Supplementary materials for Polygonal Building Extraction by Frame Field Learning

Nicolas Girard[1] Dmitriy Smirnov[2] Justin Solomon[2] Yuliya Tarabalka[3]
[1]Université Côte d'Azur, Inria [2]Massachusetts Institute of Technology [3]LuxCarta Technology

Contents

1. Frame field learning details **1**
1.1. Model architecture 1
1.2. Losses 1
1.3. Handling numerous heterogeneous losses . . 1
1.4. Training details 2

2. Frame field polygonization details **2**
2.1. Data structure 3
2.2. Active Skeleton Model 4
2.3. Corner-aware polygon simplification 6
2.4. Detecting building polygons in planar graph . 6

3. Experimental setup details **7**
3.1. Datasets 7
3.2. Metrics 7

4. Additional results **8**
4.1. CrowdAI dataset 8
4.1.1 Complexity vs. fidelity 8
4.1.2 Ablation study 8
4.1.3 Additional runtimes 10
4.2. Inria OSM dataset 10
4.3. Inria polygonized dataset 11
4.4. Private dataset 11

1. Frame field learning details

1.1. Model architecture

We show in Fig. 1 how we add a frame field output to an image segmentation backbone. The backbone can be any (possibly pretrained) network as long as it outputs an F-dimensional feature map $\widehat{y}_{backbone} \in \mathbb{R}^{F \times H \times W}$.

1.2. Losses

We define image segmentation loss functions below:

$$L_{BCE}(y, \widehat{y}) = \frac{1}{HW} \sum_{x \in I} y(x) \cdot \log(\widehat{y}(x)) + (1 - y(x)) \cdot \log(1 - \widehat{y}(x)), \quad (1)$$

$$L_{Dice}(y, \widehat{y}) = 1 - 2 \cdot \frac{|y \cdot y| + 1}{|y + y| + 1}, \quad (2)$$

$$L_{int} = \alpha \cdot L_{BCE}(y_{int}, \widehat{y}_{int}) + (1 - \alpha) \cdot L_{Dice}(y_{int}, \widehat{y}_{int}), \quad (3)$$

$$L_{edge} = \alpha \cdot L_{BCE}(y_{edge}, \widehat{y}_{edge}) + (1 - \alpha) \cdot L_{Dice}(y_{edge}, \widehat{y}_{edge}), \quad (4)$$

where $0 < \alpha < 1$ is a hyperparameter. In practice, $\alpha = 0.25$ gives good results.

For the frame field, we show a visualization of the L_{align} loss in Fig. 2.

1.3. Handling numerous heterogeneous losses

We linearly combine our eight losses using eight coefficients, which can be challenging to balance. Because the losses have different units, we first compute a normalization coefficient $N_{\langle loss\ name \rangle}$ by computing the average of each loss on a random subset of the training dataset using a randomly-initialized network. Then each loss can be normalized by this coefficient. The total loss is a linear combination of all normalized losses:

$$\begin{aligned} &\lambda_{int} \frac{L_{int}}{N_{int}} + \lambda_{edge} \frac{L_{edge}}{N_{edge}} + \lambda_{align} \frac{L_{align}}{N_{align}} + \lambda_{align90} \frac{L_{align90}}{N_{align90}} \\ &+ \lambda_{smooth} \frac{L_{smooth}}{N_{smooth}} + \lambda_{int\ align} \frac{L_{int\ align}}{N_{int\ align}} \\ &+ \lambda_{edge\ align} \frac{L_{edge\ align}}{N_{edge\ align}} + \lambda_{int\ edge} \frac{L_{int\ edge}}{N_{int\ edge}}, \end{aligned} \quad (5)$$

where the $\lambda_{\langle loss\ name \rangle}$ coefficients are to be tuned. It is also possible to separately group the main losses and the regularization losses and have a single λ coefficient balancing the two loss groups:

$$\lambda L_{main} + (1 - \lambda) L_{regularization}, \quad (6)$$

with

$$L_{main} = \frac{L_{int}}{N_{int}} + \frac{L_{edge}}{N_{edge}} + \frac{L_{align}}{N_{align}}, \quad (7)$$

$$\begin{aligned} L_{regularization} = &\frac{L_{align90}}{N_{align90}} + \frac{L_{smooth}}{N_{smooth}} \\ &+ \frac{L_{int\ align}}{N_{int\ align}} + \frac{L_{edge\ align}}{N_{edge\ align}} + \frac{L_{int\ edge}}{N_{int\ edge}}. \end{aligned} \quad (8)$$

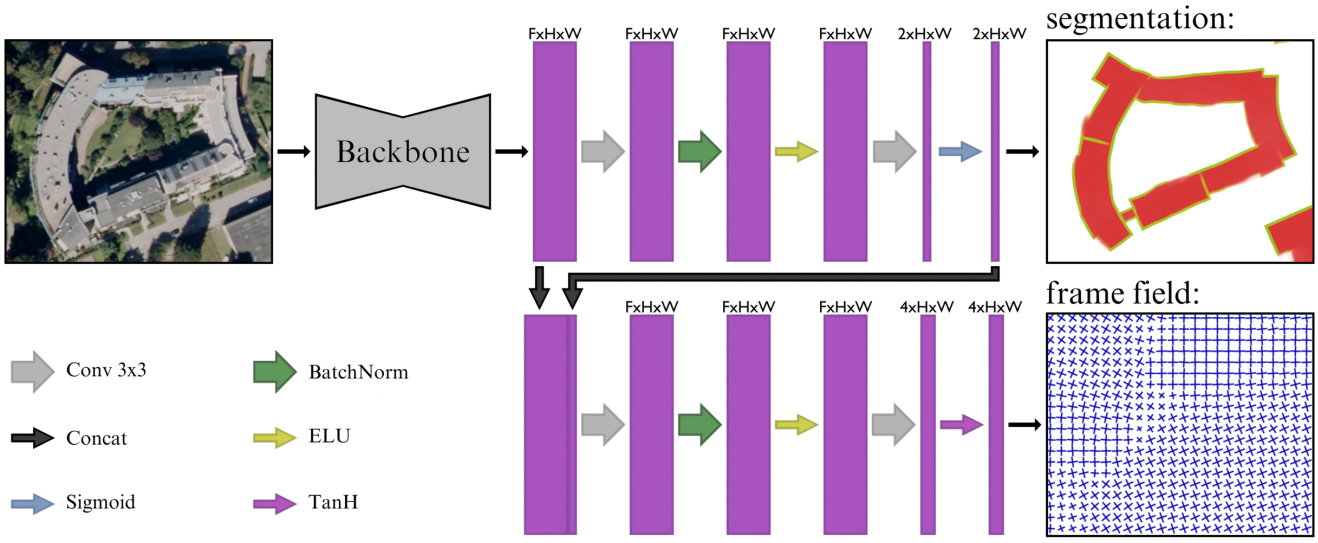


Figure 1: Details of our network's architecture with the addition of the frame field output.

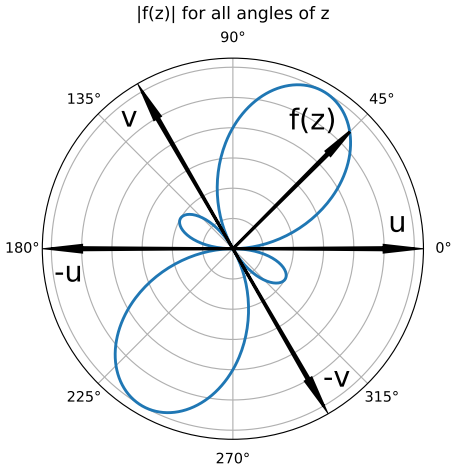


Figure 2: Visualization of the frame field align loss L_{align} (in blue) for a certain configuration of $\{-u, u, -v, v\}$ and all possible ground truth $z = e^{i\theta_\tau}$ directions.

In practice we started experiments with the single-coefficient version with $\lambda = 0.75$ and then used the multi-coefficient version to have more control by setting $\lambda_{int} = \lambda_{edge} = 10$, $\lambda_{align} = 1$, $\lambda_{align90} = 0.2$, $\lambda_{smooth} = 0.005$, $\lambda_{int\ align} = \lambda_{edge\ edge} = \lambda_{int\ edge} = 0.2$.

1.4. Training details

We do not heavily tune our hyperparameters: once we find a value that works based on validation performance we keep it across ablation experiments. We employ early stopping for the U-Net16 and DeepLabV3 models (25 and 15 epochs, respectively) chosen by first training the full method on the training set of the *CrowdAI dataset*, choosing the epoch number of the lowest validation loss, and finally re-training the model on the train and validation sets for that number of total epochs.

Segmentation losses L_{int} and L_{edge} are both a combination of 25% cross-entropy loss and 75% Dice loss. To balance the losses in ablation experiments, we used the single-coefficient version with $\lambda = 0.75$. For our best performing model UResNet101 we used the multi-coefficients version to have more control by setting $\lambda_{int} = \lambda_{edge} = 10$, $\lambda_{align} = 1$, $\lambda_{align90} = 0.2$, $\lambda_{smooth} = 0.005$, $\lambda_{int\ align} = \lambda_{edge\ edge} = \lambda_{int\ edge} = 0.2$. The U-Net16 was trained on 4 GTX 1080Ti GPUs in parallel on 512×512 patches and a batch size of 16 per GPU (effective batch size 64). For all training runs, we compute for each loss its normalization coefficient $N_{\langle loss\_name \rangle}$ on 1000 batches before optimizing the network.

Our method is implemented in PyTorch [14]. On the *CrowdAI dataset*, training takes 2 hours per epoch on 4 1080Ti GPUs for the U-Net16 model and 3.5 hours per epoch for the DeepLabV3 backbone on 4 2080Ti GPUs. Inference with the U-Net16 on a 5000×5000 image (requires splitting into 1024×1024 patches) takes 7 seconds on a Quadro M2200 (laptop GPU).

2. Frame field polygonization details

We expand here on the algorithm and implementation details of our frame field polygonization method.

2.1. Data structure

Our polygonization method needs to be initialized with geometry, which is then optimized to align to the frame field (among other objectives we will present later).

