E4 project : Maximin Affinity Learning of Image Segmentation

Quentin Garrido, Tiphanie Lamy Verdin, Josselin Lefèvre, Annie Lim

January 2020

Contents

1 Introduction **3**

2 Maximin Affinity learning **4**
2.1 Method's presentation . . . 4
2.2 Optimizing the Rand Index . . . 4
2.3 Maximin affinity and the maximin edge . . . 5
2.4 Computing an affinity graph . . . 5
2.5 Training a classifier . . . 6

3 Implementation for the original method **7**
3.1 Inputs and outputs of the NN . . . 7
3.2 Computing the maximin edge . . . 7
3.3 Interaction between Higra and PyTorch . . . 8

4 Results for the original method **10**
4.1 CREMI . . . 10
4.2 ISBI 2012 . . . 13

5 Improvements to the method **15**
5.1 Using a more potent architecture . . . 15
5.2 Constrained MALIS loss . . . 15
5.3 Two pass computation of the loss . . . 17
5.4 Seeded watershed as post processing . . . 18

6 Implementation for the improved method **19**
6.1 Neural network used . . . 19
6.2 Use of a BPT to compute the loss . . . 19
6.3 An improved way to generate edge weights . . . 19
6.3.1 Weighting edges with Higra . . . 19
6.3.2 Higra's edge weights structure . . . 21
6.3.3 Our method to generate edge weights . . . 21
6.3.4 Backpropagation . . . 22

7 Results for the improved method **23**

8 Teamwork **23**
8.1 Team organization . . . 23
8.2 Task distribution . . . 23
8.3 Obstacles and overcoming them . . . 26

9 Conclusion **26**

1 Introduction

The goal of this project is to implement Maximin Affinity Learning of Image Segmentation (or MALIS for short) as introduced in [1]. It is a method used to obtain an image segmentation that is mainly used on medical imagery.

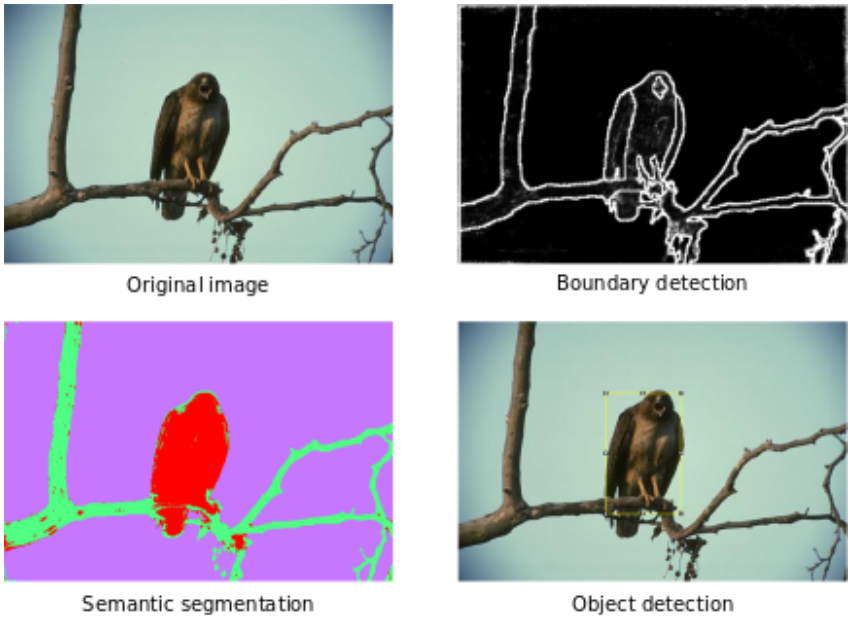


Figure 1: Illustration of image segmentation, from ImageJ

As we can see in figure 1 there are multiple ways to produce an image segmentation. In the top right corner we can see a boundary detection of the original image, where borders between objects are in white and objects are in black. Here we only get the separation between objects but no information on the objects/segments themselves.
In the bottom left corner, we have an example of semantic segmentation of our image. Here the idea is to classify every pixels in predefined classes (here red is the bird, green the tree and purple the background). We then obtain a segmentation with different information, which is not necessarily better than boundary detection since both solve different problems.
On the bottom right corner we have an example of object detection. Even though it gives us a rough estimate of the structure of the image and the objects present in it, it is not considered a segmentation and is a whole other class of algorithms.

In the case of MALIS, we will try to obtain a boundary detection, or what we could call an edge based segmentation. There are various ways to generate an image segmentation of this kind, the simplest one being an algorihm such as Canny's algorithm, or more advanced methods such as watersheds or more recently neural networks based approaches. The idea with MALIS is to optimize directly a measure of segmentation quality, which had not been done before. This is a really interesting idea since segmentations are always evaluated using various metrics (Rand Index, Variation Of Information ...) and optimizing one directly could lead to better results, at least for this metric.

Our goal in this project is to implement the original MALIS paper [1] and also it's improvement in [2].

In this report, we will first describe how MALIS works in more details. We will then look at how we implemented it and some implementation challenges that we faced. We will then look at the results we

have gotten so far, by implementing the original method. Afterwards, we will look at how the method was improved and what we will try to implement in the coming semester. Finally we will take a look at how we worked as a team throughout this project.

2 Maximin Affinity learning

2.1 Method's presentation

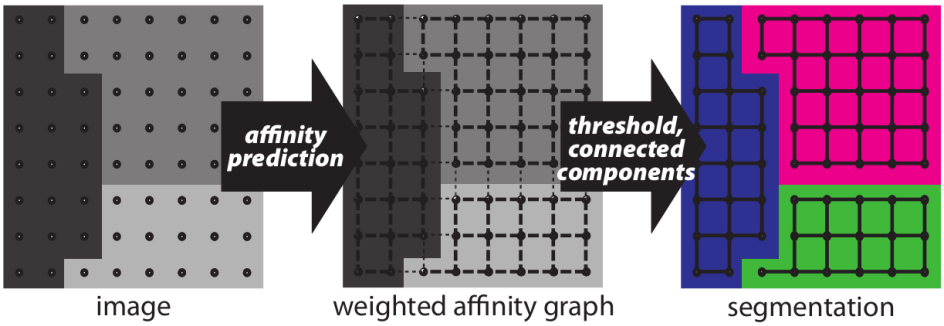


Figure 2: Description of the method, from [1]

The method, as illustrated in figure 2 works in two steps.
The first step is to compute an affinity graph G, with nodes V representing each pixel in the original image and edges E weighted by the affinity between neighbouring pixels. The graph G will be 4-connected and for every pair of pixels $i, j \in V \times V$ there exist and edge $(i, j) \in E$ if and only if i and j are neighbours. We will note the affinity between neigbouring pixels A_{ij}.
An affinity of 1 between i and j means that they belong to the same object, and an affinity of 0 means that they belong to different objects.

Once we have our affinity graph, with affinities between 0 and 1, we will threshold it and remove edges under a certain affinity to obtain connected components that will be our objects, and this thresholded affinity graph will be our final segmentation.

2.2 Optimizing the Rand Index

There exist various methods of evaluating an image segmentation, one of them being the Rand Index which is defined as follow :

$$1 - RI(\hat{S}, S) = \binom{N}{2}^{-1} \sum_{i<j} |\delta(s_i, s_j) - \delta(\hat{s}_i, \hat{s}_j)|$$

With S our groundtruth segmentation $\hat{S}$ our predicted segmentation and $\delta(s_i, s_j)$ the indicator function taking value 1 if $s_i = s_j$ (if pixels i and j are in the same segment/object) and 0 otherwise.
Intuitively, this can be seen as the fraction of image pixels where both segmentations agree.
This gives us a way to evaluate an image segmentation that penalizes when two objects are merged or split in our final segmentation as we can see in figure 3. On the left, only one edge is misclassified but this merges two objects and will be heavily penalized by the Rand Index (we would see the same results if an object was split). On the contrary if we have misclassified edges inside of objects that do not create any splits, this will not be penalized at all by the loss since our objects are still correctly delimited.

