UNIVERSITÉ DE
SHERBROOKE

Master's degree in Computer Science

IFT780 : Neural Networks

Practical work 3 : Cardiac Segmentation

Authors :

Bérenger Vuittenez *vuib2601*
Samya Souaf *sous3502*
Thibaud Merieux *mert1702*

Teacher :

Antoine Théberge

1 Introduction

This document presents the work we have carried out as part of Practical Work 3 : Cardiac Segmentation. It is associated with a set of programs, available on *turnin*. Information on the use of the programs can be found in the *README.md* file. Additionally, a Jupyter Notebook is provided to run all the commands in this report in the file.*test.ipynb*.

2 Architectures

2.1 ENet

2.1.1 Model presentation and justification

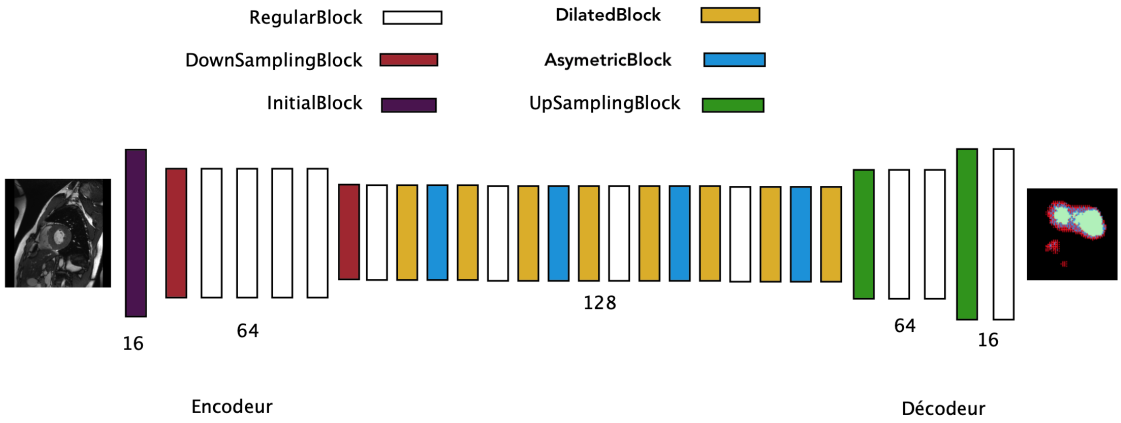


FIGURE 1 – Used ENet structure

As indicated in the course, the *ENet* architecture and its improved versions are widely used in medical imaging.

ENet (for *Efficient Network*) is described by its creators as being much more efficient as it is designed to perform real-time segmentation. [1] ENet is also designed around the concept of encoding, which allows for an increased receptive field of the convolution networks without significantly increasing the number of parameters.

2.1.2 Hyperparameter tuning

Here, we focused on searching for hyperparameters that have a highly visible influence on the network's performance :

— the learning rate
— the momentum
— the dropout probability

Pour lancer la recherche d'hyperparamètres, il suffit de saisir la commande suivante :

```
$ !python train.py acdc_train --model=ENet --dataset='./02Heart.hdf5'
    --hypersearch --num-epochs=5
```

The results obtained show that over 5 epochs, the accuracy remains fairly low when using a low learning rate, as can be seen in Figure 2 and Figure 3. Furthermore, it can be seen that the results are generally worse when the dropout rate is too high (0.5 for Figure 2).

