Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Mathematical Sciences

Jakob Gerhard Martinussen

Three-dimensional Roof Surface Geometry Inference Using Remote Sensing Data

Master's thesis in Applied Physics and Mathematics
Supervisor: Erlend Aune
July 2020

NTNU
Norwegian University of
Science and Technology

Abstract

In this master's thesis, we present an end-to-end machine learning pipeline for inferring the location, orientation, and elevation of flat roof surfaces from remote sensing data (digital surface models and/or aerial photography). We show that by making a minor modification to the output layer of the U-Net CNN architecture it is able to predict rasterized surface normal vectors with great accuracy. By clustering predicted normal vectors, using DBSCAN and *k*-NN, it is possible to partition semantic roof segmentation maps into corresponding roof surface instance segmentation maps. A single multitask CNN network which predicts both semantic segmentation masks and rasterized normal vectors is shown to be as performant as a pair of respective single-task networks. Finally, an optional vectorization procedure which produces three-dimensional vector polygons from roof surface instance segmentation maps is presented.

Sammendrag

I denne masteroppgaven presenterer vi en ende-til-ende maskinlæringsprosedyre for å identifisere beliggenheten, orientering og høyden til takoverflater ved hjelp av fjernmålinger (digitale overflatemodeller og/eller flyfoto). Vi viser at ved å gjøre små endringer på utputtlaget til U-Net CNN-arkitekturen evner den å predikere rastrerte overflatenormalvektorer med stor nøyaktighet. Ved å anvende klyngeanalyse i form av DBSCAN og k-NN på de predikerte normalvektorene, så er det mulig å partisjonere semantiske taksegmenteringer til å bli tilsvarende forekomstsegmenteringer hvor hver forekomst representerer en individuell takflate. Et CNN-nettverk som predikerer både semantiske segmenteringer samt rastrerte normalvektorer har blitt vist til å være like virkningsfull som et par med respektive nettverk som utfører disse to oppgavene uavhengig av hverandre. En valgfri vektoriseringsprosedyre som produserer tredimensjonale vektorpolygoner fra forekomstsegmenteringer er avslutningsvis presentert.

Contents

Abstract **iii**

Sammendrag **v**

1 Introduction **1**
1.1 Research questions . . . 3
1.1.1 Problem description . . . 3
1.2 Thesis disposition . . . 6

2 Data and Pre-Processing **7**
2.1 Coordinate Systems . . . 7
2.2 Data Types . . . 8
2.2.1 Vector data . . . 8
2.2.2 Raster data . . . 10
2.3 Datasets . . . 11
2.3.1 Raster datasets . . . 11
2.3.2 Vector datasets . . . 14
2.4 Tiling Algorithm . . . 15
2.5 Masking Algorithm . . . 19
2.6 Surface Rasterization Algorithm . . . 21
2.6.1 Desirable surface raster properties . . . 21
2.6.2 The “surface normal” raster format . . . 24
2.6.3 Handling overlapping surface polygons . . . 31
2.6.4 Handling expensive spatial queries . . . 33
2.6.5 Handling non-planar polygons . . . 35
2.7 Overview . . . 40

3 Modeling **41**
3.1 Convolutional Neural Networks (CNNs) . . . 41
3.1.1 Convolution . . . 42
3.1.2 Activation functions . . . 44
3.1.3 Pooling . . . 45
3.1.4 Batch normalization . . . 46
3.1.5 Dropout . . . 47
3.2 Semantic Segmentation . . . 48

3.2.1 Accuracy, sensitivity, and specificity 49
3.2.2 Intersection over union and dice coefficient 50
3.2.3 Binary cross-entropy and soft losses 52
3.2.4 State-of-the-art 53
3.2.5 The U-Net model architecture 55
3.3 Surface Normal Vector Prediction 57
3.3.1 Related work 57
3.3.2 CNN architecture for predicting surface normal vectors . . . 58
3.4 Optimization 61
3.4.1 Multitask learning 62
3.5 Raster Normalization 63
3.5.1 RGB rasters 63
3.5.2 LiDAR rasters 64

4 Post-processing 69
4.1 Connected region labeling 72
4.2 Instance Clustering 74
4.2.1 DBSCAN 75
4.2.2 *k*-nearest neighbour noise classification 76
4.3 Vectorization 78
4.3.1 Two-dimensional polygonization 78
4.3.2 Simplification 79
4.3.3 Three-dimensional reconstruction 81

5 Experiments 83
5.1 Experimental Setup 83
5.1.1 Training procedure 83
5.1.2 Software 84
5.1.3 Hardware and performance 84
5.2 Semantic Segmentation 85
5.3 Features 86
5.3.1 LiDAR data 86
5.3.2 RGB data 90
5.3.3 Combined data 93
5.4 Multitask learning 97

6 Conclusion and Further Work 111

Bibliography 113

A GIS pre-processing 121
A.1 Mapping between coordinate systems 121
A.2 Zero-buffering vector datasets 121
A.3 Merging raster datasets 122

Figures

1.1 Classification of geographic data quality. 2
1.2 Different granularities for single-class construction localization, using the Trondheim 2017 data set. 4

2.1 UTM zones covering Europe. 8
2.2 Types of vectorized polygons. 9
2.3 Illustration of how zero-buffering an invalid polygon corrects self-intersecting polygons. 9
2.4 Illustration of how zero-buffering polygons removes redundant vertices. 9
2.5 Visualization of the "Ortofoto Trondheim 2017" aerial photography dataset. 11
2.6 Example of nonproper orthophoto. 12
2.7 Visualization of LiDAR data from Trondheim. 13
2.8 Illustration of vector datasets. 14
2.9 Comparison of bounding box methods. 15
2.10 Distribution of bounding box dimensions. 16
2.11 Illustration of bounding box growing. 17
2.12 The same polygon discretized to a raster grid using two different techniques. 19
2.13 Distribution of *building density* across all produced tiles in Trondheim. 20
2.14 Three-dimensional polygonal gable roof. 21
2.15 Invertible rasterization. 22
2.16 Projection of three-dimensional polygon onto xy-plane by $\pi_{2D}(\cdot)$. 26
2.17 The decomposition and reconstruction of a three-dimensional polygon. 26
2.18 The relationship between the equation of the plane, $\boldsymbol{\beta}(P)$, and the surface normal vector, $\boldsymbol{n}(\boldsymbol{\beta}(P))$. 27
2.19 The deconstruction of a three-dimensional polygon into two separate rasters formats. 29
2.20 Illustration of surface raster formats Z, N, and S. 30
2.21 Illustration of overlapping surface polygons. 31
2.22 Trondheim distribution of number of intersecting surface polygons contained by each raster tile $B \in \mathcal{B}$. 34

2.23 Statistical summary of the regression fits of the Trondheim surface polygons. 37
2.24 The construction of LiDAR residuals from a surface elevation raster. 38
2.25 Distribution of LiDAR residuals calculated over all bounding boxes $B \in \mathcal{B}$ for the Trondheim dataset. 38
2.26 Overview of the GIS preprocessing pipeline developed in order to train machine learning models on geospatial data. 40

3.1 Illustration of the U-Net architecture for single-class segmentation, a typical example of an *encoder/decoder* structure. 42
3.2 Visualization of a kernel convolution. 43
3.3 Example of a *max-pooling* operation with a receptive field of size 2×2 and an identical stride size. 46
3.4 An example application of *max-pooling dropout* using a receptive field and stride of size 2×2. 47
3.5 Illustration of TP, TN, FP, and FN for a binary segmentation prediction. 49
3.6 Visualization of single-class IoU metric. 51
3.7 Visualization of the single-class dice coefficient metric, also known as the F_1 score. 51
3.8 U-Net model architecture. 56
3.9 Demonstration of pixel-wise cosine similarity loss. 60
3.10 Distribution density for all three color channels in the aerial photography data set covering Trondheim municipality (2017). 64
3.11 Distribution density for elevation data set covering Trondheim municipality (2017). 65
3.12 Elevation value statistics for a tile subset of sample size $n = 10\,000$. 65