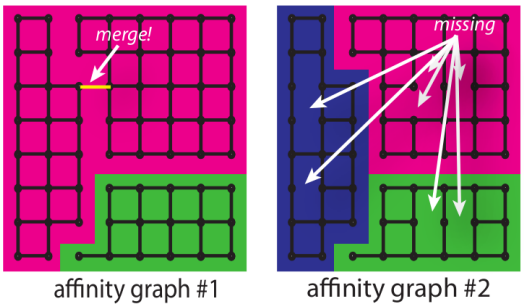


Figure 3: Example image segmentations and their affinity graph. On the left we have a merging of two objects due to a misclassified edge and on the right we have edges that should be of value 1 that have been removed. From [1]

As such the Rand Index seems a good measure of segmentation quality for our problem (edge-based segmentation). We will have to see how exactly we can optimize this metric in our case.

2.3 Maximin affinity and the maximin edge

In order to optmize the Rand Index, we must find a way to compute $\delta(\hat{s}_i, \hat{s}_j)$. To do this we will use the concept of maximin affinity, as described in [1].
Let $\mathcal{P}_{ij}$ the set of all paths between i and j in our image. For every path $P \in \mathcal{P}_{ij}$ there is an edge(s) (k, l) with minimal affinity.

This allows us to define the maximin path, which is the path which maximizes the mininal affinity. It is defined as follow :

$$P_{ij}^* = \arg \max_{P \in \mathcal{P}_{ij}} \min_{(k,l) \in P} A_{kl}$$

The edge of minimal affinity in the maximin path will be called the maximin edge and will be written $mm(i, j)$.
It allows to define the maximin affinity which is simply the affinity of the maximin edge.

$$A_{ij}^* = \max_{P \in \mathcal{P}_{ij}} \min_{(k,l) \in P} A_{kl}$$

The most important consequence of this is that a pair of pixels i, j is connected in the thresholded affinity graph if and only if A_{ij}^* is greater than the threshold value, as shown in [1].

This means that if we can find the maximin affinity efficiently, we will be able to compute the Rand Index efficiently.
In practice the maximin affinities can be computed efficiently using a maximum spanning tree (MST), since any path in the MST is a maximin path.

2.4 Computing an affinity graph

As we can see, once we have our affinity graph, obtaining the segmentation is straightforward, but the issue is : How to obtain an affinity graph?
As stated before, an affinity graph is equivalent to the contours in the image, and as such any method to find the contours in an image can be used to find the affinity graph (with relative success).
The first idea that we could have would be to use a contour detector, such as the Canny filter for example, however this would lead to an obvious over segmentation, which is not desirable here.

An other idea (the idea used in [1]) would be to use a neural network to obtain our affinity graph. Neural networks, and particularly convolutional neural networks (CNNs) have proven to be an extremely powerful tool in image processing and especially in image classification and segmentation, which makes them a good candidate for MALIS.

The architecture used originally is a fully convolutional neural network (FCNN) which is a CNN with only convolutional layers. As such they can work with various input sizes which is always a nice feature.

Now that we have seen how we can compute an affinity graph, let's see how we can train this classifier, and if it is even possible to train it with the Rand Index as a loss function.

2.5 Training a classifier

As stated before, the goal of this method is to optimize the Rand Index, defined as:

$$1 - RI(\hat{S}, S) = \binom{N}{2}^{-1} \sum_{i<j} |\delta(s_i, s_j) - \delta(\hat{s}_i, \hat{s}_j)|$$

with S our groundtruth and $\hat{S}$ our predicted segmentation.
We can define $A^*_{ij}(I, \theta)$ the maximin affinity of pixels i and j in the predicted affinity graph of our classifier on I with parameters θ. We can rewrite the previous equation as :

$$1 - RI(I, \theta, S) = \binom{N}{2}^{-1} \sum_{i<j} |\delta(s_i, s_j) - A^*_{ij}(I, \theta)|$$

However this function is not differentiable everywhere so we cannot use it directly for our training. We will replace the absolute value with a smooth loss function l such as the mean squared error. This relaxation will allow us to use gradient descent to train our classifier(refer to [1] for more details on this).

Thus our final loss function will be :

$$L(I, \theta, S) = \binom{N}{2}^{-1} \sum_{i<j} l(\delta(s_i, s_j), A^*_{ij}(I, \theta)) = \binom{N}{2}^{-1} \sum_{i<j} l(\delta(s_i, s_j), A_{mm(i,j)}(I, \theta))$$

We can now look at what the training loop will look like.

Algorithm 1 MALIS training loop, from [1]

1: **repeat**
2: Predict the affinity graph for an image I
3: Pick two random pixels i, j from I
4: Find the maximin edge $mm(i, j)$
5: Update our parameters θ with gradient

$$\nabla l(\delta(s_i, s_j), A_{mm(i,j)}(I, \theta)$$

6: **until** convergence is reached

In practice we won't train on the whole image since it would be too long to compute the maximin edge in the whole image. We will instead train on 21x21 patches of the image. We have to choose these patches carefully as most of them will mostly be in an object and not around a border.
Since borders between objects are far less common than the "inside" of an object, we may most of the time train only for pixels inside a same object.

This would result in class imbalance between our "borders" (affinity of value 0) and or "inside" (affinity of value 1). Training with such imbalance would lead to worsened results since borders would not be present most of the time in the chosen image (one example of such pathological behaviour would be the prediction of affinity 1 everywhere, far from what we desire).

As such when training we need to be careful about what training image we select.

3 Implementation for the original method

3.1 Inputs and outputs of the NN

After reading the original paper [1] we realized that there was some missing information on implementation details. One of the most important points, the input and the output of the neural network, was not clear. After some research we were lucky to find Srinivas Turaga's Phd thesis [3]. In it MALIS was presented in more details, especially the neural network used in the experiments.

The network predicts the affinity following each axis. As we can see in figure 4, in the case where the network is fed a 3D image, it outputs three affinity images following the axis X, Y and Z. Then we have to merge these three images in a way to obtain the complete affinity graph.

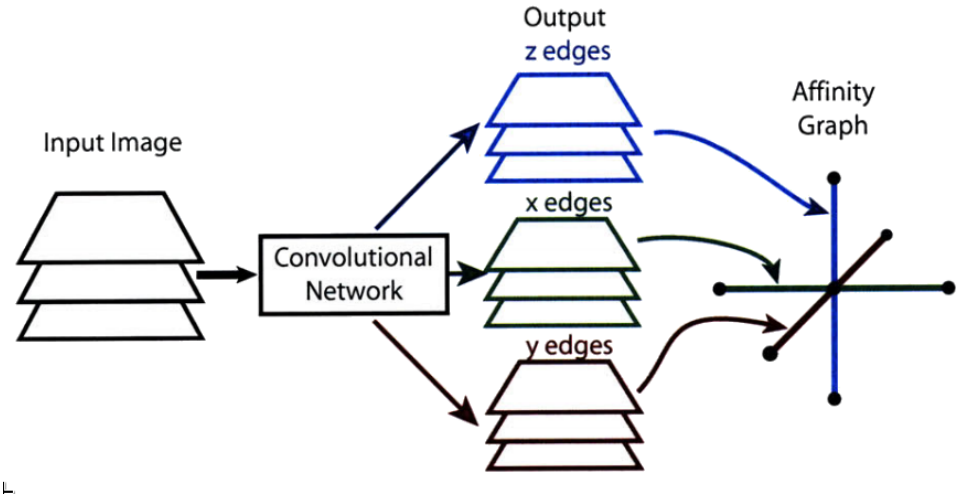


Figure 4: Creating the affinity graph using a convolutional network. The input to the network is the 3d EM image and the desired output is a set of 3d images: one each for the x, y and z directions representing the affinity graph, from [3]

Another problem was to understand the input shape. In the paper, they use a patch size of $21 \times 21 \times 21$. But it is also said that it led to an affinity classifier that uses a patch with a shape of $17 \times 17 \times 17$ to classify an affinity edge. At first it was kind of blur but we figured out that 17 correspond to the volume taking in account after four convolution layers with a kernel size of 5×5, reminiscent to the proposed architecture. The patch size is also arbitrary as we are using a FCN that, by definition, don't care about the input shape.

