Building Footprint Detection Using Remote Sensing Data

Martinussen, Jakob Gerhard

Autumn, 2019

Department of Mathematical Sciences
Norwegian University of Science and Technology

Abstract

We present a system for predicting building footprints by using remote sensing data in the form of aerial photography (RGB) and LiDAR elevation measurements (DSMs). Techniques for converting data from conventional GIS vector and raster formats to a format suitable for machine learning purposes are discussed. The U-Net model architecture is used in order to train several model variants on a labeled building footprint dataset covering the Norwegian municipality of Trondheim. A model using aerial photography in combination with LiDAR elevation data achieved the best mean IoU test score.

Contents

Introduction **1**

1 Image Segmentation — Central Concepts and Existing Work **3**

1.1 Problem Description . . . 3

1.2 Convolutional Neural Networks (CNNs) . . . 4

Convolution . . . 5

Activation functions . . . 7

Pooling . . . 8

Batch normalization . . . 8

Dropout . . . 9

1.3 Metrics, Losses, and Optimization . . . 10

Accuracy, sensitivity, and specificity . . . 10

Intersection over union and dice coefficient . . . 11

Binary cross-entropy and soft losses . . . 13

Optimization . . . 14

1.4 State-of-the-Art . . . 15

1.5 Model Architecture . . . 17

2 Data **19**

2.1 Coordinate Systems . . . 19

2.2 Data types . . . 20

Vector data . . . 20

Raster data . . . 21

2.3 Data sets . . . 22

Raster data sets . . . 22

Vector data sets . . . 24

2.4 Tiling Algorithm . . . 24

2.5 Masking Algorithm . . . 27

2.6 Raster Normalization . . . 29

RGB rasters . . . 29

LiDAR rasters . . . 30

3 Experiments **34**

3.1 Experimental Setup . . . 34

Training procedure . . . 34
Software . . . 34
Hardware and performance . . . 35
3.2 Features . . . 35
RGB data . . . 35
LiDAR data . . . 42
Combined data . . . 49
3.3 Regularization Techniques . . . 52
Batch normalization . . . 52
Dropout . . . 53
Data Augmentation . . . 54
3.4 LiDAR Normalization . . . 55
3.5 Losses . . . 57

Conclusion and Further Work **63**

Bibliography **65**

Appendices **69**

A GIS preprocessing **69**
A.1 Overview . . . 69
A.2 Mapping between coordinate systems . . . 70
A.3 Zero-buffering vector datasets . . . 70
A.4 Merging raster datasets . . . 70

Introduction

Remote sensing is the process of gathering information about an object without making physical contact, one such technology being *aerial photography*. Although aerial RGB photography is intuitively interpretable for humans, it is fundamentally two-dimensional. *LiDAR*, another remote sensing technology, is able to measure distances to object surfaces by directing a beam of light and measuring the time of arrival and wavelength of the ensuing reflection. The resulting data can therefore be used to construct a three-dimensional spatial representation of the object of interest. LiDAR has been applied in a wide array of fields such as meteorology [42], forestry analysis [10], urban flood modelling [43], and autonomous driving systems [25].

One of the applications of LiDAR technology is the construction of *digital surface models* (DSMs). DSMs are grayscale images representing the earth's surface including all above-surface objects such as natural canopy and human-made objects. In contrast, *digital terrain models* (DTMs) represent the elevation of the *bare* ground where all above-surface objects have been artificially removed. While DTMs are often used in geographic and cartographic applications, DSMs can be used for localization and classification of objects above ground.

LiDAR data and aerial photography is usually provided by the respective cadastral authority in a given country. Cadastral authorities are also responsible for keeping records of cadastral data such as cadastral plots, roads, and buildings. The exact type and quality of this data varies substantially between countries and sometimes even between administrative regions in the same country. This raises the question: "Can high-fidelity insights be inferred from otherwise low-fidelity geographic data?". Figure 1 shows an outline of the possible "data enhancements" which are of interest within this domain.

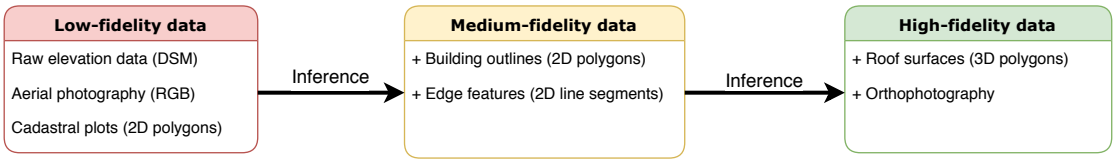


Figure 1: Classification of geographic data quality. The classifications reflect a general observed trend in data sets, and a given region may therefore not fit into exactly one of these categories. Some of the data types mentioned here will be described in Section 2.

The *Norwegian Mapping and Cadastre Authority* (*Statens Kartverk*) provides geographic data of uniquely high quality for the entirety of Norway. This offers an opportunity to train supervised machine learning models on lower fidelity data in order to infer higher fidelity features. Such models can then be applied in other regions where only low-fidelity data is available as a method of data enhancement.

Research questions

A *building outline* is a two-dimensional representation of building "footprint". Such data can be used for map annotations, flood risk analysis, and population density estimates, amongst other applications. The identification of building outlines from remote sensing data can be formulated as a *semantic segmentation task*, a heavily researched topic which has had many advances in the last decade. Geographic data, such as building outlines, are formatted in an unsuitable way for direct machine learning, and must therefore be purposefully transformed and pre-processed. The

development of a data pipeline for geographic data is the first topic of research in this thesis.

RQ1 How can geographic data representation be transformed into a format suitable for machine learning?

After having developed such a pipeline, the focus will be to develop a segmentation model for identifying building footprints with data from this pipeline. The use of aerial photography and LiDAR data from the Norwegian municipality of Trondheim will be investigated, as well as the combination of these two data sources.

RQ2 How can aerial photography and/or LiDAR data be used to predict accurate building outlines?

These research questions will lay down the ground work for my upcoming master's thesis where I will investigate the possibility of inferring roof surfaces represented as three-dimensional polygons from remote sensing data. Three-dimensional representations of buildings can for example be used for urban planning purposes. Another application, which incidentally prompted this research question in the first place, is the use of roof surface geometries to estimate the potential energy production of roof-mounted solar panel installations.

Thesis disposition

We will start by providing an overview of the problem domain of image segmentation and the methods currently being applied in the field in Section 1. An introduction to the world of *Geographic Information Systems* (GIS); the field which concerns itself with representing geographic data, will follow in Section 2. We will also describe how to preprocess such geographic data in order to train accurate machine learning models. The training procedure and experimental results will be presented and discussed in Section 3.

1 Image Segmentation — Central Concepts and Existing Work

The field of *computer vision* got started in the early 1970s [55, p. 10]. Computer vision differs from the classical discipline of *digital image processing* by concerning itself with the three-dimensional reconstruction of a scene from two-dimensional data [55, p. 10]. Most of the early research in the field revolved around manually designed feature extraction and processing techniques, but statistical techniques started to become popular in the 1990s [55, p. 15]. The statistical approach eventually morphed into the field of *machine learning*, where most of the research advances are made today [55, p. 17].

We will start by describing the particular image recognition problem of interest, namely *semantic segmentation*. *Convolutional neural networks* (CNNs) have been applied to image segmentation problems with great success [38, p. 1], and Section 1.2 provides a theoretic overview of the elementary building blocks used to construct modern CNN architectures. Section 1.3 will summarize the metrics used for evaluating the quality of image segmentation predictions. State-of-the-art CNN architectures for image segmentation will be listed in Section 1.4.

1.1 Problem Description

Image recognition seeks to answer three questions for any given image [38]:

1. **Identification:** Does the image contain any object of interest?
2. **Localization:** Where in the image are the objects situated?
3. **Classification:** To which categories do the objects belong to?

We will concern ourselves with only one object category (class) at any time, that class being building footprints, and will simplify the following theory accordingly with this simplification in mind. The localization and classification of objects in a given image can be performed at different granularity levels, as shown in Figure 2.

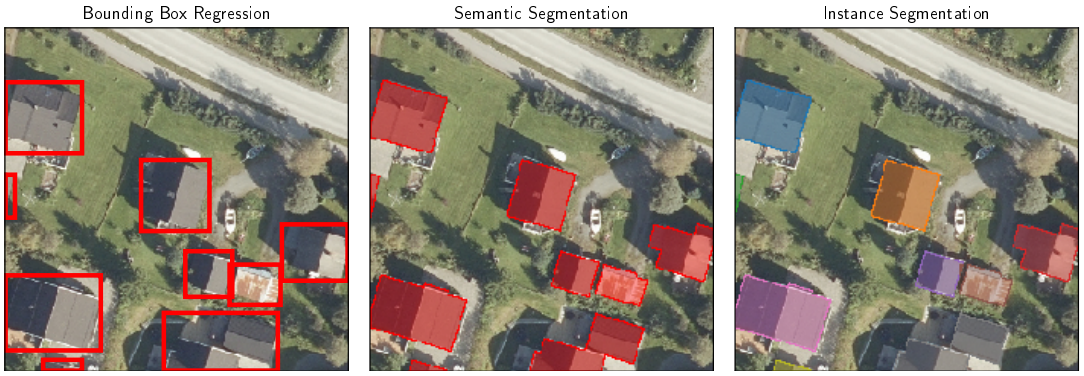


Figure 2: Different granularities for single-class building localization, using the Trondheim 2017 data set. Bounding box regression is shown on the left, semantic segmentation in the middle, and instance segmentation on the right.

Bounding box regression concerns itself with finding the smallest possible rectangles which envelopes the objects of interest. The sides of the rectangles may either by oriented parallel to the

axis directions, or rotated in order to attain the smallest possible envelope. The bounding box will therefore necessarily contain pixels that are not part of the object itself whenever the object shape is not perfectly rectangular.

Semantic segmentation rectifies this issue by classifying each pixel in the image independently, i.e. *pixel-wise* classification, producing a so-called classification *mask*. *Instance segmentation* distinguishes between pixels belonging to different objects of the same class, while *semantic segmentation* does not make this distinction. Since a bounding box can be directly derived from a semantic segmentation mask, and a semantic segmentation mask can be directly derived from instance segmentation mask; the problem complexity of these tasks are as follows:

$$\text{Bounding box regression} < \text{Semantic segmentation} < \text{Instance segmentation}$$

An image of width W and height H consisting of C channels is represented by a $W \times H \times C$ tensor, $X \in \mathbb{R}^{W \times H \times C}$. This is somewhat simplified, but we will give a more nuanced description in Section 2.2. Single-class semantic segmentation can therefore be formalized as constructing a binary predictor $\tilde{f}$ of the form:

$$\tilde{f} : \mathbb{R}^{W \times H \times C} \to \mathbb{B}^{W \times H}, \qquad \mathbb{B} := \{0, 1\}.$$

Where $\mathbb{B}^{W \times H}$ denotes a boolean matrix, 1 indicating that the pixel is part of the object class of interest, and 0 indicates the opposite. In practice, however, statistical models will often predict a pixel-wise class *confidence* in the continuous domain $[0, 1]$,

$$\hat{f} : \mathbb{R}^{W \times H \times C} \to [0, 1]^{W \times H},$$

but a binary predictor can be easily constructed by choosing a suitable threshold, T, for which to distinguish positive predictions from negative ones

$$\tilde{f}(X) = \hat{f}(X) > T, \qquad X \in \mathbb{R}^{W \times H \times C}.$$

The choice of the threshold value T will affect the resulting *sensitivity* and *specificity* metrics of the model predictions, metrics which will be explained in the upcoming Section 1.3.

1.2 Convolutional Neural Networks (CNNs)

There exists countless variations of the CNN model architecture, but there are still some elementary building blocks which they often have in common. We will start by sketching generic big picture of CNNs before going into detail about each modular building block. Figure 3 illustrates the architecture of *U-Net*, and we will use this figure to illustrate the common concepts of segmentation CNNs without considering the unique properties of U-Net specifically.

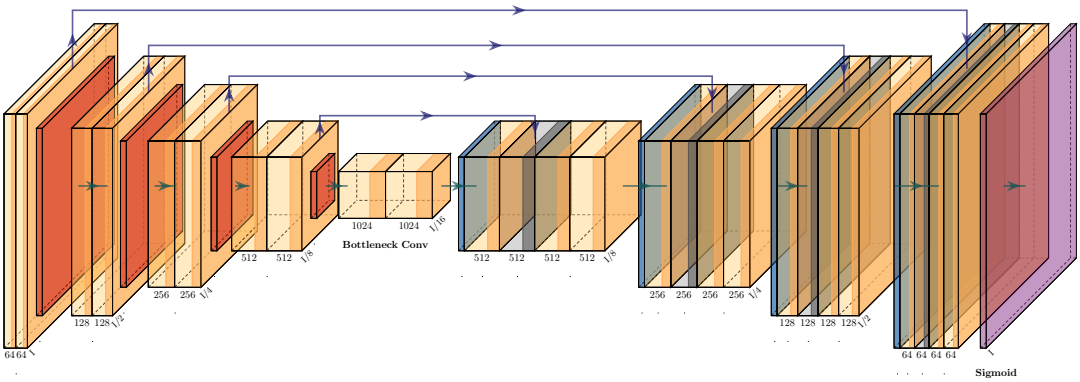


Figure 3: Illustration of the U-Net architecture for single-class segmentation, a typical example of an *encoder/decoder* structure. Convolution layers are shown in orange, and max pooling layers in red. Arrows indicates how data is forwarded through the network, top arrows being *skip connections*. The right hand side shows the upscaling performed by *transposed convolution* until the original resolution is restored and segmentation predictions can be formed with the *sigmoid* activation function (shown in purple). Figure has been generated by modifying a `tikz` example provided in the MIT licenced `PlotNeuralNet` library available at this URL: `https://github.com/HarisIqbal88/PlotNeuralNet`.

A CNN consists of several layered blocks operating over identical input dimensions within each block. These blocks are shown as contiguous boxes in Figure 3. The first layer in each block is a *convolutional layer*, which is a type of trainable feature extraction where several filtered *feature maps* are constructed. Each feature map is passed through a nonlinear *activation function* and the *activations* are subsequently downsampled in order to reduce the resolution. The downsampling is performed by a *pooling layer* and the output is forwarded to the next block. The number of feature maps that are extracted from the previous pooled activations increases as the resolution is decreased, and the right half of the architecture is eventually responsible for upsampling the resolution back to the original resolution by the means of *deconvolution*. The upsampling half of this network is not common to all CNNs, as CNNs tasked with bounding box regression and classification are not required to restore the original resolution before prediction. The upcoming sections will describe these concepts in more detail.

Convolution

As the name implies, a central concept of convolutional neural networks is the *convolution operator*. Let the *kernel*, w, be a $H_k \times W_k$ real matrix, and denote the activation of the previous layer at position (x, y) as $a_{x,y}$. The *convolution operator*, $\circledast$, is then defined as

$$w \circledast a_{x,y} = \sum_i \sum_j w_{i,j}\ a_{x-i,y-j}, \qquad a_{x,y} \in \mathbb{R},\ w \in \mathbb{R}^{H_k \times W_k},$$

where (i, j) spans the index set of the kernel. The region around $a_{x,y}$ which is involved in the convolution is referred to as the *receptive field*. We can generate a *filtered image* by moving this receptive field over the entire input image. The step size used when moving the receptive

field is referred to as the *stride size* of the convolution. Such a *moving convolution* is illustrated in Figure 4.

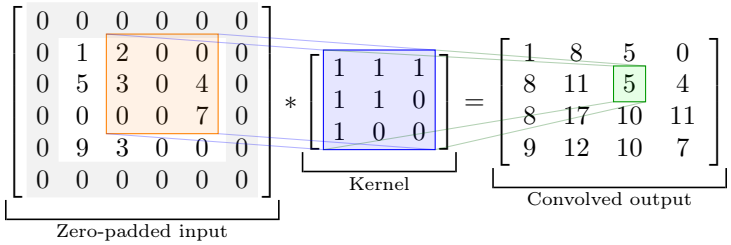


Figure 4: Visualization of a kernel convolution with a 3×3 kernel over an image of size 4×4 with additional zero-padding and stride size of 1×1. The *receptive field* is shown in orange, the respective kernel weights in blue, and the resulting convolution output in green. The zero padding of the input image is shown in gray.

In the case of input images or activations comprised of more than one channel, independent two-dimensional kernels are constructed for each channel and the convolved outputs are finally summed in order to attain a single feature map. The concept of a *kernel* predates neural networks as it has been used for feature extraction in the field of image processing for many years [55, p. 11]. The kernel weights determine the type of features being extracted from the given input image, some common interpretable kernels are given below.

$$\underbrace{w_1 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}}_{\text{Identity kernel}}, \quad \underbrace{w_2 = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}}_{\text{Edge detection kernel}}, \quad \underbrace{w_3 = \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}}_{\text{Normalized box blur kernel}}, \quad \underbrace{w_4 = \frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}}_{\text{Gaussian blur kernel}}.$$

It is important to notice that kernel convolution has the additional effect of reducing the dimensionality of the input image. Firstly, pixels along the image border are partially ignored since the receptive field can not be properly centered on these pixel. Secondly, a horizontal stride of $W_k > 1$ or a vertical stride of $H_k > 1$ will cause additional dimensional reduction. For an image of size $H \times W$ and a kernel of size $H_k \times W_k$, the input image is reduced to size

$$\left\lfloor (H - H_k + H_s)/H_s \right\rfloor \times \left\lfloor (W - W_k + W_s)/W_s \right\rfloor .$$

as shown by [57]. The reduction in dimensionality when using stride sizes of one is often undesirable, and for this reason it is common to add a *padding* filled with zero-values along the edges of the input image. Applying a padding of height H_p at the horizontal borders and a padding of width W_p at the vertical borders results in a feature map of size

$$\left\lfloor (H - H_k + H_s + \mathbf{H_p})/H_s \right\rfloor \times \left\lfloor (W - W_k + W_s + \mathbf{W_p})/W_s \right\rfloor .$$

If we assume the input height and width to be divisible by the stride height and width respectively, we can set $H_p = H_k - 1$ and $W_p = W_k - 1$ in order to attain an output shape of $(H/H_s) \times (W/W_s)$ [57]. Such a padding is shown in gray in Figure 4.

CNNs apply multiple different convolutions to the same input, resulting in a set of differently filtered outputs. After having applied the layer's activation function to the output (see upcoming section about "activation functions") and the activations have been downsampled (see upcoming "pooling" section), the filtered outputs are passed onto the next layer. The number of filters are usually increased as you move deeper into the network where the resolution has been increasingly downsampled. Unlike classical image processing, where kernel weights are carefully selected in order to construct an intended type of feature extraction, CNNs let each kernel weight be a trainable parameter. As the network is trained each kernel learns to extract features which are of use for the subsequent layers.

An important aspect of convolution is that the kernel weights remain unchanged as the receptive field is moved over the input image. This *parameter sharing* results in regions being treated identically no matter where in the image they are situated [18]. The sharing of parameters has the benefit of reducing the parametric complexity of the network, thus decreasing the computational cost of training it. Finally, compared to a more classical *fully connected feedforward network*, which operates over flattened vectors, a fully convolutional neural network operates over images in matrix form, thus taking the spatial relationship between pixels into account.

Activation functions

So far we have only explained how a convolutional neural network consists of a set of parametrized linear operations. Such a network, if left unaltered, is therefore restricted to only approximating linear functions. The solution to this predicament is to introduce the concept of an *activation function*, a nonlinear function applied to the output from the convolutional layers. These activation functions were originally inspired by the neuroscientific understanding of biological neurons [19, p. 165], but have since been shown to be a theoretical prerequisite of the *universal approximation* property of artificial neural networks [8, 37]. The *logistic sigmoid* function, with its deep roots in probability theory, has been a popular choice of activation function for neural networks since the inception of the field [46], and is defined by

$$\sigma(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{e^x+1}. \qquad \text{(Sigmoid activation function)}$$

Observe that $\lim_{x\to-\infty}\sigma(x) = 0$ and $\lim_{x\to+\infty}\sigma(x) = 1$, and that its derivative is positive over the entire real number line. This makes it a bounded, differentiable, monotonic function, and is therefore suitable for mapping the weighted output of an artificial neuron in the domain $(-\infty,\infty)$ into the range $(0,1)$. This makes it especially suitable for the final layer in neural networks intended for predicting binary 0/1-responses.

Although the sigmoid activation function has strong biological [46] and theoretical [8] underpinnings, it often suffers from the phenomenon of *vanishing gradients* for network architectures consisting of three or more layers, which in turn severely inhibits training. As an alternative to the sigmoid activation function, the *rectified linear unit* (ReLU) was introduced in a paper [20] by Hahnloser et al. in year 2000. It is defined as

$$\mathrm{ReLU}(x) = x^+ = \max(0,x). \qquad \text{(ReLU activation function)}$$

The ReLU activation function has become the dominant activation function for use in neural networks in recent years [36, p. 438] as it has been empirically shown to adapt well to deeper neural networks [17].

Pooling

The last layer in a given CNN block is conventionally a *downsampling* operation, most often referred to as a *pooling* layer. As with convolution, this operation has biological influences as it is inspired by a model of the mammalian visual cortex [18, p. 966]. The reduction in spatial resolution is considered to be one of the main reasons for why CNNs portray a high degree of translational and rotational invariance [31]. As with moving convolution, pooling is implemented by moving a receptive field of size greater than 1, typically 2×2, over the activations and mapping these values into a lower dimensional space. There are several different ways to define such a mapping, the two most common being *max pooling* and *average pooling*, which respectively retrieve the maximum value and average value from the receptive field. The former is exemplified in Figure 5.

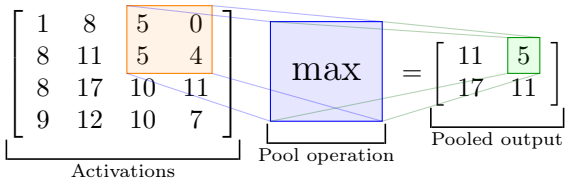


Figure 5: Example of a *max-pooling* operation with a receptive field of size 2×2 and an identical stride size. The receptive field is shown in orange and the respective pooled output is shown in green.

As can be seen in Figure 5, using a receptive field and stride of size 2×2 will yield a downsampled image with one quarter as many pixels as the original input.

Batch normalization

The reparametrization of earlier layers when training deep neural networks results in a distributional change in the feature layer forwarded to the next layers. This forces all subsequent layers to adapt to the new "distributional circumstances", which in turn impedes the convergence of the optimization. This phenomenon, referred to as *internal covariate shift*, was first identified in a paper [28] by Ioffe and Szegedy (2015) where they propose a method called *batch normalization* in order to counter this phenomenon. Suppose we have a layer activation $\boldsymbol{a}$ consisting of d dimensions, i.e. $\boldsymbol{a} = (a^{(1)}, \ldots, a^{(d)})$. First we standardize each feature dimension, k, independently

$$\widehat{a}^{(k)} = \frac{a^{(k)} - \mathrm{E}\left[a^{(k)}\right]}{\sqrt{\mathrm{Var}\left(a^{(k)}\right) + \epsilon}}, \qquad \text{(Batch standardization)}$$

where $\mathrm{E}[\cdot]$ and $\mathrm{Var}(\cdot)$ are respectively sample means and sample variances over the current mini-batch, and ϵ is added for numerical stability. The result of this standardization is a feature map where all filters have mean 0 and variance 1 for every mini-batch. The internal covariate shift has been practically eliminated as a result.

This type of normalization alone may not be optimal in all cases, though, and is best explained

by constructing a somewhat contrived pathological example. Assume a set of pooled layer activations $\boldsymbol{a}$ to be symmetrically distributed, and assume the subsequent convolution layer to preserve this symmetry. After standardizing the output, 50% of the values are expected to be negative, and all of these values will be truncated to 0 if ReLU is the activation function of choice. This informational loss may be suboptimal for the given network layer and must be accounted for. That is to say, $\mathrm{E}[a] = 0$ and $\mathrm{Var}(a)$ may be an unsuitable domain for the given activation function. For this reason, we introduce two additional trainable parameters for each feature dimension, $\gamma^{(k)}$ and $\beta^{(k)}$, and apply a second normalization step

$$y^{(k)} = \gamma^{(k)}\widehat{a}^{(k)} + \beta^{(k)}. \qquad \text{(Trainable normalization)}$$

The intent is to learn the values for the shift, $\beta^{(k)}$, and scaler, $\gamma^{(k)}$, which restores the representative power of the given layer *after* the batch standardization.

Dropout

Dropout is a regularization technique for neural networks intended to prevent "complex co-adaption of feature detectors" [26]. In practice this is achieved by randomly omitting hidden nodes from the neural network during each training step; effectively forcing hidden nodes to become less interdependent. An alternative interpretation of the dropout procedure is that it is a computationally efficient form of model averaging, each dropout permutation being a model instance. This technique has been empirically shown to significantly increase the test performance in several different settings.

Although originally intended for use in feedforward neural networks, dropout has been extensively applied in CNN architectures as well [34]. Since there are no nodes to be omitted in fully convolutional layers, the dropout procedure needs to be adapted in order to be applicable in a CNN setting. One approach is to introduce a randomly located square mask (*cutout*) in the input image [9]. *Stochastic depth dropout* randomly selects entire layers to be dropped, replacing them with identity functions instead [27]. Dropout can also be integrated into max pooling layers, ignoring values at random during the search for the maximum value in the receptive field [56]. This has become known as *max-pooling dropout* and is illustrated in Figure 6.

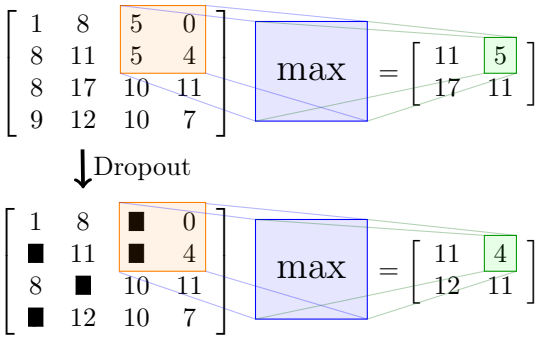


Figure 6: An example application of *max-pooling dropout* using a receptive field and stride of size 2×2. A dropout probability of $p = 0.25$ has been used. Dropped values are shown as black boxes.