4.1 Demonstration of unprocessed output from a surface raster machine learning pipeline. 69
4.2 Pseudoinverse mapping, $m^\dagger$, mapping from the raster domain to the vector domain. 70
4.3 Thresholding a probabilistic segmentation prediction in order to create a *binary* segmentation map. 72
4.4 Segmentation of a predicted surface normal raster using a binary segmentation map. 72
4.5 The result of `skimage.measure.label` applied on a binary segmentation map. 73
4.6 The distribution of the cosine similarities between the ground truth normal vectors and the predicted normal vectors of a surface normal model over 9537 test tiles. 74
4.7 Ground truth labeling of predicted normal vectors. 74
4.8 DBSCAN labeling of predicted normal vectors. 75
4.9 Application of DBSCAN on predicted surface normal raster. 76
4.10 k-NN labeling of remaining DBSCAN noise. 76

4.11 Application of k-NN on DBSCAN-identified noise, with $k = 1$ and using the ℓ^2-norm. 77
4.12 The result of applying `GDALPolygonize` on the clustered instance map $\widehat{L}$. 78
4.13 Polygons before and after RDP has been applied with bandwidth tolerance $\varepsilon = 0.75\,\mathrm{m}$. 79
4.14 Illustration of the Ramer-Douglas-Peucker algorithm. 80

5.1 Training procedure of LiDAR-only U-Net-derived architecture for predicting surface normals over 100 training epochs. 86
5.2 Distribution of instance-averaged cosine similarities (IACS) of the LiDAR-only surface normal model over the test split. 87
5.3 Median test prediction for LiDAR-only normal vector model. 88
5.4 Post-processing of median test prediction for LiDAR-only model with respect to the cosine similarity loss component. 89
5.5 Training procedure for surface normal vector models using *either* LiDAR or RGB as input. 90
5.6 Distribution of instance-averaged cosine similarities of the RGB-only surface normal model over the test split. 91
5.7 Median test prediction for RGB-only normal vector model. 91
5.8 Post-processing of median test prediction for RGB-only model with respect to the cosine similarity loss component. 92
5.9 Training procedure for surface normal vector models using LiDAR and/or RGB as input. 93
5.10 Distribution of instance-averaged cosine similarities of the combined input surface normal model over the test split. 94
5.11 Median test prediction for the combined input normal vector model. 95
5.12 Post-processing of median test prediction for combined-input model with respect to the cosine similarity loss component. 96
5.13 End-of-epoch multitask loss decomposition for different values of α over 20 training epochs. 98
5.14 Best loss component value across all 20 training epochs for each value of α. 99
5.15 Training procedure of combined input, U-Net-derived multitask architecture ($\alpha = 10^{-5}$) for predicting semantic segmentation masks *and* surface normals over 100 training epochs. 100
5.16 End-of-epoch loss components of multitask network compared to respective single-task losses. 101
5.17 Distribution of instance-averaged cosine similarities of the surface normal vector rasters produced by the multitask network over the test split. 102
5.18 Distribution of IoU evaluations of the combined-input models (single-task and multitask) over tiles from the test set. 103

5.19 Scatter plot showing the correlation between the IoU evaluation metric performance of two surface segmentation models, single-task vs. multitask. 104
5.20 Median test prediction for multitask model with respect to the cosine similarity loss component. 105
5.21 Post-processing of median test prediction for multitask model with respect to the cosine similarity loss component. 106
5.22 95th percentile test prediction (bad prediction outlier) for multitask model with respect to the cosine similarity loss component. 107
5.23 Post-processing of 95th percentile test prediction (bad prediction outlier) for multitask model with respect to the cosine similarity loss component. 108
5.24 5th percentile test prediction (good prediction outlier) for multitask model with respect to the cosine similarity loss component. 109
5.25 Post-processing of 5th percentile test prediction (good prediction outlier) for multitask model with respect to the cosine similarity loss component. 110

Tables

3.1 Aggregate statistics for each image channel distribution for the aerial photography data set covering Trondheim municipality (2017). . . 64
3.2 Aggregate statistics for elevation data set covering municipality of Trondheim (2017). 65

Code Listings

3.1 Keras layer producing raster values of absolute magnitude less than or equal to 1. 59
3.2 A Keras modeling layer implementing ℓ_2-normalization. 59
3.3 Cosine similarity loss function implemented in Tensorflow v2.1. . . 61

A.1 Coordinate system transformation using `gdaltransform`. 121
A.2 Zero-buffering vector dataset using `ogr2ogr`. 121
A.3 Virtual merger of raster tiles using `gdalbuildvrt`. 122
A.4 Merging overlapping raster datasets using `gdalbuildvrt`. 122
A.5 Setting color interpretation of multi-channel VRT rasters. 123
A.6 Specifying source bands for multi-channel VRT rasters. 123

Chapter 1

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 [1], forestry analysis [2], urban flood modeling [3], and autonomous driving systems [4].

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 1.1 shows an outline of the possible "data enhancements" which are of interest within this domain.

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.

The goal of my specialization project [5] from 2019 was to infer two-dimensional *building outlines* from aerial photography and LiDAR elevation mea-

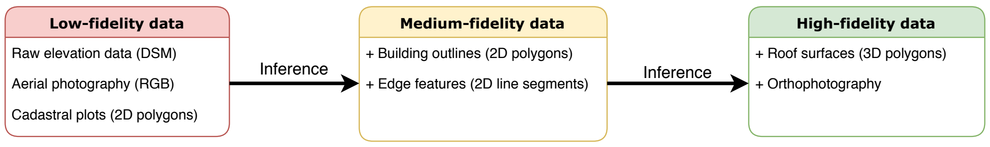


Figure 1.1: Classification of geographic data quality. The classifications reflect a general observed trend in data sets, and a given region may therefore not fit into exactly one of these categories. Some of the data types mentioned here will be described in Chapter 2.

surements. 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 is considered to be medium-fidelity target inference by using low-fidelity features, and can be formulated as a so-called *semantic segmentation* task[1].

This master's thesis concerns itself with the reconstruction of three-dimensional *roof surface polygons* from remote sensing data. Roof surface polygons are completely flat geometries, which when combined form the spatial shape of entire roof structures. The detection of roof surface polygons is formulated as an *instance segmentation* problem, a task which produces high-fidelity targets. Three-dimensional representations of roof structures can for example be used for urban planning purposes. Another application, which incidentally prompted my interest in this topic, is the use of roof surface geometries to estimate the potential energy production of roof-mounted solar panel installations.

The topic of this master's thesis is a natural extension of much of the work already presented in my specialization project. A sequential four-step method has been developed in order to detect roof surface polygons,

1. Determine which pixels that contain roof structures (semantic segmentation).
2. Assign a specific roof surface to each "roof pixel" (instance segmentation).
3. Fit a 2D polygon for each roof surface instance (polygon segmentation).
4. Reconstruct the 3D orientation and elevation of each roof surface polygon (surface segmentation).

The first step is essentially a minor reformulation of the task already solved in my specialization project. For this reason, much of the theory, general methods, and specific source code from my specialization project has been incorporated into this work.

