POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

SoftCut: a novel fully differentiable relaxed graph cut approach for image segmentation in deep learning

Tesi di Laurea Magistrale in
Computer Science and Engineering - Ingegneria Informatica

Alessio Bonfiglio, 944833

Advisor:
Prof. Matteo Matteucci

Co-advisors:
Marco Cannici
Francesco Lattari

Academic year:
2021-2022

Abstract: Graph cut algorithms always seemed to have a natural application in image segmentation, but the difficulties in extracting the needed features from the image made the convolutional neural network architectures like U-Net take the lead in this task. In recent years many solutions have been proposed in order to implement this kind of algorithms into a neural network layer. SoftCut, the approach proposed in this work, is a differentiable relaxation of the graph cut problem, equivalent to an intuitive electric circuit, that is shown outperforming both U-Net and the most promising of these solutions in terms of IoU on a real-world scenario like Cityscapes, while being faster than these methods of integrating an optimization problem into a network.

Key-words: graph cut, differentiable, image segmentation, artificial neural network, deep learning, machine learning

1. Introduction

Image segmentation is the task of assigning a label to each pixel of a given image. This assignment partitions the input into various segments that show in which parts of the image is present the relative label. Traditionally this kind of problem has been solved as a graph cut problem, which is a combinatorial optimization problem that partitions a given graph, and so, treating an image as such, has a natural application in image segmentation. As shown in [13], it is possible to partition an image in two disjoint sets F and B of foreground and background pixels by solving the problem

$$F^*, B^* = \underset{F,B}{arg\,min} \sum_{i \in F} b_i + \sum_{i \in B} f_i + \sum_{\substack{\{i,j\} \in E \\ |F \cap \{i,j\}|=1}} w_{i,j} \tag{1}$$

where f_i and b_i are a measure of how much a pixel i should be respectively part of the foreground and background partition, E is the set of edges between adjacent pixels and $w_{i,j}$ is a measure of how important the edge i, j is (i.e. $w_{i,j}$ has a large value if the pixels i, j should be part of the same set). Typically the edges used are between each pixel and the four other adjacent ones (above, below, left and right), but, in addition, it is also possible to consider the other four diagonal edges (top-left, top-right, bottom-left and bottom-right) if needed. Problem (1) can then be solved by converting it into a flow network $G = (V, E')$ and computing the maximum flow, where the elements of V are pixels of the image plus the source s and the sink t nodes and the edges of

E' are the ones from E with capacity $w_{i,j}$ plus an edge from the source to each pixel with capacity f_i and one from each pixel to the sink with capacity b_i, as shown in Figure 1.

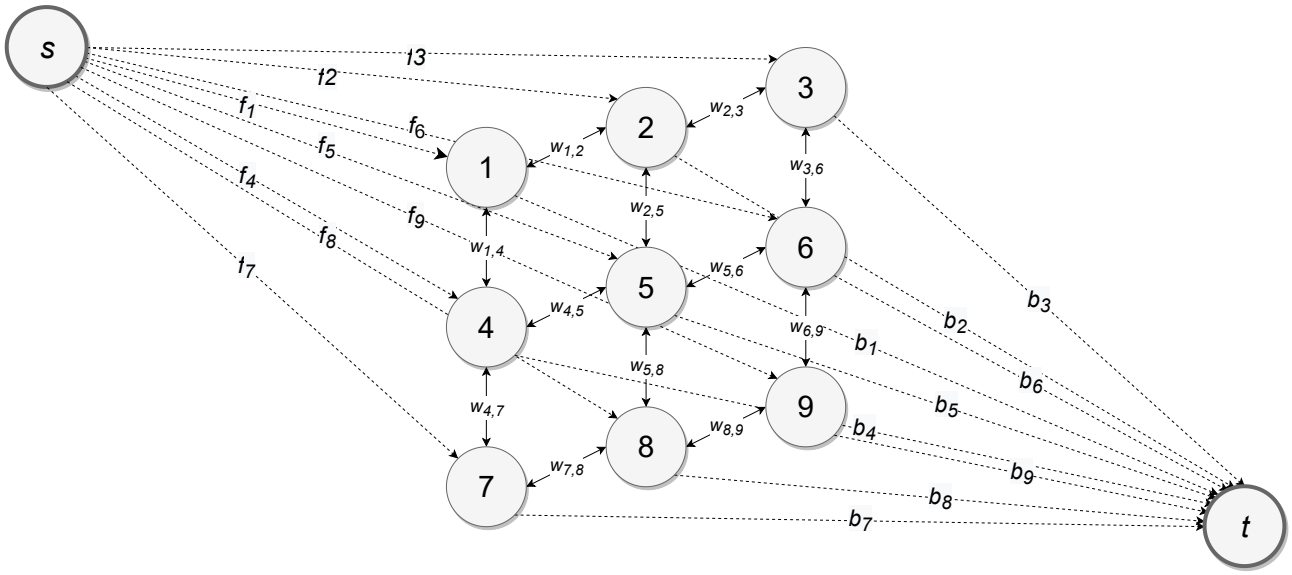


Figure 1: Example of a flow network for a 3x3 image

The effectiveness of this approach, unfortunately, relies on the values f_i, b_i and $w_{i,j}$ provided by the user, which cannot be trivially computed.

In recent years, deep learning algorithms took the lead in tasks like image classification and segmentation due to their ability to learn how to extract the features needed during the training of the model. For image segmentation, different successful artificial neural network architectures have been proposed over the years, like U-Net [22] which exploits a mix of convolutions, down and up-sampling and skip connections to compute the probability of the labels for each pixel. However, while very successful, these techniques need to learn also how to combine those extracted features to compute the resulting segmentation mask, requiring more parameters to approximate a solution for this task while being subjected to the stochasticity of the training process. Thus, having the possibility to insert a graph cut algorithm into a layer of a neural network and train it end-to-end would combine the best aspects of both the techniques, by letting the parameters of the network learn how to extract the needed features from the image and pass them to the cut layer that performs the actual segmentation. Moreover, the relative scale of the features would be learned autonomously by the network and, as proven in *Appendix A*, the receptive field of each pixel after passing through a graph cut can increase to the whole input image.

Standard neural network layers can only consist of differentiable functions, due to the need of computing the gradient of the loss function during the backward pass of the backpropagation algorithm used to train them. Graph cut, instead, is generally a problem solved algorithmically, which does not have a differentiable close form solution that can be inserted into a neural network layer. The classic formulation (1) does return a binary label for each pixel, making the solution not even continuous. Different approaches have been proposed in the last few years to overcome these problems: compute the gradient by executing the optimizer multiple times with perturbed inputs [4], relax the problem and compute the gradient via an approximated formulation [8] or relax the problem to a linear program and use the implicit function theorem to compute the gradient [1]. These approaches come with their downsides like a slow processing speed or computing an approximate gradient.

The solution proposed in this work aims to solve these problems: SoftCut is a relaxation of the graph cut problem (1) that is formulated as a system of linear equations with a continuous solution that can be solved and differentiated efficiently both exactly or approximately as needed. Moreover, this formulation also has an intuitive interpretation in the form of an electric circuit that provides an easy way to understand its working.

1.1. Background

Image segmentation is a broad term that can refer to two different tasks:

- **Binary segmentation**: where the algorithm, given an input image, classifies each pixel into a negative

or a positive class.

- **Multiclass segmentation**: where the algorithm, given an input image, classifies each pixel into one of the multiple classes it takes into consideration.

The usual way to solve an image segmentation task is through deep learning because of the ability of the models to learn how to extract the features from the image, a problem that is very difficult to solve manually in most cases. Then model have to learn a function $f : \mathbb{R}^{h\times w\times c} \to \{0,1\}^{h\times w}$, where h, w are the height and the width of the input image, while c is the number of channels (usually 3 for an RGB image) in the case of binary segmentation or $f : \mathbb{R}^{h\times w\times c} \to \{1,\dots,K\}^{h\times w}$ (K is the number of classes that the network can recognize) in the case of multiclass segmentation. Neural networks require to be composed of differentiable functions, and so the binary output must be replaced by a continuous probability between 0 and 1 of the pixel to belong to each class. Doing so requires making the output of the network pass through a function that enforces that range. The usual choices are the S-shaped sigmoid function $S : \mathbb{R} \to (0,1)$

$$S(x) = \frac{e^x}{e^x + 1} \tag{2}$$

in case of binary segmentation, or the softmax function $\sigma : \mathbb{R}^K \to (0,1)^K$

$$\sigma(\boldsymbol{x})_i = \frac{e^{x_i}}{\sum_{j=0}^{N} e^{x_j}} \tag{3}$$

for the multiclass segmentation. Those probabilities then can be converted into a hard classification by checking if they are greater than a certain threshold t in the case of binary segmentation or assigning each pixel to that class with the higher likelihood in the case of multiclass segmentation. The most famous neural network architecture for image segmentation is U-Net [22].

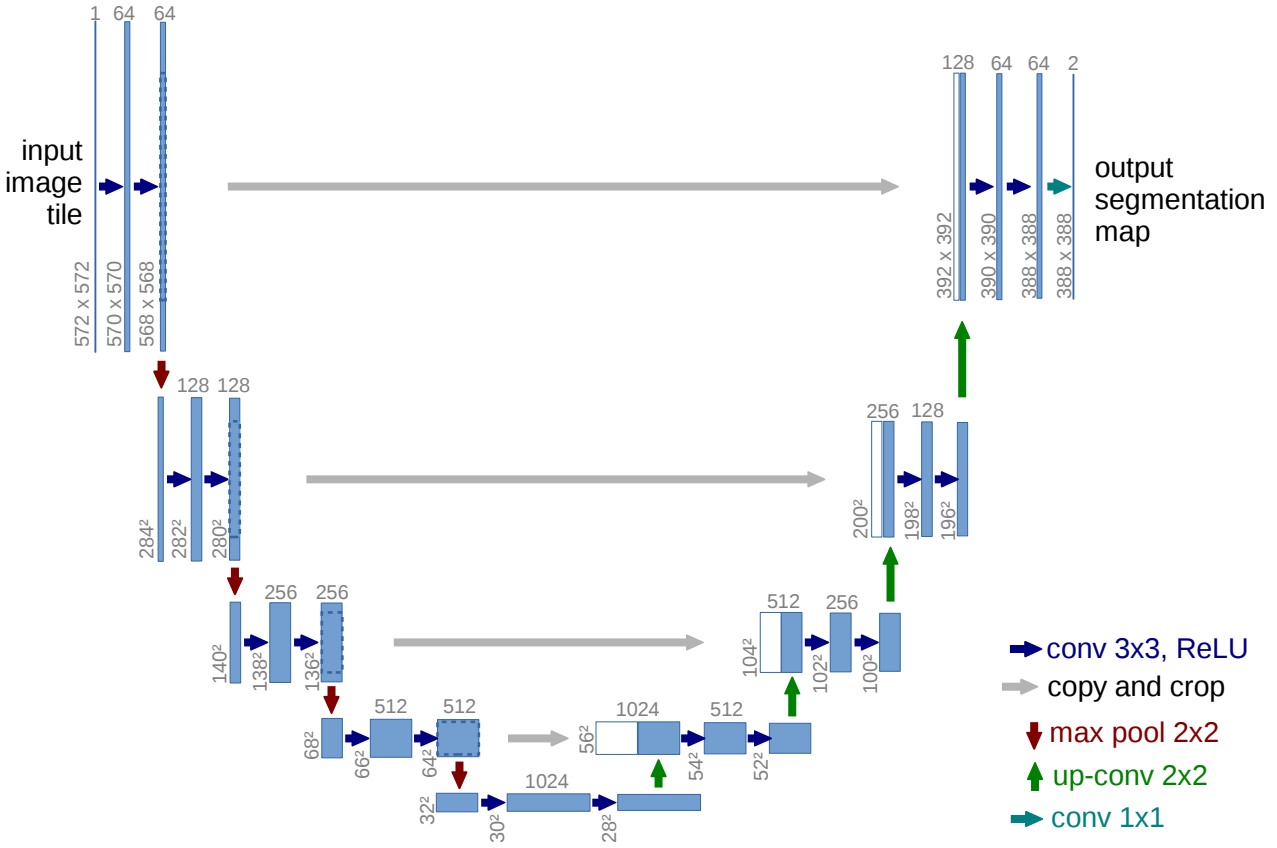


Figure 2: The U-Net architecture (credits to [22]).

As shown in 2, the name of the architecture comes from its shape: the first "downward" part is a standard convolutional neural network, that takes as input an image and applies to it a series of convolutions and max pooling, reducing at each step the resolution of the embedding while increasing its depth, and so the number of features extracted. Once the network reached the bottom of the "U" shape, instead of a fully-connected layer usually used in classification architectures like LeNet [16], AlexNet [14] or VGG [23], an up-sampling "upward" part is used. This part of the network is used to construct the full-resolution segmentation mask, but, due to

the low resolution of the feature embedding at the bottom of the architecture, it has difficulties in recreating the fine details of the input image. The idea behind U-Net is to add skip connections to the network, that connect each step of the "downward" part to the relative one of the "upward" one, in order to provide the second with the needed information to reconstruct accurately the shape of the segmented object.