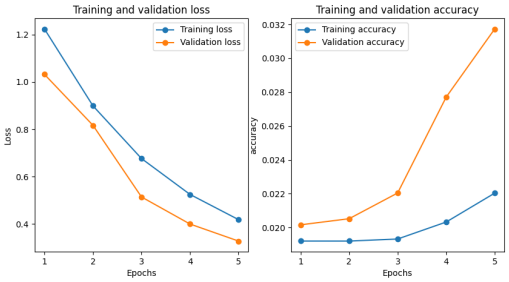


FIGURE 2 – Results with ENet (Learning rate = 0.0001, momentum = 0.6, dropout=0.5)

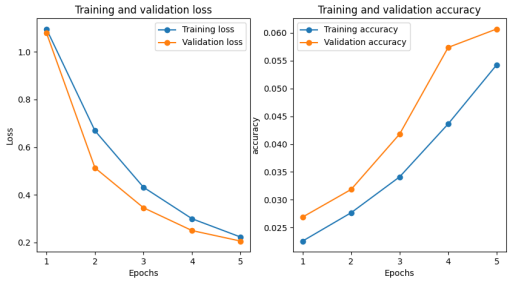


FIGURE 3 – Results with ENet (Learning rate = 0.0001, momentum = 0.6, dropout=0.01)

However, as can be seen in Figure 4, increasing the learning rate significantly increases performance (at least during the first 5 epochs, it is possible that a too high learning rate prevents us from reaching local minima).

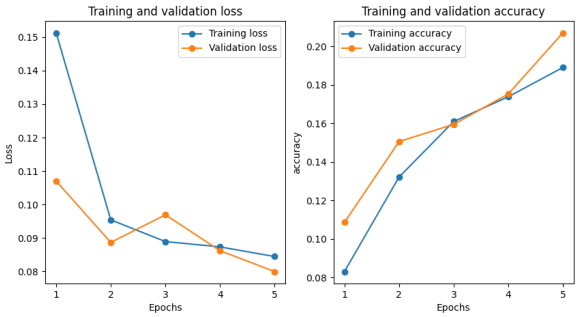


FIGURE 4 – Results with ENet (Learning rate = 0.01, momentum = 0.9, dropout=0.1)

2.1.3 Results

Thus, by training for 10 epochs with the best hyperparameter values using the following command, one can obtain the performance indicated in Figure 5.

```
$ !python train.py acdc_train --model=ENet --lr=0.01 --momentum=0.9
    --dropout=0.1 --dataset='./02Heart.hdf5' --num-epochs=10
    --batch_size=5
```

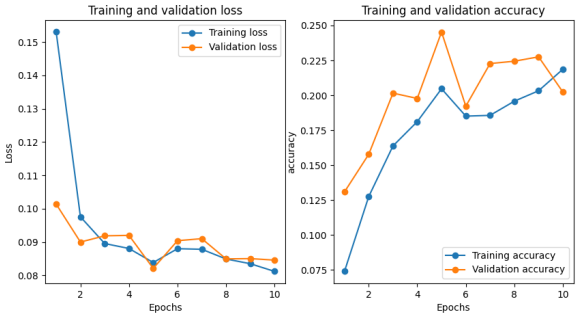


FIGURE 5 – Results with ENet (Learning rate = 0.01, momentum = 0.9, dropout=0.1)

An example of the obtained segmentation can be seen in Figure 6. As can be seen, the area is quite good but the colors are still random. The results are therefore quite disappointing, perhaps there is an error in our implementation of the *ENet*.

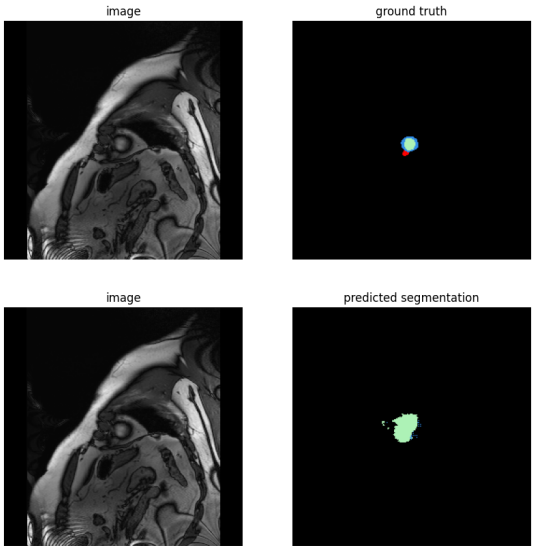


FIGURE 6 – Example of segmentation with ENet (Learning rate = 0.01, momentum = 0.9, dropout=0.1)