In the case of extracting individual buildings, we use the marching squares [10] contour finding algorithm on the predicted interior probability map y_{int} with an isovalue ℓ (set to 0.5 in practice). The result is a collection of contours $\{\mathcal{C}_i\}$ where each contour is a sequence of 2D points:

$$\mathcal{C}_i = \big((r_0, c_0), (r_1, c_1), ..., (r_{n_i-1}, c_{n_i-1})\big)\,.$$

where $r_i, c_i \in \mathbb{R}$ correspond to vertex i's position along the row axis and the column axis respectively (they are not restricted to being integers). A contour is generally closed with $(r_0, c_0) = (r_{n_i-1}, c_{n_i-1})$, but it can be open if the corresponding object touches the border of the image (therefore start and end vertices are not the same).

In the case of extracting buildings with potential adjoining buildings sharing a common wall, we extract the skeleton graph of the predicted edge probability map y_{edge}. This skeleton graph is a hyper-graph made of nodes connected together by chains of vertices (i.e., polylines) called paths (see Fig. 4 for examples). To obtain this skeleton graph, we first compute the skeleton image using the thinning method [19] on the binary edge mask (computed by thresholding y_{edge} with $\ell = 0.5$). It reduces binary objects to a one-pixel-wide representation. We then use the Skan [12] Python library to convert this representation to a graph representation connecting those pixels. The resulting graph is a collection of paths that are polylines connecting junction nodes together. We use an appropriate data structure only involving arrays (named tensors in deep learning frameworks) so that it can be manipulated by the GPU. We show in Fig. 3 an infographic of the data structure. A sequence of node coordinates "*pos*" holds the location of all nodes $i \in [0\,..\,n-1]$ belonging to the skeleton:

$$pos = \big((r_0, c_0), (r_1, c_1), ..., (r_{n-1}, c_{n-1})\big)$$

where n is the total number of skeleton pixels and $(r_i, c_i) \in [0\,..\,H-1] \times [0\,..\,W-1]$ correspond to the row number and column number, respectively (of skeleton pixel i). The skeleton graph connects junction nodes through paths, which are polylines made up of connected vertices. These paths are represented by the "paths" binary matrix $P_{p,n}$ where element (i, j) is one if node j is in path i. This $P_{p,n}$ is sparse, and, thus, it is more efficient to use the CSR (compressed sparse row) format, which represents a matrix by three (one-dimensional) arrays respectively containing nonzero values, the column indices and the extents of rows. As $P_{p,n}$ is binary we do not need the array containing nonzeros values. The column indices array, which we name "*path_index*" holds the column indices of all "on" elements:

$$path\_index = (j_0, j_1, ..., j_{n-n_{junctions}+n_{degrees\ sum}-1}),$$

where $n_{junctions}$ is the total number of junction nodes, $n_{degrees\ sum}$ is the sum of the degrees of all junction nodes and $\forall k \in [0\,..\,n-n_{junctions}+n_{degrees\ sum}-1], j_k \in [0\,..\,n-1]$. The extents of rows array which we name "$path\_delim$" holds the starting index of each row (it also contains an extra end element which is the number of non-zeros elements n for easier computation):

$$path\_delim = (s_0, s_1, ..., s_p)\,.$$

Thus, in order to get row i of $P_{p,n}$ we need to look up the slice (s_i, s_{i+1}) of $path\_index$. In the skeleton graph case, this representation is also easily interpretable. Indices of path nodes are all concatenated in *path_index* and *path_delim* is used to separate those concatenated paths. And finally a sequence of integers "degrees" stores for each node the number of nodes connected to it:

$$degrees = (d_0, d_1, ..., d_{n-1})\,.$$

As a collection of contours is a type of graph, in order to use a common data structure in our algorithm, we also use the skeleton graph representation for the contours $\{\mathcal{C}_i\}$ given by the marching squares algorithm (note we could use other contour detection algorithms for initialization). Each contour is thus an isolated path in the skeleton graph.

In order to fully leverage the parallelization capabilities of GPUs, the largest amount of data should be processed concurrently to increase throughput, i.e., we should aim to use the GPU memory at its maximum capacity. When processing a small image (such as 300×300 pixels from the *CrowdAI dataset*), only a small fraction of memory is used. We thus build a batch of such small images to process them at the same time. As an example, on a GTX 1080Ti, we use a polygonization batch size $B = 1024$ for processing the *CrowdAI dataset*, which induces a significant speedup. Building a batch of images is very simple: they can be concatenated together along an additional batch dimensions, i.e., B images $I_i \in \mathbb{R}^{3\times H\times W}$ are grouped in a tensor $\mathbf{I} \in \mathbb{R}^{B\times 3\times H\times W}$. This is the case for the output segmentation probability maps as well as the frame field. However, it is slightly more complex to build a batch of skeleton graphs because of their varying sizes. Given a collection of skeleton graphs $\{(pos_i, degrees_i, path\_index_i, path\_delim_i)\}_{i\in[..B-1]}$, all pos_i and $degrees_i$ are concatenated in their first dimension to give batch arrays:

$$pos_{batch} = [pos_0, pos_1, \ldots, pos_{B-1}]\,,$$

and:

$$degrees_{batch} = [degrees_0, degrees_1, \ldots, degrees_{B-1}].$$

All $path\_index_i$ need their indices to be shifted by a certain offset:

$$offset_i = \sum_{k=0}^{i-1} |pos_k|,$$

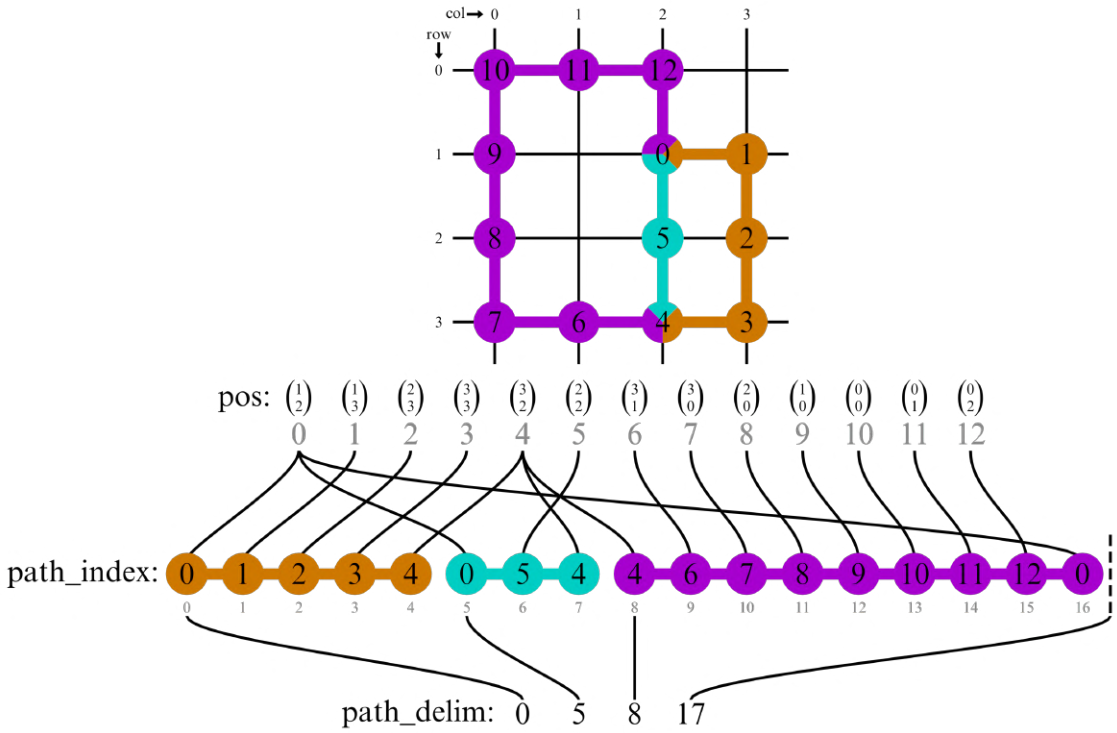


Figure 3: Our data structure of an example skeleton graph. It represents two buildings with a shared wall, necessitating 3 polyline paths. Here nodes 0 and 4 are shared among paths and are thus repeated in $path\_index$. We can see $path\_index$ is a concatenation of the node indices in pos of the paths. Finally, $path\_delim$ is used to store the separation indices in $path\_index$ of those concatenated paths. Indices of arrays are in gray.

with $|pos_k|$ the number of points in pos_k, so that they point to the new locations in pos_{batch} and $degree_{batch}$. They are then concatenated in their first dimension: $path\_index_{batch} = [path\_index_0 + offset_0, \ldots, path\_index_{B-1} + offset_{B-1}]$. In a similar manner, we concatenate all $path\_delim_i$ into $path\_delim_{batch}$ while taking care of adding the appropriate offset. We then obtain a big batch skeleton graph which is represented in the same way as a single skeleton graph. In order to later recover individual skeleton graphs in the batch, similar to *path_delim*, we need a *batch_delim* array that stores the starting index of each individual skeleton graph in the *path_delim* array (it also contains an extra end element which is the total number of paths in the batch for easier computation). While we apply the optimization on the batched arrays pos_{batch}, $path\_index_{batch}$, and so on, for readability we will now refer to them as pos, $path\_index$ and so on. Note that in the case of big images (such as 5000×5000 pixels from the *Inria dataset*), we set the batch size to 1, as the probability maps, the frame field, and the skeleton graph data structure fills the GPU's memory well.

