Automatic Detection Of Photovoltaic Panels Through Remote Sensing

Yann Claes,[1] **Gaspard Lambrechts**,[2] and **François Rozet**[3]
[1] *yann.claes@student.uliege.be (s161317)*
[2] *gaspard.lambrechts@student.uliege.be (s161826)*
[3] *francois.rozet@student.uliege.be (s161024)*

I. INTRODUCTION

Nowadays, photovoltaic (PV) panels play an important role in the transition towards renewable energy production, and more and more small/large-scale PV systems are installed worldwide. Their major strength is that anyone owning a rooftop/open field can install PV panels, hence can produce electricity in a renewable fashion.

Nevertheless, this growth also makes it very hard to estimate their geographical deployment, and it is thus hard to assess the production capacity of a given region, which influences exchanges in electrical grids.

By having access to some estimate of the production capacity of any region, one could easen the decisions made about the amounts of energy that should be produced from any possible source (nuclear, coal, etc.), to satisfy the total demand.

Therefore, this project, which we named ADOPPTRS for *Automatic Detection Of Photovoltaic Panels Through Remote Sensing*, aims at assessing the PV production capacity of a specific region using deep neural networks and aerial/satellite images processing. With these, our goal is to provide reliable estimates of all existing panels installations' locations and areas. More specifically, we want to apply our model(s) to the images of the WalOnMap website[1].

Section II will cover research related to our problem. Section III explains in details the different neural network architectures we reproduced/extended, along with the loss functions and optimizers we used. It will also present the data (pre-)processing and augmentation performed. Section IV introduces our experimental procedure and the experiments we conducted along with their results. Finally, section V concludes with a critical analysis of the results and some further developments or improvements we did not explore.

II. RELATED WORK

The DeepSolar [1] project represents our main source of inspiration. Indeed, this project shares the same goals as ours : predicting the locations and areas of solar panels, using deep learning frameworks. With their project, the authors of DeepSolar constructed a comprehensive and publicly available database of solar installations in the United States. Out of the scope of our project, they were able to highlight interesting correlations between photovoltaic development and environmental as well as socioeconomic factors, such as income and inequalities.

Their model uses 2 branches. The first one is a CNN classifier (Google Inception V3 [2]) used to detect the presence or absence of PV panels in the processed image. The second one is also a CNN that is connected to the intermediate layers of the first branch to perform segmentation and estimate the size and shape of the detected panels.

Other PV panels detection and segmentation tasks were already developed in the past. For instance, in [3], they used a very simple CNN architecture to solve their problem. It consists of a combination of convolutional layers, max-pooling layers and up-sampling layers. They also used a signed distance function of boundaries, which helps the network to better identify solar panels.

In [4] and [5], they detect PV panels in very high resolution aerial imagery, using respectively Random Forests and CNNs. Furthermore, and even more interesting for our project, they made their data set publicly available [6], providing more than 19 000 human-annotated locations over 601 images of four cities in California, covering a total spatial area of 1350 km^2.

In a more general and broader point of view, U-Net [7] and SegNet [8] can be considered as pillars in image segmentation tasks. Both rely on convolutional layers along with max-pooling and up-sampling layers, even though there are still important differences between both architectures. Since they are both networks we reproduced (to some extent), we will detail them in section III.

III. METHODS

A. Architectures

All model implementations can be found in the source code `python/models.py`.

All implemented neural networks are (fully) convolutional neural networks. They are used as *segmentation* networks, meaning that their goal is to transform the image tensor (3 channels) into other tensors successively, until creating a final mask tensor of C channels that tells for each pixel the probability it belongs to each class in C. In our case, we only have 2 classes : being a PV pixel, or being another pixel. Thus, we decided to only have 1 channel instead of 2 in the final tensor : the probability of being a PV pixel p ; the other class having probability

[1] This project was conducted in parallel with our *Big Data* project.

$1 - p$. These convolutional neural networks will use the following types of layers, *i.e.* transformations from one tensor to another :

Convolutional: A convolutional layer transforms a C-channels tensor into a D-channels one. Each new channel is associated with a different *kernel* of shape $C \times w \times h$. Each element (i, j) in a (new) channel is computed as the the sum of the elements of the element-wise product of its kernel and a $C \times w \times h$ window "surrounding" $(\cdot, i, j)$ in the previous tensor. This process is called a *convolution*.

In addition, it is possible to add a *padding*, *i.e.* additional bordering elements, to the original image to avoid (or decrease) the reduction of size in the output tensor. Indeed, without padding, if w and/or h are greater than 1, it is not possible to apply the convolution to the border elements, and therefore the output tensor shrinks.

A *stride* can also be used in order to reduce explicitly the dimension of the tensor (down-sampling) by translating the kernel several elements at a time (horizontally and vertically) instead of one by one.

In our implementations, we will use $C \times 3 \times 3$ and $C \times 1 \times 1$ kernels with a respective padding of 1 and 0, but no stride, such that only the number of channels varies.

Pooling: A pooling layer reduces the tensor dimensions but not the number of channels. This is done by considering a kernel of shape $w \times h$ that has a stride of (w, h), so that no overlapping exists between the considered windows. Then, on each of these windows is applied a non linear function, called a *filter*, instead of a weighted sum like in the convolutional layer.

In our case, we will only consider the simple *max pooling layer*. It means that for each window we simply keep the maximum value. This is done independently in each channel, unlike in convolutional layers.

Up-sampling: An up-sampling layer increases the tensor dimensions (but not the channels). In our implementations, we use two types of up-sampling layers :

1. *Classic up-sampling layers*, where the dimensions of the tensor are increased by a factor (w, h). To fill in all new elements, an interpolation technique must be used. In our implementations, we use the *bilinear* up-sampling. Starting from the four surrounding known elements, it first performs a linear interpolation in one direction, obtaining rows (or columns) of filled elements, and then another one in the other direction; the resulting interpolation is thus nonlinear, in fact quadratic.
2. *Pooling indices up-sampling layers* are used in parallel with a *max pooling* layer. They revert the max pooling operation by creating a tensor of the original shape filled with zeros except at the *positions* corresponding to elements that were selected as maximum by the *max pooling*. At these positions, the values of the input tensor are used. *N.B.* This input tensor does not necessarily contain the values of the max-pooling.

ReLU: A layer that puts all negative elements to zero, which introduces (beneficial) nonlinearities.

Sigmoid: A layer that applies the sigmoid function

$$\sigma(x) = \frac{1}{1 + \exp(-x)} \tag{1}$$

to every element.

Softmax: A layer that applies the channel-wise sigmoid function to every element, *i.e.*

$$t_{kij} \leftarrow \frac{\exp(t_{kij})}{\sum_{l=1}^{C} \exp(t_{lij})} \tag{2}$$

with $\boldsymbol{t}$ a tensor in $\mathbb{R}^{C \times W \times H}$ and k, i, j indices respectively $\leq C$, $\leq W$ and $\leq H$.

The interesting property of this transformation is that, afterwards,

$$\sum_{k=1}^{C} t_{kij} = 1 \tag{3}$$

similarly to probabilities.

Batch normalization: A batch normalization layer improves performance and stability of the network, often allowing to achieve similar or better performance with much less training epochs. Most of the time, and according to the original paper [9], it consists in an additional layer between a convolutional layer and an activation layer (like ReLU). Its role is to shift and rescale the current (mini-)batch according to the moments (mean and variance) estimated during the *training* phase. If its benefits have been demonstrated many times, there is yet no consensus on the reasons of its effectiveness.

N.B. In an early stage of our implementations, we used both Batch Normalization (BN) and Dropout [10], but it seemed to cause some convergence problems. We decided to remove the latter, as it slows down training. Furthermore, some papers, including [9], suggest that Batch Normalization could, in some cases, eliminate the need for Dropout or even interfere with it [11]. It should be noted that we haven't performed a rigorous analysis of this behaviour, as it was not the subject of our project.

With these layers, four architectures have been implemented.

a. U-Net The first model we implement (based on [12]) is U-Net [7], which is known for biomedical image segmentation. Its architecture consists in a downhill path followed by an uphill path, depicted on FIG. 1.