The networks need a loss function to be trained. The simplest choice would be to use a binary cross-entropy loss function

$$BCE(\hat{y}, y) = -y \log(\hat{y}) - (1 - y) \log(1 - \hat{y}) \tag{4}$$

over the probability of each pixel belonging to a specific class, and then average the obtained values. Unfortunately, this solution does not work well with unbalanced classes (that is usually the case in image segmentation) because the loss will tend to favor the most common class, so introducing a bias into the model towards being more accurate on that class to the detriment of the others. A solution to this problem is the Dice loss, which is a loss function based on the Dice coefficient, published for the first time in [6]. The Dice coefficient measure how much two sets X and Y are similar by the ratio between the common items and the sizes of the two sets:

$$D = \frac{2|X \cap Y|}{|X| + |Y|} \tag{5}$$

Note that if $|X| = |Y|$ then $0 \leq D \leq 1$. In case of binary classification is possible to represent the two sets X and Y as two vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ with values between 0 and 1, that represent how much the element belongs or not to the class. In this case Equation (5) can be written as

$$D = \frac{2(\boldsymbol{x} \bullet \boldsymbol{y})}{|\boldsymbol{x}|^2 + |\boldsymbol{y}|^2} \tag{6}$$

and from this the Dice loss can be obtained as $1 - D$:

$$Dice(\hat{\boldsymbol{y}}, \boldsymbol{y}) = 1 - \frac{2(\hat{\boldsymbol{y}} \bullet \boldsymbol{y})}{|\hat{\boldsymbol{y}}|^2 + |\boldsymbol{y}|^2} \tag{7}$$

This loss function can be used to measure how much the computed segmentation mask $\hat{\boldsymbol{y}}$ differs from the ground truth $\boldsymbol{y}$. Unlike the classic binary cross-entropy, which is computed per pixel and then averaged, the Dice loss is not affected by the imbalance in the number of positive pixels and negative ones, because of the denominator in Equation (6) that normalizes the value. Note that in the case of multiclass segmentation it is possible to consider each class as a binary segmentation problem, and so compute the loss for each of them and then sum those losses.

The same problem of the binary cross-entropy extends also to other metrics that are usually used to measure the goodness of a model, like the accuracy

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN} \tag{8}$$

where, in the context of binary segmentation, TP are the correctly classified pixels in the positive class, TN are the correctly classified pixels in the negative class, FP are the wrongly classified pixels in the positive class and FN are the wrongly classified pixels in the negative class. It is easy to see that, in a case where there are few pixels of the positive class in an image and the model predicts all the pixels into the negative class, the accuracy metric will still have a high value while the actual result of the model is not as good. To fix this problem the intersection over union (IoU), or Jaccard index, metric is often used instead

$$IoU(X, Y) = \frac{|X \cap Y|}{|X \cup Y|} \tag{9}$$

Where X and Y are the two sets that are getting compared which, in the case of binary segmentation, become the hard classifications predicted by the model and the ground truth. As in the case of the accuracy, the value of this metric ranges between 0 and 1.

Problem (1) is usually solved by applying the max-flow min-cut theorem and converting it into a maximum-flow problem that has an efficient specialized solver. As said in *Section 1*, it is not possible to put this kind of solver inside a natural network and train it end-to-end, due to its non-continuous nature and the lack of a differentiable closed-form solution. Over the years multiple approaches have been proposed to solve this problem. In [1] it

has been proposed a way to include quadratic optimization problems

$$\begin{aligned} \boldsymbol{x}^* = \underset{\boldsymbol{z}}{arg\,min} \frac{1}{2}\boldsymbol{z}^T Q\boldsymbol{z} + \boldsymbol{c}^T\boldsymbol{z} \\ \text{subject to } A\boldsymbol{z} = b \\ G\boldsymbol{z} \leq \boldsymbol{z} \end{aligned} \tag{10}$$

into a neural network layer. To compute the gradient of (10) the implicit function theorem has been used over the Karush–Kuhn–Tucker conditions of the problem. The KKT conditions are implicit functions that are satisfied when the solution of the problem is optimal, so their gradient will point towards the optimal solution. Unfortunately, the method proposed in (10) has cubic complexity in the number of variables, making it a viable solution only for small problems. This method could be applied to the graph cut problem (1) through a linear problem relaxation shown in [24]

$$\begin{aligned} \boldsymbol{d}^* = \underset{\boldsymbol{d}}{arg\,min} \sum_{\{i,j\}\in E} c_{i,j}d_{i,j} \\ \text{subject to } d_{i,j} - z_i + z_j \geq 0 \quad & \forall\{i,j\} \in E, i \neq s, j \neq t \\ d_{s,j} + z_j \geq 1 \quad & \forall\{s,j\} \in E, j \neq t \\ d_{i,t} - z_i \geq 0 \quad & \forall\{i,t\} \in E, i \neq s \\ d_{i,j} \geq 0 \quad & \forall\{i,j\} \in E \\ z_v \in \mathbb{R} \quad & \forall v \in V \backslash \{s,t\} \end{aligned} \tag{11}$$

where V are the nodes, E the edges and $c_{i,j}$ are the capacities of the graph shown in Figure 1. The variable z_v will be 1 if the node v is in the source partition and $d_{i,j}$ will be 1 if the edge $\{i,j\}$ is in the cut.

[4] uses a completely different approach to differentiate any algorithm, even discrete, by computing it multiple times with stochastically perturbed inputs and using the results to estimate a gradient. Given an algorithm $F(\theta)$ and a noise vector Z with density function $f(x) \propto e^{-\nu(x)}$ and a temperature parameter $\epsilon > 0$, then the jacobian of F is

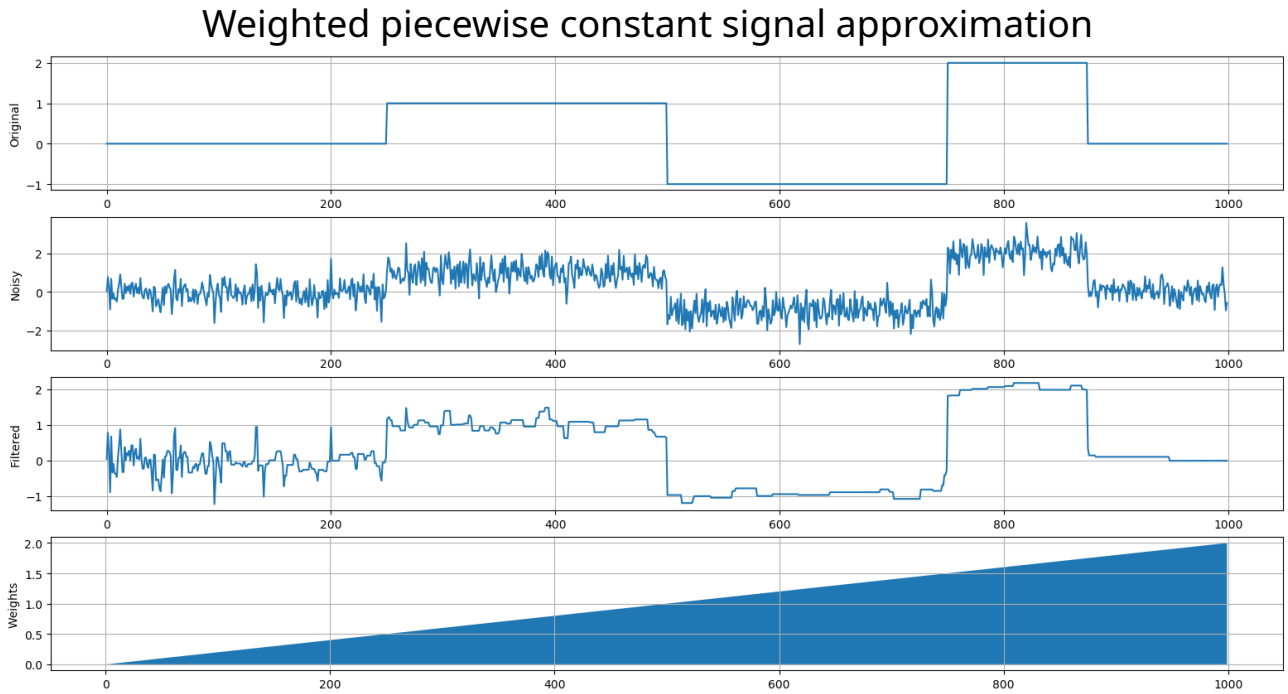
$$J_\theta F(\theta) = \mathbf{E}\left[F(\theta + \epsilon Z)\frac{\nabla_x \nu(Z)^T}{\epsilon}\right] \tag{12}$$

that can be estimated without a bias using a Monte-Carlo approach. Doing so requires computing multiple times $F(\theta+\epsilon Z)$ with the multiple noise vectors and then averaging the results, but this has the disadvantageous need to execute F multiple times, which usually is an expensive operation.

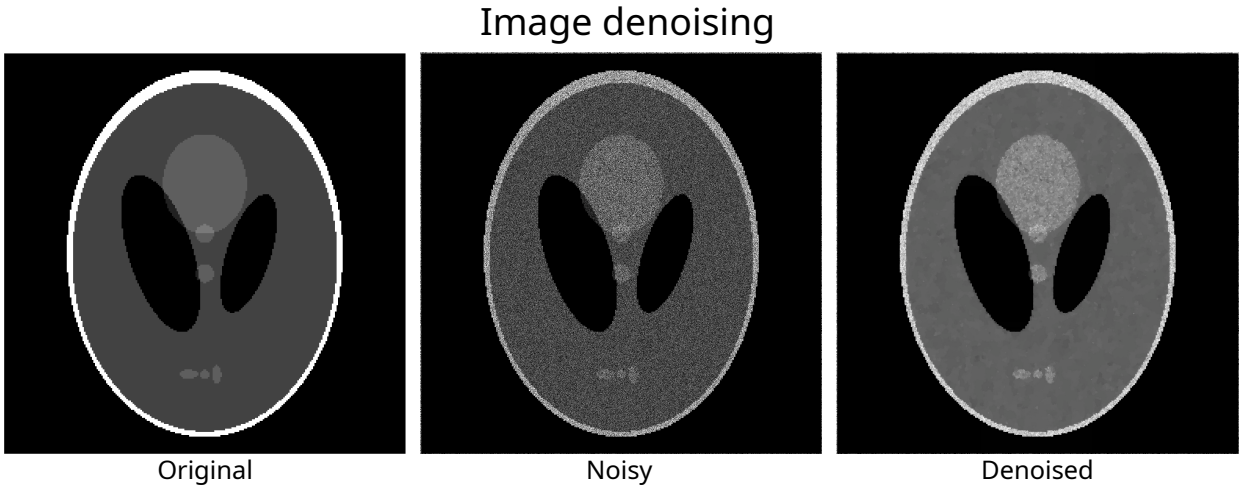
Finally [8] proposes a method that uses the theory of submodular functions to formulate an approximated gradient of a relaxed form of the graph cut problem, the total variation minimization. The total variation minimization problem can be formulated for an image as

$$Z^* = \underset{Z}{arg\,min}\ ||X - Z||_2^2 + \sum_{\substack{i\in\{1,\dots,h\} \\ j\in\{1,\dots,w-1\}}} r_{i,j}|Z_{i,j} - Z_{i,j+1}| + \sum_{\substack{i\in\{1,\dots,h-1\} \\ j\in\{1,\dots,w\}}} c_{i,j}|Z_{i,j} - Z_{i+1,j}| \tag{13}$$

Where X are the pixel values of the grayscale image of size $h \times w$, $r_{i,j}$ are the weights of the rows and $c_{i,j}$ are the weights of the columns.



(a) Example of the result of the weighted total variation minimization on a noisy 1d signal.



(b) Example of the result of the total variation minimization on a noisy image.

Figure 3: Examples of application of the total variation minimization to 1d and 2d inputs (credits to [26]).

The effect of (13) on an image, as shown in Figure 3b, is to quantize its values in order to have neighbor pixels that share the same value. Doing so on the pixel scores results in connected areas of the image whose pixels can be classified all in the same way. The edge weights $r_{i,j}$ and $c_{i,j}$ represent how much the two pixels that they connect need to share the same value. In [8], the proposed method is implemented as a PyTorch [19] layer that, in the forward pass, executes (13), implemented by [26] (that itself is based on the work of [2] and [3]), while the backward pass is obtained from the theory of submodular function as

Algorithm 1 Backward pass from [8]

1: **procedure** TV_BACKWARD(out, grad_out) ▷ *out* is the result of the forward pass and *grad_out* is the gradient w.r.t. it
2: $values \leftarrow$ distinct values of out
3: **for all** $value$ in $values$ **do**
4: $mask \leftarrow out = value$ ▷ $mask$ is *true* in the positions where out has the value $value$, *false* everywhere else
5: $grad\_x[mask] \leftarrow -mean(grad\_out[mask])$ ▷ $array[mask]$ access only the elements of the $array$ where $mask$ is *true*
6: **end for**
7: $diffs\_row \leftarrow sign(opt[:, :-1] - opt[:, 1:])$ ▷ the slicing operator follows the same syntax as in python
8: $grad\_weights\_row \leftarrow diffs\_row * (grad\_x[:, :-1] - grad\_x[:, 1:])$
9: $diffs\_col \leftarrow sign(opt[:-1, :] - opt[1:, :])$
10: $grad\_weights\_col \leftarrow diffs\_col * (grad\_x[:-1, :] - grad\_x[1:, :])$
11: **return** $grad\_x$, $grad\_weights\_row$, $grad\_weights\_col$
12: **end procedure**