3.2 Computing the maximin edge

As said earlier, we have to find the maximin edge in order to compute the loss. As we have to do this operation for each training iteration, it is very important to guarantee a very low computational cost.

Our first approach was to use the Breadth First Search algorithm on the Maximum Spanning Tree efficiently created with Higra. Sadly, this method was not good because our implementation was suffering from the slowness of Python and its $\mathcal{O}(n)$ complexity for each pair of pixel, which scales badly with the number of pairs chosen.

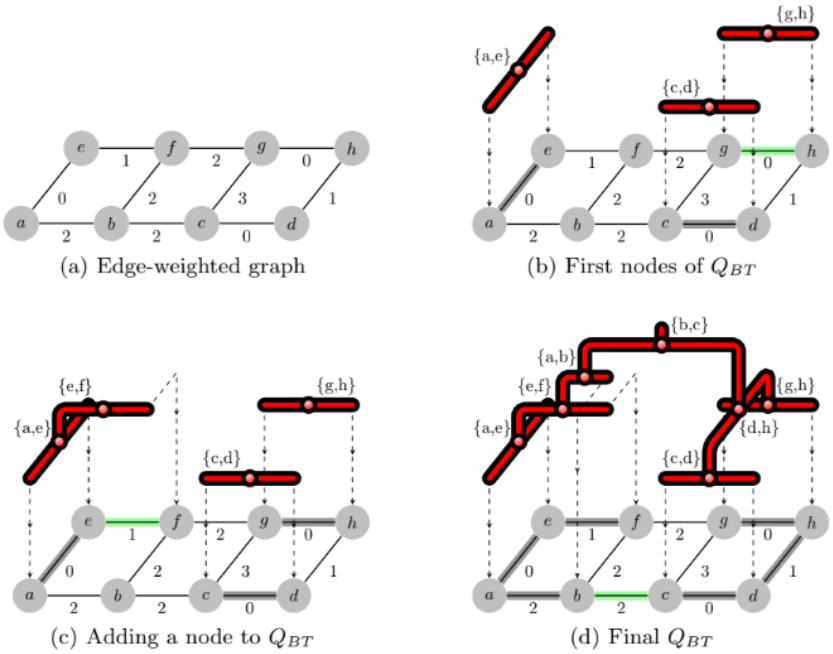


Figure 5: A simple process for obtaining a binary tree providing a strict total order relation on the edges of the MST, from [4]

In our last version we are computing a Binary Partition Tree, a binary tree by altitude ordering. This tree is build using Kruskal algorithm. The way to build this tree is straightforward, as illustrated in figure 6. Edges are added to the tree in order of altitude as our MST is built, as described in [4]. This data structure is pretty pertinent as the maximin edge between two pixels i and j is the lowest common ancestor of these two pixels in the BPT. Higra also allows us to compute the loss with a larger number of pairs without an explosion of computing time because picking the lowest common ancestor is achieved in constant time, with a preprocessing in $\mathcal{O}(n \log(n))$.

3.3 Interaction between Higra and PyTorch

We had some trouble with the interaction between Higra and Pytorch. In order to compute the loss, we have to compute a BPT on a graph build from the NN output to find the maximin edge used in the loss computation. As you can see in figure 6, the gradient history is lost by turning the affinities images in graph using Higra.
We were unable to keep tracking the gradient history by working on graphs, even after a thorough examination of each step of the process. Therefore it was impossible to train our model. But without Higra the training would be much longer. And even the previous technique using the Breadth First Search would not work as we are using Higra to compute the MST. We found the solution in a code proposed by Giovanni Chierchia and Benjamin Perret in [5] for computing the subdominant ultrametric, which is analogous to our problem. Higra has a function that allows us to make the correspondance between an edge in the graph and the output affinity image. Consequently, we are able to localise the maximin edge in the output image. Due to the fact that picking an edge does not cause gradient history

loss we are done.

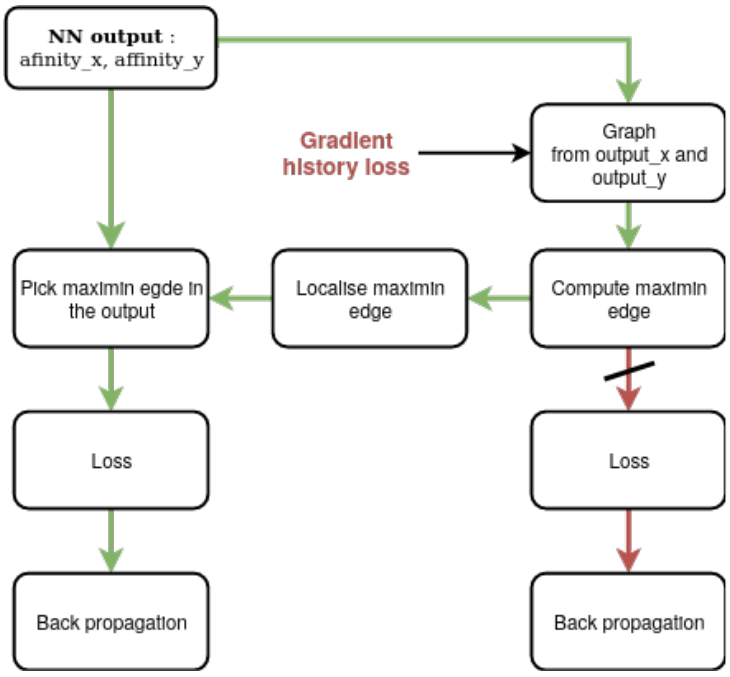


Figure 6: Process to compute the loss from our neural network. The red path was our first idea but proved unsuccesful. The green path represents the process we are using currently.

4 Results for the original method

subsectionEvaluation method We evaluated our method with two different datasets : CREMI and ISBI 2012 datasets.
Both are composed of Drosophila adult brain images.

A very simple architecture is used for the training. It is composed of 6 layers of convolution.
In [1], the original architecture had 4 convolutionnal layers with 5 features in each. We decided to add more layers and features as it gave us better results. Batch normalisation was also added as it has also given us better results in our experiments.

The architecture is described in details in table 1. As we can see, our network takes a 2D image as input whereas both datasets (as we will detail afterwards) are 3D images. We decided to use 2D images as it was a simplification of the problem and reduced training time drastically, allowing us to test our method more effectively.
In all of our test, a 3D image of size $X \times Y \times Z$ is considered as Z images of size $X \times Y$ which were stacked together to get back a 3D image.