[1]The concepts "semantic segmentation" and "instance segmentation" will be formally defined in Section 1.1.1.

1.1 Research questions

Geographic data, such as building outlines and roof surfaces, 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 representations be transformed into suitable formats for machine learning?

After having developed such a pipeline, the focus will be to develop an instance segmentation model for identifying roof surfaces with *raster* 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 in order to infer accurate roof surface instance segmentation maps?

The resulting instance segmentation map, which represents rasterized roof surfaces in two dimensions, should be *vectorized*. That is, the machine learning model's raster predictions should be converted to three-dimensional vector polygons, a data format which is more usable for the most common applications of roof surface geometries.

RQ3 How can three-dimensional roof surface *polygons* be produced from predicted instance segmentation *raster* maps.

Answering this last research question will require the development of additional post-processing methods.

We will now formalize the problem domains which these research questions belong to, namely *semantic segmentation*, *instance segmentation*, *polygon segmentation*, and *surface segmentation*.

1.1.1 Problem description

For any given image we can pose three relevant *image recognition* questions [6]:

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 roof surfaces, and will simplify the upcoming 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 by the *columns* in Figure 1.2. The *rows* of Figure 1.2 show how the specific definition of what exactly constitutes an object influences the problem to be solved, where the top

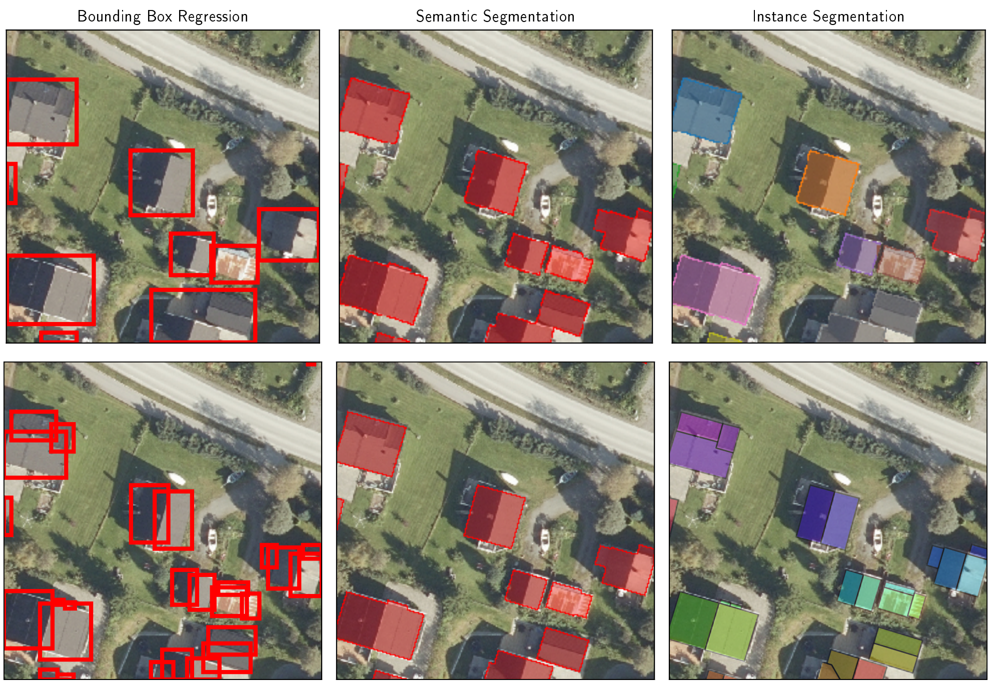


Figure 1.2: Different granularities for single-class construction localization, using the Trondheim 2017 data set. Bounding box regression is shown on the left, semantic segmentation in the middle, and instance segmentation on the right. The top row defines entire buildings as the objects of interest, while the bottom row considers each individual roof surface as distinct objects.

row considers entire building to be single objects, while the bottom row considers each individual roof surface to be distinct objects. It is the latter definition which is of interest in this work.

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 2.2.2. Single-class semantic segmentation can therefore be formalized as constructing a binary predictor $\tilde{f}$ of the form:

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

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 $[0, 1]$,

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

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 Section 3.2.

When performing single-class *instance* segmentation, a binary prediction mask is not sufficiently expressive. Assuming that no more than m individual instances can be simultaneously represented by any given input tensor X, the task is to assign an instance label $l \in \{0, 1, 2, \dots, m\}$ to every single pixel in the input tensor. The assignment of $l = 0$ means that no object overlaps the given pixel. An instance predictor $\hat{f}_{\widehat{L}}$ is therefore a function producing a label array $\widehat{L}$ of the form

$$\hat{f}_{\widehat{L}} : \mathbb{R}^{W \times H \times C} \rightarrow \{0, 1, 2, \dots, m\}^{W \times H \times C}.$$

Instead of representing each predicted instance as a set of *raster* pixels which share the same label value l in $\widehat{L}$, they can be represented as two-dimensional *vectorized* polygons which enclose each given instance instead (*polygon segmentation*). The specific data format used in order to represent vectorized polygons is presented in Section 2.2.1. For the case of three-dimensional objects, the instance polygons can be constructed to be three-dimensional as well, sometimes referred to as *surface segmentation* [7, 8]. Since the three-dimensional polygons in a surface segmentation can be projected into a two-dimensional plane in order to produce a polygon segmentation map, and since two-dimensional polygons can be *rasterized* in order to create an instance segmentation map, the problem complexity of these tasks are as follows:

$$\text{Instance segmentation} < \text{Polygon segmentation} < \text{Surface segmentation}.$$

This thesis presents a machine learning pipeline which produces vectorized surface polygon segmentations from remote sensing data. The pipeline does this by first predicting a semantic segmentation map, which is then partitioned into an instance segmentation map. This instance segmentation map is then vectorized in order to produce a polygon segmentation map. Finally, the three dimensional elevation and orientation of each polygon is inferred from the original LiDAR data in order to produce a surface segmentation map.

1.2 Thesis disposition

We will start by providing an introduction to the world of *Geographic Information Systems* (GIS); the field which concerns itself with representing geographic data, in Chapter 2. We will also describe how to pre-process such geographic data in order to produce rasters which are suitable for training accurate machine learning models. An overview of the problem domain of image segmentation and the methods currently being applied in the field will be provided in Chapter 3, a chapter which will also describe the specific model architectures which consume and produce the data formats described in the previous chapter. The post-processing required in order to produce vectorized surface polygons from predicted rasters will be described in Chapter 4. Finally, the training procedure and experimental results will be presented and discussed in Chapter 5.

Chapter 2

Data and Pre-Processing

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 datasets used for training our models. The remaining subsections will describe the pre-processing 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 surface rasterization (Section 2.6). A figurative overview of the preprocessing pipeline is provided by Figure 2.26 in Section 2.7.

2.1 Coordinate Systems

One of the most common coordinate systems for representing *arbitrary* positions on earth's surface is the *World Geodetic System* (*WGS*), the latest revision being *WGS 84* [9]. 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 WGS 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 preserving 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 projections.

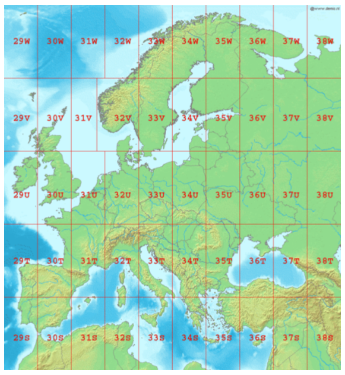


Figure 2.1: The figure shows the UTM zones required in order to cover the entirety of Europe, from `29S` to `38W`. This public domain image has been sourced from Wikimedia [10].