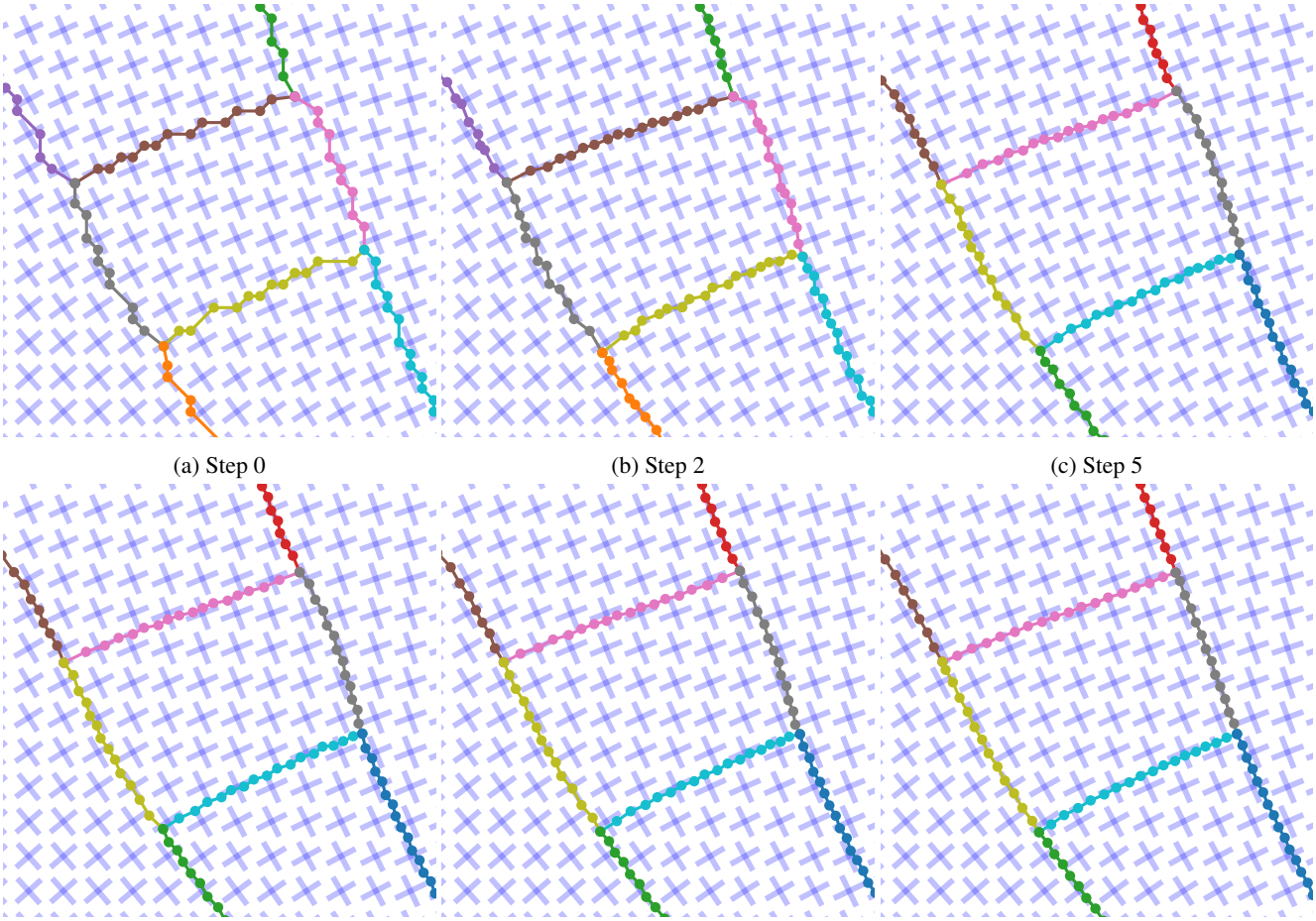
At this point the data structure is fixed, i.e., it will not change during optimization. Only the values in pos will be modified. This data structure is efficiently manipulated in parallel on the GPU. All the operations needed for the various computations performed in the next sections are run in parallel on the GPU.

We compute other tensors from this minimal data structure which will be useful for computations:

- path_pos = pos[path_index] which expands the positions tensor for each path (junction nodes are thus repeated in $path\_pos$).
- A $batch$ tensor which for each node in pos_batch stores the index $i \in [0\,..\,B-1]$ of the individual skeleton this node belongs to. This is used to easily sample the batched segmentation maps and the batched frame fields at the position of a node.

2.2. Active Skeleton Model

We adapt the formulation of the Active Contours Model (ACM) to an Active Skeleton Model (ASM) in order to optimize our batch skeleton graph. The advantage of using the energy minimization formulation of ACM is to be able to add extra terms if needed (we can imagine adding regularization terms to, e.g., reward 90°corners, uniform curvature, and straight walls).



(a) Step 0 (b) Step 2 (c) Step 5

(d) Step 10 (e) Step 25 (f) Step 100

Figure 4: ASM optimization steps (zoomed example). Frame field in blue crosses.

Energy terms will be parameterized by the node positions $\mathbf{p} \in pos$, which are the variables being optimized. The first important energy term is $E_{probability}$ which aims to fit the skeleton paths to the contour of the building interior probability map $y_{int}(v)$ at a certain probability level ℓ (which we set to 0.5 in practice, just like the isovalue used to initialize the contours by marching squares):

$$E_{probability} = \sum_{\mathbf{p} \in pos} (y_{int}(\mathbf{p}) - l)^2 \,.$$

The value $y_{int}(\mathbf{p})$ is computed by bilinear interpolation so that gradients can be back-propagated to $\mathbf{p}$. Additionally, $y_{int}(\mathbf{p})$ implicitly entails using the $batch$ array to know which slice in the batch dimension of $y_{int} \in \mathbb{R}^{B \times 1 \times H \times W}$ to sample $\mathbf{p}$ from. This will be the case anytime batched image-like tensors are sampled at a point $\mathbf{p}$. In the case of the marching squares initialization, this $E_{probability}$ energy is actually zero at the start of optimization, since the initialized contour already is at isovalue ℓ. For the skeleton graph initialization, paths that trace inner walls between adjoining buildings will not be affected since the gradient is zero in a neighborhood of homogeneous values (i.e., $y_{int} = 1$ inside buildings).

The second important energy term is $E_{frame\ field\ align}$ which aligns each edge of the skeleton paths to the frame field. Edge vectors are computed in parallel as:

$$\mathbf{e} = \text{path\_pos}\,[1{:}] \;\; - \text{path\_pos}[{:}{-}1]\,,$$

while taking care of removing from the energy computation "hallucinated" edges between paths (using the $path\_delim$ array). For readability we call E the set of valid edge vectors. For each edge vector $\mathbf{e} \in E$, we refer to its direction as $\mathbf{e}_{dir} = \frac{\mathbf{e}}{\|\mathbf{e}\|}$. We also refer to its center point as $\mathbf{e}_{center} = \frac{1}{2}(\text{path\_pos}\,[1{:}] + \text{path\_pos}\,[{:}{-}1])$. The frame field align term is defined as:

$$E_{frame\ field\ align} = \sum_{\mathbf{e} \in E} |f(\mathbf{e}_{dir}; c_0(\mathbf{e}_{center}), c_2(\mathbf{e}_{center}))|^2 \,.$$

This is the term that disambiguates between slanted walls and corners and results in regular-looking contours.

The last important term is the internal energy term E_{length} which ensures node distribution along paths remains homogeneous as well as tight:

$$E_{length} = \sum_{\mathbf{e} \in E} |\mathbf{e}|^2 .$$

All energy terms are then linearly combined: $E_{total} = \lambda_{probability} E_{probability} + \lambda_{frame\ field\ align} E_{frame\ field\ align} + \lambda_{length} E_{length}$. In practice, the final result is robust to different values of coefficients for each of these three energy terms, and we determine them using a small cross-validation set. The total energy is minimized with the RMSprop [18] gradient descent method with a smoothing constant $\gamma = 0.9$ with an initial learning rate of $\eta = 0.1$ which is exponentially decayed. The optimization is run for 300 iterations to ensure convergence. Indeed since the geometry is initialized to lie on building boundaries, it is not expected to move more than a few pixels and the optimization converges quickly. See Fig. 4 for a zoomed example of different stages of the ASM optimization.

2.3. Corner-aware polygon simplification

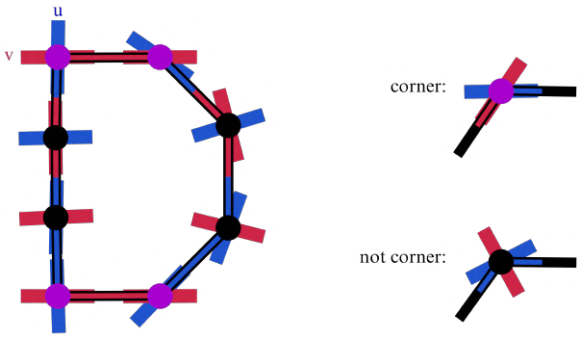


Figure 5: Corner detection using the frame field. For each vertex, the frame field is sampled at that location (with nearest neighbor) and represented by the $\{\pm u, \pm v\}$ vectors.

We now have a collection of connected polylines that forms a planar skeleton graph. As building corners should not be removed during simplification, only polylines between corners are simplified. For the moment our data structure encodes a collection of polyline paths connecting junction nodes in the skeleton graph. However, a single path can represent multiple walls. It is the case for example of an individual rectangular building: one path describes its contour while it has 4 walls. In order to split paths into sub-paths each representing a single wall we need to detect building corners along a path and add this information to our data structure. This is another reason to use a frame field input, as it implicitly models corners: at a given building corner, there are two tangents of the contour. The frame field learned to align one of u or $-u$ to the first tangent and one of v or $-v$ to the other tangent. Thus when walking along a contour path if the local direction of walking switches from $\pm u$ to $\pm u$ or vice versa, it means we have come across a corner, see Fig. 5 for an infographic for corner detection. Specifically for each node i with position $\mathbf{p} = \text{path\_pos[i]}$ its preceding and following edge vectors are computed as: $\mathbf{e}_{prev} = \text{path\_pos[i]} - \text{path\_pos[i-1]}$ and $\mathbf{e}_{next} = \text{path\_pos[i+1]} - \text{path\_pos[i]}$. As the frame field is represented by the coefficients $\{c_0, c_2\}$ at each pixel, we first need to convert it to its $\{u, v\}$ representation with the simple formulas of eq. 9.

