UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA AERONÁUTICA Y DEL ESPACIO

GRADO EN INGENIERÍA AEROESPACIAL

TRABAJO FIN DE GRADO

Parametrización y segmentación de las capas de la retina mediante técnicas de aprendizaje profundo

AUTOR: Christian Mariscal Calvo

ESPECIALIDAD: Vehículos Aeroespaciales

TUTOR DEL TRABAJO: Javier de Vicente Buendía

Septiembre de 2021

Agradecimientos

Normalmente, los agradecimientos no se suelen leer y, sin embargo, son la parte más importante del proyecto. Reflejan una culminación de etapa, las personas que ayudan a que las siguientes páginas puedan existir y que éstas se sientan un poquito más valoradas.

La primera persona a la que voy a dedicar este proyecto es a mi abuela. Aunque no tenga ni idea de qué estoy haciendo. Este trabajo culmina una carrera en la que he estado fuera de mi ciudad natal y mi abuela es la persona que más sufre. Por ello, es la primera con la que quiero compartir mi felicidad y agradecer los tuppers congelados que preparaba a su nieto para llevar a Madrid a pesar de sus 80 años.

La segunda es mi novia, Luciana, por aguantarme durante toda la carrera. A pesar de los altibajos, no podría concebir estos 4 años de carrera sin haber estado con ella. Me encanta haber podido llegar aquí a tu lado y poder celebrar más éxitos juntos.

También voy a escribir una pequeña dedicatoria para la hermana de Luciana, Sara. Sólo tiene 15 años, pero apunta maneras y estoy seguro de que en unos años podré pedirle que me meta en sus agradecimientos también.

Mi familia también merece agradecerles la ayuda prestada desde que me mudé a Madrid. Sin ellos todo habría sido mil veces más difícil.

Por último, no me olvido de mi 'tutor no profesional', que es Álvaro Cuartero, así como todos aquellos que también mueven el proyecto en la sombra y son olvidados muchas veces. Aunque de 'tutor no profesional' tiene poco, ya que su implicación demuestra que es un profesional de los pies a la cabeza.

Muchas gracias a todos.

Index

Abstract

This project aims to segment retinal layers from OCT images employing machine learning and semantic segmentation. Images were provided by ophthalmology department from Hospital Universitario Fundación Jiménez Díaz. Two different segmentations were designed. First, whole retina segmentation is performed. Secondly, a more detailed one including different retinal layers is done.

First step is treating images in dataset. CV2 Python library is used for that. Then, model architectures are implemented. In particular, UNet, Resnet-50, Resnet-101, Deeplabv3-resnet 50 and Deeplabv3-resnet 101 are employed, as they provide accurate results for semantic segmentation. Then the model is trained and tested and metrics and predicted images are calculated and saved on computer or Google Drive. Metrics calculated in this project are accuracy, IoU and Binary Cross Entropy with logits loss. Finally, 3D representations of retina start and end points and its thickness are created, obtaining a better understanding of what are the main differences between a healthy and a sick patient's retina.

Results have been satisfactory, with an Intersection over Union of 80% approximately and accurate visual predictions, even avoiding original dataset mistakes in some cases.

Keywords: OCT, retina, Machine Learning, fovea, semantic segmentation, data augmentation, Python, PyTorch, One Hot Encoding, kernel, convolution.

Introduction

Fundus retinal diseases are commonly diagnosed by ophthalmologists. It is estimated that more than 300 million people in the world suffer from fundus retina diseases such as diabetic retinopathy. Studies have shown that in most fundus diseases, morphological changes occur before visual field alteration. Thus, diagnosing them is vital to avoid damages in the future.

Furthermore, in recent years developments in artificial intelligence allowed boosting calculus power, enriching scientists with a powerful tool. This project uses image segmenting to perform the retinal layers analysis. This idea is not new, as previous studies had already studied it before, obtaining highly accurate results. For example, in the study carried out by Qiaoliang Li et al. [1], an intersection over union (IoU) of 90,41% was achieved. This high performance was one of the reasons to conduct this research.

Therefore, the main aim of this project is, apart from its educational objective, to create an applicable computer program on Python to segment retinal layers and extract its main characteristics, in order to make the ophthalmologist able to determine the presence of a disease in an efficient way.

The full software program has been developed in Python, making use of libraries such as Numpy and PyTorch. There were two Artificial Intelligence (AI) libraries possible to use (TensorFlow and Pytorch), but the last one was selected as its programming method allows visualizing better the required steps to create and train a model and TensorFlow is a more compact library. This way, a more understanding is achieved by the student.

This project uses the neural network architectures UNet, Resnet50, Resnet101, deeplabv3_resnet50 and deeplabv3_resnet101 that will be explained in detail in the next sections. These architectures provided to be highly efficient to perform image segmentation, as shown in Fully Convolutional Networks for Semantic Segmentation and Rethinking Atrous Convolution for Semantic Image Segmentation

The dataset used has been provided by previous students who developed a similar thesis. It consists of 1058 Optical Coherence Tomography (OCT) images and its segmented masks. Nevertheless, not all of these images will be employed for the training, as will be explained later.

OCT is a non-invasive, micro-resolution medical imaging tool ideal for examining cross-section images in biological systems [2] that is widely used in ophthalmology as well as other medical fields and to perform non-destructive testing (NDT). It will be explained in more detail later.

The first step will be the dataset treatment, in order to prepare the images for their entering into the convolutional neural networks. After that, a model is created (based on a specific architecture) and trained. Finally, results are evaluated using different metrics.

Materials and methods

Preliminary Concepts

Retina Structure Overview

The image below shows the retina as seen by an ophthalmologist. In the centre of the retina, the optic nerve can be found, measuring about 2x1.5 mm across. The optic nerve radiates the major blood vessels of the retina. Approximately 4.5-5 mm to the left of the optic nerve a blood-vessel-free red spot called the fovea is found. The fovea is located in the middle of what is called the macula, a crucial part of the eye that allows perceiving vision details.

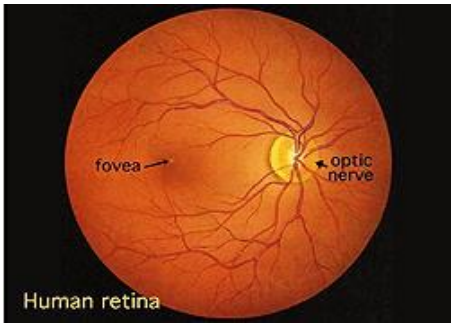

Figure 1. Human retina through an ophthalmoscope

A circular field of approximately 6 mm around the fovea is considered the central retina, whilst the rest stretches to ora serrata, up to 21 mm from the fovea. This part is called the peripheral retina. In this project, similar images to the one above will be considered (SLO images). However, most of the work developed has been done using OCT images, which will be explained in the next chapter.

The next images show the general structure of the eye, where ora serrata can be found too. It is remarkable to perceive how fovea is a little dip in the retina. This part will be very easy to distinguish in OCT images.

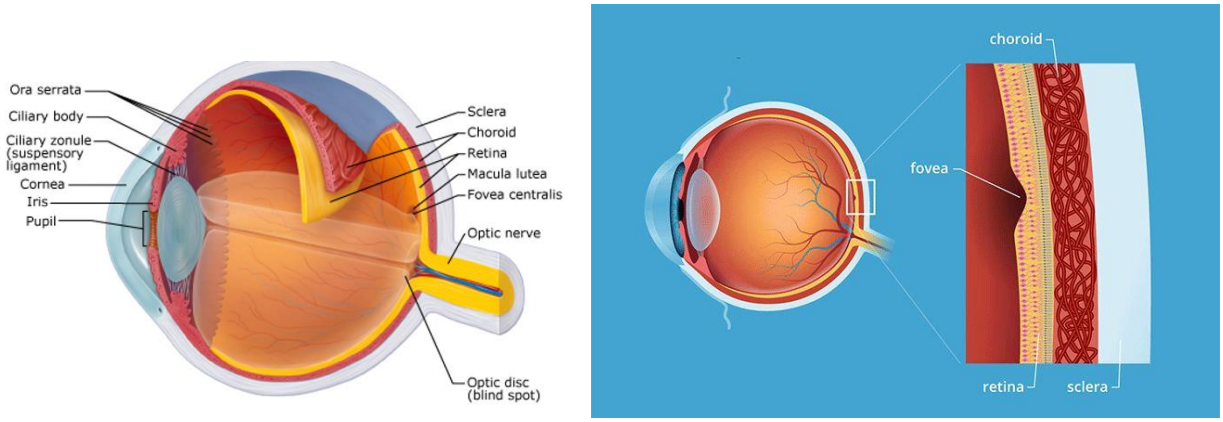

Figure 2. General eye structure

Nevertheless, the retina has a lot of different layers, which are represented in the next figure. This figure is an OCT scan, which is the main tool in this project and will provide the information needed to make the segmentation. The different layers segmented in this project are coloured.

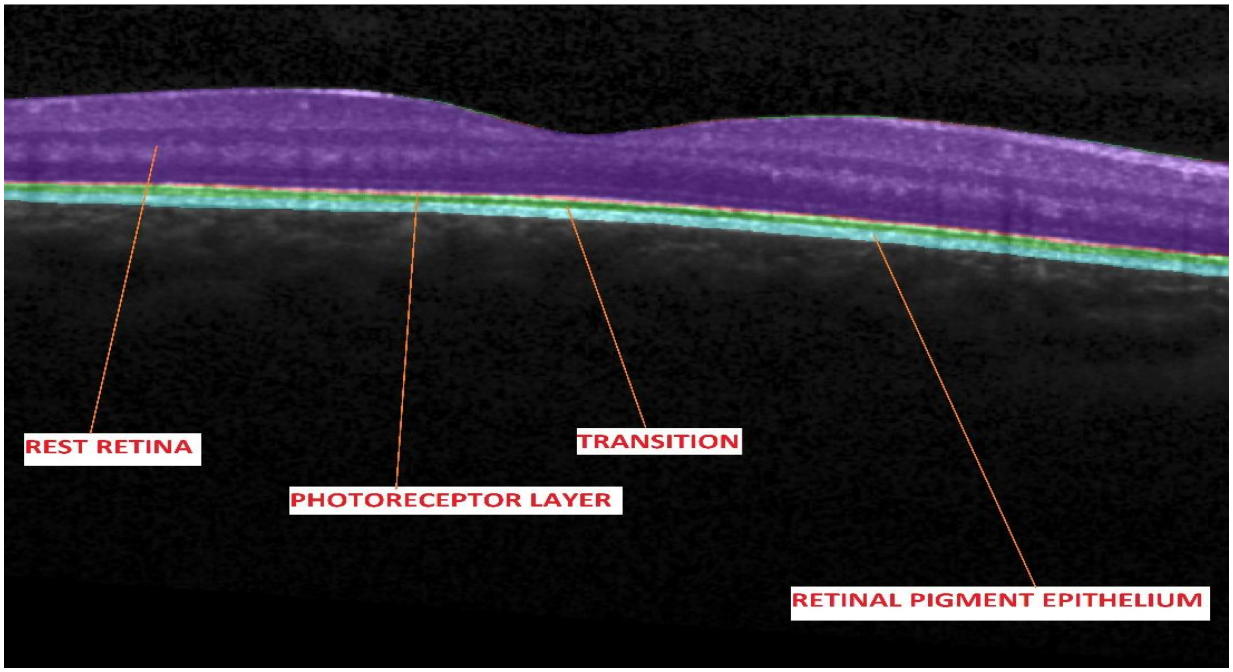

Figure 3. Segmented Layers

The retinal layers segmented in this project will be the retinal pigment epithelium, the photoreceptor layer, a transition between them and what has been named as 'Rest Retina'.

Optical Coherence Tomography

Optical Coherence Tomography (OCT) is an imaging technique which uses low-coherence light to capture micrometer-resolution used for medical imaging and industrial nondestructive testing. It typically employs near-infrared light. This relatively long wavelength allows it to penetrate the retina in our case.

In this case, Spectralis OCT Software from Heidelberg Company is used. Once a scan is performed, this software creates a .vol file containing:

- SLO Image
- OCT Scans
- Other useful parameters, such as pixel-mm ratios or layers thickness.

Some of these parameters will be used later to obtain layers thickness and compare the real ones with the predicted, given a quick overview of how accurate the model is.