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 [11, p. 48]. The UTM zones covering Europe are shown in Figure 2.1. 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 transformations. Practical instructions for how to map between different coordinate systems are given in Appendix A.1.

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.

2.2.1 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 2.2 illustrates these concepts for polygons with and without interior hulls.

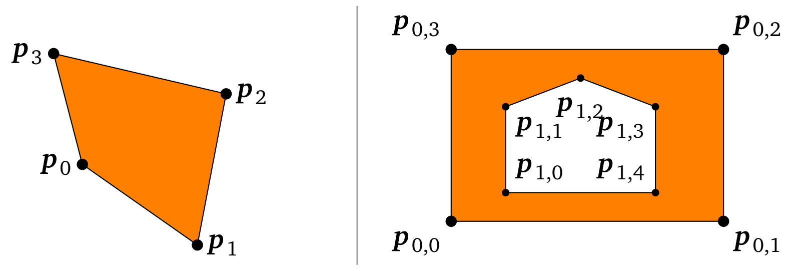


Figure 2.2: Simple polygon with four unique vertices is shown on the left hand side. A complex polygon with an outer hull and an interior hull is shown on the right hand side for comparison.

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 2.3.

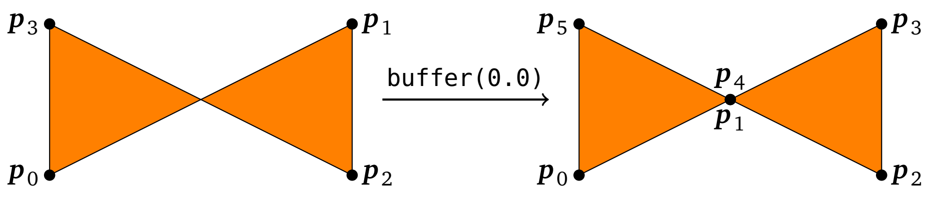
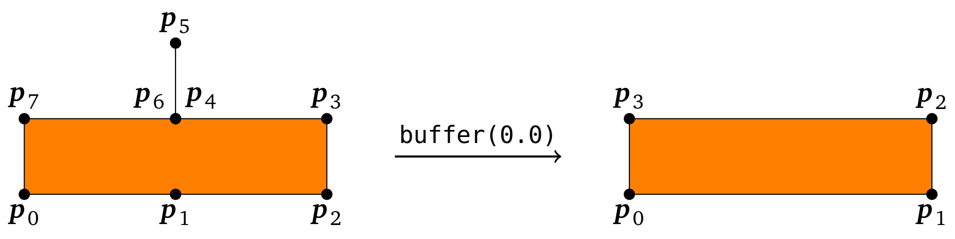


Figure 2.3: Illustration of how zero-buffering an invalid polygon corrects self-intersecting polygons.

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 2.4.



.

Figure 2.4: 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.2. 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.

2.2.2 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.

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

A LiDAR elevation map, which we will denote as E, 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

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

These two raster types must be handled differently during data-standardization and -normalization due to their different value domains, which we will come back to in Section 3.5. Whenever we refer to remote sensing raster data in *general*, be it LiDAR and/or RGB, we will denote the input raster as X.

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, $X_{1,1}$; 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 value $X_{i,j,c}$ therefore represents a rectangle of width Δ_x and height Δ_y centered at the spatial coordinate $\boldsymbol{r}_0 + [\Delta_y i, \Delta_x j]$ 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 $\texttt{nodata} = -2^{127} \times (2 - 2^{-23}) \approx -3.4028234664 \times 10^{38}$, 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 3.5.

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.3 for how this is performed in practice.

2.3 Datasets

The modeling results presented in Chapter 5 are trained on GIS data covering the Norwegian municipality of Trondheim. All datasets, except from the three-dimensional roof polygon dataset kindly provided by Norkart, 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[1] which restricts the use to non-commercial purposes.

2.3.1 Raster datasets

We will use the "Ortofoto Trondheim 2017"[2] aerial photography dataset 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 resampled resolution of 0.1 m for consistency, that is, each pixel is of size 0.1 m × 0.1 m. The reported accuracy is ±0.35 m [12], although the exact type of this accuracy is not specified. An exemplified region is visualized in Figure 2.5.

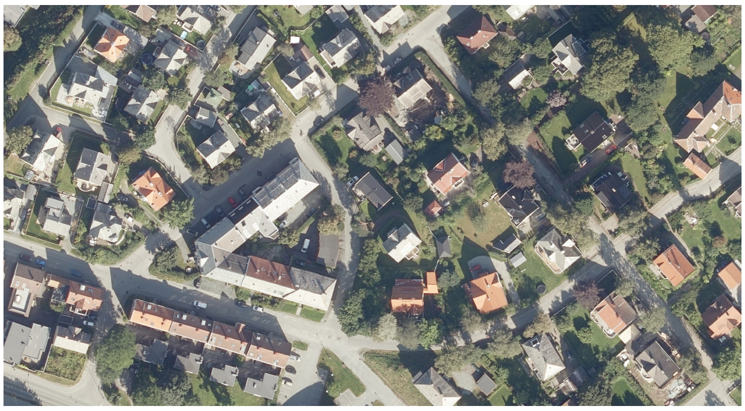


Figure 2.5: Visualization of the "Ortofoto Trondheim 2017" aerial photography dataset. © Kartverket.

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 [13], including Trondheim. Aerial

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

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

photography which has not been properly "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 2.6.

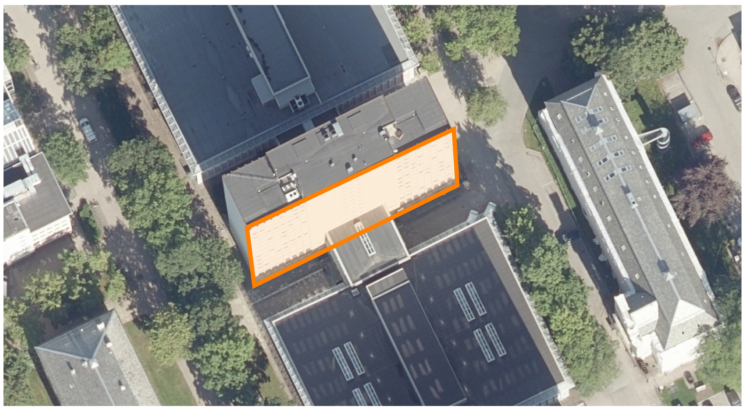


Figure 2.6: Example of nonproper orthophoto. The building centered in the image is 14 stories tall. The orange area annotates a clearly visible building wall. © Kartverket.

As can be seen in Figure 2.6, the "Ortofoto Trondheim 2017" dataset 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 semantic segmentation predictions has been investigated in our previous work [5], the conclusion being that it does *not* impede predictive accuracy to a major degree.

The LiDAR dataset used is "Høydedata Trondheim 5pkt 2017"[3] from and requires 25 GB of storage space. The pixel size is 0.25 m × 0.25 m and the LiDAR measurements have a reported standard deviation of 0.02 m [14]. LiDAR visualized as a grayscale image over the same region as in Figure 2.5 is presented in Figure 2.7.

