    full_sum = np.zeros((N_image, preds.shape[1],img_w,img_h),
                                                        dtype=np.float32)

    #iterator over all the patches
    # nrows, ncols = len(range(N_patches_w)), len(range(N_patches_h))
    k = 0
    for i in range(N_image):
        for w in range(N_patches_w):
            for h in range(N_patches_h):
                full_sum[i, :, w*stride:(w*stride)+patch_size,
h*stride:(h*stride)+patch_size] += 1
                full_prob[i, :, w*stride:(w*stride)+patch_size,
h*stride:(h*stride)+patch_size] += preds[k]
                k += 1

    assert(np.min(full_sum) >=  1.0)  #at least one
    final_avg = full_prob/full_sum
    print('final_avg.shape: ', final_avg.shape)
    assert(np.min(final_avg) >= 0.0)  # min value for a pixel is 0.0
    assert(np.max(final_avg) <= 1.0) # max value for a pixel is 1.0

    return final_avg
```

```
# PRE_processing (use for both training and testing!)
def my_PreProc(data):
    assert (len(data.shape) == 4)
    assert (data.shape[1] == 3)  # use original images

    # Black-white conversion
    train_imgs = rgb2gray(data)

    # My preprocessing:
    train_imgs = dataset_normalized(train_imgs)
    train_imgs = clahe_equalized(train_imgs)
    train_imgs = adjust_gamma(train_imgs, 1.2)
    print('train_imgs.dtype: ', train_imgs.dtype)

    return train_imgs


# ============================================================
# ========= PRE PROCESSING FUNCTIONS ========================#
# ============================================================

def rgb2gray(rgb):
    # convert RGB image in black and white
    assert (len(rgb.shape) == 4)  # 4D arrays
    assert (rgb.shape[1] == 3)
    bn_imgs = np.float32(rgb[:, 0, :, :]*0.299 + rgb[:, 1, :, :]*0.587
                                                    + rgb[:, 2, :, :]*0.114)
    bn_imgs = np.reshape(bn_imgs, (rgb.shape[0], 1, rgb.shape[2], rgb.shape[3]))

    return bn_imgs


def dataset_normalized(imgs):
    # normalize over the dataset
    assert (len(imgs.shape) == 4)  # 4D arrays
    assert (imgs.shape[1] == 1)  # check if the channel is 1
    imgs_std = np.std(imgs)
    imgs_mean = np.mean(imgs)
    imgs_normalized = np.float32((imgs - imgs_mean) / imgs_std)
    for i in range(imgs.shape[0]):
        imgs_normalized[i] =
               np.float32(((imgs_normalized[i] - np.min(imgs_normalized[i])) /
               (np.max(imgs_normalized[i]) - np.min(imgs_normalized[i]))) * 255)

    return imgs_normalized


def clahe_equalized(imgs):
    # Contrast Limited Adaptive Histogram Equalization
    # enhances the contrast between small vessels and background
    assert (len(imgs.shape) == 4)  # 4D arrays
    assert (imgs.shape[1] == 1)  # check if the channel is 1
    # create a CLAHE object (Arguments are optional).
    clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8, 8))
    imgs_equalized = np.zeros(imgs.shape, np.uint8)
    for i in range(imgs.shape[0]):
        imgs_equalized[i, 0] = clahe.apply(np.array(imgs[i, 0], dtype=np.uint8))
    return imgs_equalized
```

```
def adjust_gamma(imgs, gamma=1.0):
    # build a lookup table mapping the pixel values [0, 255] to their
    # adjusted gamma values
    assert (len(imgs.shape) == 4)  # 4D arrays
    assert (imgs.shape[1] == 1)  # check if the channel is 1
    invGamma = 1.0 / gamma
    table = np.array([((i / 255.0) ** invGamma) * 255 for i in
                                          np.arange(0, 256)]).astype("uint8")

    # apply gamma correction using the lookup table
    new_imgs = np.zeros(imgs.shape, np.uint8)
    for i in range(imgs.shape[0]):
        # print(cv2.LUT(np.array(imgs[i, 0], dtype=np.uint8), table).dtype)
        new_imgs[i, 0] = cv2.LUT(np.array(imgs[i, 0], dtype=np.uint8), table)

    return new_imgs

#============================================================================


def main():