Should the reader be interested, further information can be found in this paper Optical coherence tomography: fundamental principles, instrumental designs and biomedical applications.

Google Colab. GPU and Multiprocessing

Google Colab is the Python environment that has been used in the project. The reason to use this platform and not others such as Spyder or Jupyter Notebook is because Colab allows their users to have access to powerful GPUs that enhance the code speed and saves enormous amounts of time while training (hours). In addition, it is very simple to move variables and models to GPU memory, using the 'cuda' device function, according to PyTorch Documentation. Further information can be found in the next link Cuda Semantics from PyTorch.

In addition, Colab allows to easily share content, for example, with Google Drive.

Another option to boost the code speed is Multiprocessing. Multiprocessing consists in using multiple CPU Cores or Threads (depending on the configuration), allowing to perform several

calculations at the same time. PyTorch provides a library called *torch.multiprocessing*, which is a replacement from the previous Python's multiprocessing module.

In specific, this work implements the Hogwild strategy for model training, using multiprocessing as a tool to enhance code speed. Nonetheless, results obtained from this implementation showed a not despicable acceleration but too far from the GPU results. All acceleration processes had approximately the same metrics. Further information on Hogwild strategy can be found in the next article Hogwild! A lock free approach to parallelizing stochastic gradient descent. Time results for epoch training a Resnet_50 model are shown in the next table:

RESNET 50 MODEL. TRAINING TIME PER EPOCH		
Without Multiprocessing nor GPU	Multiprocessing (6 cores)	GPU
Around 7 hours	From 3 to 4 hours	Up to 15 minutes

Table 1: Time results from training 1 epoch with Resnet 50 using different acceleration processes

Data Preparation

The first step before starting to create the convolutional neural network model is to prepare the available data, which will include several processes such as normalization, resize or one-hot encoding. In addition, masks used for this project had certain problems when joined together.

One-hot Encoding

In this project two different segmentations were performed. The first one included differentiating the whole retina from the background. In the second one, the different layers explained before were segmented.

The first step to take was to create a unique mask which included the different layers required with different pixel values depending on the layer. However, what the dataset included was a mask of each retinal layer. Therefore, it was necessary to join all these images and then, pass them to the model.

To pass the images, a method called *One-Hot Encoding* was applied. This method is quite simple and the reason to use it is the following. When a pixel value is assigned to each layer, what is being applied is *Label Encoding*, which consists in assigning, for instance, 1,2,3 and 4 values if there are 4 categories. The problem lies here. If our code performs, for example, the average of first and third class (which is 2) it will be assigned to the second class, which will be nonsense.

To solve that, label values are converted to binary values, as shown in the table below.

LABEL VALUE	ONE-HOT VALUE
0	[1,0,0,0,0]
1	[0,1,0,0,0]
2	[0,0,1,0,0]
3	[0,0,0,1,0]
4	[0,0,0,0,1]

Table 2. One hot pixel values

The piece of code used to perform One-Hot encoding is the following one:

```
OH = (np.arange(5) == image[...,None]).astype(np.float32)
```

Figure 4. One Hot Code

As it can be observed, it is quite simple.

Normalization

Imagine there were two features in the images used, each one having a different range of values (for instance, one has an order of 10 and the other one 100). The different order would lead to later mistakes, as the largest variables will have a greater weight when performing the training, leading to worse metrics. The purpose of normalization is to avoid this problem transforming the features to have similar distributions and numerical orders.

In this project, two normalizations were performed. The first one, in the dataset, setting all the pixel values to a number between 0 and 1. Though input images were already normalized (thus no initial normalization was needed) it is important to understand the main techniques that exist nowadays.

The first one is called 'rescaling' or 'min-max normalization'. It is the simplest method and calculated as:

$$x'(i) = \frac{x(i) - min(x)}{max(x) - min(x)}$$

Another widely used method is called Z-score normalization or standardization. It basically converts to a standard normal distribution. It is calculated as follows:

$$x'(i) = \frac{x(i) - \mu}{\sigma}$$

Where μ is the population mean and σ is the standard deviation.

The second normalization performed is called batch normalization and is applied adding a layer to the model architecture, which is very simple in PyTorch, using BatchNorm2d:

```
def conv3x3_bn(ci,co):
  return torch.nn.Sequential(
      torch.nn.Conv2d(ci,co,3,padding=1),
      torch.nn.BatchNorm2d(co),
      torch.nn.ReLU(inplace=True)
  )
```

Figure 5. BatchNorm Method in PyTorch

What this method does is normalize the outputs after each layer, avoiding an overweight in the next layers. Next image shows the location of the BatchNorm layer in a general model, indicated with the sentence ‘Normalize the outputs’.

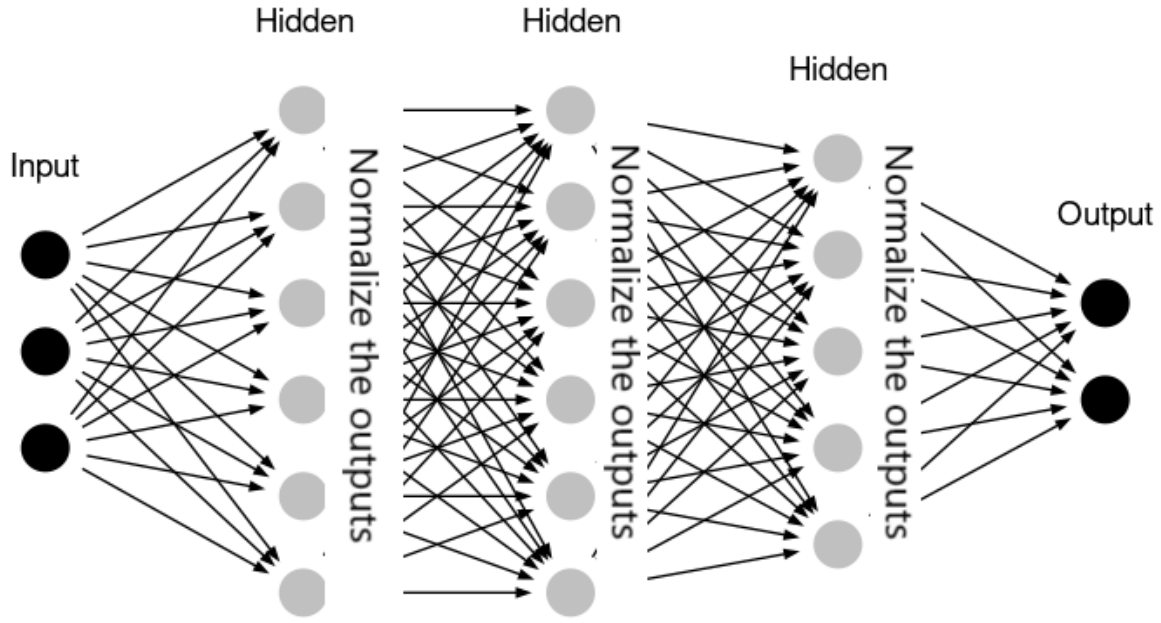

Figure 6. Batch Normalization Layer

Avoid dataset superposition

While joining different layer masks in order to create a unified one with different colours for each layer, a problem happened. Retinal pigment epithelium layer values did not match with their corresponding position in whole retina mask. This problem was detected when retinal pigment epithelium image was subtracted from whole retina mask, obtaining next result:

Figure 6.1. Superposition problem in Dataset

A lot of -1 pixel values were obtained, due to several white pixels belonging to retinal pigment epithelium and not being in whole retina mask. To fix it, the next solution was applied. A loop by columns was created, from top to bottom of the image. If previous row pixel value was white, and actual pixel value was black, all next rows in column were coloured in black.

Resizing

After performing the data normalization, another problem arises. Not all the images in the dataset have the same size. Nevertheless, the model must be trained with a specific size. If not, after applying a kernel to the image, the output will be different and there will be a wrong match between the weights obtained in the model and the size of the input image.

There is no denying the fact that resizing an image makes it lose some quality. This problem occurred during the project, as shown in the images below:

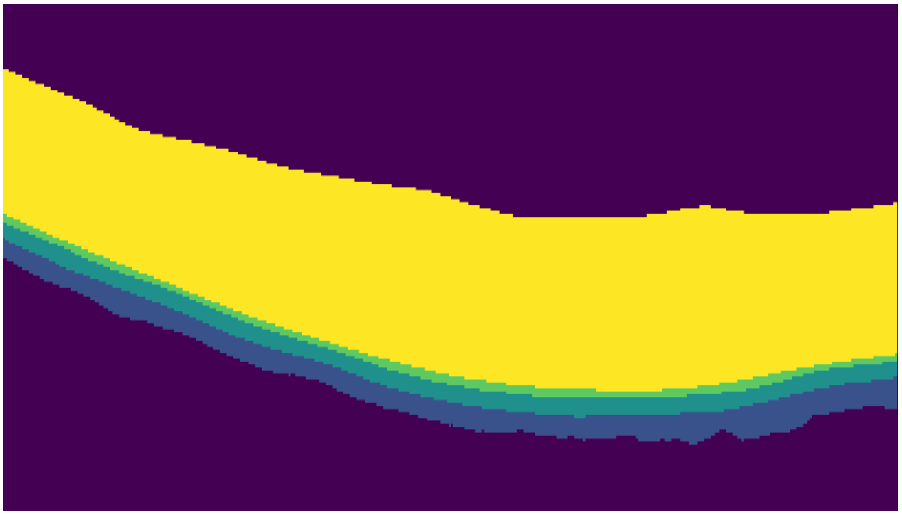

Figure 7. Image after applying One-Hot Encoding, Original Size.

Figure 8. After Resize. Several pixels have been altered

To solve this problem, the next solution was applied. If after the resize method, there was any pixel different from the ones established after the One-Hot encoding, it was changed to the next row pixel value. At the same time, if the image is analysed by columns, a pattern will be extracted. When a new class starts, the first-class pixel has a different colour from the previous one but not from the next, which will be of the same class. When a class ends, the next pixel is different but

the previous one should be the same. This eliminates the option of having different previous and next pixel values. Therefore, with an 'if' condition this invalid situation is searched and corrected.

```
try:
  if (image[j,i]!=1) and (image[j,i]!=2) and (image[j,i]!=3) and (image[j,i]!=4) and (image[j,i]!=0):
    image[j,i] = image[j+1,i]
  if (image[j,i]!=image[j-1,i]) and ((image[j,i]!=image[j+1,i])):
    image[j,i] = image[j+1,i]
except:
```

Figure 9. Loop to solve resize problem

Results obtained were rather satisfactory, as shown below:

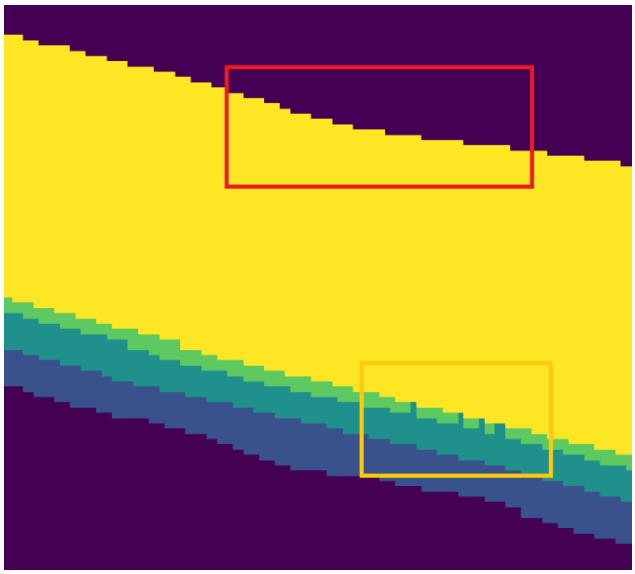

Figure 10. Results after increasing quality

The red rectangle shows the problem was solved in the upper part of the picture. However, several pixels were classified wrong (yellow rectangle) though their values were changed to the possible range (0 to 4) set up in One-Hot Encoding.

Dataset, dataloader and Data Augmentation

After applying all these transformations, a correct DataSet will be obtained to enter the model architecture. Nevertheless, before that the different images must be precharged in Batches, which are simply groups of images. This action can be performed with PyTorch DataSet and Dataloader classes, improving data entering the model.

