Convolutional neural networks for multiple sclerosis lesion detection

Francisco Javier Blázquez Martínez

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Embedded system laboratory,
Institute of Electrical and Micro Engineering

Project director: Dr. David Atienza Alonso
Project supervisors: Dr. Arman Iranfar,
Dr. Tomás Teijeiro,
Dr. Marina Zapater

June 2021

Abstract

DeepHealth project arises with the aim to offer a unified framework completely adapted to exploit underlying heterogeneous High-Performance Computing and Big Data architectures, providing state-of-the-art techniques in Deep Learning and Computer Vision tools for biomedical uses. The part of this framework in charge of providing the required Deep Learning infrastructure is the EDDL library.

The aim of this project is to create Convolutional Neural Networks for Multiple Sclerosis lesion detection using the EDDL library, evaluating and profiling the hardware requirements in both training and predictions generation and at the same time comparing it to another commonly used library for Deep Learning.

We first analyze 2-D Convolutional Neural Networks, specifically the U-Net and the Double U-Net and then we analyze the usage of a cascade model of 3-D Convolutional Neural Networks.

Acknowledgements

In first place I would like to thank Prof. David Atienza, not only for giving me the opportunity to work in this project but also for the whole year I have been studying at the EPFL. Now that it's finishing I know it has been one of the most didactic and beautiful experiences in my live and I am aware that this wouldn't have been possible without his work, support and appreciation of the Complutense University.

I also want to thank Dr. Marina Zapater, Dr. Tomás Teijeiro and especially Dr. Arman Iranfar for giving me their time and helping me during the development of the project. It has been an honor and a pleasure to discover this world by their hand.

By last, I would like to thank my family for their unconditional support in every single aspect of my live.

To all, thank you very much from the heart.

To my father,

for all the things that I could only learn from his example.

Contents

Chapter 1

Introduction

In the last years, with the increase of the available biomedical data and the computation power in GPUs together with the appearance of new Artificial Intelligence (AI) models and techniques, it has existed and exists a growing interest and research effort on applying Artificial Intelligence to several biomedical tasks. In many of these tasks, most of them currently performed manually by doctors or medicine professionals, there have appeared automatic tools based on Artificial Intelligence models which show a performance close to the accuracy of a doctor. It's undeniable that the application of Artificial Intelligence to medicine has proven to be useful and is definitely part of the future of the discipline. It will help by saving doctor's time, increasing the efficiency and the scope of the medicine professionals, helping them to provide more accurate diagnoses or even creating diagnoses by themselves. Ultimately, they will help save lives.

Aware of this, the European Comission starts in 2019 the DeepHealth project. This aims to offer a unified framework completely adapted to exploit underlying heterogeneous High-Performance Computing (HPC) and Big Data architectures, framework to be assembled with state-of-the-art techniques in Deep Learning (DL) and Computer Vision. In particular,the project combines High-Performance Computing infrastructures with Deep Learning (DL) and Artificial Intelligence techniques to support biomedical applications that require the analysis of large and complex biomedical datasets and thus, new and more efficient ways of diagnosis, monitoring and treatment of diseases. The framework consist on two general purpopse libraries which are currently being developed, EDDL for Deep Learning computing infrastructure and ECVL for computer vision tools.

The goal of this project is to create state-of-the art Deep Learning models to detect Multiple Sclerosis (MS) lesions in Magnetic Resonance Images (MRI) using EDDL library, analyzing its performance and comparing it with other commonly used libraries.

1.1 Multiple Sclerosis lesion segmentation

Multiple Sclerosis lesion segmentation consists on identifying in an MRI the damaged parts of the brain. This task plays a role in every stage of the disease, it has a huge importance in the patients for an early diagnose, for following the evolution of the disease and for measuring the effects of the treatment. However, this task has to be done manually by experts and involves analyzing a huge amount of data so it's highly time-consuming and prone to errors. Considering also the how relatively common this disease is, it's clear the importance of trying to optimize it. In fact, MS lesion segmentation was established as one of the main use cases for the development of the DeepHealth project.

The problem with MS lesion segmentation is not only that it is a difficult task *per se*, but also the difficulty and cost of obtaining and labeling the data. Few years ago there were no public datasets and now the existing ones count only with a few samples, what makes almost any Machine Learning approach to be unfeasible. However, in recent years, within MICCAI2008 and MICCAI2016 challenges, there have been a big research effort which has led to the emergence of various different approaches and solutions. MICCAI2016 challeng proceedings [1] shows the usage of classical image segmentation networks, Random Forests, Hybrid Artificial Neural Networks, Automated Multimodal Graph Cut, unsupervised approaches using Rules and Level Sets...

In this project we'll focus our attention on the state-of-the-art approaches using 2-D and 3-D Convolutional Neural Networks (CNN).

We first analyze the application of 2-D CNN, starting with the U-Net [2] as the standard for biomedical image segmentation, building, training, evaluating and profiling the model. Then we try a newer approach, the Double U-Net [3], which has proven to improve U-Net for some tasks, comparing both the performance of EDDL library with respect to a keras-tensorflow implementation and the models between themselves.

By last, we analyze the application of a cascade 3-D CNN. Profiling the model and comparing it to an equivalent implementation with keras-tensorflow.

Chapter 2

Data

The two main datasets for MS lesion segmentation are those for MICCAI 2008 and MICCAI 2016 challenges. In this project we'll work over MICCAI 2016 dataset.

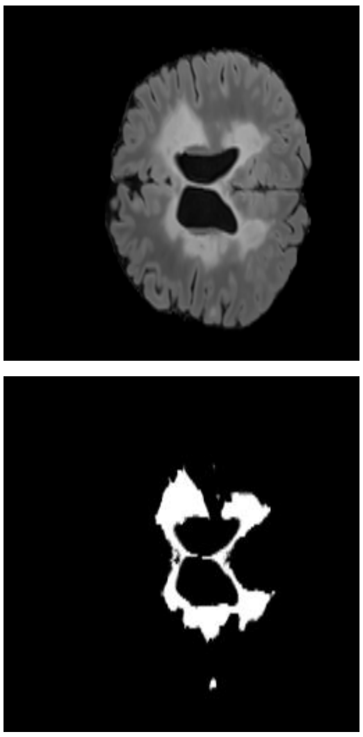
2.1 MICCAI 2016 dataset

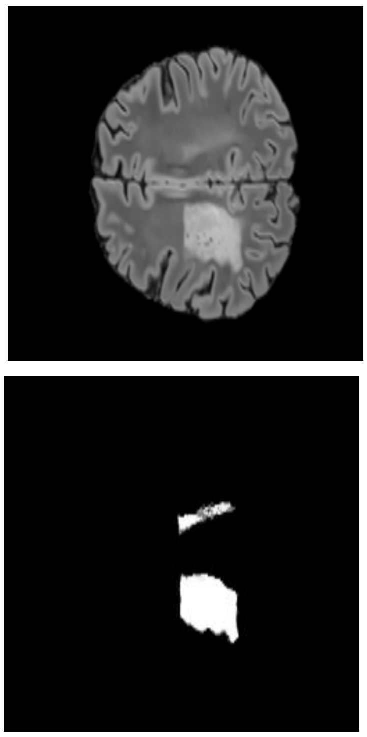
MICCAI 2016 dataset consists on 15 MRIs together with a lession mask generated by consensus among various doctors. These are generated with three different scanners, each one with a certain data shape. We verified that all the MRIs are present with the same orientation, what will be meaningfull when slicing the 3-D MRIs for the 2-D models. The dataset contains also preprocessed data in which the skull has been removed from the MRIs, for simplicity, we'll be using these for all the models.

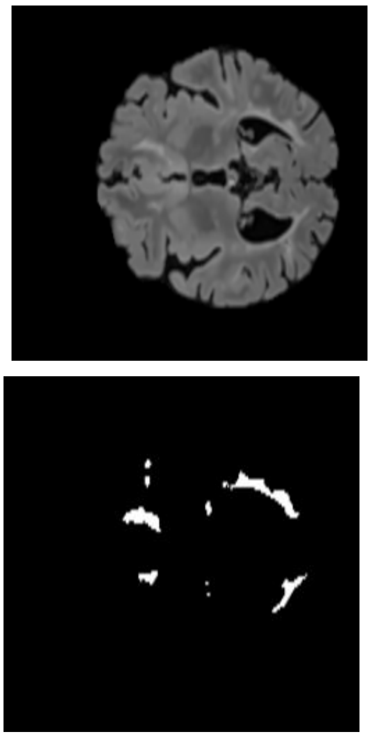
We split randomly the data in a train and a validation set with size 12 and 3 MRIs respectively. We don't keep a test set because given the fact that the size of this would be reduced to only one or two MRIs at most and the inhomogeneity in the different lesions shape, size, quantity and distribution, the evaluation over the test set would lead to a unreliable measurement. This is something certain after seeing the big differences in the metrics over different validation MRIs. Therefore we encourage to everyone who wants to use this models in practice to previously evaluate them over his own data.