1.3 Metrics, Losses, and Optimization

Metrics and losses are of central importance when training and evaluating machine learning models. Denote the parametrization of a given machine learning model, $\hat{f}$, as $\boldsymbol{\theta}$, the input features as X, and the corresponding *ground truth* labels as Y. In order to evaluate the performance of a given model parametrization, we must formulate a cost- or performance-*metric*, $P(\hat{f}(X;\theta);Y)$, which we intend to respectively minimize or maximize. The performance metric encodes our notion of what constitutes as a good model fit.

While the performance metric is what we really want to optimize, it may not be suitable for numerical optimization, for example due to being non-differentiable or too computationally costly. Machine learning optimization differs from classical optimization in that the performance metric is indirectly maximized through the optimization of a surrogate *loss* function [19, p. 272], $\mathcal{L}(\hat{f}(X;\boldsymbol{\theta});Y)$. The loss metric is minimized in the hope of improving the performance metric indirectly, and it is therefore of vital importance that there is a strong relationship between performing well on the loss function and performing well on the performance metric.

We will provide a summary of popular losses and metrics for single-class semantic segmentation.

Accuracy, sensitivity, and specificity

In order to describe segmentation metrics, it is useful to define the following quantities:

Condition Positive (P): Number of object class pixels in ground truth mask.

Condition Negative (N): Number of non-object class pixels in ground truth mask.

True Positive (TP): Number of pixels correctly predicted as being part of object class (correctly identified).

True Negative (TN): Number of pixels correctly predicted as *not* being part of object class (correctly rejected).

False Positive (FP): Number of pixel incorrectly predicted as being part of object class (incorrectly identified).

False Negative (FN): Number of pixel incorrectly predicted as *not* being part of object class (incorrectly rejected).

False positives (FP) are often knows as *type I errors* in statistics, and false negatives (FN) as *Type II errors*. The greater the values of TP and TN, the better, and the smaller the values of FP and FN, the better. A visual representation of these classifications is given in Figure 7.

The simplest metric for semantic segmentation is the *pixel accuracy* metric. This metric simply reports the percentage of pixels that were correctly classified. More formally, it can be defined as:

$$\textbf{accuracy} = \frac{TP+TN}{TP+TN+FP+FN} = \frac{TP+TN}{P+N}$$

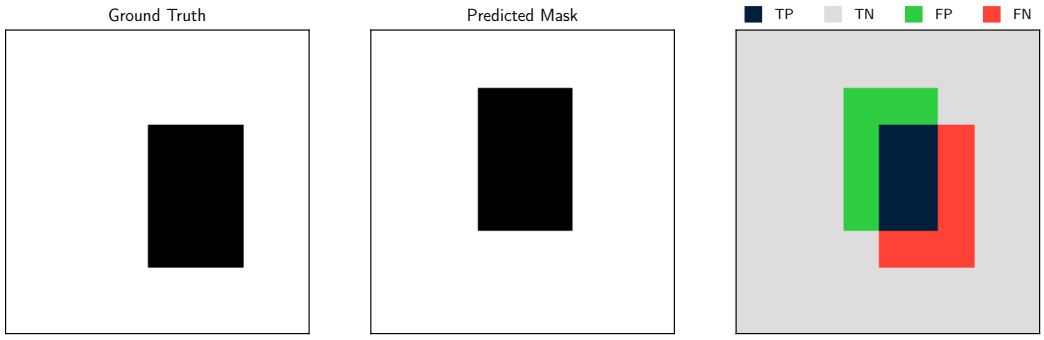


Figure 7: Binary segmentation problem of size 256×256. The ground truth, a rectangle of size 120×80 is shown on the left. The "predicted" mask, shown in the middle, is of the same size, but offset by $(-30, -30)$. The right figure shows the visual equivalent of a confusion matrix. True positives are shown in dark blue, true negatives in light gray, false positives in green, and false negatives in red.

The problem with the pixel-wise accuracy metric is that it does not take class imbalances into account. Consider a problem where 95% of all pixels are considered to be of class 0, and the remaining 5% of class 1. If we construct a model which predicts 0 regardless of the feature inputs provided to the model, the model will achieve a 95% accuracy score. This makes pixel-wise accuracy scores hard to interpret when you do not know the class balance of the respective dataset and the accuracy grouped by class. This is why it is often replaced by other metrics which take imbalances into account. A pair of such metrics are *sensitivity* and *specificity*, formally defined as:

$$\textbf{sensitivity} = \frac{\text{number of true positives}}{\text{number of true positives} + \text{number of false negatives}} = \frac{TP}{TP + FN} = \frac{TP}{P}$$
$$\textbf{specificity} = \frac{\text{number of true negatives}}{\text{number of true negatives} + \text{number of false positives}} = \frac{TN}{TN + FP} = \frac{TN}{N}$$

The *sensitivity* is therefore a measure of how good a given model prediction is able to identify positives as a relative, fractional value. Likewise, the *specificity* is a measure of how good a given model prediction is able to identify negatives.

Intersection over union and dice coefficient

Although the sensitivity and specificity metrics address the issue of class imbalances, they are still *two* distinct metrics that need to be simultaneously inspected in order to get a full overview of the model performance. Is it possible to construct a *single* scalar metric which incorporates the ideas of both sensitivity and specificity? The *intersection over union* (IoU) and *dice coefficient* (F_1) are two metrics which try to do exactly this.

The IoU metric, also known as the *Jaccard index*, is defined as the area of the intersection between the predicted segmentation mask and the ground truth mask divided by the union of these two

masks, or more formally,

$$\text{IoU} = \frac{|\text{prediction} \cap \text{truth}|}{|\text{prediction} \cup \text{truth}|} = \frac{\text{TP}}{\text{TP} + \text{FP} + \text{FN}}.$$

In the case of multiple classes IoU is calculated for each class independently and the result is averaged, known as *mean intersection over union* (MIoU). MIoU is the most commonly used segmentation metric in research and competitions due to its simplicity and representativeness [11]. Notice how the IoU metric is bounded between 0 and 1; IoU = 0 represents a complete "predictive miss", while IoU = 1 represents a prediction in perfect accordance with the ground truth. A visualization of this metric is given in Figure 8 below.

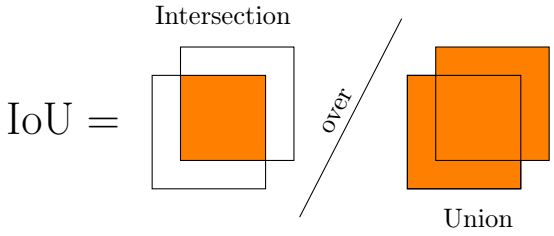


Figure 8: Visualization of single-class IoU metric.

An alternative metric is the dice coefficient, also known as the F_1 *score*. The dice coefficient is defined by taking twice the area of the intersection and dividing by the sum of the areas of the two masks:

$$\text{F}_1 = \frac{2 \cdot |\text{prediction} \cap \text{truth}|}{|\text{prediction}| + |\text{truth}|} = \frac{2 \cdot \text{TP}}{2 \cdot \text{TP} + \text{FP} + \text{FN}}.$$

Again we observe that this metric is bounded to the interval $[0, 1]$, with the same interpretation of the endpoints 0 and 1 as with the IoU metric. The visual representation of this metric is given in Figure 9.

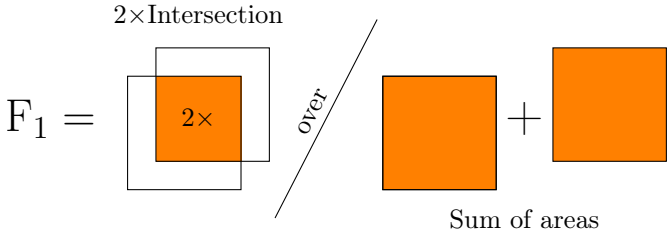


Figure 9: Visualization of the single-class dice coefficient metric, also known as the F_1 score.

You may have noticed that these two metrics are quite similar; they involve the same quantities, only weighted differently, and map into the same interval. In fact, we can construct an exact relationship between these two metrics[1]

$$\frac{\text{IoU}}{\text{F}_1} = \frac{1}{2} + \frac{\text{IoU}}{2}.$$

[1]The following relationship and the ensuing inequality bounds were noted by the *Cross Validated Stack Exchange* user "Willem" here: `https://stats.stackexchange.com/a/276144`.

By inspection the two metrics must always be positively correlated, that is, as one metric increases or decreases, the other must follow suit. A useful insight for understanding how these two metrics actually differ is to observe how the IoU metric is bounded by the dice coefficient:

$$\frac{\mathrm{F}_1}{2} \leq \mathrm{IoU} \leq \mathrm{F}_1.$$

The IoU is *always* less than or equal to the dice coefficient, but never smaller than half the value. The fraction $\mathrm{IoU}/\mathrm{F}_1$ is equal to 1 whenever the prediction coincides with the ground truth and is equal to $1/2$ whenever there is no overlap at all. By drawing an analogy to the $p = 1$ (absolute/Manhattan) norm and $p = 2$ (Euclidean) norm, we can say that the IoU metric weighs the worst case of a prediction more than the average case, and vice versa for the F_1 metric.

Binary cross-entropy and soft losses

So far we have only discussed metrics which are discrete, non-differentiable functions, thus making them unsuitable for direct optimization. As discussed earlier, we need to introduce a differentiable surrogate loss function which can be optimized. The key "trick" is to define a loss function over the continuous probability domain before it is discretized to the classification domain by thresholding. In order to formulate proper loss functions, we will start by establishing some notation. Denote the ground truth binary classification mask as $Y \in \mathbb{B}^{H \times W}$ and the corresponding features as $X \in \mathbb{R}^{H \times W \times C}$. Assume a model $\hat{f}$ parametrized according to $\boldsymbol{\theta}$ which provides a probability estimate for Y, the probability estimate denoted as $P = \hat{f}(X; \boldsymbol{\theta}) \in [0, 1]^{H \times W}$. For notational convenience we will use a linear index in order to denote single matrix elements, for instance $P_i \in [0, 1]$ for $i = 1, \ldots, HW$.

The most common loss function for binary classification tasks is the *binary cross entropy* (BCE) loss function defined as

$$\mathcal{L}_{\mathrm{BCE}}(P; Y) = -\sum_{i=1}^{HW} Y_i \log(P_i) + (1 - Y_i) \log(1 - P_i). \tag{1}$$

However, there are several issues with using the BCE as the loss function for segmentation tasks. Firstly, it does not take class imbalances into account. The *weighted binary cross entropy* (wBCE) is one attempt at accounting for class imbalances, but weighting is highly task-dependent and has been shown to have negligible performance improvement over BCE [4, p. 98]. Another issue with BCE and wBCE is that they are poor surrogates for the segmentation metrics introduced in the previous subsection. The solution is to introduce differentiable approximations of these discrete segmentation metrics. Such an approximation for the IoU metric is the *soft Jaccard loss* also known as the *Jaccard distance* [7], defined by

$$\mathcal{L}_{\mathrm{SJL}}(P; Y) = 1 - \frac{\sum_{i=1}^{HW} P_i Y_i}{\sum_{i=1}^{HW} (P_i + Y_i - P_i Y_i)} \approx 1 - \mathrm{IoU} \tag{2}$$

Notice that if P_i is restricted to only take values in $\{0, 1\}$ then $\mathcal{L}_{\mathrm{SJL}}$ becomes equal to $1 - \mathrm{IoU}$. Other variants exists, and it is also common to add a smoothing factor by adding a value δ to both the numerator and denominator and multiplying the entire loss with the same value. A similar differentiable approximation of the dice coefficient, called *soft dice loss*, has also been derived [41].

$$\mathcal{L}_{\text{SDL}}(P;Y) = \frac{2\sum_{i=1}^{HW} P_i Y_i}{\sum_{i=1}^{HW} P_i^2 + \sum_{i=1}^{HW} Y_i^2} \approx 1 - F_1 \tag{3}$$

Optimizing these two metric-sensitive losses have been shown theoretically and empirically to indirectly maximize their respective surrogate metrics [4]. You would think that if the dice coefficient has been chosen as the metric of interest for a given problem, the soft dice loss should be used instead of soft Jaccard loss. However, Bertels et al. have shown [4] that these two metric-sensitive losses are equally good surrogates for each others metrics, and the choice is therefore mainly a preferential one.

Optimization

So far we have only mentioned that we must define a loss function to be optimized for a given neural network, but we have not mentioned exactly how this optimization is performed. Deep learning optimization is a huge field of research, and for the sake of brevity we will glance over a lot of detail and focus on the techniques applied in our models presented in Section 3.

For *supervised* machine learning we start with a labeled data set, $\mathcal{D}$, containing n observations:

$$\mathcal{D} = \{(X_1, Y_1), (X_2, Y_2), \ldots, (X_n, Y_n)\}$$

This data set is then partitioned into three disjunctive subsets, referred to as the *training*, *validation*, and *test* splits.

$$\mathcal{D}_{\text{train}} \cup \mathcal{D}_{\text{validation}} \cup \mathcal{D}_{\text{test}} = \mathcal{D}$$
$$\mathcal{D}_i \cap \mathcal{D}_j = \emptyset, \quad \text{for } i \neq j$$

A common method is to shuffle the data and then allocate 70% of the original data as training data and 15% for the two remaining splits. Now, as the name implies, the training split $\mathcal{D}_{\text{train}}$ is used for training the neural network, one of the simplest optimization algorithms being iterated *gradient descent*. At training step s the network parametrization $\boldsymbol{\theta}$ is updated according to

$$\boldsymbol{\theta}^{(s+1)} = \boldsymbol{\theta}^{(s)} - \alpha \nabla_\theta \sum_{X_i, Y_i \in \mathcal{D}_{\text{train}}} \mathcal{L}(\hat{f}(X_i; \boldsymbol{\theta}^{(s)}); Y_i),$$

where α is the learning rate. Efficient calculation of the gradient $\nabla_{\boldsymbol{\theta}}$ for nonlinear networks of arbitrary connectivity is enabled by a method called *backpropagation* [49]. For each training step, the input data is passed forward through the neural network in order to calculate new predictions. Errors are then subsequently propagated backwards through the network in order to efficiently calculate the partial derivatives of the loss function with respect to each parameter θ_i. The initial parametrization, $\boldsymbol{\theta}^{(0)}$, is not simply filled with zeroes or random values as that can cause certain problems. We will initialize the weights with the *He normal* method devised by He et al. which draws values from a truncated normal distribution. More details can be found in the original paper [23].

A common modification to this scheme is the so-called *mini-batch gradient descent* algorithm [48]. The training set is yet again partitioned into even smaller splits called *batches*, and during training gradient descent is applied iteratively over each batch. After each *epoch*, when all batches have

been evaluated, the training split is shuffled and a new batch partition is formed. For sufficiently large batch sizes the feature distribution of each batch can be considered a good approximation of the entire sample space, while still having decreased the computational cost of each training step.

The validation split is used for hyperparameter tuning such as the selection of the number of training epochs by *early stopping* [5]. After each epoch, a given validation loss or metric is evaluated over the validation split. As soon as the validation split has not improved for a given number of epochs, referred to as the early stopping *patience*, the training is stopped and model parametrization corresponding to the best validation metric is chosen as the final model. Early stopping is intended to prevent overfitting the model on the training data by using the validation metric as an indication of generalizability. The test split, in contrast to the training and validation splits, is completely isolated from the model training procedure, and is solely kept for a final evaluation of the trained model.

The *Adam optimizer* published in 2015 [32] has become a popular gradient-based optimization algorithm for machine learning problems. The algorithm has relatively low computational and memory requirements, copes well with large data sets and parameter spaces, and is relatively well-behaved when faced with noisy and sparse gradients. This is the optimization algorithm we will use for our model training experiments, the results being presented in Section 3.

1.4 State-of-the-Art

At the time of this writing, CNNs have largely surpassed all previous methods for performing image segmentation [11], but it is still a relatively new field with constantly new improvements being made. In the following section we will provide an overview of the current state-of-the-art methods being applied within this field, focusing on the unique aspects of each approach. Four CNN architectures are considered especially influential as they have become essential building blocks for many segmentation architectures; *AlexNet*, *VGG-16*, *GoogLeNet*, and *ResNet* [11]. Note that these architectures were initially intended for *classification* and *localization* tasks only, but their conceptual ideas are important for *segmentation* architectures as well.

AlexNet [33] won several image classification competitions when it was first published in 2012, including the ILSVRC-2012 competition [11]. By employing five convolutional layers, max-pooling layers, ReLU activation functions, and dropout, followed up by a fully connected feed-forward classification network, it outperformed the 2nd place contender by a relatively large margin.

The **VGG-16** architecture [51] published in 2014 introduced the idea of stacking several convolution filters with small receptive fields in early layers. VGG-16 distinguishes itself by stacking several convolutional layers with small receptive fields in the first layers instead of using few convolutional layers with large receptive fields. The result is a network with fewer parameters and more applications of the non-linear activation functions leading to an increased ability to discriminate inputs and reduced training times. VGG-16 achieved an impressive 92.7% TOP-5 test accuracy in the ILSVRC-2013 classification competition, inspiring further research involving the techniques employed by the architecture [11].

Substantially deep networks are prone to overfitting and are subject to additional computational overhead. The **GoogLeNet** architecture [54] from 2014 introduced the *inception module* in order to combat this problem, a building block which allow networks to grow in depth and width with modest increases in computational overhead. The inception module discards the usual

approach of ordering convolutions in a sequential manner, instead opting for several parallel pooled convolution branches with different dimensional properties. Finally a 1×1 convolution is applied to each branch in order to reduce the dimensionality of the output and the concatenated result is passed onto the next layer.

The **ResNet** architecture [22] from 2016 was the result of a continued effort to make deeper architectures feasible. By training a model with 152 layers ResNet won the ILSVRC-2016 competition with a remarkable 96.4% accuracy [11]. This depth is achieved by introducing *skip connections* between layers, an effective way to combat *vanishing gradients*.

The success of convolutional architectures for classification tasks were eventually adapted for segmentation tasks as well. A fully convolutional, pixel-to-pixel classification network was first published by Long, Shelhamer, and Darrell in 2015 [39]. AlexNet, VGG-16, and GoogLeNet were successfully adapted in order to achieve state-of-the-art performance on the PASCAL-VOC segmentation dataset.

Fully convolutional neural networks (FCNN) quickly became the dominant technique used in segmentation challenges after the success in classification and localization challenges. The **U-Net** architecture, originally published in 2015 and intended for biomedical image segmentation, has become one of the more popular segmentation architectures. U-Net has an *encoder/decoder*-structure; the network starts with a contracting path where context is extracted from the input image. This is followed by a symmetric expanding path in order to upscale the segmentation to the original resolution by the use of *transposed convolution*, a trainable procedure also known as *deconvolution*. *Skip-connections* are introduced in order to forward information from the contracting layers to the respective expanding layers. **SegNet** [3], an architecture from the same time period, has a similar encoder/decoder structure as U-Net. The difference between the two architectures is that SegNet only copies over the max-pool indices in the skip connections instead of forwarding the entire feature layer, thus decreasing the memory requirements of the network.

R-CNN [16], and the subsequent improvements **Fast R-CNN** [15] and **Faster R-CNN** [44], made great strides in image classification and localization tasks in 2014 and 2015. The crux of their success lies in the *region proposal network* (RPN), a parallel network which is responsible for identifying *regions of interest* (RoIs) in the convolved feature maps. These RoIs are transformed to consistent dimensions by a custom pooling method called *RoIPool*, or alternatively *RoIWarp*, and subsequently classified and localized with a fully connected feedforward network. **Mask R-CNN** [24], published by the Facebook AI research group in 2017, sought to expand Faster R-CNN in order to predict segmentation as well. Mask R-CNN replaces RoIPool with *RoIAlign*, a region of interest pooling method which preserves a one-to-one pixel mapping between the original feature map and the extracted region of interest. The output of the pooling operation is forwarded to a parallel FCNN branch in order to perform pixel-wise segmentation. This segmentation branch predicts independent masks without inter-class competition, and reuses the work performed by the classification branch in order to select which mask to apply to a given region.

Capsule networks has become a topic of large interest in the research community as of late. First introduced in a paper [50] by Sabour, Frosst, and Hinton, it has since been applied to segmentation tasks as well. One such adaption is the **SegCaps** architecture [35]. The main idea behind capsule networks is to output more data from each neuron, effectively allowing the network to make more informed decisions with this new context. Instead of only storing a single scalar in each neuron they store a contextual vector instead. Each vector encodes information about the spatial orientation, magnitude, prevalence, and other attributes related to the extracted features. These *capsule vectors* are *dynamically routed* to the capsules in the next layer based on vector

similarity.

1.5 Model Architecture

We have chosen the U-Net architecture for segmenting building outlines and we will present numerical experiments in Section 3. The U-Net model has already been briefly described in the previous section and the architecture has been illustrated in Figure 3, but we will provide a more detailed summary of the U-Net architecture here. An alternative visual representation of the U-Net architecture is provided in Figure 10.

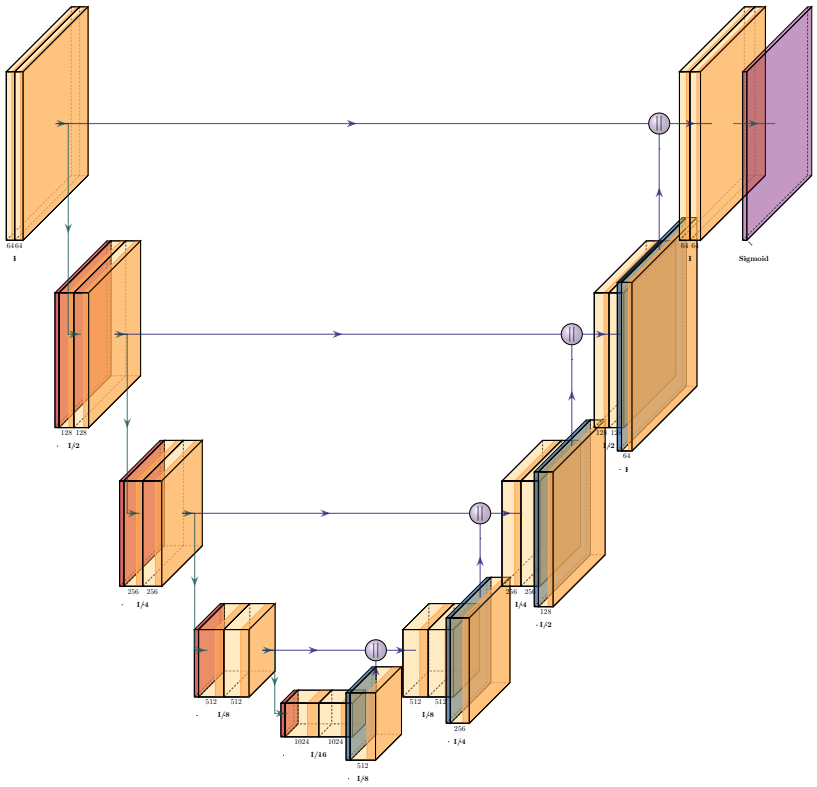


Figure 10: U-Net model architecture. The vertical axis denotes the resolution of the features, resulting in the U-shape of U-Net. Figure has been generated by modifying a `tikz` example provided in the MIT licenced `PlotNeuralNet` library available at this URL: `https://github.com/HarisIqbal88/PlotNeuralNet`.

As can be seen in Figure 10, the U-Net architecture consists of four sequential "encoder modules", each module applying a set number of convolutional filters followed by the application of the ReLU activation function. The number of trained convolutional filters in each encoder module is respectively: 64, 128, 256, and 512. Each module ends with a downsampling operation in form of max-pooling of size 2. Since our input images have resolution 256×256, we end up with inputs of size 16×16 to the "bottleneck convolution module" where 1024 convolutional filters are trained. The bottleneck convolution module is placed at the bottom of the U-shape in Figure 10. Each decoder block utilizes batch normalization and max-pooling dropout, although

models without these building blocks will be tested in Section 3.3. The "decoder modules" apply transposed convolutions in order to upsample the resolution by a factor of two, the number of filters being equivalent to their respective "mirror encoders", i.e. the encoder modules handling inputs with identical resolutions. Four such modules are applied in order to yield a final output resolution of size 256×256, the original input resolution. The outputs of the mirror encoder modules are concatenated to the input to the decoder modules in order to aid the upsampling procedure. Finally, a sigmoid convolution with filter size 1 is applied in order to produce the final segmentation probabilities. This model has been implemented using the declarative Keras API in Tensorflow v2.0 and the source code is available at the `JakobGM/project-thesis` repository on GitHub[2]. The final network has 7 025 329 trainable parameters.

[2]All source code used in order to produce and present the results in this paper are available from the following public GitHub repository: `https://github.com/JakobGM/project-thesis`. Specifically, the implementation of the U-Net architecture has been made available here: `https://github.com/JakobGM/project-thesis/blob/master/remsen/models.py`.

2 Data

Geographic data is in wide use by both the public and private sector, and is a huge subject in and of itself. The storage, processing, and inspection of such data is handled by *Geographic Information Systems* (GIS). In this section we will explain a few core GIS concepts relevant for the problem at hand, concepts which will inform decisions for how to prepare the data for machine learning purposes. Section 2.1 will give a brief introduction to the coordinate systems used to represent geographic data. GIS data can be largely bisected into two categories, vector data and raster data, and both types will be described in Section 2.2. Section 2.3 will present the data sets used for training our models. The remaining subsections will describe the preprocessing pipeline which has been developed for our specific purposes, preprocessing in the form of cadastral tiling (Section 2.4), segmentation masking (Section 2.5), and raster normalization (Section 2.6). A figurative overview of the preprocessing pipeline is provided by Figure 59 in Appendix A.1.

2.1 Coordinate Systems

One of the most common coordinate system for representing *arbitrary* positions on earth's surface is the *geographic coordinate system* (GPS). A given point, $\boldsymbol{p} = (\phi, \lambda, z)$, is represented by an angular latitude and longitude, ϕ and λ respectively, and a radial distance from the mean sea level, z. A negative value for z does not necessarily imply that the given point is below ground, as certain areas (such as in the Netherlands) are situated below sea level. It is therefore not sufficient to represent elevation data with unsigned floating point numbers.

Although GPS is able to uniquely represent arbitrary geographic points with a high degree of accuracy, it is still unsuitable for many applications. Cartesian transformations and distance norms are cumbersome to calculate, and data structures and visualizations which are fundamentally two dimensional in nature, such as maps, rasters, and matrices, are difficult to construct from spherical coordinates while protecting important properties of the data.