	$\lvert\langle \mathbf{e}_{prev}, u_{\mathbf{p}} \rangle\rvert < \lvert\langle \mathbf{e}_{prev}, v_{\mathbf{p}} \rangle\rvert$	$\lvert\langle \mathbf{e}_{prev}, v_{\mathbf{p}} \rangle\rvert < \lvert\langle \mathbf{e}_{prev}, u_{\mathbf{p}} \rangle\rvert$
$\lvert\langle \mathbf{e}_{next}, u_{\mathbf{p}} \rangle\rvert < \lvert\langle \mathbf{e}_{next}, v_{\mathbf{p}} \rangle\rvert$	False	True
$\lvert\langle \mathbf{e}_{next}, v_{\mathbf{p}} \rangle\rvert < \lvert\langle \mathbf{e}_{next}, u_{\mathbf{p}} \rangle\rvert$	True	False

Table 1: Summary table for deciding if node i with position $\mathbf{p} = \text{path\_pos[i]}$ is a corner (True) or not (False).

$$\begin{cases} c_0 = u^2 v^2 \\ c_2 = -(u^2 + v^2) \end{cases} \iff \begin{cases} u^2 = -\frac{1}{2}\left(c_2 + \sqrt{c_2^2 - 4c_0}\right) \\ v^2 = -\frac{1}{2}\left(c_2 - \sqrt{c_2^2 - 4c_0}\right) \end{cases} . \quad (9)$$

The frame field is sampled at that position $\mathbf{p}$: $u_{\mathbf{p}} = u(\mathbf{p})$ and $v_{\mathbf{p}} = v(\mathbf{p})$. Alignment between $\mathbf{e}_{prev}, \mathbf{e}_{next}$ and $\pm u_{\mathbf{p}}, \pm v_{\mathbf{p}}$ is measured with the absolute scalar product so that it is agnostic to the sign of u and v. For example alignment between $\mathbf{e}_{prev}$ and $\pm u_{\mathbf{p}}$ is measured by $|\langle \mathbf{e}_{prev}, u_{\mathbf{p}} \rangle|$ and if $|\langle \mathbf{e}_{prev}, u_{\mathbf{p}} \rangle| < |\langle \mathbf{e}_{prev}, v_{\mathbf{p}} \rangle|$ then $\mathbf{e}_{prev}$ is aligned to $\pm v$ and not $\pm u$. The same is done for $\mathbf{e}_{next}$. Finally if $\mathbf{e}_{prev}$ and $\mathbf{e}_{prev}$ do not align to the same frame field direction, then node i is a corner. As a summary for corner cases we refer to Table. 1.

Because the path positions are concatenated together in the $path\_pos$ tensor, some care must be taken for nodes at the extremities of paths (i.e., junction nodes) as they do not have both preceding and following edges. The $path\_delim$ tensor is used to mark those nodes as not corners. Once corners are detected we obtain a tensor $is\_corner\_index = \{i \mid \text{node } i \text{ is a corner}\}$ which can be used to separate paths into sub-paths each representing a single wall by merging is_corner_index with the $path\_delim$ tensor through concatenation and sorting.

Now that each sub-path polyline represents a single wall between two corners, we apply the Ramer-Douglas-Peucker [15, 3] simplification algorithm separately on all sub-path polylines. As explained in the related works, the simplification tolerance ε represents the maximum Hausdorff distance between the original polyline and the simplified one.

2.4. Detecting building polygons in planar graph

To obtain our final output of building polygons, the collection of polylines is polygonized by detecting connecting

regions separated by the polylines. A list of polygonal cells that partition the entire image is thus obtained. The last step computes a building probability value for each polygon using the predicted interior probability map and removes low-probability polygons (in practice those that have an average probability less than 50%).

3. Experimental setup details

3.1. Datasets

CrowdAI dataset. The *CrowdAI dataset* [16] originally has 280741 training images, 60317 validation images, and 60697 test images. All images are 300×300 pixels with unknown ground sampling distance, although they are aerial images. As the ground truth annotations of the test set are unreleased because of the challenge, we use the original validation set as our test set and discard the original test images as is commonly done by other methods comparing themselves with that dataset [7, 6]. We then use 75% of the original training images as our initial training set and 25% for validation. Out final models are then trained on the entire original training set with hyperparameters selected using our validation test.

Inria dataset. The *Inria dataset* [11] has 360 aerial images of 5000×5000 pixels each with a Ground Sampling Distance of 30 cm. In total, 10 cities from Europe and the USA are represented, each city having 36 images. Each image is accompanied by its building ground truth mask with an average of a few thousand buildings per image. This dataset provides building ground truth in the form of binary mask images for each image. However, our method requires the ground truth annotations to be in vector format (polygons) so that the ground truth for the frame field can be computed: the tangent angle θ_τ used in L_{align}. We thus build two dataset variants with vector annotations.

The first variant is the *Inria OSM dataset* for which we discard completely the original ground truth masks and instead download annotations from Open Street Map (OSM) [13]. Because the OSM annotations are not always aligned, we align them using [4]. We randomly split the images into train (50%), validation (25%), and test (25%) sets. Because the OSM annotations have a lot of missing buildings in certain images, our test results on this dataset are somewhat skewed. Thus, for the test images, we manually select those with few missing buildings in the annotations, giving us 54 test images in total.

The second variant is the *Inria Polygonized dataset* for which we take the original ground truth masks and convert them to polygon format with our polygonization method. In this setting, the input to our network (we used the small U-Net16) is just the binary mask and the output a frame field. In order to train this model, we need a dataset of (binary masks, θ_τ) pairs. We used the OSM annotations of the *Inria OSM dataset*, which we rasterized to obtain the input binary masks and which we used to compute θ_τ. After our model finished training, we applied our frame field polygonization method on the original binary masks of the *Inria dataset* and their predicted frame fields. The new *Inria polygonized dataset* is thus made of (RGB image, polygonized annotations) pairs. We thus obtain the same ground truth as the original dataset but in vector format. This allows us to only use the same ground truth data as the other competitors of the Inria Aerial Image Labeling challenge and thus we can directly compare our method to them. Thus we keep the original train and test splits which do not have any cities overlap and tests cross-city generalization (the principal aim of the associated challenge). We then split the original train split into our train (75%) and validation (25%) splits.

Private dataset. The *private dataset* is a large-scale dataset of satellite images built by a company we collaborate with. The images in this dataset were acquired using three types of satellites (Pleiades, WorldView, and GeoEye) over different types of cities (dense, industrial, residential areas, and city centers). We uniformized the image sampling at 50 cm/pixel spatial resolution, with 3-band RGB images. 57 images of 30 cities across 5 continents are present in the training dataset. The size of images varies from around 2000×2000 pixels to 20000×20000 pixels. The total dataset covers an area spanning around 700 km^2. The building outline polygons were manually labeled precisely by an expert. Satellite images are more challenging than aerial images (such as the CrowdAI and Inria images) because they are less clear due to atmospheric effects. This dataset also contains much more varied images compared to CrowdAI and Inria, making up for its smaller size. We preprocess the training images by splitting them into smaller 512×512 pixel patches. We then keep 90% of patches for training and 10% for validation.

3.2. Metrics

IoU, AP and AR. The usual metric for the image segmentation task is Intersection over Union (IoU) which computes the overlap between a predicted segmentation and the ground truth annotation. The IoU is then used to compute other metrics such as the MS COCO [8] Average Precision (AP and its variants AP_{50}, AP_{75}, AP_S, AP_M, AP_L) and Average Recall (AR and its variants AR_{50}, AR_{75}, AR_S, AR_M, AR_L) evaluation metrics. Precision and recall are computed for a certain IoU threshold: detections with an IoU above the threshold are counted as true positives whiles others are false positives and ground truth annotations with an IoU below the threshold are false negatives. Each object is also given a score value representing the model's confidence in the detection. In our case, it is the mean value of the in-

terior probability map inside the detection. The Precision-Recall curve can be obtained by varying the score threshold that determines what is counted as a model-predicted positive detection. Average Precision (AP) is the average value of the precision across all recall values and Average Recall (AR) is the maximum recall given a fixed number of detections per image (100 in our case). Finally, the mean Average Precision (mAP) is calculated by taking the mean AP over multiple IoU thresholds (from 0.50 to 0.95 with a step of 0.05). Likewise for the mean Average Recall (mAR). Following MS COCO's convention, we make no distinction between AP and mAP (and likewise AR and mAR) and assume the difference is clear from context. The AP_{50} variant is AP computed with a single IoU threshold of 50% (similarly for AP_{75}, AR_{50}, and AR_{75}). The AP_S, AP_M and AP_L variants are AP computed for small ($area < 32^2$), medium ($32^2 < area < 96^2$) and large ($area > 96^2$) objects respectively (like-wise for the AR equivalents).

Max tangent angle error. We introduce a max tangent angle error metric between predicted polygons and the ground truth to capture the regularity of the predicted contours. A max tangent angle scalar error is computed for each predicted contour. Only predicted contours with at least 50% overlap with the ground truth are selected, so that their measure makes sense. Each predicted contour is first sampled homogeneously with points $\{P_i\}_{i\in[1..n]}$ (specifically a point is sampled every 0.1 pixel). Then the P_i points are projected to the ground truth, meaning for each P_i we find the closest point Q_i belonging to the ground truth annotation. For both sequences of points P_i and Q_i, corresponding normed tangent directions are computed as:

$$T(P_i) = \frac{P_{i+1} - P_i}{\|P_{i+1} - P_i\|} \quad \text{and} \quad T(Q_i) = \frac{Q_{i+1} - Q_i}{\|Q_{i+1} - Q_i\|} \,.$$

The angle differences between the two are computed from the scalar product:

$$\Delta\theta_i = cos^{-1}(\langle T(P_i), T(Q_i)\rangle) \,.$$

Before computing the maximum angle error $\max_i \Delta\theta_i$ along the whole contour, some angle errors $\Delta\theta_i$ need to be filtered out as they are invalid. Angle error invalidity is due to the projection step. Indeed around ground truth corners, part of the predicted contour will we be squashed to be zero-length for example. Another issue is when P_i and P_{i+1} are projected to two different ground truth polygon sides: the projected edge $P_{i+1} - P_i$ does not represent a ground truth tangent anymore. We thus filter out tangents whose projection is stretched more than a factor of 2, i.e., we keep all $\Delta\theta_j, \forall j \in V$ where $V = \{j \mid j \in [1\,..\,n], \frac{1}{2} < \frac{\|Q_{i+1} - Q_i\|}{\|P_{i+1} - P_i\|} < 2\}$. The final max tangent angle error for that contour is then:

$$E_{\textit{max tangent angle}} = \max_{j\in V} \Delta\theta_j \,.$$

As each contour gives a scalar error, we aggregate all the errors for a certain dataset by averaging this max tangent angle error metric.