Layer	Kernel	Strides	Features	BN	Activation	Output shape
Input						(21, 21, 1)
Convolution	(5, 5)	(1, 1)	8	Y	ReLU	(21, 21, 8)
Convolution	(5, 5)	(1, 1)	32	Y	ReLU	(21, 21, 32)
Convolution	(5, 5)	(1, 1)	32	Y	ReLU	(21, 21, 32)
Convolution	(5, 5)	(1, 1)	32	Y	ReLU	(21, 21, 32)
Convolution	(5, 5)	(1, 1)	8	Y	ReLU	(21, 21, 8)
Convolution	(5, 5)	(1, 1)	2	N	sigmoid	(21, 21, 2)

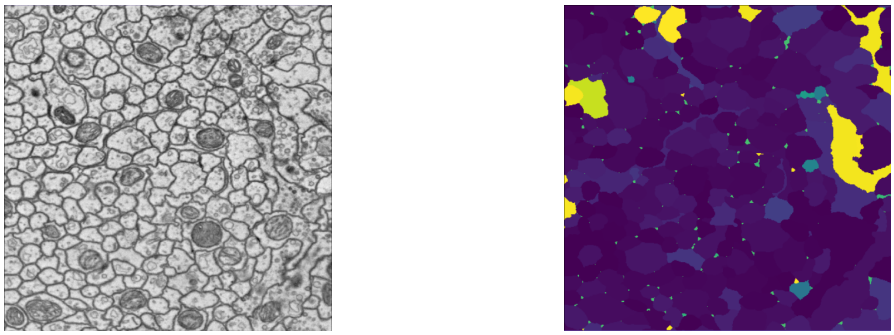
Table 1: Architecture used in all of our experiments

4.1 CREMI

The CREMI (Circuit Reconstruction from Electron Microscopy Images) dataset has three volumes but we decided to use only one (volume A) for our training. This allows us to use the remaining two for validation and testing.
This 3D image has a size of 1250x1250x125. Its corresponding grountruth was also provided in the dataset. The segmentation is constituted of labeled connected components with really thin edges, we can see an example in figure 7.
Having labeled connected components instead of borders was really helpful for our training since we need to know if two pixels belong to the same object, which was pretty straightforward with this format.
For the evaluation, we used the CREMI library that was given with the dataset in Python 2. We, then, adapted it in Python 3 as to use it more efficiently with our existing codebase.

Once our model is trained a question arises : how to get an image segmentation from the affinity graph ?
The affinities were averaged/weighted in [1] but other methods could also work.

We used two different methods to answer this problem.
First of all, a BPT (Binary Partition Tree) of our affinity graph and then a graph cut could be a first idea, which is a bit more complex than just averaging the affinities.



(a) Original image (b) Groundtruth

Figure 7: CREMI dataset (volume A)

However, even with a strong threshold for our cut (around 0.99), isthmus appeared and fused two different objects together. This is heavily penalized by the Rand Index and not a desirable outcome.
An other method, that got rid of isthmus, was to average the affinities and then threshold the result, which we used in all of our testing as it was the best solution (relative to the BPT + graph cut).
An illustration of the isthmus issue can be seen in figure 8 .

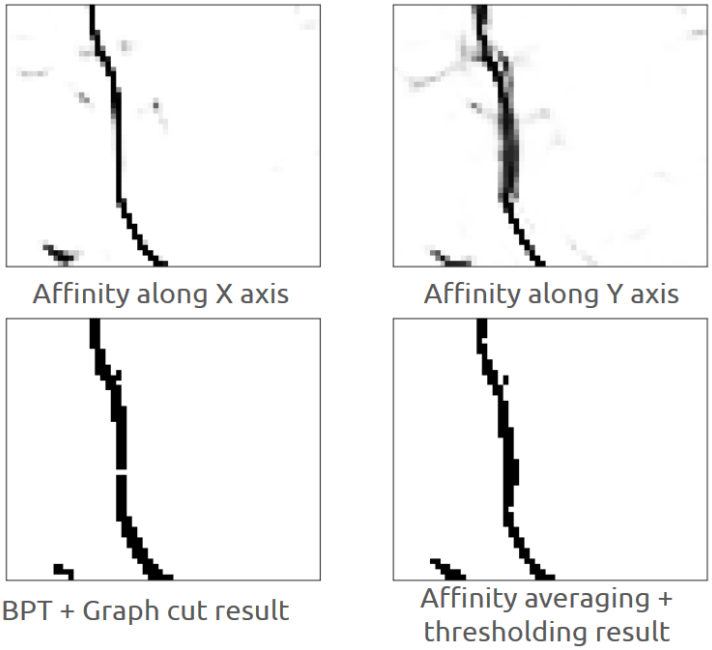
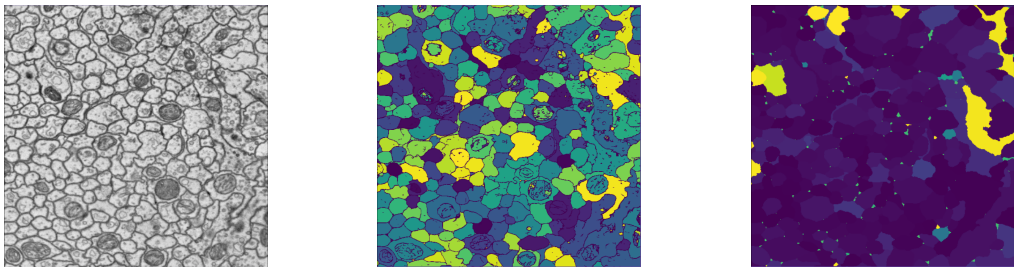


Figure 8: Illustration of the isthmus issue in our segmentation

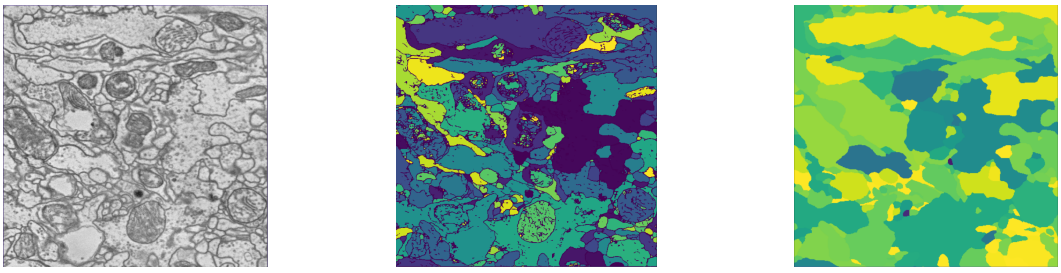
Our results are promising as the different objects are well segmented, as illustrated in figure 9. However we can see that the nuclei are not well segmented as they should be part of the cell. This is a logical issue since on a local scale, it is hard to differentiate between the border of a nucleus and the border of a cell.

The second volume (volume B) was used as the test set. The image was similar to the one in the first volume but the objects are more "stretched out."
As we can see in figure 10 the objects are well segmented but we had the same issues as on our training set.



(a) Original image (b) Our segmentation (c) Groundtruth

Figure 9: Results on the CREMI dataset volume A (Training set)



(a) Original image (b) Our segmentation (c) Groundtruth

Figure 10: Results on the CREMI dataset volume B (Test set)

The segmentation is obviously a bit worse since we did not train on this image, but they are still encouraging.

To get a better understanding of our results, we can evaluate the segmentation, according to the Rand index and the VOI (variation of information) merge and split.
The Rand index should be the highest as possible, closer to 1 and VOI should be lower to be better.
In this example, we evaluated on all slices of our 3D images and averaged the final result.

	Rand index	VOI merge (lower is better)	VOI split (lower is better)
MALIS : Training set (original architecture)	0.53	2.08	1.32
MALIS : Training set	0.61	1.25	1.03
MALIS : Test set	0.53	1.57	1.38

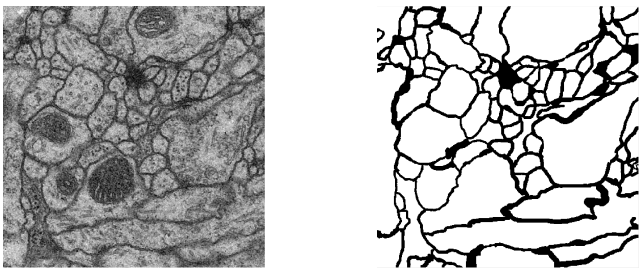
Table 2: Results on the CREMI dataset

As we can see in table 2, in the original architecture of 4 layers, the Rand index is 0.53 while we got 0.61 with our training set and 0.53 with our test set. We achieved better results than the original architecture with our small improvements, but it will be hard to improve them much more with the current method.