In order to solve this problem we define a set of coordinate system *projections* which approximate predefined regions of the earth's surface as flat planes. The resulting coordinate systems are Cartesian and thus allow us to represent geographic points in the more common $\boldsymbol{p} = (x, y, z)$ format. Cartesian distance norms such as $||\boldsymbol{p}_1 - \boldsymbol{p}_2||_2$ and Cartesian translations $\boldsymbol{p}_1 + \boldsymbol{p}_2$ stay within predefined error tolerances as long as operations are contained to the validity region of the given projection.

One such Cartesian approximation of the earth's surface is the Universal Transverse Mercator (UTM) coordinate system which divides the earth into 60 rectangular zones [52, p. 48]. The UTM zones covering Europe are shown in Figure 11. We will exclusively use UTM zone `32V` for our datasets covering the municipality of Trondheim situated in the southern part of Norway. Data provided in alternative coordinate systems will be mapped to this UTM zone before we start using the data. Since this is an affine coordinate system, we can easily generalize any models to other coordinate systems by applying the correct affine

Figure 11:
The figure shows the UTM zones required in order to cover the entirety of Europe, from `29S` to `38W`. This public domain image has been sourced from Wikimedia [6].

transformations. Technical details for how to map between different coordinate systems are given in Appendix A.2 for reproducibility.

2.2 Data types

We will provide a brief overview of the two main categories of GIS data, namely *vector data* and *raster data*, and how to prepare these data types for machine learning purposes.

Vector data

A *line string* is an ordered collection of geographic points $(\boldsymbol{p}_0, \ldots, \boldsymbol{p}_n)$ defining a path which connects each consecutive point by a straight line. The points are therefore necessarily order dependent. A *simple* line string is a path which does *not* intersect itself, while a *complex* line string is one that does. When the first and last points of a line string are identical it is considered a *linear ring*, i.e. $l = (\boldsymbol{p}_0, \ldots, \boldsymbol{p}_n, \boldsymbol{p}_0)$. A *polygon* can therefore be represented by a simple linear ring which defines its *exterior hull* and any number of simple linear strings which defines its *interior hulls*. Figure 12 illustrates these concepts for polygons with and without interior hulls.

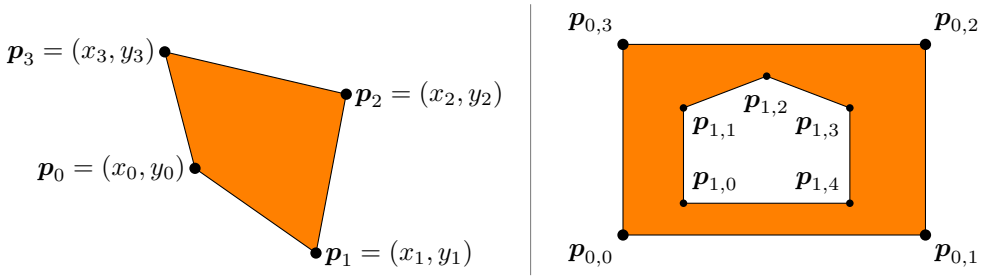


Figure 12: Simple polygon with four unique vertices is shown on the left hand side. A complex polygon with an outer hull and an interior hull is shown on the right hand side for comparison.

A polygon is considered invalid if one or more of its linear rings are self-intersecting, i.e. if any of its rings is considered to be complex. Data providers frequently provide polygons in invalid states and such polygons must be corrected since they are often not processable by common GIS tools. Zero-buffering invalid polygons (growing the polygon in all directions by zero units) fixes such problems, as can be seen in Figure 13.

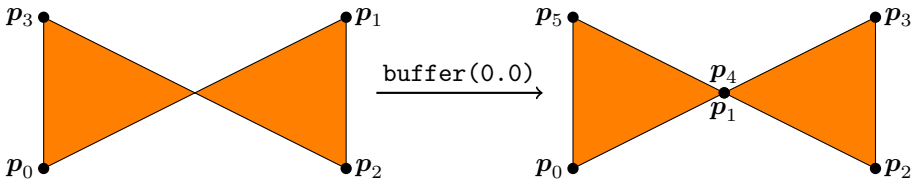


Figure 13: Illustration of how zero-buffering an invalid polygon corrects self-intersecting polygons.

Zero-buffering polygons has the added benefit of normalizing vector data by re-ordering the polygon vertices in an anti-clockwise manner and removing redundant vertices as shown in Figure 14.

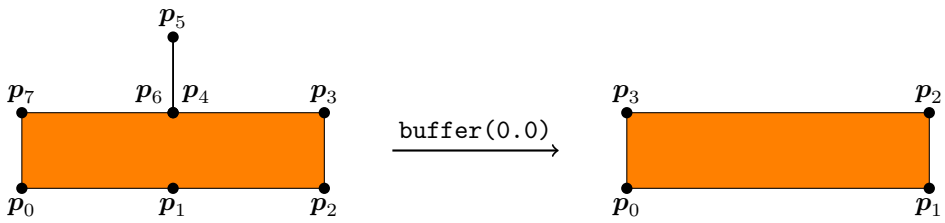


Figure 14: Illustration of how zero-buffering polygons removes redundant vertices.

This allows you to apply simpler similarity measures for comparing polygons, and reduces computational costs when processing the polygons. Technical details for applying zero-buffers to vector data is provided in Appendix A.3. We will come back to how to combine vector and raster datasets by *rasterization* in Section 2.5 where it will also become clear why the removal of redundant vertices is of importance.

Raster data

Raster data consists of a set of scalar measurements imposed onto a grid. A color image, I, of width W and height H, will contain three color channels; red, green, and blue (RGB), and can be represented by a three-dimensional array of size $\mathrm{H} \times \mathrm{W} \times 3$. Each color channel for a given pixel is represented by an unsigned 8-bit integer, i.e.

$$I_{i,j,c} \in \{0, 1, \ldots, 255\}, \qquad i = 1, \ldots, \mathrm{H}, \quad j = 1, \ldots, \mathrm{W}, \quad c = 1, 2, 3.$$

A LiDAR elevation map, which we will denote as Z, is likewise encoded as a single-channel grayscale image of size $\mathrm{W} \times \mathrm{H}$. Each pixel is represented by a signed 32-bit floating point value which gives the following approximate value domain

$$Z_{i,j} \in \mathbb{R}, \qquad i = 0, \ldots, \mathrm{H} - 1, \quad j = 0, \ldots, \mathrm{W} - 1.$$

These two raster types must be handled differently during data standardization/normalization due to their different value domains, which we will come back to in Section 2.6.

For GIS rasters specifically we must additionally provide the spatial extent of the given raster defined by:

- A coordinate system, for example UTM `32V`.
- The coordinate of the center of the upper left pixel, $I_{1,1}$; the *origin* $\boldsymbol{r}_0 = [x_0, y_0]^T$.
- The pixel step size, $\boldsymbol{\Delta} = [\Delta_x, \Delta_y]^T$, for example $[0.25\,\mathrm{m}, -0.25\,\mathrm{m}]^T$.

The pixel $I_{i,j,c}$ therefore represents a rectangle of width Δ_x and height Δ_y centered at the spatial coordinate $\boldsymbol{r}_0 + \boldsymbol{\Delta} \cdot [i, j]$ interpreted in the given coordinate system.

Missing data in remote sensing rasters is specified by filling in a predefined `nodata` placeholder value. For RGB data this is often set to 0, resulting in a black pixel. LiDAR rasters often use `nodata` $= -2^{127} \times (2 - 2^{-23}) \approx -3.4028234664 \times 10^{38}$, the most negative normal number representable by a single-precision floating point number. Such `nodata` values may arise from measurement errors or by pixels situated outside the given coverage area of the dataset, and must be special-cased during data normalization, which we will come back to in Section 2.6.

When we will train models on the combination of LiDAR and aerial photography data, these two types of rasters must be merged in order to attain a consistent three-dimensional array of size $H \times W \times 4$. These rasters can not be simply superposed when their pixel sizes $\boldsymbol{\Delta}$ and/or origins $\boldsymbol{r}_0$ differ. In such cases we will apply bilinear interpolation on the raster of greatest resolution and subsequently downsample it in order to align all pixels. See Appendix A.4 for how this is performed in practice.

2.3 Data sets

The modelling results presented in Section 3 are trained on GIS data covering the Norwegian municipality of Trondheim. All data sets have been made available by the "Norge digitalt"-partnership and have been downloaded from `https://geonorge.no`, an online service hosted by *Norwegian Mapping and Cadastre Authority* (*Statens Kartverk*). All data, unless otherwise stated, are licenced under the "Norge digitalt"-licence[3] which restricts the use to non-commercial purposes.

Raster data sets

We will use the "Ortofoto Trondheim 2017"[4] aerial photography data set from 2017 which requires 161 GB of storage space. The real image resolution is 0.04 m – 0.15 m, but is provided with an upsampled resolution of 0.1 m for consistency. The reported accuracy is ±0.35 m [29], although the exact type of this accuracy is not specified. An exemplified region is visualized in Figure 15.

Figure 15: Visualization of "Ortofoto Trondheim 2017" aerial photography data set. ©Kartverket.

An *orthophoto* is an image where the geographic scale is uniform over the entire image. Proper orthophotos are expensive to manufacture and are therefore seldomly available for most geographic regions [30], including Trondheim. Aerial photography which has not been properly

[3] Information regarding the "Norge digitalt"-licence can be found here: `https://www.geonorge.no/Geodataarbeid/Norge-digitalt/Avtaler-og-maler/Norge-digitalt-lisens/`.

[4] Product specification for "Ortofoto Trondheim 2017" can be found here: `https://kartkatalog.geonorge.no/metadata/cd105955-6507-416f-86d2-6d95c1b74278`.

"ortho-rectified" may impede location-based inference as there exists no exact one-to-one mapping between image pixels and geographic coordinates. This problem is best understood by an example, as shown in Figure 16.

Figure 16: Example of nonproper orthophoto. The building centered in the image is 14 stories tall. The orange area annotates a clearly visible building wall. ©Kartverket.

As can be seen in Figure 16, the "Ortofoto Trondheim 2017" data set clearly shows one side of a building due to the perspective of the plane capturing the image. An ideal orthophoto would capture all vertical building walls as single, straight lines, no matter the perspective. The effect of this "parallax error" on segmentation predictions will be investigated in Section 3.2.

The LiDAR data set used is "Høydedata Trondheim 5pkt 2017"[5] from 2017-10-10 and requires 25 GB of storage space. The resolution is 0.2 m and has a reported standard deviation of 0.02 m [2]. LiDAR visualized as a grayscale image over the same region as in Figure 15 is presented in Figure 17.

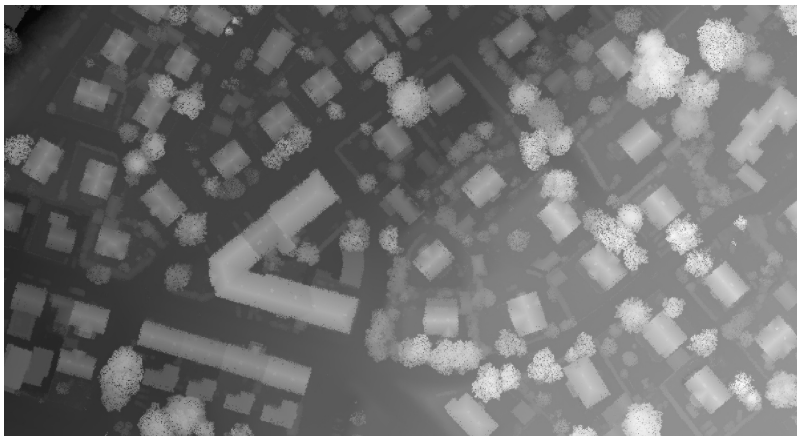


Figure 17: Visualization of "Høydedata Trondheim 5pkt 2017" LiDAR data set. ©Kartverket.

[5]Product specification for "Høydedata Trondheim 5pkt 2017" can be found here: `https://kartkatalog.geonorge.no/metadata/bec4616f-9a62-4ecc-95b0-c0a4c29401dc`.

The LiDAR dataset is partially incomplete due to measurement errors, and certain pixels are therefore filled in with `nodata` placeholder values as explained in Section 2.2. Table 1 shows the frequency of such `nodata` values in the data set. The table has been produced by moving a rolling window of size 10 m × 10 m over the entire coverage area and calculating the proportion of pixels with valid values within each non-overlapping window.

Point density (m^{-2})	Proportion (%)
$> 100\%$	97.7
85 % – 100 %	1.2
60 % – 85 %	1.1

Table 1: Control of point cloud density of the Trondheim 2017 LiDAR data set. The densities are calculated within rolling windows of size 10 m × 10 m [2].

Vector data sets

The "Matrikkelen - Eiendomskart Teig"[6] data set contains all cadastral plots in Trondheim, the use of which will be explained in Section 2.4. The "FKB-bygning"[7] data set contains all registered building outlines in Trondheim. The building outlines will be used to construct binary classification masks as outlined in Section 2.5. Both data sets are illustrated in Figure 18.

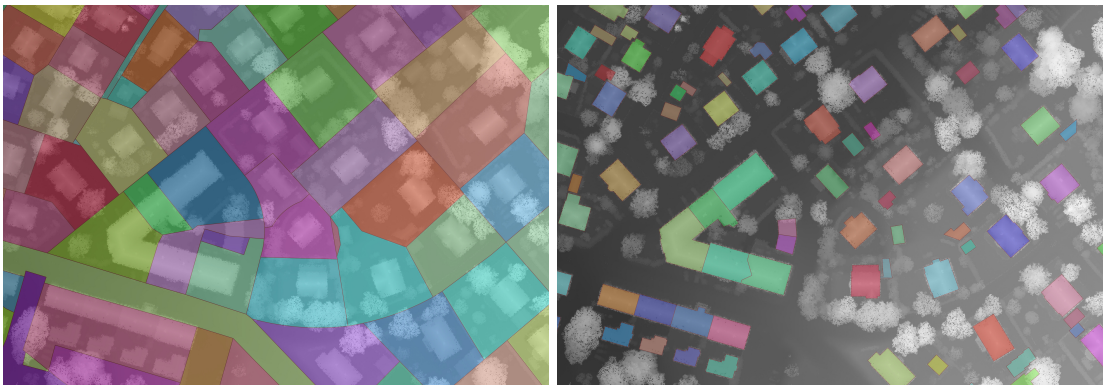


Figure 18: Illustration of vector data sets. Cadastral plots are shown on the left while building outlines are shown on the right. ©Kartverket.

2.4 Tiling Algorithm

The data sets provided to us are in a state unsuitable for direct use by machine learning frameworks. For this reason we need to develop a preprocessing pipeline that transforms the data

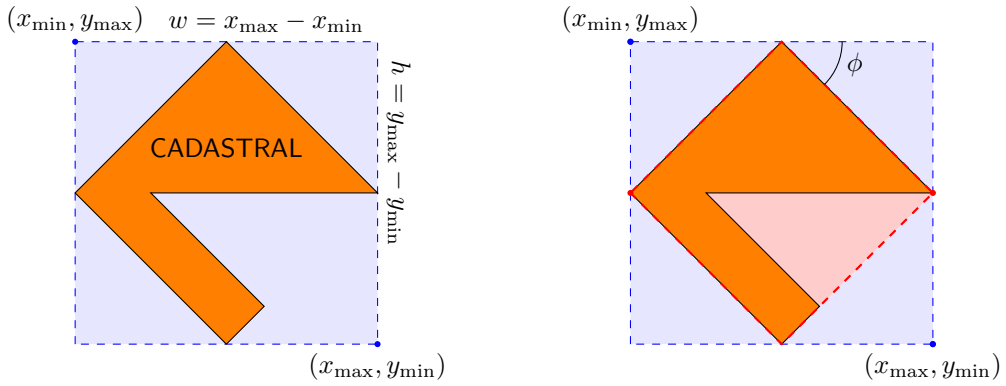
[6]Product specification for "Matrikkelen - Eiendomskart Teig" can be found here: `https://kartkatalog.geonorge.no/metadata/74340c24-1c8a-4454-b813-bfe498e80f16`.

[7]Product specification for "FKB-bygning" can be found here: `https://kartkatalog.geonorge.no/metadata/8b4304ea-4fb0-479c-a24d-fa225e2c6e97`.

into a more customary format. The data preprocessing should be generalizable to different regions, data formats, data types (vector vs. raster), coordinate systems, and so on. The goal is to implement a modelling pipeline that can be applied to other geographic regions in the future.

Our data sets are defined over a single, contiguous geographic area, and we must therefore define a *sample space* which allows us to split the data into training-, validation-, and test-sets. The collection of all cadastral plots in a given region is a suitable sample space since cadastral plots are non-overlapping regions of relatively small size and have a high probability of containing one or more buildings. A large raster dataset covering a sparsely populated region can therefore be substantially reduced in size before training. An alternative approach is to split the entire data set into regularly sized tiles and use this tile collection as the sample space. A tiled sample space, for anything other than densely populated areas, will suffer from class imbalances due to low building densities in most tiles.

Given a specific geographic region, defined by the extent of the cadastral plot, we must retrieve the raster which covers the region of interest. The simplest approach is to calculate the *axis-aligned bounding box* of the plot, the minimum-area enclosing rectangle of the given plot. A bounding box is uniquely defined by its centroid $\boldsymbol{c} = [1/2(x_{\min} + x_{\max}), 1/2(y_{\min} + y_{\max})]$, width $w = x_{\max} - x_{\min}$, and height $h = y_{\max} - y_{\min}$, and we will denote it by $B(\boldsymbol{c}, w, h)$. This is shown in Figure 19a.



(a) Bounding box calculation for a given cadastral. The cadastral is shown in orange, and the resulting bounding box is annotated with blue dashed lines.

(b) Figure showing the difference between a regular bounding box shown in blue, and a minimum rotated rectangle shown in red. Angle of rectangle rotation denoted by ϕ.

Figure 19: Comparison of bounding box methods.

The edges of the bounding box is by definition oriented parallel to the coordinate axes. An alternative method is to calculate the *arbitrarily oriented minimum bounding box* (AOMBB), a rectangle rotated by ϕ degrees w.r.t. the x-axis, as shown in Figure 19b.

While AOMBB yields regions with less superfluous raster data, it requires warping of the original raw raster whenever ϕ is not a multiple of 90°, i.e. $\phi \notin \{0°, 90°, 180°, 270°\}$. Such warping requires data interpolation of the original raster data due to the rotation of the coordinate system, and may introduce artifacts to the warped raster without careful parameter tuning. AOMBB is therefore not a viable approach during the preprocessing stage, and we will therefore use axis-aligned minimum bounding boxes instead, from now on simply referred to as *bounding boxes*.

Calculating bounding boxes for the cadastral plots in our data sets will yield rectangles of variable dimensions. Variable input sizes will cause issues for model architectures which require predefined input dimensions. Convolutional neural networks do handle variable input sizes, but dimensions off all images in a *single* training batch must be of the same size. It is therefore preferable to normalize the size of each bounding box.

The distributions of the bounding box widths (w), heights (h), and maximal dimensions ($m = \max\{w, h\}$) are shown in Figure 20.

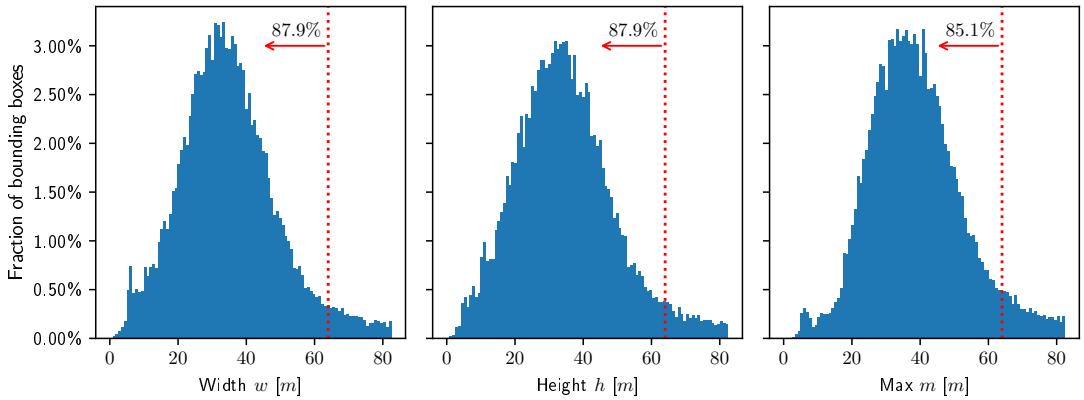


Figure 20: Distribution of bounding box widths w (left), heights h (middle), and largest dimension $m = \max\{w, h\}$ (right). The cut-off value of 64 m is shown by red dotted vertical lines. The fraction of bounding boxes with dimension ≤ 64 m is annotated as well. The x-axis has been cut off at the 90th percentile. *Dataset: Trondheim cadastre.*

As can be seen in Figure 20, the distributions of h and w are quite similar, as expected. A square $1:1$ aspect ratio is therefore suitable for the normalized bounding box size. Specifically, a $64\,\text{m} \times 64\,\text{m}$ bounding box will be of sufficient size to contain $\approx 85\,\%$ of all cadastre plots in a single tile. With a LiDAR resolution of 0.2 m, this results in a final image resolution of $256\,\text{px} \times 256\,\text{px}$. This resolution has the added benefit of being a common resolution for CNNs.

How should the bounding boxes be normalized to to $256\,\text{px} \times 256\,\text{px}$? A common technique is to resize the original image by use of methods such as bilinear interpolation or Lanczos resampling. While this is tolerable for normal photographs, where each pixel has a variable surface area mapping, it is an especially lossy transformation for remote sensing data. In the Trondheim 2017 LiDAR data set, for instance, each pixel represents a $0.2\,\text{m} \times 0.2\,\text{m}$ real world area. If the highly variable extent of each bounding box is scaled to $256\,\text{px} \times 256\,\text{px}$, the real world area of each pixel will differ greatly between cadastral plots. Resized images will also become distorted whenever the original aspect ratio is not $1:1$.

A better method utilizes the fact that the remote sensing data covers a continuous geographic region, which allows us to expand the feature space beyond the original region of interest. The original bounding box is denoted as $B(\boldsymbol{c}, w, h)$. Now, define the following "enlarged" width and height:

$$h^* := \left\lceil \frac{h}{64\,\text{m}} \right\rceil \cdot 64\,\text{m}, \qquad w^* := \left\lceil \frac{w}{64\,\text{m}} \right\rceil \cdot 64\,\text{m}$$

The new bounding box, $B(\boldsymbol{c}, w^*, h^*)$, covers the original bounding box and is divisible by 256 px

in both dimensions. In other words, the original bounding box is grown in all directions until both the width and height are multiples of 64 m (256 px). This is demonstrated in Figure 21.

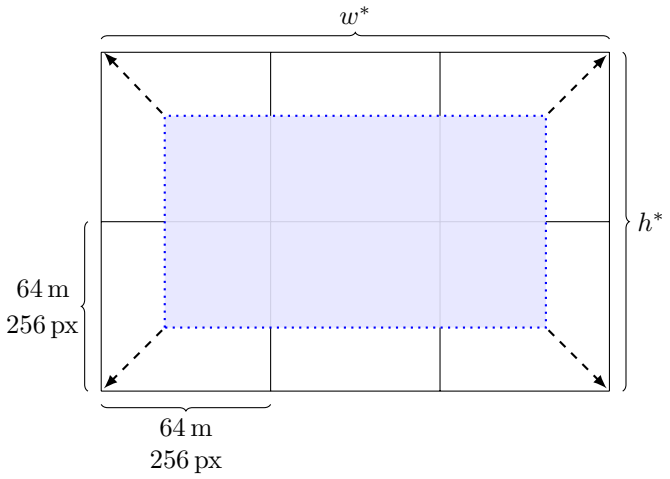


Figure 21: Bounding box of width $2.25 \cdot 64\,\text{m} = 144\,\text{m}$ and height $1.25 \cdot 64\,\text{m} = 80\,\text{m}$. The bounding box is grown until it is 3 tiles wide and 2 tiles tall, i.e. $192\,\text{m} \times 128\,\text{m}$.

The resulting bounding box can now be divided into $w^*h^*/64^2$ tiled images of resolution $256\,\text{px} \times 256\,\text{px}$, every pixel representing a $0.2\,\text{m} \times 0.2\,\text{m}$ surface area, and no spatial information has been lost in the process. Each tile's geographic extent is uniquely defined by the coordinate of the upper left corner (*tile origin*), since the tile dimensions are identical. An affine transformation from the UTM zone into the tile's discretized coordinate system can be constructed from the tile origin.

The additional area, $B(\boldsymbol{c}, w, h) \setminus B(\boldsymbol{c}, w^*, h^*)$, is filled with real raster data and respective target masks, and therefore may cause expanded bounding boxes to partially overlap. This will result in certain cadastral plots to share features, and must therefore be carefully dealt with in order to prevent data leakage across training, validation, and test splits. Another approach is to fill in the additional area with zero-values, effectively preventing all data leakage between cadastral plots. A disadvantage with this approach is that all models are now required to learn to ignore this additional, fake data, and this could result in reduced predictive performance and/or longer training times.

2.5 Masking Algorithm

In order to create a ground truth segmentation mask we must convert the vector-formatted mask polygons, building outlines in our case, into the same rasterized format as the remote sensing data. The construction of discretized segmentation masks from vectorized mask polygons is performed by Algorithm 1.

Algorithm 1: `Discretized masking`

1. Transform the mask polygons into the pixel coordinate system of the raster tile, using the affine transformation defined by the tile origin.
2. Superimpose the polygon on the discretized pixel grid and crop polygons outside the pixel region $(0, 255) \times (0, 255)$.
3. Fill in the value 1 for any pixel contained by the polygon exterior hulls, while not contained by any interior hull.
4. Set remaining values to 0.

A problem arises when pixels are partially contained by a polygon exterior and interior, i.e. when the pixel overlaps the polygon's boundary. The pixel must be rather arbitrarily considered as either contained (decision rule A) or not contained (decision rule B) by the polygon. Both decision rules are shown in Figure 22.