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

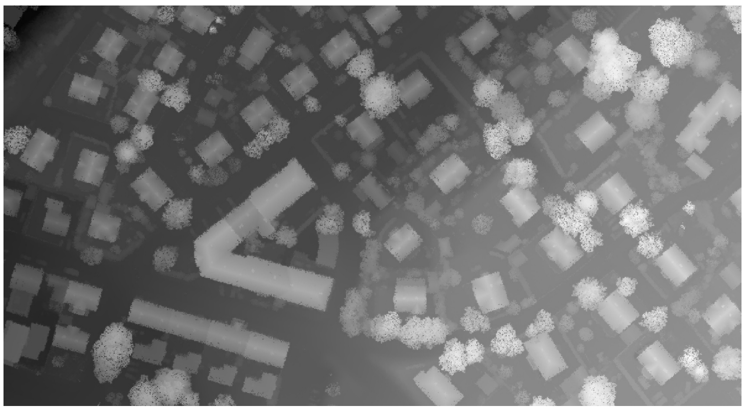


Figure 2.7: Visualization of the “Høydedata Trondheim 5pkt 2017” LiDAR dataset. © Kartverket.

2.3.2 Vector datasets

The “Matrikkelen - Eiendomskart Teig”[4] dataset contains all cadastral plots in Trondheim, the use of which will be explained in Section 2.4. The “FKB-bygning”[5] dataset contains all registered building outlines in Trondheim. The building outlines will be used to construct binary classification masks as outlined in Section 2.5. Additionally, the “FKB-bygning” dataset includes a complete collection of descriptive building lines, such as ridge lines, verge lines, and so on. This dataset is of no direct use to us, but Norkart has constructed an algorithm for merging these lines in order to construct three-dimensional roof surface polygons. This polygonized dataset has been kindly provided to us by Norkart, and will be used as outlined in Section 2.6. These four datasets are illustrated in Figure 2.8.

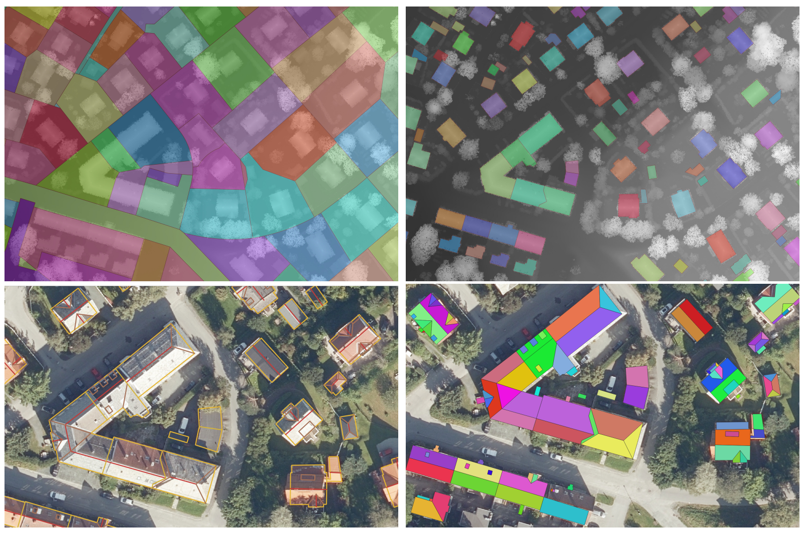


Figure 2.8: Illustration of vector datasets. Cadastral plots are shown on the top left while building outlines are shown on the top right. Descriptive building lines are shown on the bottom left, such as ridge lines shown in red, while the polygonized roof surfaces produced by Norkart are shown on the bottom right. © Kartverket.

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

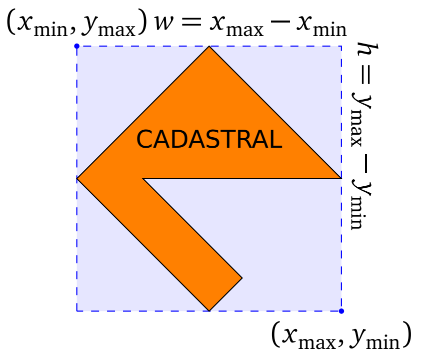
[5]Product specification for “FKB-bygning” can be found here: `https://kartkatalog.geonorge.no/metadata/8b4304ea-4fb0-479c-a24d-fa225e2c6e97`.

2.4 Tiling Algorithm

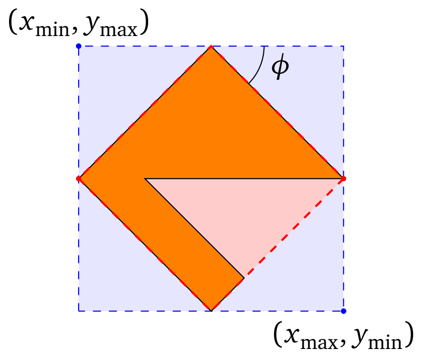
The data sets provided to us are in a state unsuitable for direct use by machine learning frameworks. For this reason we need to develop a preprocessing pipeline that transforms the data into a more customary format. The data preprocessing should be generalizable to different regions, data formats, data types (vector vs. raster), coordinate systems, and so on. The goal is to implement a modeling 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 $\mathbf{c} = [1/2(x_{\text{min}} + x_{\text{max}}), 1/2(y_{\text{min}} + y_{\text{max}})]$, width $w = x_{\text{max}} - x_{\text{min}}$, and height $h = y_{\text{max}} - y_{\text{min}}$, and we will denote it by $B(\mathbf{c}, w, h)$. This is shown in Figure 2.9a.


$(x_{\text{min}}, y_{\text{max}})$
$w = x_{\text{max}} - x_{\text{min}}$
$h = y_{\text{max}} - y_{\text{min}}$
CADASTRAL
$(x_{\text{max}}, y_{\text{min}})$

(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.


$(x_{\text{min}}, y_{\text{max}})$
ϕ
$(x_{\text{max}}, y_{\text{min}})$

(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 2.9: 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 2.9b.

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 2.10.

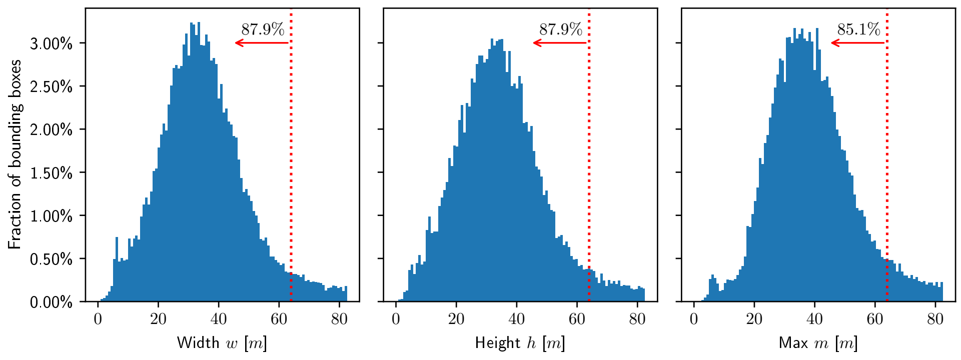


Figure 2.10: Distribution of bounding box widths w (left), heights h (middle), and largest dimension $m = \max\{w, h\}$ (right). The cut-off value of 64 m is shown by red dotted vertical lines. The fraction of bounding boxes with dimension $\leq$ 64 m is annotated as well. The x-axis has been cut off at the 90th percentile. *Dataset: Trondheim cadastre*.

As can be seen in Figure 2.10, 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 m × 64 m bounding box will be of sufficient size to contain ≈ 85 % of all cadastre plots in a single tile. With a LiDAR resolution of 0.25 m, this results in a final image resolution of 256 px × 256 px. This resolution has the added benefit of being a common resolution for CNNs.