Thus, our results are hopeful knowing the architecture used was really simple.
Our results are still far from the state of the art but it could get even better by improving various parts of the method, as we will detail later.

4.2 ISBI 2012

We also evaluated our method on the ISBI 2012 Challenge dataset which is composed of a single $512 \times 512 \times 30$ image of drosophilia brain image.

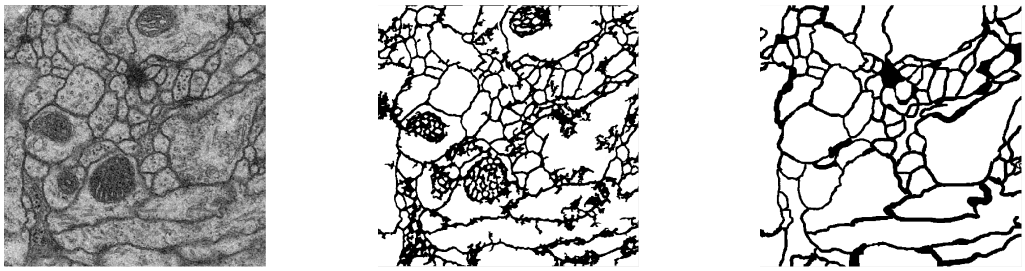


(a) Original image (b) Groundtruth

Figure 11: ISBI dataset example

The groundtruth provided with the dataset is composed of borders in black and cells in white, which we then computed the labeled connected components of as pre processing for our training. The test set was given with no groundtruth (as it is a challenge) and we had to submit our output to the challenge's leaderboard to get our scores. We evaluated using FIJI (Fiji Is Just ImageJ) on the training set as an evaluation script was given for it in the challenge.

The same architecture that we used for the CREMI dataset was used for the ISBI dataset.

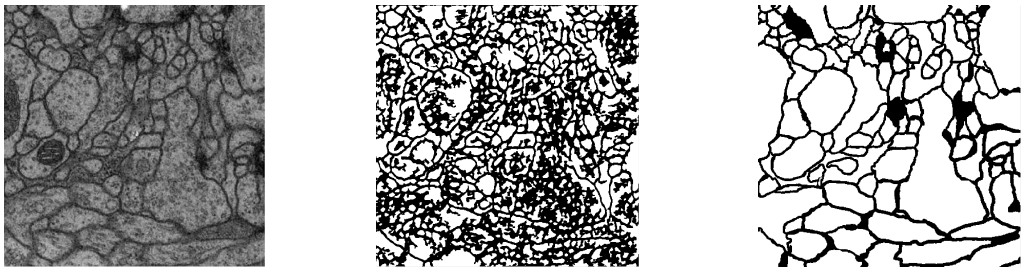


(a) Original image (b) Our segmentation (c) Groundtruth

Figure 12: Results on the ISBI dataset (Training set)

As we can see in figure 12 we are able to get a good segmentation, with the same issue as before with the darker nuclei. We can also see some "cracks" in our objects which was not an issue with the CREMI dataset.
However the results are not always this good. As we can see in figure 13, we obtain a poor segmentation, where every object is oversegmented and few borders are visible. This was the worst result that we got. This image is also the least constrasted, that's why we can hypothesize that the oversegmentation issue comes from the lack of contrast of the image.

We evaluate through the same metrics as for the CREMI dataset, that is to say the Rand index and the VOI. This time the VOI should be higher to be better.
We were not able to find exactly how the VOI was computed in both cases and where the difference lies as to which values are obtained. More works need to be put on understanding the computations of those metrics as to be able to analyze them better.



(a) Original image (b) Our segmentation (c) Groundtruth

Figure 13: Results on a less constrasted image of the ISBI dataset (Training set)

	Rand index	VOI
MALIS : Training set	0.76	0.89
MALIS : Test set	0.73	0.87
Thresholding : Test set	0.752	0.82

Table 3: Results on the ISBI dataset

As we can see in table 3 the Rand index on our training set was 0.76 and 0.73 for the test test, which is similar but still higher than a thresholding which gets a Rand index of 0.72. Our VOI is also a bit higher than the threshold.
Those are better results than CREMI's probably due to the small size of the image and the thick borders in the output segmentation.

The results are also encouraging, as iss to be expected since both datasets are very similar.

5 Improvements to the method

As we have seen before, MALIS performs really well, but can still be improved.
It was most notably improved in [2] where they were able to improve the affinity prediction using more recent architectures, by improving the training and taking full advantage of the MST and by applying a post-processing on the affinity graph instead of a simple thresholding.

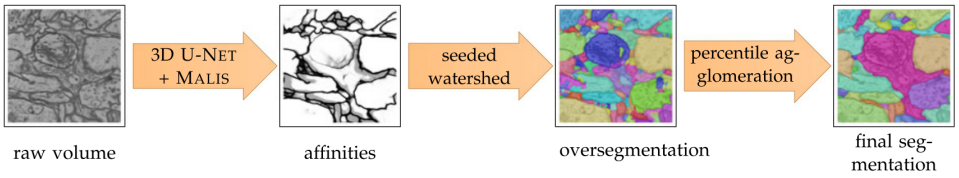


Figure 14: Improves MALIS as described in [2]

As we can see in figure 14 the CNN was replaced by a U-Net, which we will describe afterwards. Then the segmentation is obtained using a seeded-watershed, which is then improved using a percentile agglomeration of small objects.

5.1 Using a more potent architecture

Indeed, one of the limits of the previous method was the use of a relatively simple neural network to predict the affinity. This is mostly due to the fact that neural networks have greatly improved since the original paper [1] in 2009.

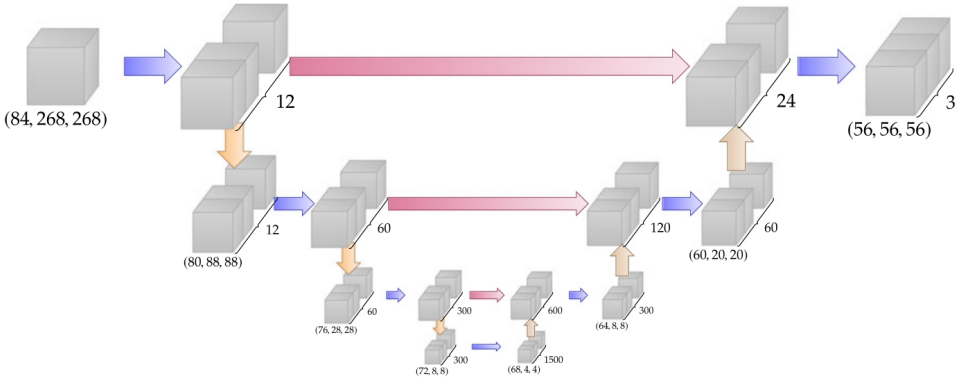


Figure 15: U-Net architecture used on the CREMI dataset from [2]

U-nets especially have been widely used for image segmentation and are thus a natural choice in our case. The architecture used in [2] is shown in figure 15. As we can see, the training will obviously be much longer than previously, but the results should be significantly better.
A question that can arise is why not simply use just a U-Net without the MALIS loss, and as we will see later, the MALIS loss gives us better results on differents measures of segmentation quality.

5.2 Constrained MALIS loss

Previously, we only computed the maximin edge for a pair of pixels (or a finite amount of pairs), which means that some information from the MST was not used.
However, we would like to compute the maximin edge for all pairs of pixels in the image. This was not done in the previous paper for time efficiency reasons.

However they describe a way to compute the loss with in quasilinear time instead of in polynomial time.

The main idea is that all maximin edges are in the MST, and there are $n-1$ edges if we have n pixels in our image. We could then simply compute the loss over all those edges but this would mean that they are all as important as the others. However since we have n^2 pairs of points and $n-1$ edges in our MST, they will be the maximin edge for a different number of pixel-pairs. So we must find a way to see how often an edge from the MST is a maximin edge.