2.2 GridNet

2.2.1 Model presentation and justification

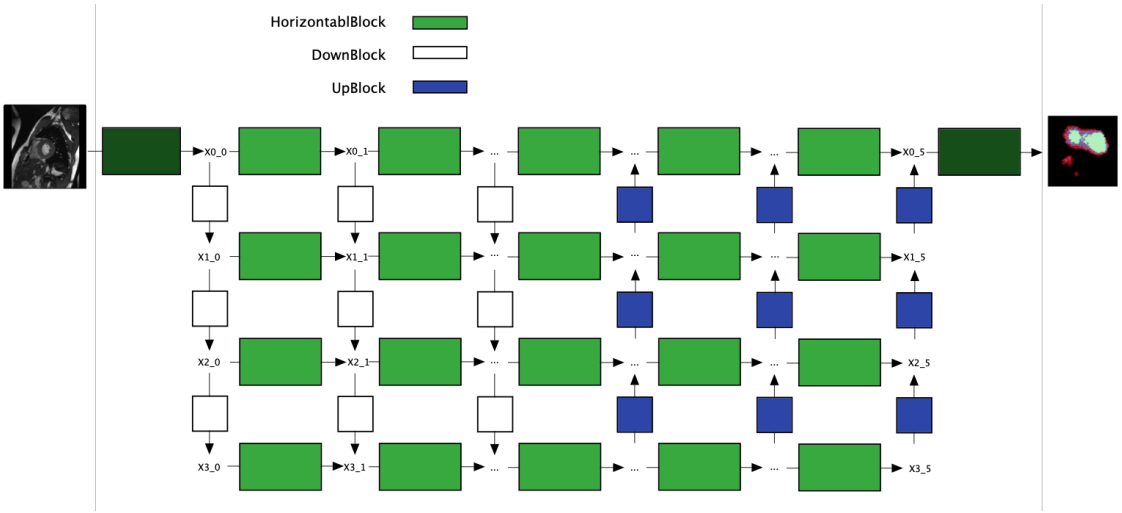


FIGURE 7 – GridNet architecture used

As indicated by the authors of the article that originated *GridNet* [2], this one combines the interests of many other architectures. This can be explained by the paths that the information can travel. In fact, the network can be both a *UNet*, a *Fully-Resolution residual* and a *Fully-Convolutional network*, this is visually translated by Figure 8.

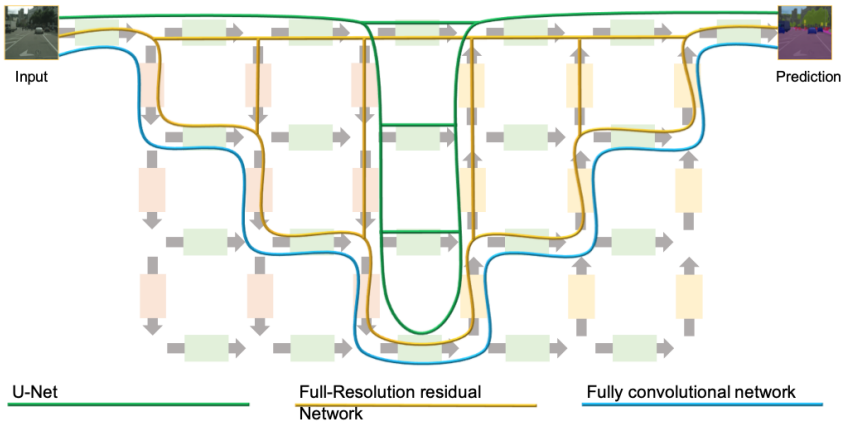


FIGURE 8 – Possible paths in GridNet

The blocks indicated in figure 7 are of 4 different types :

— The *HorizontalBlocks*, which perform 2 convolutions that preserve the number of input channels.