    # SETTINGS

    if FLAGS.computer == 'mariachiara':

        train_path='/home/dlabate/Documents/TESI/retina_pytorch/DRIVE/training'

        test_path='/home/dlabate/Documents/TESI/retina_pytorch/DRIVE/test'

        save_path='/home/dlabate/Documents/TESI/retina_pytorch/DRIVE/np_dataset'

    elif FLAGS.computer == 'sabine':  # cluster

        train_path = '/brazos/labate/DRIVE/training'

        test_path = '/brazos/labate/DRIVE/test'

        save_path = '/brazos/labate/DRIVE/np_dataset'


#============================================================================

    train_images_path = [os.path.join(train_path, 'images',
              '%d_training.tif' % k) for k in range(21, 41)] # [0:FLAGS.tot_img]

    train_masks_path = [os.path.join(train_path, 'mask',
         '%d_training_mask.gif' % k) for k in range(21, 41)] # [0:FLAGS.tot_img]

    test_images_path = sorted(glob(os.path.join(test_path, 'images',
                                                '*.tif'))) # [0:FLAGS.tot_img]

    test_masks_path = sorted(glob(os.path.join(test_path, 'mask',
                                                '*.gif'))) # [0:FLAGS.tot_img]

    assert len(train_images_path) == len(test_images_path) == \
           len(train_masks_path) == len(test_masks_path)

    Nimages, Nmasks = len(train_images_path), len(train_masks_path)
    Ntest_images, Ntest_masks = len(test_images_path), len(test_masks_path)
```

```
    train_images = np.zeros((FLAGS.N * Nimages, FLAGS.patch_size,
                         FLAGS.patch_size, FLAGS.nchannels), dtype=np.uint8)


    train_masks = np.zeros((FLAGS.N * Nmasks, FLAGS.patch_size,
                                      FLAGS.patch_size), dtype=np.uint8)

    test_images = np.zeros((FLAGS.Ntest_patches * Ntest_images,
           FLAGS.patch_size, FLAGS.patch_size, FLAGS.nchannels), dtype=np.uint8)

    test_masks = np.zeros((FLAGS.Ntest_patches * Ntest_masks,
                          FLAGS.patch_size, FLAGS.patch_size, 1), dtype=np.uint8)


#==============================TRAINING_DATASET================================

    print('\n\nTRAINING\n')
    for k, (item1, item2) in tqdm(enumerate(zip(train_images_path,
                                                    train_masks_path))):

        image = np.asarray(Image.open(item1))
        image = image[9:574, :, :]
        mask = np.asarray(Image.open(item2))
        mask = mask[9:574, :]
        # print('mask shape, image shape:  ', mask.shape, image.shape)

        W, H = image.shape[0], image.shape[1]

        l = 0
        while l < FLAGS.N:
            i = np.random.randint(FLAGS.patch_size//2,
                                      W - FLAGS.patch_size//2, 1)[0]
            j = np.random.randint(FLAGS.patch_size//2,
                                      H - FLAGS.patch_size//2, 1)[0]

            # check whether the patch is fully contained in the FOV
            if not is_patch_inside_FOV(j, i, H, W, FLAGS.patch_size):
                continue

            train_images[FLAGS.N*k+l] = image[
                                  i-FLAGS.patch_size//2: i+FLAGS.patch_size//2,
                                  j-FLAGS.patch_size//2: j+FLAGS.patch_size//2]

            train_masks[FLAGS.N*k+l] = mask[
                                 i-FLAGS.patch_size//2: i+FLAGS.patch_size//2,
                                 j-FLAGS.patch_size//2: j+FLAGS.patch_size//2]
            l += 1

    train_images = np.transpose(train_images, (0, 3, 1, 2))
    train_masks = np.expand_dims(train_masks, axis=1)
    train_masks = np.concatenate((train_masks, 1-train_masks), axis=1)
    print('images.shape', train_images.shape)
    print('masks.shape', train_masks.shape)
    train_images = my_PreProc(train_images)
    print('train_images.shape', train_images.shape)
    print('train_masks.shape', train_masks.shape)

    np.save(os.path.join(save_path, 'train_images.npy'), train_images)
    np.save(os.path.join(save_path, 'train_masks.npy'), train_masks)
```