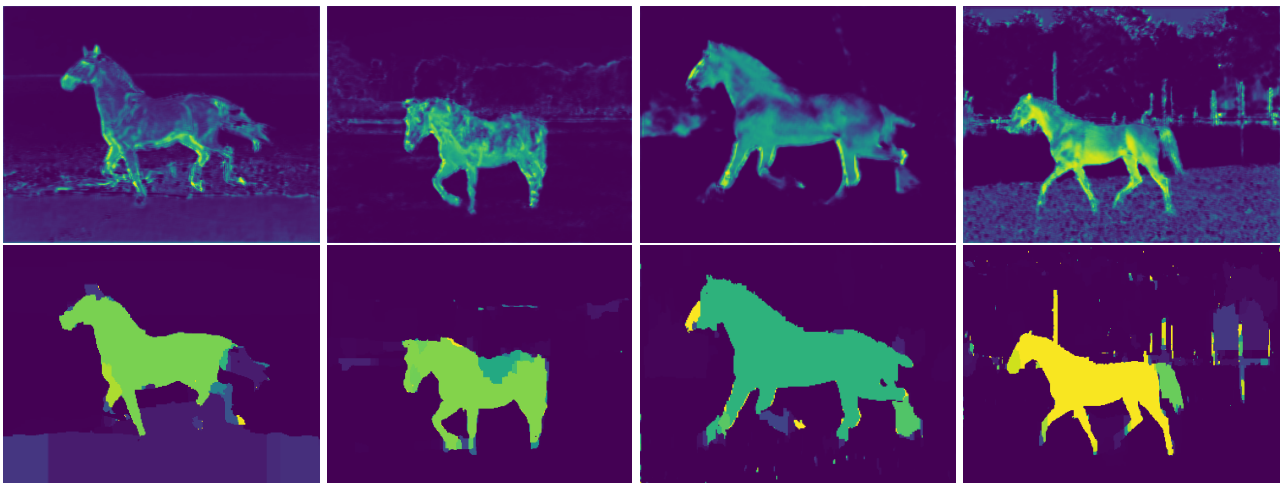
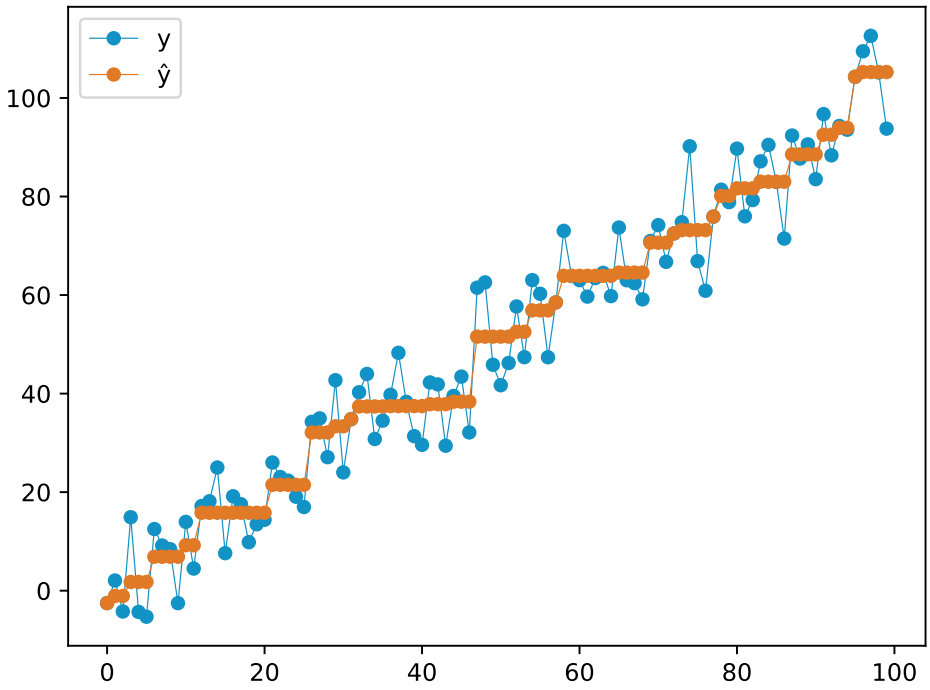


Figure 4: Above the results on a CNN trained to segment the horse in the image, below the result of the same network trained with the additional submodular layer (credits to [8]).

The implementation of this layer from [7] (that is the library used for testing in *Section 3*), offers two more improvements to the one from [8]. First of all, instead of a mask (line 4 of Algorithm 1) for each value of the result of the total variation minimization, a mask for each group of connected pixels with the same value has been used instead, so that, if two groups of pixels have been assigned to the same value but are in different portions of the image, their gradient it is now independent of each other. Moreover, the library offers the possibility to refine the output of the total variation minimization using the isotonic regression algorithm from [20].



(a) Result of the isotonic regression on $\boldsymbol{y}$.

$$\hat{\boldsymbol{y}}^* = \underset{\hat{\boldsymbol{y}}}{arg\,min} \sum_{i=1}^{n} w_i(\hat{y}_i - y_i)^2$$
$$\text{subject to } \hat{y}_i \leq \hat{y}_{i+1} \ \forall y_i \in \boldsymbol{y}$$

(b) Formulation of the isotonic regression.

Figure 5: Example of isotonic regression for a 1d vector $\boldsymbol{y}$, with weights $\boldsymbol{w} = [1, \ldots, 1]$. The Figure 5a shows how the isotonic regression tends to quantize the values of its input, producing a result that is much less affected by noise.

Finally, the conjugate gradient method [12], used by SoftCut to compute its approximated solution, is a technique used to solve numerically and iteratively a system of linear equations, whose matrix is positive-definite. Unlike other iterative techniques, like gradient descent, it converges faster (as shown in Figure 6), making it a great choice when it is applicable.

Algorithm 2 Conjugate Gradient Method

procedure CGM($\boldsymbol{A}, \boldsymbol{b}, \boldsymbol{x}_0$) ▷ $\boldsymbol{A}$ and $\boldsymbol{b}$ are the values of system $\boldsymbol{Ax} = \boldsymbol{b}$ and $\boldsymbol{x}_0$ is the initialization of $\boldsymbol{x}$

  $\boldsymbol{r}_0 \leftarrow \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_0$

  if $\boldsymbol{r}_0$ is small enough **then**

    return $\boldsymbol{x}_0$

  end if

  $\boldsymbol{p}_0 \leftarrow \boldsymbol{r}_0$

  $k \leftarrow 0$

  loop

    $\alpha_k \leftarrow \frac{\boldsymbol{r}_k^T \boldsymbol{r}_k}{\boldsymbol{p}_k^T \boldsymbol{A} \boldsymbol{p}_k}$

    $\boldsymbol{x}_{k+1} \leftarrow \boldsymbol{x}_k + \alpha_k \boldsymbol{p}_k$

    $\boldsymbol{r}_{k+1} \leftarrow \boldsymbol{r}_k - \alpha_k \boldsymbol{A} \boldsymbol{p}_k$

    if $\boldsymbol{r}_{k+1}$ is small enough **then**

      exit loop

    end if

    $\beta_k \leftarrow \frac{\boldsymbol{r}_{k+1}^T \boldsymbol{r}_{k+1}}{\boldsymbol{r}_k^T \boldsymbol{r}_k}$

    $\boldsymbol{p}_{k+1} \leftarrow \boldsymbol{r}_{k+1} - \beta_k \boldsymbol{p}_k$

    $k \leftarrow k + 1$

  end loop

  return $\boldsymbol{x}_{k+1}$

end procedure

The initialization values $\boldsymbol{x}_0$ are usually all zeros or some sort of estimation of the final results. Note that the most expensive operations in the Algorithm 2 are algebraic sums, scalar products and matrix-vector products, that can be executed in parallel on a GPU.

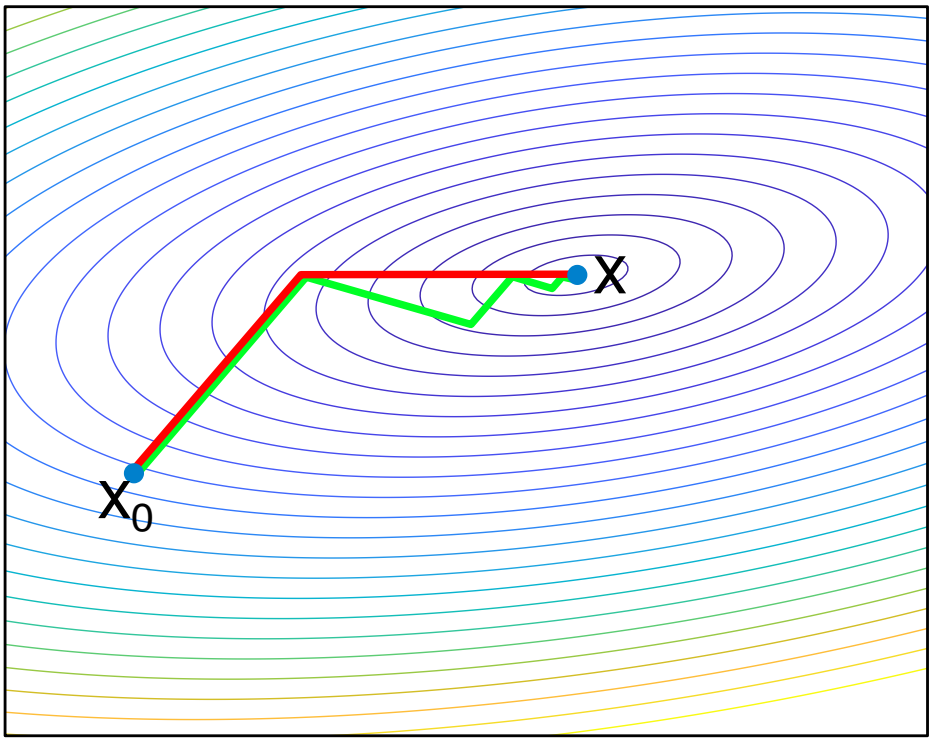


Figure 6: A comparison of the conjugate gradient method (in red) and gradient descent with optimal step size (in green) for minimizing the quadratic function associated with the example linear system $\left[\begin{smallmatrix} 4 & -2 \\ -2 & 9 \end{smallmatrix}\right]\left[\begin{smallmatrix} x_1 \\ x_2 \end{smallmatrix}\right] = \left[\begin{smallmatrix} 1 \\ 2 \end{smallmatrix}\right]$. While gradient descent uses multiple steps in order to get infinitesimally near to the solution $\boldsymbol{X}$, the conjugate gradient method uses at most n steps (n is the size of the matrix of the system) when the arithmetic operations are computed exactly.

2. Method

SoftCut takes as input the features of an image, as shown in Figure 7, of size $h \times w$ in the form of three matrices:

- A pixel scores matrix $P \in \mathbb{R}^{h \times w}$ whose elements $p_{i,j}$ tend to $+\infty$ proportionally to how much the pixel $\{i, j\}$ is certain to belong in the foreground, or to $-\infty$ if it is thought to be in the background;
- A column edge weights matrix $C \in \mathbb{R}_{+}^{h-1 \times w}$ whose elements $c_{i,j}$ are proportional to the strength of the column edge between the pixels $\{i, j\}$ and $\{i+1, j\}$;
- A row edge weights matrix $R \in \mathbb{R}_{+}^{h \times w-1}$ whose elements $r_{i,j}$ are proportional to the strength of the row edge between the pixels $\{i, j\}$ and $\{i, j+1\}$.

Both $c_{i,j}$ and $r_{i,j}$ tend to $+\infty$ if two pixels of the edge are thought to be in the same class or to 0 otherwise.

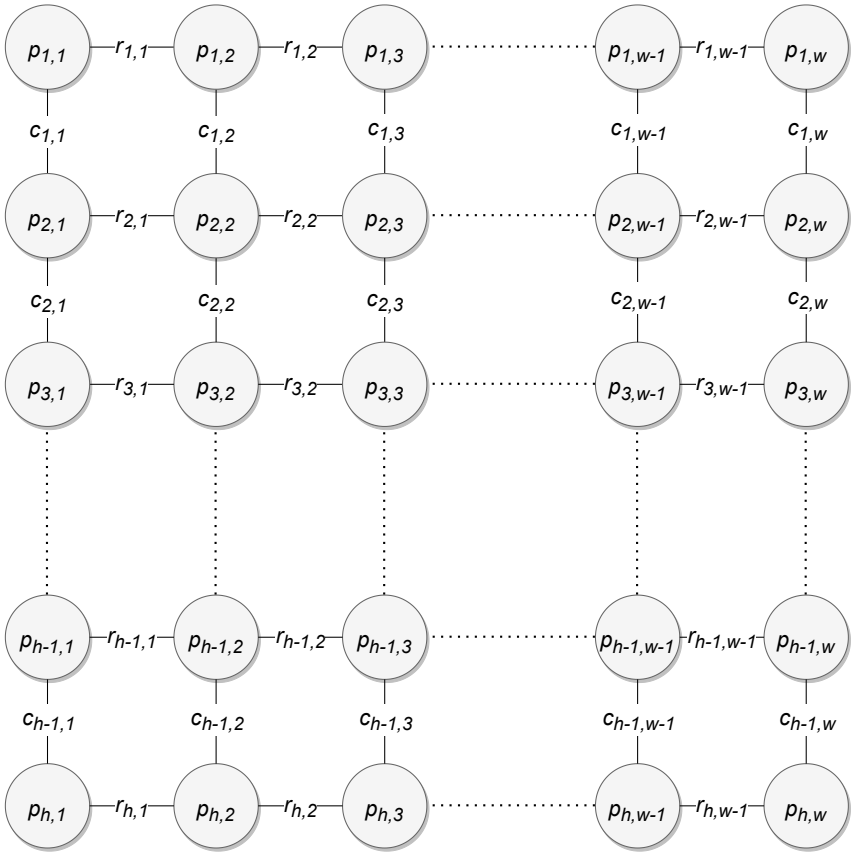


Figure 7: Features of a $h \times w$ image required by SoftCut

The output is defined as the solution $\boldsymbol{x} \in \mathbb{R}^{1 \times hw}$ (whose elements, in the same way of those of P, represent how much a pixel i, j belongs to the foreground or the background) of the linear system

$$A\boldsymbol{x} = \boldsymbol{b} \tag{14}$$

with $A \in \mathbb{R}^{hw \times hw}$ and $\boldsymbol{b} \in \mathbb{R}^{hw \times 1}$ defined as

$$A_{k,l} = \begin{cases} 1 + \tilde{c}_{i-1,j} + \tilde{c}_{i,j} + \tilde{r}_{i,j-1} + \tilde{r}_{i,j} & \text{if } k = l = flat(i,j) \\ -\tilde{c}_{i-1,j} & \text{if } k = flat(i,j), l = flat(i-1,j) \\ -\tilde{c}_{i,j} & \text{if } k = flat(i,j), l = flat(i+1,j) \\ -\tilde{r}_{i,j-1} & \text{if } k = flat(i,j), l = flat(i,j-1) \\ -\tilde{r}_{i,j} & \text{if } k = flat(i,j), l = flat(i,j+1) \\ 0 & \text{otherwise} \end{cases} \tag{15}$$