We present in the next page some slices of MRIs used as benchmarks for error detection and visual analysis of the models from both the train and validation datasets.

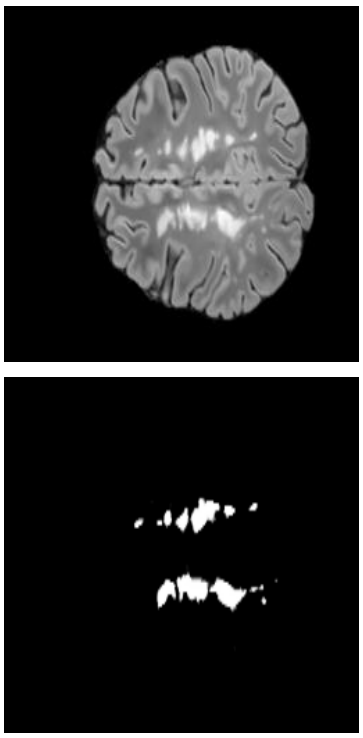
Becnhmark training images:

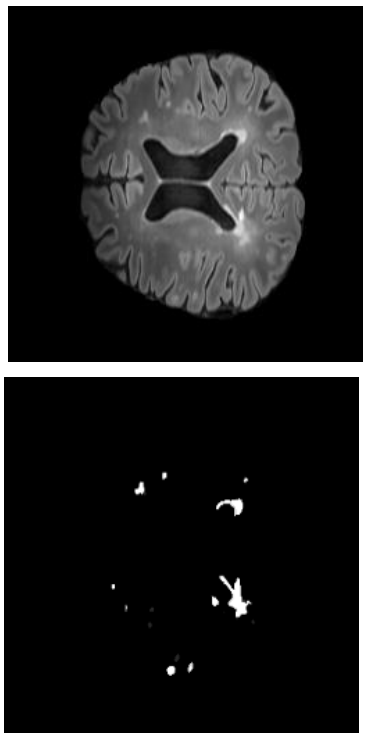

(a) tr175

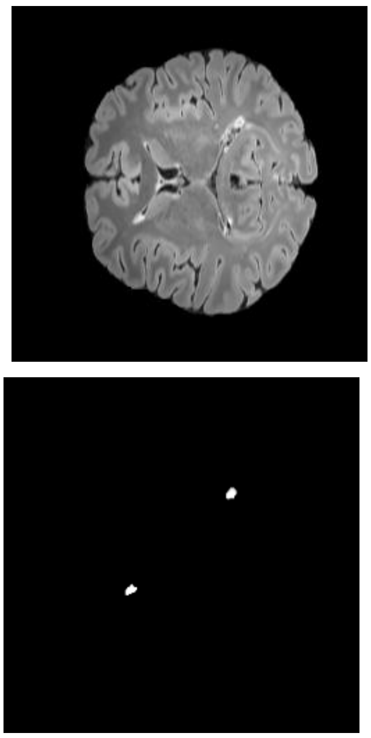

(b) tr943


(c) tr1426

Benchmark validation images:


(d) ts175


(e) ts431


(f) ts687

2.2 Evaluation metrics

During the training processes we have used *binary cross entropy* as loss function. For evaluation we have used the *Dice score*, a classical metric for image segmentation, much more representative than others used in classification tasks as *Mean Squared Error* or *Binary accuracy*. *Dice score* is implemented in several similar but not fully equivalent ways and it's not a built-in metric in keras-tensorflow so we had to create our custom implementations for both the tensorflow and PyEDDL models. This way we also ensure they are strictly equivalent.

$$Dice(Pred, Mask) = 2 * \frac{|Pred \cap Mask|}{|Pred| + |Mask|}$$

```
import keras.backend as K

def dice(ytrue, ypred, smooth=1):
    intersection = K.sum(ytrue * ypred, axis=[1,2,3])
    union = K.sum(ytrue, axis=[1,2,3]) + K.sum(ypred, axis=[1,2,3])
    dice = K.mean((2. * intersection + smooth)/(union + smooth), axis=0)
    return dice
```

LISTING 2.1: Keras backend Dice implementation

```
from pyeddl.core  import Metric

class Dice(Metric):
    def  init (self, threshold=0.5):
        Metric. init (self, ”pydice”)
        self.threshold = threshold

    def value(self, msk, out, smooth=1):
        outnp = out.getdata() ¿= self.threshold
        msknp = msk.getdata().astype(np.bool)
        intersection = np.logicaland(msknp, outnp)

        dice = 0
        for i in range(msknp.shape[0]):
            unioni = np.sum(msknp[i]) + np.sum(outnp[i])
            dice += (2*np.sum(intersection[i])+smooth) / (unioni+smooth)

        return dice
```

LISTING 2.2: PyEDDL Dice implementation

Chapter 3

2-D Convolutional Nerual Networks

During the development of the project, we have worked with four different PyEDDL versions (0.12, 0.13, 0.14 and 1.0.0) since this is a library currently in development. 3-D Convolutional Neural Networks were finally supported in PyEDDL version 1.0.0, which was released on May 27th 2021. Therefore the first CNN had to be 2-dimensionals.

This has two main drawbacks. The first one is the loss of some context information since our CNN can not analize at once a whole region of the MRI but it has to slice it, an issue inherent of working in a lower dimension. The second one is the need of having to consider also the different axles and orientations, which can be solved with data preprocessing.

3.1 Data preprocessing

For the 2-D CNN we have only used *FLAIR* modality of the MRI. We first resize the *FLAIR* image to $(256 \times 256 \times 256)$ so that, independently of the scanner used, the input has the same shape. After this we slice the data by the last axis. We decided this shape even when the first axis is being upsampled in all the MRI because it allows us to change the orientation or slicing axis and still having the same input shape so transfer learning techniques are possible. As a final preprocessing step we normalize each image.

No data augmentation techniques have been used since the final purpose of this project is not achieving the best accuracy (we remit to MICCAI 2016 challenge and later research works for that) but having a fair analysis and comparison of EDDL library.

3.2 U-Net

U-Net [2] first appeared in 2015 and it quickly became the standard for biomedical image segmentation. We chose it as the model for our first approach. The network structure can be seen in the image below, the only change in the network structure in our implementation with respect to the one originally presented is that we have added batch normalization after each convolution for a better convergence and stability.

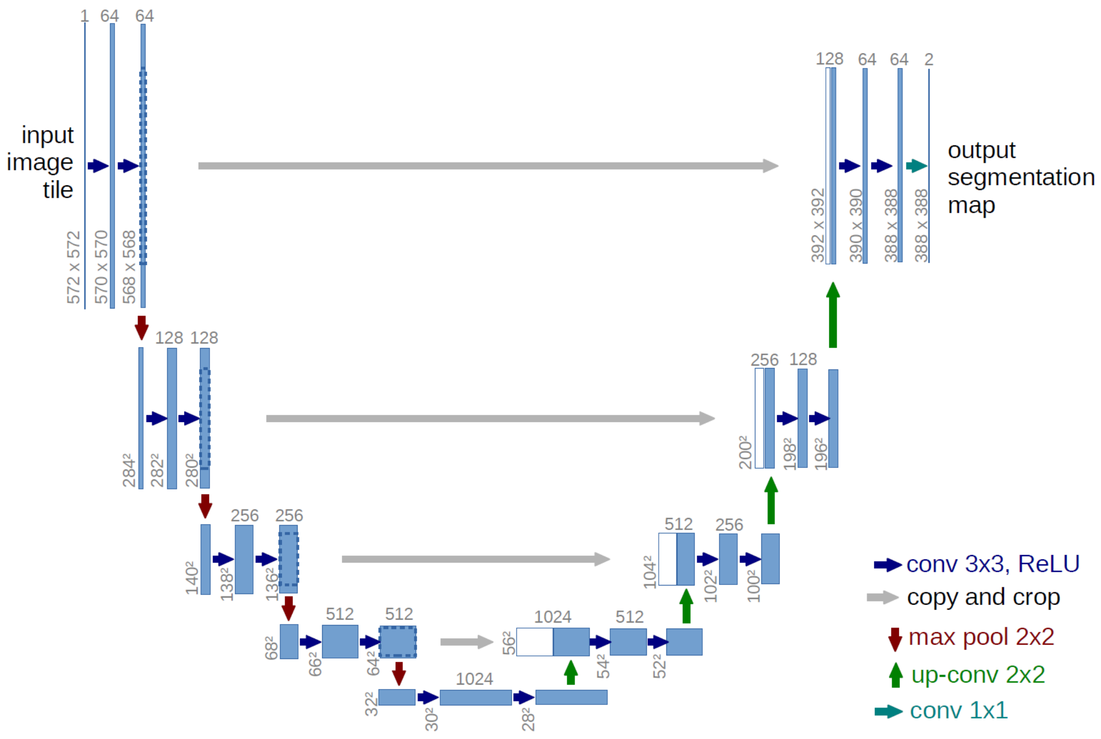
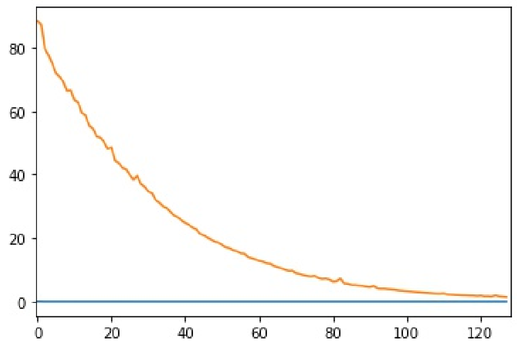