```
#============================TESTING_DATASET=====================================

    print('\n\nTESTING\n')

    for k, (item1, item2) in tqdm(enumerate(zip(test_images_path,
                                                test_masks_path))):

        image = np.asarray(Image.open(item1))
        mask = np.expand_dims(np.asarray(Image.open(item2)), 2)

        image = zero_padding(image, FLAGS.patch_size, FLAGS.stride)
        mask = zero_padding(mask, FLAGS.patch_size, FLAGS.stride)
        # print('mask shape, image shape:  ', mask.shape, image.shape)

        Nstrides_V = (image.shape[0] - FLAGS.patch_size) //
                                                    FLAGS.stride + 1   # 109
        Nstrides_H = (image.shape[1] - FLAGS.patch_size) //
                                                    FLAGS.stride + 1   # 105

        Ntest_patches = Nstrides_V * Nstrides_H
        assert Ntest_patches == FLAGS.Ntest_patches


        test_images[FLAGS.Ntest_patches*k: FLAGS.Ntest_patches*(k+1)] = \
            extract_ordered_overlap(image, FLAGS.patch_size, FLAGS.stride)

        test_masks[FLAGS.Ntest_patches*k: FLAGS.Ntest_patches*(k+1)] = \
            extract_ordered_overlap(mask, FLAGS.patch_size, FLAGS.stride)


    test_images = np.transpose(test_images, (0, 3, 1, 2))
    test_masks = np.transpose(test_masks, (0, 3, 1, 2))
    test_masks = np.concatenate((test_masks, 1 - test_masks), axis=1)
    print('images.shape', test_images.shape)
    print('masks.shape', test_masks.shape)
    test_images = my_PreProc(test_images)
    print('test_images.shape', test_images.shape)
    print('test_masks.shape', test_masks.shape)

    np.save(os.path.join(save_path, 'test_images.npy'), test_images)
    np.save(os.path.join(save_path, 'test_masks.npy'), test_masks)


if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='PreProcessing:
                            retina fundus images _ retina imaging PyTorch')

    parser.add_argument('--tot_img', type=float, default=20,
                        help='total number of images (default: 20)')

    parser.add_argument('--N', type=float, default=2000,
                        help='total number of images extracted from each image
                              in the TRAINING (default: 9500)')

    parser.add_argument('--patch_size', type=int, default=48,
                        help='width and height of each extracted patch')
    parser.add_argument('--stride', type=float, default=5,
                        help='stride')
    parser.add_argument('--nchannels', type=int, default=3,
                        help='number of channels')
    parser.add_argument('--Ntest_patches', type=int, default=11445,
                        help='number of patches for the testing dataset')
```

```
    parser.add_argument('--computer', type=str, default='mariachiara',
                        help='RUN: mariachiara or sabine; TERMINAL: argv[1]')

    FLAGS, _ = parser.parse_known_args()
    main()
```

A.2 Utils_dataset

```
import torch
import numpy as np
from torch.utils.data import TensorDataset


class Retina(TensorDataset):
    def __init__(self, images_path, masks_path, transform=None):
        super(Retina, self).__init__()
        self.images = np.load(images_path)
        self.masks = np.load(masks_path)
        self.transform = transform

    def __getitem__(self, idx):
        sample = \
            {
                'image': self.images[idx],
                'mask': self.masks[idx],
            }

        if self.transform:
            sample = self.transform(sample)
        return sample

    def __len__(self):
        # print('len(self.images): ', len(self.images))
        return len(self.images)


class Scale(object):
    def __call__(self, sample):
        sample['image'] = np.float32((1./255) * sample['image'])
        sample['mask'] = np.float32((1./255) * sample['mask'])
        return sample


class ToTensor(object):
    """Convert ndarrays in sample to Tensors."""