The downhill path of the original U-Net is made of 5 successive *double convolutions* interleaved with 2×2 max-pooling. Each convolution is composed, in that order, of

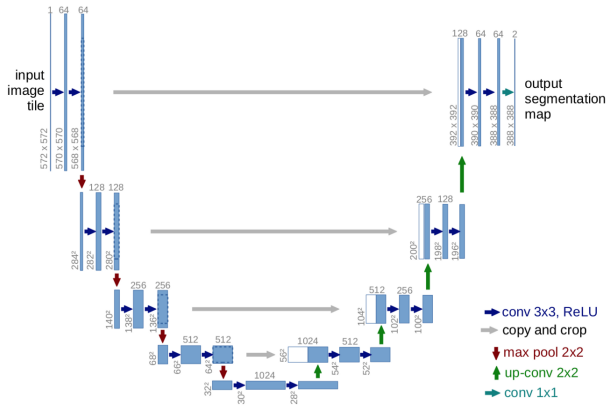

FIG. 1. Original U-Net architecture. [7]

- A 2-D padded[2] convolution with a kernel of size $C \times 3 \times 3$;
- A ReLU activation function.

We keep the same structure for the double convolution operation but have added a batch normalization layer between the convolution and the ReLU in order to improve convergence and stability; this will be discussed in section IV.

Similarly to the original architecture (FIG. 1), we double the number of channels at each double convolution, except the first time where we move from 3 channels to 64. At the same time, the dimensions of the image are divided by two due to the max-pooling after each double convolution. Note that the fifth double convolution operation is not followed by a max-pooling afterwards.

At the end, the feature map is thus composed of $64 \times 2^4 = 1024$ channels with dimensions that depend on the input image dimensions, but that are $2^4 = 16$ times smaller.

The uphill path consists of 4 repetitions of the following layers :

- An up-sampling of the feature maps, with a scale factor of 2. This means that the resulting output feature map will be twice as big (along each dimension) as the down-sampled feature map. As up-sampling method, we use the bilinear up-sampling;
- A concatenation with the corresponding feature map along the channel axis;
- A *double convolution* (with batch normalization) such that the number of channels is divided by three.

Hence, at each up-sampling step, the number of channels is divided by two because the concatenation increase the number of channels by 1.5, while the dimensions are doubled. In our case, since we use a padding of 1, consistently with the kernel of shape $C \times 3 \times 3$, we do not need to crop the feature maps before concatenating them with the up-sampled feature map.

The concatenation with the corresponding feature map is in fact a *passthrough* connection that allows the successive up-sampled feature maps to benefit from the more *local* features present in the feature maps of the downhill path (see gray arrows in FIG. 1).

Finally, there is the last layer. The original network uses a 1×1 convolution to map each feature vector to the appropriate number of classes. In our network, we use the same layer followed by a sigmoid activation to obtain a prediction of the likelihood that a pixel is within a PV panel.

b. SegNet The second model we reproduce is SegNet [8]. As described in the paper, SegNet is composed of an encoder network, a decoder network and a pixelwise classification layer.

In the original architecture, as can be seen on FIG. 2, the encoder consists of 13 convolutional layers, arranged precisely in the following way:

$2 \times$ a *double convolution*, the same as in U-Net (with batch normalization), followed by a max-pooling layer retaining the pooling indices for later up-sampling.

$3 \times$ a *triple convolution*, which is identical to double convolution except that it has 3 convolutions instead of 2.

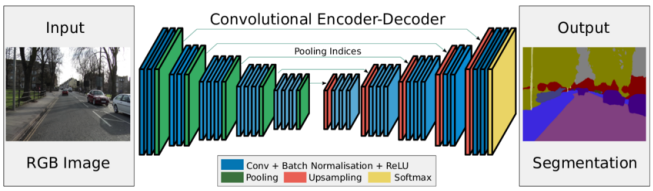

FIG. 2. Original SegNet architecture. [8]

The decoder is the symmetrical counterpart of the encoder, using pooling indices up-sampling layers instead of pooling layers. It differs from U-Net as it doesn't rely on *concatenation* but rather on pooling indices to carry information.

However, for this project, we do not to use the full SegNet architecture. Rather, we reproduce a smaller SegNet [13], dropping all triple convolutions and using 3 layers of double convolution, each followed by a max-pooling layer, for the downhill path. This architecture is depicted on FIG. 3.

As far as the number of convolutional filters in the convolutional layers are concerned, it is similar to [13], *i.e.*

[2] It should be noted that, in the original implementation, there was no padding.

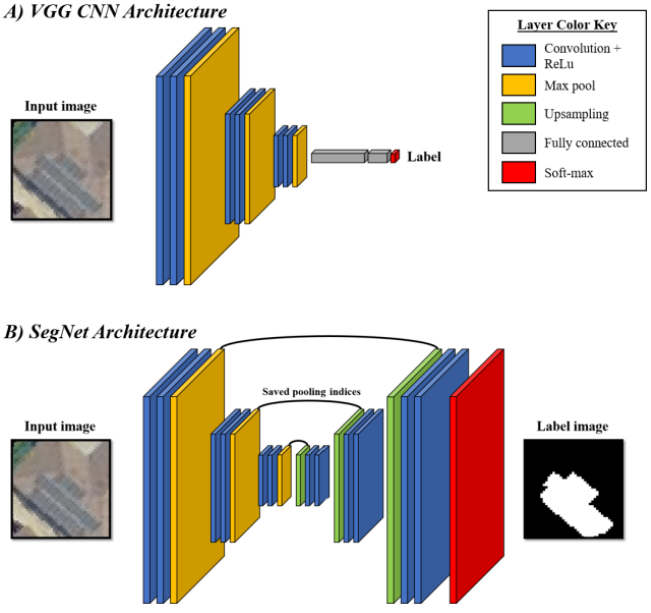

FIG. 3. Truncated version of SegNet. [13]

- The first *double convolution* moves from 3 channels to 64 channels;
- The second one increases the number of channels to 128;
- The third one keeps the number of channels equal to 128.

As explained above for the general SegNet architecture, the decoder is simply the symmetrical counterpart of these 3 layers, and makes use of pooling indices for its up-sampling layers. Hence, its first *double convolution* layer keeps the number of channels at 128, its second one moves to 64 channels while the last one moves to a single channel.

In addition, as final layer, we use a 1×1 convolutional layer followed by a sigmoid activation, highlighting the presence/absence of panel.

One can notice the similarities between U-Net and Segnet. Indeed, except for the number of hidden layers, they present the same idea of a down-sampling downhill path followed by an up-sampling uphill path that uses *passthrough* connections to gather prior information.

The difference lies in the way this information is carried. Indeed, in U-Net it is done by concatenating the feature maps, while, in SegNet, the only information comes from the max-pooling indices. The latter obviously carries less information, but is also much lighter memory-wise and complexity-wise.

c. Multi-task U-Net For now, our U-Net implementation differs from the original by the following aspects

- Batch normalization in the double convolutions;
- Padding in the convolutions, removing the need for cropping the feature map in the concatenation process;
- Adapted to binary classification with a sigmoid layer and a single channel output;

Even if this U-Net implementation had already given us satisfying quantitative results, a qualitative analysis of the output masks has revealed that U-Net had some trouble to delimit the boundaries of the PV arrays. To overcome this issue, we got inspired by an adaptation of SegNet that specifically addresses this problem in the case of building segmentation [14].

In this paper, they review the classical approaches to overcome this problem, including post-processing of the mask or training MLP on top of the fully convolutional network. To avoid the difficulty of the former and the computational cost of the latter, they introduced a *multi-head* fully convolutional network along with an adapted *multi-task* loss and showed that this approach allowed reaching better quantitative results.

The whole idea behind this new head is to add a geometric term that incorporates the boundary information in the loss, in addition to the semantic term that is about classifying each individual pixel. This is done by trying to predict the *truncated signed distance transform* of the mask, instead of the mask itself. The transformed architecture can be visualized on FIG. 4, in the case of a Segnet encoder-decoder instead of U-Net.

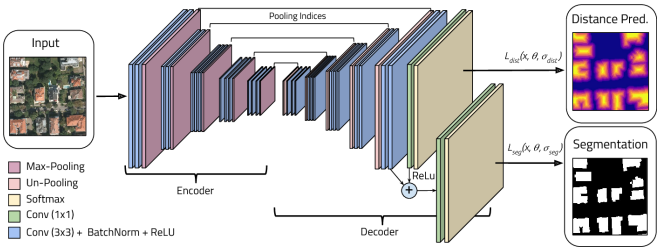

FIG. 4. Multi-head Segnet architecture. [14]

The structure of the Multi-task U-Net is exactly the same as U-Net without the last 1×1 convolutional and sigmoid layers. Instead, the distances are computed from the output of the decoder by using a 1×1 convolutional layer followed by a softmax layer. The 1×1 convolutional layer has $2R + 1$ output channels corresponding to the classes of discretized *distances* (see *N.B.*).

In parallel, the last layer of the decoder and the output of the aforementioned 1×1 convolutional layer – passed through a ReLU layer – are concatenated along the channel axis, whereupon they are passed through another 1×1 convolutional layer and sigmoid in order to produce the prediction *mask*.