— The *DownBlocks*, which decrease the activation map dimension while increasing the number of channels (an operation that increases the receptive field).

— The *UpBlocks*, which increase the activation map dimension while decreasing the number of channels.

— The initialization and finalization layers, which are special *HorizontalBlocks* that modify the number of channels.

At each node, the incoming arrows are summed and constitute the information for the following arcs.

2.2.2 Hyperparameter Search

The architecture of this network has interesting hyperparameters : the number of columns and the number of rows in the grid. However, this architecture is quite complex to set up, especially dynamically, so these hyperparameters are not part of the automated search directly from the command line.

On the other hand, it is still possible to study the influence of some general hyperparameters :

— learning rate

— momentum

To start the hyperparameter search, simply enter the following command :

```
$ !python train.py acdc_train --model=GridNet
    --dataset='./02Heart.hdf5' --hypersearch --num-epochs=5
```

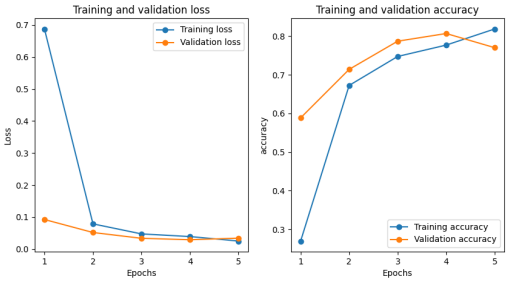


FIGURE 9 – Results with GridNet (Learning rate = 0.0001, momentum = 0.3, dropout=0.1)

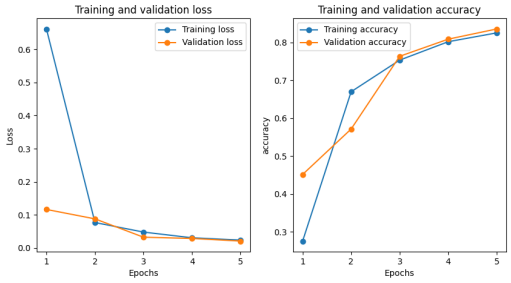


FIGURE 10 – Results with GridNet (Learning rate = 0.0001, momentum = 0.6, dropout=0.01)

The results obtained show that the accuracy is generally around 80As can also be seen, increasing the momentum means worse results for the first 3 epochs, but a more stable trend for the rest.

2.2.3 Results

The results obtained with the *GridNet* are really very good. As can be seen in Figure 11, the errors made are minimal.

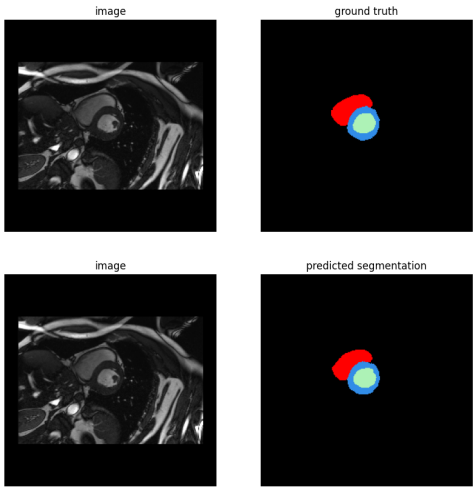


FIGURE 11 – GridNet segmentation example (Learning rate = 0.0001, momentum = 0.6) entraîné sur 10 epochs

2.2.4 Going Further

To improve the performance of the *GridNet*, the article discusses implementing random dropout for certain paths in the grid. Due to time constraints, we did not have the time to make this improvement to our architecture.