    def __call__(self, sample):
        # print(sample['image'].shape, sample['mask'].shape)
        return {'image': torch.from_numpy(sample['image']),
                'mask': torch.from_numpy(sample['mask'])}
```

A.3 Model

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import math


class Unet(nn.Module):
    def __init__(self, input_channels):

        super(Unet, self).__init__()
        self.c_in = input_channels
        self.nonlinearity = nn.ReLU(inplace=True)

        # 1 - Contracting path
        self.conv1 = nn.Conv2d(self.c_in, 32, 3, padding=1)
        self.dp1 = nn.Dropout(.2)
        self.conv2 = nn.Conv2d(32, 32, 3, padding=1)

        # 2
        self.conv3 = nn.Conv2d(32, 64, 3, padding=1)
        self.dp2 = nn.Dropout(.2)
        self.conv4 = nn.Conv2d(64, 64, 3, padding=1)

        # 3
        self.conv5 = nn.Conv2d(64, 128, 3, padding=1)
        self.dp3 = nn.Dropout(.2)
        self.conv6 = nn.Conv2d(128, 128, 3, padding=1)

        # Encoding layers
        self.fc1 = nn.Linear(12 * 12 * 128, 128)
        self.fc2 = nn.Linear(128, 128 * 12 * 12)

        # 4 - Expansive path
        self.conv7 = nn.Conv2d(192, 64, 3, padding=1)
        self.dp4 = nn.Dropout(.2)
        self.conv8 = nn.Conv2d(64, 64, 3, padding=1)

        # 5
        self.conv9 = nn.Conv2d(96, 32, 3, padding=1)
        self.dp5 = nn.Dropout(.2)
        self.conv10 = nn.Conv2d(32, 32, 3, padding=1)

        # 6
        self.conv11 = nn.Conv2d(32, 2, 1, padding=0)

        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                stdv = 1. / math.sqrt(m.weight.size(1))
                m.weight.data.uniform_(-stdv, stdv)
                if m.bias is not None:
                    m.bias.data.uniform_(-stdv, stdv)
```

```
def forward(self, x):

    # 1
    x = self.conv1(x)
    x1 = self.nonlinearity(x)
    x1 = self.dp1(x1)
    x1 = self.conv2(x1)
    x1 = self.nonlinearity(x1)
    pool1 = nn.MaxPool2d((2, 2))(x1)

    # 2
    x2 = self.conv3(pool1)
    x2 = self.nonlinearity(x2)
    x2 = self.dp2(x2)
    x2 = self.conv4(x2)
    x2 = self.nonlinearity(x2)
    pool2 = nn.MaxPool2d((2, 2))(x2)

    # 3
    x3 = self.conv5(pool2)
    x3 = self.nonlinearity(x3)
    x3 = self.dp3(x3)
    x3 = self.conv6(x3)
    x3 = self.nonlinearity(x3)

    # encoding layers
    e3 = x3.view(-1, 128*12*12)
    e3 = self.fc1(e3)
    e3 = nn.ReLU(inplace=True)(e3)
    e3 = self.fc2(e3)
    e3 = nn.ReLU(inplace=True)(e3)
    e3 = e3.view(-1, 128, 12, 12)

    # 4
    u4 = F.interpolate(input=e3, scale_factor=2, mode='nearest',)
    u4 = torch.cat((x2, u4), dim=1)
    x4 = self.conv7(u4)
    x4 = self.nonlinearity(x4)
    x4 = self.dp4(x4)
    x4 = self.conv8(x4)
    x4 = self.nonlinearity(x4)

    # 5
    u5 = F.interpolate(input=x4, scale_factor=2, mode='nearest',)
    u5 = torch.cat((x1, u5), dim=1)
    x5 = self.conv9(u5)
    x5 = self.nonlinearity(x5)
    x5 = self.dp5(x5)
    x5 = self.conv10(x5)
    x5 = self.nonlinearity(x5)

    # 6
    x5 = self.conv11(x5)
    x5 = self.nonlinearity(x5)
    out = nn.Softmax2d()(x5)

    return out
```