DataSet Class

As said before, PyTorch implements an abstract class that represents a DataSet, located in *torch.utils.data.Dataset*. This class is implemented to make it easy for the DataLoader class to access the images and masks of the original dataset. Basically, DataSet class includes three methods, which are *__init__*, *__len__ and __getitem__* (further information can be found in torch.utils.data.Dataset PyTorch documentation). In this project, two Dataset classes were implemented: the first one for training data and the second one for testing data.

DataLoader Class

This class allows taking Dataset *__getitem__* method outputs in batches. As mentioned before, a batch is simply a group of images and their masks. Depending on the batch size, the computer can run out of RAM memory, so it is a crucial factor to take into consideration.

Data Augmentation

An interesting tool is called Data Augmentation, which consists in applying slight changes to the original images in order to create a new image. These transformations include image rotation, changing brightness… In this project, Data Augmentation was applied when performing whole retina segmentation. However, there was not an improvement in the results obtained. Library used to implement this tool was Albumentations.

Metrics

Metrics are the parameters that are used to evaluate how good the model is. Though a lot of different metrics exist, in this project only the most popular ones have been used, which are loss, accuracy, confusion matrix and Intersection over Union (IoU). Though most of the metrics used have an existing implementation in libraries such as Scipy, accuracy and IoU were implemented by the author.

Loss

Loss functions are used to determine the error between the outputs of algorithms and the given target value (which in this case are masks). In this project *torch.nn.BCEWithLogitsLoss* is used, as it is a widely used loss function for semantic segmentation and reported good results. Further information can be found in A survey of loss functions for semantic segmentation . Its equation is described below:

$$l = mean\{l_1, l_2 \dots l_n\}, \qquad l_n = -w_n[y_n * log\sigma(x_n) + (1 - y_n) * log(1 - \sigma(x_n))]$$

This loss combines a Sigmoid layer and the BCELoss in one single class. Further information can be found in PyTorch BCEWithLogitsLoss Documentation.

Accuracy

It might be the simplest metric used. Its equation is described below:

$$Acc = \frac{Number\ of\ correct\ predictions}{Total\ number\ of\ predictions\ made}$$

Nevertheless, accuracy is not always the best metric possible. In this case, the number of background pixels was much bigger than the ones corresponding to retinal layers. This situation yielded incredibly high accuracy metrics because background pixels were much easier to predict. Therefore, other metrics should be taken into account, such as intersection over union.

Intersection over Union (IoU)

Though this metric is widely used in object detection, it reported good results in this project also. Our IoU implementation basically detects the true positives (TP) that the One-hot encoding mask and the output have after transforming them into binary numbers (applying an ‘if’ condition) and divides them by true positives plus false positives and false negatives (TP+FP+FN).

Usually, this metric is represented as in the image below:

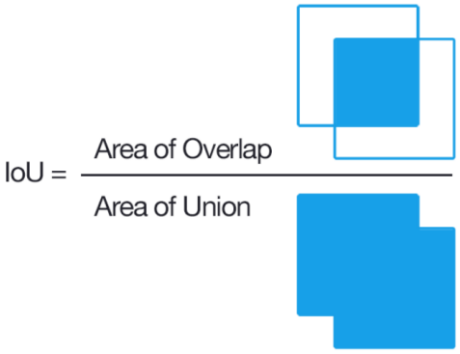

Figure 11. IoU

Confusion Matrix

Confusion Matrix is the most intuitive and explanatory metric used. It creates a matrix with N*N cells (N is the number of classes) showing exactly each class behaviour and which are the classes it is most of the times confused with. Confusion Matrix was imported using *torchmetrics* library. The image below illustrates one of the confusion matrix results obtained during training:

Classes	0	1	2	3	4
0	1.3278e+08	9.3584e+04	1.4e+01	1e+0	9.2013e+04
1	7.3148e+04	2.3533e+06	7.2476e+04	7.5e+01	0
2	6e+0	6.9057e+04	1.8458e+06	5.7779e+04	1.735e+03
3	4.1e+01	1.1e+02	1.3499e+05	6.9688e+05	8.8389e+04
4	1.0001e+05	0	2.4500e+02	7.1034e+04	1.8839e+07

Table 3. Confusion matrix obtained during a training

In the table above some of the cells were highlighted in dark blue. These cells correspond to the right predictions (original class matches predicted class). As it can be noticed, these values are the biggest ones, which is a good sign of the good work of the artificial intelligence model created. Class 3 was the one that reported worst results, which is due to resizing images and losing quality.

Architectures

In this project development, 5 different architectures were used: UNet, Resnet50, Resnet101, Deeplabv3_resnet50 and Deeplabv3_resnet101. UNet was implemented manually making use of *torch.nn.functional* class. The rest of them were actually imported from *torchvision.models.segmentation*. Nevertheless, few changes were required to do in order to make the model work. They involved changing the predefined number of classes (which was easy to do just accessing the last layer) and adapting the input image to accept grayscale images like the ones that were being used.

First, there is a need to understand one of the basic calculations of machine learning: convolution. A convolution is the process of extracting features from input data using kernels. A kernel (or filter) is a matrix that is moved along the input, multiplying each value of the kernel by its match in the input. After that, results are summed creating a feature map. The convolution process is shown in the image below:

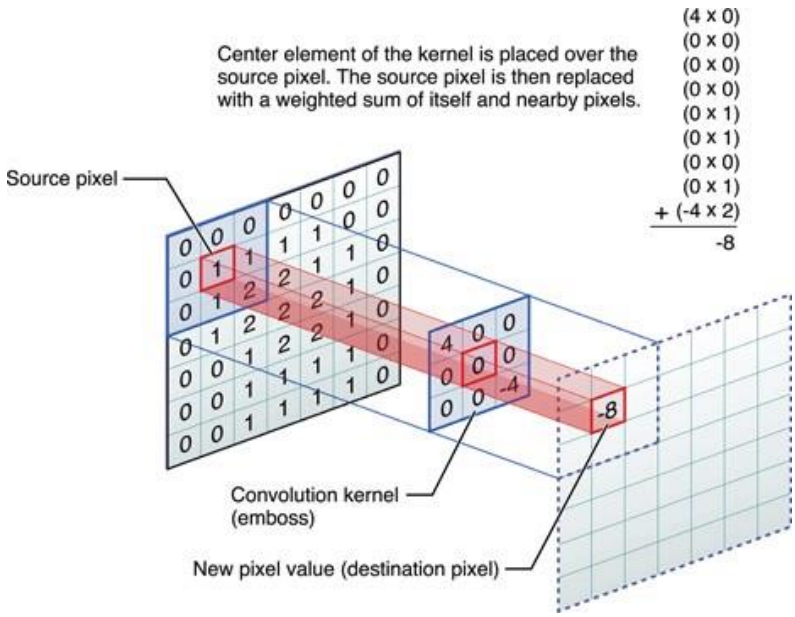

Figure 12. Convolution applying a 3x3 kernel

This process allows extracting image features as explained in the image below:

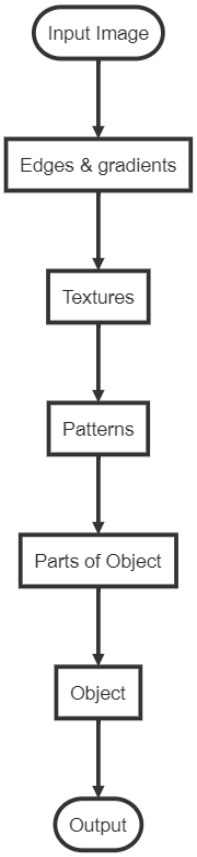

Figure 13. Features extraction process

In the next paragraphs, each architecture structure will be explained.

UNet

It is a U-shaped encoder-decoder network architecture, which consists of four encoder blocks and four decoder blocks connected via a bridge. The encoder network (which is the contracting path) halves the spatial dimensions and doubles the number of filters at each encoder block.

Each convolution is followed by a ReLU (Rectified Linear Unit) activation function, which is necessary to introduce non-linearity into the network, helping to get better results. ReLU equation is described below. It basically transforms negative results to 0. If results are positive, it does nothing.

$$f(x) = \max(0, x)$$

Max Pooling layers reduce spatial dimensions of the filters (feature maps) reducing computational cost.

Skip Connections (*copy and crop* in the image) act as a shortcut connection that helps in the direct flow of gradients to the earlier layers, improving backpropagation.

Further information can be found in [U-Net: Convolutional Networks for Biomedical Image Segmentation] by Olaf Ronneberger et al.

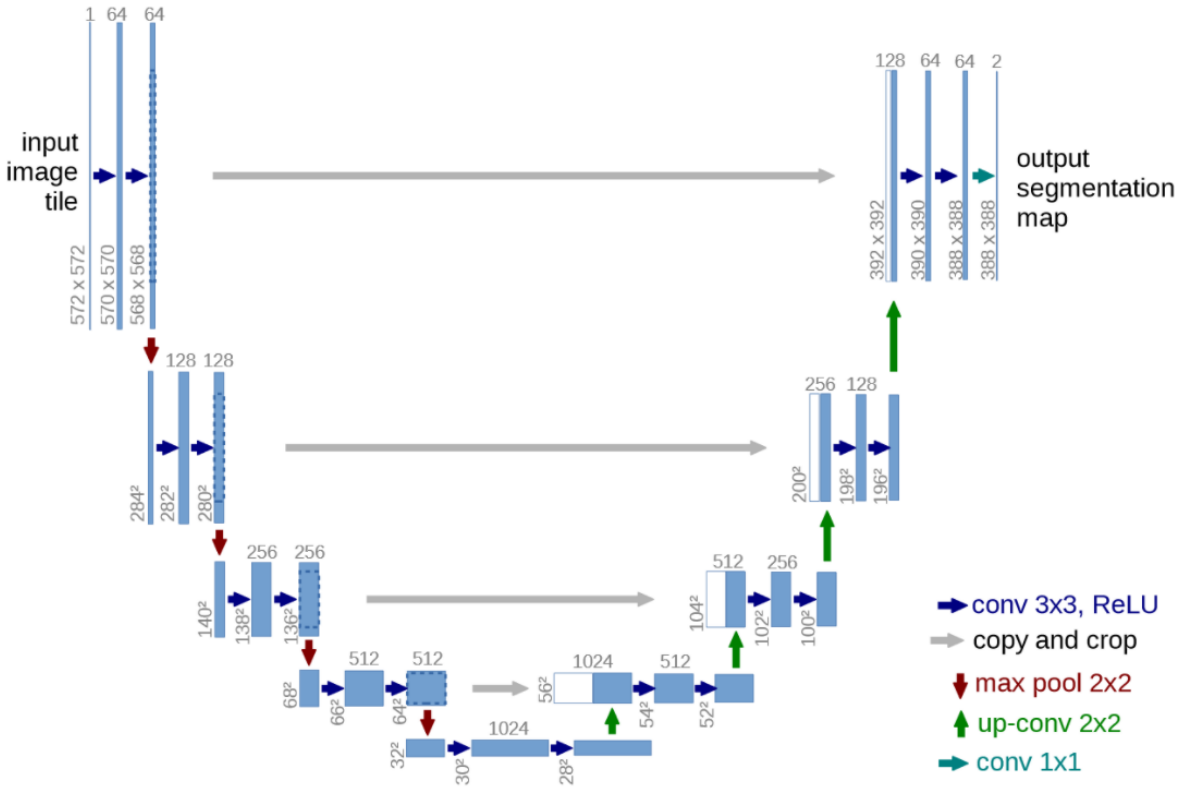

Figure 14. UNet architecture

Resnet 50 and Resnet 101

Residual Networks (ResNets) are inspired by biological neurons, which are not only connected to the contiguous ones but also to further neurons. Their major impact was allowing to train really deep neural networks (more than 100 layers) for the first time, avoiding the vanishing gradient problem.

Below are two images. The first one shows the basic connection in ResNet architectures (ResNet block), where each convolution output is not only connected with the contiguous convolution.The second one shows the architecture of a ResNet-34 model. To build a ResNet-50 model from ResNet-34, each 2-layer ResNet block is replaced by a 3-layer ResNet Block. To build the ResNet-101 model, more 3-layer ResNet Blocks are added.