$$\boldsymbol{b} = \begin{bmatrix} p_{0,0} \\ \vdots \\ p_{m,n} \end{bmatrix} \tag{16}$$

with $k, l \in \{1, 2, \dots, hw\}$, $flat(i, j)$ is the invertible function used to convert 2d indices $\{i, j\}$ of the pixels of the input image to a single dimension one used in A and $\boldsymbol{b}$, $\tilde{c}_{i,j} = \begin{cases} c_{i,j} & \text{if } 1 \leq i \leq h-1, 1 \leq j \leq w \\ 0 & \text{otherwise} \end{cases}$ and $\tilde{r}_{i,j} = \begin{cases} r_{i,j} & \text{if } 1 \leq i \leq h, 1 \leq j \leq w-1 \\ 0 & \text{otherwise} \end{cases}$ ($\tilde{c}$ and $\tilde{r}$ are used to conveniently take care of the cases when the pixels are on the border of the image and so do not have an edge on one or two sides).
Note that A, while having a size of $hw \times hw$, is almost empty, with at most 5 entries per row, allowing the possibility to efficiently store it in memory as a sparse matrix with a space usage of $O(5hw)$ and solve the system with a sparse linear solver like [18] (the system has exactly one solution as stated by Rouché–Capelli theorem, because from Theorem 2.2 it is known that $det(A) > 0$, so $rank(A) = rank([A|b]) = hw$).
For the backward pass of the backpropagation algorithm it is needed to compute the gradient of the loss function L w.r.t. the inputs of the SoftCut layer, given $\frac{\partial L}{\partial \boldsymbol{x}}$. Applying the chain rule it is possible to decompose $\frac{\partial L}{\partial P}$, $\frac{\partial L}{\partial C}$ and $\frac{\partial L}{\partial R}$ in the following way

$$\frac{\partial L}{\partial P} = \frac{\partial L}{\partial \boldsymbol{b}} \cdot \frac{\partial \boldsymbol{b}}{\partial P}, \tag{17}$$

$$\frac{\partial L}{\partial C} = \frac{\partial L}{\partial A} \cdot \frac{\partial A}{\partial C}, \tag{18}$$

$$\frac{\partial L}{\partial R} = \frac{\partial L}{\partial A} \cdot \frac{\partial A}{\partial R}. \tag{19}$$

The $\frac{\partial \boldsymbol{b}}{\partial P}$, $\frac{\partial A}{\partial C}$ and $\frac{\partial A}{\partial R}$ terms from (17), (18) and (19) can be trivially computed because they only involve algebraic sums and multiplications by a constant. As shown in [10]

$$\begin{aligned} \frac{\partial L}{\partial \boldsymbol{b}} &= \frac{\partial L}{\partial x_i} \frac{\partial x_i}{\partial b_j} \\ &= \frac{\partial L}{\partial x_i} \frac{\partial}{\partial b_k} (A^{-1}_{ij} b_j) \\ &= \frac{\partial L}{\partial x_i} A^{-1}_{ij} \frac{\partial b_j}{\partial b_k} \\ &= \frac{\partial L}{\partial x_i} A^{-1}_{ij} \delta_{jk} \\ &= \frac{\partial L}{\partial x_i} A^{-1}_{ik} \\ &= (A^{-1})^T \frac{\partial L}{\partial \boldsymbol{x}} \\ &= solve(A^T, \frac{\partial L}{\partial \boldsymbol{x}}) \end{aligned} \tag{20}$$

$$\begin{aligned} \frac{\partial L}{\partial A} &= \frac{\partial L}{\partial x_i} \frac{\partial x_i}{\partial A_{mn}} \\ &= \frac{\partial L}{\partial x_i} \frac{\partial}{\partial A_{mn}} (A^{-1}_{ij} b_j) \\ &= -\frac{\partial L}{\partial x_i} A^{-1}_{ij} \frac{\partial A_{jk}}{\partial A_{mn}} A^{-1}_{kl} b_l \\ &= -\frac{\partial L}{\partial x_i} A^{-1}_{ij} \delta_{jm} \delta_{kn} A^{-1}_{kl} b_l \\ &= -\frac{\partial L}{\partial x_i} A^{-1}_{im} A^{-1}_{nl} b_l \\ &= -((A^{-1})^T \frac{\partial L}{\partial \boldsymbol{x}}) \otimes (A^{-1} \boldsymbol{b}) \\ &= -\frac{\partial L}{\partial \boldsymbol{b}} \otimes \boldsymbol{x} \end{aligned} \tag{21}$$

As stated by Theorem 2.1, A is symmetric, so $\frac{\partial L}{\partial \boldsymbol{b}}$ in (20) becomes the solution of the system $A\boldsymbol{x} = \frac{\partial L}{\partial \boldsymbol{x}}$ that can be solved in the same way as (14).

It is possible to generalize the SoftCut formulation to work with d-dimension segmentation. Given a d-dimension volume of size $n_1 \times n_2 \times \cdots \times n_d = N$, the inputs of the layer now become $d + 1$ matrices. The pixels scores matrix $P \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_d}$ is now d-dimensional. Every pixel of this volume has at most (when not on the

border of the volume) $2 \cdot d$ edges that are stored in d matrices $E^1, E^2, \ldots, E^d$ with sizes $[n_1 - 1 \times n_2 \times \cdots \times n_d], [n_1 \times n_2 - 1 \times \cdots \times n_d], \ldots, [n_1 \times n_2 \times \cdots \times n_d - 1]$ and elements $e^m_{i_1,i_2,\ldots,i_d} \in \mathbb{R}_+, m \in \{1, 2, \ldots, d\}$. In this formulation $A \in \mathbb{R}^{N \times N}$ and $\boldsymbol{b} \in \mathbb{R}^{N \times 1}$ are defined as

$$A_{k,l} = \begin{cases} 1 + \tilde{e}^1_{i_1-1,i_2\ldots,i_d} + \tilde{e}^1_{i_1,i_2\ldots,i_d} & \\ \quad + \ldots & \text{if } k = l = flat(i_1, i_2 \ldots, i_d) \\ \quad + \tilde{e}^d_{i_1,i_2\ldots,i_d-1} + \tilde{e}^d_{i_1,i_2\ldots,i_d} & \\ -\tilde{e}^1_{i_1-1,i_2\ldots,i_d} & \text{if } k = flat(i_1, i_2 \ldots, i_d), l = flat(i_1 - 1, i_2 \ldots, i_d) \\ -\tilde{e}^1_{i_1,i_2\ldots,i_d} & \text{if } k = flat(i_1, i_2 \ldots, i_d), l = flat(i_1 + 1, i_2 \ldots, i_d) \\ & \vdots \\ -\tilde{e}^d_{i_1,i_2\ldots,i_d-1} & \text{if } k = flat(i_1, i_2 \ldots, i_d), l = flat(i_1, i_2 \ldots, i_d - 1) \\ -\tilde{e}^d_{i_1,i_2\ldots,i_d} & \text{if } k = flat(i_1, i_2 \ldots, i_d), l = flat(i_1, i_2 \ldots, i_d + 1) \\ 0 & \text{otherwise} \end{cases} \tag{22}$$

$$\boldsymbol{b} = \begin{bmatrix} p_{0,0,\ldots,0} \\ \vdots \\ p_{n_1,n_2,\ldots,n_d} \end{bmatrix} \tag{23}$$

with $k, l \in \{1, 2, \ldots, N\}$, $flat(i_1, i_2 \ldots, i_d)$ is the invertible function used to convert d-dimensional indices $\{d_1, d_2, \ldots, d_n\}$ of the pixels of the input volume to a single dimension one used in A and $\boldsymbol{b}$ and lastly $\tilde{e}^m_{i_1,i_2,\ldots,i_d} = \begin{cases} e^m_{i_1,i_2,\ldots,i_d} & \text{if } i_1, i_2, \ldots, i_d \text{ are in the bounds of } E^m \\ 0 & \text{otherwise} \end{cases}$. In this formulating the space usage of A is $O((1+2d)N)$, still linear w.r.t the number of pixels and both the system and the gradient can be solved in the same way as for the 2d formulation.

2.1. Equivalence to an Electrical Circuit

The linear system of equations solved by SoftCut represents an electric circuit capable of computing a relaxed version of the graph cut problem. The circuit is built by repeating the module shown in Figure 8 once for each pixel of the input, keeping their grid alignment as in Figure 7.

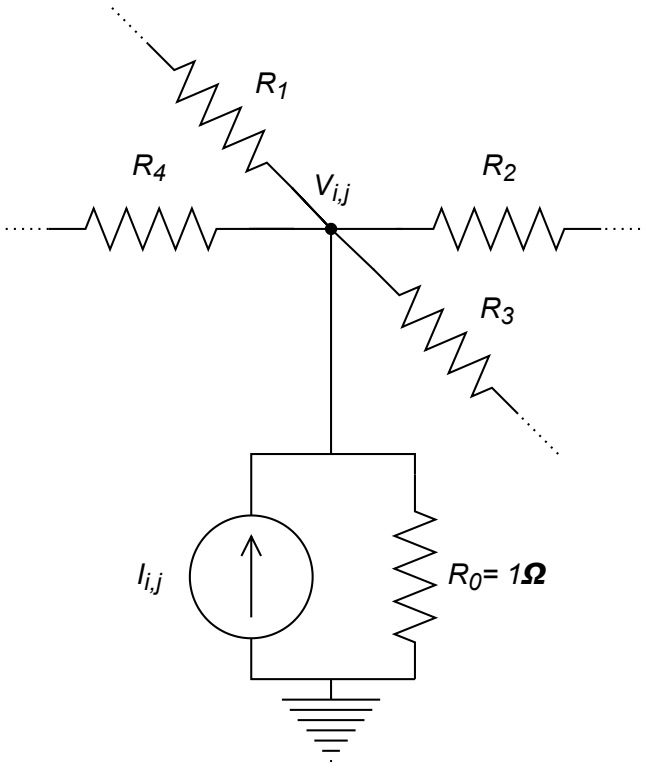


Figure 8: The module of the circuit for each pixel

The resistors R_1, R_2, R_3 and R_4 are in common by each two adjacent modules (and ignored when pixels are on the border of the image and so the resistor is connected to nothing) so that, for example, resistor R_2 of pixel $\{i, j\}$ is the resistor R_4 of pixel $\{i, j+1\}$. The current of the generator $I_{i,j}$ is set equal to the score $p_{i,j}$ of said pixel. The conductance of resistors R_1 and R_3 are provided by the edge weights $c_{i-1,j}$ and $c_{i,j}$ from matrix C, while the ones of resistors R_2 and R_4 came from $r_{i,j-1}$ and $r_{i,j}$ of matrix R. Finally the system is solved to find the values of the various $V_{i,j}$.

It is possible to understand the functioning of this approach by building more and more complex examples. For starters, let us consider the case of an image made up of a single pixel: resistors R_1, R_2, R_3 and R_4 would be connected to nothing, so it is possible to ignore them. In this setup, $V_{i,j}$ becomes equal to the voltage drop across resistor R_0, that is, by Ohm's law, $R_0 \cdot I_{i,j} = 1 \cdot I_{i,j} = I_{i,j}$, so the result is the same pixel score received as input. Now let us consider the case where only two pixels $\{i, j\}$ and $\{i, j+1\}$ with the conductance of the resistor between them equals to $+\infty\,\Omega^{-1}$ (so the resistance is $0\,\Omega$), as shown in Figure 9.

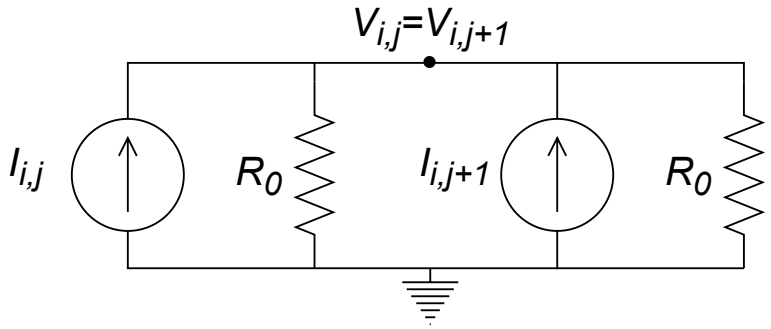


Figure 9: SoftCut in the case of two pixels and a very strong edge between them

Now $V_{i,j} = V_{i,j+1}$ is equal to the voltage drop across the resistor equivalent to the two resistors R_0 in parallel, that is $R_{eq} = \frac{1}{\frac{1}{1\,\Omega}+\frac{1}{1\,\Omega}} = \frac{1}{2}\,\Omega$. According to Ohm's law, $V_{i,j} = V_{i,j+1} = R_{eq} \cdot (I_{i,j} + I_{i,j+1}) = \frac{I_{i,j}+I_{i,j+1}}{2}$, so the result is the average between the two pixel scores. In the general case with N pixels connected with zero resistance as before, the equivalent resistor will have resistance $R_{eq} = \frac{1}{N}\,\Omega$, so the resulting voltage, and therefore the solution, for all of these pixels is still the average of their pixel score. This property makes it so that pixels scores with very high absolute value can compensate for those that are connected to them, but are not as sure (Figure 10).