When we add an edge using Kruskal's algorithm, we can look at the "size" of the trees it merges and deduce the number of pairs for which the current edge is the maximin edge.

From this, we can define the positive weight of an edge e as the number of pairs from the same object/segment merged by adding e to the MST. More formally we have :

$$w_p(e) = |\{(u,v) \in F^2 \mid \delta(u,v) = 1, e = mm(u,v)\}|$$

Similarly we can define the negative weight of an edge as :

$$w_n(e) = |\{(u,v) \in F^2 \mid \delta(u,v) = 0, e = mm(u,v)\}|$$

These formulations allow us to rewrite our loss function as :

$$L(I,\theta,S) = \sum_{e \in MST(G)} w_p(e) l(1, A_e(I,\theta)) + w_n l(0, A_e(I,\theta))$$

With A_e the affinity of an edge e.

The idea to compute w_n and w_p is that every time that an edge e is added to our MST, it merges two trees T_1 and T_2.
We will define V_{T_i} as the set of vertices in the tree i. If we take two pixels $i \in V_{T_1}$ and $j \in VT_2$ we can be sure that e is their maximin edge, as it is the edge with lowest affinity in the path from i to j.
As such we can already deduce that there are $|T_1||T_2|$ pairs of pixels with e as their maximin edge, but we don't know yet for which of these pairs if they should be in the same object or not, which is necessary to compute w_n and w_p.

To do so we need to not only keep track of the areas when computing our MST, which is straigthforward, but also keep track of areas per label.
To get the area per label when adding an edge, we can simply add together the areas per label from both trees that are merged by this edge.

By defining $V_{T_i}^j$ as the set of vertices in tree T_i with label j, we can compute w_n and w_p as follows:

$$w_p(e) = \sum_{i \in 1 \dots k} |V_{T_1}^i||V_{T_2}^i|$$

$$w_n(e) = \sum_{i \neq j \in 1 \dots k} |V_{T_1}^i||V_{T_2}^j| = |V_{T_1}||V_{T_2}| - w_p(e)$$

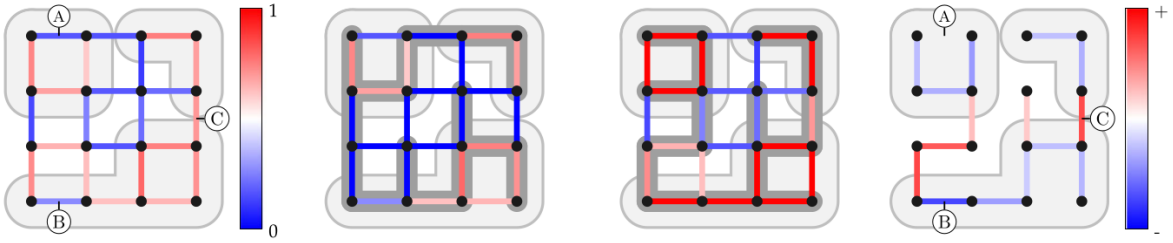
Since this is computable in $\mathcal{O}(k)$ and we need to to this step n times, the added complexity is only in $\mathcal{O}(kn)$, making the total computation of the loss in $\mathcal{O}(n\log(n) + kn)$.

With this loss function being able to be computed in quasilinear time, this will allow us to use the whole patch for the loss computation instead of a few pairs of pixels, which should improve the results. This loss function is the same as before, it still is related to the Rand Index, but it should now approximate it much better.

5.3 Two pass computation of the loss

An issue with the aforementioned loss is that all edges have both a positive weight, where we will compare them to the value 1 and a negative weight where we will compare them to 0.

However an edge as an optimal value of either 1 or 0, not both, so how can we only take only into account the right value ?



(a) Predicted affinities. (b) Positive pass. (c) Negative pass. (d) Gradient of loss.

Figure 16: Illustration of the constrained MALIS loss, blue edges represent a low affinity and red edges a high affinity; from [2]

As we can see in figure 16, we will compute the loss in two passes. In the positive pass, (b), affinities of edges between ground-truth regions are set to zero (blue), in the negative pass (c), affinities within ground-truth regions are set to one (red). In either case, a maximal spanning tree (shown as shadow) is constructed to identify maximin edges. Note that, in this example, edge A is not a maximin edge in the positive pass since the incident voxels are already connected by a high affinity path. In contrast, edge B is the maximin edge of the bottom left voxel to any other voxel in the same region and thus contributes to the loss. Similarly, C is the maximin edge connecting voxels of different ground-truth regions and contributes during the negative pass to the loss. The resulting gradients of the loss with respect to each edge affinity is shown in (d) (positive values in red, negative in blue).

To really see the benefit we need to look at how each pass influences the loss function. In the positive pass, since we set all negative edges (edges between objects) to 0, their contribution to the loss will be 0, so the loss can be rewritten as :

$$L(I, \theta, S) = \sum_{e \in MST(G)} w_p(e) l(1, A_e(I, \theta))$$

But can those negative edges have a non zero w_p ?
Inside an object, we will have weights having a non zero value, or at worst having value 0. So if we take i, j in the same object, there will be a maximin path that stays in this object, so no negative edge is the maximin edge for any pair of pixel inside the same object, giving all of our negative edges a positive weight equal to zero.
As such during the positive pass we only take into account the positive edges (edges inside the same object) and their positive weight, which is exactly what we wanted.

We will do similarly during the negative pass, where we will only take into account the negative weight of negative edges.

Using these two passes we are able to only account for the useful terms inside this loss, and overall achieve a more accurate loss than before.
The only issue with the two passes is the computation time. Although it doesn't change the complexity, doing two passes doubles the time required to compute the loss. However this is still far superior than the first formulation of the loss, or the version in $\mathcal{O}(n^2)$.

5.4 Seeded watershed as post processing

Remember that before, we computed the segmentation by thresholding our affinity graph. However this doesn't give optimal results, as for example locally another threshold would perform better.
An issue that was also encountered was small objects that were inside bigger ones. These objects lead to an oversegmentation and we would like a way to automatically remove those inaccuracies, or at least part of them.

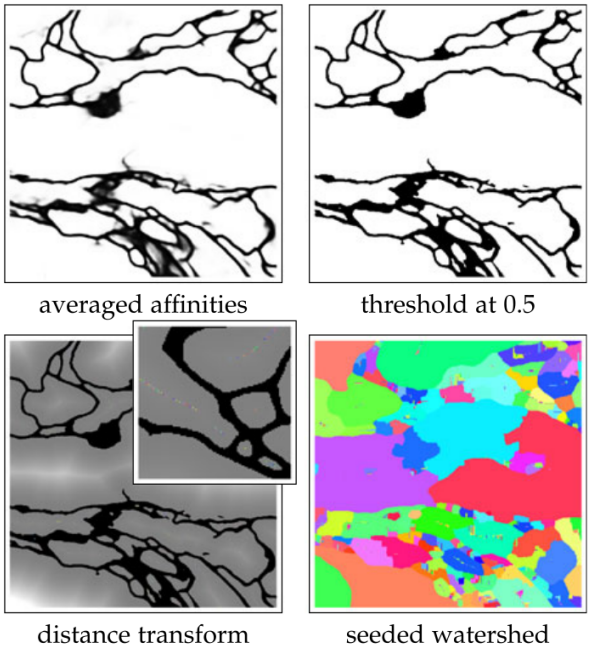


Figure 17: Seeded watershed on an affinity graph, as described in [2]

This is where the framework described in [2] comes in play. As we can see in figure 17, the first step of the process is to average the affinities, as we did before. Afterwards the averaged affinities are thresholded at 0.5 (here this threshold is not a parameter). Then a distance transform (or a distance map) is computed from the objects to the borders. In this distance transform, the furthest points from the borders will have the higher values. Then, all the local maxima are taken as seeds for a seeded watershed, which gives us a first segmentation.