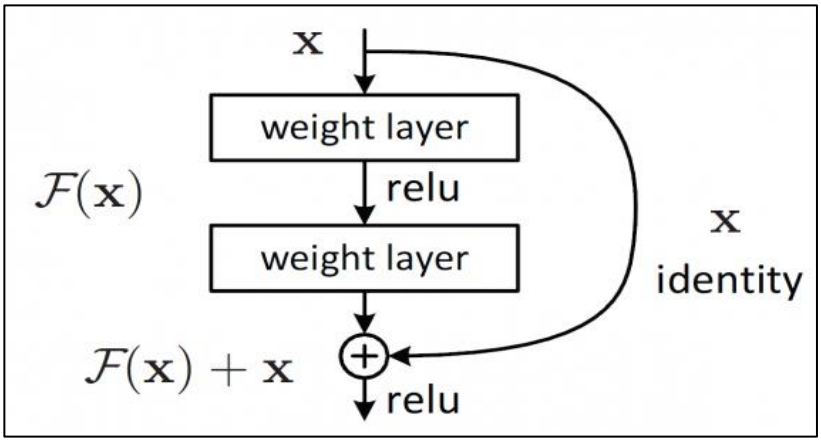

Figure 15. 2 layer ResNet Block

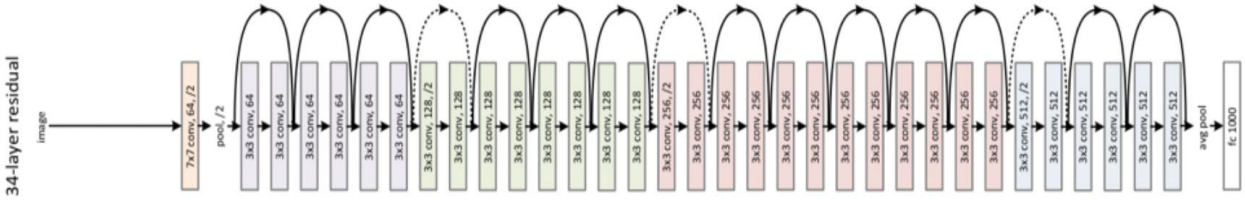

Figure 16. Resnet 34 architecture

Further information can be found in the next paper [Deep Residual Learning for Image Recognition] by Kaiming He et al.

Deeplabv3 resnet-50 and Deeplabv3 resnet-101

These two CNN architectures are based on *atrous convolution*, which is a French term that stands for *convolution with holes*. What identifies this convolution is each kernel pixel field of view. Whereas in a standard convolution there is no gap between original pixels, an atrous convolution has what is called 'dilation rate', which is the number of gaps that exist between two real pixels. This way, a kernel of size 3x3 with a dilation rate of 2 has a field of view of 5x5 pixels.

Dilated or atrous convolutions can be used where the object size is almost equal to the size of the image, when learning a few features of the object will give the right prediction. Hence there is no need to learn every feature to make predictions, leading to faster results and a reduction of computational cost.

Though retina size in OCT images does not have a similar size to the whole image, the results obtained were acceptable. This was the reason to include these architectures in the project.

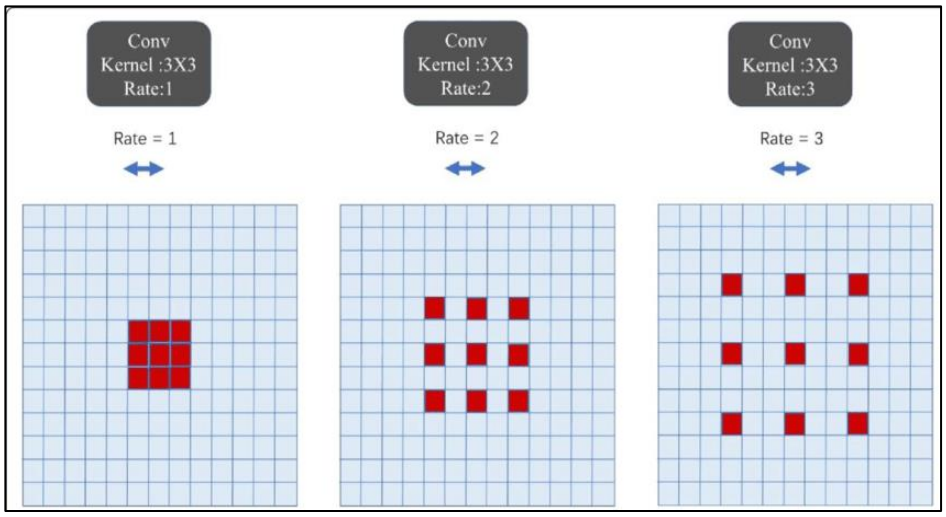

Figure 17. Atrous convolution

$$y[i] = \sum_{k} x[i + r \cdot k] w[k]$$

Figure 18. Atrous convolution equation

For each location *i* in the output *y* and a filter *w*, atrous convolution is applied over the input feature map *x* with an atrous rate r.

The image below shows how deeplabv3 architecture works in comparison without atrous convolution. Atrous convolution applied from block 3 allows minimizing computational cost, allowing to have a larger field-of-view without increasing the number of parameters or the amount of computation.

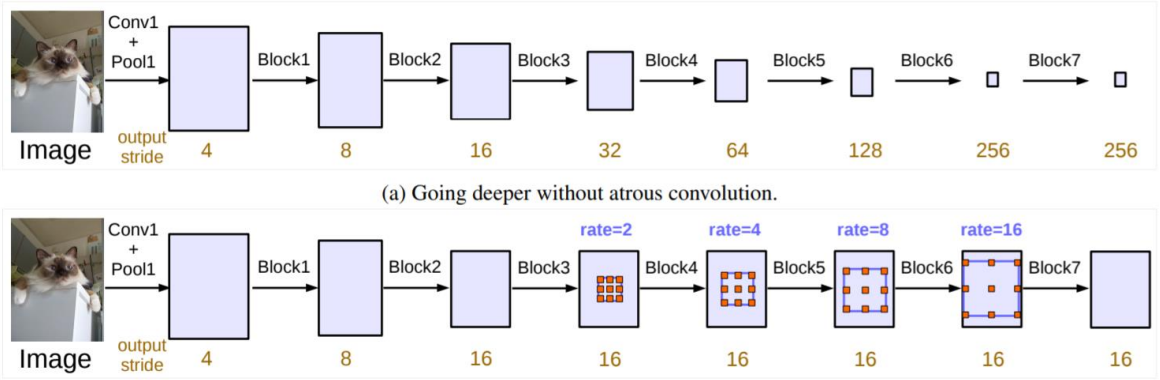

Figure 19. Atrous convolution vs. Convolution+MaxPooling

For further information, visit Rethinking Atrous Convolution for Semantic Image Segmentation, by Liang-Chieh Chen et al.

Model training and evaluation

In this section, the code developed will be explained in detail. First of all, it is good to understand how the training function has been structured.

The first step is to know which hyperparameters the user wants to select. Hyperparameters are external inputs selected that the model will use to train and evaluate the model to make predictions. The hyperparameters are the model architecture, learning rate and epochs. In addition, a regularization method called 'early stopping' was implemented. 'Early stopping' makes the training stop when results do not improve after a selected number of epochs.

In order to ask the user for these hyperparameters, a form was implemented in Google Colab. This platform includes an easy way to create forms, transforming our code into a more manageable one.

Next image shows the result:

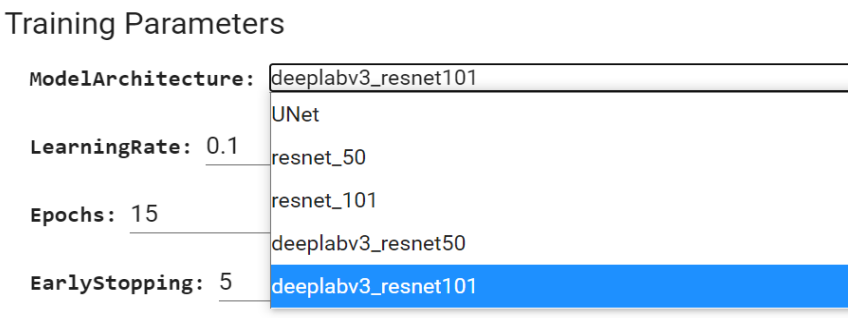

Figure 20. Hyperparameters form

Training and test

After that, the training function is defined. Two different functions were defined depending on whether GPU was available or not. If not, multiprocessing button should be turned on in order to select the corresponding function. Next flow diagram shows how training and test function is defined.

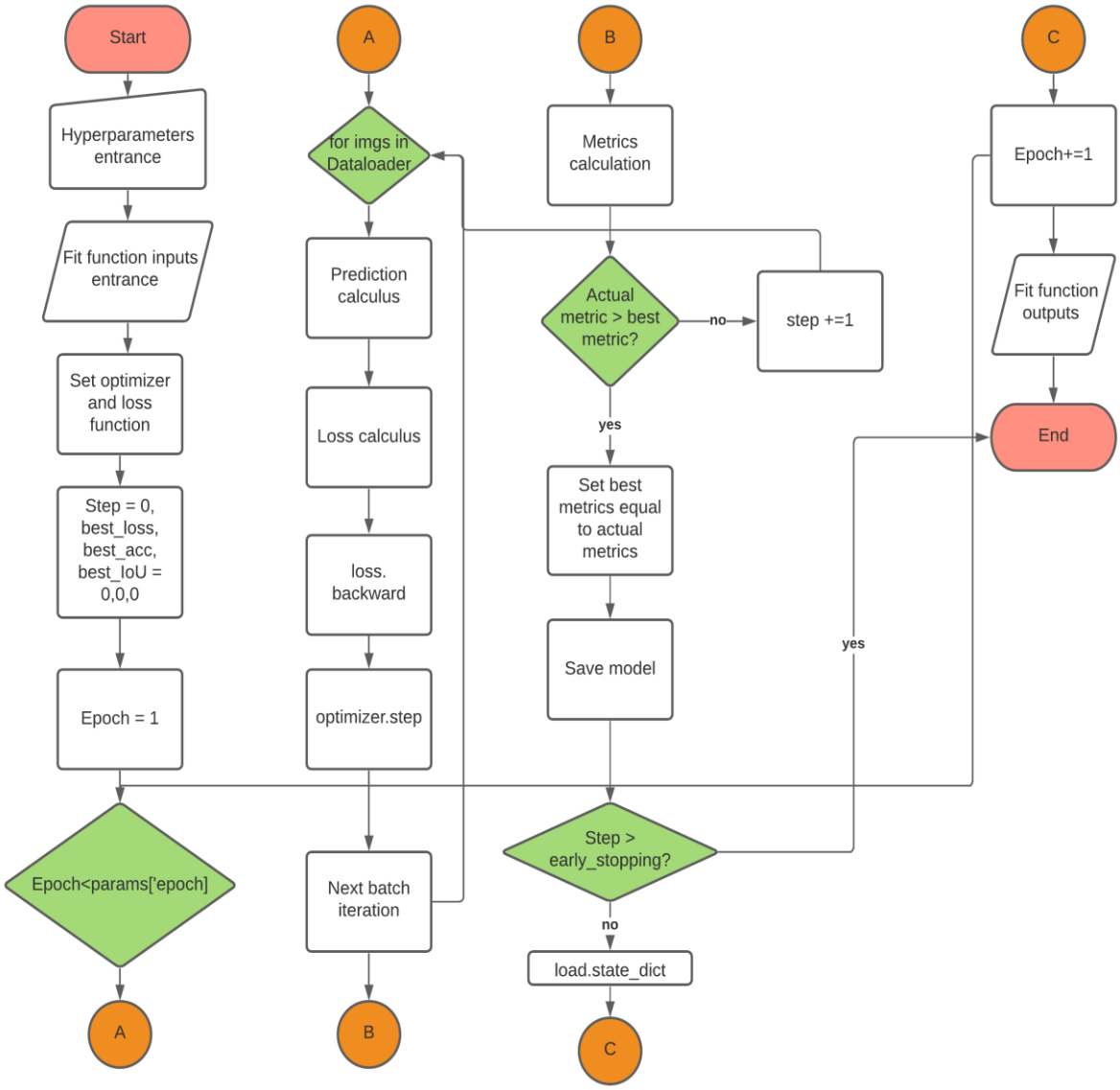

Figure 21. Fit function flow diagram

First step is to set up all inputs for the training function, which are passed via the form mentioned before. After that, loss function and an optimizer are defined.

An optimizer is a math function that allows calculating a minimum or maximum. In this case, the loss is minimized.