A.4 Main

```
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision.transforms import Compose
from torch.utils.data import TensorDataset

import os
import gc
import numpy as np
import matplotlib.pyplot as plt
import argparse
from tqdm import tqdm
from multiprocessing import cpu_count
from sys import argv

from utils_dataset import Retina, Scale, ToTensor
from model import Unet
from preprocessing import extract_ordered_overlap, recompone_overlap


FLAGS = None


def train(sample, device, optimizer, model, criterion, batch_idx, N_batches,
epoch):
    image, mask = sample['image'].to(device), sample['mask'].to(device)
    # print(image.shape, mask.shape) # [256, 1, 48, 48]
    optimizer.zero_grad()
    out = model(image)
    # print(out.shape) # [256, 1, 48, 48]
    loss = criterion(out, mask)
    loss.backward()
    optimizer.step()
    if (batch_idx + 1) % FLAGS.log_interval == 0 or
                                            (batch_idx + 1) == N_batches:
        print('\n==>>> epoch: {}, batch index: {}, train loss:
                        {:.6f}'.format(epoch, batch_idx+1, loss.item()))
        gc.collect()


def test(model, test_loader, device, batch_size, scores_per_patch):
    model.eval()
    print('Total number of batches for Testing: ', len(test_loader))

    for batch_idx, sample in tqdm(enumerate(test_loader)):
        image = sample['image'].to(device)
        out = model(image)
        scores_per_patch[batch_idx * batch_size: batch_idx * batch_size +
                                                        len(out)] =

            out.cpu().data.numpy()

        if (batch_idx+1) % FLAGS.test_interval == 0:
            print('test image (.1, .25, .5, .75, .9) quantiles: ',
                  np.quantile(image.cpu().data.numpy(), (.1, .25, .5, .75, .9)))
            print('test out (.1, .25, .5, .75, .9) quantiles: ',
                  np.quantile(out.cpu().data.numpy(), (.1, .25, .5, .75, .9)))
    gc.collect()

    return scores_per_patch
```

```
def main():

    # SETTINGS

    # print("number of cpus: ", cpu_count())
    use_cuda = torch.cuda.is_available()
    kwargs = {'num_workers': cpu_count(),
              'pin_memory': True} \
        if use_cuda else {}
    device = torch.device("cuda" if use_cuda else "cpu")

    W0 = 588  # from 584
    H0 = 568  # from 565
    NpatchesV = (W0-FLAGS.patch_size)//FLAGS.stride+1  # 109
    NpatchesH = (H0-FLAGS.patch_size)//FLAGS.stride+1  # 105

    score_per_image = np.zeros((FLAGS.N_epochs, FLAGS.tot_img, 1, W0, H0),
                                                              dtype=np.float32)


#================================================================================

    if FLAGS.computer == 'mariachiara':

        main_path =
                   '/home/dlabate/Documents/TESI/retina_pytorch/DRIVE/np_dataset'

        png_path =
       '/home/dlabate/Documents/TESI/retina_pytorch/retina_imaging/test_results'

    elif FLAGS.computer == 'sabine':  # cluster

        main_path = '/brazos/labate/DRIVE/np_dataset'

        png_path = '/brazos/labate/retina_imaging/test_results'


#================================================================================

    # TRAINING
    train_ds = Retina(
        os.path.join(main_path, 'train_images.npy'),
        os.path.join(main_path, 'train_masks.npy'),
        transform=Compose([
            Scale(),
            ToTensor()])
    )
    train_loader = torch.utils.data.DataLoader(train_ds,
                      batch_size=FLAGS.batch_size, shuffle=True, **kwargs)

    # TESTING
    test_ds = Retina(
        os.path.join(main_path, 'test_images.npy'),
        os.path.join(main_path, 'test_masks.npy'),
        transform=Compose([
            Scale(),
            ToTensor()])
    )
    test_loader = torch.utils.data.DataLoader(test_ds,
                      batch_size=FLAGS.batch_size, shuffle=False, **kwargs)

#================================================================================
```

```
    model = Unet(input_channels=1)