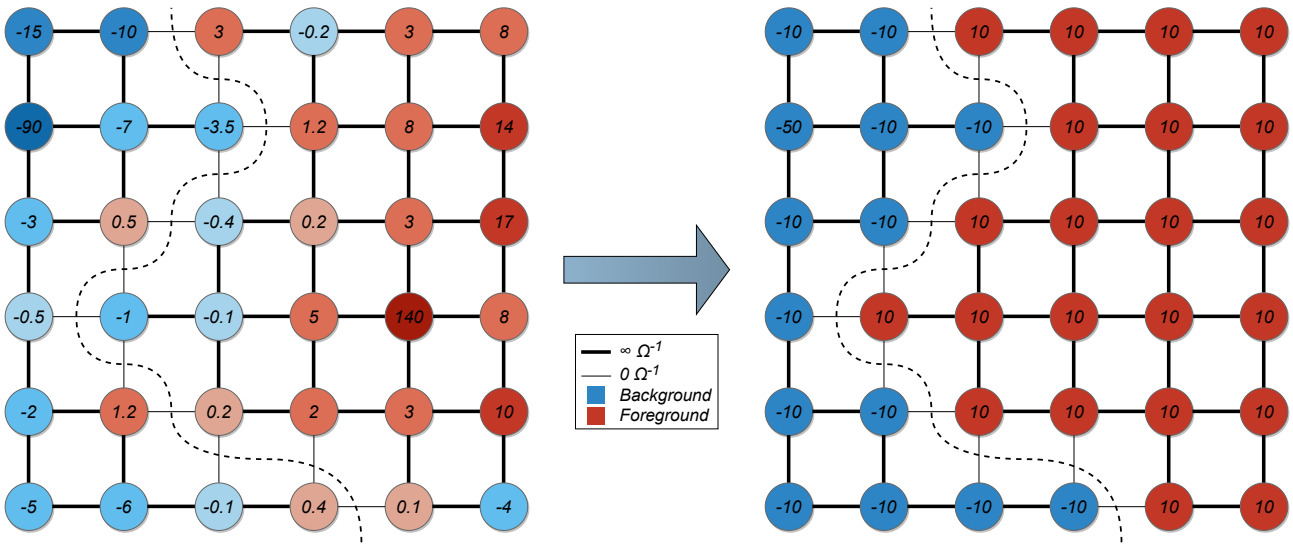


Figure 10: Example of an input and output of SoftCut in the case of perfect edges ($0\Omega^{-1}$ and $+\infty\Omega^{-1}$) that partition the graph exactly around the dashed line

If, instead, the resistor between $\{i, j\}$ and $\{i, j+1\}$ has a conductance $\leq +\infty\,\Omega^{-1}$, the circuit becomes as show in Figure 11.

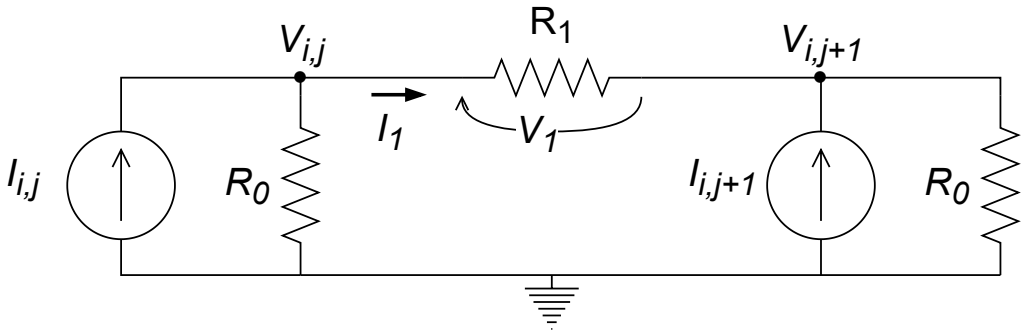


Figure 11: SoftCut in the case of two pixels and a medium edge between them

In this case, $V_{i,j} \neq V_{i,j+1}$, but it still possible to calculate them by applying Kirchhoff's voltage law (24a) and Kirchhoff's current law (24b) (24c) on the nodes of the voltages.

$$\begin{cases} V_{i,j} = V_1 + V_{i,j+1}, & \text{(24a)} \\ I_{i,j} = I_1 + \dfrac{V_{i,j}}{R_0}, & \text{(24b)} \\ I_{i,j+1} + I_1 = \dfrac{V_{i,j+1}}{R_0}. & \text{(24c)} \end{cases}$$

The solution of Equations (24) is $V_{i,j} = \frac{G_1(I_{i,j}+I_{i,j+1})+I_{i,j}}{2G_1+1}, V_{i,j+1} = \frac{G_1(I_{i,j}+I_{i,j+1})+I_{i,j+1}}{2G_1+1}$, with $G_1 = \frac{1}{R_1}$, that shows how $V_{i,j}$ and $V_{i,j+1}$ gradually go from $I_{i,j}$ and $I_{i,j+1}$ to $\frac{I_{i,j}+I_{i,j+1}}{2}$ when G_1 goes from $0\,\Omega^{-1}$ to $+\infty\,\Omega^{-1}$. Doing this, SoftCut is able to propagate the influence of a pixel score to its neighbors in a controlled way, using the information provided by the weights of the edges (Figure 12).

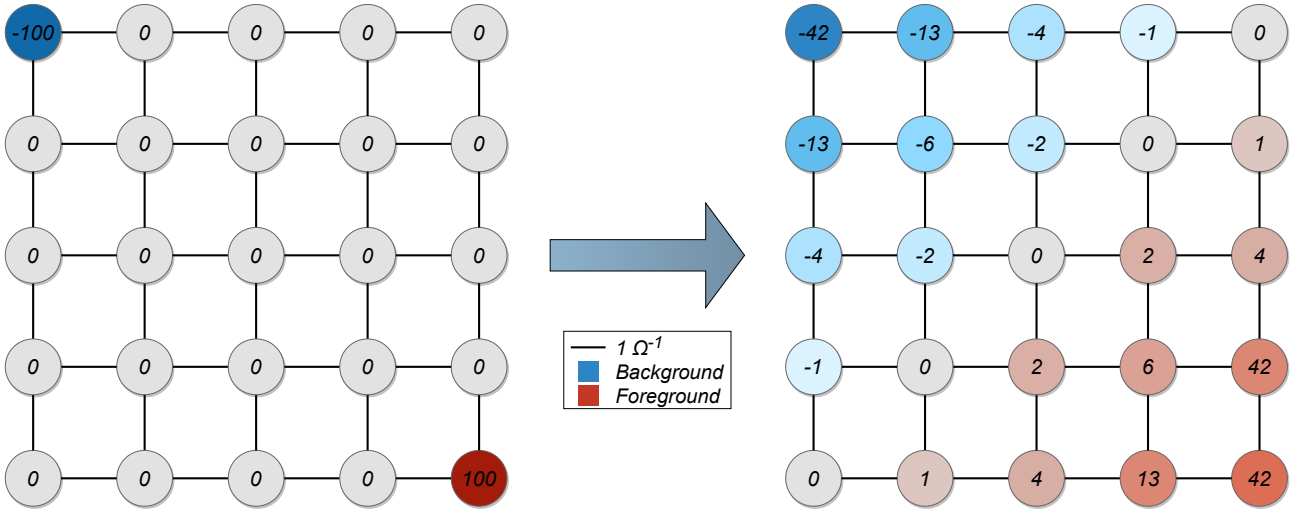


Figure 12: Example of an input and output of SoftCut in the case of soft edges

These results show how the weights of the edges provide the shape of how the image should be partitioned, while the pixel score values diffuse in these shapes. In this way, the network before the SoftCut layer only needs to compute how much two adjacent pixels are correlated to each other and an estimate measure of the belonging of each pixel to the background or the foreground class. Then SoftCut is able to compute the final score of each pixel by smoothing out uncertainties or disagreements of the pixel scores while taking into account how correlated they are.

System (14) is obtained, for each row, by writing down the Equation (25) obtained by Kirchhoff's current law. Considering the naming used in Figure 8, it is possible to write:

$$
\begin{aligned}
I_{i,j} + \frac{V_{i-1,j} - V_{i,j}}{R_1} + \frac{V_{i,j-1} - V_{i,j}}{R_4} &= \frac{V_{i,j}}{R_0} + \frac{V_{i,j} - V_{i+1,j}}{R_3} + \frac{V_{i,j} - V_{i,j+1}}{R_2} \\
(\frac{1}{R_0} + \frac{1}{R_1} + \frac{1}{R_2} + \frac{1}{R_3} + \frac{1}{R_4})V_{i,j} - \frac{V_{i-1,j}}{R_1} - \frac{V_{i,j+1}}{R_2} - \frac{V_{i+1,j}}{R_3} - \frac{V_{i,j-1}}{R_4} &= I_{i,j} \\
(1 + G_1 + G_2 + G_3 + G_4)V_{i,j} - G_1V_{i-1,j} - G_2V_{i,j+1} - G_3V_{i+1,j} - G_4V_{i,j-1} &= I_{i,j} \\
(1 + c_{i-1,j} + r_{i-1,j} + c_{i,j} + r_{i,j-1})x_{i,j} - c_{i-1,j}x_{i-1,j} - r_{i,j}x_{i,j+1} - c_{i,j}x_{i+1,j} - r_{i,j-1}x_{i,j-1} &= p_{i,j}
\end{aligned}
\tag{25}
$$

In the case of a d-dimensional segmentation, the pixels would have, as said in *Section 2*, $2 \cdot d$ neighbors. Modifying the circuit in Figure 8 by adding a resistor for each one of the neighbors of pixel $\{i, j\}$ it is possible to obtain back the formulations (22) and (23) in the same way as done for the 2d formulation.

2.2. Approximate solution

While algorithms like KLU [18] are very efficient at solving sparse linear systems, they have the downside of being sequential, due to the very nature of the problem. On the other hand, the training of neural networks exploits GPUs' parallel computation capabilities to speed up the whole process. It is then natural to wonder whether it is possible to solve system (14) in parallel on a GPU in order to speed up the computation and avoid the inefficient data transfer between it and the CPU.
Fortunately, it is possible to solve sparse linear systems using an iterative algorithm like the conjugate gradient method [12] on a GPU. This method however requires that matrix A is positive definite, a property that is proved in Theorem 2.2.

Theorem 2.1. *Matrix A (15) is symmetric.*

Proof. Formulation (15) provides a description of matrix A by each row k. The first case is always the diagonal element $\{k, l = k\}$, so it is only needed to check the other entries. The second case says that element $\{k, l\}, k = flat(i, j), l = flat(i - 1, j)$ is $-\tilde{c}_{i-1,j}$, and we can check that the symmetric element $\{k', l'\}, k' = flat(i - 1, j) = l, l' = flat(i, j) = k$ is actually the third case of row k' that is still equal to $-\tilde{c}_{i-1,j}$. In the same way it can be shown the symmetry for the next three cases. Finally, the sixth case applies to only the remaining elements, by assigning to all of them the same value, hence matrix A is symmetric.

Theorem 2.2. *Matrix A (15) is positive definite.*

Proof. Theorem 2.1 states that Matrix A (15) is symmetric. Moreover, A is also strictly diagonally dominant because $\forall k$ is it true that

$$
\begin{aligned}
|a_{k,k}| &> \sum_{k \neq l} |a_{k,l}| \\
|1 + \tilde{c}_{i-1,j} + \tilde{c}_{i,j} + \tilde{r}_{i,j-1} + \tilde{r}_{i,j}| &> |-\tilde{c}_{i-1,j}| + |-\tilde{c}_{i1,j}| + |-\tilde{r}_{i,j-1}| + |-\tilde{r}_{i,j}|
\end{aligned}
$$

because $\tilde{c}_{i-1,j}$, $\tilde{c}_{i,j}$, $\tilde{r}_{i,j-1}$ and $\tilde{r}_{i,j}$ are ≥ 0 by definition. Thus A is positive definite.

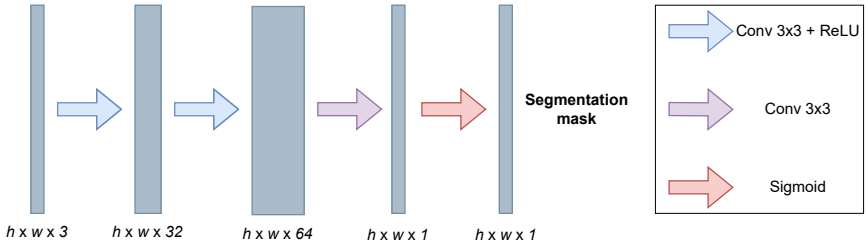
3. Experiments

All the experiments have been implemented in PyTorch [19] and PyTorch Lightning [9] and executed on a machine equipped with an NVIDIA Quadro RTX 6000 GPU paired with an Intel Xeon CPU E5-2687W v4. In all experiments, after the training was completed, the checkpoint with the best validation loss (that was computed at the end of each epoch) has been selected for testing on the test set and the results are shown in Section 3.1. The implementation of the exact solver for the sparse linear system comes from [15], a python library that provides a differentiable PyTorch function capable of solving (14) on the CPU using [18]. The library [15] has also been modified in order to parallelize the computation of the various elements of the mini-batch across multiple cores. The approximate solution of SoftCut is computed by a custom implementation of Algorithm 2 in a PyTorch layer. The Algorithm 2 has been set up in order to stop when $\sum_{r \in \boldsymbol{r}} |r| < 10^{-6} bhw$ (where b is the batch size) and as $\boldsymbol{x}_0$, the initialization value of the result $\boldsymbol{x}$, it has been used $\boldsymbol{b}$, because the pixel scores are a first approximation of the actual solution. Finally, SoftCut has also been tested against the submodular approach from [8], whose implementation has been provided by [7] in the form of a PyTorch layer. Again this library has also been modified in order to parallelize the CPU computations across multiple cores, exploiting the data parallelism between the elements of the mini-batch.