Method	Mean max angle error ↓
UResNet101 (no field), simple poly.	51.9°
UResNet101 (with field), simple poly.	45.1°
U-Net variant [2], ASIP poly. [6]	44.0°
U-Net variant [2], UResNet101 **our** poly.	36.6°
Zorzi et al. [20] poly.	36.8°
UResNet101 (no L_{smooth}), **our** poly.	33.6°
UResNet101 (no $L_{int\ align}$ and $L_{edge\ align}$), **our** poly.	33.5°
UResNet101 (no $L_{int\ edge}$), **our** poly.	33.4°
UResNet101 (no $L_{align90}$), **our** poly.	33.2°
PolyMapper [7]	33.1°
UResNet101 (with field), **our** poly.	31.9°

Table 2: Mean *max tangent angle errors* over all the original validation polygons of the *CrowdAI dataset* [16].

4. Additional results

4.1. CrowdAI dataset

4.1.1 Complexity vs. fidelity

For the polygon complexity/fidelity trade-off ablation study we plot the AP and AR scores for difference simplification tolerance values ε on the *CrowdAI dataset*.

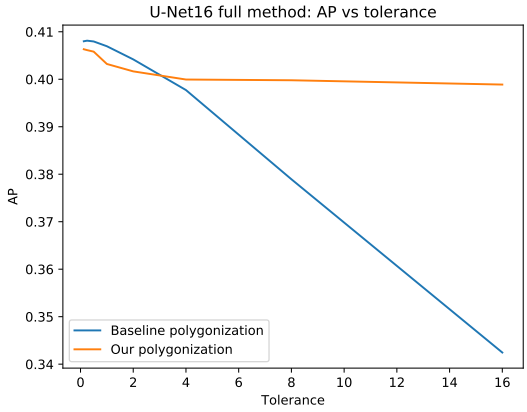
We perform an analysis of the polygonization complexity/fidelity trade-off by changing the tolerance value ε of the baseline simplification method and our corner-aware method. Fig. 6a shows that preventing the removal of building corners ensures key points of the contours and the global shape of the building remain intact even with extreme simplification tolerance values. We also plot the AP and AR values of both methods while increasing the tolerance value ε in Fig. 6. As expected the score of our method does not drop, unlike the simple polygonization method.

Our polygonization method allows the complexity-to-fidelity ratio to be tuned with the easy-to-interpret tolerance value ε of the Ramer-Douglas-Peucker algorithm, unlike ASIP [6], which uses a non-intuitive parameter λ to balance complexity and fidelity energies during polygonal partition optimization. Finally, PolyMapper [7] does not have the ability to tune the complexity-to-fidelity ratio.

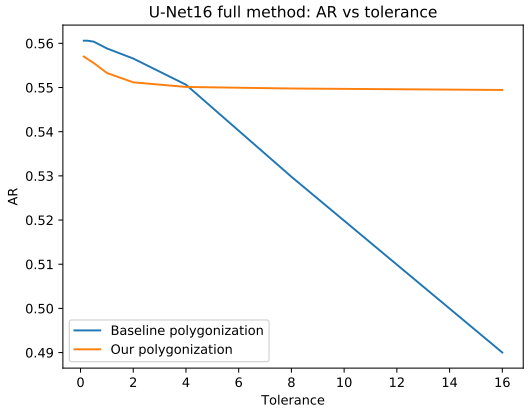
4.1.2 Ablation study

We visualize the predicted classification maps from each ablation study for an example test sample in Fig. 7. Both for the U-Net16 and DeepLab101 backbones, the (full) method yields more regular classification maps with sharper corners compared to (no field). Additionally, only learning the

(a) Effect of increasing the simplification tolerance value ε from 0.5 px (left), then 2 px, then 8 px and 16 px (right).



(b) AP for both our corner-aware method and the simple polygonization for various tolerance value ε.



(c) AR for both our corner-aware method and the simple polygonization for various tolerance value ε.

Figure 6: Comparison between the baseline simplification algorithm with our corner-aware one. Both take the same classification map as input, but the baseline does not use the frame field. The corner-aware simplification guarantees that no corners will be simplified, regardless of the tolerance value ε.

frame field with (no coupling losses) is insufficient, as can be seen in Fig. 7d.

We observe the effect of only optimizing for IoU when removing coupling losses: we see that it does not impact AP and AR metrics in Table 3, while in Fig. 7 the (full) segmentations are clearly sharper compared to the (no coupling losses) ones.

In terms of AP and AR metrics, adding a frame field improves the final score (full) compared to (no field) for all backbones: U-Net16, DeepLab101 and UResNet101 (see Table 3).

We also visually compare our frame field polygonization method with the simple baseline polygonization algorithm (both when the frame field is computed and when it is not) in Fig. 8. The UResNet101 without frame field learning and whose results are polygonized with the simple method performs the worst (see Fig. 8c), with the UResNet101 with frame field learning and whose results are polygonized with the simple method performs already much better (see Fig. 8b). Our UResNet101 with frame field learning and whose results are polygonized with our frame field polygonization method performs the best, with better corners

Method	$AP\uparrow$	$AP_{50}\uparrow$	$AP_{75}\uparrow$	$AP_S\uparrow$	$AP_M\uparrow$	$AP_L\uparrow$	$AR\uparrow$	$AR_{50}\uparrow$	$AR_{75}\uparrow$	$AR_S\uparrow$	$AR_M\uparrow$	$AR_L\uparrow$
U-Net16 (no field), mask	50.9	74.3	59.5	24.5	65.6	66.3	55.9	77.9	64.7	29.8	71.2	74.6
U-Net16 (no field), simple poly.	50.5	76.6	59.1	22.6	66.2	69.3	54.8	78.5	63.5	26.8	71.2	75.2
U-Net16 (no coupling losses), mask	53.7	77.7	62.8	25.7	69.0	68.9	57.7	79.2	66.4	31.0	73.4	74.4
U-Net16 (with field), mask	53.6	77.8	62.8	25.1	69.4	69.5	57.6	79.0	66.4	29.7	74.1	75.2
U-Net16 (with field), simple poly.	49.6	73.8	58.1	21.2	65.5	67.0	53.8	75.6	62.2	25.5	70.5	72.5
U-Net16 (with field), **our** poly.	50.5	76.6	59.3	20.4	67.4	69.0	55.3	78.1	64.0	25.7	72.8	75.0
DeepLab101 (with field)	54.9	78.1	64.9	25.6	71.2	76.8	58.7	79.8	68.1	29.5	75.8	81.6
DeepLab101 (no field)	52.8	75.2	61.8	26.1	67.7	75.0	57.8	78.4	66.7	30.3	73.7	81.8
UResNet101 (no field), mask	62.4	86.7	72.7	36.2	76.3	81.1	67.5	90.5	77.4	46.8	79.5	86.5
UResNet101 (no field), simple poly.	61.1	87.4	71.2	35.1	74.5	82.3	64.7	89.4	74.1	41.7	77.9	85.7
UResNet101 (with field), mask	64.5	89.3	74.6	40.3	76.6	84.0	68.1	91.0	77.7	47.5	80.0	86.7
UResNet101 (with field), simple poly.	61.7	87.7	71.5	35.8	74.9	83.0	65.4	89.9	74.6	42.6	78.6	85.9
UResNet101 (with field), **our** poly.	61.3	87.5	70.6	34.0	75.1	83.1	65.0	89.4	73.9	41.2	78.7	86.0
UResNet101 (no $L_{align90}$), mask	64.2	88.6	74.6	40.0	76.4	83.7	67.8	90.9	77.5	47.1	79.7	86.4
UResNet101 (no $L_{align90}$), simple poly.	61.4	87.7	71.4	35.4	74.5	82.7	65.0	89.7	74.4	42.1	78.2	85.6
UResNet101 (no $L_{align90}$), **our** poly.	61.1	87.5	70.6	34.1	74.9	82.8	64.7	89.3	73.8	41.2	78.4	85.6
UResNet101 (no $L_{int\ edge}$), mask	63.8	88.5	74.4	39.6	75.9	83.3	67.3	90.7	77.0	46.6	79.3	86.2
UResNet101 (no $L_{int\ edge}$), simple poly.	61.0	87.6	70.6	35.2	74.1	82.4	64.6	89.5	74.0	41.7	77.8	85.3
UResNet101 (no $L_{int\ edge}$), **our** poly.	60.9	87.4	70.5	33.7	74.4	82.5	64.4	89.1	73.4	40.7	78.1	85.4
UResNet101 (no $L_{int\ align}$ and $L_{edge\ align}$), mask	64.7	89.3	74.7	40.5	76.7	84.2	68.2	91.0	77.9	47.6	80.1	86.8
UResNet101 (no $L_{int\ align}$ and $L_{edge\ align}$), simple poly.	61.8	87.7	71.5	35.8	74.9	83.3	65.4	89.9	74.7	42.5	78.6	86.0
UResNet101 (no $L_{int\ align}$ and $L_{edge\ align}$), **our** poly.	61.5	87.5	71.3	34.2	75.2	83.4	65.0	89.5	74.0	41.3	78.8	86.1
UResNet101 (no L_{smooth}), mask	64.2	88.6	74.6	40.1	76.5	83.5	67.8	90.8	77.5	47.2	79.8	86.1
UResNet101 (no L_{smooth}), simple poly.	61.6	87.7	71.5	35.7	74.8	82.6	65.2	89.7	74.5	42.3	78.4	85.4
UResNet101 (no L_{smooth}), **our** poly.	61.3	87.5	70.7	34.1	75.0	82.7	64.8	89.3	73.9	41.1	78.6	85.5
U-Net variant [2], UResNet101 **our** poly.	67.0	92.1	75.6	42.1	84.2	92.7	73.2	93.5	81.1	48.8	87.3	95.4
Mask R-CNN [5] [17]	41.9	67.5	48.8	12.4	58.1	51.9	47.6	70.8	55.5	18.1	65.2	63.3
PANet [9]	50.7	73.9	62.6	19.8	68.5	65.8	54.4	74.5	65.2	21.8	73.5	75.0
PolyMapper [7]	55.7	86.0	65.1	30.7	68.5	58.4	62.1	88.6	71.4	39.4	75.6	75.4
U-Net variant [2], ASIP poly. [6]	65.8	87.6	73.4	39.3	87.0	91.9	78.7	94.3	86.1	57.2	91.2	97.6

Table 3: AP and AR results on the *CrowdAI dataset* [16] for all polygonization experiments. (with field) refers to models trained with our full frame field learning method. (no field) refers to models trained without any frame field output. "mask" refers to the output raster segmentation mask of the network, "our poly." refers to our frame field polygonization method, and "simple poly." refers to the baseline polygonization of marching squares followed by Ramer-Douglas-Peucker simplification.

using fewer vertices (see Fig. 8a). We can see our method provides the missing information needed to resolve ambiguous cases for polygonization and outputs more regular polygons.