    if torch.cuda.device_count() > 1:
        model = torch.nn.DataParallel(model)
    model.to(device)

    # Stochastic Gradient Descent optimizer
    optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.3,
                                                        weight_decay=1e-6)

    criterion = nn.BCELoss() # Binary Cross Entropy Loss function

    print('Total number of epochs: N_epochs = ', FLAGS.N_epochs)
    N_batches = len(train_loader)

    for epoch in tqdm(range(1, FLAGS.N_epochs+1)):

        # TRAINING
        print("\nEpoch ", epoch, " ==>>> Total number of batches for Training:",
                                                                    N_batches)

        model.train()

        for batch_idx, sample in tqdm(enumerate(train_loader)):
            train(sample, device, optimizer, model, criterion, batch_idx,
                                                        N_batches, epoch)

        # TESTING
        print('\nEpoch ', epoch)

        scores_per_patch = np.zeros((FLAGS.tot_img * (NpatchesV * NpatchesH), 2,
               FLAGS.patch_size, FLAGS.patch_size), dtype=np.float32)

        scores_per_patch = \
                       test(model, test_loader, device, FLAGS.batch_size,
                       scores_per_patch=scores_per_patch)

        print('\nscores_per_patch: ', scores_per_patch.shape) #(228900,2,48,48)

        score_per_image[epoch-1] = \
            recompone_overlap(np.expand_dims(scores_per_patch[:,0,:,:], axis=1),
                                                    W0, H0, FLAGS.stride)

        print('\nscore_per_image[epoch-1]: ', score_per_image[epoch-1].shape)
        print('UNIQUE: ', np.unique(score_per_image[epoch-1]))
        print('percentile: ', np.percentile(score_per_image[epoch-1].flatten(),
                                                        (25, 50, 75)))


    print('\nscore_per_image.shape: ', score_per_image.shape)
    print('UNIQUE score_per_image: ', np.unique(score_per_image))
    print('percentile score_per_image:',np.percentile(score_per_image.flatten(),
                                                        (25, 50, 75)))

    np.save(os.path.join(main_path, 'score_per_image.npy'), score_per_image)


#==============================================================================

    for i in range(FLAGS.tot_img):
        plt.imshow(score_per_image[FLAGS.N_epochs-1, i, 0, :, :])
        plt.savefig(os.path.join(png_path, 'image_%s.png' % str(i+1)))
        plt.show()
```

```
if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='retina fundus images _ retina
                                                  imaging PyTorch')

    parser.add_argument('--tot_img', type=int, default=20,
                        help='total number of images (default: 20)')

    parser.add_argument('--N_epochs', type=int, default=5,
                        help='total number of epochs (default: 100)')

    parser.add_argument('--log_interval', type=int, default=20,
                        help='how many batches to wait before printing
                              training status (default: 20)')

    parser.add_argument('--test_interval', type=int, default=100,
                        help='how many batches to wait before printing
                              testing status (default: 100)')

    parser.add_argument('--batch_size', type=int, default=256,
                        help='batch_size (default: 256)')
    parser.add_argument('--stride', type=int, default=5,
                        help='stride (default: 5)')
    parser.add_argument('--patch_size', type=int, default=48,
                        help='patch_size (default: 48)')
    parser.add_argument('--Ntest_patches', type=int, default=11445,
                        help='number of patches for the testing dataset')

    parser.add_argument('--computer', type=str, default='mariachiara',
                        help='RUN: mariachiara or sabine; TERMINAL: argv[1]')

    FLAGS, _ = parser.parse_known_args()
    main()
```