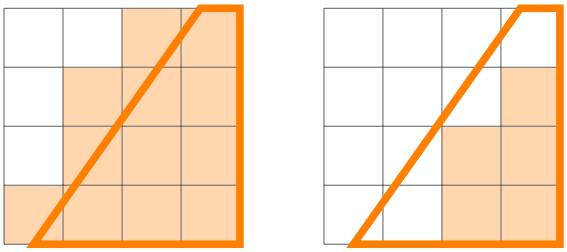


Figure 22: The same polygon discretized to a raster grid using two different techniques. In the left figure, all pixels being *touched* by the interior of the polygon are considered a part of the polygon (decision rule A), while in the left figure, only pixels entirely *contained* within the interior are considered being part of the polygon (decision rule B).

An alternative is to average the two masks, resulting in mask values of 0.5 where the two decision rules disagree. Approximately 9.2 % of mask pixels of value 1 are situated along the boundary of a discretized mask polygon (1.7 % of *all* pixels regardless of value) and may therefore be affected by this decision. We have opted for decision rule A. The distribution of the mask class balance across all produced tiles is shown in Figure 23.

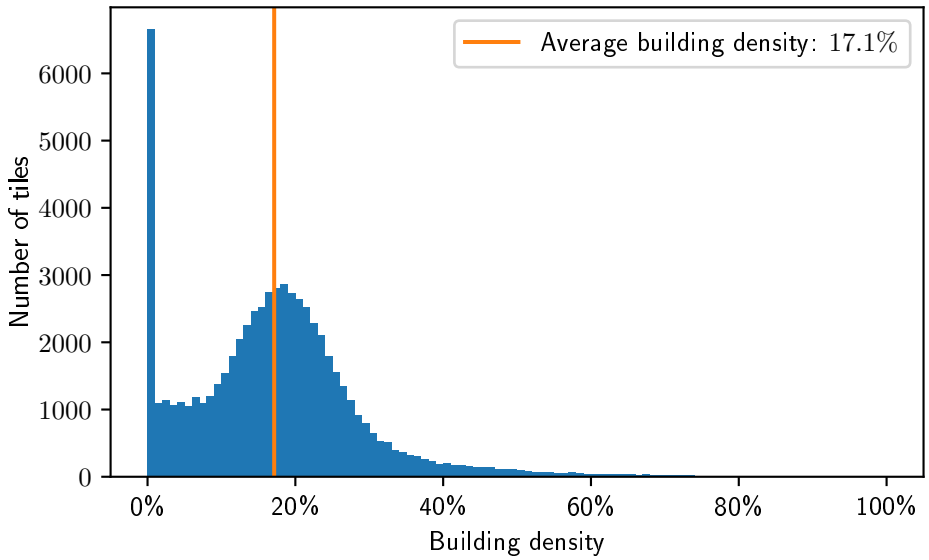


Figure 23: Distribution of *building density* across all produced tiles in Trondheim. Building density is defined by number of pixels positioned on top of buildings divided by total number of pixels.

The average tile has a building density of approximately 17 %, that is 700 m^2 of 4096 m^2 is occupied by buildings. Of all the produced tiles approximately 8.32 % end up having no positive mask pixels, i.e. no buildings are situated within these tiles.

2.6 Raster Normalization

Input data normalization has been found to be of vital importance when training neural networks, in certain cases reducing predictive errors by several orders of magnitude and training times by one order of magnitude [53]. How to normalize input data depends on distribution of the feature space, which will be investigated here.

RGB rasters

A given RGB pixel is an unsigned 8-bit integer and therefore takes values in a bounded, integer domain

$$I_{i,j,c} \in \{0, 1, \ldots, 255\}, \text{ for } c \in \{r, g, b\}.$$

The distribution of each color channel over the entire coverage area of the Trondheim aerial photography data set is shown in Figure 24, and aggregate statistics for each channel are listed in Table 2.

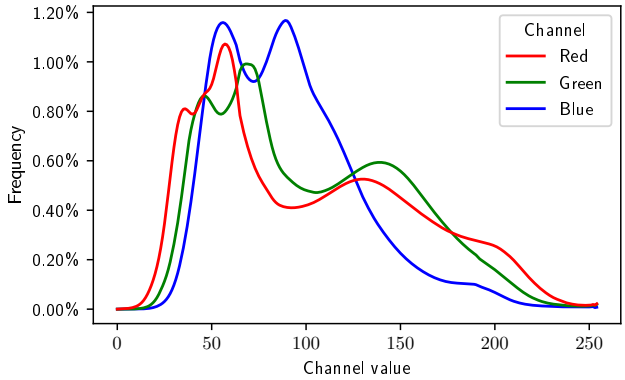


Figure 24:
Distribution density for all three color channels in the aerial photography data set covering Trondheim municipality (2017).

Channel	Mean [1]	SD [1]
Red	101.6	55.0
Green	102.7	48.7
Blue	91.3	37.2

Table 2:
Aggregate statistics for each image channel distribution for the aerial photography data set covering Trondheim municipality (2017).

The image channels can be easily normalized to the domain $[0, 1]$ by dividing by 255 across all three image channels. This is in essence a lossless transformation, since the normalization function $f(x) = x/255$ is trivially invertible, and thus no information is lost by this normalization.

LiDAR rasters

The "z channel" represents elevation data from the respective digital surface model. Elevation measurements are represented by 32-bit, single-precision floating point numbers, and can theoretically take values in the domain $I_{i,j,z} \in (-3.4 \times 10^{38}\,\mathrm{m},\ 3.4 \times 10^{8}\,\mathrm{m})$. In practice, the measurements are bounded by the regional extrema, $(-433\,\mathrm{m},\ 8848\,\mathrm{m})$ for dry land globally, and $(-9\,\mathrm{m},\ 569\,\mathrm{m})$ for the Trondheim region. The distribution of z channel values for the Trondheim region is shown in Figure 25, and aggregate statistics are listed in Table 3.

A normalization technique analogue to the RGB min-max scaling for elevation tile number k, denoted as $Z^{(k)}$, would therefore be

$$\begin{aligned}\hat{Z}_{i,j}^{(k)} &= \frac{Z_{i,j}^{(k)} - \min\limits_{t\in\mathrm{TRD}} Z^{(t)}}{\max\limits_{t\in\mathrm{TRD}} Z^{(t)} - \min\limits_{t\in\mathrm{TRD}} Z^{(t)}} \\ &= \frac{Z_{i,j}^{(k)} + 9\,\mathrm{m}}{578\,\mathrm{m}},\end{aligned} \qquad \text{(Global min-max normalization)}$$

where TRD is the index set of all tiles belonging to the Trondheim region. The normalized raster elevation values in $\hat{Z}^{(k)}$ are guaranteed to be bounded to the interval $[0, 1]$, as with the RGB raster. In order to evaluate if this will properly normalize the z raster channel across tiles, we plot the "tile-by-tile" z channel statistics in Figure 26.

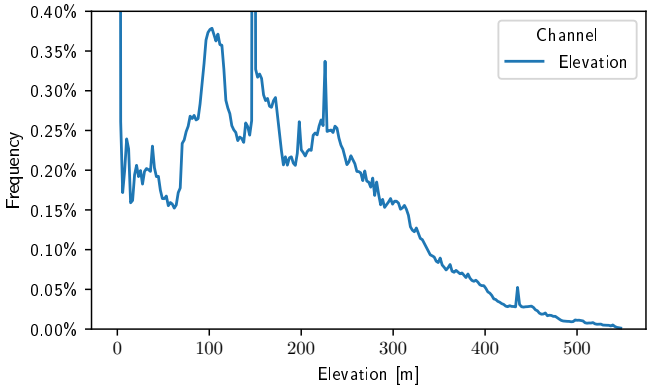


Figure 25: Distribution density for elevation data set covering Trondheim municipality (2017). Outlier values (0 m, 2.74 %) and (148 m, 1.93 %) have been cropped.

Channel	Mean [m]	SD [m]
Elevation	155.4	116.5

Table 3: Aggregate statistics for elevation data set covering municipality of Trondheim (2017).

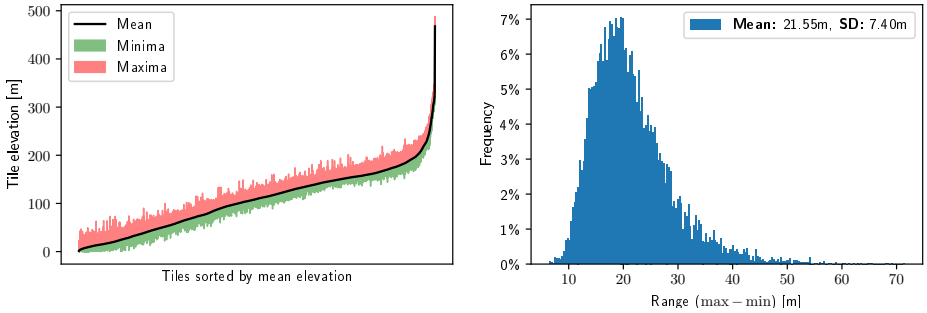


Figure 26: Elevation value statistics for a tile subset of sample size $n = 10\,000$. The left figure shows the minimum, mean, and maximum elevation, sorted by increasing mean from left to right. The right figure shows the histogram of the tile elevation *ranges* (difference between maximum elevation and minimum elevation within tile).

While the global elevation range is 569 m − (−9 m) = 578 m, the elevation range within each respective tile is on average approximately 22 m±8 m(SD), that is, much less than 578 m. Coupled with the fact that the tile elevation *means* are somewhat uniformly distributed between 0 m and 200 m, ignoring the right tail, a global normalization will yield tile elevation values with small standard deviations and highly variable means. We can therefore conclude that global min-max scaling is not suitable for the elevation image channel. A proposed solution to this problem is to scale each tile independently to the domain $[0, 1]$, what we will refer to as "local min-max

normalization".

$$\begin{aligned}\hat{Z}_{i,j}^{(k)} &= \frac{Z_{i,j}^{(k)} - \min\ Z^{(k)}}{\max\ Z^{(k)} - \min\ Z^{(k)}} && \text{(Local min-max normalization)}\\ &= \frac{Z_{i,j}^{(k)} - \beta}{\alpha - \beta}, && \text{where } \alpha := \max\ Z^{(k)},\ \beta := \min\ Z^{(k)}.\end{aligned}$$

The scaling factor $\alpha - \beta$ is constructed such that the normalized tile minimum becomes 0 and maximum becomes 1 for all tiles.

Any elevation normalization method must account for the fact that missing data values are replaced by a pre-defined `nodata` placeholder value, usually -3.4×10^{38} m. Otherwise a large negative bias is introduced for all tiles with any missing data. Leaving `nodata` values unnormalized with such extreme values will heavily influence the weighted sum calculated by nodes in any neural network, and must therefore filled in with values from the normalized domain. Filling in 0 values for all `nodata` indices has been shown to work well in most cases. The "`nodata`-aware" min-max normalization algorithm used for preprocessing elevation input data is given in Algorithm 2.

Algorithm 2: `Nodata-aware local min-max normalization`

1 Calculate the valid index set defined by $\mathcal{V} = \{(i,j) : Z_{i,j} \neq \texttt{nodata}\}$.

2 Calculate $\alpha = \max\limits_{(i,j)\in\mathcal{V}} Z_{i,j}$ and $\beta = \min\limits_{(i,j)\in\mathcal{V}} Z_{i,j}$.

3 Construct normalized raster defined by

$$\hat{Z}_{i,j} = \begin{cases} \frac{Z_{i,j}-\beta}{\alpha-\beta}, & \text{if } (i,j) \in \mathcal{V}, \\ 0, & \text{otherwise.} \end{cases}$$

One of the core issues with local min-max normalization is that it is essentially a lossy operation. As each tile is independently scaled, it is no way to accurately reconstruct the original elevation map in metric units. One way to determine if a roof-like surface belongs to a proper building or a shed, for instance, is to inspect its relative height, which becomes impossible without knowing the relative scaling of each tile with respect to each other. We therefore hypothesize that the variable scaling imposed by local min-max normalization could impede the performance of models trained on such data. An alternative normalization method is therefore proposed where the scaling factor $\alpha - \beta$ is replaced by a predefined *constant* scaler $\gamma > 0$. The translation β is kept as-is since there is no reason to distinguish between cadastral plots situated at sea-level and other altitudes when it comes to building outline detection. This "metric normalization" is therefore defined as:

$$\hat{Z}_{i,j}^{(k)} = f\left(Z_{i,j}^{(k)}\right) = \frac{Z_{i,j}^{(k)} - \min\ Z^{(k)}}{\gamma}, \qquad \gamma > 0. \qquad \text{(Metric normalization)}$$

Elevation values in the z channel now have a consistent physical interpretation given in units m/γ across all tiles. The modified metric normalization method is provided in Algorithm 3.

Algorithm 3: `Nodata-aware metric normalization`

1 Calculate the valid index set defined by $\mathcal{V} = \{(i,j) : Z_{i,j} \neq \texttt{nodata}\}$.

2 Calculate $\beta = \min_{(i,j)\in\mathcal{V}} Z_{i,j}$ and define a global scaler $\gamma > 0$.

3 Construct normalized raster defined by

$$\hat{Z}_{i,j,z} = \begin{cases} \frac{Z_{i,j}-\beta}{\gamma}, & \text{if } (i,j) \in \mathcal{V}, \\ 0, & \text{otherwise.} \end{cases}$$

A comparison of these two normalization methods, Algorithm 2 and Algorithm 3 that is, will be provided in Section 3.4.

3 Experiments

This section will investigate the prediction of building outlines using data produced by the pipeline outlined in Section 2. The U-Net model architecture presented in Section 1.5 is used for semantic segmentation. We start by describing the general experimental setup in Section 3.1. A comparative investigation into the suitability of the different raster data types (aerial photography and/or LiDAR DSMs) for predicting building outlines is presented in Section 3.2. In Section 3.3 we investigate if techniques intended to combat overfitting and increase training speed actually have their intended effect; specifically batch normalization, dropout, and data augmentation. The LiDAR raster normalization methods presented in Algorithm 2 and Algorithm 3 are implemented and compared in Section 3.4. Finally, Section 3.5 compares the empirical efficiency of different surrogate loss functions.

3.1 Experimental Setup

Training procedure

Training summary
- 58 559 labeled geographic tiles: $256\,\text{px} \times 256\,\text{px} = 64\,\text{m} \times 64\,\text{m}$.
- 41 018 / 8788 / 8753 training / validation / test.
- Random shuffling.
- 16 variations of augmentation.
- Adam optimizer.
- Validation IoU early stopping.

The Trondheim dataset produces 58 559 geographic tiles after being processed, each tile including aerial photography (RGB) data, elevation data (LiDAR elevation), and ground truth masks for building footprints. This sample space is split into a customary 70% / 15% / 15% training–validation–testing split. The training data is randomly shuffled and augmented at the beginning of each epoch in order to reduce overfitting. The data augmentation consists of a random application of horizontal and/or vertical flipping in addition to a rotation by a random integer multiple of 90 degrees. The training data is subsequently grouped into batches of size 16 before applying the Adam optimizer. Training is continued until observed convergence by the use of the IoU evaluation of the validation split. The weights corresponding to the epoch yielding the best validation metric is used as the final model parametrization.

Software

The source code written in order to produce and present the results in this paper is openly available at `https://github.com/JakobGM/project-thesis`. The majority of the source code is written in Python as it arguably has the best software ecosystem for both GIS *and* deep learning workflows. This work would not have been possible if not for the vast array of high quality open source software available. The Geospatial Data Abstraction Library (GDAL) [12] has been extensively used in order to process GIS data, and the python wrappers for GDAL, Rasterio [14] for raster data and Fiona [13] for vector data, are central building blocks of the data processing pipeline. The machine learning framework of choice has been the new 2.0 release of TensorFlow [1], most of the modelling code having been written with the declarative Keras API. This is not an exhaustive list of all dependencies, but a complete list of software dependencies and a reproducible Docker [40] image is provided with the source code for this project.

Hardware and performance

All numerical experiments have been performed by a desktop class computer with the following relevant technical specifications:

- **Processor:** *AMD Ryzen 9 3900X*.
  12 cores / 24 threads, 3.8 GHz base clock / 4.6 GHz boost clock.
- **Graphics card:** *MSI GeForce 2070 Super*.
  8 GB GDDR6 VRAM, 1605 MHz clock speed, 9.062 TFLOPS 32-bit performance.
- **Memory:** *Corsair Vengeance LPX DDR4* 3200 MHz *32GB*.
- **Storage:** *Intel 660p 1TB M.2 SSD*.
  Up to 1800 MB s^{-1} read and write speed.

Model performance

- 218 ms per training step (batch 16)
  $\implies$ 14 ms per sample.
- 11 min per training epoch
  $\implies \approx 16.5$ h per experiment.
- 8 ms per prediction (batch 1)
  $\implies$ 125 predictions per second.

With a batch size of 16, each training step requires 218 ms of computation, resulting in approximately 14 ms per geographic tile. When including the streaming of data from disk, updating weights based 2563 training batches of size 16, validating the model on 549 additional validation batches, and executing various Keras callbacks, each epoch lasts for 11 minutes from end to end. Most experiments have been trained for 90 epochs, hence requiring altogether 16 and a half hours of training. The final models are able to produce 125 predictions of size 256 px $\times$ 256 px per second.

3.2 Features

The two types of available remote sensing data is LiDAR elevation data and aerial photography, the latter simply referred to as *RGB* data from now on. We will investigate to what degree these features are useful for predicting building outline segmentation masks. Of special interest is how these two feature types compare to each other when it comes to the predictive accuracy. Both feature types provide birds-view perspectives, and building outline segmentation is therefore essentially equivalent to roof surface detection. We hypothesize that models based on LiDAR will fare better than RGB models due to the importance of the flatness and relative height of roof surfaces, a more characteristic property than the specific visual appearance of roof surface textures. A model using *both* features types combined will also be constructed and trained, and we will compare the accuracy of this model to the two models using the respective feature types in isolation.

RGB data

We start by training a model based solely on RGB data, every color channel normalized as explained in Section 2.6. The training procedure is illustrated in Figure 27.

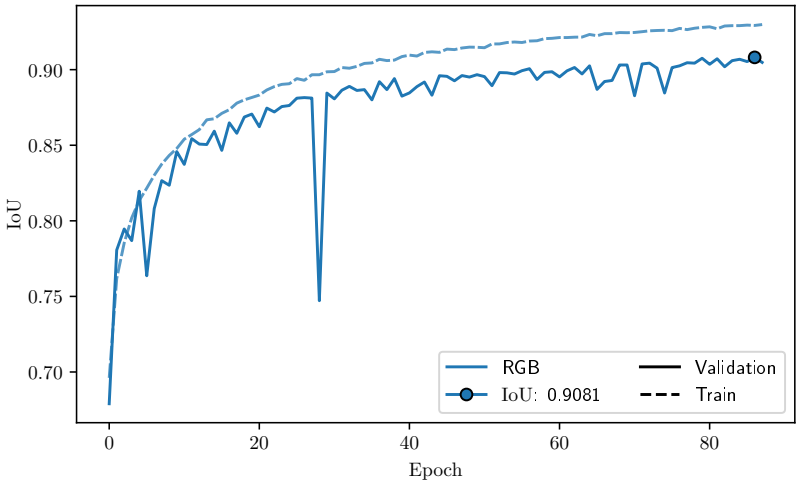


Figure 27: Training of U-Net model for 89 epochs, using RGB data. The training epochs are given along the horizontal axis, while the end-of-epoch IoU evaluations are given along the vertical axis. Validation split IoU is shown as a blue solid line, while the training split IoU is shown as a blue dashed line. The epoch yielding the best validation IoU is annotated as a solid blue circle, in this case the 88th epoch with a validation IoU of 0.9081.

As can be seen in Figure 27, the training and validation IoU metrics improve relatively consistently from epoch to epoch, with the exception of epoch 24 where a large spike can be observed in the validation IoU. Such spikes will reappear in later training procedures, but the models always recover in the subsequent epochs. Training is continued until validation IoU does not improve, and the epoch corresponding to the best validation IoU used used as the final model parametrization. The final model is evaluated on the test set and the distribution of the resulting IoU test metrics is shown in Figure 28.

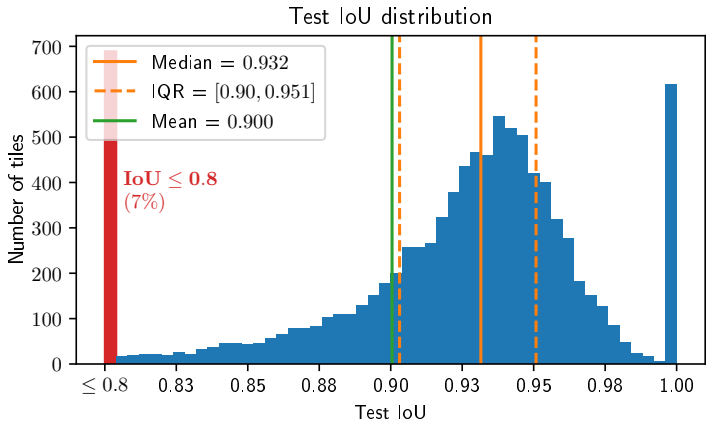


Figure 28: Distribution of IoU evaluations of the RGB model over tiles from the test set. The left tail of the distribution (IoU ≤ 0.8) constituting 7 % of the data has been cropped and included into the left-most bin colored in red. The interquartile range (IQR) is annotated in orange and the mean in green.

Figure 28 shows that the IoU test metrics portray a left-skewed distribution with mean 0.9, while 7% of all test cases have IoU metrics less than or equal to 0.8. In order to get a more intuitive understanding of the model performance we plot the segmentation corresponding to the *median* IoU metric of the test set in Figure 29. All upcoming prediction plots, unless otherwise stated, will use features exclusively from the test set.

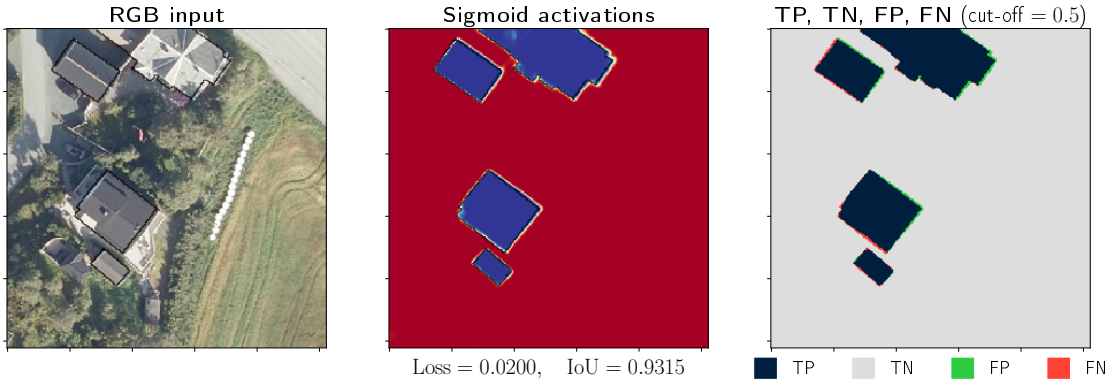


Figure 29: Median IoU prediction from the test set. The left panel shows the RGB input provided to the model before normalization. The middle panel shows the final sigmoid output of the model. A diverging color scheme is used for the activations where red indicates output values close to 0 and blue indicates output values close to 1. Values close to 0.5 are shown in white. The pixels situated along the borders of the discretized ground truth mask are shown in black in both the left and middle tile. Finally, the right tile shows the classification of each segmentation pixel, either true positive (TP), true negative (TN), false positive (FP), and false negative (FN). These classifications are calculated by using a threshold of 0.5 and comparing the thresholded values to the ground truth mask.

Half of the model predictions using the test set are at least as good as the prediction shown in Figure 29, and likewise for worse predictions. We will now investigate the worst-case model predictions in order to identify the conditions under which the model does *not* perform well. These conditions can be divided into two categories: those conditions which are closely related to the nature of RGB data, and those who are not. Two illustrative examples from the latter category are shown in Figure 30. Figure 30 shows the worst outliers in the test set, prediction (30 a) being the worst prediction *overall*, while prediction (30 b) is the worst prediction with an above-average building density. These two predictions demonstrate the two main causes for negative outliers in the metrics. The first one, as shown in prediction (30 a), is when segmentation mask becomes vanishingly small. Small masks are not just generally difficult for CNNs to segment, they are also negatively affected by the fact that the IoU metric becomes more sensitive to single-pixel changes. That is, misclassifying 100 pixels when the ground truth mask contains 10 000 pixels yields a much greater IoU metric compared a ground truth mask of only 1000 positive pixels. The consequence of this phenomenon is demonstrated in Figure 31, where it becomes clear that the worse model evaluations are generally characterized by being low building density tiles.

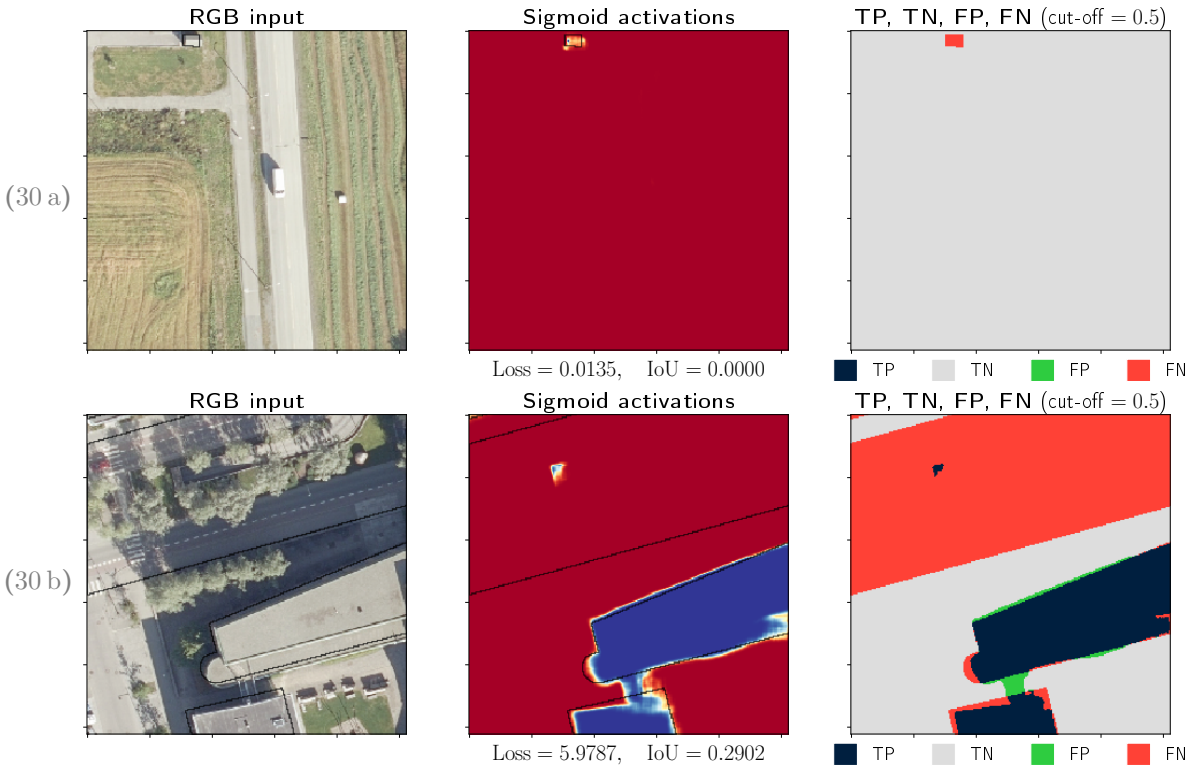


Figure 30: Prediction 30a, shown on the top, is the worst prediction in the test set with an IoU metric of 0. Prediction 30b, shown on the bottom, is the worst prediction amongst all test cases with above-average building densities (17.1 %). See caption of Figure 29 for detailed figure explanation.