Finally Table 2 and 3 also hold results for additional experiments of the ablation study which each remove a loss during training. We observe that removing one of those losses does not impact the AP or AR result of the final polygonization. However, if one of those loss is removed we observe a performance drop in terms of *max tangent angle errors*, with result polygons for all such experiments having a mean error slightly higher than PolyMapper (at 33.1°) while our full method achieves a mean error of 31.6°.

4.1.3 Additional runtimes

We report here the average runtimes for a 300×300 pixel patch of the different steps of the building polygonization pipeline of Zorzi et al. [20] along with corresponding GPU memory allocation (GTX 1080Ti):

1. segmentation: 0.152s with 20% GPU memory,
2. regularization: 0.269s with 12% GPU memory,
3. mask2poly: 0.257s with 19% GPU memory.

As we optimized our own method for maximum throughput, we want to compare to previous methods assuming perfect parallelization (as is done for the ASIP method in the main paper). Zorzi et al. would then get these runtimes:

1. segmentation: 0.0304s,
2. regularization: 0.03228s,
3. mask2poly: 0,04883s,

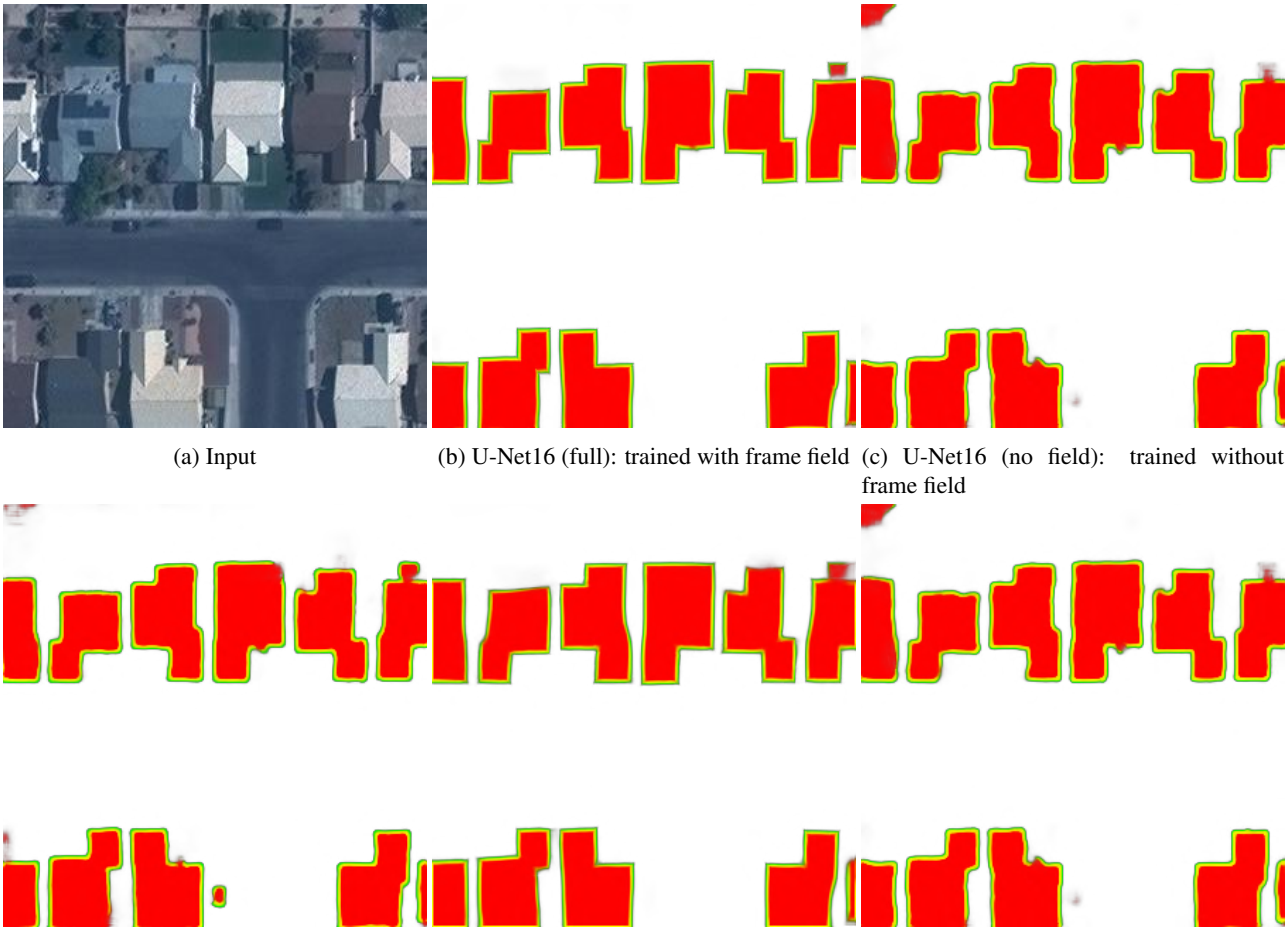
for a total of 0,11151s. For comparison we include the runtimes of all methods in Table 4, where we observe our method being competitive compared to previous works.

Method	Time (sec) ↓	Hardware
PolyMapper [7]	0.38	GTX 1080Ti
ASIP [6]	0.15	Laptop CPU
Zorzi et al. [20]	0.11	GTX 1080Ti
Ours	**0.04**	GTX 1080Ti

Table 4: Average time to extract buildings from a 300×300 pixel patch. **Ours** refers to UResNet101 (with field), our poly. ASIP's time does not include model inference.

4.2. Inria OSM dataset

We show bigger crops of the result of our frame field polygonization in Fig. 9, 10, and 11. We observe the ability of our method to separate adjoining buildings, handle complex shapes with big buildings having non-rectangular



(a) Input (b) U-Net16 (full): trained with frame field (c) U-Net16 (no field): trained without frame field

(d) U-Net16 (no coupling losses): trained with frame field but without coupling losses (e) DeepLab101 (full): trained with frame field (f) DeepLab101 (no field): trained without frame field

Figure 7: Classification predictions on a test sample for all training ablation studies.

shapes and possibly holes.

4.3. Inria polygonized dataset

We show a larger result comparison to other methods on the *Inria Polygonized dataset* in Fig 12, including the two best methods on the public leaderboard[1]. While the result from "Eugene Khvedchenya" and ICTNet acheive an mIoU over 80%, they detect buildings with segmentation masks that need polygonization. We thus used the simple polygonization method which follows the boundaries in the segmentation raster image. Their results have blob-like features, with rounded corners and non-regular contours.

In order to compare to the ASIP polygonization method, we started to run the ASIP algorithm on the 180 output probability maps of our network, corresponding to the 180 test images. However the ASIP method is not well-suited for such big images (5000×5000 pixels) with thousands of buildings, requiring a very high number of iterations (that we set to 10000). The runtime of ASIP varies greatly depending on the building density of images. For the most dense ones, it did not finish within a day of computation, making it impractical to run on the whole test dataset. As such we compare to the ASIP method only on the *CrowdAI dataset*.

4.4. Private dataset

Because training on the *private dataset* must be done on a restricted computer with limited access, we only train two models: U-Net16 (full) and U-Net16 (no field) until validation loss converges (around 1500 epochs). First we display segmentation raster outputs in Fig 13, 14 and 15 and final polygonal buildings in Fig 16, 17 and 18. Satellite images being more challenging than aerial images, non-regularized segmentations (no field) appear even more rounded than usual. However, our frame field learning and polygoniza-

[1] https://project.inria.fr/aerialimagelabeling/leaderboard/

(a) **Ours**: UResNet101 with frame field learning (full) and frame field polygonization

(b) UResNet101 with frame field learning and simple polygonization

(c) UResNet101 (no field) learning and simple polygonization

Figure 8: Extracted polygons with: our (full) frame field learning and polygonization method; our frame field learning and simple polygonization method; the (no field) learning and simple polygonization baseline. A low tolerance of $\varepsilon = 0.125$ pixel was chosen to compare precise contours.

tion (with field) still outputs clean, regular geometry, and separates adjoining buildings.

U-Net16 (no field), simple poly.

Ours: U-Net16 (with field), our poly.

Figure 9: Crop of results on the "sfo19" image from the *Inria OSM dataset*.

U-Net16 (no field), simple poly.

Ours: U-Net16 (with field), our poly.