How should the bounding boxes be normalized to to 256 px × 256 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 $0.25\,\text{m} \times 0.25\,\text{m}$ 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 $B(\mathbf{c}, w, h)$. 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, $B(\mathbf{c}, w^*, h^*)$, 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 2.11.

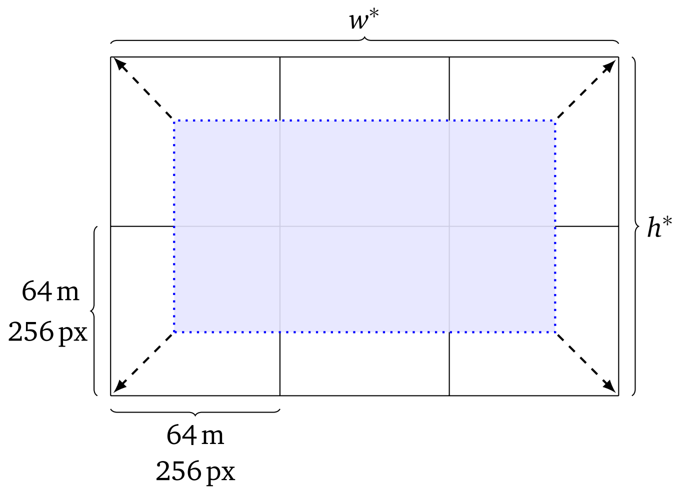


Figure 2.11: Bounding box of width $2.25 \cdot 64\,\text{m} = 144\,\text{m}$ and height $1.25 \cdot 64\,\text{m} = 80\,\text{m}$. The bounding box is grown until it is 3 tiles wide and 2 tiles tall, i.e. $192\,\text{m} \times 128\,\text{m}$.

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 $0.25\,\text{m} \times 0.25\,\text{m}$ 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, $B(\boldsymbol{c}, w, h) \setminus B(\boldsymbol{c}, w^*, h^*)$[6], 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.

[6]Given geographic regions A and B, the region $A \setminus B$ is defined as the region covered by A but *not* by B.

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 poly-
    gons 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 2.12.

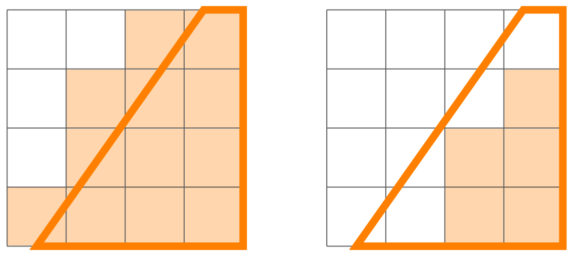


Figure 2.12: The same polygon discretized to a raster grid using two different techniques. In the left figure, all pixels being *touched* by the interior of the polygon are considered a part of the polygon (decision rule A), while in the left figure, only pixels entirely *contained* within the interior 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 B, as it has been observed to preserve symmetries and seams to a larger degree than decision rule A. The distribution of the mask class balance across all produced tiles is shown in Figure 2.13.

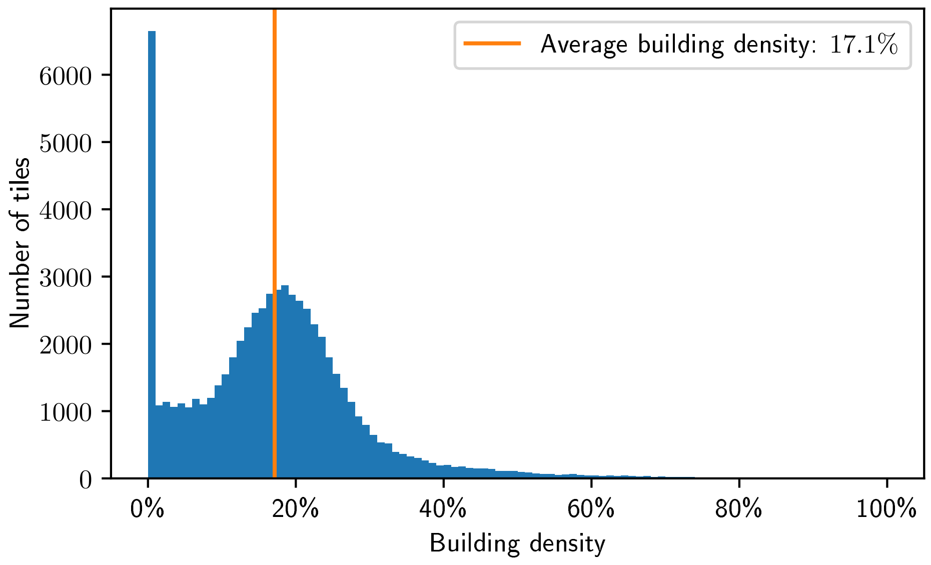


Figure 2.13: Distribution of *building density* across all produced tiles in Trondheim. Building density is defined by number of pixels positioned on top of buildings divided by total number of pixels.

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 Surface Rasterization Algorithm

The roof of a given building can be decomposed into a collection of entirely flat polygons. This is an accurate data decomposition in most cases, the exception being conic shapes and other surfaces with continuous curvature which cannot be perfectly represented with a finite set of flat surfaces. A *gable roof*[7], for instance, can be considered as a collection of two flat polygons, which when combined accurately represent the roof in its entirety. Our intent is to construct a machine learning pipeline which is able to identify such three-dimensional roof surfaces from remote sensing data.

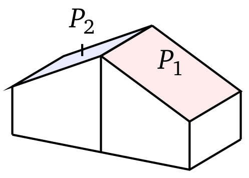


Figure 2.14: Three-dimensional polygonal gable roof.

Although a set of vectorized, flat polygons is often the most suitable data representation for geometric roof data in the GIS domain, it is *not* considered an ideal representation for traditional machine vision data pipelines. Polygons can consist of an arbitrary number of linear rings, and each linear ring can be represented by an arbitrary number of vertices. The number of rings and vertices depends on the complexity of the polygon's shape. Deep learning model architectures, on the other hand, are often restricted to training on and predicting observations of *consistent* dimensionality. This is why machine vision architectures most often consume and/or produce spatial information in the form of *rasters* rather than vectors, that is, numeric arrays of consistent size and dimension. In order to reconcile these conflicting requirements we now pose the following question,

> "How can an arbitrarily sized set of three-dimensional polygons, all of arbitrary complexity, shape, and orientation, be accurately represented in the form of a raster?".

The ideal raster format would allow us to train on and predict roof surfaces in vectorized polygon form, all the while applying the tried and tested techniques from the machine vision literature which mainly concerns itself with rasters. We will refer to such a raster as a *surface raster* in order to distinguish it from other remote sensing raster types such as aerial photography and LiDAR data. The careful formulation and construction of this surface raster format is considered one of the most important problems to be solved in order to construct an efficient machine learning pipeline for predicting roof surfaces.

2.6.1 Desirable surface raster properties

The ideal surface raster format should be both *representative* and *targetable*, properties which we now shall formally define. Start by denoting the domain which consists of polygon *sets* of arbitrary size as V. A polygon in vector format will be denoted as P, while a set of vector polygons will be denoted as $\mathcal{P} = \{P_1, P_2, \dots\}$. A

[7]*Gable roof* — A roof which consists of two flat roof surfaces which slope in opposite directions. The roof surfaces are connected along the highest, horizontal edge (ridge).