This is still an oversegmentation, but a fragment agglomeration algorithm is then used to fuse regions together. multiple criteria can be used to determine which regions to merge together and they are described in [2].
When all of those steps are done, we obtain the final segmentation.

Describe agglomeration

As we can see the process is greatly improved from the first version of MALIS, with the loss function, the neural network architecture and the post processign used being much more powerful than their previous counterparts.

6 Implementation for the improved method

6.1 Neural network used

6.2 Use of a BPT to compute the loss

6.3 An improved way to generate edge weights

One of the issues that we had before was that when computing the edge weights we lost the gradient information, which forced us to go back to our neural network's output, which was costly. However a change in Higra's function to compute edge weights forced us to adapt and find a better method, which will describe here.

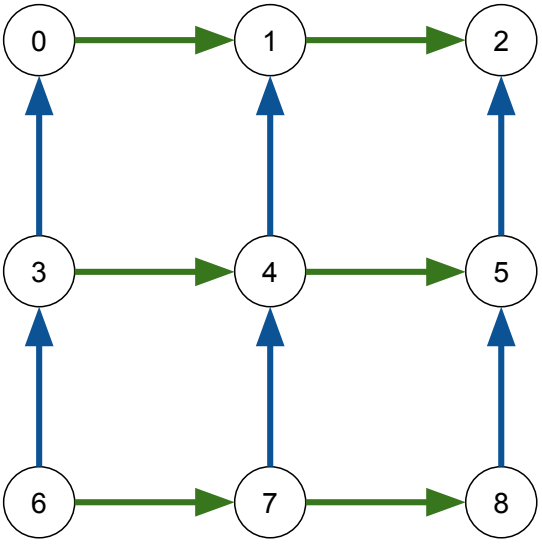


Figure 18: Construction of a 4 connected graph in 2d. Green edges represent x edges and blue edges represent y edges

As we can see in figure 18 there is a straightworward way to relate our edges and their counterpart in the neural network output.

The only constraint that we have for the construction of the edge weights is to use operations supported by our deep learning library of choice. For PyTorch, this includes if statements, min, max, all array indexing methods etc.

6.3.1 Weighting edges with Higra

Once we have our edge weighting function, we can directly use Higra to get our edge weights.

```
import higra as hg

# Both variables were obtained from our NN
x_edges = ...
y_edges = ...

#Building our graph
```

```
graph = hg.get_4_adjacency_graph(x_edges.shape)

#Getting our edges and computing their weights
src, dst = graph.edge_list()
weights = weight_function(src,dst,x_edges,y_edges)
```

This will work in the general case, but in the case of a 4 connected graph and with the way our neural network generates edge weights we can optimise this. Indeed the edge list is obtained in a structured and deterministic way which we will now decsribe.

6.3.2 Higra's edge weights structure

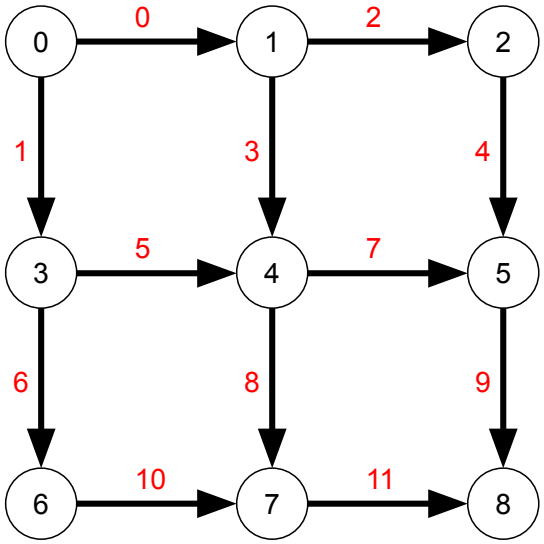


Figure 19: Oder of edges obtained using Higras's *edge_list* function.

As we can see in figure 19 edges are always obtained in the same fashion, which will allow us to develop a function that bypasses the call to *edge_list* since they will always be in the same order.
The constraint are thus that we need to respect this order in our function, and we need to be careful about code maintenance and unit tests, as a change of the way the edges are obtained would break our method.
To alleviate those risks, the best method is to create tests comparing our method and the official method, which is guaranteed to work.

6.3.3 Our method to generate edge weights

As we can see in figure 20, we get edge matrices the same size as our image, but if our image is of dimension $n \times m$ we only need a matrix of size $n \times (m-1)$ for our edges in the y direction and of size $(n-1) \times m$ for our edge in the x direction.
The choice to have those irrelevant edge weights is simply for ease of use, and to ease the design of the NN output.

As we can also see, our edge weights are simply a kind of entanglement of our x and y edges. This leads us to the following code to generate of edge weights which doesn't rely on Higra's function :

```
import higra as hg
import torch

# Both variables were obtained from our NN
x_edges = ...
y_edges = ...

# Getting our edge weights

height, width=x.shape
# Removing the out of graph edges
```

$$\text{x\_edges} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \qquad \text{y\_edges} = \begin{bmatrix} -1 & -2 & -3 \\ -4 & -5 & -6 \\ -7 & -8 & -9 \end{bmatrix}$$

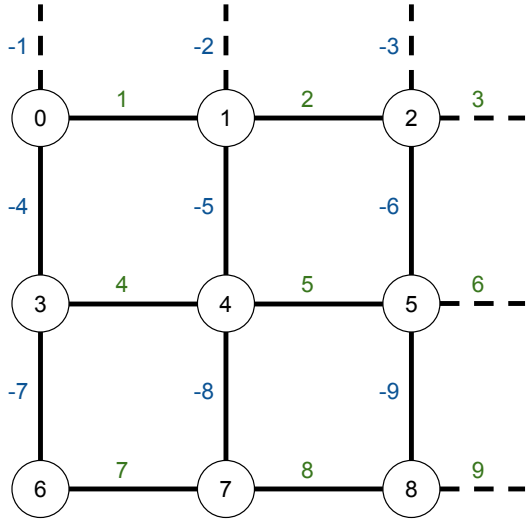


Figure 20: Example of desired results from our x and y edges predicted by our neural network. Notice that some edges are not contained in the image. Even though they are present in our NN output we will remove them from the graph

```
x_u  = x_edges[:-1].T[:-1]
y_u = y_edges[1:].T

edge_weights = torch.from_numpy(np.empty((2*width-1,height-1),dtype=np.float64))
edge_weights = edge_weights.type(torch.FloatTensor).to(device)

# Creating the entanglement
edge_weights[0:-1:2,:] = x_u
edge_weights[1::2,:] = y_u[:-1]
edge_weights[-1] = y_u[-1]

#Adding back missing x edges
edge_weights= torch.cat((edge_weights.T.flatten(),x_edges[-1,:-1]))
```

This example here is using PyTorch, but an analogous version for numpy can be made by replacing function names.

Using this code we are able to generate edge weights very fast and we are sure that our edge weights will contain the gradient history, which is what we wanted.

6.3.4 Backpropagation

Once we have our edge weights as an array, for any processing that we will have to do, we can refer to the edge weights which are guaranteed to contain the gradient information. This is easier that going back to the NN output, where we would have to reverse the code used to get our edge weights to find their position in the output.

This process of going back to our neural network was very costly and caused a very important bottleneck in our code, which this solution alleviated instantly. We had a backward pass that was taking 100 times as long as our forward pass, which should not be the case, we expected it to be 2 or 3 times as ong at most.

7 Results for the improved method

8 Teamwork

8.1 Team organization

In order to have an overview of the tasks, we are using the software Trello. With this, we can write all tasks that we need to do, who is responsible of the task and the progression of the project: we can see what is already done and what the other members are currently working on. We communicate with each other using Slack in which we put information and additional contents related to the project.
Using Trello was really helpful as it always gave us the feeling that we were progressing, even when we didn't get substantial improvements, we knew that we had worked and that we were closer to our goal. This helped us stay motivated at all times.