Gradient Descent could be called 'the father of all optimizers', as most of them are based on the same idea. This idea consists in calculating the slope of the function and applying the next formula in each step (this action will be performed in *optimizer.step*).

$$New\ weight\ =\ Old\ weight\ -\ learning\ rate\ x\ \frac{\partial(loss)}{\partial\ (old\ weight)}$$

As it can be inferred from the formula, tiny learning rates will result in tiny steps. This last action takes more time than a smaller learning rate but there is more likelihood of finding the exact minimum value. If learning rate is too big, a cyclic state could start when reaching values near minimum, without minimizing correctly the loss function.

This situation is reflected in next images:

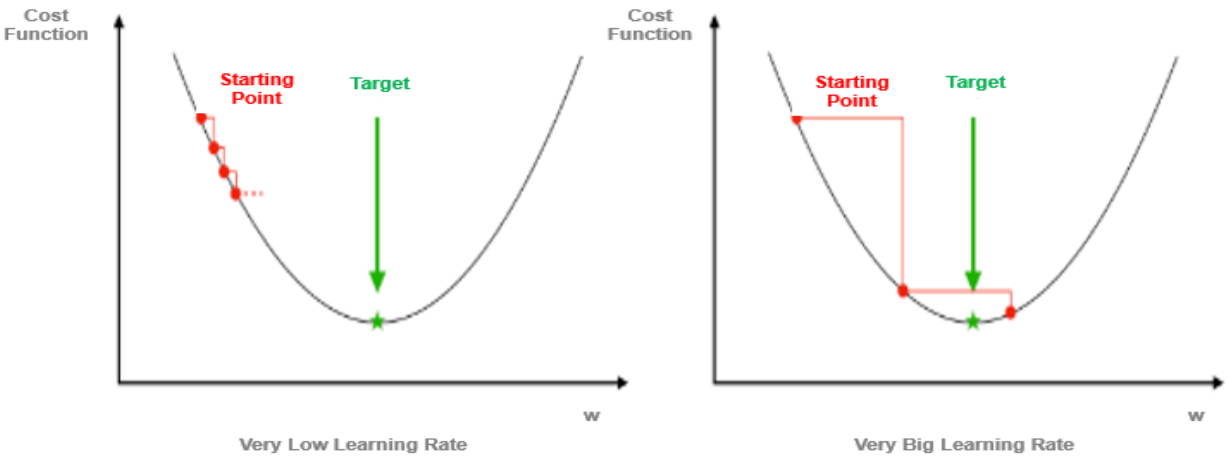

Figure 22. Effect of learning rate in minimizing loss function

The optimizer applied in this project was Adam optimizer, as documentation showed it reported better results than other existing optimizers. Further information about this optimizer can be found in Adam: A method for stochastic optimization by Diederik P. Kingma et al.

Gradients are calculated this way. The first step is to compute the gradient of vector loss function, which is saved in a Jacobian matrix:

$$\mathbf{J_l} = \begin{bmatrix} \frac{\partial l_1}{\partial h_1} & \frac{\partial l_2}{\partial h_1} & \cdots & \frac{\partial l_m}{\partial h_1} \\ \frac{\partial l_1}{\partial h_2} & \frac{\partial l_2}{\partial h_2} & \cdots & \frac{\partial l_m}{\partial h_2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial l_1}{\partial h_n} & \frac{\partial l_2}{\partial h_n} & \cdots & \frac{\partial l_m}{\partial h_n} \end{bmatrix}$$

Where the loss gradient is calculated for each layer. After that, a loss gradient vector is obtained:

$$\frac{\partial l}{\partial h_i} = \frac{\partial l_1}{\partial h_i} + \frac{\partial l_2}{\partial h_i} + \cdots + \frac{\partial l_m}{\partial h_i}$$

This is called the Jacobian vector product. This is the vector that enters as an argument to the backward function.

The next step involves calculating all gradients along layers in order to compute then their corresponding weights making use of Adam optimizer. To do that, the following calculations are done:

Loss function depends on the output values (in this case, predicted images). If the output is called 'y', there is a dependency between loss and 'y', so loss(y). In addition, convolutional neural networks obtain outputs (y) from inputs(x), which can be described in a simple way as:

$$y = \sum_i w_i * x_i$$

So, applying chain rule $\frac{\partial loss}{\partial w} = \frac{\partial loss}{\partial y} \, x \, \frac{\partial y}{\partial w}$ results can be 'backpropagated', which is the function of *loss.backpropagation*.

One of the main problems that appeared during training model was where the model was saved. Depending on whether GPU is being used or not, device parameter is set as 'cuda' or not. Therefore, model will be saved in GPU memory and if any calculations have to be performed after that, it is good to save model in CPU memory. As mentioned before, GPU allows accelerating training process.

In addition, if batch size (which is set in dataloader class) is too big, GPU might run out of memory. In order to solve this problem, batch size needs to be reduced, but *torch.cuda.empty_cache()* function can complement this action also.

As three different metrics were used, several options could have been implemented to define when new trained model was better than the previous one. For instance, if IoU is higher in the new model than the previous one, this new model can be saved as best model. However, the author opted for an option where all the different metrics were included. As loss is needed to be as smallest as possible and IoU and accuracy the highest, loss was inverted and multiply by a specific weight, yielding a result of the same order as IoU and accuracy metrics. After that, all metrics were summed obtaining what was called *actual_metric* variable.

Though this option was accurate in segmenting the whole retina, when coming to layers segmentation, loss function was rather smaller, so its inverse was bigger, yielding to loss metric ruling above IoU and accuracy. Another idea could be to create a dynamic weight that changes according to loss value, though it was not implemented.

Evaluation

After performing training and test, three different metrics are obtained: IoU, accuracy and loss. Confusion matrix is also obtained. However, as general concepts about this metric were already explained, the author does not consider it necessary to include more information about it.

Whole retina segmentation

Next table shows different metrics obtained in the whole retina segmentation:

Model:UNet, BatchSize:12, Learning rate:0.1, Epochs:50, Early Stopping:10, Train Dataset Length:800, Data Augmentation:True	
Test Loss	0.0188
Test IoU	0.9826
Test Accuracy	0.8251
Model:UNet, BatchSize:12, Learning rate:0.1, Epochs:50, Early Stopping:10, Train Dataset Length:800, Data Augmentation:False	
Test Loss	0.0198

Test IoU	0.9803
Test Accuracy	0.8251
Model:resnet_50, BatchSize:12, Learning rate:0.1, Epochs:50, Early Stopping:10, Train Dataset Length:800, Data Augmentation:False	
Test Loss	0.0127
Test IoU	0.9860
Test Accuracy	0.8256
Model:resnet_50, BatchSize:12, Learning rate:0.1, Epochs:50, Early Stopping:10, Train Dataset Length:800, Data Augmentation:True	
Test Loss	0.0183
Test IoU	0.9800
Test Accuracy	0.8547
Model:resnet_101, BatchSize:6, Learning rate:0.1, Epochs:50, Early Stopping:10, Train Dataset Length:800, Data Augmentation:True	
Test Loss	0.0154
Test IoU	0.9844
Test Accuracy	0.8449
Model:resnet_101, BatchSize:6, Learning rate:0.1, Epochs:50, Early Stopping:10, Train Dataset Length:800, Data Augmentation:False	
Test Loss	0.0097
Test IoU	0.9868
Test Accuracy	0.8740
Model:deep_lab_v3_resnet50, BatchSize:8, Learning rate:0.1, Epochs:15, Early Stopping:5, Train Dataset Length:800, Data Augmentation:False	
Test Loss	0.0385
Test IoU	0.9682
Test Accuracy	0.8572

Table 4. Metrics obtained in whole retina segmentation.

Also graphs showing metrics evolution during training were obtained. Next image shows an example of them:

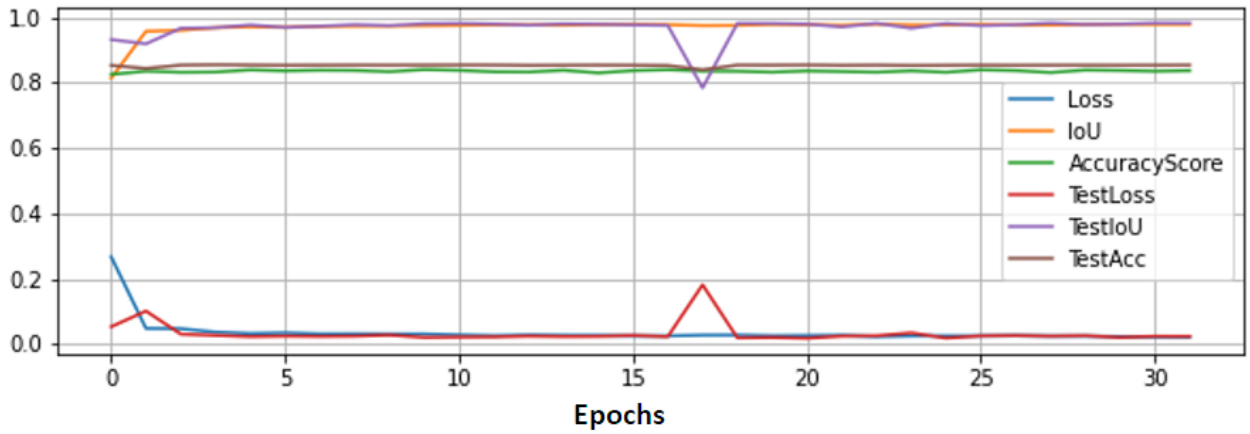

Figure 23. Metrics evolution whole retina segmentation for resnet 50 with learning rate 0.1 model. Data Augmentation False, Batch Size 12, Epochs 50, Early Stopping 10, Train Dataset Length 800.

Highest values are achieved in a few epochs. Nevertheless, results continue to improve until early stopping breaks code. All models obtained have similar graphs.

Retinal layers segmentation

Next table shows different metrics obtained in layers segmentation:

Model: resnet_50, BatchSize:4, Learning rate:0.1, Epochs:15, Early Stopping: 5, Train Dataset Length:600, Data Augmentation:False	
Test Loss	0.0084
Test IoU	0.7745
Test Accuracy	0.9927
Model: resnet_50, BatchSize:8, Learning rate:0.01, Epochs:15, Early Stopping:5, Train Dataset Length:600, Data Augmentation:False	
Test Loss	0.0074
Test IoU	0.8277
Test Accuracy	0.9940
Model: resnet_50, BatchSize:8, Learning rate:0.001, Epochs:15, Early Stopping:5, Train Dataset Length:600, Data Augmentation:False	
Test Loss	0.0068
Test IoU	0.8307
Test Accuracy	0.9949
Model: resnet_101, BatchSize:4, Learning rate:0.1, Epochs:15, Early Stopping:5, Train Dataset Length:600, Data Augmentation:False	
Test Loss	0.0072
Test IoU	0.8083
Test Accuracy	0.9942
Model: resnet_101, BatchSize:4, Learning rate:0.01, Epochs:15, Early Stopping:5, Train Dataset Length:600, Data Augmentation:False	
Test Loss	0.0064
Test IoU	0.8015
Test Accuracy	0.9945

Model: resnet_101, BatchSize:3, Learning rate:0.001, Epochs:15, Early Stopping:5, Train Dataset Length:600, Data Augmentation:False	
Test Loss	0.0067
Test IoU	0.8320
Test Accuracy	0.9949
Model: deeplabv3_resnet101, BatchSize:4, Learning rate:0.1, Epochs:15, Early Stopping:5, Train Dataset Length:600, Data Augmentation:False	
Test Loss	0.0092
Test IoU	0.7716
Test Accuracy	0.9924

Table 5. Metrics obtained in whole retina segmentation.