2.3 GrEdNet

2.3.1 Model Presentation and Justification

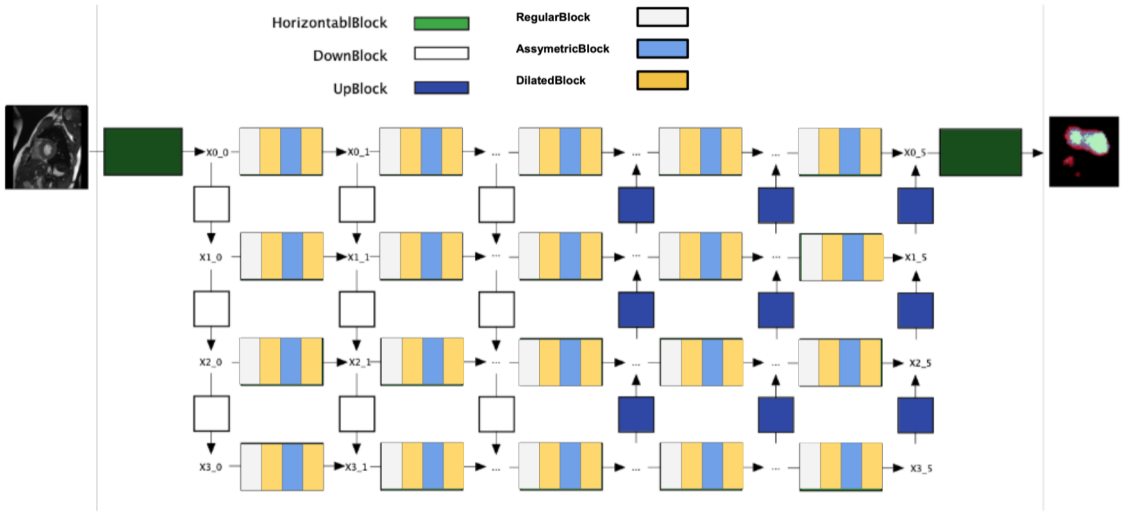


FIGURE 12 – Scheme of the GrEdNet architecture used

We chose to combine the previous two models while retaining the general architecture of the GridNet.

The combination of the two models is such that each *horizontal* block becomes a block inspired by ENet.

2.3.2 Hyperparameter Search

The large number of different layers in the network requires even more than the hyperparameter search of the GridNet to study only a few different configurations. We then retain as previously :

— the learning rate

— the momentum

To start the hyperparameter search, simply enter the following command :

```
$ !python train.py acdc_train --model=GridNetCustom
    --dataset='./02Heart.hdf5' --hypersearch --num-epochs=5
```

We can also question the influence of a modification of the number of rows and columns. With 6 rows and 6 columns, the results are as follows :