N.B. The *distance transform* of a mask is the map of distances of each pixel to its closest *contour*. A contour is the frontier between the 0 and 1 pixels. The *signed distance transform* is the distance transform, negatively signed when the pixel is a background pixel (0's). Finally, the *truncated signed distance transform* is bounded between $-R$ and R and discretized into $2R + 1$ different

classes. What is pleasantly surprising is that the *distance transform* can be computed in *linear* time.

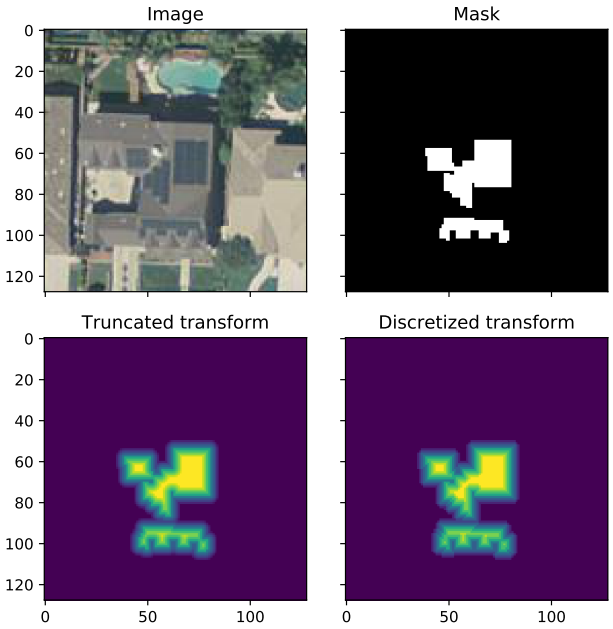

FIG. 5. Truncated distance transform representation.

Formally, if Q denotes the set of pixels of the contours, the truncated signed distance transform of a pixel p is defined as

$$D(p) = \delta_p \min\left(\min_{\forall q \in Q} d(p,q), R\right) \tag{4}$$

$$\delta_p = \begin{cases} +1 & \text{if } p \in \text{foreground} \\ -1 & \text{else} \end{cases} \tag{5}$$

where d is a distance metric. In our work, only the Euclidean and Manhattan distances have been considered, but the Manhattan distance had convergence issues which lead us to use the Euclidean distance, even though it is a little heavier to compute.

Concerning the *multi-task* loss, it is simply the regular loss of the "mask-head" output with the target plus a penalizing term for the miss-classification of the pixels distance classes by the "distance-head". This composed loss will enforce the network to learn geometric information about the contours of the mask. Losses will be further discussed in the next section.

d. Multi-task SegNet Multi-task SegNet has exactly the same architecture as the "small" SegNet, with the same modifications of the head as in Multi-task U-Net. In fact, it resulted in what was implemented in the paper that inspired our multi-task network [14], with the exception that we use the smaller version of SegNet.

B. Loss functions

As explained in the previous section, different loss functions were used for the different architectures (cf. `python/criterions.py`). However, they all share the same loss function for the mask-head : the well known *dice loss*. Let A be the predicted probability mask and B the ground truth, where each element is either 1 or 0 if it respectively belongs or not to a PV panel. The *dice loss* is computed as

$$DiceLoss(B, A) = 1 - Dice(B, A) \tag{6}$$

where $Dice(B, A)$ is the *dice similarity coefficient* of B and A. It can be expressed as

$$Dice(A, B) = \frac{2\,|A \cap B|}{|A| + |B|} = Dice(B, A) \tag{7}$$

where $|\cdot|$ means the sum of the elements and $\cdot \cap \cdot$ means the element-wise product, in this context. We can notice that the numerator is always smaller or equal to the denominator which implies that the dice similarity coefficient is within the interval $[0; 1]$; so does the dice loss.

For the multi-headed networks, in addition to the *dice loss*, we also need to penalize the miss-classification of the distance-head, which is done with the *cross entropy* loss :

$$CrossEntropyLoss(y, p) = -\frac{1}{N}\sum_{i=1}^{N} \log p_i[y_i] \tag{8}$$

where N is the number of pixels, p_i the predicted class distribution of the pixel i and y_i its ground truth class.

Eventually, the multi-task loss is computed as

$$\begin{aligned} l(y, x; \theta) = {} & DiceLoss(y, \text{mask}(x; \theta)) \\ & + CrossEntropyLoss(D(y), \text{dist}(x; \theta)) \end{aligned} \tag{9}$$

where "mask" and "dist" distinguishes the output of, respectively, the mask-head and the distance-head and $D(y)$ denotes the discretized truncated signed distance transform of the target.

In [14], the terms of the loss corresponding to each different task were weighted by a term proportional to the the uncertainty on this specific task, as has been initially developed in [15]. But, having only two classes and two losses of similar magnitude, we don't weight their sum.

C. Optimizers

For all our networks, we use the *Adam* optimizer [16], which is one of the default optimizers nowadays. We will start by explaining how *batch*, *stochastic* and *mini-batch* gradient descents work, before explaining Adam.

Batch gradient descent purpose is to optimize the parameters (vector) θ of a predictive model $f(x; \theta)$ such that it minimizes the empirical error $\mathcal{L}(\theta)$ with respect

to the N pairs (x_i, y_i) of a dataset, called the *batch*. This empirical error is computed as the mean of the individual losses of each pair in the batch, *i.e.*

$$\mathcal{L}(\theta) = \frac{1}{N}\sum_{i=1}^{N} l(y_i, f(x_i;\theta)) \tag{10}$$

where l is a loss function.

To achieve this goal, at each training time step t, the gradient descent technique computes the *gradient*

$$g_t = \nabla_\theta \mathcal{L}(\theta_t) = \frac{1}{N}\sum_{i=1}^{N} \nabla_\theta \ell\left(y_i, f\left(x_i;\theta_t\right)\right) \tag{11}$$

of the empirical error and updates the parameters θ according to

$$\theta_{t+1} = \theta_t - \gamma g_t \tag{12}$$

where γ is called the *learning rate*.

Although doing so will eventually reach a (local) minimum of the empirical error, convergence can be really slow, as the closer we get to the minimum, the smaller the gradient becomes, hence leading to smaller step sizes. Furthermore, computing the sum on the whole batch can be computationally expensive.

The idea of stochastic gradient descent is that, since the empirical error is an unbiased estimation of the error, any partial sum of the individual losses is unbiased as well. Especially, computing g_t for one pair of the batch and updating θ_t accordingly, should eventually achieve the same objective but much more efficiently.

However, this procedure has a much greater variance as well, which could increase dramatically the number of steps to perform. In order to reduce its *stochasticity*, one could instead compute g_t over a subsample – called *mini-batch* – of the batch. In fact, the greater the size of the mini-batches, the lower the variance of the gradients.

But, there is a major problem with vanilla gradient descent(s) : the learning rate is common to all parameters, which comes to assume the *isotropy* of the curvature.

Conversely, Adam optimizer adapts the learning rate differently for each parameter. To do so, it uses *exponential moving averages* of the first (m_t) and second (v_t) moments of the gradient. More specifically, the parameters are updated following[3]

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\, g_t \tag{13a}$$

$$\hat{m}_t = \frac{m_t}{1-(\beta_1)^t} \tag{13b}$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)\,(g_t)^2 \tag{13c}$$

$$\hat{v}_t = \frac{v_t}{1-(\beta_2)^t} \tag{13d}$$

$$\theta_{t+1} = \theta_t - \gamma \frac{\hat{m}_t}{\epsilon + \sqrt{\hat{v}_t}} \tag{13e}$$

[3] Every operation involving vectors is performed element-wise.

where β_1 and β_2 are the *forgetting* factors of the moving averages. The smaller their value, the quicker the contributions of older m_t and v_t vanish. This allows v_t to remain bounded, which prevents the step size to reach zero.

However, m_t and v_t are initially null, which introduces an *initialization bias* towards small θ updates. To correct this, Adam uses 13b and 13d to increase the norm of m_t and v_t in the early stages.

As the goal of this project is not to study the influence of hyper-parameters on the training process, we always apply the same standard ones :

$$\gamma = 10^{-3}, \quad \beta_1 = 0.9, \quad \beta_2 = 0.999$$

D. Data processing and augmentation

As mentioned in section II, our data set is composed of 19 000 PV annotations over 601 aerial images. These annotations are given as the form of polygon *vertices*, which can be easily transformed into *segmentation masks*. However, the original images are too big (5000×5000) to feed them directly to our neural networks. Therefore, we cut them up into smaller 256×256 images. Unfortunately, this causes our data set to become very unbalanced as most of the covered area doesn't present any PV panel. In order to balance it during training, we prioritize the regions that are "surrounding" the annotations. But, since this procedure introduces a bias in what is learned by the network, we also alternate with images chosen completely at random.