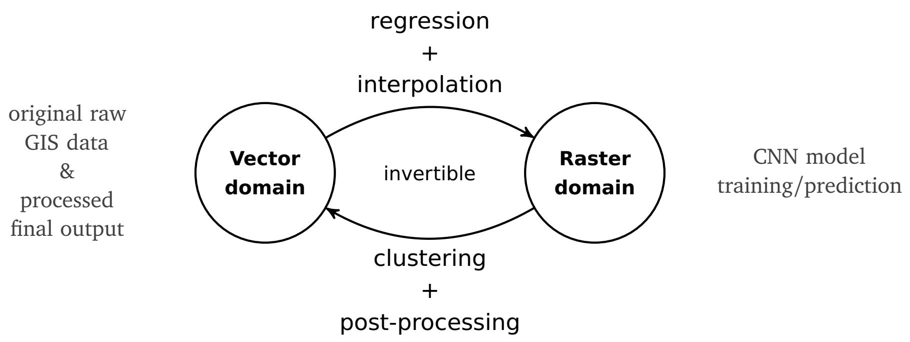


Figure 2.15: Invertible rasterization.

vector polygon set $\mathcal{P}$ is therefore a member of the superset domain V, denoted as $\mathcal{P} \in V$. Assume that we want to construct a surface raster of resolution height H and resolution width W, and that this raster will consists of C_R raster channels. If we denote this surface raster domain as R and assume that raster values will take values from the real number line, then we have $R = \mathbb{R}^{H \times W \times C_R}$. Finally, assume that the remote sensing raster data, X, has all the same dimensional properties as the surface raster data with the possible exception of the number of raster channels, which we will denote as C_X, i.e. $X = \mathbb{R}^{H \times W \times C_X}$. We can now define the surface raster properties of representativeness and targetability with this notation in mind.

Representative

Converting surface polygons to the surface raster format and then back again incurs negligible loss of information.

Define a suitable distance metric $d : V \times V \to R$ which incorporates some notion of the difference in spatial location and orientation between two polygons sets. A *perfectly representative* raster format would allow us to define a mapping from the vector domain to the surface raster domain, $m : V \to R$, for which there exists a functional inverse, m^{-1}, such that

$$\begin{aligned} & m^{-1}\left(m(\mathcal{P})\right) \equiv \mathcal{P}, && \forall \mathcal{P} \in V. \\ \implies & d\left(m^{-1}(m(\mathcal{P})),\ \mathcal{P}\right) \equiv 0, && \forall \mathcal{P} \in V. \end{aligned}$$

That is, we can map from the vector polygon domain to the raster polygon domain, and then back, all without losing *any* information. The vector domain V is infinite-dimensional, while the raster domain R is by necessity of finite and consistent size. It can therefore be concluded that no such invertible mapping exists. For this reason we introduce the concept of a *pseudoinverse*, $m^{\dagger}$, a function which minimizes $d(m^{\dagger}(m(\mathcal{P})),\ \mathcal{P})$ for all $\mathcal{P} \in V$. A raster domain (and an associated mapping and pseudoinvertible mapping) which produces negligible distance metrics after a round-trip mapping of arbitrary polygon sets is considered to be *representative*.

Targetable

A raster format which is feasible as a modeling target for a machine learning architecture.

We intend to construct a predictor, $\hat{f}$, which accepts remote sensing raster data as input and produces the aforementioned surface raster data representation as output, i.e. $\hat{f} : X \rightarrow R$, or equivalently $\hat{f} : \mathbb{R}^{H \times W \times C_X} \rightarrow \mathbb{R}^{H \times W \times C_R}$. Now assume $\hat{f}$ to be parametrized according to the parameter vector $\boldsymbol{\theta}$, and denote the parametrized prediction as $\hat{Y} := \hat{f}(X; \boldsymbol{\theta})$. The intention is that surface raster prediction $\hat{Y}$ constructed from remote sensing data should be as similar to the polygon-constructed raster $Y := m(\mathcal{P})$ as possible. Similar to the distance metric d defined previously, we now define a suitable differentiable *loss* function $\mathcal{L} : R \times R \rightarrow \mathbb{R}$ which incorporates some notion of difference between two surface rasters. The predictor $\hat{f}$ can therefore be parametrized such that this loss function is minimized when evaluated on the predicted surface raster in conjunction with the ground truth surface raster $m(\mathcal{P})$:

$$\begin{aligned} \boldsymbol{\theta}_{\text{opt}} &:= \operatorname*{argmin}_{\boldsymbol{\theta}} \sum_{(\hat{Y}, Y)} \mathcal{L}\big(\hat{Y}; Y\big) \\ &= \operatorname*{argmin}_{\boldsymbol{\theta}} \sum_{(X, \mathcal{P})} \mathcal{L}\big(\hat{f}(X; \boldsymbol{\theta}); m(\mathcal{P})\big) \end{aligned}$$

A raster mapping for which a suitable loss function can be constructed and minimized is considered to be *targetable*.

The considerations of representativeness and targetability are in many ways diametrically opposed when constructing a suitable surface raster format. As an instructive example consider the choice of C_R, the number of raster channels used by the surface raster. Since the mapping m maps from a infinite-dimensional space to a finite-dimensional one, it can be considered a compression method of sorts. The smaller the value of C_R, the greater the compression, and subsequently its associated compression loss. Thus the greater number of raster channels, the more representative the raster format can become. On the other hand, when C_R grows large it is natural to assume that the increasing degree of freedom in the data format allows for many equivalently accurate representations of the same polygon collection. This ambiguity will result in a difficulties when formulating a proper, convex loss function with single, global minima. The targetability of the raster format may therefore suffer from large values of C_R.

Although it is the surface raster loss function we minimize in practice, it is actually not what we are *really* interested in minimizing. The surface raster is in essence only an intermediate data format which is intended to be converted back into the vector domain by the pseuo-inverse $m^{\dagger}$. The underlying idea is that if we minimize the difference between $\hat{Y}$ and Y, we implicitly minimize the difference between $m^{\dagger}(\hat{Y})$ and $\mathcal{P}$. This assumes that $\mathcal{L}$ is a good loss surrogate for functional composition $d \circ m^\dagger$, by which we mean that $\boldsymbol{\theta}_{\text{opt}}$ also is a good minimizer for:

$$\sum_{(X,\mathcal{P})} d\left(m^\dagger(\hat{f}(X;\boldsymbol{\theta})),\ m^\dagger(m(\mathcal{P}))\right).$$

For a sufficiently representative raster mapping this should also imply that $\boldsymbol{\theta}_{\text{opt}}$ also is a good minimizer for:

$$\sum_{(X,\mathcal{P})} d\left(m^\dagger(\hat{f}(X;\boldsymbol{\theta})),\ \mathcal{P}\right).$$

This is the metric we really intend to minimize, but can only do so implicitly with a good surrogate loss function, and a raster format that is sufficiently representative and targetable.