FIGURE 3.2.1: U-Net model [2]

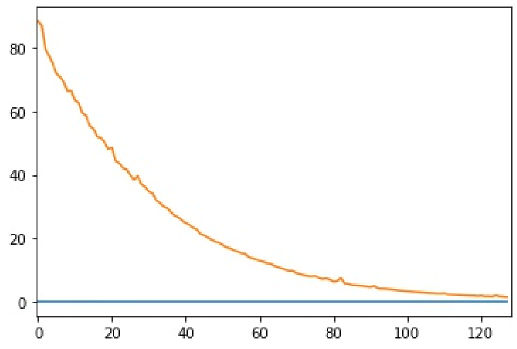
As mentioned before, we are using *Binary Cross Entropy* as loss function. In all the 2-D models we use *Adam* optimizer with *Learning Rate* set to 0.001 and batch size of 8 for both training and evaluation. In the case of the PyEDDL implementations, when creating the computing service (CPU or GPU) we set memory utilisation to *low_mem* so that all the models can be trained in a GPU with 16GB of memory. For a fair comparison, we maintain this value also when the target computing service is the CPU.

We present the results of the profiling and evaluation of both a keras-tensorflow implementation and PyEDDL one (in blue and orange respectively) in the following sections:

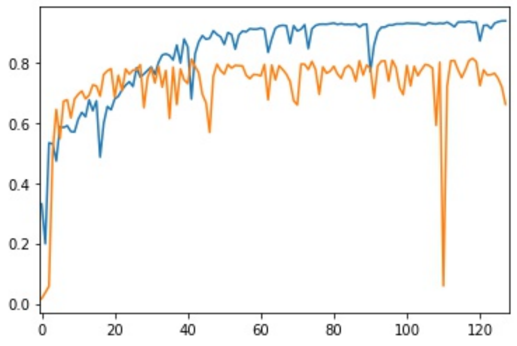
3.2.1 U-Net evaluation



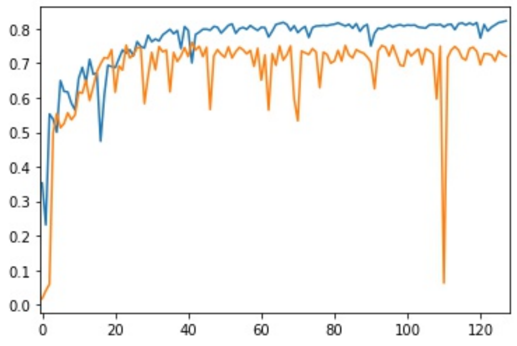
(a) Loss function in training per epoch



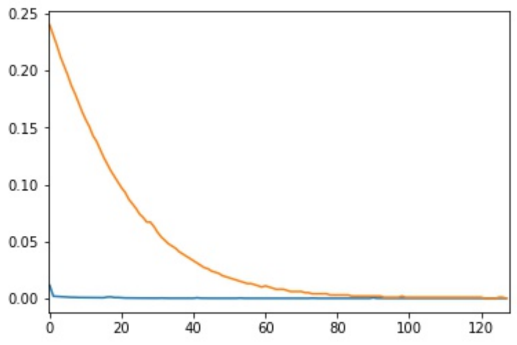
(b) Loss function in validation per epoch



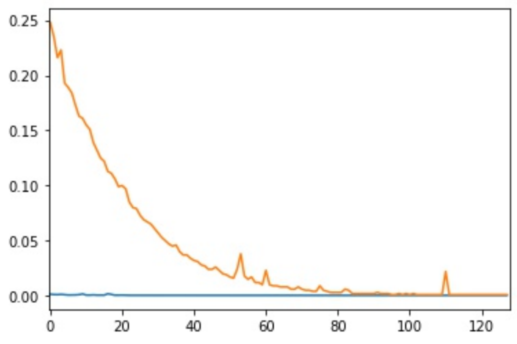
(c) Dice score in training per epoch



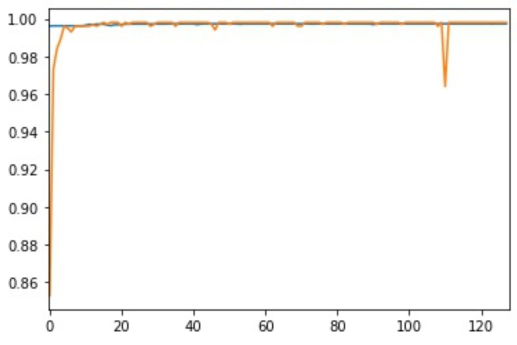
(d) Dice score in validation per epoch



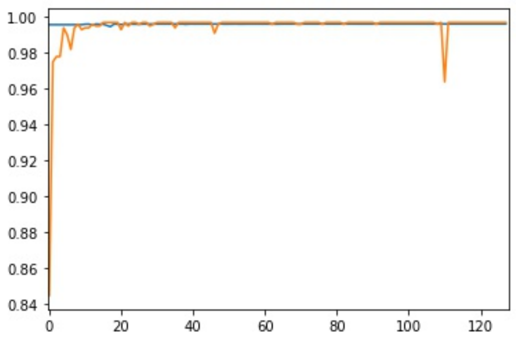
(e) Mean Squared Error in training per epoch



(f) Mean Squared Error in validation per epoch

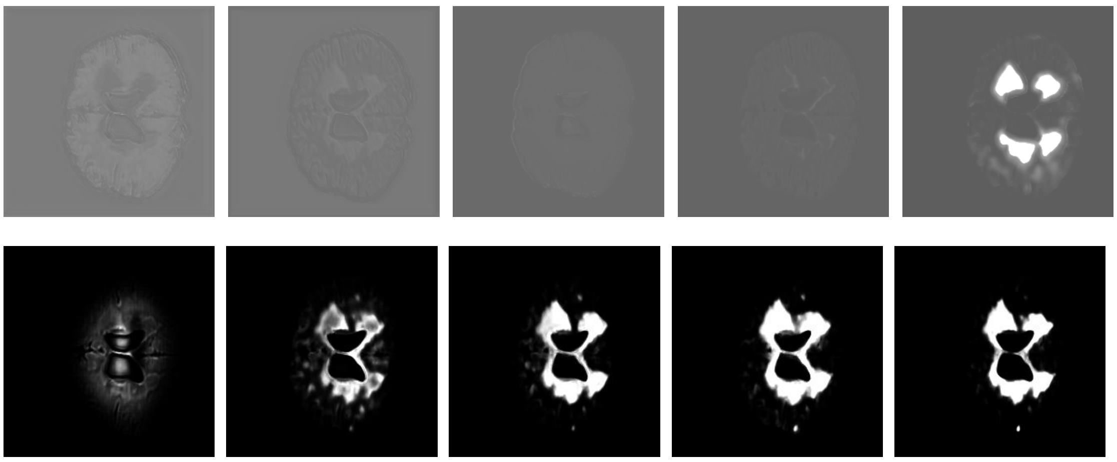


(g) Binary Accuracy in training per epoch



(h) Binary Accuracy in validation per epoch

As we can see, there are huge differences in the loss function between the PyEDDL and keras-tensorflow implementations. Since these start from the very first epoch, this can be caused by the weights initialization. We can ensure this is the case by having a look to our benchmark images in the first epochs. PyEDDL U-Net predictions are shown above and keras-tensorflow model predictions below for the benchmark image 2.1(a) at epochs 1,4,7,10 and 13.

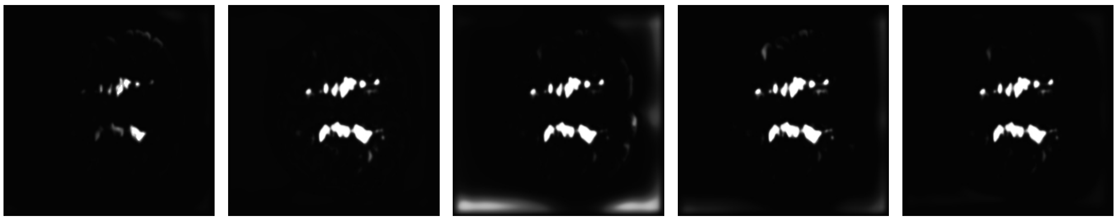

(n) epoch 1 (o) epoch 4 (p) epoch 7 (q) epoch 10 (r) epoch 13

We can also appreciate how the *dice* metric is not so affected after few epochs. This is because, while *binary cross entropy* takes in consideration the output of the network, the probability of each pixel to be part of a damaged lession, the *dice* metric applies a threshold and only considers the class label assigned to that pixel. We appreciate the exact same contrast with the *mean squared error* and *binary accuracy.* We decided to also include them for this first evaluation to make even more clear that they are not valid metrics for our task.