Furthermore, as explained in the introduction, we want to apply our models to the WalOnMap images. But, we noticed that the images of our training set (California) are quite different from the ones of WalOnMap, in terms of colorimetry, exposure, sharpness, etc. Therefore, our models have to be robust with respect to this kind of features.

Hence, we *augment* our data set by applying random transformations to the images while training. Some transformations we apply are quarter rotations (90°, 180° or 270°), flips (horizontal or vertical), brightness alteration, contrast alteration, smoothing, sharpening, etc.

A small sample of cropped images along with their masks and random augmentations of them is presented in FIG. 6.

Furthermore, we have annotated, using the "VGG Image Annotator" [17], 661 WalOnMap images (512×512) to *fine tune* our networks after having trained them on the Californian dataset.

E. Metrics

In [1], [4] and [5], the metrics used to characterize the *detection* performances are the *precision* and *recall*. Precision measures the ratio of correct predictions among

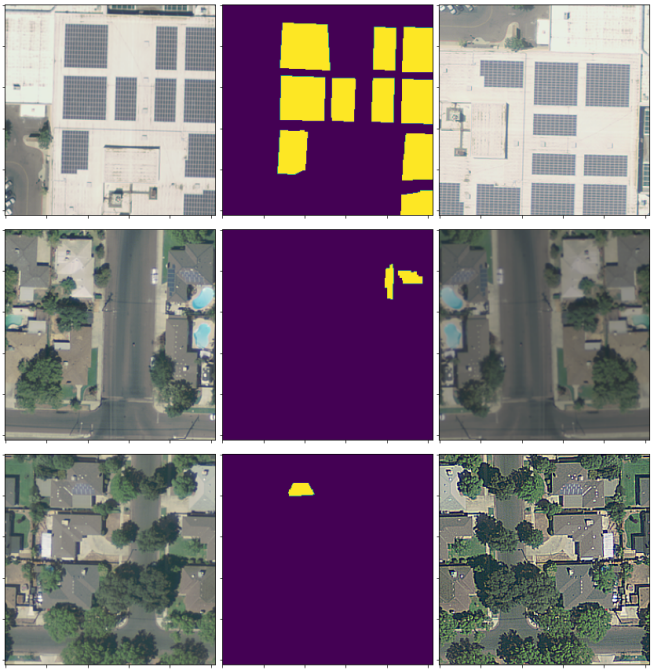

FIG. 6. Californian dataset sample with their masks (center) and some random augmentations (right).

positive predictions. Recall measures the ratio of correct predictions among positive instances. However, conversely to these papers, we don't have *detection* networks; strictly speaking. Therefore, in order to compare our *detection* capabilities, we perform a "matching" between the contours of the target mask and those of the output mask. Basically, a target and an output contours match if their intersection is not empty, even barely. It should be noted that very small contours in the output mask are not considered as positive predictions.

For the size estimation, *i.e* segmentation, we use the precision and recall metrics as well, but at the pixel level and only for predictions that we know are correct (true positive detection).

Furthermore, the authors of DeepSolar [1], introduced the *mean relative error* (MRE) as

$$\mathrm{MRE} = 1 - \frac{\sum_{\text{true positive}} \text{estimated area}}{\sum_{\text{true positive}} \text{true area}} \tag{14}$$

which we also compute for the sake of comparison.

Finally, our measures of precision and recall are dependant on a *threshold* parameter. Therefore, it is sometimes preferred to use the *average precision* instead, which can be visualized as the area under the *precision-recall* curve. The average precision is the metric we use to compare our models.

Note. It should be noted that the threshold is shared for detection and segmentation.

IV. RESULTS

In this section, we present the results of training the implemented networks for 20 epochs on the Californian dataset. In order to assess the uncertainty of our metrics, we perform a *5-fold cross validation*. The principle of k-fold cross validation is to divide a training set into k folds and to train k independent models on all folds but one. The latter is the validation fold, while the others are the training folds.

Note. A few prediction examples on the Californian dataset are provided in the Appendix A 2 as well as additional graphs in the Appendix A 1.

a. Convergence To assess the convergence of our models, we evaluate their loss during the training phase on the training folds. The considered losses depend on the network architecture, as defined in section III B.

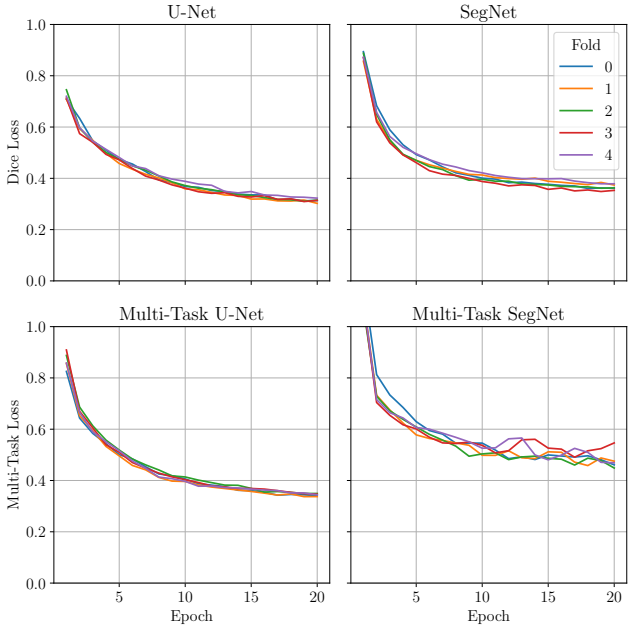

FIG. 7. Mean training losses during 20 epochs. Each color corresponds to a different training-validation folds set.

As can be seen from FIG. 7, all four models seem to converge to some fixed value. Convergence looks similar in terms of speed, and the variance w.r.t. the folds looks rather small, which suggests that the folds are representative of the whole dataset. However, both SegNet networks have higher losses and variances than their U-Net counterpart, which means they have more trouble to learn. This would not be surprising as they are also significantly smaller.

Moreover, we can observe that the Multi-Task loss is only slightly over the corresponding Dice Loss, suggesting that the distances are well learnt or that they help learning the mask. That being said, the cross-entropy loss does not take into account the imbalance between the different classes, probably leading to a very low loss

thanks to the background class (most negative distance).

b. Detection As mentioned in section III E, we use precision and recall on the validation fold to assess the *detection* performance of all four models.

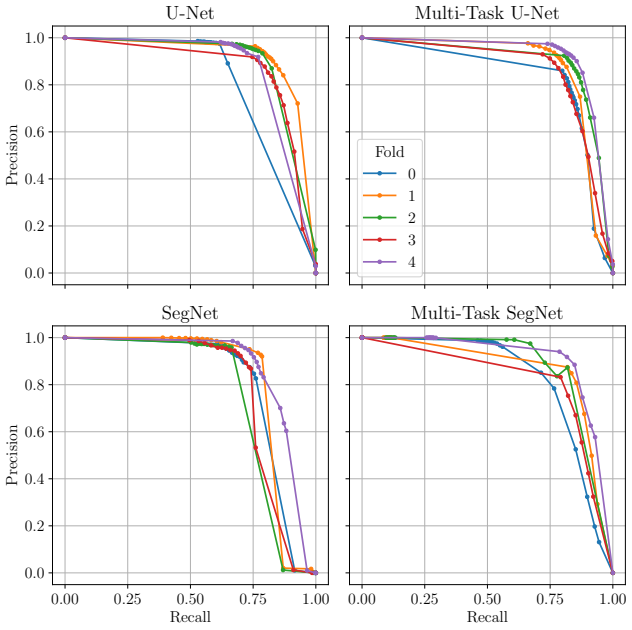

FIG. 8. Detection precision and recall of the networks after 20 epochs of training. Each color corresponds to a different training-validation folds set. The thresholds are $0, 10^{-9}, 10^{-8}, \ldots, 10^{-1}, 0.5, 1-10^{-1}, 1-10^{-2}, \ldots, 1-10^{-9}$ and 1.

All models do not perform equivalently in terms of detection, as can be seen from FIG. 8. For both U-Net models and SegNet, most precision-recall pairs are located in the up-right corner, meaning that the background and foreground are well segregated; either near 0 or near 1. Furthermore, the curves seem consistent with respect to the fold, especially for Multi-Task U-Net.