Also graphs showing metrics evolution during training were obtained. Next image shows an example of them:

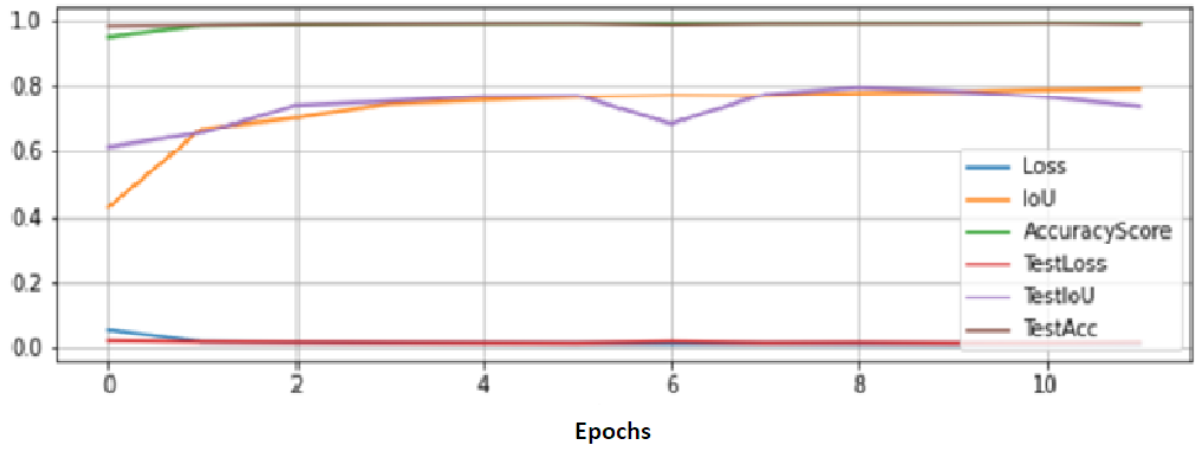

Figure 24. Deeplabv3_resnet101 with learning rate 0.1

In these multiple layer segmentation models, curver graphs are obtained, which shows that it takes more epochs to reach the best metrics possible.

Saving models in Google Drive

After training the model, it is quite simple to upload both graphs and model parameters in Drive. In addition, model metrics are added to a text file called *Results.txt.* This way, it is immediate to obtain all the results and improve its interconnectivity within Drive and Google Colab.

Evaluation

Apart from obtaining all metrics and graphs, it is absolutely necessary to see the predicted images. To do that, a .zip folder with all predicted images from test dataset is saved.

This process entailed some problems. For example, masks were built on 0,1,2,3,4 pixel values. After saving them as a .png image, images were totally black, as image reader was set up for pixel values between 0 and 255. To solve this problem, first 5 different colours were chosen to apply to each layer. Then, the following flow diagram was coded to build the *TORGB* function, whose aim is to convert masks to readable values between 0 and 255 in three channels.

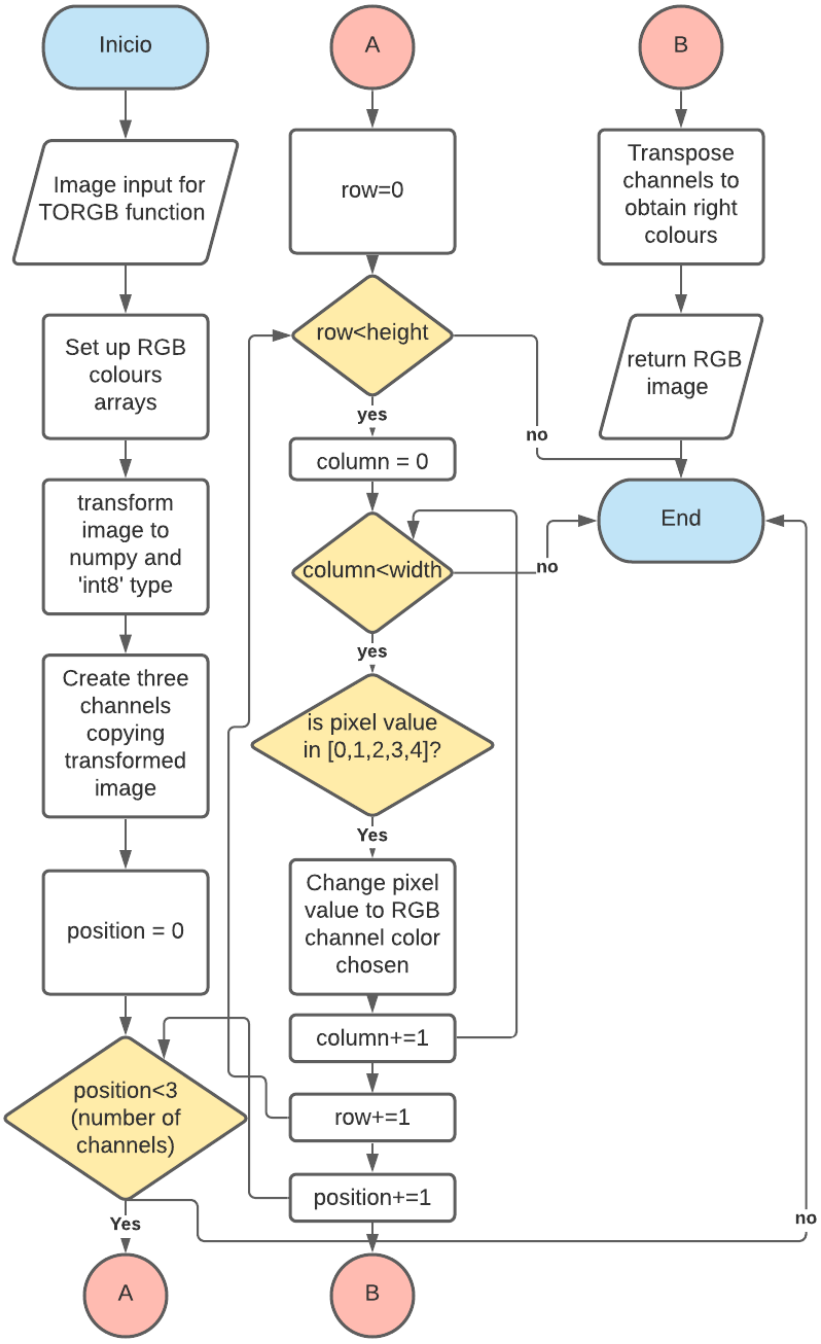

Figure 25. TORGB function flow diagram

Finally, predicted images from test dataset, masks after applying *TORGB* function and original OCTs are saved in Google Drive Paths.

To save all Drive changes, *drive.flush_and_unmount* function is used.

Results

Whole retina segmentation

Next images show results obtained after whole retina segmentation:

Figure 26. Predicted images after whole mask segmentation

As it can be perceived, predicted masks are quite accurate, even in some cases avoiding original masks failures, such as in the circled ones, which is successful.

Retinal layers segmentation

Next images show results obtained in layers segmentation

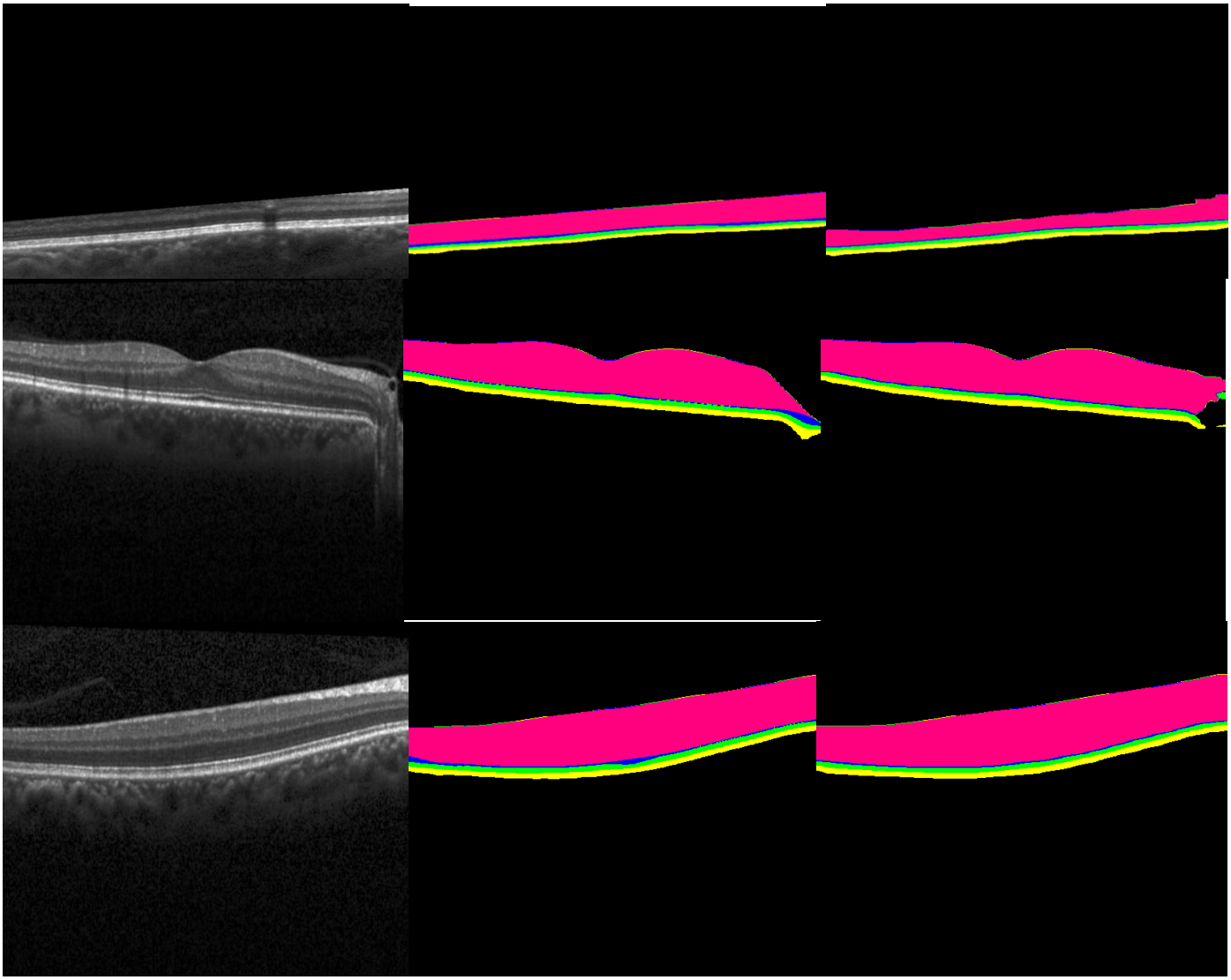

Figure 27. Several sets of original OCT Scan (0), original mask (1) and predicted mask (2) in this order.

As it can be perceived results are accurate though several mistakes can be seen at first sight, though these mistakes relate to OCTs different from the most typical ones.

In addition, if the reader looks at the next picture, some mistakes could be perceived in the first pixel line. This is due to resizing images to their original shapes after predicting them, which is a similar problem to the one explained in previous chapter *Data preparation*. These mistakes are not in fact provoked by training. Thus, model is making good predictions.

Figure 28. OCT Prediction in Layer Segmentation. First line of pixels is not correct due to resizing to original shapes

Obtained retinal thickness for 3D graphics

First step: Extracting data from .vol file

As an additional resource apart from image visualization, it is good to obtain a 3D graph of the results obtained and a heat map.

To perform this task, binary information exported by Spectralis Software in a .vol file needs to be treated. In Spectralis documentation, different parameters obtained by software are explained, including its corresponding sizes and types in binary information. Therefore, these bites can be searched in the file and stored in variables for further calculations. To do that, first all variable names, binary types, lengths and offsets (distance from top of file to variable) are saved in dictionaries. After that, binary file is read and data are extracted with a loop, making use of *unpack* function from *struct* library.

This process needs to be done twice. First time, general data from the OCT Scan are extracted. Then, looping through OCT images data, their useful parameters and also themselves are extracted and saved in a new folder.

In particular, this project uses the following variables to obtain retinal thicknesses from Spectralis software:

- SegArray: An array of NumSeg consecutive vectors containing the segmentation data for the different layers found in the B-Scan. NumSeg is another parameter extracted.
- StartX: X-coordinate of the B-scan's start point in mm.
- StartY: Y-coordinate of the B-scan's start point in mm.
- EndX: X-coordinate of the B-scan's end point in mm.
- EndY: Y-coordinate of the B-scan's end point in mm.

With all these parameters, a 3D representation has been built, locating each OCT-Scan coordinates and creating an array with its thickness values.

Second step: obtain thickness

Now it is time to extract thicknesses from predicted images to compare how similar results were between Spectralis software and our predictions.