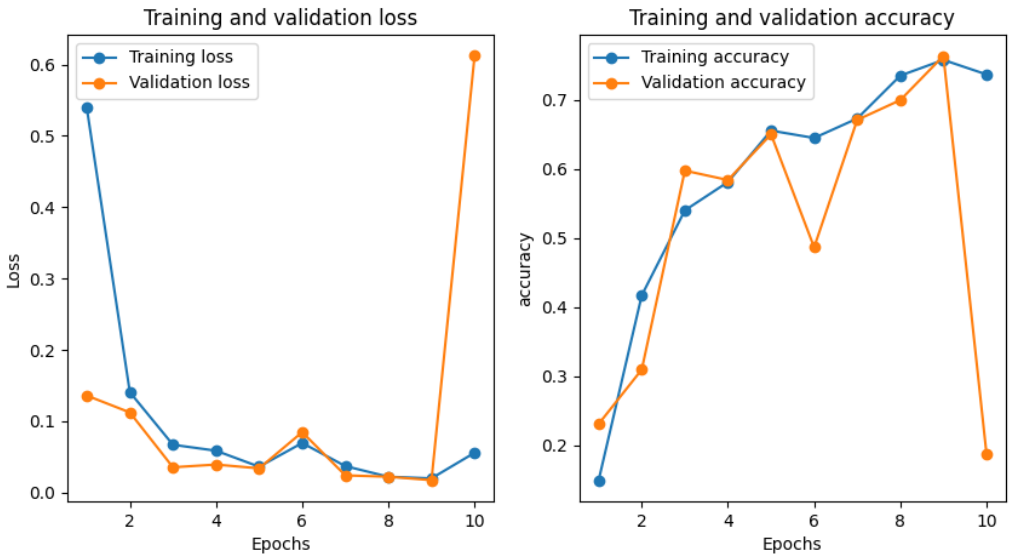


FIGURE 13 – GrEdNet learning curves with 6 rows and 6 columns (Learning rate = 0.0001, momentum = 0.9, dropout = 0.1) trained over 10 epochs

In this curve, we see the major problem with this architecture : largely due to its complexity, CUDA memory problems arise and the only applicable workaround is to limit the size of each data batch to 2 in our case. This limitation has a high price, which is to establish a high variability in the results (in addition to a very long training due to the much more numerous backpropagations), something that is quite well seen in the graph.

The training is here too long and on batch sizes too small to be considered.

2.3.3 Results

The variability of the model's results, as previously explained, is still present, although partly limited by a batch size of 2, the maximum possible according to the machines used.

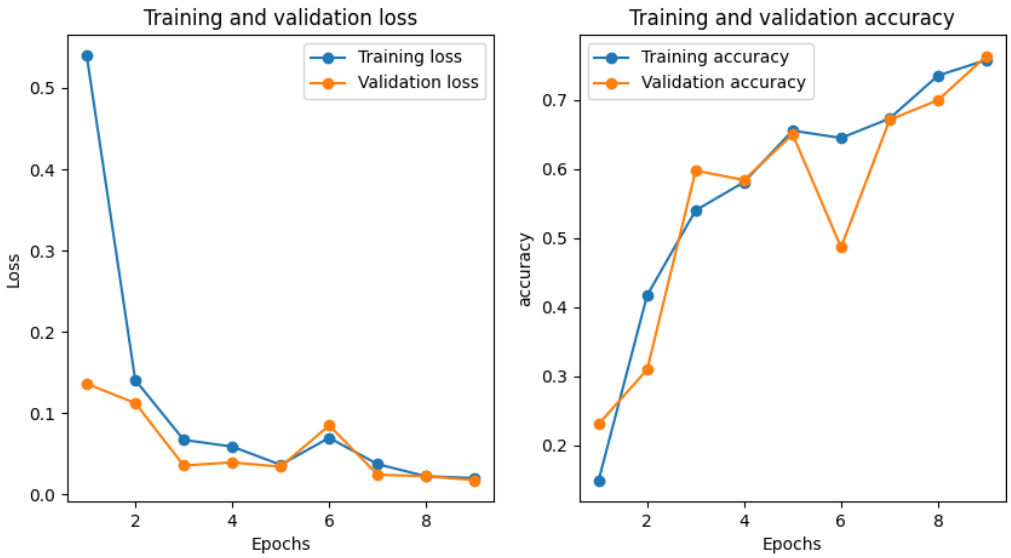


FIGURE 14 – Learning curves of GrEdNet (Learning rate = 0.0001, momentum = 0.9, dropout = 0.1) trained for 10 epochs

It is also noted at the end that the accuracy of 75

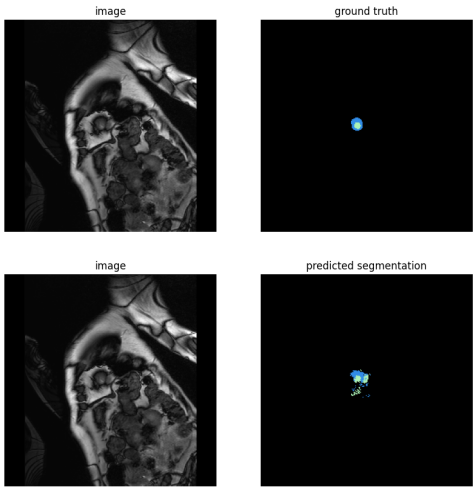


FIGURE 15 – Example of segmentation with GrEdNet (Learning rate = 0.0001, momentum = 0.9, dropout = 0.1) trained for 10 epochs