There are three main appreciations when looking at the *dice* score of the models. The first one and more clear is that the tensorflow (TF) model is obtaining better lesions segmentations. The second one is that, while the PyEDDL model converges, the TF model experiences a continuous improvement during the whole 128 epochs. Finally, it's important to remark that the TF model achieves a experiences a more stable progress. Both models make missteps, i.e., epochs in which we appreaciate a drastic reduction on the *dice* score, however, the TF model progres is more stable.

Since this can be caused by the proper nature of the stochastic optimization process, even more with our small batch size, we recommend a repetition of the experiment before reaching conclusions.

In the images below we show an explanation of the drastic reduction in the *dice* score of the PyEDDL U-Net at epoch 110. Looking at the benchmark image 2.1(d), in different stages of the training process we can appreciate how the model detects lesions in the frame of the image, being this a result of course of overfitting.



(s) epoch 108 (t) epoch 109 (u) epoch 110 (v) epoch 111 (w) epoch 112

FIGURE 3.2.2: Drastic dice score reduction at epoch 110

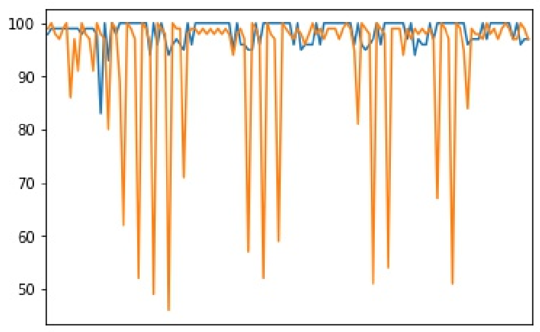
3.2.2 U-Net GPU profiling

The profiling was done for PyEDDL 0.14.0 and TF 2.5.0 in a Tesla T-4 GPU with 16GB.

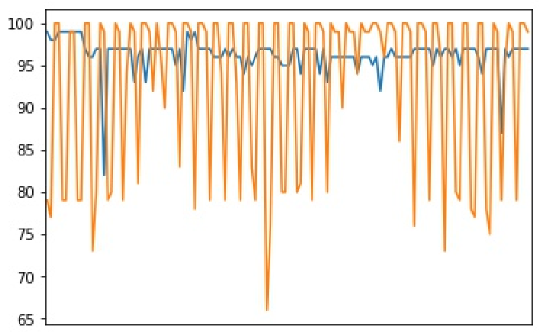
As mentioned before, we had to set the configurable memory usage parameter as *low_mem* when selecting this GPU because with a more ambitious parameter we experienced a *run out of memory* error. This is a key difference. Tensorflow analyzes and evaluates the available resources at runtime and adapts to them for a faster execution and with PyEDDL we have to pre-fix a limit to the memory usage. This makes that tensorflow can extract the maximum of the available resources, using more memory and fully using the GPU while with PyEDDL we have our memory usage limited by this parameter (even when we have more memory available) and our GPU is not able to run at its full capacity.

On the other hand, the power consumption is the same, being a bit more stable in the case of tensorflow and the GPU temperature is similar.

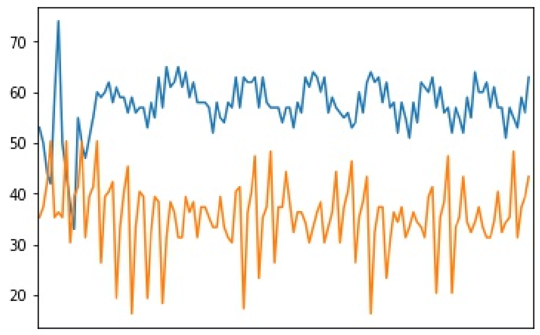
For the temperature profiling, which showed a very stable behaviour, we decided to profile it both when starting the execution from rest until stabilization and in the middle of the execution, in a stable middle point. This is the reason behind the differences in the train and evaluation GPU temperature profiling. The training profile 3.3(g), in the left, shows the the increase in temperature while the evaluation profile, in the right, shows the fluctuations in the middle of the execution. Both models behave in a very similar way being the temperature of the GPU a bit higher in the TF model.



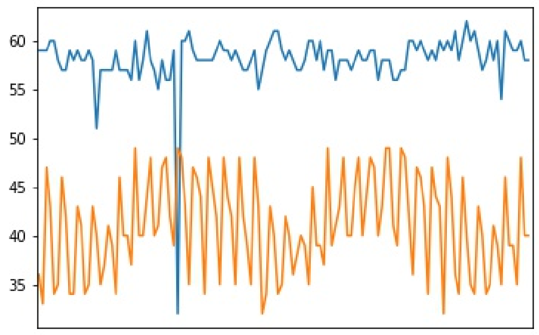
(a) Training GPU usage (%)



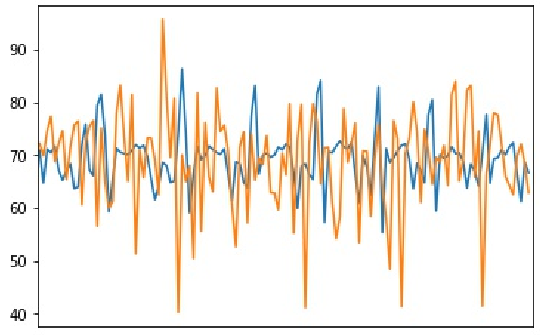
(b) Evaluation GPU usage (%)



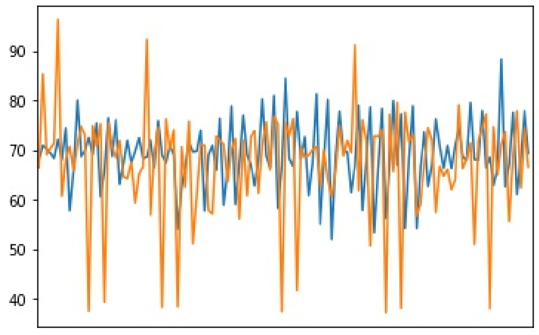
(c) Training memory usage (%)



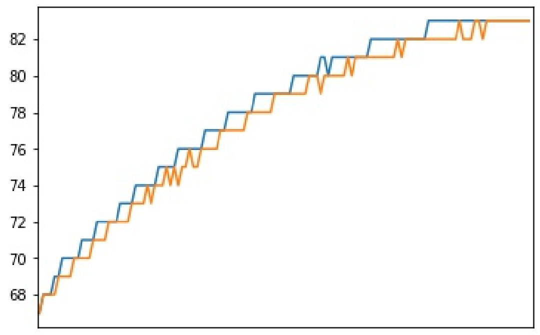
(d) Evaluation memory usage (%)



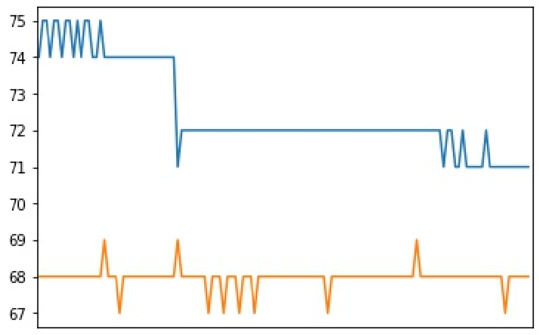
(e) Training power consumption (W)



(f) Evaluation power consumption (W)



(g) Training GPU temperature (°C)

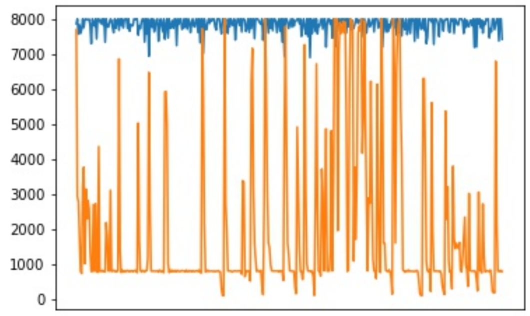


(h) Evaluation GPU temperature (°C)

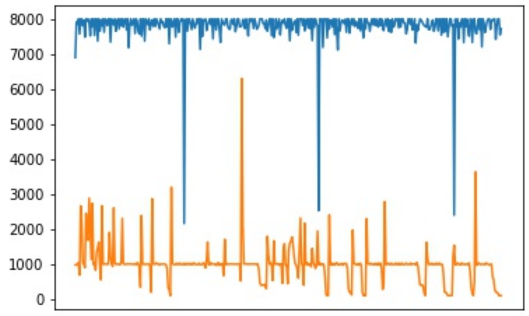
FIGURE 3.2.3: U-Net Tesla T-4 profile

3.2.3 U-Net CPU profiling

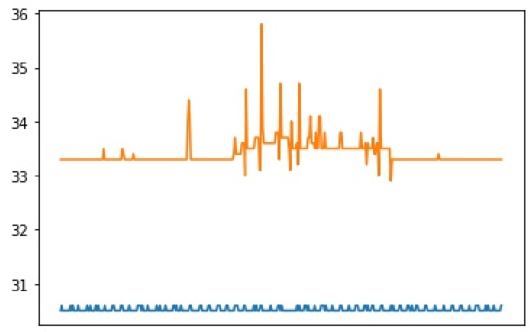
The CPU profiling was done for PyEDDL 1.0.0 and TF 2.5.0 (latest stable versions when writting this report) in a server of the *Embedded System Laboratory* of the *École polytechnique fédérale de Lausanne*. The server counts with 80 CPUs and a total of 376GB of RAM memory.