Conversely, for Multi-Task SegNet the pairs are mostly located in the up-left corner, reflecting a significant uncertainty in the predictions, which is emphasised by the notable changes of distribution from one fold to the other for the low thresholds. As a consequence, even though the precision recall curves don't necessarily look terrible, it is not actually possible to choose an adequate threshold for this model.

c. Segmentation Based on the true positive detection predictions, we compute the precision and recall of the panel segmentation.

The results of FIG. 9 show a significant performance difference between U-Net models and SegNet ones. More specifically, the recall of SegNet and Multi-Task Segnet drops rapidly as the threshold increases, which can be clearly observed with FIG. 17. Conversely, U-Net and Multi-Task U-Net are particularly steady w.r.t. to the threshold. These behaviours are emphasized by FIG. 10.

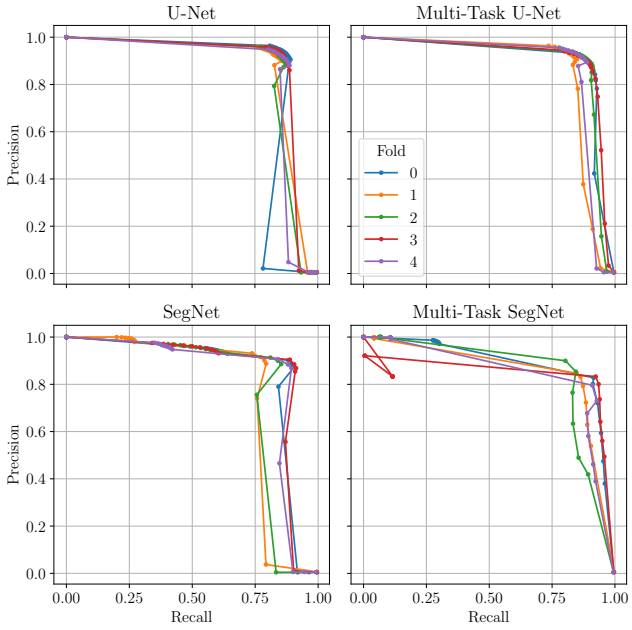

FIG. 9. Segmentation precision and recall of the networks after 20 epochs of training. Each color corresponds to a different training-validation folds set. The thresholds are the same as in FIG. 8.

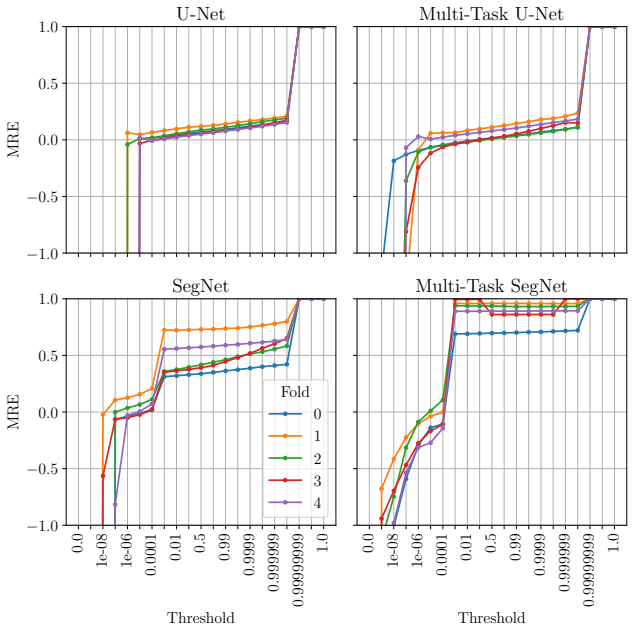

FIG. 10. Segmentation MRE of the networks after 20 epochs of training, w.r.t. the threshold. Each color corresponds to a different training-validation folds set.

Indeed, we observe that U-Net and Multi-Task U-Net are nearly unbiased (MRE $\simeq 0$) for the majority of the thresholds while SegNet and Multi-Task SegNet either underestimate or overestimate largely the area.

d. Comparison We estimate the average precision of a model as the mean of its average precision on the

different folds.

	Detection AP	Segmentation AP
U-Net	0.868 ± 0.038	0.848 ± 0.019
Multi-Task U-Net	0.875 ± 0.030	0.881 ± 0.020
SegNet	0.806 ± 0.039	0.805 ± 0.039
Multi-Task SegNet	0.858 ± 0.032	0.848 ± 0.016

TABLE I. Average precision of the networks after 20 epochs of training.

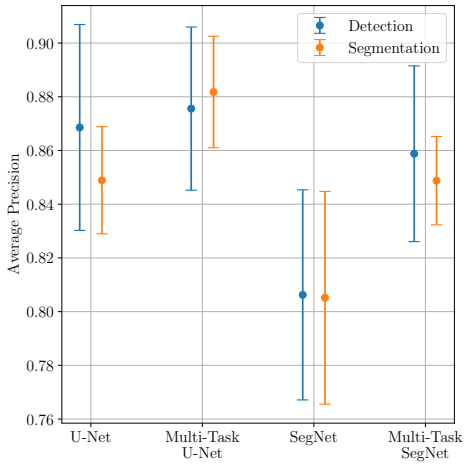

FIG. 11. Average precision of the networks after 20 epochs of training.

Our best model is Multi-Task U-Net both in detection and segmentation, reaching above 0.87 average precision in both tasks. For the rest of this report, we only use this model. Also, we choose a threshold of 0.5, as it is the most consistent threshold among all folds and as it balances quite well precision and recall.

	Metric	Value
Detection	Precision	0.841 ± 0.078
	Recall	0.826 ± 0.026
Segmentation	Precision	0.912 ± 0.013
	Recall	0.869 ± 0.027
	MRE	0.046 ± 0.041

TABLE II. Multi-Task U-Net average metrics at 0.5 threshold.

e. Fine tuning As explained in the previous section, the model we wish to apply to WalOnMap images is Multi-Task U-Net. However, even though it was trained for 20 epochs on the Californian dataset with substantial data augmentation[4], the model didn't generalize well to WalOnMap. Therefore, we fine tune our model for 10 more epochs on the 661 WalOnMap images we annotated.

To better assess the performance boost provided by the fine tuning, we divide the 661 images into 5 folds and perform 5-fold cross validation.

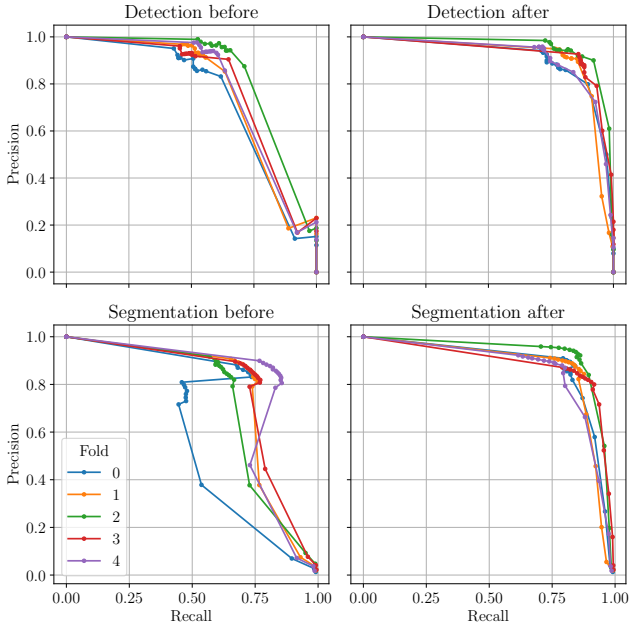

FIG. 12. Detection and segmentation precision and recall before and after 10 epochs of fine tuning. Each color corresponds to a different training-validation folds set.

As one can see in FIG. 12 and TABLE III, the fine tuning improves significantly and consistently the performances of the network both in detection and segmentation.

	Detection AP	Segmentation AP
Before fine-tuning	0.783 ± 0.033	0.703 ± 0.068
After fine-tuning	0.922 ± 0.018	0.861 ± 0.019

TABLE III. Average precision (AP) before and after 10 epochs of fine tuning.

Finally, the model was fine tuned with all 661 images and used to detect the PV installations in the Province of Liège. The result is showcased as an interactive map at `https://francois-rozet.github.io/adopptrs/`.

V. DISCUSSION

Overall, we are satisfied by our results. With a detection precision of 0.841 and a recall of 0.826, if we are still far from DeepSolar [1] performances (0.934 precision and 0.895 recall), we actually outperform significantly Malof et al. [5] detector (0.72 precision and 0.8 recall).

However, it should be noted that DeepSolar's dataset is quite different from ours, therefore the comparison isn't

[4] It should be noted that, to match WalOnMap and California scales, we had to up-scale the latter's images by a factor 2.

very relevant. Furthermore, both DeepSolar and Malof et al. implemented models with the primary purpose of detecting the panels while, in our case, detection is kind of a *by-product* of the segmentation. Therefore, our results are all the more respectable.