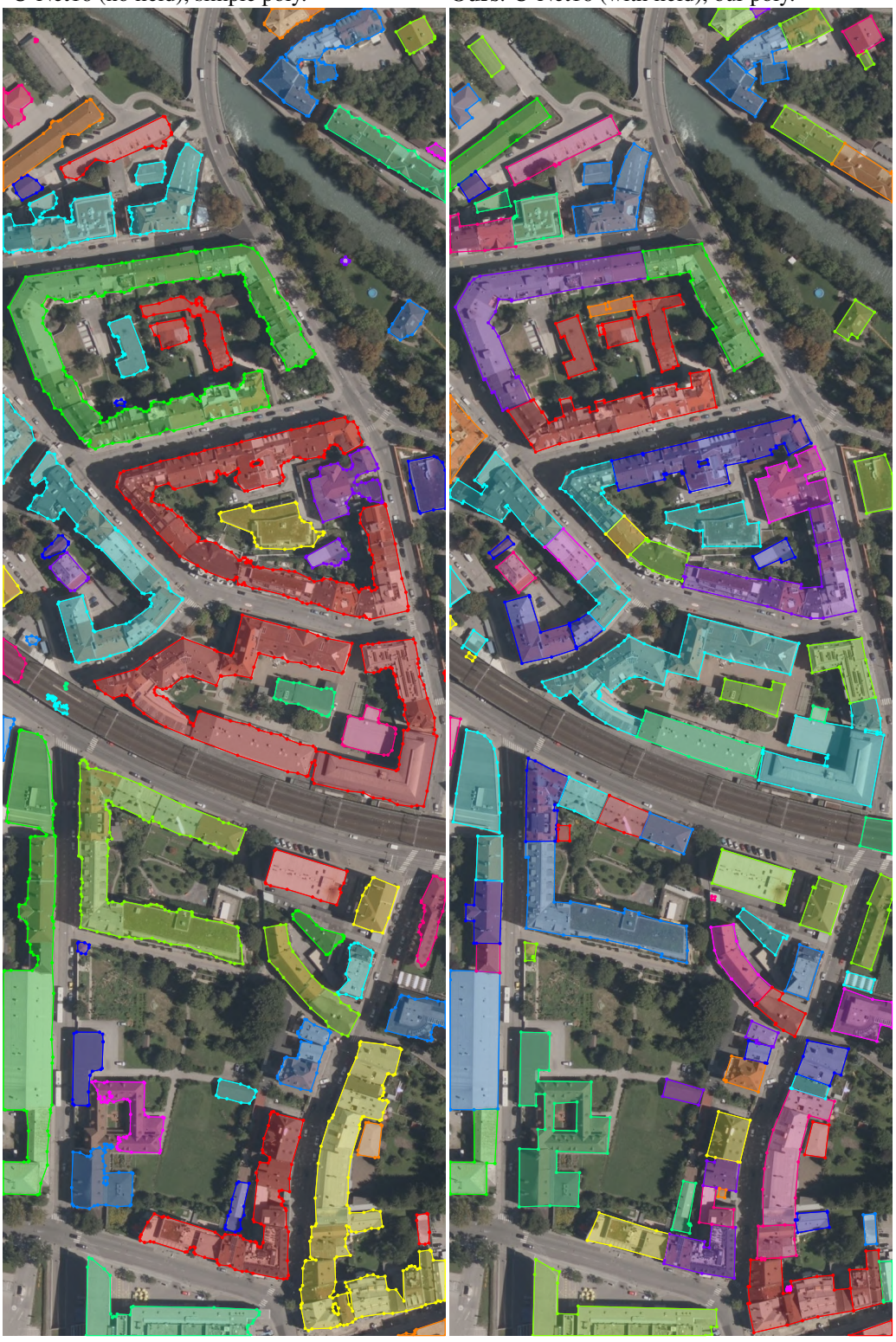


Figure 10: Crop of results on the “innsbruck19” image from the *Inria OSM dataset*.

U-Net16 (no field), simple poly.

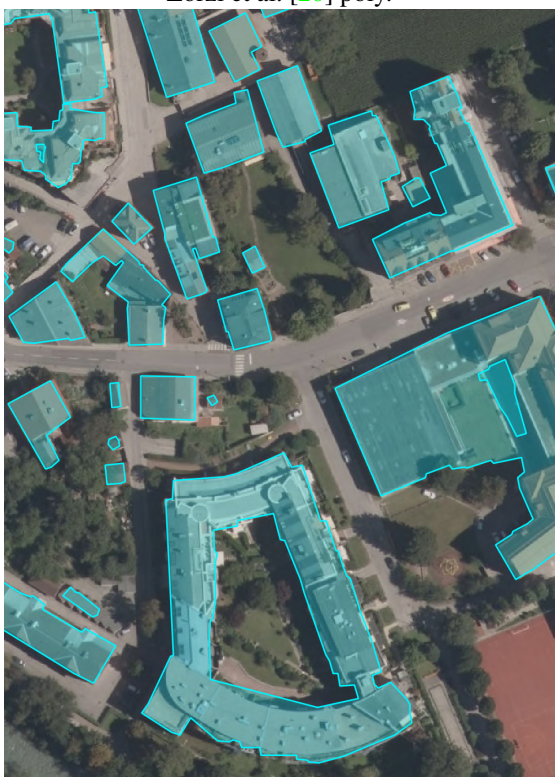
Ours: U-Net16 (with field), our poly.

Figure 11: Crop of results on the "vienna36" image from the *Inria OSM dataset*.

Eugene Khvedchenya[1], simple poly.

ICTNet [1], simple poly.

Zorzi et al. [20] poly.



Ours: UResNet101 (with field), our poly.

Figure 12: Crop of results on an *Inria Polygonized dataset* test image.

U-Net16 (no field)

Ours: U-Net16 (with field)

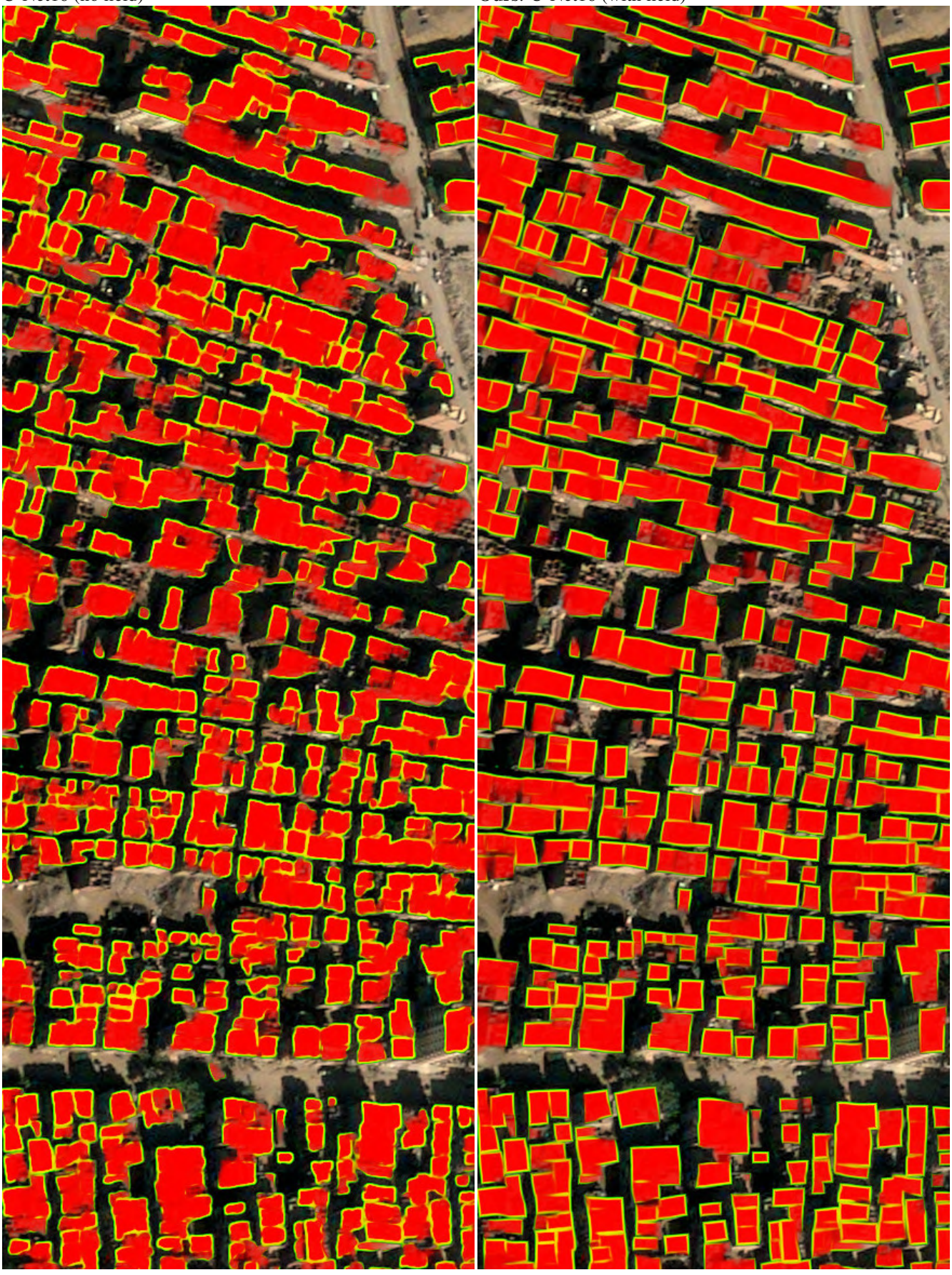


Figure 13: Crop results on the “Egypt” test image of the private dataset.

U-Net16 (no field)

Ours: U-Net16 (with field)

Figure 14: Crop results on the “Bangkok” test image of the private dataset.

U-Net16 (no field)

Ours: U-Net16 (with field)