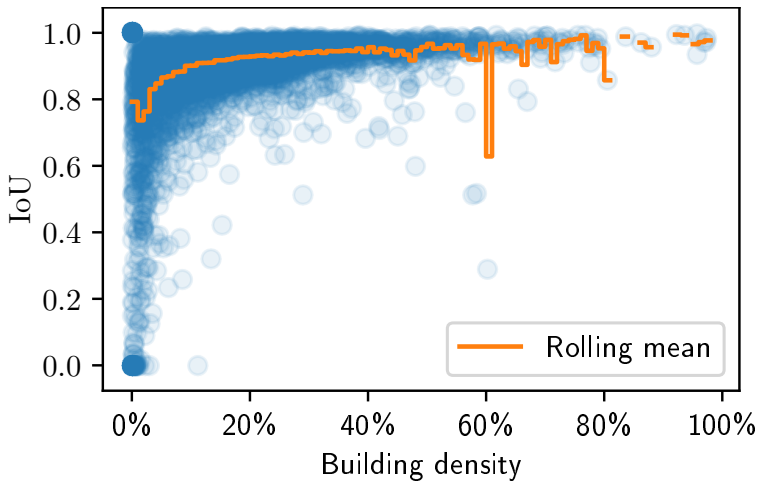


Figure 31: Figure showing the relationship between ground truth building density and the respective IoU evaluation for the test set. The orange line indicates the interval mean IoU along the building density axis, interval length being 1%.

The second issue causing negative outliers is the presence of wrong data in the ground truth segmentation masks, as shown in prediction (30 b) in Figure 30. Such errors are almost exclusively caused by buildings having been built or demolished in the intermittent time period between the datum of the feature data set and the datum of the ground truth data set. The presence of errors in the ground truth mask is fortunately rarely observed.

We will now look at the remaining category of negative outliers, a category much more related to the intrinsic properties of RGB data and its use as a predictor. Inspection of these failures may help us gain some insight into the internals of the model and how it constructs predictions. We present four illustrative examples of when the RGB model faces difficulties in Figure 32. Prediction (32 a) demonstrates the importance of contrast in order to distinguish the edges of building outlines. Ground truth mask edges with low RGB contrast are often wrongly segmented. The same can be said of mask interiors with high contrast "fake" edges as in (32 b). The texture of the roof surface also seems to be taken into account by the model, as shown in prediction (32 c) where the presence of white flecks on the eastern roof impedes the model's ability to recognize the surface as being part of a roof, while this issue is not observed with the roof surface on the western half of prediction (32 c) where a more common roof texture is present. Roof surfaces covered with greenery as shown in prediction (32 d) also cause difficulties for the model; not unexpected since it can be considered a type of camouflage.

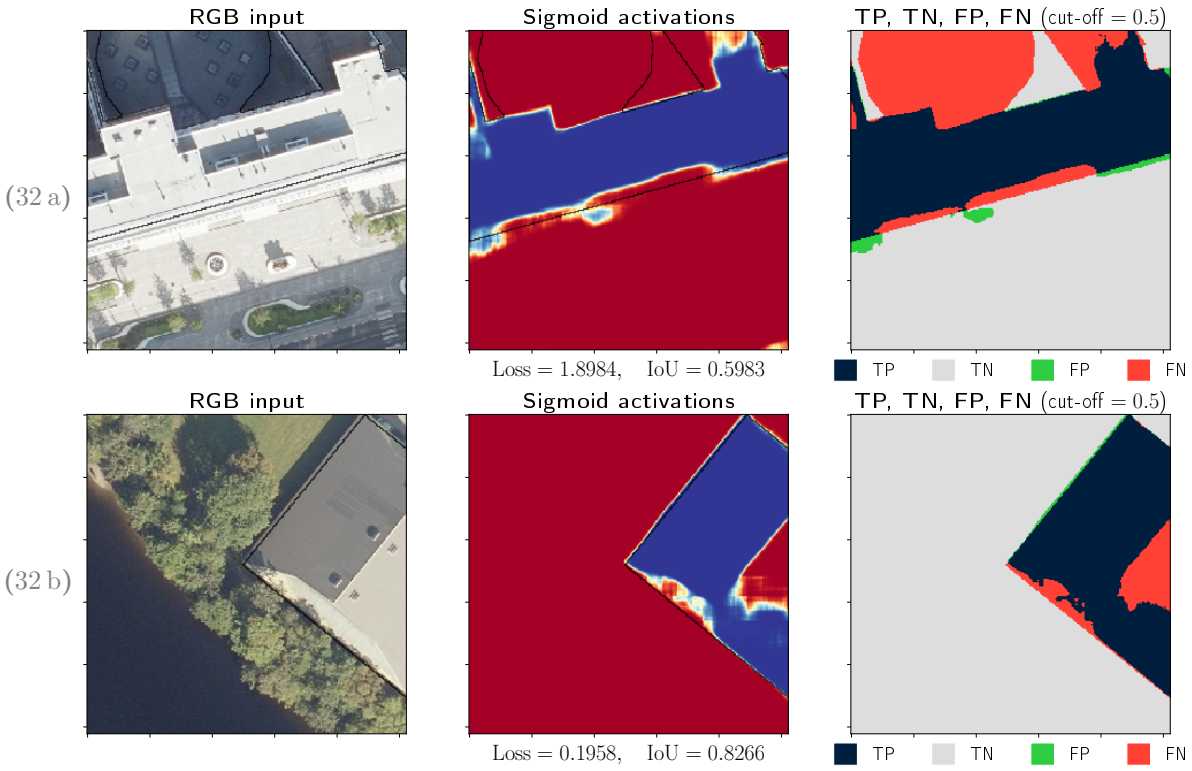


Figure 32: Continued on next page...

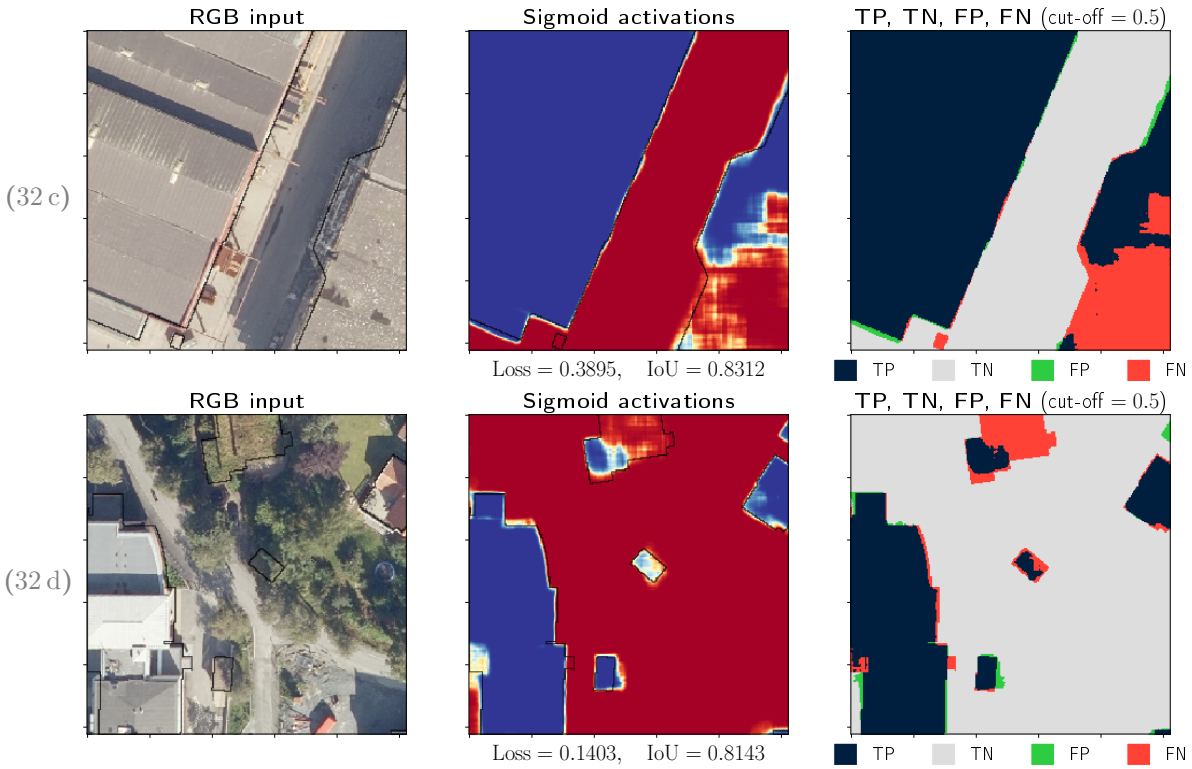


Figure 32: Illustrative failures of CNN segmentation of building outlines using RGB data.

We hypothesized in Section 2.3 that improper orthophotos, aerial images with non-orthogonal, non-vertical perspectives, would cause a high degree of segmentation misalignment due to the RGB photo being pixel-misaligned with respect to the geographically specified building outline. In practice, however, the RGB model seems to be remarkably well adjusted to misaligned perspectives as can be seen in Figure 33. The "RGB input" panel shown in prediction (33 a), for instance, shows a ground truth mask shifted southwards relative to the apparent north edge of the roof. Prediction (33 a) produces a relatively good segmentation mask under the circumstances, remarkably predicting the north edge of the roof quite accurately. This correction of perspective is highly intentional, as it is not the pixels themselves we would like to segment per se, but rather the *geographic location* of the building outline.

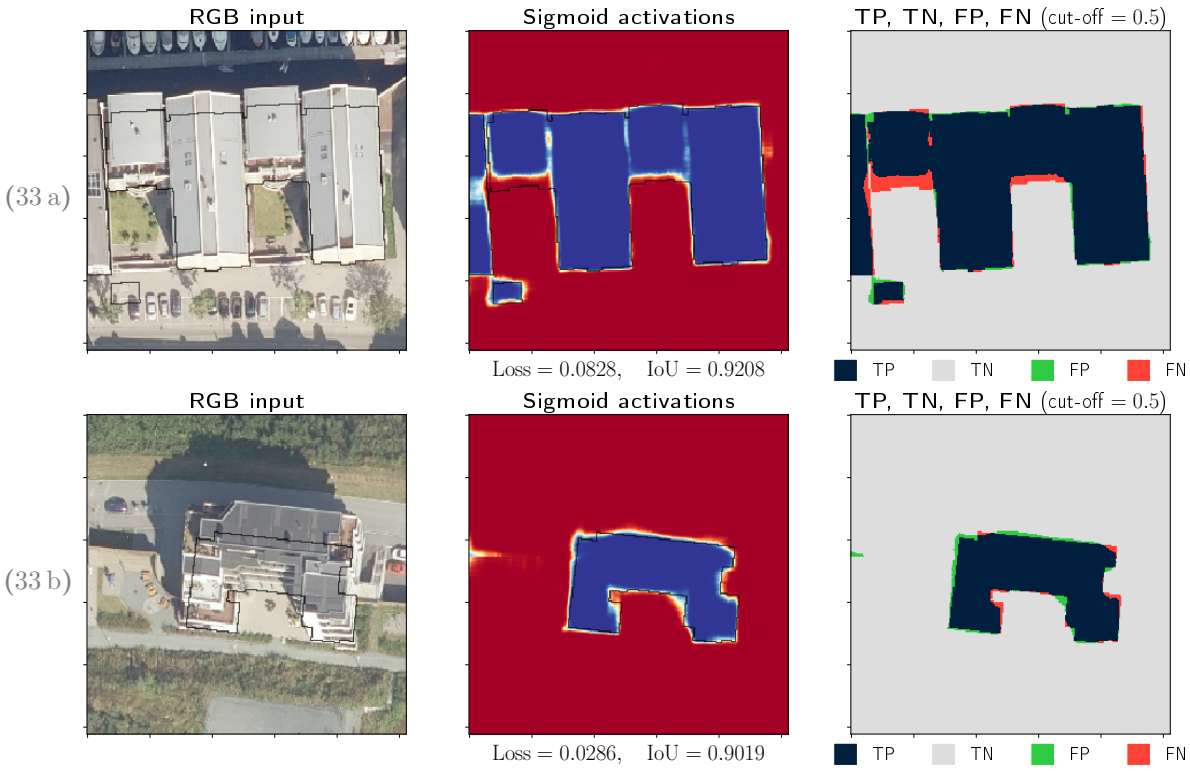


Figure 33: RGB model predictions on images with a high degree of perspective misalignment.

LiDAR data

A model using only LiDAR data is trained, and the training procedure is summarized in Figure 34.

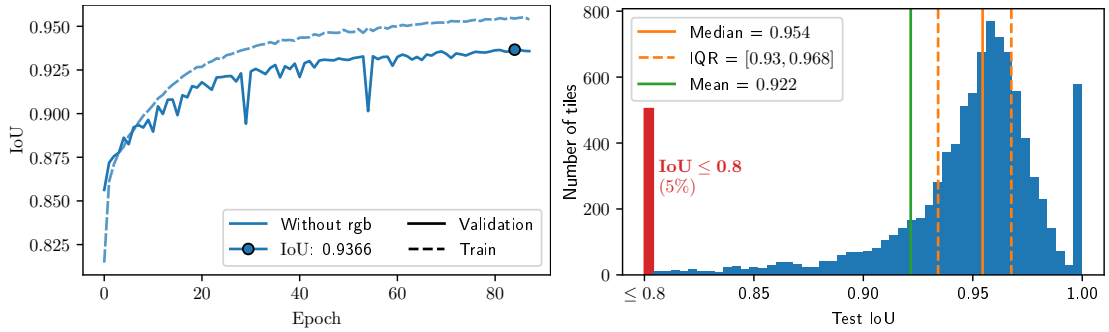


Figure 34: Left – IoU evaluations during training of the LiDAR U-Net model for 89 epochs. **Right** – Test IoU distribution of LiDAR model, left 5 % of data cropped. See caption of Figure 28 for detailed description.

The LiDAR model performs better than the RGB model in all observed aggregate performance measures. Replacing RGB data with LiDAR data increases the mean test IoU from 0.900 to 0.922 and the median test IoU from 0.932 to 0.954, for instance. The number of negative outliers in the test set, that is tiles with IoU ≤ 0.8, also decreases from 7% to 5%. As with the LiDAR model we present the median performing prediction in the test set in Figure 35.

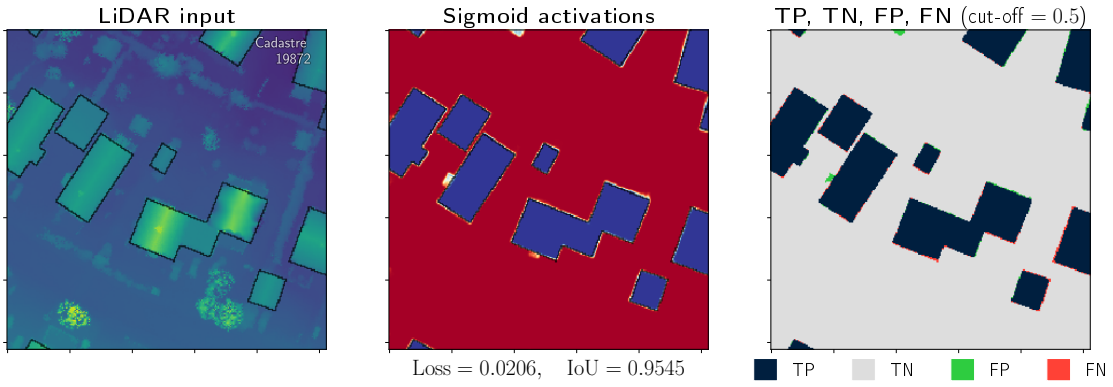


Figure 35: LiDAR model median IoU prediction from the test set. The left panel shows the single-channel LiDAR input provided to the model. The remaining two panels are identical to previous prediction plots; see caption of Figure 29 for detailed explanation.

The LiDAR model has a comparative advantage over the RGB model in that it manages to predict straight edges with a greater degree of confidence and accuracy. Whenever the LiDAR models fails it often includes and/or excludes a "well-defined" region and as a result still produces properly shaped building outlines. The erroneous inclusion of an extended roof overhang over a front door or the exclusion of a small and low building annex are two examples of commonly observed "well-behaved failures".

While it has been established that the LiDAR model outperforms the RGB model *in aggregate*, it is still of interest to compare these two models on a more case-by-case basis. The two models are compared tile-by-tile in the IoU scatter plot presented in Figure 36.

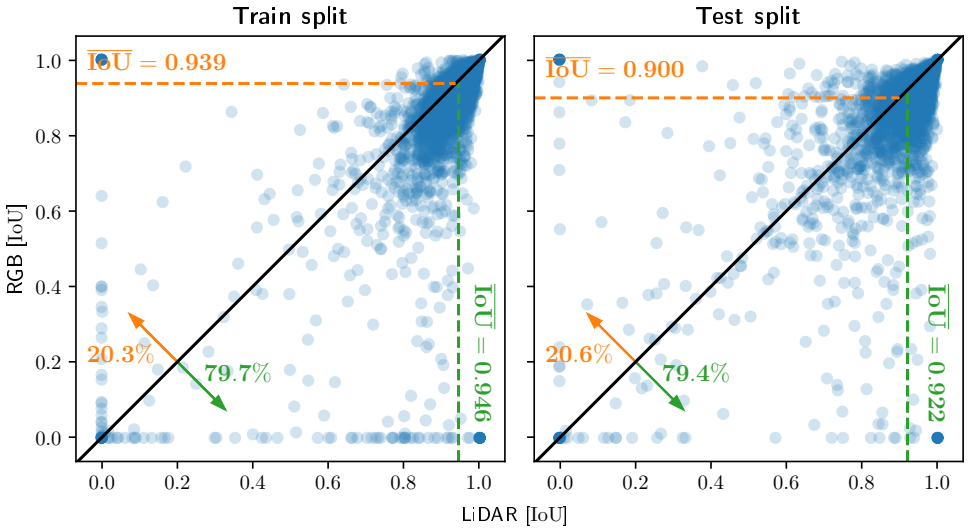
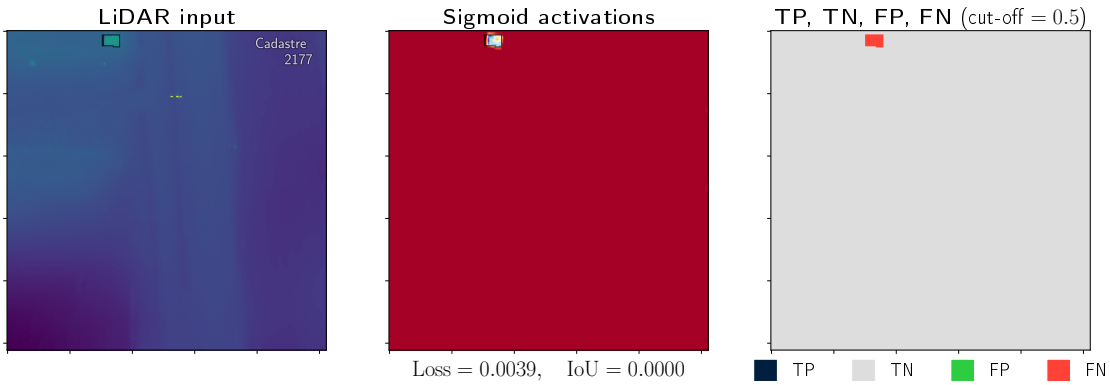


Figure 36: Scatter plot showing the correlation between the evaluation metric performance of two models using different data, LiDAR vs. RGB. Each blue scatter point (x_i, y_i) corresponds to a given tile, i, where the x-coordinate is the IoU metric of the LiDAR model prediction and the y-coordinate is the IoU metric of the RGB model prediction for that given tile. Tiles belonging to the train split are shown in the left half while the tiles belonging to the test split are shown in the right. The horizontal dashed lines in orange indicate the *mean* IoU metric of the RGB model for the respective splits, while the vertical dashed lines in green indicate the *mean* IoU for the LiDAR model. Diagonal black lines indicates $x = y$, and the arrows with accompanying percentages indicate the fraction of points above and below this line. Scatter points located *above* the black diagonal line indicate tiles where the RGB model performs better than the LiDAR model, while scatter points located *below* the diagonal represent tiles where the LiDAR model performs better than the RGB model.

If the RGB and LiDAR models would have been indistinguishable w.r.t. predictive performance then the scatter points would be entirely situated along the diagonal black lines in Figure 36, which is clearly not the case here due to the LiDAR model outperforming the RGB model. While LiDAR is on average better than RGB, RGB still outperforms LiDAR in about 21 % of the test cases. This may be partly caused by the randomness introduced into the training procedure, and thus the final model parametrization, but may also be an indication of RGB containing predictive information that LiDAR does not possess in certain cases. If this is the case, then a combined data model which uses both LiDAR *and* RGB might outperform both of these single data source models.

(37 a)



(37 b)

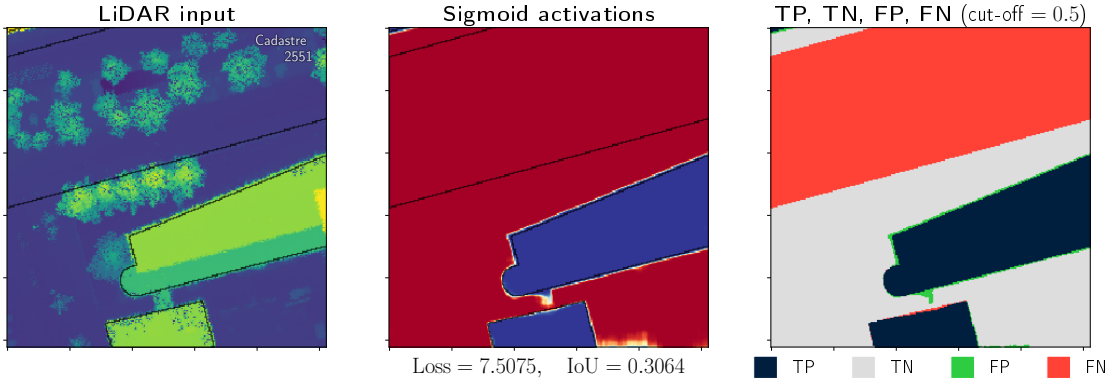


Figure 37: LiDAR model prediction on same input as presented in Figure 30.

Figure 37 presents the predictions produced by the LiDAR model over the same geographic area as the RGB model predictions presented in Figure 30. The LiDAR model predictions, (37 a) and (37 b), demonstrates the same issues as seen with the RGB model, namely vanishingly small segmentation masks and erroneous ground truths. This comes as no surprise, especially since these issues are considered unrelated to the intrinsic properties of RGB data. What about the prediction issues exemplified in Figure 32 which were considered specific to RGB data; are these issues remedied by the LiDAR model? Figure 38 presents the LiDAR model predictions over the same geographic area as used by RGB model predictions presented in Figure 32.

(38 a)

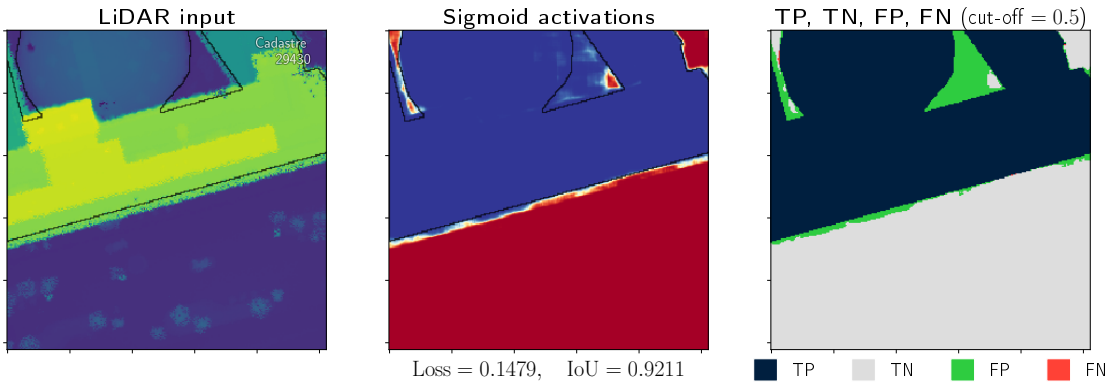


Figure 38: Continued on next page...