Concerning the segmentation, on average, our final model underestimates slightly (4.6 % MRE) the size of the PV arrays. However, as is mentioned in the Appendix A 2, some annotations of the Californian dataset are inaccurate which could lead to a biased metric.

From a broader perspective, it should be emphasized that, separately, the detection and segmentation metrics do not necessarily reflect the actual behaviour of the models as, in a real use-case, these errors would normally add up.

Another point to highlight is that, in this project, we focus mainly on the architecture of the networks, while barely touching to the training hyper-parameters. Also, we only trained our networks for 20 epochs[5], which might be not enough to learn the task. If we were to continue this project, further investigation of the training procedure would certainly be a priority.

Also, if we decided early on to use Batch Normalization and Data Augmentation, we actually never measured quantitatively the influence they had on our models. This is also a point we should investigate in hypothetical future works.

In addition, when we used our final model to detect all PV installations in the Province of Liège, if most panels were accurately detected and segmented, we noticed qualitatively that a lot of shadows and canopies were misleadingly predicted as PV panels. We don't believe it to be due to the network itself, but rather to the low number and variety of images we hand-annotated. The most obvious way to improve our results would be to annotate more images, which we unfortunately had not the time to do.

Finally, initially and for most of the project, we were only considering the images surrounding the annotations (cf. section III D) for both training and validation. Obviously, we obtained very good results but they were extremely biased and did not reflect in other tests we made, which gave us terrible headaches. If there is something we have learned with this project, it is that we should check very carefully and *from the start* that our training and, most importantly, validation procedures are free of bias.

[1] J. Yu, Z. Wang, A. Majumdar, and R. Rajagopal, "Deepsolar: A machine learning framework to efficiently construct a solar deployment database in the united states," *Joule*, vol. 2, no. 12, pp. 2605–2617, 2018.

[2] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2818–2826, 2016.

[3] J. Yuan, H.-H. L. Yang, O. A. Omitaomu, and B. L. Bhaduri, "Large-scale solar panel mapping from aerial images using deep convolutional networks," in *2016 IEEE International Conference on Big Data (Big Data)*, pp. 2703–2708, IEEE, 2016.

[4] J. M. Malof, K. Bradbury, L. M. Collins, and R. G. Newell, "Automatic detection of solar photovoltaic arrays in high resolution aerial imagery," *Applied energy*, vol. 183, pp. 229–240, 2016.

[5] J. M. Malof, L. M. Collins, K. Bradbury, and R. G. Newell, "A deep convolutional neural network and a random forest classifier for solar photovoltaic array detection in aerial imagery," in *2016 IEEE International Conference on Renewable Energy Research and Applications (ICRERA)*, pp. 650–654, IEEE, 2016.

[6] K. Bradbury, R. Saboo, T. L. Johnson, J. M. Malof, A. Devarajan, W. Zhang, L. M. Collins, and R. G. Newell, "Distributed solar photovoltaic array location and extent dataset for remote sensing object identification," *Scientific data*, vol. 3, no. 1, pp. 1–9, 2016.

[7] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," in *International Conference on Medical image computing and computer-assisted intervention*, pp. 234–241, Springer, 2015.

[8] V. Badrinarayanan, A. Kendall, and R. Cipolla, "Segnet: A deep convolutional encoder-decoder architecture for image segmentation," *IEEE transactions on pattern analysis and machine intelligence*, vol. 39, no. 12, pp. 2481–2495, 2017.

[9] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *arXiv preprint arXiv:1502.03167*, 2015.

[10] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[11] X. Li, S. Chen, X. Hu, and J. Yang, "Understanding the disharmony between dropout and batch normalization by variance shift," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2682–2690, 2019.

[12] N. Usuyama, "Unet/fcn pytorch," 2018.

[13] J. Camilo, R. Wang, L. M. Collins, K. Bradbury, and J. M. Malof, "Application of a semantic segmentation convolutional neural network for accurate automatic detection and mapping of solar photovoltaic arrays in aerial imagery," *arXiv preprint arXiv:1801.04018*, 2018.

[14] B. Bischke, P. Helber, J. Folz, D. Borth, and A. Dengel,

[5] Initially, we had not implemented the cross validation. But we realized, quite late, that error bars were asked. Therefore, we started new training jobs, but we did not have the time for more than 20 epochs.

"Multi-task learning for segmentation of building footprints with deep neural networks," in *2019 IEEE International Conference on Image Processing (ICIP)*, pp. 1480–1484, IEEE, 2019.

[15] A. Kendall, Y. Gal, and R. Cipolla, "Multi-task learning using uncertainty to weigh losses for scene geometry and semantics," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 7482–7491, 2018.

[16] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[17] A. Dutta and A. Zisserman, "The vgg image annotator (via)," *arXiv preprint arXiv:1904.10699*, 2019.

Appendix A: Figures

1. Metrics

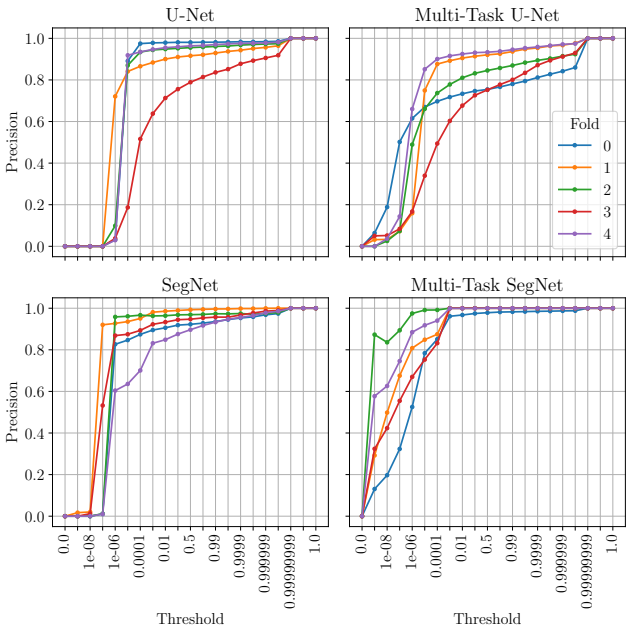

FIG. 13. Detection precision w.r.t. the threshold after 20 epochs of training.

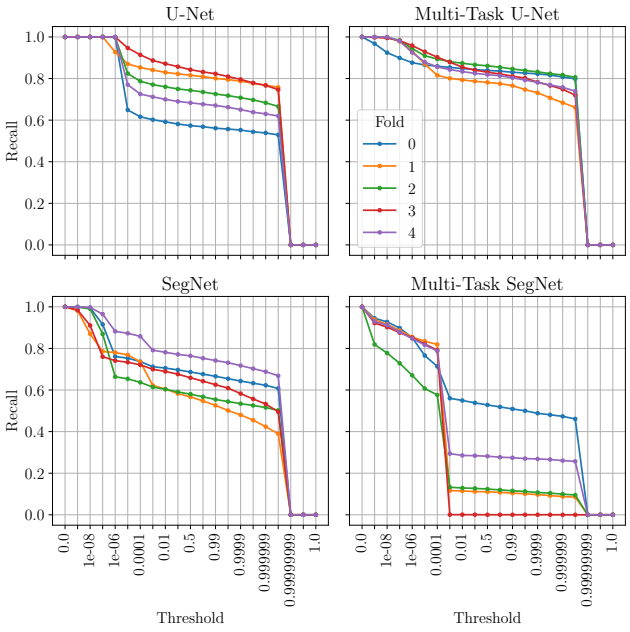

FIG. 14. Detection recall w.r.t. the threshold after 20 epochs of training.

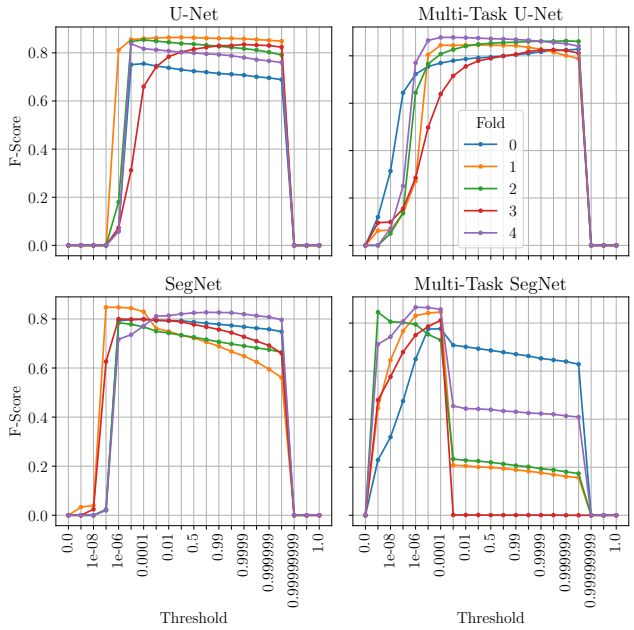

FIG. 15. Detection F-Score w.r.t. the threshold after 20 epochs of training.