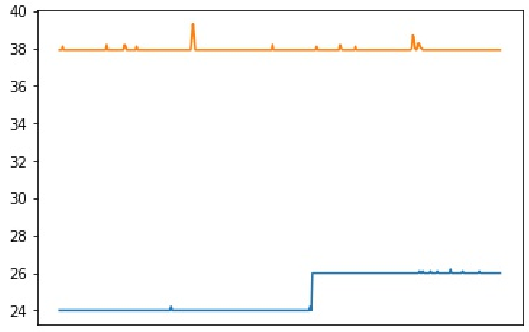
(a) Training CPU usage (%)



(b) Evaluation CPU usage (%)



(c) Training memory usage (GB)



(d) Evaluation memory usage (GB)

For the profile we again set the memory parameter of the PyEDDL CPU computing service to *low_mem* and we also set *th=-1* so that we allow PyEDDL to use all the available CPUs, configuring TF also in the same way.

At this point, the data speaks for itself. There are evident differences in the resources management. While TF is able to fully use all the CPUs with a reasonable cost of memory, PyEDDL is only able to use around one eigth of the computation power, the equivalent of 10 CPUs, and with a much higher memory cost. During training we can appreciate some peaks of CPU utilization reaching all the server capacity but these are unstable.

As in previous versions of PyEDDL, TF offers a better memory consumption and resources management. These are are two of the main niches of improvement of this library in development.

3.3 Double U-Net

The second 2-D CNN implemented to address the problem of MS lesion segmentation was the Double U-Net [3]. It appeared last year and, following the encode-decode logic of the U-Net model, duplicates this adopting in its structure some networks which have proven to be useful or a significative improvement in several tasks, as the VGG-19 and the ASPP. As U-Net, it's a fully CNN, it processes the image in its totality.

The original model structure can be seen below. The only modifications done with respect to it is that our output is *output2* and that we had to add concatenations of a layer with itself at the beggining of the *Network 2* because PyEDDL only supports multiplication of tensors with the exact same shape.

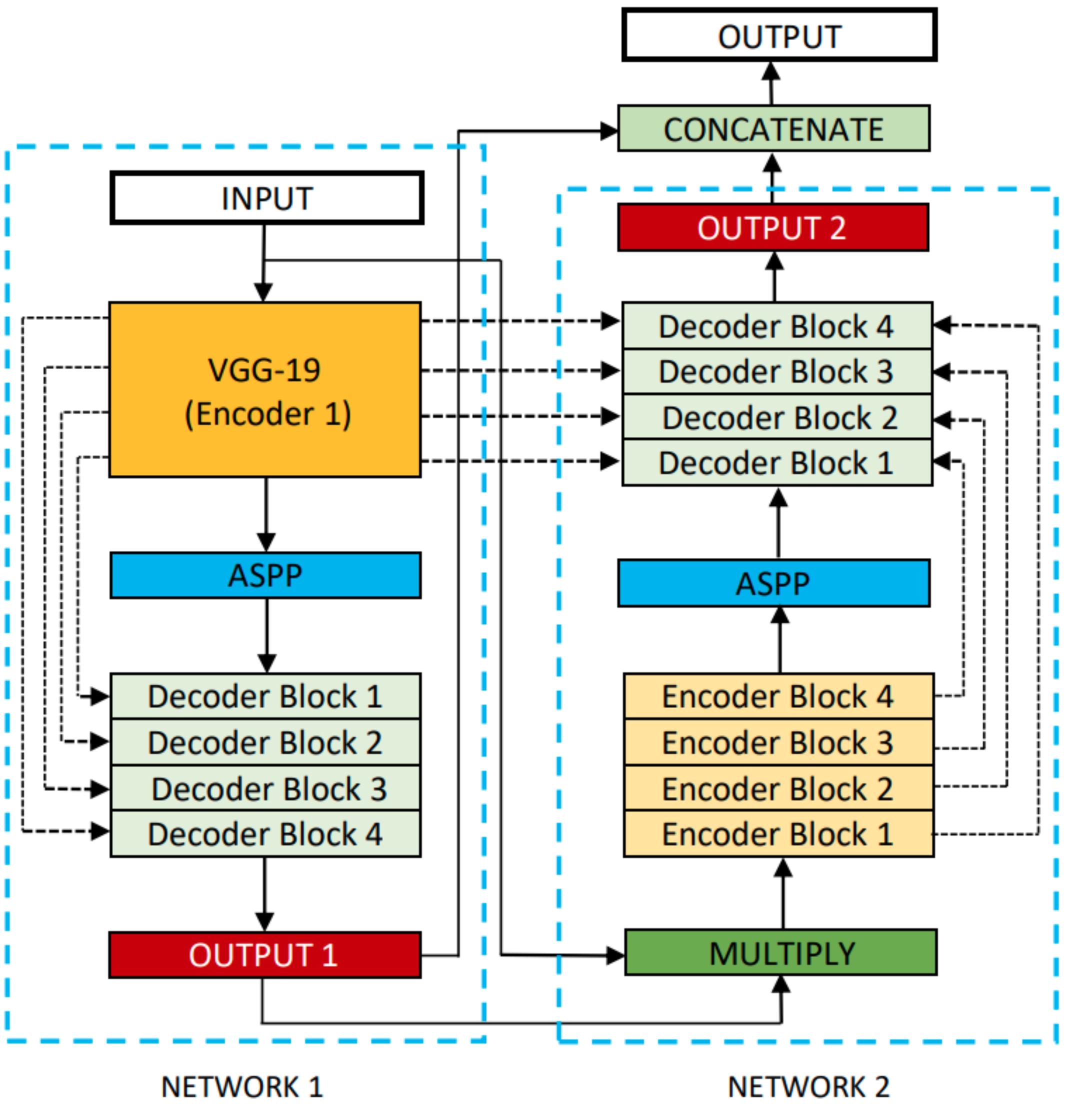
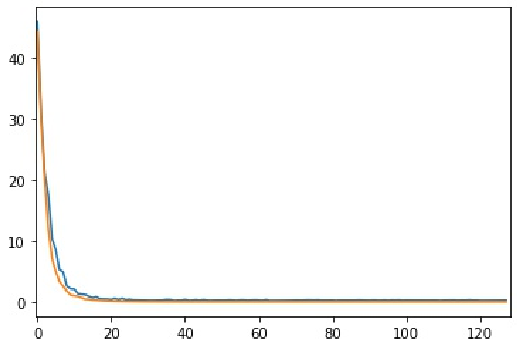
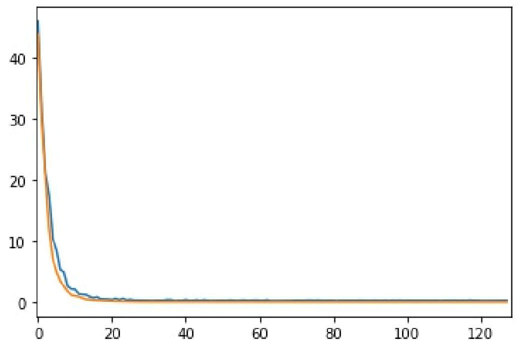
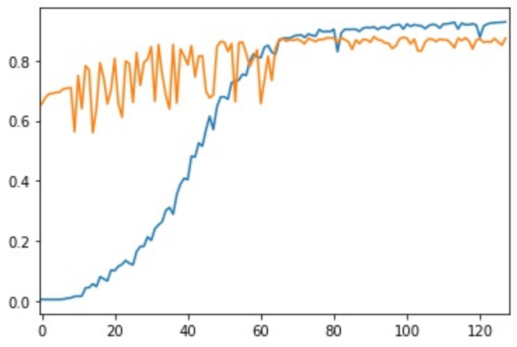


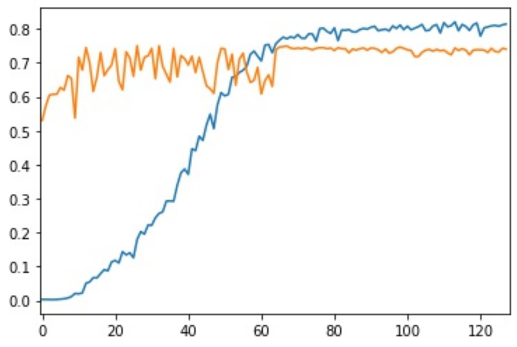
FIGURE 3.3.1: Double U-Net model [3]