3 Hyperparameter Search

Hyperparameter search is performed directly in the *train.py* file. To do this, if the mode chosen is "Hyperparameter Search", i.e., if the user entered *–hypersearch* in their command, we first determine which hyperparameters are relevant to vary for the chosen model.
Subsequently, training is carried out with different combinations of chosen parameters and information is displayed on the screen.
The selection of the best hyperparameter combinations is then made manually by consulting the different accuracy values obtained.

4 Checkpointing

4.1 Saving

Checkpointing is implemented directly within the *TrainTestManager.py* file. It occurs at the end of each epoch by updating the *Modelname.pt* file, where all information related to the model is stored.

4.2 Loading

If the user uses the *–resume* option in their command line, then the program resumes the training of the *Modelname.pt* file by loading several information related to the model :

— the model's weights
— the state of the optimizer
— the number of already processed epochs

4.3 Further exploration

To further explore this feature, it would be useful to store all information related to the tracing of curves. This would make it possible to resume training while allowing to display the complete history of loss and accuracy.

5 Data augmentation

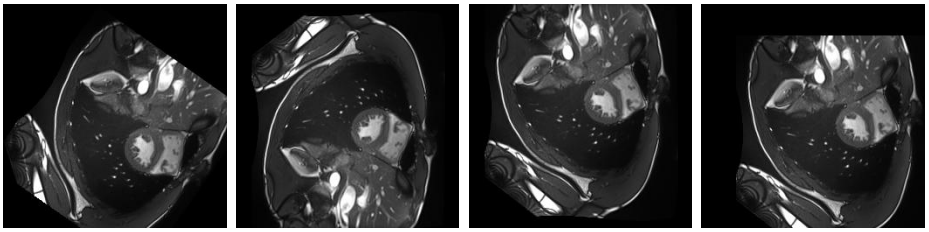
5.1 Retained transformations

We have implemented 4 types of data augmentation :

— random rotation (from 0 to 90 degrees)

— random noise (white noise added to the entire image)
— random crop (resizing of the image)
— mirror along multiple axes.

Examples of these transformations are available in Figure 16. Note here that we chose to use a value of 0.4 for the probability of performing each of these transformations. The image can therefore undergo several of these transformations combined.



(a) Rotation (b) Mirroir axe 1 (c) Mirroir axe 2 (d) Crop

FIGURE 16 – Exemple de transformations obtenues

5.2 Encountered difficulties

Some of these transformations, like random noise, should not be applied to the labeled image, while for others (mirror, rotation, and crop), it is essential.

Additionally, at first, we did not apply the same transformation to the labeled image as the original image due to internal random variables. Reversing the random values and passing them as parameters to the following functions allowed us to overcome this problem.

5.3 Performance gains

5.3.1 CNNet

Initially, we trained the provided *CNNet* with and without using *data augmentation* to visualize performance differences. The results are thus available in Figure 19. As can be seen, over the first 10 epochs, data augmentation provides very interesting performance gains, but exposes to greater instability over the epochs (this is due to the random nature of the transformations).

5.3.2 GridNet

Ensuite, il nous a paru intéressant de voir quel gain pouvait apporter la data-augmentation au modèle avec lequel nous obtenions les meilleurs résultats : *Grid-*