A.5 Evaluation

```
import os
import numpy as np
import matplotlib.pyplot as plt
import argparse
from tqdm import tqdm
from sklearn.metrics import accuracy_score
from PIL import Image
from glob import glob
from sys import argv

from preprocessing import zero_padding


FLAGS = None


def main():

    # SETTINGS

    if FLAGS.computer == 'mariachiara':
```

```
        test_path = '/home/dlabate/Documents/TESI/retina_pytorch/DRIVE/test'

        results_path =
       '/home/dlabate/Documents/TESI/retina_pytorch/retina_imaging/test_results'

        score_path =
                  '/home/dlabate/Documents/TESI/retina_pytorch/DRIVE/np_dataset'

    elif FLAGS.computer == 'sabine':   # cluster

        test_path = '/brazos/labate/DRIVE/test'

        results_path = '/brazos/labate/retina_imaging/test_results'

        score_path = '/brazos/labate/DRIVE/np_dataset'

    W0 = 588  # from 584
    H0 = 568  # from 565

    score_per_image = np.load(os.path.join(score_path, 'score_per_image.npy'))


#================================================================================

    print('EVALUATION')

    test_masks_path = sorted(glob(os.path.join(test_path, 'mask', '*.gif')))
                                                          # [0:FLAGS.tot_img]

    test_masks = np.zeros((FLAGS.tot_img, 2, W0, H0), dtype=np.float32)

    for k, item in tqdm(enumerate(test_masks_path)):
        mask = np.expand_dims(np.asarray(Image.open(item)), 2)
                                                    .reshape((584, 565, 1))
        mask = zero_padding(mask, FLAGS.patch_size, FLAGS.stride)
                                                    .reshape((1, W0, H0))/255
        test_masks[k] = mask
        test_masks[k] = np.concatenate((mask, 1 - mask), axis=0)

    print('\ntest_masks.shape: ', test_masks.shape)

    labels = np.expand_dims(test_masks[:,0,:,:], axis=1)
    print('\nlabels shape: ', labels.shape)
    print('labels type: ', labels.dtype)
    labels = labels.flatten()
    labels[labels > FLAGS.thres ] = 1
    labels[labels <= FLAGS.thres ] = 0
    print('UNIQUE labels: ', np.unique(labels))
    print('np.sum(labels==1): ', np.sum(labels == 1),', np.sum(labels==0): ',
                                                        np.sum(labels == 0))

    accuracy_scores = np.zeros(FLAGS.N_epochs)
    epochs = np.arange(FLAGS.N_epochs) + 1

    for epoch in range(FLAGS.N_epochs):
        print('\nEpoch: ', epoch+1)

        p = score_per_image[epoch].flatten()
        print('score_per_image type: ', p.dtype)
        p[p > FLAGS.thres ] = 1
        p[p <= FLAGS.thres ] = 0
        print('UNIQUE p: ', np.unique(p))
        print('np.sum(p==1): ', np.sum(p == 1),', np.sum(p==0): ',
```

```
                                                        np.sum(p == 0))

        # Accuracy
        accuracy = accuracy_score(labels, p)
        print('==>> epoch: {}, accuracy: {}'.format(epochs[epoch],
                                                    accuracy * 100))

        accuracy_scores[epoch] = accuracy * 100

        # Segmented images (last epoch)
        p = p.reshape(FLAGS.tot_img, 1, W0, H0)
        if epoch == (FLAGS.N_epochs - 1):
            for i in range(FLAGS.tot_img):
                plt.imshow(p[i, 0, :, :])
                plt.savefig(os.path.join(results_path, 'segm_%s.png'
                                                    % str(i + 1)))
                plt.show()


#=============================================================================

    # SAVE THE RESULTS

    accuracy_mean = accuracy_scores.mean()

    print('\nEpoch {} ==>> Accuracy: {:.5f}'.format(FLAGS.N_epochs,
                                    accuracy_scores[FLAGS.N_epochs-1]))

    print('Mean Accuracy: {:.5f}'.format(accuracy_mean))

    file_perf = open(os.path.join(results_path, 'performances.txt'), 'w')
    file_perf.write("Last epoch accuracy: 100*(TP+TN)/total_scores = %s"
                                    % accuracy_scores[FLAGS.N_epochs-1]

                    + "\nMean accurancy: %s" % accuracy_mean)

    plt.figure()
    plt.title('Accurancy values')
    plt.plot(epochs, accuracy_scores, 'b',
                                label='accuracy_mean = %0.5f' % accuracy_mean)
    plt.xlim([1, FLAGS.N_epochs])
    plt.xlabel('epochs')
    plt.ylabel('Accurancy')
    plt.savefig(os.path.join(results_path, 'Accurancy_values.png'))
    plt.show()


if __name__ == '__main__':

    parser = argparse.ArgumentParser(description='retina fundus images _ retina
                                                imaging PyTorch')

    parser.add_argument('--tot_img', type=int, default=20,
                        help='total number of images (default: 20)')

    parser.add_argument('--N_epochs', type=int, default=50,
                        help='total number of epochs (default: 100)')

    parser.add_argument('--thres', type=int, default=0.95,
                        help='threshold (default: 0.9)')

    parser.add_argument('--patch_size', type=int, default=48,
                        help='width and height of each extracted patch')
```

```
parser.add_argument('--stride', type=float, default=5,
                    help='stride')
parser.add_argument('--computer', type=str, default='mariachiara',
                    help='RUN: mariachiara or sabine; TERMINAL: argv[1]')

FLAGS, _ = parser.parse_known_args()
main()
```