3.3.1 Double U-Net evaluation

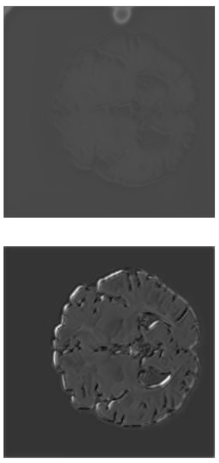

(a) Loss function in training per epoch

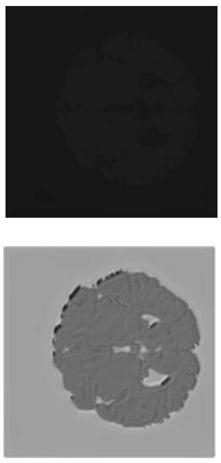

(b) Loss function in validation per epoch

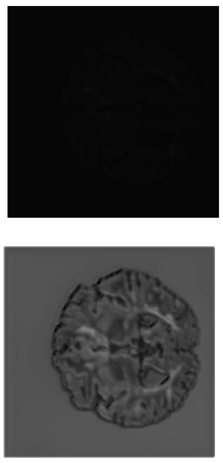

(c) Dice score in training per epoch

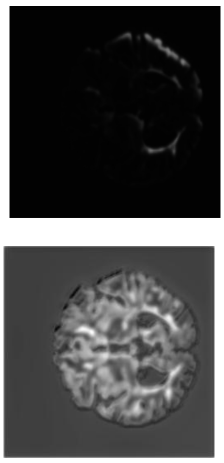

(d) Dice score in validation per epoch

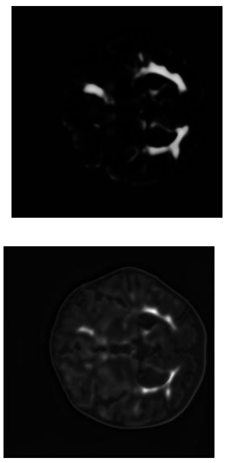
We can appreciate again how the TF model follows a continuous improvement while the PyEDDL model stabilizes with a lower *dice* score. It's remarcable however how slow is the increase of the quality of the segmentations of the TF model. Again, this is due to weights initialization. Looking at the benchmark image 3.2(d) can appreciate how the PyEDDL model (above) starts from a closer initialization to the output mask and aproximates to this much faster than the TF model (below).


(j) epoch 1

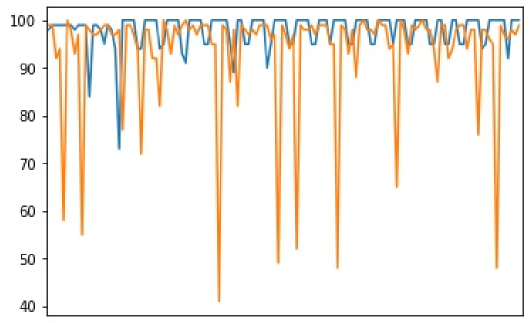

(k) epoch 4

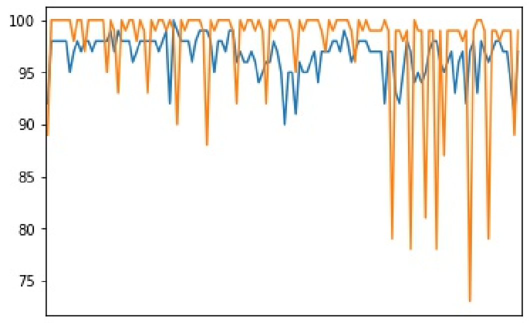

(l) epoch 7

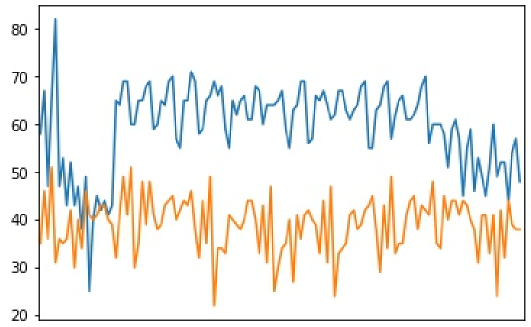

(m) epoch 10

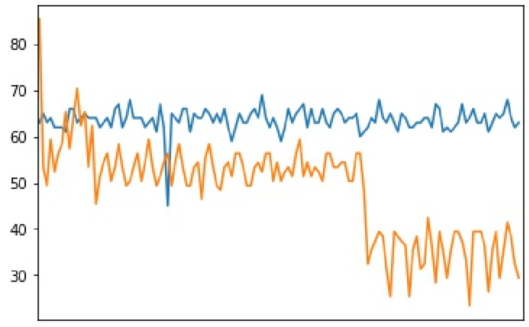

(n) epoch 13

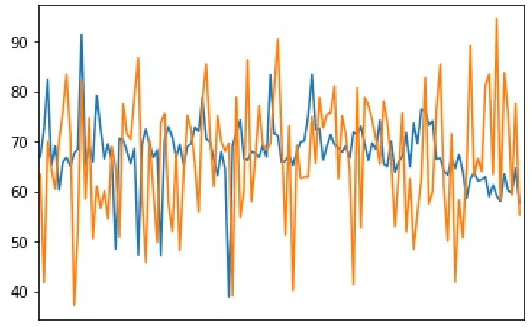
3.3.2 Double U-Net GPU profiling

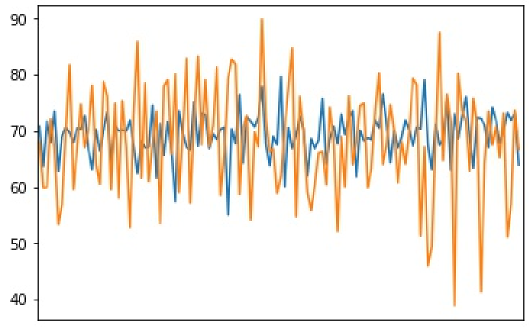

(o) Training GPU usage (%)

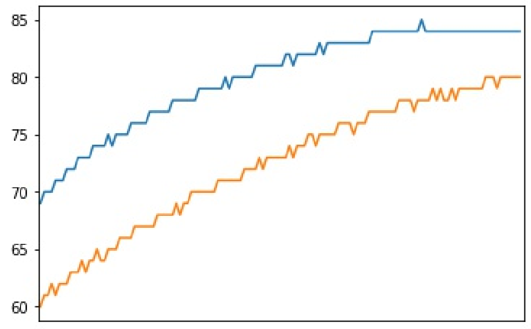

(p) Evaluation GPU usage (%)


(q) Training memory usage (%)


(r) Evaluation memory usage (%)


(s) Training power consumption (W)


(t) Evaluation power consumption (W)


(u) Training GPU temperature (°C)

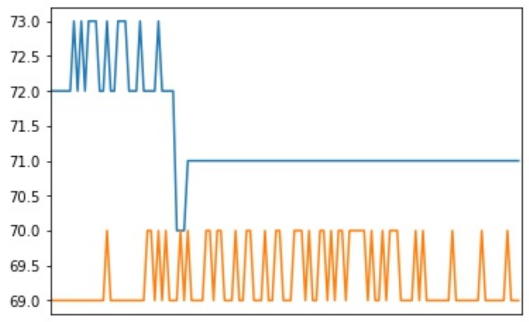

(v) Evaluation GPU temperature (°C)

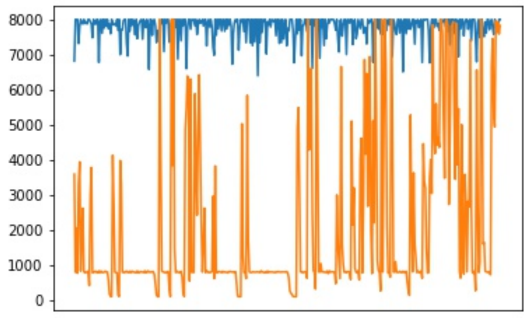
FIGURE 3.3.2: Double U-Net GPU profile

The profile was done under the same conditions than 3.2.2 and the results extracted are totally analogous to the ones presented for the U-Net case, with the exception of the memory usage, where the Double U-Net requires a little more memory (less than 300MB in average). This reaffirms our previous conclusions about the comparison of PyEDDL and TF so for avoiding repetition we remit to 3.2.2 and we won't further discuss it.

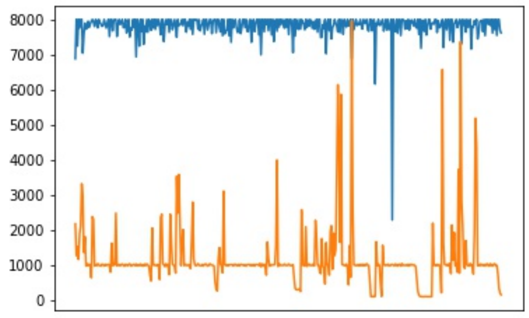
3.3.3 Double U-Net CPU profiling

Double U-Net CPU profiling was done under the exact same conditions than 3.2.3 the results we obtained are consistent with the U-net profile. We again see huge differences in the ratio memory-cpu usage, being PyEDDL again unable to distribute the training or evaluation process in a more efficient way to take advantage of all hardware resources.