To do that, the next flow diagram is done:

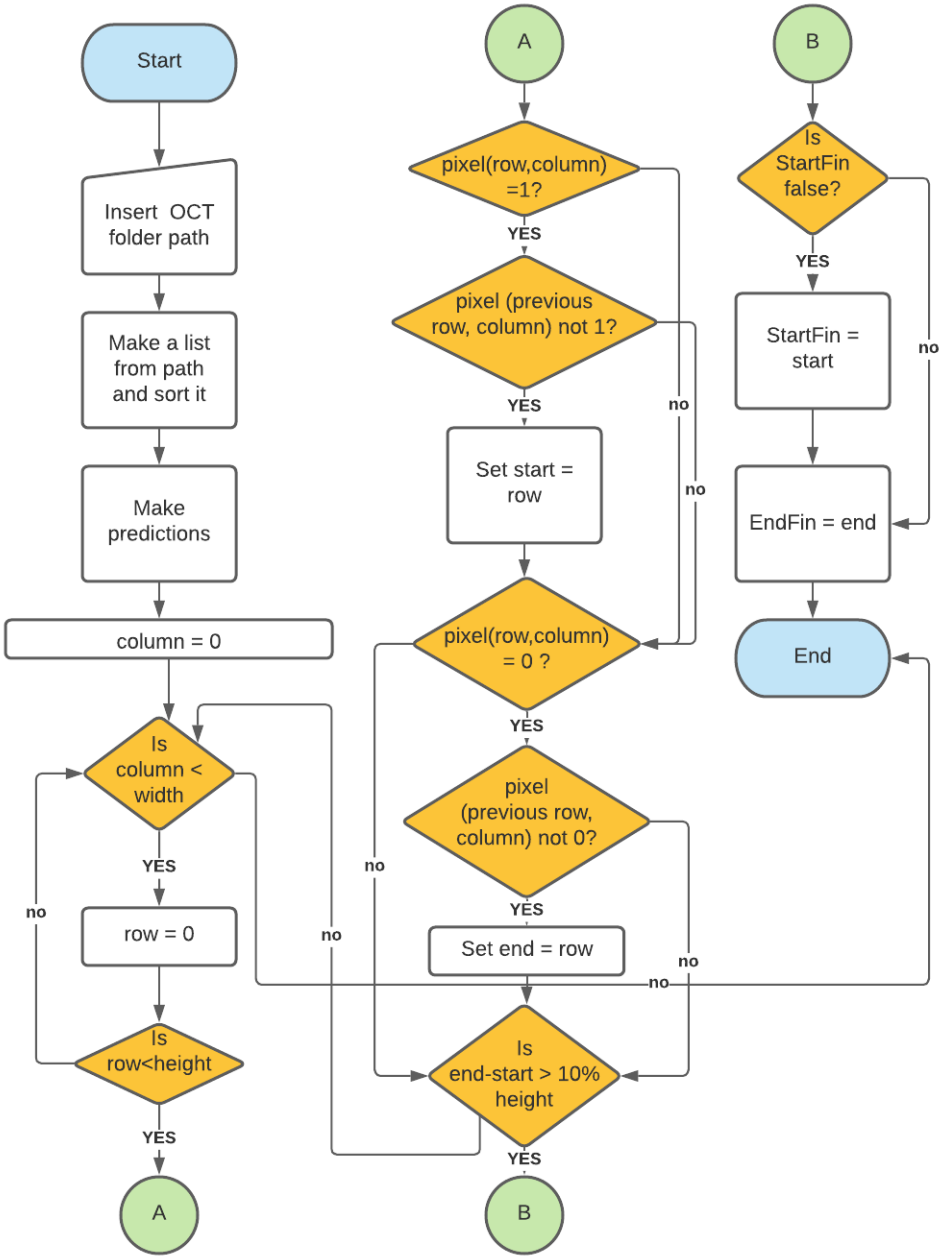

Figure 29. Thickness limits values definition

In last diagram, it is important to know what 4 variables are: *start, end, StartFin and EndFin.*

What code is doing is search in OCT image, by columns, changes in colours. First, from black to white, what will be the starting point (start variable) to count thickness. Then, from white to black, what will be the end point (end variable).

If *end-start* has a size of more than 10% of image height, then *start* value is assigned to *StartFin* if this is set to False. *End* value is assigned to *EndFin* always. This condition of having a minimum

of 10% height is due to the next reason. Sometimes there are mistakes when making the prediction, due to model imprecision or because OCT images are not well scanned. If this happens, code would be setting *StartFin* and *EndFin* values to wrong pixels, obtaining wrong thickness.

After performing this piece of code, *end-start* subtraction provides a thickness value for each column. Composing all column values, a thickness map can be extracted. When all maps are obtained, 3D representation is built. To do that, PlotLy library is used. PlotLy provides an interactive 3D graph, where several actions such as axis rotation or zoom can be done.

Next figures show results obtained in an OCT prediction for a healthy patient:

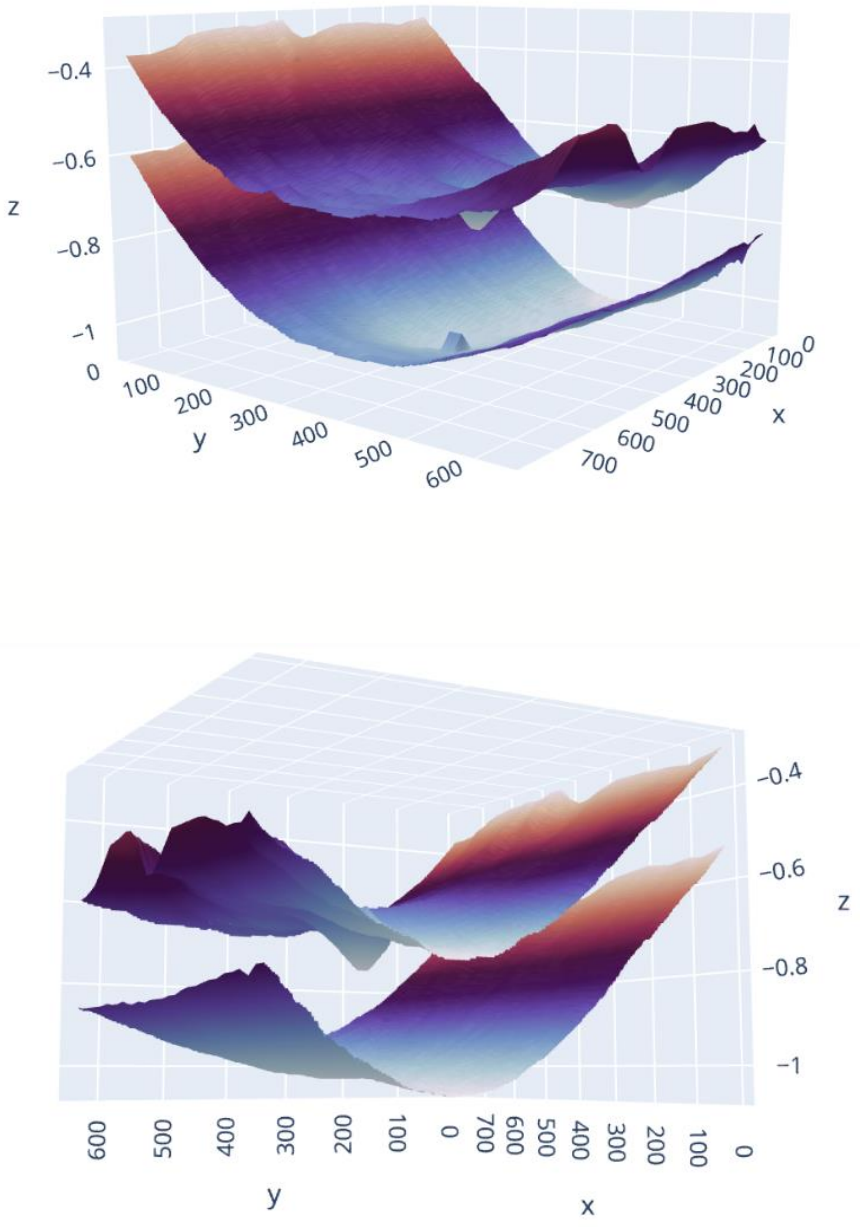

Figure 30. 3D graphs representing start and end points of retina OCT scans in healthy patient

Fovea section can be observed in last representations as a small dip in the centre of the picture.

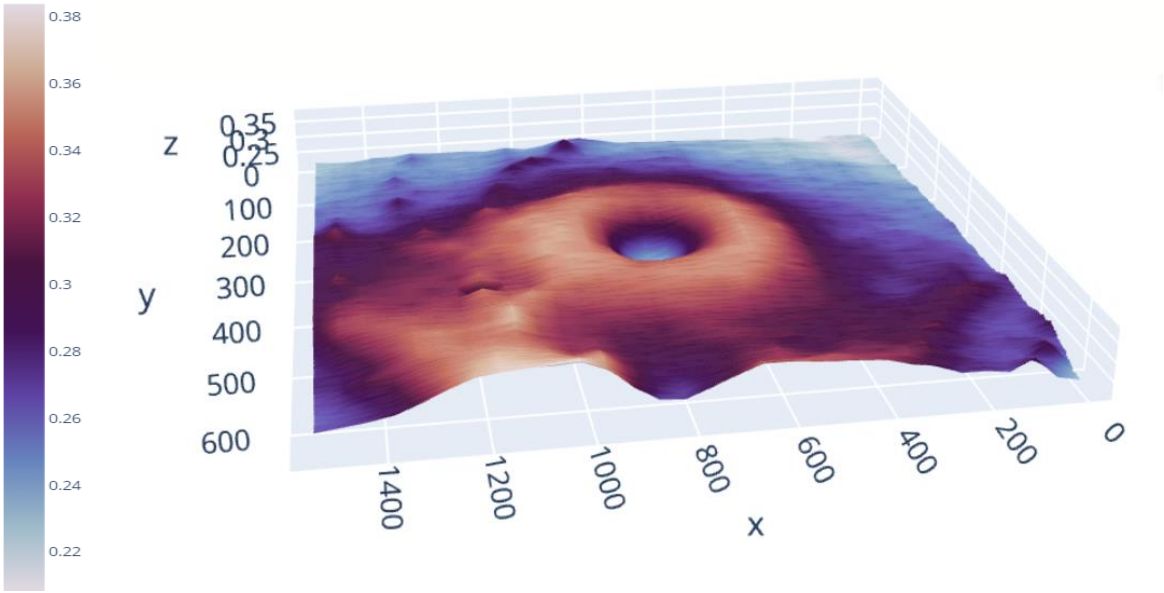

Figure 31. Thickness heat map for OCT retina predictions in healthy patient

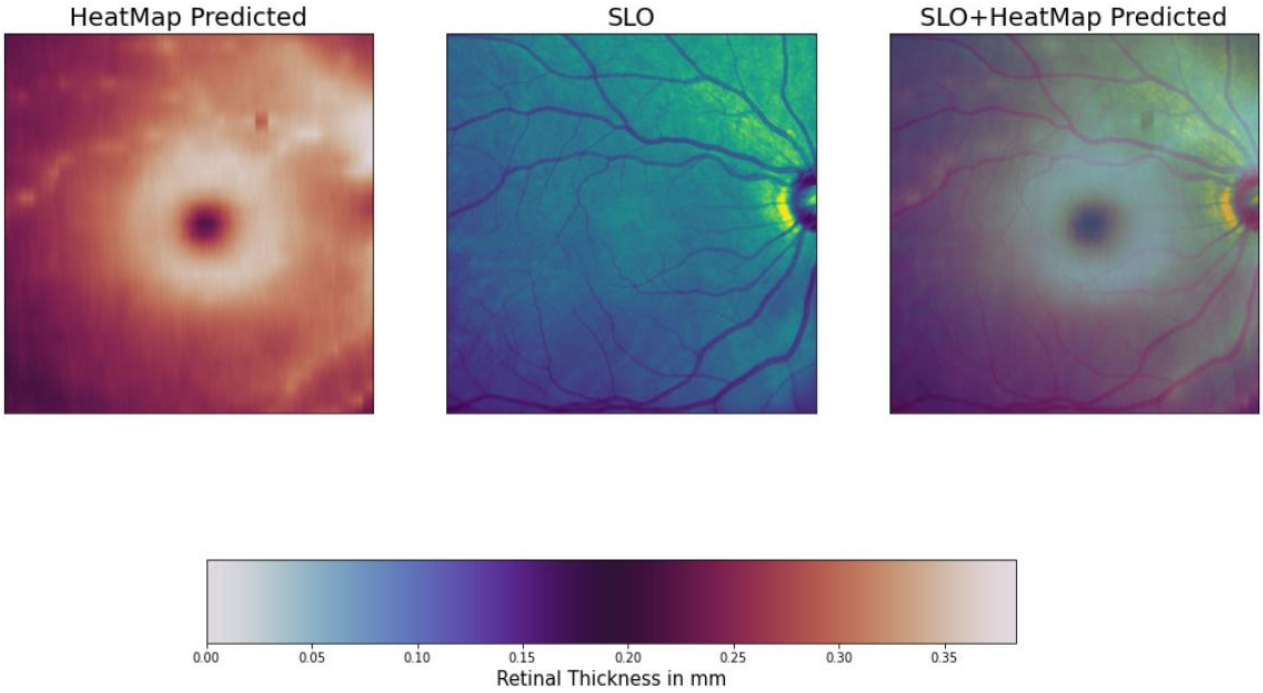

Figure 32. Retina Thickness Heat Map in healthy patient

Last two figures represent thickness along retina, overlapping predicted image and SLO image. Fovea is represented as a small black hole in middle of the heat map.