After knowing what tasks we had to do, we split the work based on preferences of everyone. Quentin was chosen team leader, he worked on a lot of things in the project and supervised the other team members. He worked on the computation of the affinity graph from the output of the neural networks. When working on the second paper, the improved MALIS, he implemented the new loss and optimized it. He did data augmentation, worked on the U-net with Annie and Raphaël and on image visualization. Annie worked on the datasets, both by exploring them and preparing them. Josselin worked on the computation of the loss and the maximum spanning tree. He then tried to generates a 6 adjacency graph for 3D images with Higra, using C++ and made an extension. After that, he focused on the post processing, especially Mumford-Shah method. Tiphanie worked on the evaluation and the image generation from the output of our neural network. She then studied the different post-processing we could apply to our outputs to get better results and tested them. Raphaël, who joined the team for the second semester, first immersed himself in the project subject and what we have done so far to have a better overall understanding and what needs to be implemented in the second semester. He then worked on the implementation of the U-net and tweaked it with Annie and Quentin. He also created the video explaining our subject and what we have done in the project. We kept everybody informed at all time of what other people did so that everybody could understand every part of the work.

Every week, we had a meeting with our supervisor in which we discussed about what we did during the week, our issues and solutions if we had some and what we will do afterwards. During each meeting, a different person lead the discussion. This helped us improve our communication skills and understanding of the project as we had to present both our work and the works others did, and made sure that everybody spoke for the same amount of time overall. We implemented this since discussions were almost one-sided before which was not the best solution.
Since the beginning of the quarantine, we are doing our meetings with Hangouts, allowing us to talk to each other in real time. We also use more social networks like Discord, allowing us to talk with the team members when working. We can also show our screen to the others.

We write a report every week in which we note what we did in the week to prepare for the meeting and to show results, or more graphical elements. It also allows us to keep track of our progress during the project. We wrote all the codes in documented Jupyter notebooks that we will clean and put on GitHub.

8.2 Task distribution

Annie	Josselin	Quentin	Tiphanie	Raphaël
Find dataset	Computation of MST	Finalize the saliency notebook	Affinity graph thresholding	Learn PyTorch

Annie	Josselin	Quentin	Tiphanie	Raphaël
Exploration of dataset	Computation and optimisation of the path between i and j	Add features in the saliency notebook	Computation of connected components	Implement U-net
Analyze what is the intput and output of the neural network and their size	Computation of the maximum path in the MST	Graph generation from output of neural network	Image generation	Tweak U-net hyperparameters
Understand how to use hdf files	Computation of the loss	Preparation of the dataset	Finishing inference by creating segmentation	Creation of the video with Manim
Create architecture of convolutional neural network	Find interesting patches in dataset	Get vertices pairs in same and different object	Learn PyTorch	
Load ISBI-2012 data	Train and evaluate on ISBI in 2D	Train the network on maximin affinty	Study post processing	
Learn PyTorch	Learn PyTorch	Image reconstruction with inference	Test watershed hierarchy functions	
Implement U-net	Create 6 connected graph and an add it on Higra	Fiji for evaluation on ISBI	Test other methods: region adjacency graph, horizontal cut	
Study contrast problems between images	Work on Mumford-Shah post processing	Load ISBI-2012 data		
Image normalization		Train and evaluate on ISBI in 2D		
Prepare BSDS dataset		Learn PyTorch		
Prepare notebooks for training data		Implement two passes loss (MALIS constrained loss) and optimize it		
Tweak U-net hyperparameters		Incorporate new loss into the U-net and tweak its hyperparameters		
		Pre-processing on CREMI dataset		

Table 4: List of the main tasks and their repartition (from our Trello board)

As we can see in table 4 we tried to distribute the work based on affinity, and even though the table is unbalanced, everybody put their fair share of work into the project.

One aspect that is absent from this table is all that we learned. As we will detail in the next section, everybody did not have the same knowledge at the beginning of the project and it took everybody a different amount of time to learn the necessary information.

8.3 Obstacles and overcoming them

The subject of this project was quite difficult at first because we did not have any knowledge on image segmentation or morphology. Moreover, it was also hard to read and understand the scientific papers because they were written in English and we did not have the knowledge to understand them clearly at first. To overcome this difficulty, we planned several hours to go through the important things that we needed to retain, and over time we were able to grasp more detail from the papers since our knowledge improved throughout the project.

After knowing what we had to do, the next step was to figure out how to implement these ideas.
We took some time to know exactly what we needed to do and how to do it in details for the implementation. Once we knew, we just needed to find the right functions in Higra. It took us some time to understand how the functions work, as we never used them before. Documentation is available online and commented notebooks showing examples are also provided, helping us a lot to understand their implementations. We spent quite a lot of time figuring out how to use them at the beginning, but we managed to familiarize ourselves with the library as time passed.

Afterwards, we choose to use the PyTorch library for the deep learning component of the project, which nobody was familiar with in the group. This lack of experience with it meant that we had to spend more time learning it. However there is a nice tutorial on PyTorch's website that allowed us to be able to use it fairly quickly. Their documentation on Autograd was also really helpful when debugging the gradient history loss. Overall the PyTorch API was really helpful and allowed us to use PyTorch with a relative ease.

The one main big difference that we had compared to the first semester was the quarantine. Because of that, we could not meet with each other and work in a proper environment. First because of the internet connection, we also may not have good computers with enough RAM to run some codes on Jupyter Notebook. That aside, we adapted to the situation by continuing to get in touch with each other, by using social networks like I said earlier. We planned sessions during the week to work on the project and kept doing written reports and meetings.

During this project, we received a lot of help from Quentin. This project would have been really hard to do if he was not there, having no great knowledge in image segmentation and machine learning. Thanks to him, we were able to have a better understanding of the papers by having him explain them to us. It would have taken us a lot longer to understand the papers and how to implement them without him and we would not be able to present decent results in the end of the first semester presentation.

Overall, being aware of the work done by others helped us immensely when facing issues as we could all think together and challenge each other's understanding of different topics. This allowed us to overcome all those difficulties more easily and in a more enjoyable way.

9 Conclusion

As we have seen before, even with the implementation of the originak MALIS paper [1] we were able to get promising results on both the CREMI and ISBI datasets. We were also able to see that the method was greatly improved in [2] and this will be the goal that we aim to reach for the coming semester.
We were able to work great as a team and hope that with Raphaël joining us for this semester we will be able to produce even better results. We also hope that we will be able to apply the methods to other segmentation tasks, such as the BSDS500 dataset.

References

[1] S. C. Turaga, K. L. Briggman, M. Helmstaedter, W. Denk, and H. S. Seung, "Maximin affinity learning of image segmentation," *arXiv:0911.5372 [cs]*, Nov. 2009. arXiv: 0911.5372.

[2] J. Funke, F. Tschopp, W. Grisaitis, A. Sheridan, C. Singh, S. Saalfeld, and S. C. Turaga, "Large Scale Image Segmentation with Structured Loss Based Deep Learning for Connectome Reconstruction," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 41, pp. 1669–1680, July 2019.

[3] S. Turaga, "Learning image segmentation and hierarchies by learning ultrametric distances," Apr. 2010.

[4] L. Najman, J. Cousty, and B. Perret, "Playing with Kruskal: Algorithms for Morphological Trees in Edge-Weighted Graphs," in *Mathematical Morphology and Its Applications to Signal and Image Processing*, vol. 7883, pp. 135–146, Berlin, Heidelberg: Springer Berlin Heidelberg, 2013.

[5] G. Chierchia and B. Perret, "Ultrametric Fitting by Gradient Descent," *arXiv:1905.10566 [cs, stat]*, Oct. 2019. arXiv: 1905.10566.