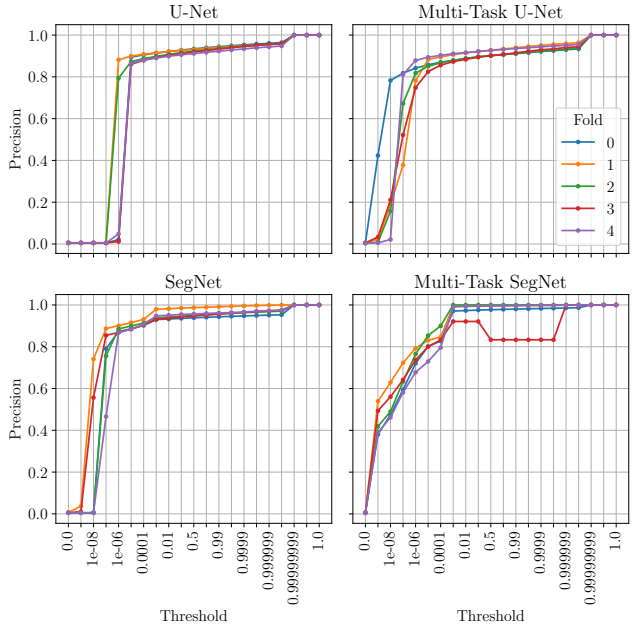

FIG. 16. Segmentation precision w.r.t. the threshold after 20 epochs of training.

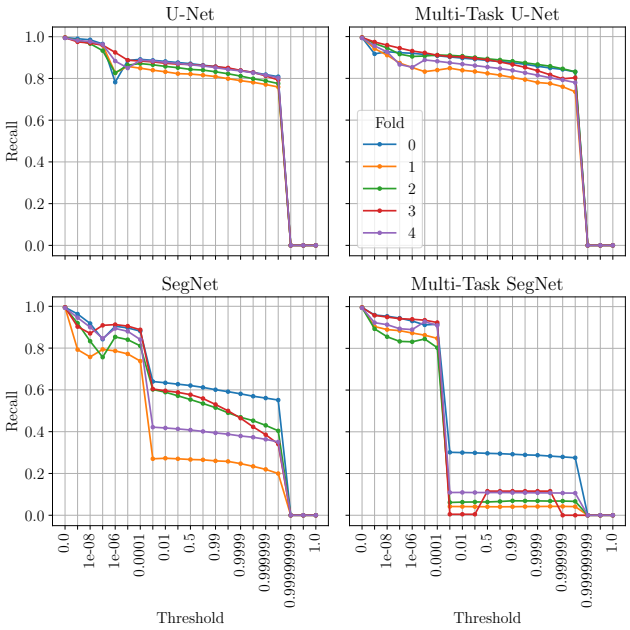

FIG. 17. Segmentation recall w.r.t. the threshold after 20 epochs of training.

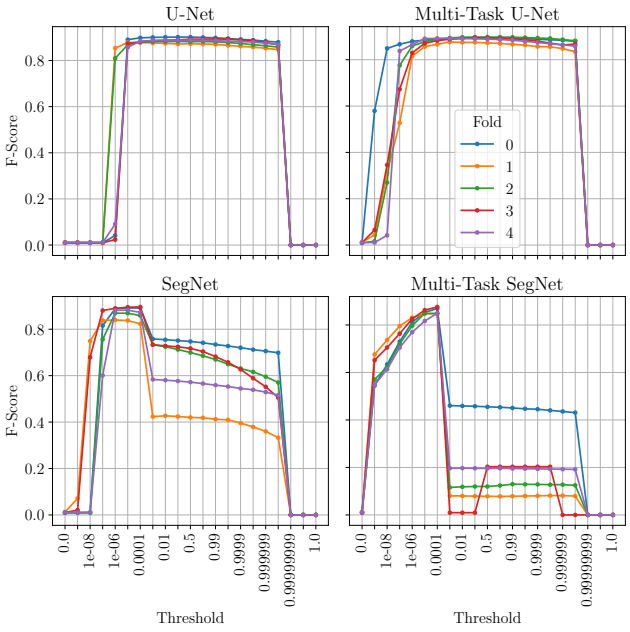

FIG. 18. Segmentation F-Score w.r.t. the threshold after 20 epochs of training.

2. Prediction examples

In this section, a few predictions on positive images are presented. The images were chosen within the fold 1 and so as to cover a fair variety of cases.

It should be noted that their distribution is *not* respected. In each of the following figures, each column is composed of a raw image, its mask and the predictions of, in this order, the U-Net, Multi-Task U-Net, SegNet and Multi-Task SegNet networks.

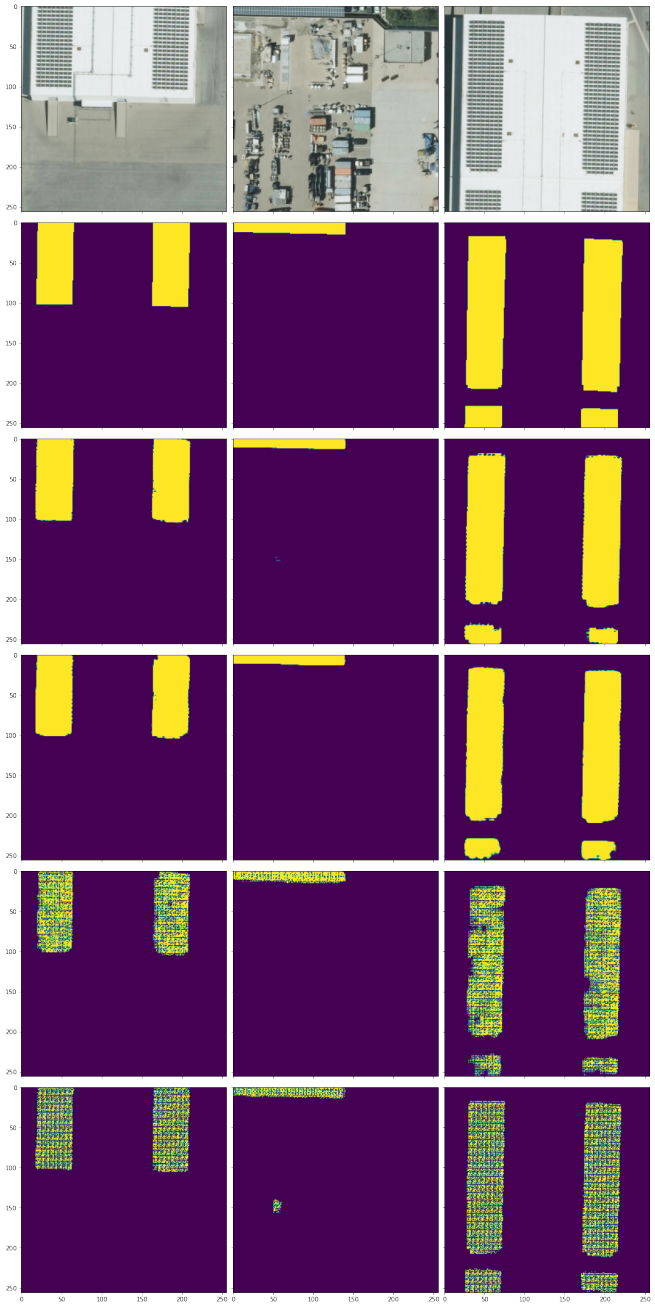

FIG. 19. Sample of easy predictions.

In FIG. 19, the four networks detect accurately the panels. However, where U-Net and Multi-Task U-Net are barely hesitant, SegNet and Multi-Task SegNet show a lot of uncertainty. It is actually the case for every prediction. In order to improve the predictions, we could have considered the *closing* operation to merge nearby spots together.