Bibliography

[1] Giancardo L., Roberts K., Zhao Z. (2017). *Representation Learning for Retinal Vasculature Embeddings.* In: Cardoso M. et al. (Eds) Fetal, Infant and Ophthalmic Medical Image Analysis. OMIA 2017, FIFI 2017. Lecture Notes in Computer Science, vol 10554. Springer, Cham.

[2] Ronneberger O., Fischer P., Brox T. (2015). *U-Net: Convolutional Networks for Biomedical Image Segmentation.* In: Navab N., Hornegger J., Wells W., Frangi A. (eds) Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015. MICCAI 2015. Lecture Notes in Computer Science, vol 9351. Springer, Cham.

[3] Breda, Pedro Filipe Cavaleiro. *Deep Learning for the Segmentation of Vessels in Retinal Fundus images and its Interpretation.* Ph.D. Thesis, Faculdade de Engenharia da Universidade do Porto, October 2018.

[4] Giancardo L., *Automated fundus images analysis techniques to screen retinal diseases in diabetic patients.* Ph.D. Thesis, Université de Bourgogne, 2011.

[5] H. R Taylor, J. E Keeffe. *World blindness: a 21st century perspective.* British Journal of Ophthalmology, 85(3):261–266, 2001.

[6] B.R. Masters. *Graefe's Archive for Clinical and Experimental Ophthalmology.* In: Kanski's Clinical Ophthalmology, a systematic approach. Eighth edition Brad Bowling (2016) 917pp., 2,600 illustrations ISBN: 9780702055720 Elsevier.

[7] José G. Cunha-Vaz. *The blood–retinal barriers system. Basic concepts and clinical evaluation.* Experimental eye research, 2004, 78.3: 715-721.

[8] Howard C. Howland. *The human eye: Structure and function. clyde w. oyster.* The Quarterly Review of Biology, 75(1):90–90, 2000.

[9] M. M. Franz, et al. *Blood vessel segmentation methodologies in retinal images–a survey.* Computer methods and programs in biomedicine, 2012, 108.1: 407-433.

[10] Guoqiang Zhong, et al. *An overview on data representation learning: From traditional feature learning to recent deep learning.* The Journal of Finance and Data Science, 2016, 2.4: 265-278.

[11] Shuangling Wang, et al. *Hierarchical retinal blood vessel segmentation based on feature and ensemble learning.* In: Neurocomputing, 2015, 149: 708-717.

[12] Stéphane Mallat. *Understanding deep convolutional networks.* Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences, 2016, 374.2065: 20150203.

[13] Acner Camino, Zhuo Wang, Jie Wang, Mark E. Pennesi, Paul Yang, David Huang, Dengwang Li, and Yali Jia. *Deep learning for the segmentation of preserved photoreceptors on en face optical coherence tomography in two inherited retinal diseases.* Biomedical Optics Express 3092, Vol. 9, No. 7, 1 Jul 2018

[14] Justin Johnson, Andrej Karpathy. Stanford CS class CS231n: *Convolutional Neural Networks for Visual Recognition.* `http://cs231n.github.io`

[15] Michael Nielsen, June 2019. *Neural networks and deep learning.* `http://neuralnetworksanddeeplearning.com`

[16] Lorenzo Govoni, 2019. *Python e le librerie principali per il machine learning.*
`https://lorenzogovoni.com/python-librerie-machine-learning/`

[17] PyTorch Official Site - *From research to production.*
`https://pytorch.org`

[18] Python Official Site. `https://www.python.org`