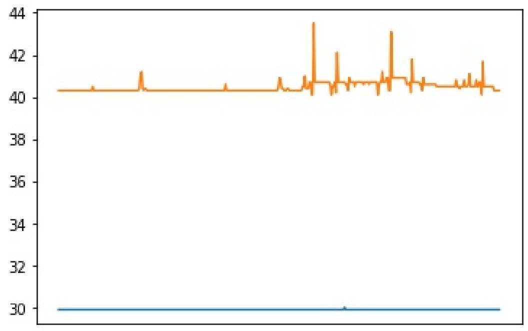
The CPU utilization is the same as in the U-Net, with the PyEDDL model around the compute capacity of 10 CPUs but, in this case, differences in memory usage between the U-Net and the Double U-Net model in the PyEDDL model are significant, around 9GB more in the Double U-Net. As this is not the case in the TF library we can't conclude that Double U-Net is a more memory-consuming model itself.



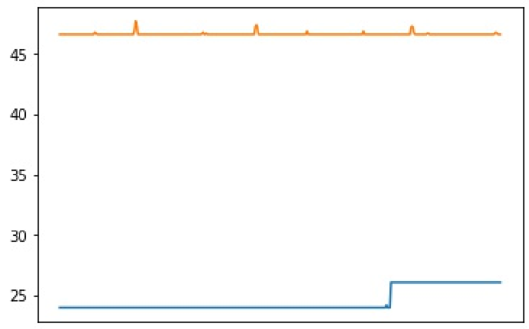
(a) Training CPU usage (%)



(b) Evaluation CPU usage (%)



(c) Training memory usage (GB)



(d) Evaluation memory usage (GB)

3.4 Comparison

We provide the result of our experiments so that both models and libraries can be compared. Given the amount of non-deterministic factors involved, as the weights initialization and the intrinsic randomness of the stochastic optimization processes, we encourage anybody with the intention of using this data to replicate several times the experiment to obtain more accurate values.

U-NET	**PyEDDL**	**Tensorflow**
Trainable parameters	31,060,241	31,031,685
Non-trainable parameters	0	0
Epoch training time (CPU)	≈ 10h	≈ 32min
Evaluation time (CPU)	≈ 70min	≈ 1min
Best dice score in training	0.81	0.93
Best dice score in validation	0.76	0.82
GPU utilization in training (avg %)	97.30	98.54
GPU memory utilization in training (avg %)	49.84	57.86
CPU utilization in training (avg %)	1791.22	7834.15
CPU memory utilization in training (avg GB)	33.39	30.52
GPU utilization in validation (avg %)	92.74	96.26
GPU memory utilization in validation (avg %)	41.01	58.31
CPU utilization in validation (avg %)	1012.64	7795.93
CPU memory utilization in validation (avg GB)	37.91	25.16

DOUBLE U-NET	**PyEDDL**	**Tensorflow**
Trainable parameters	29,320,226	29,288,546
Non-trainable parameters	7,296	7,296
Epoch training time (CPU)	≈ 13h	≈ 46min
Evaluation time (CPU)	≈ 85min	≈ 1.5min
Best dice score in training	0.88	0.92
Best dice score in validation	0.75	0.82
GPU utilization in training (avg %)	93.67	98.22
GPU memory utilization in training (avg %)	39.50	62.21
CPU utilization in training (avg %)	2000.80	7778.40
CPU memory utilization in training (avg GB)	40.43	29.90
GPU utilization in validation (avg %)	96.65	96.62
GPU memory utilization in validation (avg %)	45.08	63.72
CPU utilization in validation (avg %)	1134.80	7810.52
CPU memory utilization in validation (avg GB)	46.62	24.63

Chapter 4

3-D Convolutional Nerual Networks

Finally, with the release of PyEDDL version 1.0.0 less than a month ago, which supports 3-D CNN, we were able to test this approach to the MS lesion segmentation problem. The model implemented was introduced in 2017 in [4] and consists on two 3-D CNN models in cascade.

4.1 Data preprocessing

For this model we use *FLAIR*, *T1* and *T2* modalities of the MRI images. First, the intensities of the images are normalized in order to speed up training and then for each of these we compute 3-D axial patches of size 11, centered on the voxel of interest and, obviously, keeping the correspondence voxel-mask label. The selection of the patch size is again the dilemma of losing context information or increasing data and model complexity and training and evaluation times. During preprocessing patches in which the voxel of interest has intensity lower than 0.5 in the *FLAIR* modality are removed.

Data agumentation is also performed by applying rotations to the patches, however, it's applied at batch time so it doesn't require any preprocessing. After the previous steps, the input is therefore a set of 3-D patches with three different channels corresponding the three modalities and size $(11 \times 11 \times 11)$.

Note that we implemented the data preprocessing manually, however, the second main component of the DeepHealth framework, the ECVL library, is intended to support common operations for biomedical images processing.

4.2 Cascade model

The idea of the cascade model is that the output of the first model if used to decide the input of the second one. This way the second model can process and debug those patches (actually their output via the first model) in which the first model is known to be less accurate. The first model is trained to be more sensitive to detect possible candidate lesion voxels and the second model is trained to reduce the number of misclassified voxels coming from the first one.

The structure of the cascade model is shown below, for all the details as well as the structure of the 3-CNN models themselves we refer to [4].

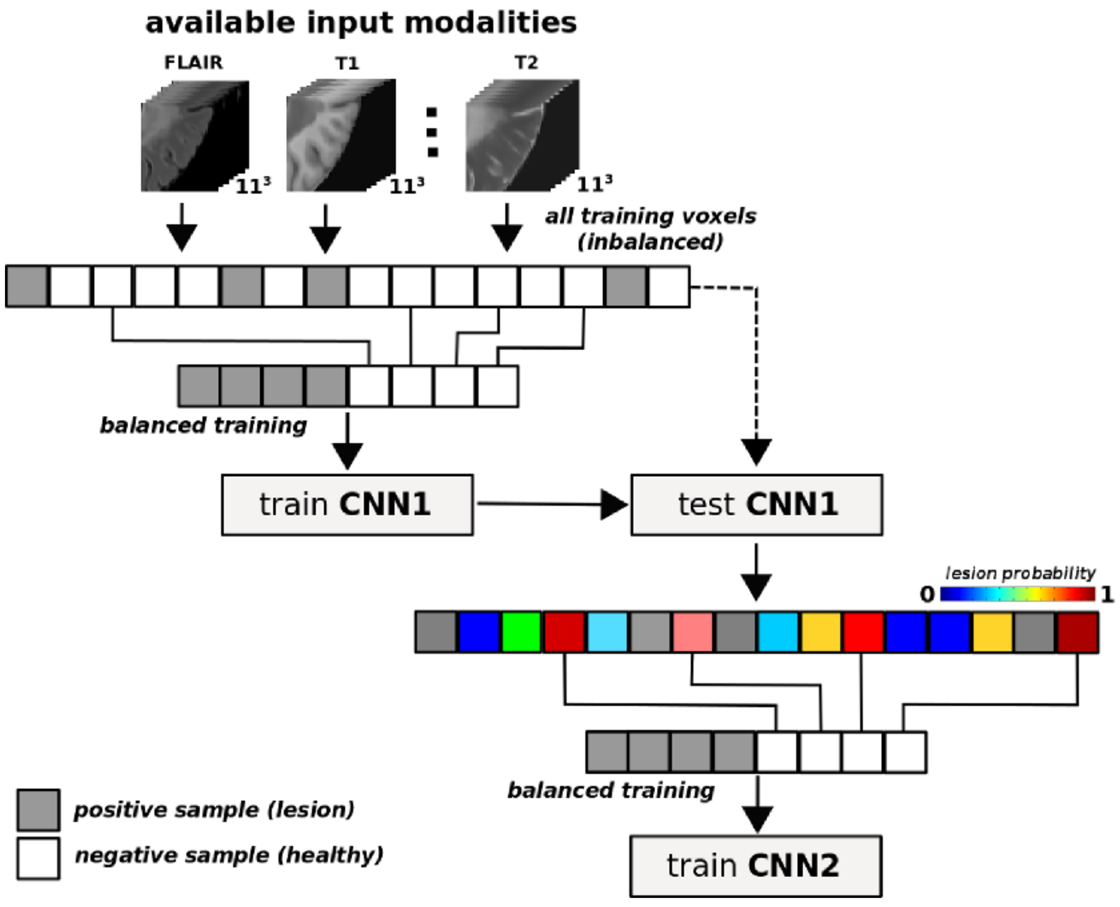


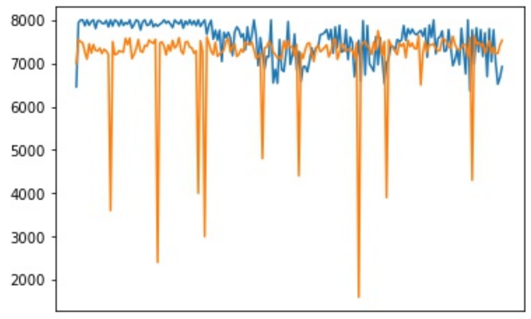
FIGURE 4.2.1: 3-D Convolutional Neural Network cascade model [4]

4.2.1 Cascade model CPU profiling

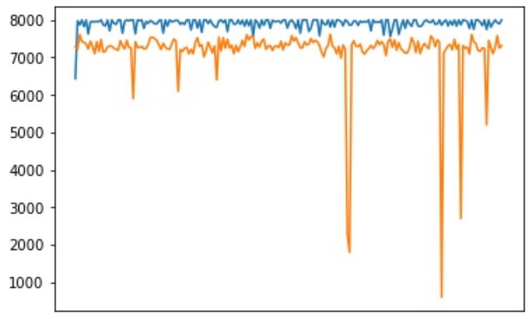
The profiling was done under the same conditions as in 3.2.3. We first analyze the training of both CNN models. The profiling shows a similar behaviour in both models, being the second one lighter in terms of memory usage. We can appreciate how the PyEDDL model is able to almost fully use all the CPUs available showing less insatiability than in previous more complex models.