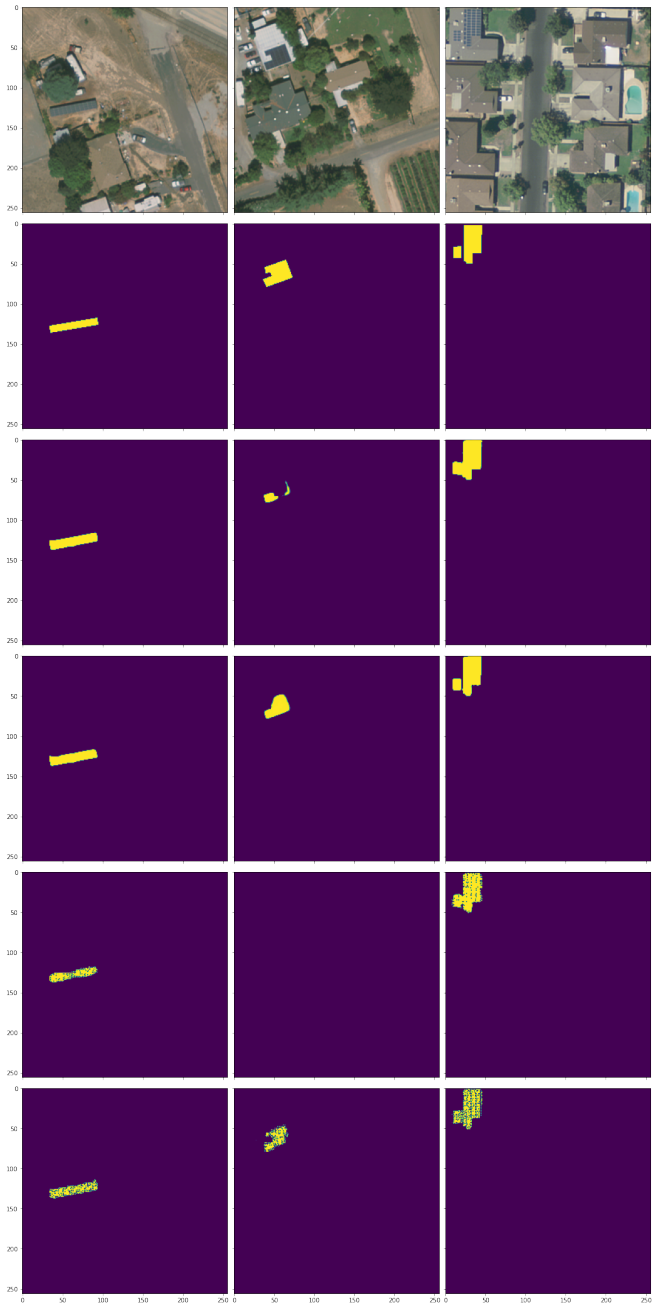

FIG. 20. Sample of average predictions.

In FIG. 20, it appears that Multi-Task U-Net delimits better the boundaries than U-Net, especially for the third image where it successfully differentiates the two sub-arrays.

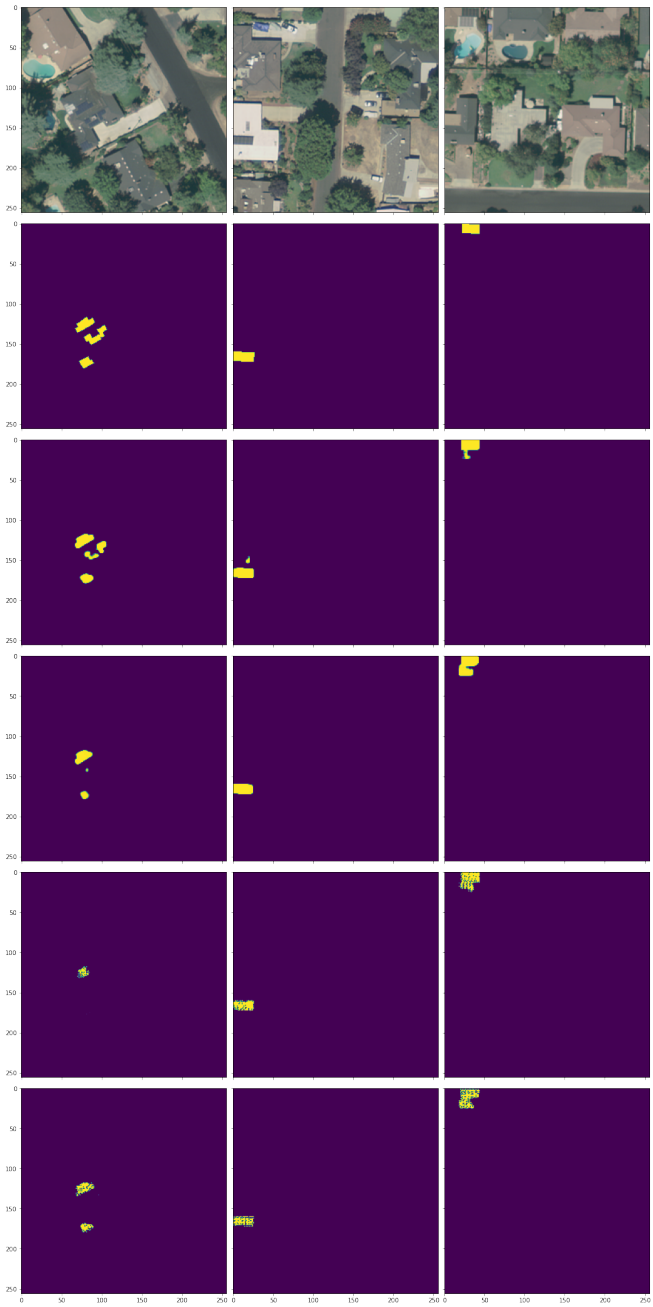

FIG. 21. Sample of average predictions.

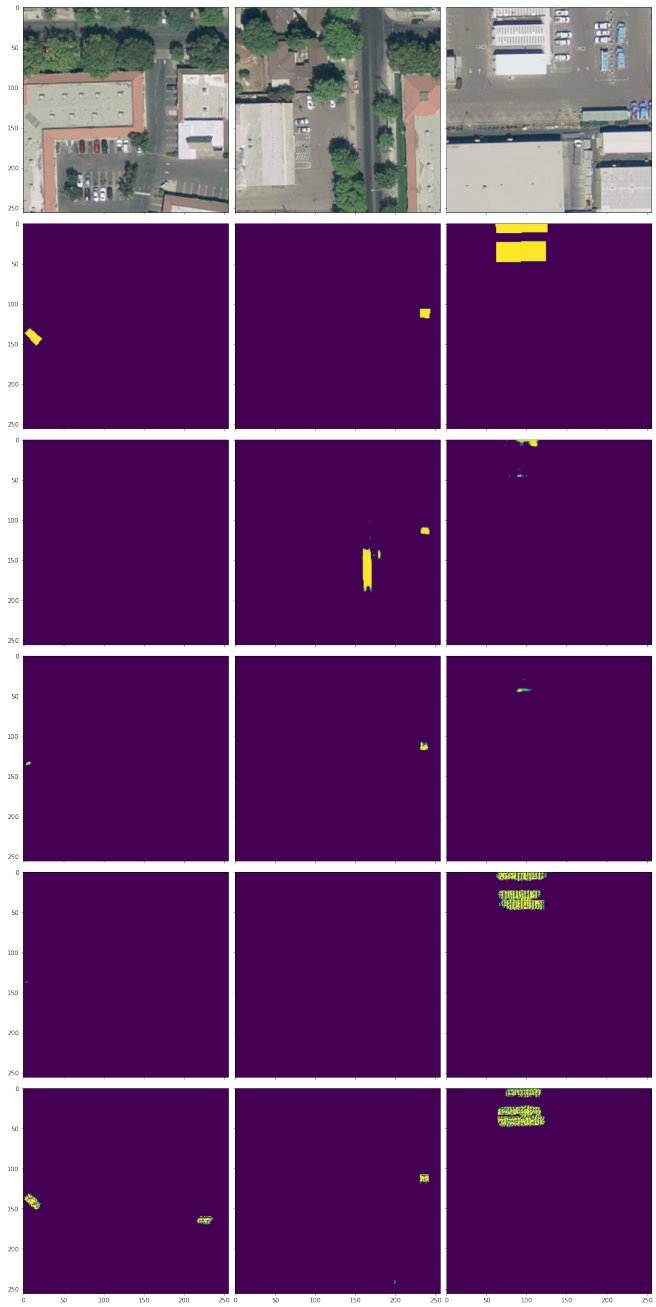

FIG. 22. Sample of bad predictions.

Images of FIG. 21, are pretty average as well. However, one can note that all four networks seem to detect an array that is not annotated in the third image. By looking closely, we see that it is indeed the case. This raises the question of the quality of our training set.

In FIG. 22, the U-Net and Multi-Task U-Net seem to struggle with the third image while SegNet and Multi-Task SegNet don't. This might be due to the unusual look of the annotated arrays.