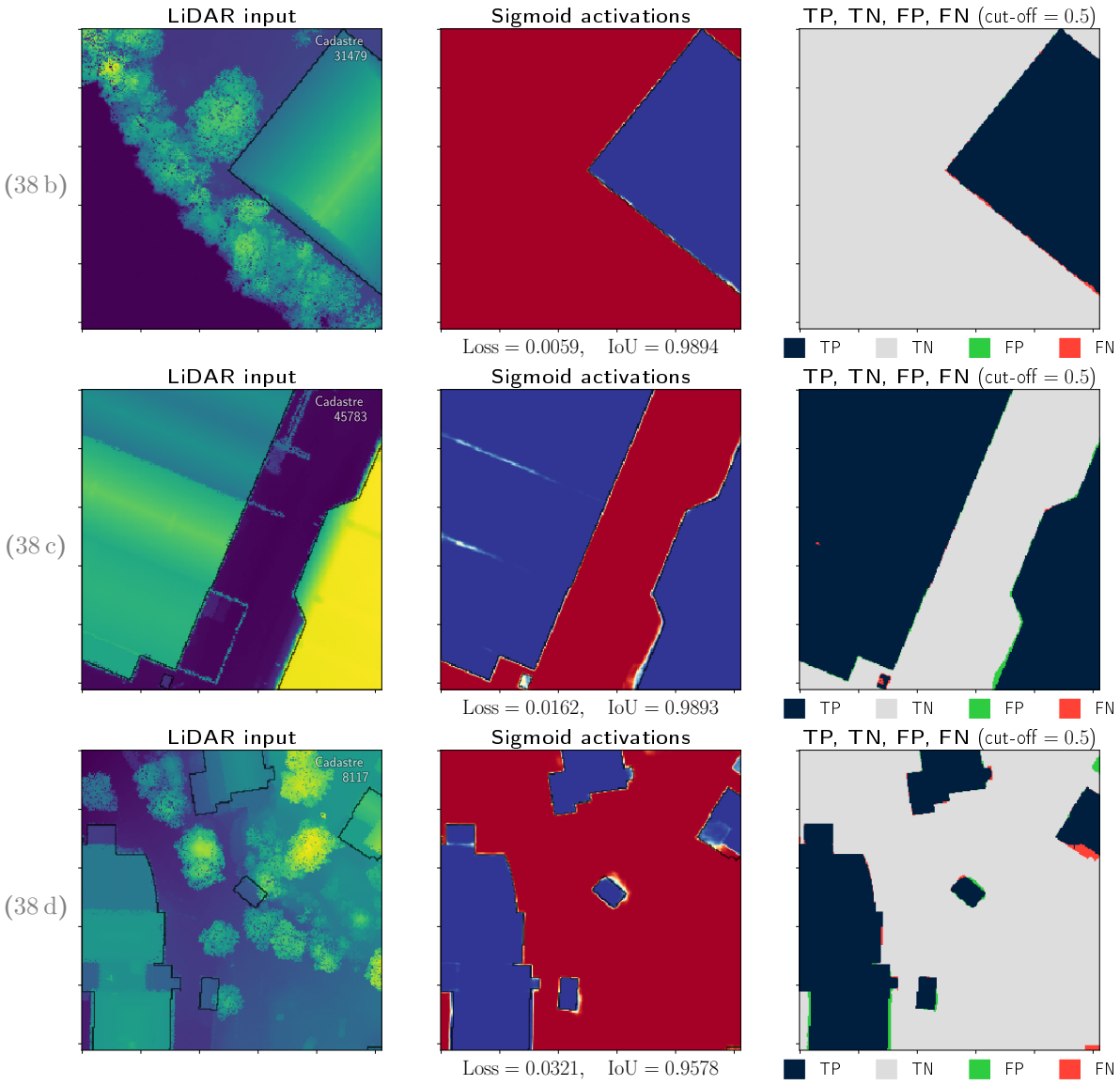


Figure 38: LiDAR model prediction over same geographic area as used in Figure 32.

As can be seen in Figure 38, the texture and contrast issues observed in Figure 32 have been largely corrected in the LiDAR model predictions, although prediction (38 a) still has some errors.

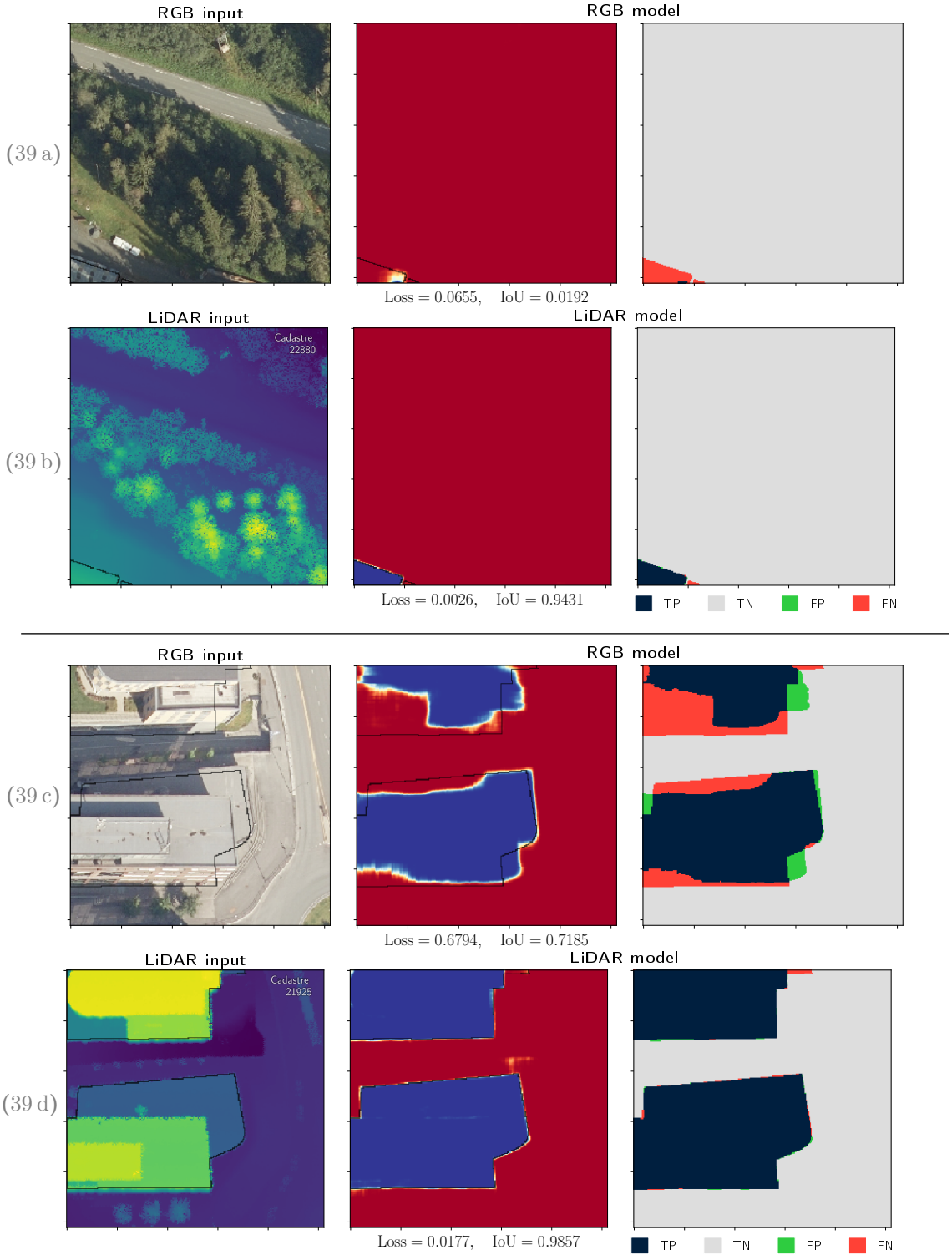


Figure 39: Geographic test tiles where the LiDAR model performs much better than the RGB model. The top half shows the greatest improvement when going from RGB to LiDAR, while the bottom half shows the next best improvement for the geographic tiles with above-average building density.

Figure 39 presents two test cases where the LiDAR model outperforms the RGB model to a large degree. The LiDAR model generally performs better than the RGB model when encountering building outlines situated along the borders of the input tiles, usually requiring less spatial context before being able to distinguish buildings. RGB prediction (39 c) suffers from three common RGB issues: contrasts, roof greenery, and non-orthogonal perspective, while LiDAR prediction (39 d) suffers from none of these issues.

As can be seen in Figure 36 there are certain test cases where the RGB model outperforms the LiDAR model to a significant degree, two such cases being presented in Figure 40. Prediction (40 b) demonstrates how good the RGB model is in detecting the presence of orange roof tiles, no matter how small the area, while LiDAR prediction (40 a) faces difficulty due to the dense greenery. LiDAR prediction (40 c) seems to have too little context in order to determine what is ground level and what is not, while RGB prediciton (40 d) manages much better, probably due to the typical roof texture.

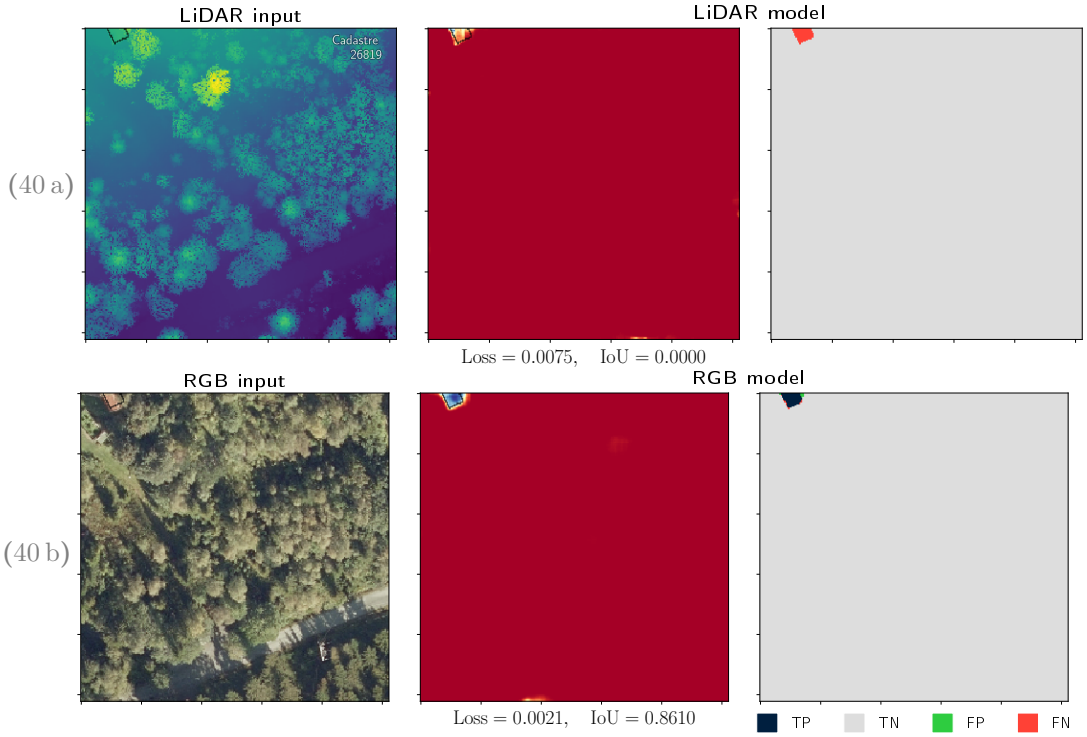


Figure 40: Continued on next page...

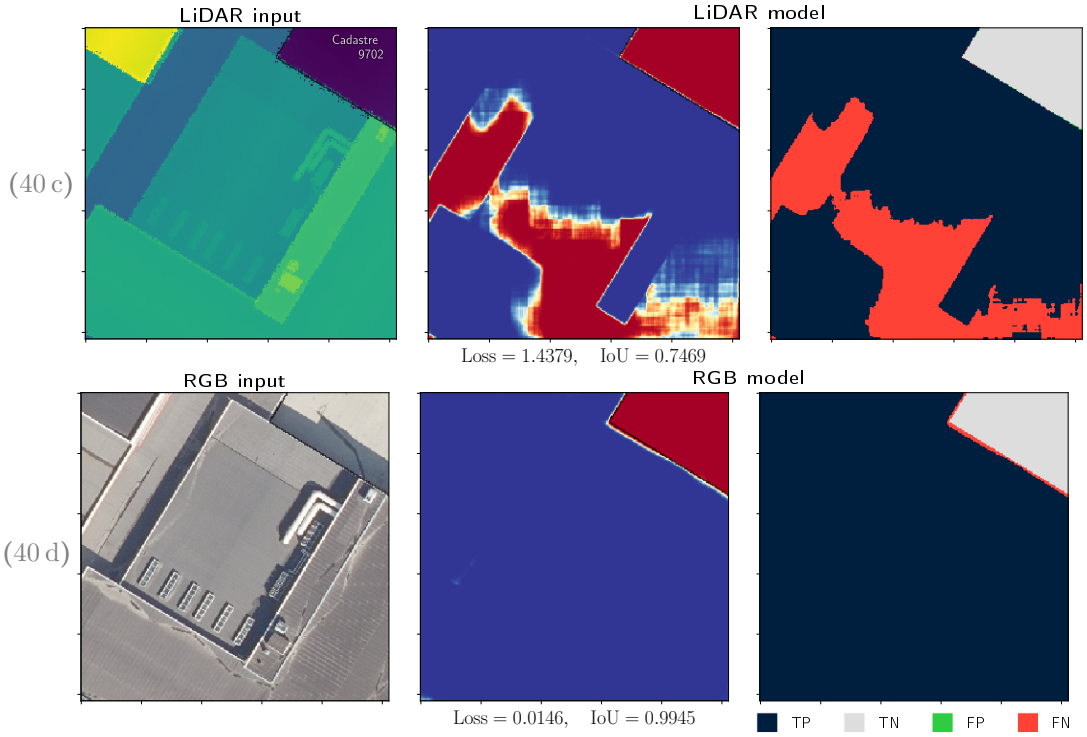


Figure 40: Geographic test tiles where the RGB model performs significantly better than the LiDAR model.

Combined data

We now construct a model which uses both LiDAR and RGB data in combination in order to produce predictions. The training procedure of the combined data model is shown in Figure 41, and the training procedure of the LiDAR model has been included for comparison.

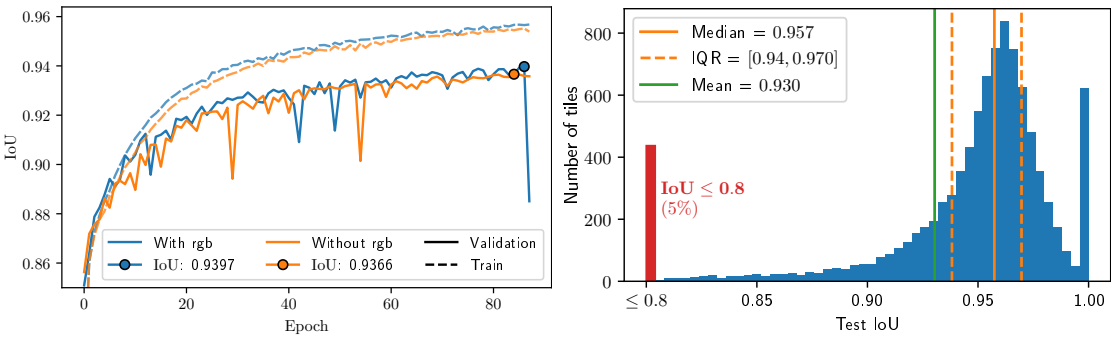


Figure 41: Left – IoU evaluation during training of U-Net models for 89 epochs. Blue indicates the model using just LiDAR data, while orange is used to indicate the combined data model (RGB + LiDAR). **Right** – Test IoU distribution of combined data model, left 5 % of data cropped. See caption of Figure 28 for detailed description.

The median performing test prediction is shown in Figure 42.

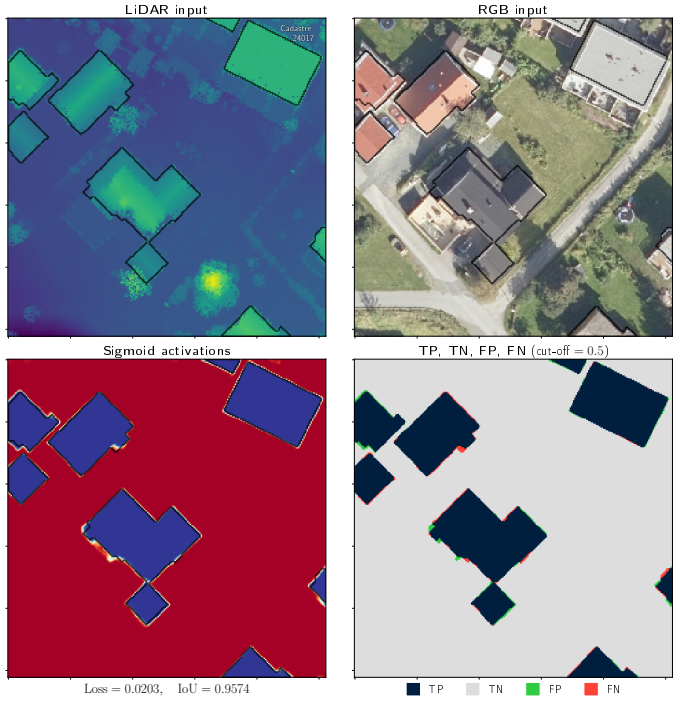


Figure 42: Median IoU prediction from the test set using both remote sensing data types, RGB and LiDAR.

Most predictions produced by the combined model show the same behaviour and general weaknesses as the model based solely on LiDAR data. Although the LiDAR model outperformed the RGB model in every aggregate metric, indicating that RGB is a worse predictor than LiDAR, using RGB *in addition* to LiDAR as input data still increases the test performance of the resulting model. When adding RGB to to LiDAR model, the mean IoU on the test set improves from 0.934 to 0.938, not an insignificant improvement. A tile-by-tile comparison of the LiDAR model and the combined data model, similar to the comparison presented in Figure 36, is shown in Figure 43. The combined data model outperforms the LiDAR model in 61.2 % of all test cases.

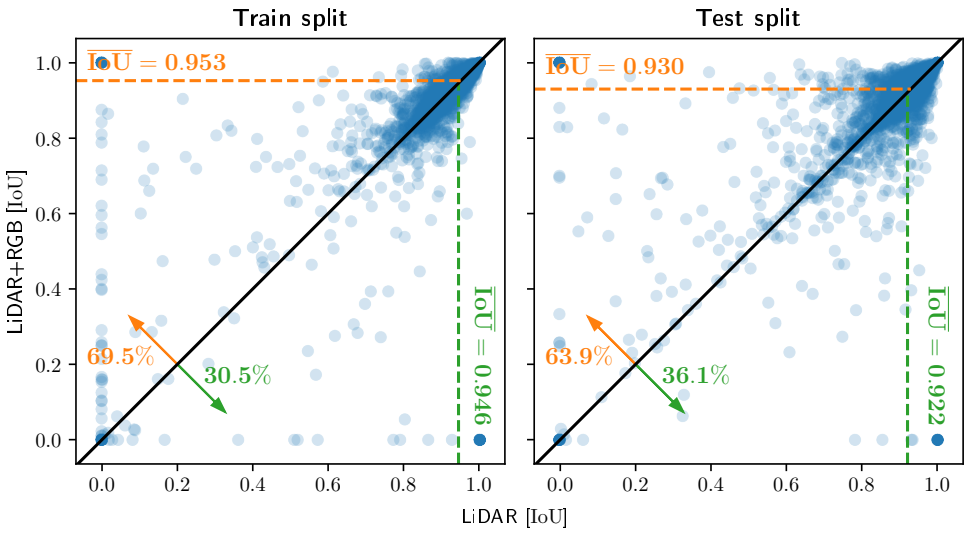


Figure 43: Scatter plot showing the correlation between the evaluation metric performance of two models, LiDAR vs. RGB + LiDAR. The combined data model is shown along the vertical axis, while the model using just LiDAR data is shown along the horizontal axis. See caption of Figure 36 for detailed figure explanation.

In what way does the combined data model outperform the LiDAR model? The combined model usually produces predictions almost identical to the LiDAR model, but in the minority of the cases where the RGB model performs *better* than the LiDAR model, the combined model seems to mimic the RGB model rather than the LiDAR model. Previously, in Figure 40, we presented two test tiles where the RGB model evaluated *better* than the LiDAR model. The combined data model predictions on the same test tiles are presented in Figure 44.

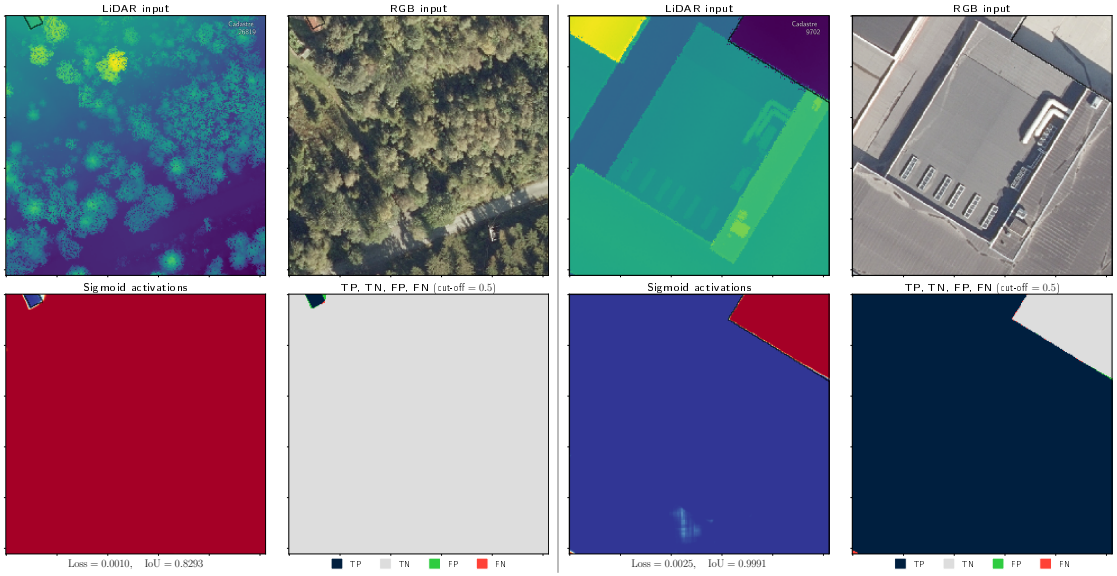


Figure 44: Predictions on test tiles using the model trained on both LiDAR data and RGB data in combination. The individual data model predictions using the same tiles are given in Figure 40.

The observed pattern is that the combined data model usually imitates the LiDAR-only model, only deviating whenever the RGB data is more descriptive than the LiDAR data. Figure 45 substantiates this interpretation of how the combined data model improves upon the LiDAR model.

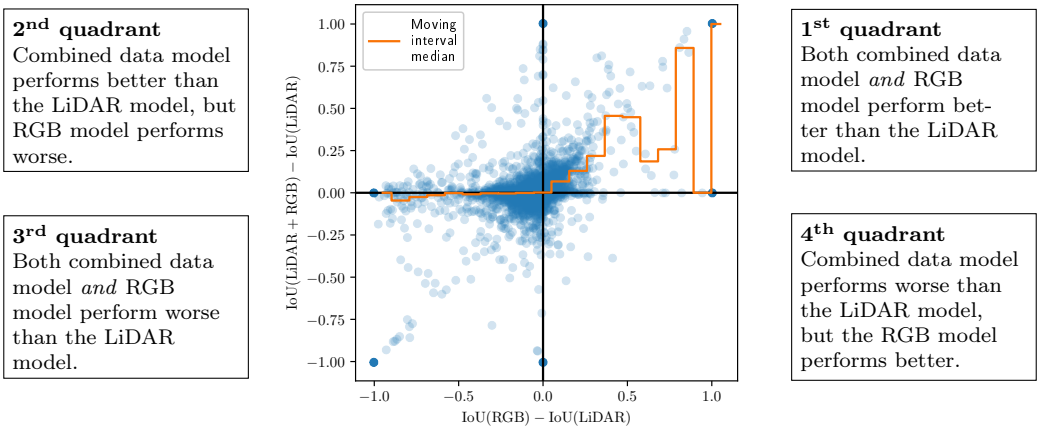


Figure 45: Scatter plot of points (x_i, y_i), one point for each test tile, i. The x-coordinate is the difference between the RGB model's IoU evaluation and the LiDAR model's IoU evaluation, i.e. how much better RGB performs than LiDAR. The y-coordinate is the difference between the combined data model's IoU evaluation and the LiDAR model's evaluation, i.e. how much better the combined data performs compared to using LiDAR in isolation. The median y-coordinate has been calculated for non-overlapping bins of width $\Delta x = 0.1$ and have been annotated in orange.

Each scatter point in Figure 45 represents a geographic tile in the test split, and the coordinates (x_i, y_i) are derived from linear combinations of the IoU metrics of the LiDAR model, RGB model, and the combined data model. The x-coordinate is the difference between the IoU of the RGB model prediction and the IoU of the LiDAR model prediction. In other words, whenever $x > 0$, the RGB model performs better than the LiDAR model. The y-coordinate is the difference between the combined data model's IoU metric and the LiDAR model's IoU metric. Whenever $y > 0$, the combined data model performs better than the model based on just LiDAR data. Consider a hypothetical model which is able to omnisciently select between the RGB and LiDAR model based on which maximizes the resulting evaluation performance; such a model would produce scatter points according to the relation $y = \max(0, x)$ in Figure 45. The combined data model does in fact produce predictions somewhat in accordance to this relation as illustrated by the "moving median" in Figure 45.

3.3 Regularization Techniques

In previous sections we have presented techniques such as batch normalization, dropout, and data augmentation, and claimed that these techniques have been empirically shown to combat overfitting and/or decrease training times. Unfortunately, machine learning techniques are often highly context dependent with respect to their efficiency, and this section is therefore intended as a verification of these techniques in the context of remote sensing data and building footprint segmentation.

Batch normalization

We have trained two U-Net models on LiDAR data, one *with* batch normalization and one *without*, the training procedure of both these models being presented in Figure 46.

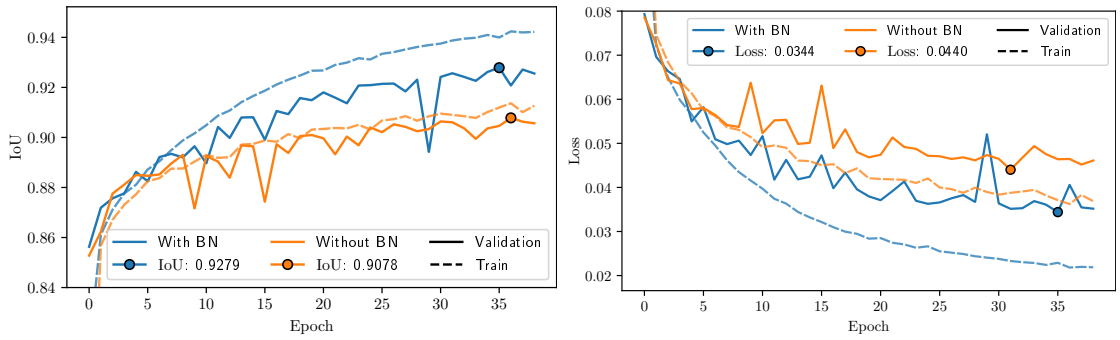


Figure 46: Training procedure of two U-Net models on LiDAR data, one employed with batch normalization shown in blue, while the other has no batch normalization and shown in orange. Left panel shows the IoU metric evaluations, while the right panel shows the binary cross-entropy loss.