Since both libraries achieve a similar CPU utilization, with less than a 10% more for the TF models, the main differences are in the memory usage. Apparently, this goes against the conclusions obtained in all the previous profiles. In fact, at first it could look like an error in the measurements, however, this memory profile was repeated in several independent executions.

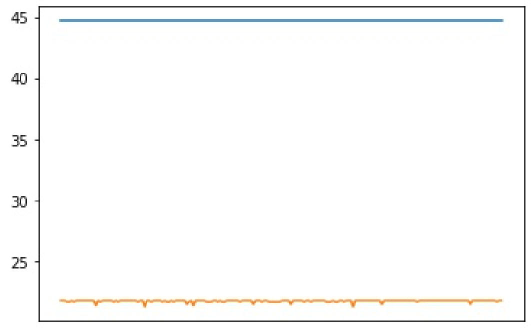
After reviewing the code, we could see that the key behind this huge difference is the usage of *generator functions*. These yield the data when it's required, while the *fit* method in TF loads the whole data passed as parameter. Therefore we conclude that this difference is due to having non-equivalent implementations for both libraries and not due to the libraries themselves.



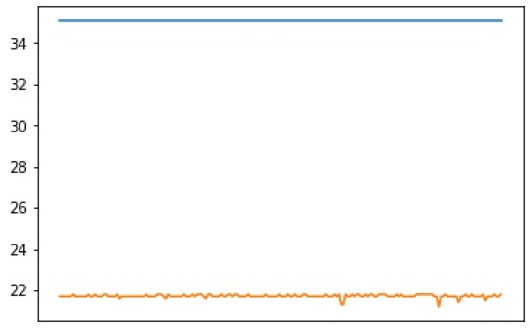
(a) Model 1 training CPU usage (%)



(b) Model 2 training CPU usage (%)



(c) Model 1 training memory usage (GB)

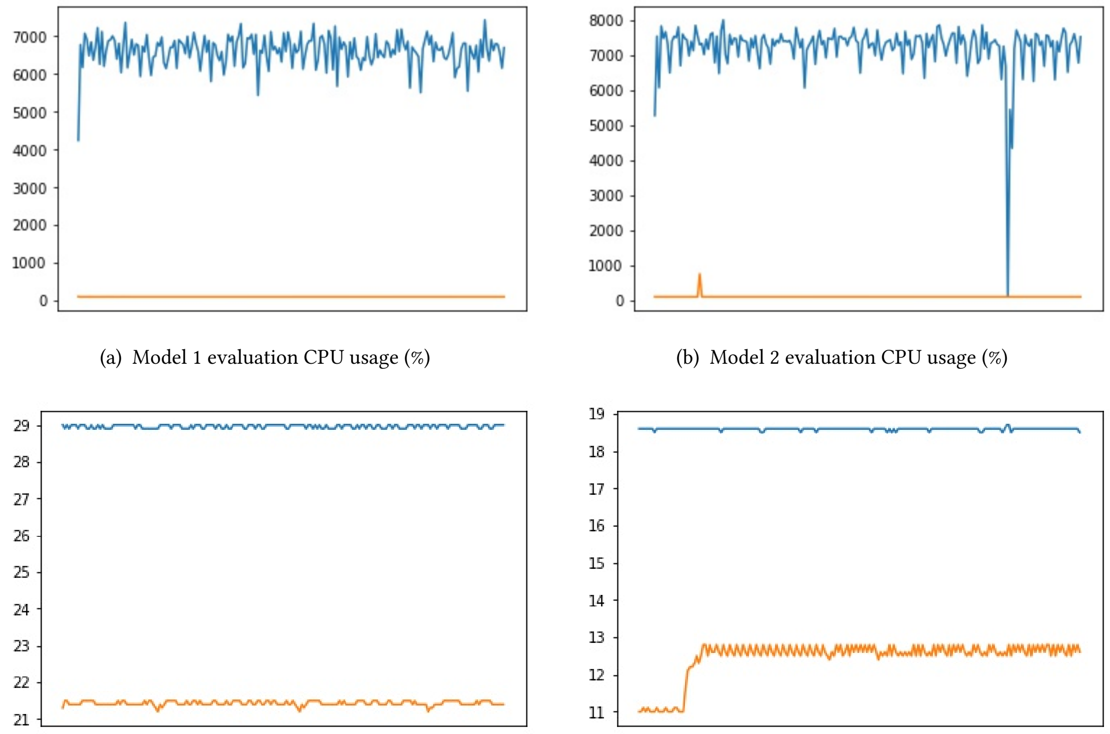


(d) Model 2 training memory usage (GB)

FIGURE 4.2.2: Models training profile

In the case of the models' evaluation we see a very significant difference. Again the memory usage of the PyEDDL model is much lower but now, the ratio cpu-memory usage is much worse in this library than in TF. This is because, while the TF models are able to almost fully utilize all the CPUs, PyEDDL is using only one out of the 80 availables. We suppose this is because in PyEDDL 1.0.0 3-D CNN computations can not be parallelized since these have been just added to the library.

It's clear that this is a major drawback of this library. It increases severely the evaluation time, in our case up to seventy times more but we can't forget that it's a library currently in development, so it's more a drawback of the version.



(a) Model 1 evaluation CPU usage (%)

(b) Model 2 evaluation CPU usage (%)

(c) Model 1 evaluation memory usage (GB)

(d) Model 2 evaluation memory usage (GB)

FIGURE 4.2.3: Models evaluation profile

Finally we have to add that we couldn't run this cascade model on GPU (and therefore we couldn't profile it) because due to an inner bug in the library, we were receiving a *segmentation fault*.

Chapter 5

Conclusions

We can extract two main conclusions of this project. The first one is that EDDL is a library with a lower performance with respect to TF in computing resources management, memory usage, execution time, model's learning progress.... However, it's also remarkable that EDDL is a new library still in development which has proven to support complex network architectures as the Double U-Net or 3-D CNN and is able to achieve good results, being the difference with TF in simpler models almost inappreciable.

The second main conclusion is that CNN have proven to be a useful tool for biomedical image segmentation and, in our case, for Multiple Sclerosis lesion segmentation. We are confident that they will play an important role in the future in medicine.

Finally, when analyzing our CNN, we have also seen that Double U-Net doesn't present any remarkable improvement to U-Net when applied to MS lesion detection.

Bibliography

[1] Olivier Commowick, Frédéric Cervenansky, and Roxana Ameli. Msseg challenge proceedings: Multiple sclerosis lesions segmentation challenge using a data management and processing infrastructure. 2016.

[2] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. 2015.

[3] Debesh Jha, Michael A. Riegler, Dag Johansen, Pål Halvorsen, and Håvard D. Johansen. Doubleu-net: A deep convolutional neural network for medical image segmentation. 2015.

[4] Sergi Valverde, Mariano Cabezas, Eloy Roura, Sandra González-Villà, Deborah Pareto, Joan-Carles Vilanova, LLuís Ramió-Torrentà, Àlex Rovira, Arnau Oliver, and Xavier Lladó. Improving automated multiple sclerosis lesion segmentation with a cascaded 3d convolutional neural network approach. 2017.

[5] Sergi Valverde, Mostafa Salem, Mariano Cabezas, Deborah Pareto, Joan C. Vilanova, Lluís Ramió-Torrentà, Àlex Rovira, Joaquim Salvi, Arnau Oliver, and Xavier Lladó. One-shot domain adaptation in multiple sclerosis lesion segmentation using convolutional neural networks. 2018.

[6] Sergi Valverde, Mariano Cabezas, Eloy Roura, Sandra González-Villà, Joaquim Salvi, Arnau Oliver, , and Xavier Lladó. Multiple sclerosis lesion detection and segmentation using a convolutional neural network of 3d patches. 2016.