Maximum thickness obtained is approximately of 0.35 mm whilst minimum is 0.20 mm.

It is important to highlight two major problems that appeared while 3D graphs were developed. First one is related to OCT images where retina does not occupy the whole width of the image (as images below shows). If this happens, previous code to detect 'start' and 'end' retina points does not work. Therefore, several peaks appear in the borders of the image. To improve 3D representation, the decision of removing these values from image was taken, obtaining a softer curve and a better scale.

The second problem is due to number of OCTs that exist in a .vol file. As this number is around 30, results have only 30 values along 'y' axis, so an interpolation is needed. After performing interpolation, graphics were much less irregular.

Figure 32. Example of OCT predicted image where retina does not occupy whole width leading to problems for thickness detection.

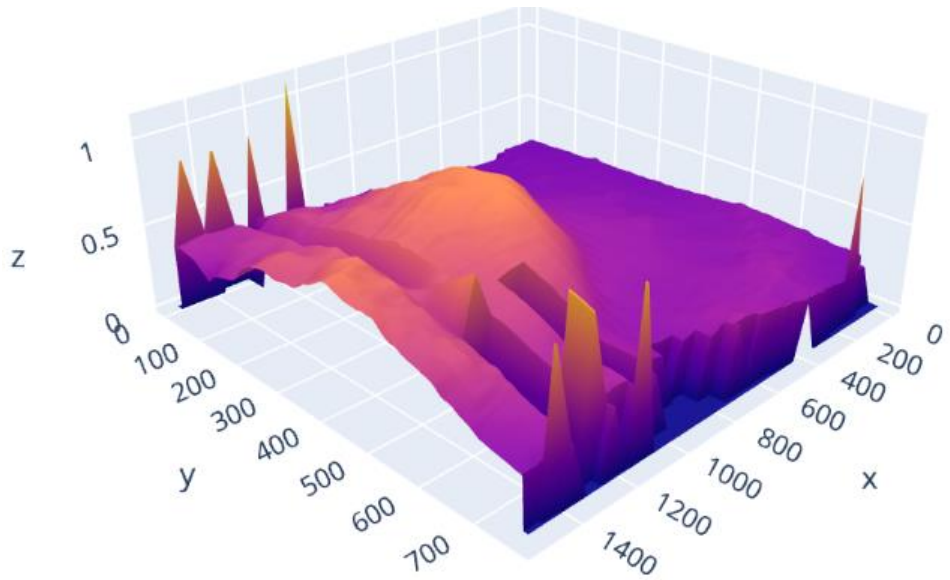

Figure 33. Peaks in 3D borders led to wrong representations in sick patient.

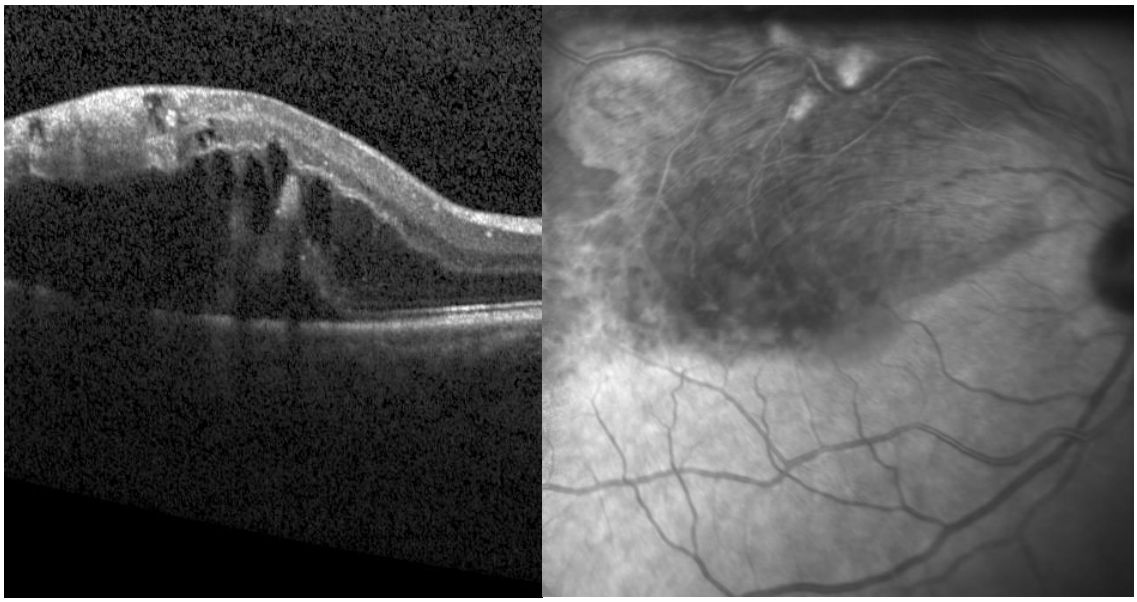

Figure 34. OCT and SLO images of a sick patient.

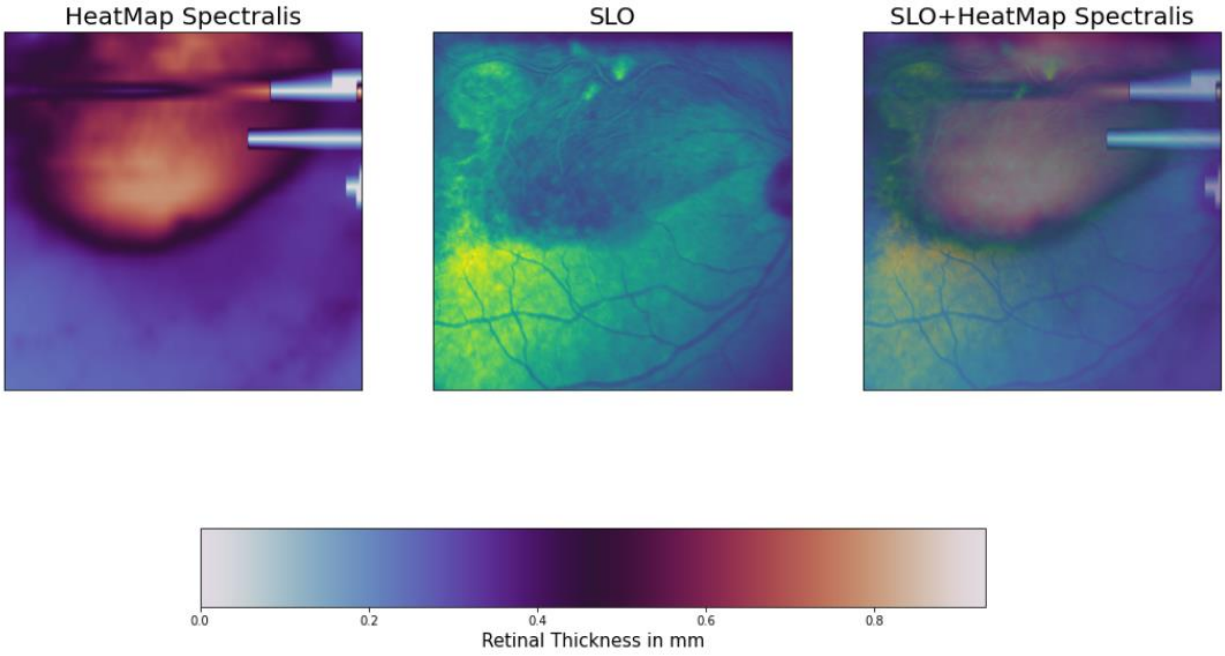

Figure 35. Retina Thickness Heat Map in sick patient

Conclusion

In general, it can be stated that accurate results were obtained. Resultant metrics are high and predicted images show good visual results that could help future health professionals to quickly detect any anomaly related to retinal layers also.

Though several architectures were developed, approximately same metrics were obtained in all of them. Decrease in learning rate leads to better results, as it can be inferred from the ones obtained in layers segmentation with resnet-101 architecture, were an IoU of 0.8320 was obtained with learning rate of 0.001 in contrast with 0.8083 with learning rate of 0.1. Resnet-50 has same results.

The lack of a more powerful computer led to less experimentation also. When performing whole retina training and test, each epoch finished in less than 4 minutes. Thus, results are obtained for 50 epochs. Nevertheless, Google Colab time limits yielded a 15 epoch training in layers segmentation. Therefore, worse metrics are available. It is also important to highlight that U-Net is the least complex architecture (therefore it is time saving also) but approximately has the same metrics as the rest of architectures, so it is the most efficient one.

Furthermore, there was a quick improvement in metrics in a few epochs while training, taking a lot of epochs to continue improving them. This is useful if there is a lack of time while training. In addition, whole retina segmentation results show that right work is done even if original masks have some mistakes, leading to a robust model.

No overfitting was detected neither, as training and test metrics were quite similar and data augmentation did not improve results either. This indicates the presence of a proper Dataset size.

Though the author had no previous knowledge of machine learning, final models implemented and results support the aim of this project, which was obtaining segmented images from a treated dataset. General concepts of machine learning and in particular, image segmentation, have been learnt, providing the student with a different outlook of what an aerospace engineer can do even without having previous knowledge.

There is a need to highlight machine learning’s potential, as it is a relatively new field that needs to be explored and can lead to incredible results in future years.

Future steps

Dataset should be corrected for future trainings, as it was proved that several mistakes existed in masks for layers segmentation. In addition, more layers should be included in dataset. As it can be appreciated from segmented images, there is a layer called ‘rest of retina’ that includes several retinal layers. Therefore, including more segmented layers will lead to more useful results. For each OCT scan, its corresponding segmented masks for each layer should be joined in a unique mask, leading to a size reduction of dataset folder and, therefore, less programming and computational cost. If more layers are segmented, then different features can be extracted, like each layer thickness or the whole retina volume.

A classification model should be added too, in order to detect sick patients as a first step. Then, as a second one, layers segmentation and feature extraction would be performed, widen data available.

There is a need to obtain a more powerful computer too. Google Colab dependency leads to time limits for GPU use and even sometimes GPUs are not available. Therefore, less experimentation can be performed. With a more powerful computer, more hyperparameters could be modified and therefore more conclusions obtained.

As a further step, a completely integrated application for health professionals could be developed. This application would include both classification and segmentation and a sickness probability as output, for example. Furthermore, if all patient data can be joined together, a statistical study should be carried out to predict future population retinal diseases.

Bibliography

[1]: Li, Q., Li, S., He, Z., Guan, H., Chen, R., Xu, Y., Wang, T., Qi, S., Mei, J., & Wang, W. (2020, December 9). *DeepRetina: Layer segmentation of retina in OCT images using deep learning*. Translational vision science & technology. Retrieved September 14, 2021.

[2] Dan P. Popescu et al. (2011, August), *Optical coherence tomography: fundamental principles, instrumental designs and biomedical applications (nih.gov)*.

[3]: Long, J., Shelhamer, E., & Darrell, T. (2015, March 8*). Fully convolutional networks for semantic segmentation*. Retrieved September 14, 2021.

[4]: Chen, L.-C., Papandreou, G., Schroff, F., & Adam, H. (2017, December 5), *Rethinking atrous convolution for semantic image segmentation*. Retrieved September 14, 2021.

[5]: PyTorch, *PyTorch Documentation*. Retrieved September 14, 2021.

[6]: *Sensio. Inteligencia Artificial desde cero en castellano*. Retrieved September 14, 2021.