The comparative performance improvement of the batch normalized model over the model without batch normalization becomes immediately clear from Figure 46. What is of particular interest is that the batch normalization does not only increase the speed of optimizing the loss function, but it also improves the final model performance in form of validation IoU. Of all the A/B tests conducted in this section, this test has had the most significant effect.

Dropout

As with the batch normalization experiment, we now train one model *with* max-pooling dropout and one *without*, and the training results are presented in Figure 47.

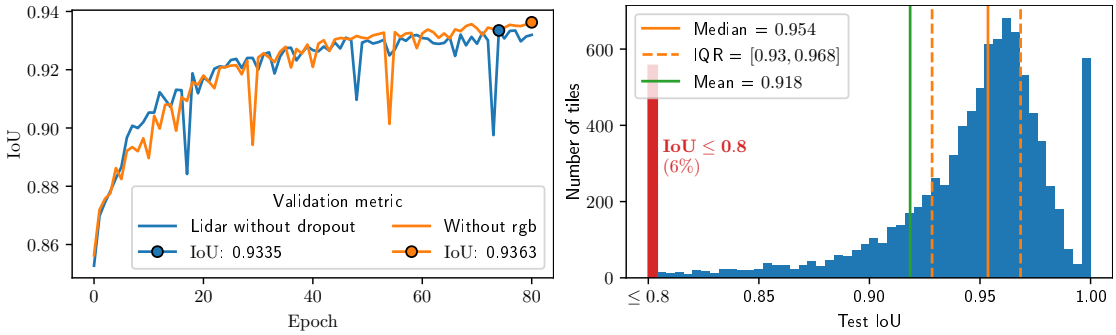


Figure 47: Left – Training procedure of two U-Net models on LiDAR data, one employed with max-pooling dropout shown in blue, while the other uses no max-pooling dropout and is shown in orange. **Right** – Test IoU distribution of LiDAR model without dropout, left 6 % of data cropped. See caption of Figure 28 for detailed description.

While the improvement of batch normalization was immediately obvious, the effect of dropout is more difficult to interpret from the left panel of Figure 47 alone as we only observe a marginal IoU validation metric improvement from 0.9335 to 0.9363. On the other hand, the right panel of Figure 47 shows that the interquartile range and median of the test IoU metrics are identical across the two models, but that the right-end tail of the distribution has grown thicker. Methods preventing overfitting will hypothetically bridge the gap between the training- and test-evaluation of the model. As dropout is primarily intended as a measure to prevent overfitting, we will investigate the performance of both models on the training set and compare this with their performance on the test set. Figure 48 presents a comparison of these two models, a plot similar to the one shown in Figure 36.

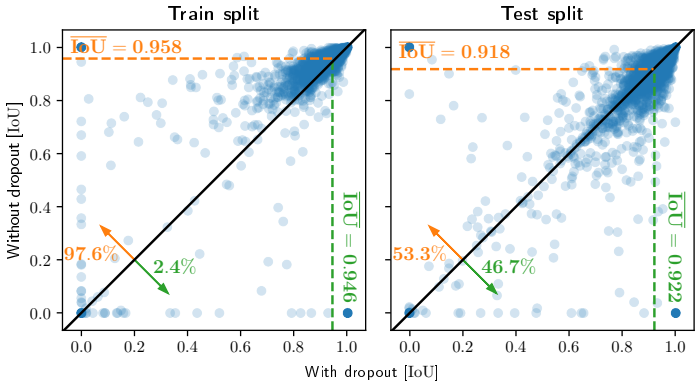


Figure 48: Scatter plot showing the correlation between the evaluation metric performance of two models, one with max-pooling dropout and one without. The model without dropout is shown along the vertical axis, while the model with dropout is shown along the horizontal axis. See caption of Figure 36 for detailed figure explanation.

The U-Net model *without* dropout largely outperforms the model with dropout on the *training* set, having a better IoU evaluation on about 98% of all training tiles and a mean training IoU of 0.958, which is substantially better than the dropout model with a mean training IoU of 0.946. What is of particular interest, though, is that the dropout model generalizes *much* better to the test tile set, so much so that it narrowly outperforms the model without dropout on the test set. The 98% / 2% split is reduced to a more even 53% / 47% split, making the two models approximately tied. The mean test IoU of the dropout model, 0.922, is also better than the mean test IoU of the non-dropout model, 0.918. Altogether this can be considered quite strong evidence in favor of max-pooling dropout having reduced the overfitting of our model. The application of dropout during model training likely increases the generalizability of our model as it improves its ability to predict building outlines from previously unseen remote sensing data.

Data Augmentation

Finally we investigate the effect of data augmentation when training the LiDAR model, comparisons of the two models being presented in Figure 49. The data augmentation consists of random application of horizontal and/or vertical flipping in addition to a rotation by a random multiple of 90 degrees, resulting in altogether 16 random configurations of each training tile.

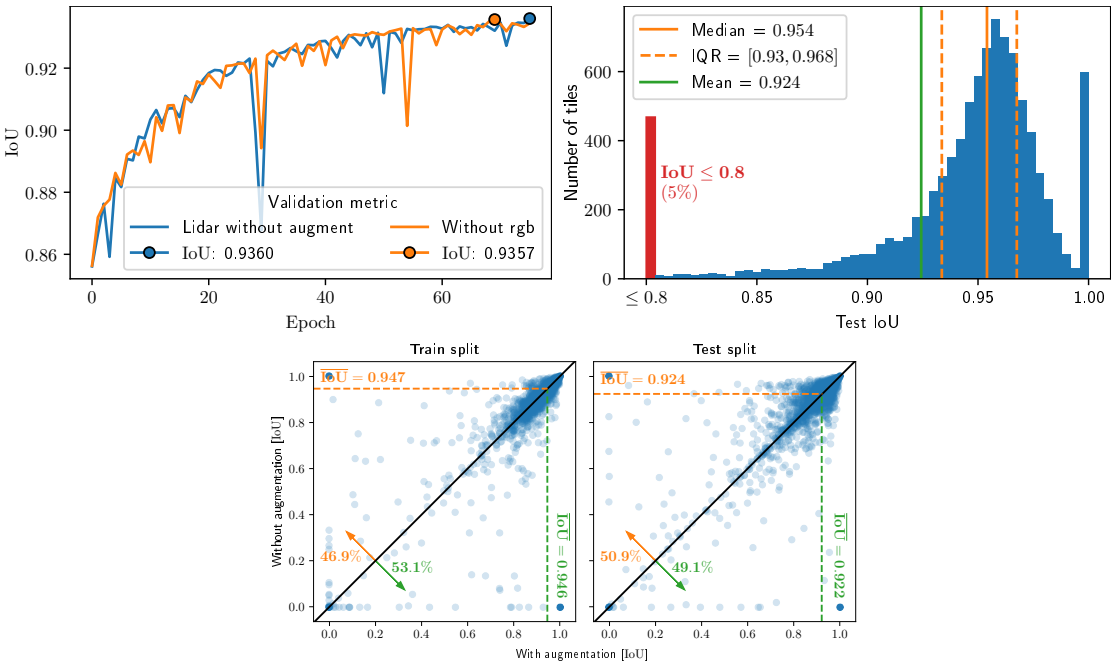


Figure 49: Top left – Training procedure of two U-Net models on LiDAR data, one employed with data augmentation shown in blue, while the other uses no data augmentation and is shown in orange. **Top right** – Test IoU distribution of model trained without data augmentation, left 5 % of data cropped. See caption of Figure 28 for detailed description. **Bottom** – Scatter plot showing the correlation between the evaluation metric performance of two models, one with data augmentation (horizontal) and one without (vertical). See caption of Figure 36 for detailed figure explanation.

No significant difference between the two training schemes can be observed in Figure 49. The observed differences between the two resulting models can as likely be attributed to random noise than any causal effect of the data augmentation procedure. The augmentation techniques applied during training, namely flipping and 90 degree rotations, have intentionally been selected in order to be negligible in computational cost. They do not require any de facto calculations as they can be implemented by simply traversing the memory layout in a different manner, and are therefore constant $\mathcal{O}(1)$ time cost operations. Considering that data augmentation is cheaply performed and we have no evidence contrary to it being a positive influence of the generalizability of the model, we conclude that data augmentation should be performed during training. Finally, it is worth noting that the data augmentations applied in this case are rather simple and minor forms of data augmentation. More major augmentation forms might have a bigger effect.

3.4 LiDAR Normalization

We will now investigate how the normalization of LiDAR raster elevation values influences the predictive performance of the resulting model. The "`nodata`-aware local min-max normalization" method described in Algorithm 2 on page 32 will be simply referred to as "dynamic scaling" as it scales each elevation tile individually to the $[0, 1]$ domain. Likewise, the "`nodata`-aware metric normalization" method described in Algorithm 3 on page 33 will be simply referred to as "constant scaling" as it always scales by a constant factor γ^{-1}. The training procedures of two models employing these two different LiDAR normalization methods are shown in Figure 50. The global scaler γ, as specified in Algorithm 3, is chosen to be 30. Small values for this global scaler, $\gamma < 10$, results in no training convergence whatsoever, while large enough values have been shown to not differ significantly in performance.

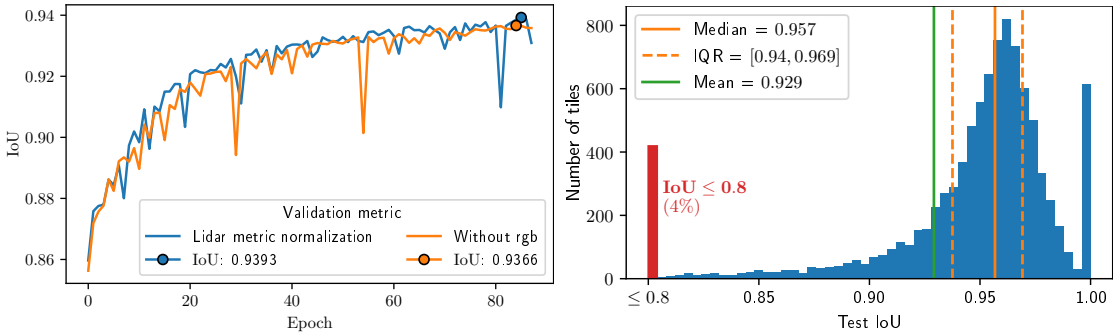


Figure 50: Left – Training procedure of U-Net LiDAR model using two different normalization methods. The model using "dynamic scaling" as specified in Algorithm 2 is shown in blue, while the model using "constant scaling" with $\gamma = 30$ as specified in Algorithm 3 is shown in orange. **Right** – Test IoU distribution "dynamic scaling" model, left 4 % of data cropped. See caption of Figure 28 for detailed description.

Figure 50 shows a small improvement of using the constant scaling over the dynamic method, an improvement of validation IoU from 0.9366 to 0.9393, although not much can be concluded from this figure alone. A comparison over the training and test splits of the two models is presented in Figure 51.

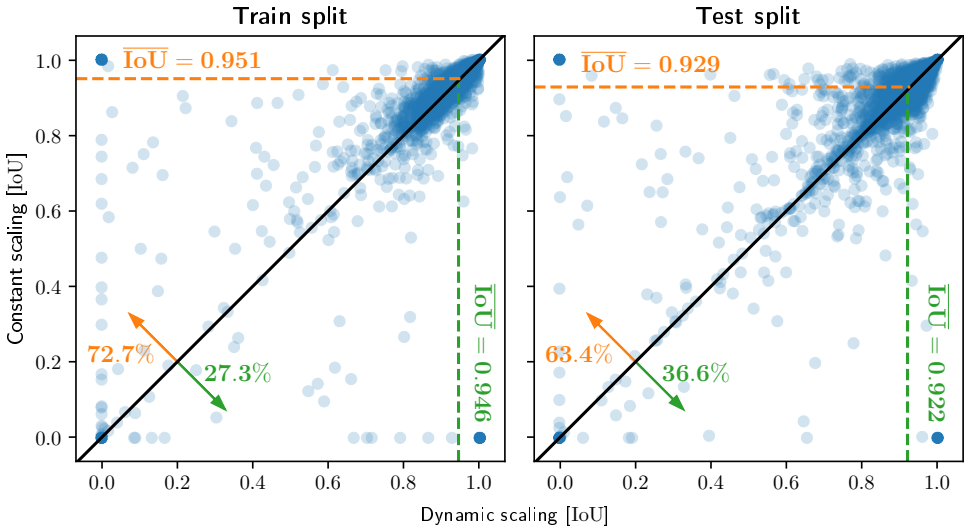


Figure 51: Scatter plot showing the correlation between models using different LiDAR normalization methods, IoU of the "constant scaling" model annotated along the vertical axis while the IoU of the "dynamic scaling" model is annotated along the horizontal axis. See caption of Figure 36 for detailed figure explanation.

The "constant scaling" normalization method does in fact perform better than the "dynamic scaling" normalization method over all three sample splits: training, validation, and test. When the normalization method is changed to the constant scaling method, the mean test IoU metric improves from 0.922 to 0.929, and 63% of the test cases perform better. Some of the test cases where the constant scaling model performs better are tiles containing large, flat areas that are *not* part of a roof, for example cadastral plots situated along the shoreline as presented in Figure 52. Although we have established that constant scaling is an overall improvement over dynamic scaling, we have not been able to identify any further problem characteristics where constant scaling generally performs better. It was hypothesized that constant scaling would outperform dynamic scaling whenever the elevation range within a given tile became very small or very large, this being due to the lossy compression of dynamic scaling forcing all elevation values into the $[0, 1]$ value range. Such an effect has not been observed. The largest improvement of the constant scaling model over the dynamic scaling model has largely been due to an improvement in recall from 88.92 % to 89.33 %, while precision only improved from 89.21 % to 89.22 %.

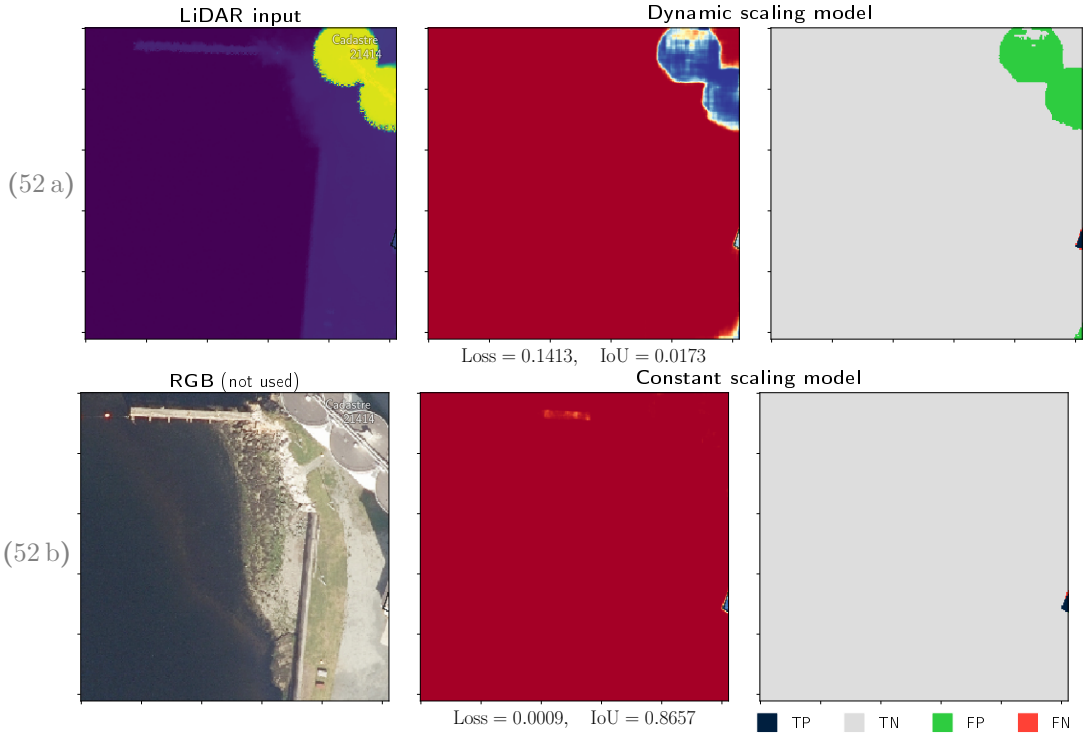


Figure 52: Geographic test tiles where the constant scaling model (top) performs much better than the dynamic scaling model (bottom).

3.5 Losses

So far all model experiments have been exclusively trained by optimizing the *binary cross-entropy* loss (BCEL) function given in equation (1). While BCEL is by far the most popular loss function for binary classification tasks, it is still considered a suboptimal surrogate loss function for segmentation metrics such as IoU. Alternative loss functions were discussed in Section 1.3, the so-called soft loss variants being of greatest interest. The *soft Jaccard loss* given in equation (2) and *soft dice loss* given in equation (3) have specifically been shown to be efficient surrogate loss functions for the IoU metric, both theoretically and empirically. Three models have been trained and evaluated for this numerical experiment, the only difference being which loss function that has been used during training: binary cross-entropy, soft Jaccard loss, or soft dice loss. The training procedures of these three models are visualized in Figure 53. The soft dice loss model is almost identical to the soft Jaccard loss model in behaviour and performance, so in order to avoid repetitiveness we will mainly compare the BCEL model to the soft Jaccard model and ignore the soft dice model.

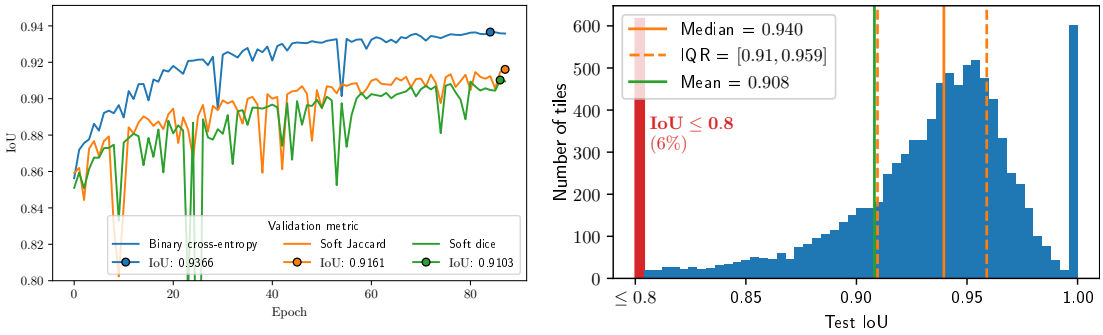


Figure 53: Left – Three U-Net LiDAR models trained with different loss functions. Binary cross-entropy model shown in blue, soft Jaccard in orange, and soft dice shown in green. **Right** – Test IoU distribution of soft Jaccard model, left 6 % of data cropped. See caption of Figure 28 for detailed description.

The three models presented in the left panel of Figure 53 start out at approximately the same point after one epoch, but the binary cross-entropy model quickly outperforms the two other models when it comes to mean validation IoU. The same can be said of the test IoU metrics of the soft Jaccard model as presented in the right panel of Figure 53, the soft loss model being a performance regression over the BCEL model in every conceivable way. The soft losses seem to be *worse* surrogate losses for the IoU metric rather than better ones, completely contradicting our prior beliefs.

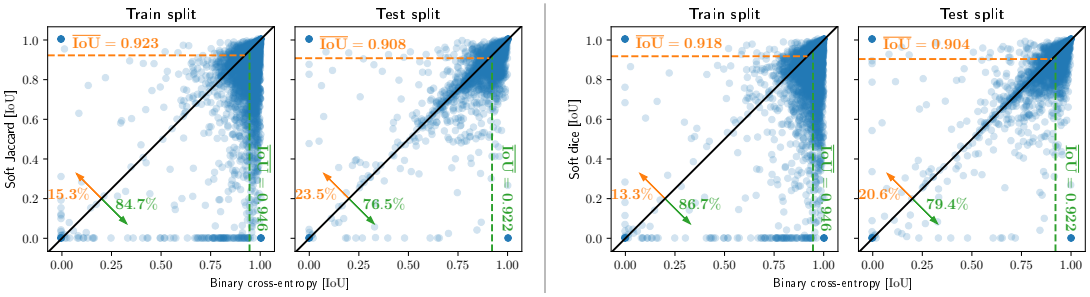


Figure 54: Scatter plot showing the correlation between models using different losses during training. Both the left and right half of this figure compares model IoU against the binary cross-entropy IoU along the horizontal axis. Left figure half shows the soft Jaccard model on the vertical axis, while the right half shows the soft dice loss along the vertical axis. See caption of Figure 36 for detailed figure explanation.

Figure 54 shows that the soft models perform worse than the BCEL model on the test set. Of greater interest, however, is the large discrepancy between how the BCEL model and the soft models perform on the *training* set. There are certain training tiles where the soft models are not able to learn from the labeled data at all, while the BCEL model has no such difficulties. When inspecting these cases, they are usually characterized by one of two properties:

1) bad data used as ground truth, or…
2) exceptionally difficult problems in the form of vanishingly small building outlines.

Both these cases are illustrated in Figure 55.

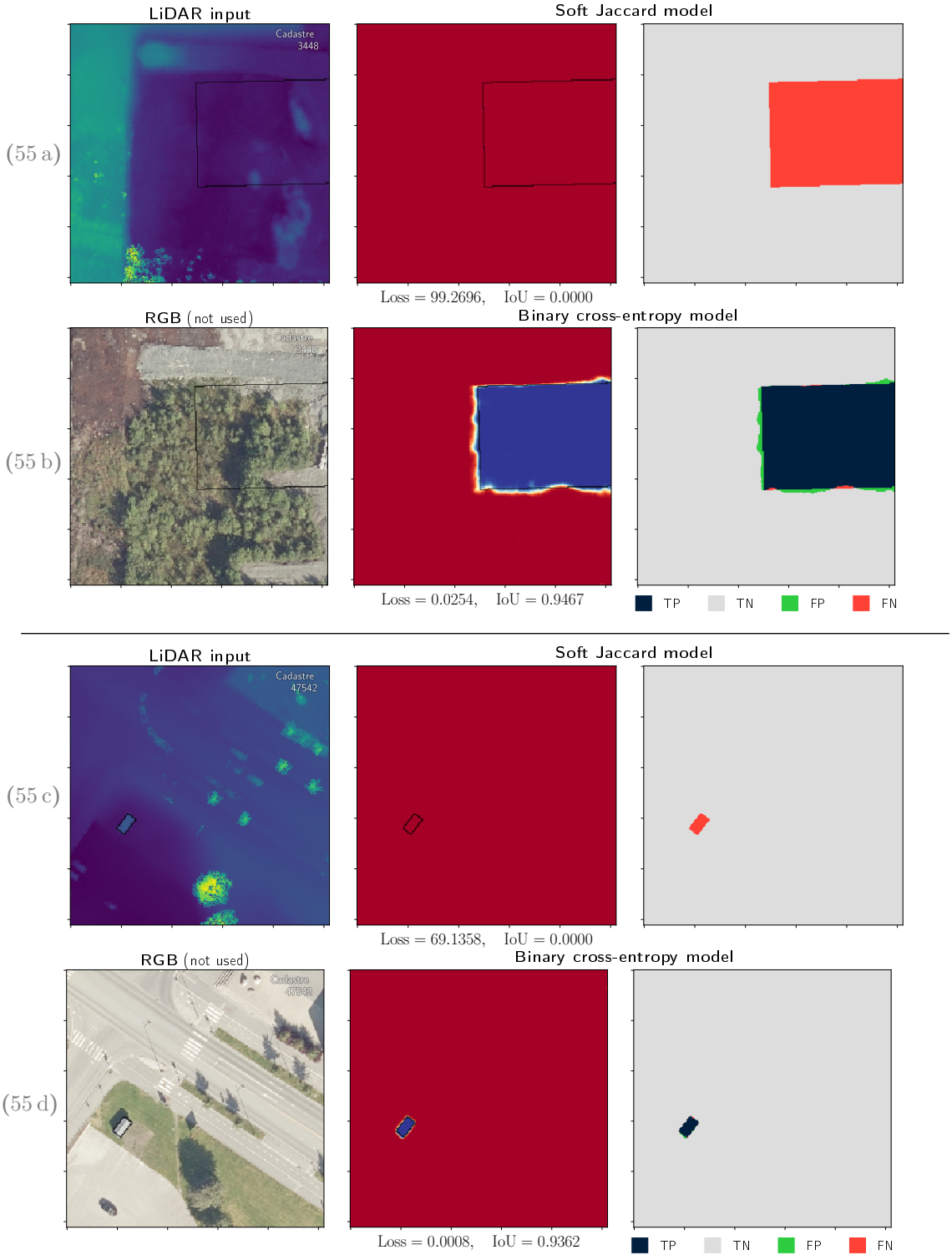


Figure 55: *Training* tiles where the BCEL model performs substantially better than the soft Jaccard model.

The regression in performance of the soft Jaccard model on the training set translates to a worse test performance as well, although to a lesser degree. Interestingly, whenever the soft Jaccard model performs worse than the BCEL model on test cases, it usually produces "well-behaved" failures, failures that could conceivably be made by humans as well. Such failures are presented in Figure 56. In the opposite case, whenever the soft Jaccard model performs better than the BCEL model, it is usually due to the BCEL model having made "badly behaved" failures, failures that would never have been produced by a human. The BCEL model does in certain cases produce egregiously bad false positives. This can conceivably be caused by the willingness of the BCEL model to learn from clearly wrong ground truth masks in the training set. Such false positives are in certain cases corrected by the soft Jaccard model, as presented in Figure 57.