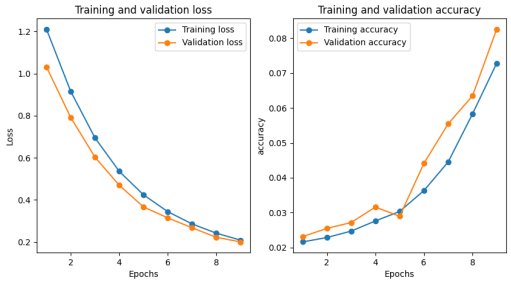


FIGURE 17 – Résults with CNNet sans data-augmentation

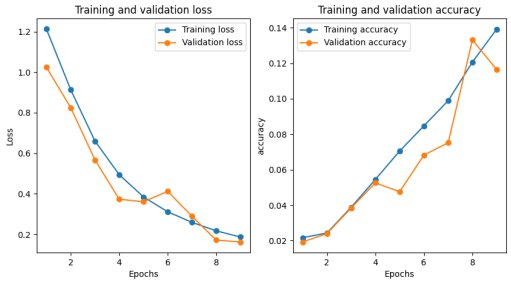


FIGURE 18 – Résults with CNNet avec data-augmentation

FIGURE 19 – Comparison of résults with and without data-augmentation (CNNet)

Net. Ainsi, une comparaison similaire est disponible pour ce réseau à la figure 22. Comme on peut le voir, les gains obtenus grâce à la data-augmentation sont plus discrets ici, voir inexistants. On note même un léger recul de la précision. Cependant, on peut supposer qu'en entraînant sur de plus nombreuses epochs, le gain serait bien présent.

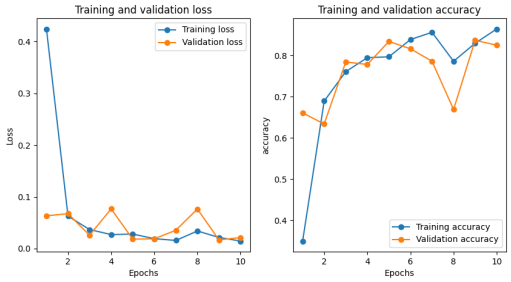


FIGURE 20 – Résults with GridNet without data-augmentation

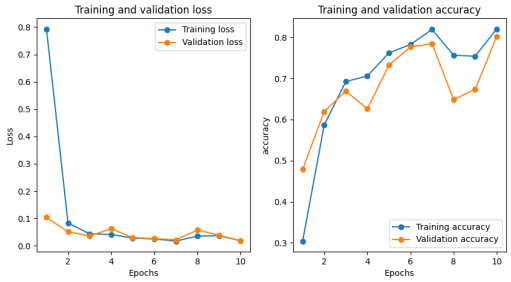


FIGURE 21 – Résults with GridNet with data-augmentation

FIGURE 22 – Comparaison des résultats avec et sans data-augmentation (GridNet)

5.3.3 GrEdNet

The objective of studying precision gains with GrEdNet is to determine whether transforming each horizontal block, thereby adding more parameters overall, requires increasing the dataset, or if GrEdNet's limited ability for data augmentation does not allow for improved performance.

As with GridNet, the performance is very similar with and without data augmentation.

The real issue with this architecture is its high complexity, requiring the use of very small batches and resulting in a training time that is 2 to 3 times longer than simple training of the GridNet.

FIGURE 23 – Results with GrEdNet without data-augmentation

FIGURE 24 – Results with GrEdNet with data-augmentation

FIGURE 25 – Comparison of results with and without data-augmentation (GrEdNet)

6 Conclusion

To conclude, while our ENet doesn't seem to live up to the expectations we had for it, the GridNet appears to be achieving very good results for this segmentation task. The hybrid of the two models does not seem to lead to any improvement in performance.

Data augmentation does not also result in significant performance gains in the early epochs, with the exception of the CNNet.

Références

[1] Sangpil Kim Eugenio Culurciello Adam Paszke Abhishek Chaurasia. "ENet : A Deep Neural Network Architecture for Real-Time Semantic Segmentation". In : *arXiv* (2017).

[2] Elisa Fromont Damien Muselet Alain Tremeau Christian Wolf Damien Fourrure Rémi Emonet. "Residual Conv-Deconv Grid Network for Semantic Segmentation". In : *arXiv* (2017).