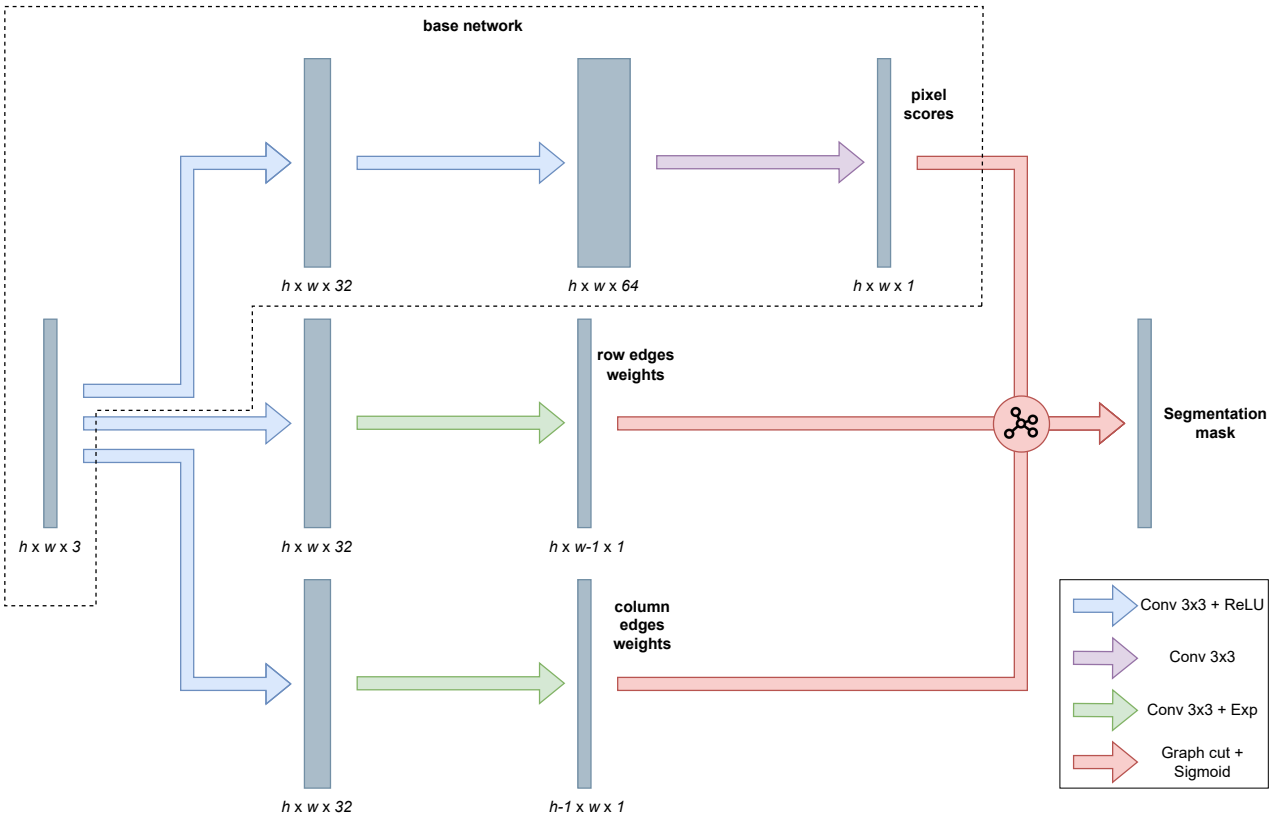
For testing the SoftCut layer it has been added after a base network, followed by an activation function, and trained with it in an end-to-end fashion on different segmentation tasks:

- *Binary Segmentation*: the network predicts the probability of each pixel belonging to one specific class.
- *Specific Object Segmentation*: the network predicts the probability of each pixel belonging to one specific object in the image that is identified by an additional input.
- *Multiclass Segmentation*: the network predicts the probabilities of each pixel belonging to each class.

The base network itself is capable of doing image segmentation, so these tests show how the addition of a graph cut layer could improve the scores. Moreover, the base network is only capable of computing the pixel scores if used in combination with graph cut as is, so some other variations are needed in order to also compute the edges weighs needed. The first architecture tested is the one from [8], shown in Figure 13, in order to see how SoftCut compares to that technique.



(a) The base network alone.

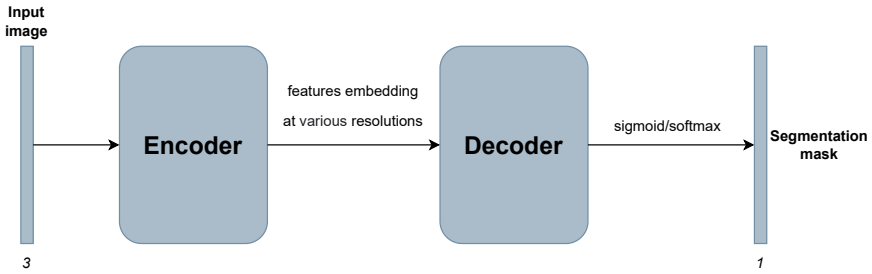


(b) The extended network with the graph cut layer.

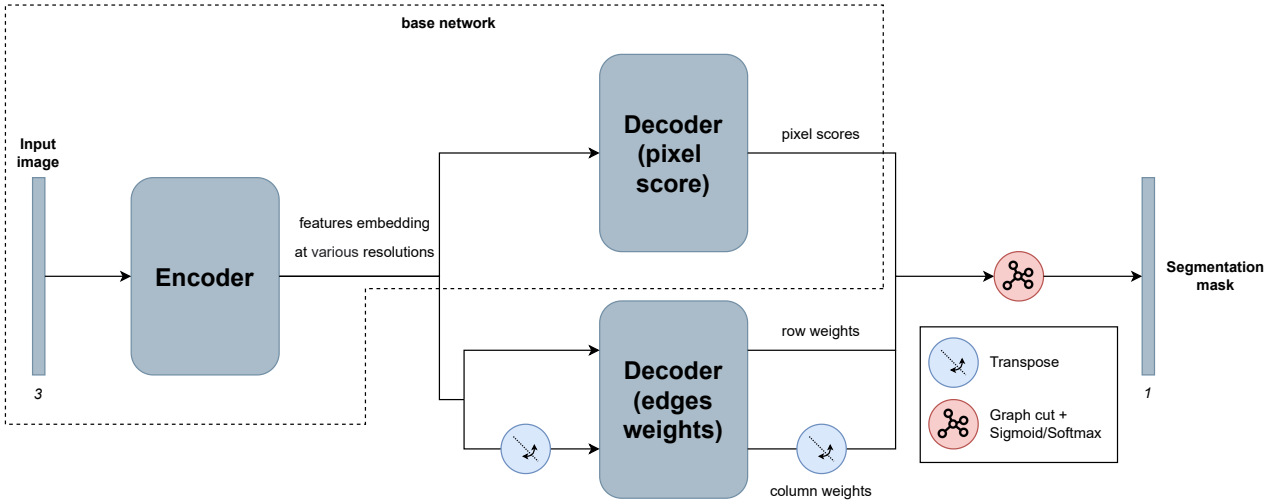
Figure 13: The architecture is made just by 3 convolutions for the base network, that produce the segmentation mask when alone or the pixel scores when in combination with a graph cut layer and the other four convolutions used to compute the edge weights. Note that this architecture does not use any kind of pooling, so the receptive field of each pixel is quite limited.

With this setup, the task is binary segmentation trained on the *Weizmann Horse* dataset, with a split of 180, 50 and 98 for the training, validation and test set. All the models have been trained for 100 epochs, with a batch size of 1, by minimizing the binary cross-entropy on each of the 0.1% of the revealed pixels, using both *Adam* and *Adagrad* as optimizers with a learning rate of 10^{-2} and 10^{-3}, as done in the original paper. The layer from [8] has been tested also with and without applying an isotonic regression to the results of the total variation algorithm, which should improve the quality of the output.

Then SoftCut has also been tested using U-Net as the base network, in order to see if it is able to improve the results of an already state-of-the-art architecture. The implementation of U-Net used came from Segmentation Models Pytorch [25], a python library that conveniently provides different segmentation architectures, with the possibility of selecting the desired encoder and manually composing the complete network by directly accessing the encoder and decoder parts.



(a) The base network alone.



(b) The extended network with the graph cut layer.

Figure 14: The encoders shown are the "downward" part of the U-Net architecture, that outputs the lowest resolution embeddings plus all the previous skip connections. The encoders instead are the "upward" part of U-Net, that takes as input the various resolution features and returns a full-resolution output. Moreover, instead of the exponential function used in Figure 13b for assuring that the edge weights were ≥ 0, the SoftPlus function has been applied for numerical stability.

As shown in Figure 14, two decoders have been used: one for computing the pixel scores and another (not used in the base network alone) for the edge weights. Two different encoder models have been tested: *resnet18* (11M parameters) and *resnet34* (21M parameters), two residual CNNs with 18 and 34 layers respectively, discussed in [11]. The second decoder used to compute the edge weights has been reused for both the column and row weights, exploiting the roto-reflection invariance property of images by transposing the inputs and the output of the decoder. Note that the decoders use much fewer parameters (3M) with respect to the encoder, so a second decoder for the edge weight does not give an unfair advantage to the models with graph cut. For the multiclass segmentation task, the graph cut layer is repeated for each channel and the final sigmoid activation function has been replaced by a softmax one, so a new background class has been introduced for classifying the pixels that do not belong to any other one. This architecture, except when said otherwise, has been trained for 100 epochs, with a batch size of 32, by minimizing the Dice loss using *Adam* as optimizer with a learning rate of 10^{-3}.

This more complex architecture has been tested on different datasets, derived from *Cityscapes*. The images have been divided into tiles of size 256×256 and then the tiles have been sampled in order that the vast majority of them contain the selected classes. The *Cityscapes* original training set has been split into a training and validation one, while the original validation one has been used as test set. Here is the list of datasets obtained in this way:

- Custom Cityscapes for *Binary segmentation*. Used classes: *traffic signs*. 13078 train samples, 1024 validation samples and 2344 test samples.
- Custom Cityscapes for *Specific Object Detection*. Used classes: *traffic signs, cars*. 21644 train samples, 609 validation samples and 3724 test samples. In addition to the three channels of the input image, an additional fourth one of all zeros except a single one in the middle of the selected object has been added.
- Custom Cityscapes for *Multiclass Segmentation*. Used classes: *traffic signs, cars*. 11983 train samples, 877 validation samples and 4833 test samples.

In order to check if SoftCut could work well also in real-world scenarios, a final dataset (*Cityscapes Full Resolution*) has been made from *Cityscapes*. It uses all the training and validation samples at their original full resolution, but, in order to train the networks efficiently, they get cropped into overlapping tiles (of size $256 + 32 \times 256 + 32$) before getting fed to the network, and then the output segmentation masks are recombined back into a full-resolution one. The overlapping between the tiles is done in order to give the network the context needed to classify the pixels on the border of the non-overlapping parts. During the training, the loss function has been computed also on the overlapping parts, while at inference time they got discarded. Because of the limited computation power available for the training and size of the networks, only the *traffic signs, cars* and *persons* classes have been considered.

Moreover, the additional custom loss function (28), discussed in *Appendix B*, has been tested in to see if capable of improving the training on the networks. Finally, because both SoftCut and the technique from [8] ended up computing the edge weights in addition to the pixel scores, it has been tested if replacing those differentiable techniques at inference time (because, in that moment, the constraint of the differentiability of the functions that compose the network is not required anymore) with PyMaxflow [21], a python library, based on the work of [5], for constructing graphs like Figure 1 and solving them as maximum flow problem, could improve the results even more or the relaxation of the problem 1 leads to the learning of unsuitable values for f_i, b_i and $w_{i,j}$.

3.1. Results

The goal of the first experiment was to replicate the results from [8] and see how SoftCut compares in this setup. Multiple combinations of the learning rates and the optimizer have been tested, but, for sake of simplicity, only the result with better mIoU is reported in the table for each model.

	Experiment 1
Architecture	*Basic CNN*
Dataset	*Weizmann Horse*

Table 1: Experiment with the same setup as in [8]

Model	**mIoU**	**loss**	**# distinct figures**	**train time**	**lr**	**optimizer**
Base Network	0.4720 (0.0243)	0.3936 (0.0300)	72.336 (5.5184)	**5min**	*0.001*	*Adam*
Submodular	0.5304 (0.0282)	0.3485 (0.0279)	2.0204 (0.1450)	50min	*0.001*	*Adam*
Submodular (+ isotonic)	**0.6832 (0.0220)**	**0.2637 (0.0176)**	**1.8571 (0.1547)**	50min	*0.001*	*Adam*
SoftCut (exact)	0.5937 (0.0218)	0.3024 (0.0306)	7.4693 (0.8242)	11h35min	*0.001*	*Adam*
SoftCut (approximate)	0.6022 (0.0211)	0.3041 (0.0285)	8.7346 (0.8209)	35min	*0.001*	*Adam*

Table 2: Results of Experiment 1.

From the results, it is clear how all the graph cut methods behave better than the base network alone, probably due to the increased receptive field, as shown in *Appendix A*. Moreover, SoftCut performs better than the method from [8] without the isotonic regression refinement, and the approximate solution is 30% faster than those methods, but, on the other hand, the exact solution slow down the training tremendously.
The following experiments have been executed to see if the previous results were still valid in a more advanced setup, with a better architecture like U-Net, bigger models, better loss functions and a much larger dataset. For the submodular technique, from now on, only the version with the isotonic regression has been tested in order to save training time, because it always outperforms the version without it.

	Experiment 2	Experiment 3	Experiment 4
Architecture	*U-Net*	*U-Net*	*U-Net*
Encoder	*resnet18*	*resnet34*	*resnet18*
Dataset	*Binary Segmentation*	*Binary Segmentation*	*Binary Segmentation*
Optimizer	*Adam*	*Adam*	*Adagrad*

Table 3: The first batch of experiments with U-Net architecture.

Model	mIoU	loss	# distinct figures
Base Network	0.4324 (0.0074)	0.4918 (0.0079)	1.5059 (0.0289)
Submodular (+ isotonic)	0.4141 (0.0076)	0.5188 (0.0081)	**0.9914 (0.017)**
SoftCut (exact)	0.4240 (0.0075)	0.5050 (0.0081)	1.1335 (0.0198)
SoftCut (approximate)	**0.4790 (0.0075)**	**0.4468 (0.0078)**	1.2636 (0.0212)

Table 4: Results of Experiment 2.

Model	mIoU	loss	# distinct figures
Base Network	0.4512 (0.0074)	0.4724 (0.0079)	2.1467 (0.042)
Submodular (+ isotonic)	0.4252 (0.0076)	0.5049 (0.0081)	**1.023 (0.0173)**
SoftCut (exact)	0.4357 (0.0076)	0.4933 (0.0081)	1.1766 (0.0211)
SoftCut (approximate)	**0.4784 (0.0077)**	**0.4512 (0.0081)**	1.1139 (0.0188)

Table 5: Results of Experiment 3.