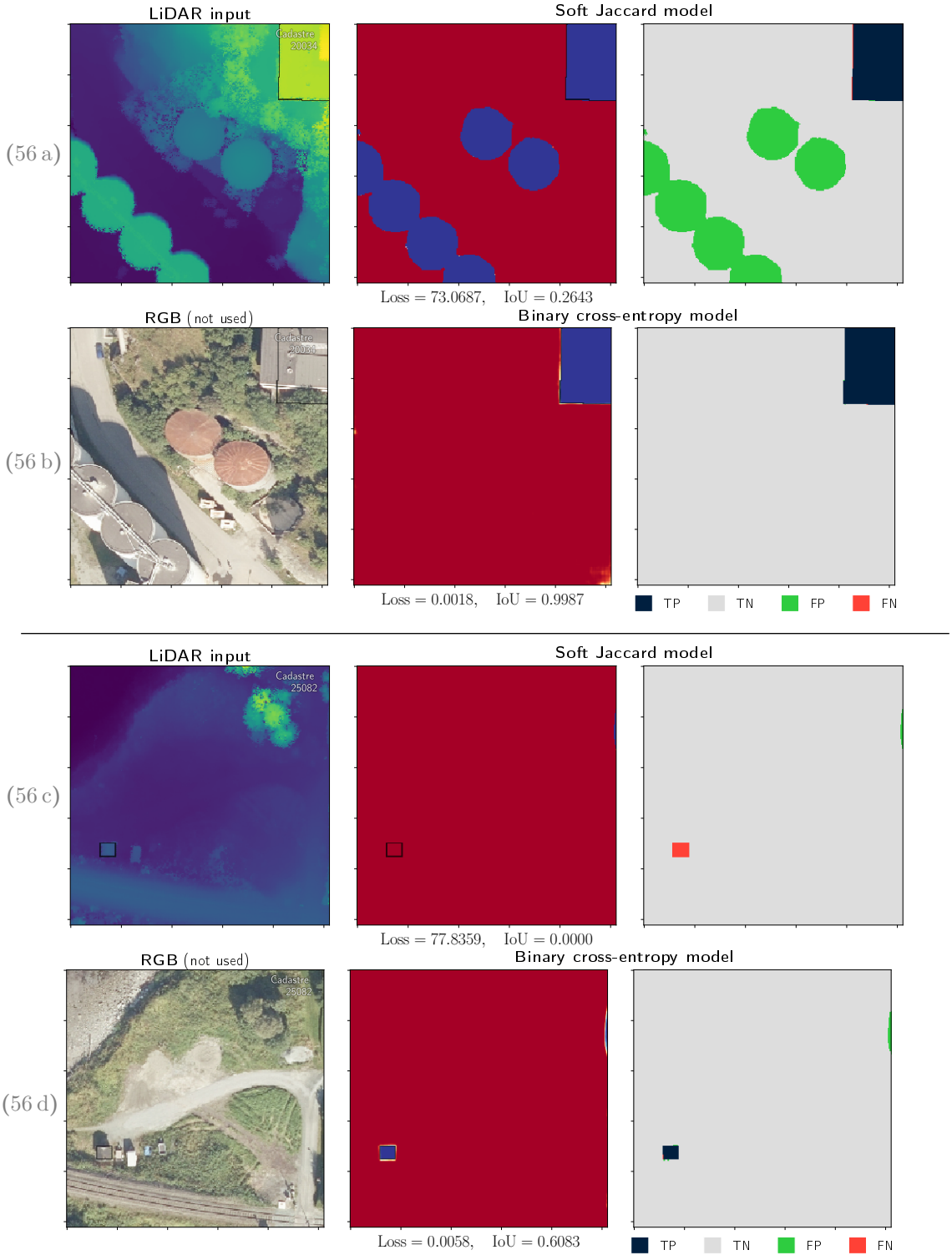


Figure 56: *Test* tiles where the BCEL model performs substantially better than the soft Jaccard model.

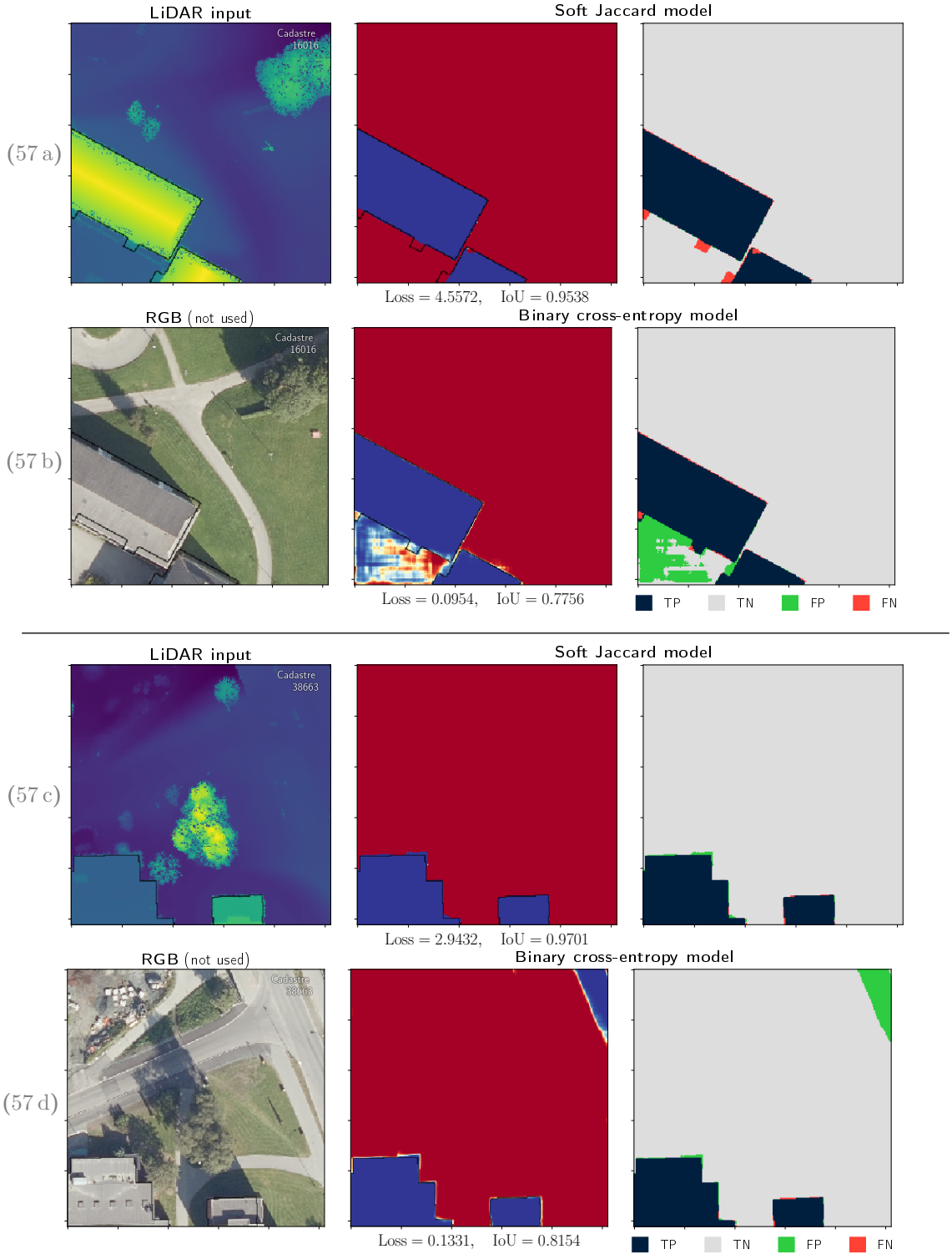


Figure 57: *Test* tiles where the soft Jaccard model performs substantially better than the BCEL model.

Conclusion and Further Work

A test performance summary of some of the model experiments from Section 3 is presented in Table 4.

Model	IoU	Accuracy	Precision	Recall
RGB	0.9005	98.67%	87.85%	87.26%
LiDAR	0.9216	99.03%	89.21%	88.92%
Combined	0.9304	99.12%	89.30%	89.29%
Constant scaling	0.9292	99.10%	89.22%	89.33%
Soft dice loss	0.9039	98.58%	87.90%	87.97%
Soft Jaccard loss	0.9081	98.64%	88.38%	87.86%

Table 4: Summary of numerical experiments. All metrics are averages over the test set. The best model metric along each column is annotated in green, while the worst model metric is annotated in red.

We can conclude that LiDAR data is more suitable for segmenting building footprints than RGB data, but by combining LiDAR *and* RGB you end up with a model that performs even better than LiDAR in isolation. When it comes to the method of normalizing LiDAR elevation rasters, dividing by a constant, global scaler produces better results than "dynamic min-max" scaling. It remains to be determined if this "constant" scaling method works in a combined data setting as well.

In a purely quantitative sense the models trained with soft variant losses perform strictly worse than the models trained with binary cross-entropy loss. On the other hand, the soft loss models seem to be less prone to overfitting in addition to portraying a general degree of "common sense" in its predictions, even when they fail. Since there are certain properties of both loss types that are preferable to replicate in an *ideal* model, we propose training a model with a combined loss function, $\mathcal{L}^*$, in the form

$$\mathcal{L}^*(P; Y, \alpha) = \alpha \cdot \mathcal{L}_{\text{BCE}}(P; Y) + (1 - \alpha) \cdot \mathcal{L}_{\text{SJL}}(P; Y), \quad \alpha \in [0, 1],$$

where α is a hyperparameter to be tuned, and the losses $\mathcal{L}_{\text{BCE}}$ and $\mathcal{L}_{\text{SJL}}$ are respectively defined in equation (1) and (2).

Building footprints can be considered a low-fidelity geographic data type. In my upcoming master's thesis I will investigate if the methods presented here can be modified in order to predict higher-fidelity targets, specifically targets related to the three-dimensional geometry of roof surfaces. The bottom of Figure 58 demonstrates the type of higher-fidelity data that is available in Norway, namely three-dimensional line segments classified into categories such as "ridge lines". Being able to predict such features accurately from remote sensing data would provide a much more complex understanding of roof surfaces geometries, yielding insight into interesting properties of surfaces such as orientation, shape, and size. The number of real world applications using such features is uncountable, and the quality and fidelity of labeled data available in Norway offers an unique opportunity to apply deep learning techniques in order to predict such features from remote sensing data.

Figure 58: **Top** – Low-fidelity building footprint data marked in purple. **Bottom** – High-fidelity geometric line segments defined from roof geometries. Ridge lines, for example, are shown in red. ©Kartverket.

Bibliography

References

[1] Martin Abadi et al. "TensorFlow: A system for large-scale machine learning". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 2016, pp. 265–283. URL: https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf.

[2] Terratec AS. *Rapport for laserskanning*. Comissioned by Trondheim kommune. 2017.

[3] V. Badrinarayanan, A. Kendall, and R. Cipolla. "SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 39.12 (Dec. 2017), pp. 2481–2495. ISSN: 1939-3539. DOI: 10.1109/TPAMI.2016.2644615.

[4] Jeroen Bertels et al. "Optimizing the Dice Score and Jaccard Index for Medical Image Segmentation: Theory and Practice". In: *Medical Image Computing and Computer Assisted Intervention – MICCAI 2019*. Ed. by Dinggang Shen et al. Cham: Springer International Publishing, 2019, pp. 92–100. ISBN: 978-3-030-32245-8.

[5] Rich Caruana, Steve Lawrence, and C Lee Giles. "Overfitting in neural nets: Backpropagation, conjugate gradient, and early stopping". In: *Advances in neural information processing systems*. 2001, pp. 402–408.

[6] Wikimedia Commons. *File:LA2-Europe-UTM-zones.png — Wikimedia Commons, the free media repository*. 2015. URL: https://commons.wikimedia.org/w/index.php?title=File:LA2-Europe-UTM-zones.png&oldid=146057239 (visited on 11/04/2019).

[7] Gabriela Csurka et al. "What is a good evaluation measure for semantic segmentation?." In: *BMVC*. Vol. 27. Citeseer. 2013, p. 2013.

[8] George Cybenko. "Approximations by superpositions of a sigmoidal function". In: *Mathematics of Control, Signals and Systems* 2 (1989), pp. 183–192.

[9] Terrance DeVries and Graham W. Taylor. "Improved Regularization of Convolutional Neural Networks with Cutout". In: (2017). arXiv: 1708.04552 [cs.CV].

[10] Ralph O. Dubayah and Jason B. Drake. "Lidar Remote Sensing for Forestry". In: *Journal of Forestry* 98.6 (June 2000), pp. 44–46. ISSN: 0022-1201. DOI: 10.1093/jof/98.6.44. eprint: http://oup.prod.sis.lan/jof/article-pdf/98/6/44/22558157/jof0044.pdf. URL: https://doi.org/10.1093/jof/98.6.44.

[11] Alberto Garcia-Garcia et al. "A Review on Deep Learning Techniques Applied to Semantic Segmentation". In: (2017). arXiv: 1704.06857 [cs.CV].

[12] GDAL/OGR contributors. *GDAL/OGR Geospatial Data Abstraction software Library*. Open Source Geospatial Foundation. 2019. URL: https://gdal.org.

[13] Sean Gillies et al. *Fiona is OGR's neat, nimble, no-nonsense API*. Toblerity, 2011–. URL: https://github.com/Toblerity/Fiona.

[14] Sean Gillies et al. *Rasterio: geospatial raster I/O for Python programmers*. Mapbox, 2013–. URL: https://github.com/mapbox/rasterio.

[15] Ross Girshick. "Fast R-CNN". In: (2015). arXiv: 1504.08083 [cs.CV].

[16] Ross Girshick et al. “Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation”. In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2014.

[17] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. “Deep sparse rectifier neural networks”. In: *Proceedings of the fourteenth international conference on artificial intelligence and statistics*. 2011, pp. 315–323.

[18] Rafael C Gonzalez. *Digital Image Processing*. eng. New York, 2018.

[19] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[20] Richard HR Hahnloser et al. “Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit”. In: *Nature* 405.6789 (2000), p. 947.

[21] Juho Häme. *Shapefile vs. GeoJSON vs. GeoPackage*. Terramonitor Feed. July 3, 2019. URL: https://feed.terramonitor.com/shapefile-vs-geopackage-vs-geojson/ (visited on 08/29/2019).

[22] Kaiming He et al. “Deep Residual Learning for Image Recognition”. In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2016.

[23] Kaiming He et al. “Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification”. In: *The IEEE International Conference on Computer Vision (ICCV)*. Dec. 2015.

[24] Kaiming He et al. “Mask R-CNN”. In: *The IEEE International Conference on Computer Vision (ICCV)*. Oct. 2017.

[25] Jeff Hecht. “Lidar for self-driving cars”. In: *Optics and Photonics News* 29.1 (2018), pp. 26–33.

[26] Geoffrey E. Hinton et al. “Improving neural networks by preventing co-adaptation of feature detectors”. In: *arXiv preprint arXiv:1207.0580* (2012). arXiv: 1207.0580 [cs.NE].

[27] Gao Huang et al. “Deep Networks with Stochastic Depth”. In: *Computer Vision – ECCV 2016*. Ed. by Bastian Leibe et al. Cham: Springer International Publishing, 2016, pp. 646–661. ISBN: 978-3-319-46493-0.

[28] Sergey Ioffe and Christian Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. 2015. arXiv: 1502.03167 [cs.LG].

[29] Statens kartverk. *Ortofoto Trondheim 2017*. May 2018. URL: https://kartkatalog.geonorge.no/metadata/cd105955-6507-416f-86d2-6d95c1b74278 (visited on 11/07/2019).

[30] Statens kartverk. *Produktspesifikasjon for ortofoto i Norge*. 2003. URL: https://register.geonorge.no/data/documents/produktspesifikasjoner_Digitale%20ortofoto_v1_ortofoto-spesifikasjon-v1-2003_.pdf.

[31] Eric Kauderer-Abrams. *Quantifying Translation-Invariance in Convolutional Neural Networks*. 2017. arXiv: 1801.01450 [cs.CV].

[32] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2014. arXiv: 1412.6980 [cs.LG].

[33] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Advances in Neural Information Processing Systems 25*. Ed. by F. Pereira et al. Curran Associates, Inc., 2012, pp. 1097–1105. URL: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[34] Alex Labach, Hojjat Salehinejad, and Shahrokh Valaee. "Survey of Dropout Methods for Deep Neural Networks". In: *arXiv preprint arXiv:1904.13310* (2019). arXiv: 1904.13310 [cs.NE].

[35] Rodney LaLonde and Ulas Bagci. "Capsules for Object Segmentation". In: (2018). arXiv: 1804.04241 [stat.ML].

[36] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep learning". In: *nature* 521.7553 (2015), pp. 436–444.

[37] Moshe Leshno et al. "Multilayer feedforward networks with a nonpolynomial activation function can approximate any function". In: *Neural networks* 6.6 (1993), pp. 861–867.

[38] Yu Han Liu. "Feature Extraction and Image Recognition with Convolutional Neural Networks". In: *Journal of Physics: Conference Series*. Vol. 1087. 6. IOP Publishing. 2018, p. 062032.

[39] Jonathan Long, Evan Shelhamer, and Trevor Darrell. "Fully Convolutional Networks for Semantic Segmentation". In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2015.

[40] Dirk Merkel. "Docker: Lightweight Linux Containers for Consistent Development and Deployment". In: *Linux J.* 2014.239 (Mar. 2014). ISSN: 1075-3583.

[41] F. Milletari, N. Navab, and S. Ahmadi. "V-Net: Fully Convolutional Neural Networks for Volumetric Medical Image Segmentation". In: *2016 Fourth International Conference on 3D Vision (3DV)*. Oct. 2016, pp. 565–571. DOI: 10.1109/3DV.2016.79.

[42] C. A. Northend, R. C. Honey, and W. E. Evans. "Laser Radar (Lidar) for Meteorological Observations". In: *Review of Scientific Instruments* 37.4 (1966), pp. 393–400. DOI: 10.1063/1.1720199. eprint: https://doi.org/10.1063/1.1720199. URL: https://doi.org/10.1063/1.1720199.

[43] H Ozdemir et al. "Evaluating scale and roughness effects in urban flood modelling using terrestrial LIDAR data". In: *Hydrology and Earth System Sciences* 10 (2013), pp. 5903–5942.

[44] Shaoqing Ren et al. "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks". In: *Advances in Neural Information Processing Systems 28*. Ed. by C. Cortes et al. Curran Associates, Inc., 2015, pp. 91–99. URL: http://papers.nips.cc/paper/5638-faster-r-cnn-towards-real-time-object-detection-with-region-proposal-networks.pdf.

[45] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. "U-Net: Convolutional Networks for Biomedical Image Segmentation". In: *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*. Ed. by Nassir Navab et al. Cham: Springer International Publishing, 2015, pp. 234–241. ISBN: 978-3-319-24574-4.

[46] Frank Rosenblatt. "The perceptron: a probabilistic model for information storage and organization in the brain." In: *Psychological review* 65.6 (1958), p. 386.

[47] Even Rouault. *File:gdalvrt.xsd — XML Schema for GDAL VRT files*. 2015. URL: https://raw.githubusercontent.com/OSGeo/gdal/master/gdal/data/gdalvrt.xsd (visited on 11/07/2019).

[48] Sebastian Ruder. *An overview of gradient descent optimization algorithms*. 2016. arXiv: 1609.04747 [cs.LG].

[49] David E Rumelhart et al. "Backpropagation: The basic theory". In: *Backpropagation: Theory, architectures and applications* (1995), pp. 1–34.

[50] Sara Sabour, Nicholas Frosst, and Geoffrey E Hinton. "Dynamic Routing Between Capsules". In: *Advances in Neural Information Processing Systems 30*. Ed. by I. Guyon et al. Curran Associates, Inc., 2017, pp. 3856–3866. URL: http://papers.nips.cc/paper/6975-dynamic-routing-between-capsules.pdf.

[51] Karen Simonyan and Andrew Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: (2014). arXiv: 1409.1556 [cs.CV].

[52] John Parr Snyder. *Map projections–A working manual*. Vol. 1395. US Government Printing Office, 1987.

[53] J. Sola and J. Sevilla. "Importance of input data normalization for the application of neural networks to complex industrial problems". In: *IEEE Transactions on Nuclear Science* 44.3 (June 1997), pp. 1464–1468. DOI: 10.1109/23.589532.

[54] Christian Szegedy et al. "Going Deeper with Convolutions". In: (2014). arXiv: 1409.4842 [cs.CV].

[55] Richard Szeliski. *Computer vision: algorithms and applications*. Springer Science & Business Media, 2010.

[56] Haibing Wu and Xiaodong Gu. "Towards dropout training for convolutional neural networks". In: *Neural Networks* 71 (2015), pp. 1–10. ISSN: 0893-6080. DOI: https://doi.org/10.1016/j.neunet.2015.07.007. URL: http://www.sciencedirect.com/science/article/pii/S0893608015001446.

[57] Aston Zhang et al. *Dive into Deep Learning*. Release 0.7.0. URL: https://d2l.ai (visited on 11/28/2019).

Appendices

A GIS preprocessing

A.1 Overview

The preprocessing pipeline responsible for transforming raw GIS data into a format suitable for machine learning is outlined in Figure 59.

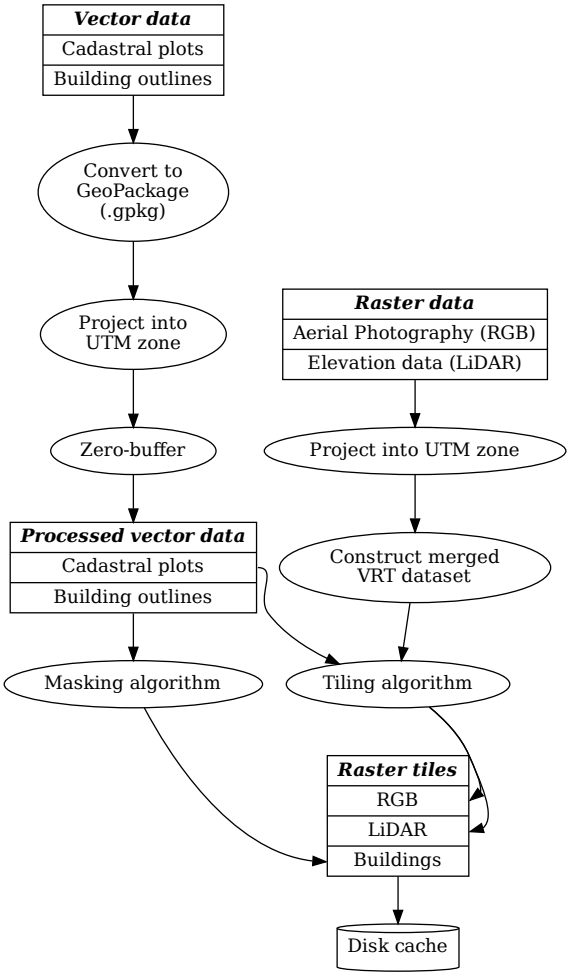


Figure 59: Overview of the GIS preprocessing pipeline developed in order to train machine learning models on geospatial data.

A.2 Mapping between coordinate systems

`GDAL` provides the `gdaltransform` utility for transforming GIS data between coordinate systems, for example converting from GPS to UTM `32V` here:

```
$ gdaltransform \
    -s_srs EPSG:4236 \
    -t_srs EPSG:25832 \
    ${source_data} ${target}
```

Where `$` indicates a shell, such as `bash`, where `GDAL` has been installed and is available in the `PATH`.

A.3 Zero-buffering vector datasets

Section 2.2 discusses irregular vector data and how such vector features can be corrected by applying a zero buffer. The `ST_buffer()` PostGIS function can be applied to arbitrary geometric data with the `ogr2ogr` utility like so:

```
$ ogr2ogr -f "GPKG" ${output_file} ${input_file} \
    -sql "select ST_buffer(Geometry, 0.0)"
```

Here we have also converted the given vector data file to the *GeoPackage* format. While geographic data providers use a wide array of file formats, most commonly GeoJSON, ESRI Shapefiles, and GML, we convert all files to the modern GeoPackage format. GeoPackage supports unicode characters and has no length limit on data fields, and is therefore considered the best format for modern GIS pipelines [21]. `ogr2ogr` supports file conversions between most common vector file formats, which makes the data pipeline generalizable data sourced from different providers.

A.4 Merging raster datasets

Aerial photography and LiDAR data is usually provided in several smaller raster files organized in a tiled pattern in order to reduce individual file sizes. Each file is a `.geotiff` file, a container format which specifies relevant metadata and the underlying image data in a lossless format such as `PNG`. This poses the problem of having to look up which files that cover a given geographic region of interest and merging these files together before processing them.

A simpler approach is to create a *GDAL Virtual Format* file (`VRT`), a virtual dataset file referencing all the respective tiles and bands (GIS uses the term bands for what we would otherwise refer to as image channels). In simple cases, a `VRT` file can be autogenerated with the `gdalbuildvrt` GDAL utility.

```
$ gdalbuildvrt raster.vrt ${raster_directory}/*.tif
```

The resulting `vrt` file behaves like single, merged file, and can be read and processed by most GIS tools. In practice it is just a simple XML file referencing all the underlying `.geotiff` files, thus alleviating the need to load the entire raster dataset into memory every time.

Using the same file format, we can also combine overlapping raster datasets by expanding the number of channels in the resulting raster,

```
$ gdalbuildvrt \
  -resolution ${resolution} \
  combined.vrt \
  -separate \
  ${vrt1} ${vrt2}
```

where `-resolution` can be set to either `highest`, `lowest`, or `average`, depending on how you want to handle datasets with different raster resolutions. This is how we merge the aerial photography (RGB) data with the DSM data (Z), resulting in a single consistent ZRGB dataset. The resulting `VRT` file will only contain the first band from each source file, and needs to be manually edited according to the VRT schema [47] in order to include the green and blue bands of the original RGB dataset. Color interpretations for a ZRGB VRT raster are specified as follows:

```
<ColorInterp>Gray</ColorInterp>
<ColorInterp>Red</ColorInterp>
<ColorInterp>Green</ColorInterp>
<ColorInterp>Blue</ColorInterp>
```

Remember to increment the `band` and `SourceBand` numbers as well; the following eight lines should be placed at suitable locations in the VRT XML file.

```
<VRTRasterBand dataType="Byte" band="1">
<VRTRasterBand dataType="Byte" band="2">
<VRTRasterBand dataType="Byte" band="3">
<VRTRasterBand dataType="Byte" band="4">

<SourceBand>1</SourceBand>
<SourceBand>2</SourceBand>
<SourceBand>3</SourceBand>
<SourceBand>4</SourceBand>
```