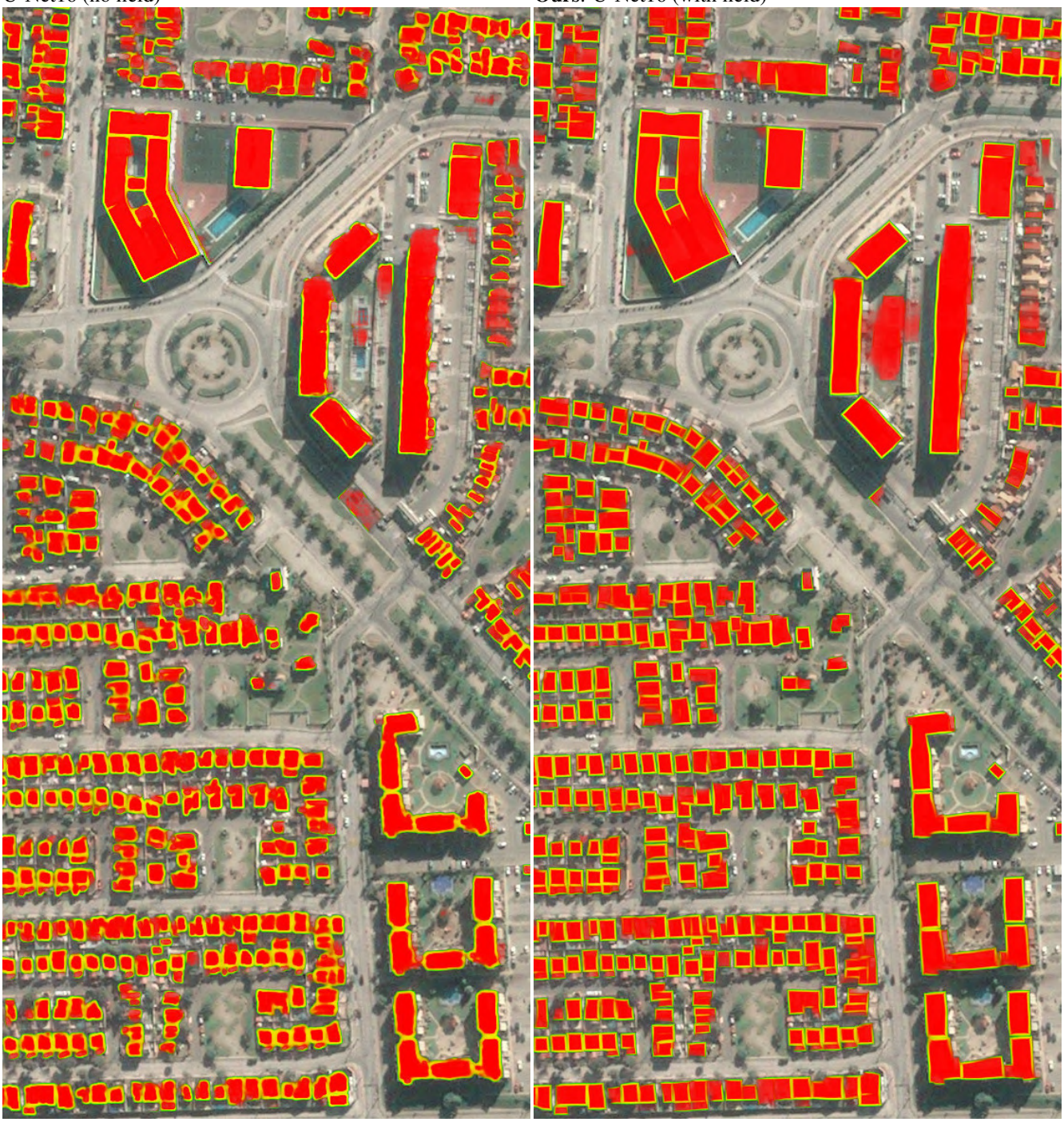


Figure 15: Crop results on the “Chile” test image of the private dataset.

U-Net16 (no field), simple poly.

Ours: U-Net16 (with field), our poly.

Figure 16: Crop results on the "Egypt" test image of the *private dataset*.

U-Net16 (no field), simple poly.

Ours: U-Net16 (with field), our poly.

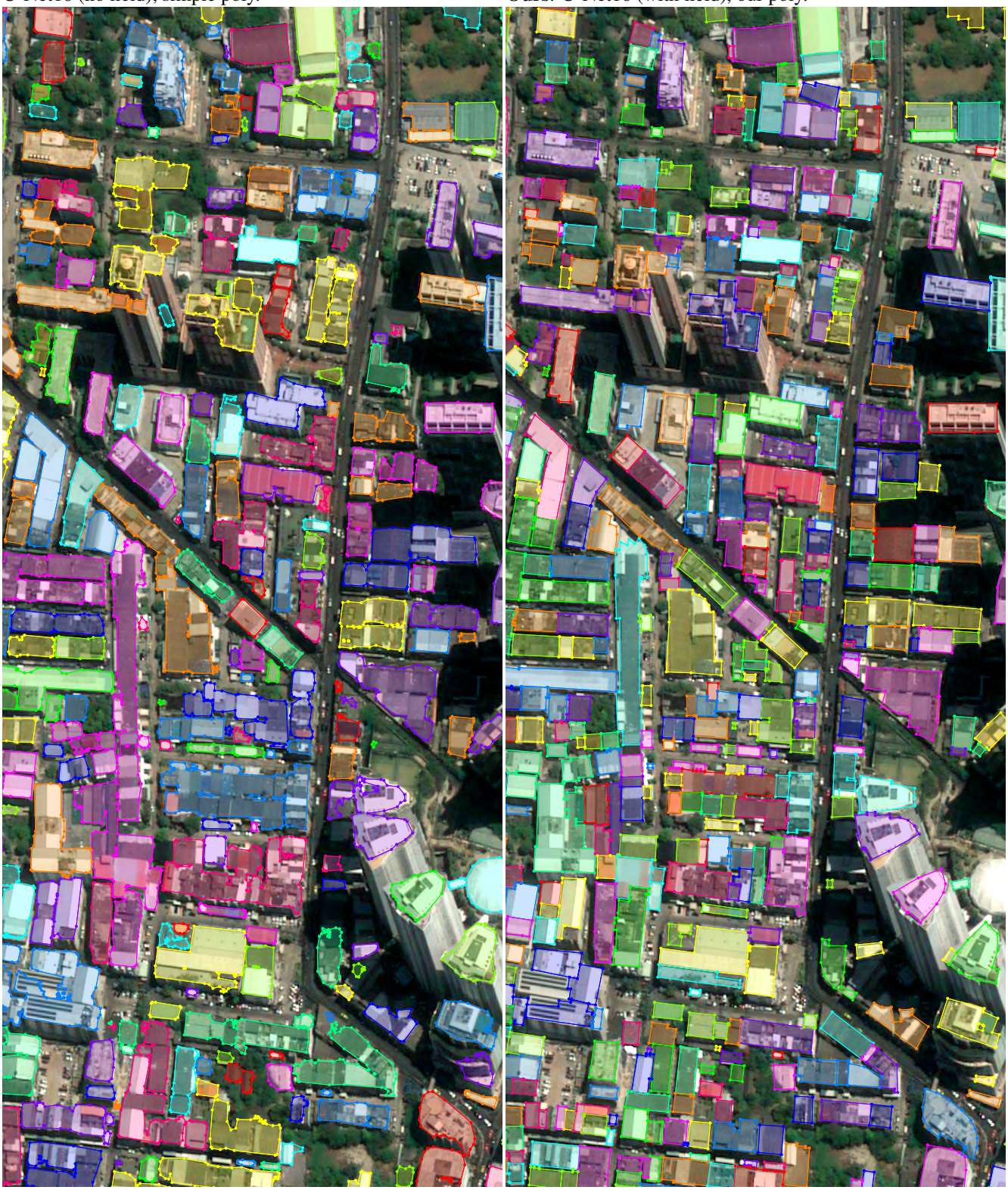


Figure 17: Crop results on the “Bangkok” test image of the *private dataset*.

U-Net16 (no field), simple poly.

Ours: U-Net16 (with field), our poly.

Figure 18: Crop results on the “Chile” test image of the *private dataset*.

References

[1] Bodhiswatta Chatterjee and Charalambos Poullis. On building classification from remote sensor imagery using deep neural networks and the relation between classification and reconstruction accuracy using border localization as proxy. In *2019 16th Conference on Computer and Robot Vision (CRV)*, pages 41–48, 2019. 16

[2] Jakub Czakon, Kamil A. Kaczmarek, Andrzej Pyskir, and Piotr Tarasiewicz. Best practices for elegant experimentation in data science projects. *EuroPython*, 2018. 8, 10

[3] David H. Douglas and Thomas K. Peucker. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *The Canadian Cartographer*, 10(2):112–122, 1973. 6

[4] Nicolas Girard, Guillaume Charpiat, and Yuliya Tarabalka. Noisy Supervision for Correcting Misaligned Cadaster Maps Without Perfect Ground Truth Data. In *IEEE International Geoscience and Remote Sensing Symposium (IGARSS)*, 2019. 7

[5] Kaiming He, Georgia Gkioxari, Piotr Dollar, and Ross Girshick. Mask r-cnn. In *The IEEE International Conference on Computer Vision (ICCV)*, Oct 2017. 10

[6] Muxingzi Li, Florent Lafarge, and Renaud Marlet. Approximating shapes in images with low-complexity polygons. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020. 7, 8, 10

[7] Zuoyue Li, Jan Dirk Wegner, and Aurélien Lucchi. Topological map extraction from overhead images. In *The IEEE International Conference on Computer Vision (ICCV)*, 2019. 7, 8, 10

[8] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. Microsoft coco: Common objects in context. In David Fleet, Tomas Pajdla, Bernt Schiele, and Tinne Tuytelaars, editors, *The European Conference on Computer Vision (ECCV)*, pages 740–755, Cham, 2014. Springer International Publishing. 7

[9] Shu Liu, Lu Qi, Haifang Qin, Jianping Shi, and Jiaya Jia. Path aggregation network for instance segmentation. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018. 10

[10] William Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. 1987. 3

[11] Emmanuel Maggiori, Yuliya Tarabalka, Guillaume Charpiat, and Pierre Alliez. Can semantic labeling methods generalize to any city? the inria aerial image labeling benchmark. In *IEEE International Geoscience and Remote Sensing Symposium (IGARSS)*, 2017. 7

[12] Juan Nunez-Iglesias. skan: skeleton analysis in python. https://github.com/jni/skan, 2017. 3

[13] OpenStreetMap contributors. Planet dump retrieved from https://planet.osm.org , 2017. 7

[14] 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 *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. 2

[15] Urs Ramer. An iterative procedure for the polygonal approximation of plane curves. *Computer graphics and image processing*, 1(3):244–256, 1972. 6

[16] Sharada Prasanna Mohanty. Crowdai dataset. https://www.crowdai.org/challenges/mapping-challenge/dataset_files, 2018. 7, 8, 10

[17] Sharada Prasanna Mohanty. Crowdai mapping challenge 2018: Baseline with mask rcnn. https://github.com/crowdai/crowdai-mapping-challenge-mask-rcnn, 2018. 10

[18] T. Tieleman and G. Hinton. Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural Networks for Machine Learning, 2012. 6

[19] T. Y. Zhang and C. Y. Suen. A fast parallel algorithm for thinning digital patterns. *Commun. ACM*, 27(3):236–239, Mar. 1984. 3

[20] Stefano Zorzi, Ksenia Bittner, and Friedrich Fraundorfer. Machine-learned regularization and polygonization of building segmentation masks, 2020. arXiv:2007.12587. 8, 10, 16