Model	mIoU	loss	# distinct figures
Base Network	0.3452 (0.0069)	0.5806 (0.0076)	1.63012 (0.0288)
Submodular (+ isotonic)	0.3563 (0.0073)	0.5795 (0.0079)	**0.914249 (0.0168)**
SoftCut (approximate)	**0.3647 (0.0070)**	**0.5640 (0.0076)**	1.25853 (0.0220)

Table 6: Results of Experiment 4.

As shown by the results, it is easy to see that the approximate solution of SoftCut provides the best results in terms of IoU, while the submodular approach seems to return a minor number of distinct figures than SoftCut, at the cost of a much lower score.
The reasons behind why both the approach from [8] and the exact solution of SoftCut perform worse than the base network is not completely clear, but probably the nature of the graph cut problem provides an uneven gradient landscape, that makes it difficult to minimize, while approximating the solution filters out those irregularities.
Moreover, it is interesting to notice that, even with more complex encoders, the approximated SoftCut seems not to increase its performance, differently from the other techniques, but maintains the lead in both cases.

Due to the slow training time of the exact solution of SoftCut and seeing how the approximate version ends with better scores anyway, the following experiments did not include the first method. The next batch of experiments is focused on another task, the *Specific Object Detection*.

	Experiment 5	Experiment 6
Architecture	*U-Net*	*U-Net*
Encoder	*resnet18*	*resnet34*
Dataset	*Specific Object Detection*	*Specific Object Detection*

Table 7: The second batch of experiments with U-Net architecture, over the *Specific Object Detection* dataset.

Model	mIoU	loss	# distinct figures
Base Network	0.8454 (0.0022)	0.0930 (0.0016)	1.3053 (0.0140)
Submodular (+ isotonic)	0.8138 (0.0025)	0.1168 (0.0019)	**1.0048 (0.0011)**
SoftCut (approximate)	**0.8540 (0.0021)**	**0.0886 (0.0015)**	1.0208 (0.0024)

Table 8: Results of Experiment 5.

Model	mIoU	loss	# distinct figures
Base Network	0.8434 (0.0022)	0.0942 (0.0016)	1.39437 (0.0180)
Submodular (+ isotonic)	0.8025 (0.0027)	0.1270 (0.0022)	**1.00229 (0.0016)**
SoftCut (approximate)	**0.8517 (0.0021)**	**0.0907 (0.0016)**	1.0173 (0.0024)

Table 9: Results of Experiment 6.

As before, even in this new task with a different dataset, SoftCut provided the best IoU score, while the submodular approach provides a slightly lower number of distinct figures. Interestingly, the bigger encoder does not seems to provide any benefit to any of the models, probably because of the lower difficulty of this task.

Next, it has been tested if the Focal Dice loss (with $t = 0.5$ and $\gamma = 2$) discussed in *Appendix B* can provide any benefit to the trained models.

	Experiment 7
Architecture	*U-Net*
Encoder	*resnet18*
Dataset	*Binary Segmentation*
Loss	*Focal Dice*

Table 10: Same setup as in Experiment 2, but with the Focal Dice loss instead of the plain Dice loss.

Model	mIoU	loss	# distinct figures
Base Network	*0.4324 (0.0074)*	*0.4918 (0.0079)*	*1.5059 (0.0289)*
Submodular (+ isotonic)	*0.4141 (0.0076)*	*0.5188 (0.0081)*	*0.9914 (0.017)*
SoftCut (exact)	*0.4240 (0.0075)*	*0.5050 (0.0081)*	*1.1335 (0.0198)*
SoftCut (approximate)	*0.4790 (0.0075)*	*0.4468 (0.0078)*	*1.2636 (0.0212)*
Base Network	0.4528 (0.0073)	0.4694 (0.0078)	2.0486 (0.040)
Submodular (+ isotonic)	0.4067 (0.0077)	0.5277 (0.0082)	**0.874147 (0.0180)**
SoftCut (approximate)	**0.4887 (0.0076)**	**0.4391 (0.0079)**	1.2077 (0.0205)

Table 11: Results of Experiment 7 in addition to those of Experiment 2 in italic.

The results, displayed together with those of Experiment 2 in order to compare them, show an improvement for both the base network and SoftCut, as opposed to the submodular approach.

This turn, the use of a non-relaxed graph cut, in the form of PyMaxflow [21], at inference time for the SoftCut and the submodular networks has been tested.

	Experiment 8
Architecture	*U-Net*
Encoder	*resnet18*
Dataset	*Binary Segmentation*
Inference time graph cut	*MaxFlow*

Table 12: Same setup as in Experiment 2, but with the addition of using PyMaxflow at inference time for the graph cut models.

Model	mIoU	loss	# distinct figures
Base Network	*0.4324 (0.0074)*	*0.4918 (0.0079)*	*1.5059 (0.0289)*
Submodular (+ isotonic)	*0.4141 (0.0076)*	*0.5188 (0.0081)*	*0.9914 (0.017)*
SoftCut (exact)	*0.4240 (0.0075)*	*0.5050 (0.0081)*	*1.1335 (0.0198)*
SoftCut (approximate)	*0.4790 (0.0075)*	*0.4468 (0.0078)*	*1.2636 (0.0212)*
Submodular (+ isotonic) + MaxFlow	0.4021 (0.0077)	0.5342 (0.0084)	**0.8805 (0.0162)**
SoftCut (approximate) + MaxFlow	**0.4875 (0.0074)**	**0.4354 (0.0078)**	1.4680 (0.0239)

Table 13: Results of Experiment 8 in addition to those of Experiment 2 in italic

The results, shown again together with those of Experiment 2, show how this solution improves the results of SoftCut while not doing the same for the submodular technique, implying that the pixel scores and the edge weight computed in this case badly adapt to a true graph cut.

Seeing these two final improvements for SoftCut, it has been tested if combining these techniques could improve the IoU score even more.

	Experiment 9
Architecture	*U-Net*
Encoder	*resnet18*
Dataset	*Binary Segmentation*
Loss	*Focal Dice*
Inference time graph cut	*MaxFlow*

Table 14: Same setup as in Experiment 8, but using the Focal Dice as loss function.

Model	mIoU	loss	# distinct figures
SoftCut (approximate) + MaxFlow	**0.4911 (0.0074)**	**0.4334 (0.0079)**	1.5849 (0.0255)

Table 15: Results of Experiment 9.

The result of this experiment shows that both the Focal Dice loss and the non-relaxed graph cut can improve the performance of SoftCut.

Afterward, it has been tested how SoftCut behaves in a multiclass configuration. Each score has been computed as the average between all the values of each class, including the background.

	Experiment 10
Architecture	*U-Net*
Encoder	*resnet18*
Dataset	*Multiclass Segmentation*

Table 16: Same setup as in Experiment 2, but used against a multiclass dataset.

Model	mIoU	loss	# distinct figures
Base Network	0.7505 (0.0027)	0.1673 (0.0022)	5.2375 (0.0672)
Submodular (+ isotonic)	0.7417 (0.0027)	0.1936 (0.0023)	**2.3482 (0.0164)**
SoftCut (approximate)	**0.7788 (0.0026)**	**0.1612 (0.0022)**	2.7159 (0.0202)

Table 17: Results of Experiment 10.

Again, as in the other tasks, SoftCut produced the best results in terms of IoU.

Finally, SoftCut has been tested against the full resolution Cityscapes dataset, to mimic a real-world scenario. Only four cores have been given access to each of the various training processes, in order to show how much faster SoftCut is against the approach from [8].

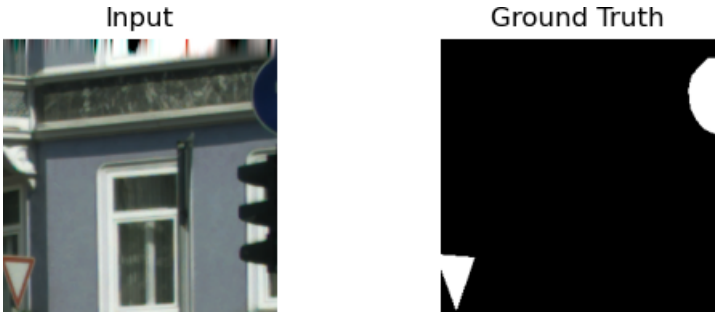
	Experiment 11
Architecture	*U-Net*
Encoder	*resnet18*
Dataset	*Cityscapes Full Resolution*

Table 18: Same setup as in Experiment 2, but used against the full resolution Cityscapes dataset.

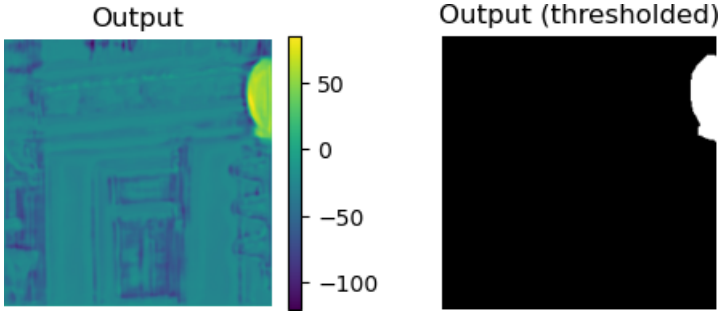
Model	**mIoU**	**loss**	**# distinct figures**	**train time**
Base Network	0.5910 (0.0064)	0.2660 (0.0059)	62.106 (1.6043)	**2h53min**
Submodular (+ isotonic)	0.5641 (0.0054)	0.2903 (0.0055)	54.020 (1.2437)	94h12min
SoftCut (approximate)	**0.6202 (0.0059)**	**0.2493 (0.0053)**	29.512 (0.7691)	35h30min

Table 19: Results of Experiment 11.

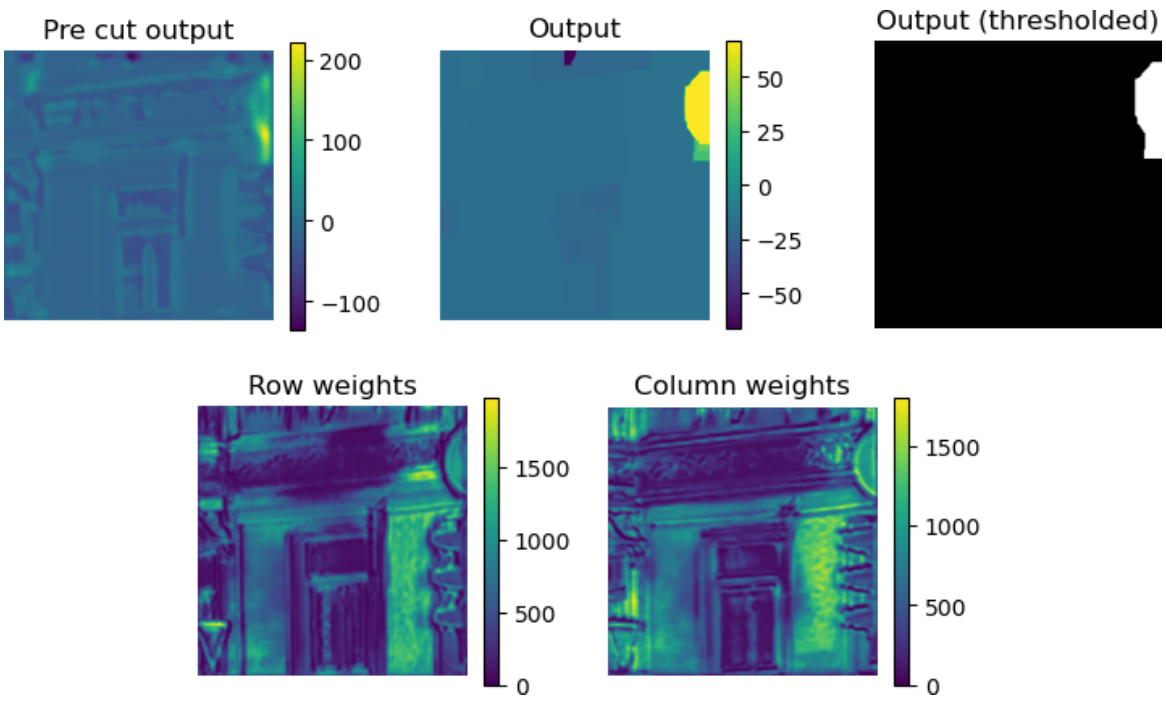
These final results again show how SoftCut outperforms both the base U-Net and the submodular technique in terms of IoU, while being much less computation-heavy than this last approach.



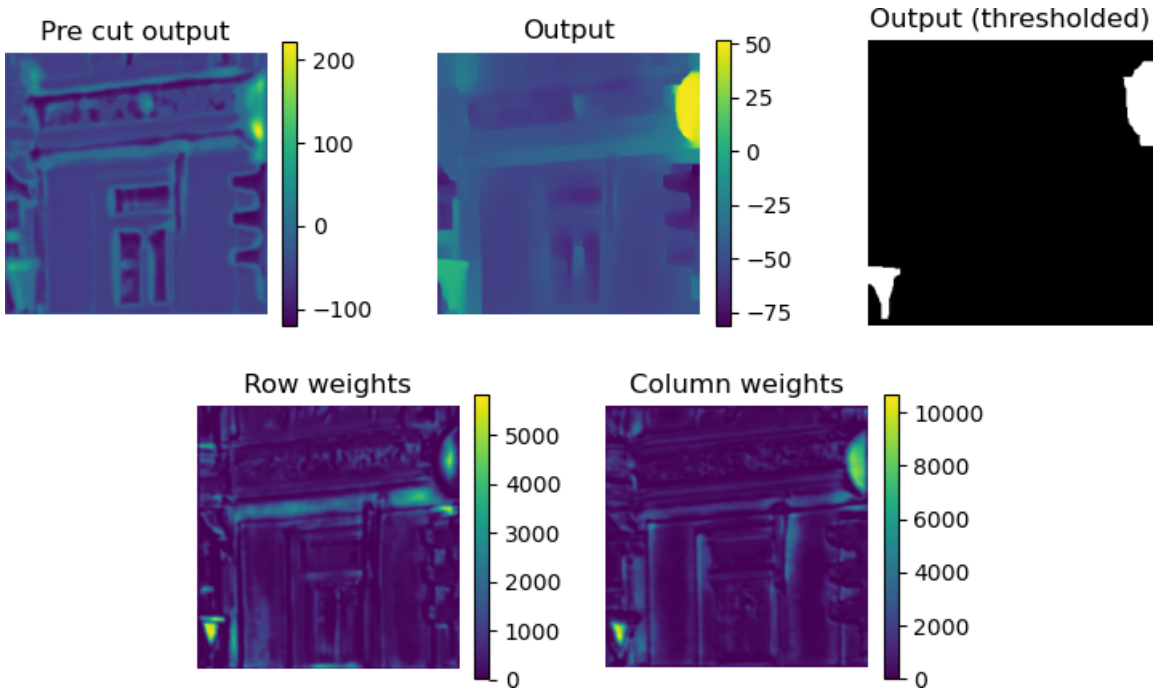
(a) The input image example (left) and the relative ground truth (right).



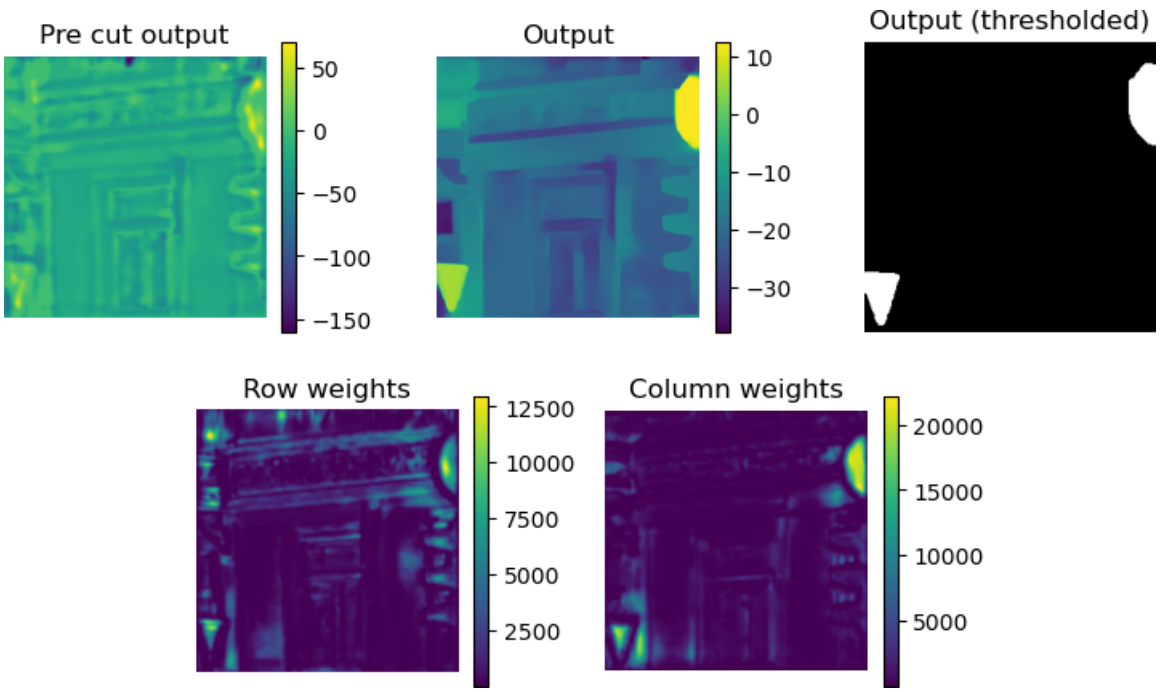
(b) The output of the base network without graph cut before (left) and after passing through a sigmoid function and being thresholded (right).



(c) Above the pixel scores of the network with the submodular approach (left), the total variation minimization layer output (center) and its output after passing through a sigmoid function and being thresholded (right). Below are the row and column weights computed by the second decoder and used for the total variation minimization.



(d) Above the pixel scores of the network with SoftCut computed exactly (left), the SoftCut output (center) and the SoftCut output after passing through a sigmoid function and being thresholded (right). Below the row and column weights computed by the second decoder and used for the SoftCut.



(e) Above the pixel scores of the network with SoftCut computed approximately (left), the SoftCut output (center) and the SoftCut output after passing through a sigmoid function and being thresholded (right). Below the row and column weights computed by the second decoder and used for the SoftCut.

Figure 11: Example of the results on a sample of the test set from Experiment 2. Note how the graph cut methods do not need the pixel scores to be an exact segmentation, but just a rough idea of the class of the pixels and then they use the edge weights to compute the shape of the objects.

4. Conclusion

From the results of the experiments of *Section 3* it is clear that SoftCut not only provides better results if used in combination with very simple CNNs, but can also improve the scores of much bigger and more complex architectures like U-Net, unlike other techniques like [8], while remaining faster. The possibility to effectively use a regular graph cut algorithm at inference time instead of SoftCut is also a valuable quality of this technique. Furthermore, the Focal Dice loss seems to be more effective than the plain Dice loss for training networks for image segmentation. Future works could focus on the advantages and disadvantages of SoftCut used in combination with other architectures, among which some that do make so extensive use of pooling layers, due to the increased receptive field of the pixels gained from a soft cut layer. Additionally, the reason why SoftCut with an exact solver performs worse than with the approximate one could be researched, and maybe see if this can not always be the case, maybe with the help of an ad hoc optimizer. Moreover, because of the still relatively slow speed of both the solvers, particularly the exact one, this can become the reason behind the search for further optimizations, like using an exact sparse linear system solver directly on the GPU. Taking into consideration that SoftCut is equivalent to an electric circuit only made by generators and (variable) resistors, a hardware accelerator could even be designed to solve it. Finally, new applications of SoftCut, like volumetric or even higher dimension segmentation, may be the topic of future studies.

References

[1] Brandon Amos and J Zico Kolter. Optnet: Differentiable optimization as a layer in neural networks. In *International Conference on Machine Learning*, pages 136–145. PMLR, 2017.

[2] Álvaro Barbero and Suvrit Sra. Fast newton-type methods for total variation regularization. In Lise Getoor and Tobias Scheffer, editors, *ICML*, pages 313–320. Omnipress, 2011.

[3] Alvaro Barbero and Suvrit Sra. Modular proximal optimization for multidimensional total-variation regularization. *Journal of Machine Learning Research*, 19(56):1–82, 2018.

[4] Quentin Berthet, Mathieu Blondel, Olivier Teboul, Marco Cuturi, Jean-Philippe Vert, and Francis Bach. Learning with differentiable pertubed optimizers. *Advances in neural information processing systems*, 33:9508–9519, 2020.

[5] Yuri Boykov and Vladimir Kolmogorov. An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *IEEE transactions on pattern analysis and machine intelligence*, 26(9):1124–1137, 2004.

[6] Lee R Dice. Measures of the amount of ecologic association between species. *Ecology*, 26(3):297–302, 1945.

[7] Josip Djolonga. torch-submod. `https://github.com/josipd/torch-submod`, 2017.

[8] Josip Djolonga and Andreas Krause. Differentiable learning of submodular models. *Advances in Neural Information Processing Systems*, 30, 2017.

[9] William Falcon and The PyTorch Lightning team. PyTorch Lightning. `https://github.com/Lightning-AI/lightning`, 3 2019.

[10] Floris Laporte. Creating a pytorch solver for sparse linear systems. `https://blog.flaport.net/solving-sparse-linear-systems-in-pytorch.html`, 2020.

[11] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[12] Magnus R Hestenes and Eduard Stiefel. Methods of conjugate gradients for solving. *Journal of research of the National Bureau of Standards*, 49(6):409, 1952.

[13] Jon M. Kleinberg and Éva Tardos. *Algorithm design*. Addison-Wesley, 2006.

[14] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.

[15] Floris Laporte. Torch sparse solve. `https://github.com/flaport/torch_sparse_solve`, 2020.

[16] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[17] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. Focal loss for dense object detection. In *Proceedings of the IEEE international conference on computer vision*, pages 2980–2988, 2017.

[18] Ekanathan Palamadai Natarajan. *KLU–A high performance sparse linear solver for circuit simulation problems*. PhD thesis, University of Florida, 2005.

[19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[20] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[21] pmneila. Pymaxflow. `https://github.com/pmneila/PyMaxflow`, 2011.

[22] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pages 234–241. Springer, 2015.

[23] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[24] Luca Trevisan. Lecture 15 from cs261: Optimization. `https://theory.stanford.edu/~trevisan/cs261/lecture15.pdf`, 2011.

[25] Pavel Yakubovskiy. Segmentation models pytorch. `https://github.com/qubvel/segmentation_models.pytorch`, 2020.

[26] Álvaro Barbero Jiménez. proxtv. `https://github.com/albarji/proxTV`, 2014.

A. Graph Cut Receptive Field

Theorem A.1. *The receptive field of a pixel p that is a solution of a graph cut problem* (1) *is not limited by a integer $k < d_{border}(p)$, where p belongs to the input image I (in the form of a 4-connected or 8-connected lattice graph) $d_{border}(p)$ is the maximum distance from the pixel p to a pixel p' on the border of I.*

Proof. Assume it exists $k < d_{border}(p)$ that limits the receptive field of pixel p. It is possible to define a set of weights b_i, f_i and $w_{i,j}$ as input of the graph cut solver such that p is in a connected subgraph A of I that include at least another pixel p' at $k+1$ distance from p. Moreover A is connected to the rest of I by edges with value $w_{i,j} = 0$. Due to the condition over the edges $w_{i,j}$ that surround A, the classifications of the other pixels that do not belong to it do not influence those of the pixels inside of it. Assume that all the weights of the edges between the pixels in A are equal to M and the values of b_i and f_i of those pixels are all, respectively, B and F. There are now two cases:

1. If $B > F$ all the pixels in the area will be classified as foreground.
2. If $B < F$ all the pixels in the area will be classified as background.

Now assume that we are in case 1 for the rest of the proof (for case 2 the proof is analogous). Let us vary $f_{p'}$ such that $nB < (n-1)F + f_{p'}$ where n is the number of pixels in the area and assume that $M > (n-1)F + f_{p'}$ so that every pixel in the area will be still classified in the same way. In this way, assigning now each pixel of the area to the background will minimize (1). This proves that the value of a pixel p', distant from p more than k, can influence the classification of p, and so be part of its receptive field.

For the case of SoftCut, Theorem A.1 is still valid and the proof just needs some slight variations due to the different formulation of the problem (b_i and f_i are replaced by the single pixel score value and the resulting belonging of a pixel i to one of the two sets is encoded as the sign of x_i), but note that because of the continuous nature of the formulation, any variation of the pixel score of pixel p' would influence the result of pixel p if there exist path between the two pixels such that all of its edge weights are > 0.

B. Focal Dice loss

The Focal Dice loss is a variation of the classic Dice loss (7), that follow the same idea as the one behind the Focal loss [17]: pixels predicted correctly but with a low certainty should penalize less the loss compared to the wrong classified ones. In the case of binary classification, the Focal loss can be written as

$$Focal(\hat{y}, y) = -y(1-\hat{y})^{\gamma}\log(\hat{y}) - (1-y)\hat{y}^{\gamma}\log(1-\hat{y}) \tag{26}$$

The factors $(1-\hat{y})^{\gamma}$ and $\hat{y}^{\gamma}$ added to the classic binary cross entropy decreases the loss in the cases where $y = 1$ and $\hat{y}$ is high and where $y = 0$ and $\hat{y}$ is low, so penalizing less the correct but uncertain predictions. For applying the same idea to the Dice loss (7) it is needed to "move" the values of $\hat{y}$ towards 0 if the pixel would happen to be classified in the negative class and towards 1 if the pixel would happen to be classified in the positive one. Assuming a threshold of $0 < t < 1$ (usually 0.5) for deciding if a pixel belongs to the negative or positive class, it is possible to define a differentiable s-shaped function

$$S_{t,\gamma}(x) = \frac{(x-t+0.5)^{\gamma}}{(x-t+0.5)^{\gamma} - (1-(x-t+0.5))^{\gamma}} \tag{27}$$

and use it to transform each element of $\hat{y}$ in the Dice loss, obtaining the Focal Dice loss

$$Focal\ Dice_{t,\gamma}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = 1 - \frac{2(S_{t,\gamma}(\hat{\boldsymbol{y}}) \bullet \boldsymbol{y})}{|S_{t,\gamma}(\hat{\boldsymbol{y}})|^2 + |\boldsymbol{y}|^2} \tag{28}$$

Abstract in lingua italiana

Gli algoritmi di graph cut sono sempre sembrati avere un'applicazione naturale nella segmentzione delle immagini, ma le difficoltà nell'estrarre le caratteristiche necessarie dell'immagine hanno fatto sì che architetture di reti neurali convoluzionali come U-Net presero il loro posto. Negli ultimi anni diverse soluzioni sono state proposte per implementare questo tipo di algorimi dentro uno strato di una rete neurale. SoftCut, l'approccio proposto in questo lavoro, è un rilassamento differenziabile del problema di graph cut, equivalente ad un intuitivo circuito elettrico, di cui viene mostrato come superi in termine di indice di Jaccard sia U-Net che la soluzione più promettente fra quelle di cui prima in uno scenario reale come Cityscapes, pur rimanendo più veloce di questi metodi per integrare un problema di ottimizzazione in una rete.

Parole chiave: taglio del grafo, differenziabile, segmentazione delle immagini, rete neurale artificiale, apprendimento profondo, apprendimento automatico

Acknowledgements

Ai miei genitori, e ai miei amici Clarence, Federico, Francesco, Nicola, Paolo e Silvio.