2.6.2 The "surface normal" raster format

We want to construct raster arrays (tiles) which represent different types of GIS data. Each such raster tile represents GIS data corresponding to a specific geographic area, the extent of which is specified by a bounding box $B(\boldsymbol{c}, w, h)$. In our case we intend to construct square tiles with areas of $4096\,\text{m}^2$, i.e. having width and height $w = h = 64\,\text{m}$. All of the bounding boxes have centroids $\boldsymbol{c}$ such that they are situated within the Norwegian municipality of Trondheim. Denote the set of all these bounding boxes as $\mathcal{B}$,

$$\begin{aligned}\mathcal{B} &= \left\{B(\boldsymbol{c}_1, 64\,\text{m}, 64\,\text{m}), B(\boldsymbol{c}_2, 64\,\text{m}, 64\,\text{m}), \dots B(\boldsymbol{c}_{|\mathcal{B}|}, 64\,\text{m}, 64\,\text{m})\right\} \\ &= \left\{B_1, B_2, \dots, B_{|\mathcal{B}|}\right\}.\end{aligned}$$

Our bounding boxes are constructed from cadastral plots in Trondheim, a set of size $|\mathcal{B}| = 64\,146$. We will restrict ourselves to constructing raster arrays of consistent resolution height $H = 256$ and resolution width $W = 256$. This results in 256^2 "pixelized" GIS measurements per raster tile, or equivalently, 16 raster pixel values per meter squared. A specific bounding box $B(\boldsymbol{c}, 64\,\text{m}, 64\,\text{m})$ will therefore be represented by a set of $256 \times 256 \times C$ raster arrays consisting of C raster channels. We have already described raster arrays representing different types of GIS data, such as RGB aerial photography ($C = 3$), LiDAR elevation data ($C = 1$), or building footprint segmentation masks ($C = 1$). The idea is now to construct an entirely new raster format which is able to represent a set of three-dimensional polygons,

$$\mathcal{P} = \{P_1, P_2, \dots, P_{|\mathcal{P}|}\}.$$

In our case $\mathcal{P}$ consists of all roof surfaces in Trondheim, a set of size $|\mathcal{P}| = 293\,336$. Our surface raster format, which we will refer to as the "surface normal" raster, is intended as a format which is both representative and targetable. We start by imposing two key assumptions on the polygons contained by $\mathcal{P}$.

A1 All polygons $P \in \mathcal{P}$ are **perfectly planar**. That is, for every polygon you can determine β_0, β_x, and β_y such that $z = \beta_0 + \beta_x x + \beta_y y$ for *all* vertices (x, y, z) representing the given polygon.

A2 All polygons $P \in \mathcal{P}$ are **mutually non-overlapping** when projected into the xy-plane, the xy-plane being the sea level.

These assumptions are in fact *not* satisfied by the Trondheim roof surface polygon dataset, but these issues are rectifiable. Assumption A1 is *nearly* satisfied and can be solved by regression with negligible error. This will be elaborated upon in Section 2.6.5. Assumption A2 can be solved with a suitable "conflict resolution" method which determines which polygon should be considered at each pixel location. Such a conflict resolution method will be described in Section 2.6.3. For now it is easier to describe the surface normal raster with these assumptions in place. We will introduce one final assumption:

A3 All polygons $P \in \mathcal{P}$ are **simple** polygons without any interior hulls, and are thus representable by a single exterior ring. The exterior ring is represented as an ordered sequences of (x, y, z) coordinate tuples, and we can therefore denote any polygon $P \in \mathcal{P}$ as

$$P = [(x_1, y_1, z_1), (x_2, y_2, z_2), \dots, (x_{|P|}, y_{|P|}, z_{|P|}), (x_1, y_1, z_1)],$$

where $|P|$ denotes the number of *unique* vertices. The first and last coordinate tuples in any linear ring are always identical in order to close the ring.

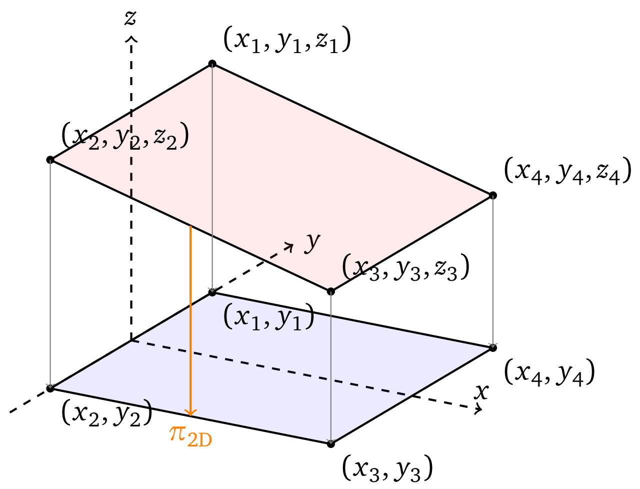
Assumption A3 is solely introduced for notational simplicity. Keeping track of several linear rings for each polygon will substantially complicate all expressions that will follow. There is nothing preventing the surface normal raster format for being generalized to polygons with interior hulls, and our implementation does in fact take interior hulls into account when constructing the surface normal raster.

Under the assumption of all polygons in $\mathcal{P}$ being perfectly planar we are able to decompose any polygon $P \in \mathcal{P}$ into two constituent sub-components: its *two-dimensional projection* and its *planar equation*. The first sub-component is the projection of the polygon into the xy-plane, $\pi_{2D}(P)$, making the three-dimensional polygon two-dimensional. This simple projection is simply performed by truncating the z-component of each (x, y, z)-vertex in the polygon:

$$\begin{aligned}\pi_{2D}(P) &= \pi_{2D}\big([(x_1, y_1, z_1), (x_2, y_2, z_2), \dots, (x_{|P|}, y_{|P|}, z_{|P|}), (x_1, y_1, z_1)]\big) \\ &= [(x_1, y_1), (x_2, y_2), \dots, (x_{|P|}, y_{|P|}), (x_1, y_1)]). \qquad \text{(Projection mapping)}\end{aligned}$$

The projection mapping $\pi_{2D}(P)$ has also been illustrated in Figure 2.16.

The second sub-component is the parametric description of the plane on which *all* vertices of the given polygon lie. Denote this mapping as $\boldsymbol{\beta}(P)$ and define it



The three-dimensional polygon is shown in red, while the two-dimensional projection is shown in blue.

Figure 2.16: Projection of three-dimensional polygon onto xy-plane by $\pi_{2D}(\cdot)$.

according to the following relationship,

$$\boldsymbol{\beta}(P) := \begin{bmatrix} \beta_0 \\ \beta_x \\ \beta_y \end{bmatrix}, \text{ such that } z = \beta_0 + \beta_x x + \beta_y y \text{ for } \textit{all} \text{ vertices } (x, y, z) \in P \tag{Planar mapping}$$

The original three-dimensional polygon can be easily reconstructed in a lossless manner from the two sub-components, as illustrated in Figure 2.17, while still being a less redundant representation of the polygon.

$$\begin{tikzcd}
P = [(x_1, y_1, z_1), (x_2, y_2, z_2), \ldots, (x_1, y_1, z_1)] \arrow[d, dash] & \\
\boxed{\text{decomposition}} \arrow[d] \arrow[dr, bend left] & \\
\pi_{2D}(P) = [(x_1, y_1), (x_2, y_2), \ldots, (x_1, y_1)] \arrow[d, dash] & \boldsymbol{\beta} = \begin{bmatrix} \beta_0 \\ \beta_x \\ \beta_y \end{bmatrix} \arrow[dl, dash, bend left] \\
\boxed{\text{reconstruction}} \arrow[d] & \\
P = [(x, y, \beta_0 + \beta_x x + \beta_y y) \mid (x, y) \in \pi_{2D}(P)] &
\end{tikzcd}$$

Figure 2.17: The decomposition and reconstruction of a three-dimensional polygon.

Now the idea is to create two separate rasters, one which represents $\pi_{2D}(P)$, and another one which represents $\boldsymbol{\beta}(P)$. The task of rasterizing $\pi_{2D}(P)$ is quite simple, it is a two-dimensional polygon which can be represented as a binary mask as explained earlier.

$$S_{i,j} = \begin{cases} 1, & \text{if there exists } P \in \mathcal{P} \text{ such that } \pi_{2D}(P) \text{ covers } \pi_B(i,\ j). \\ 0, & \text{otherwise.} \end{cases} \tag{semantic segmentation raster}$$

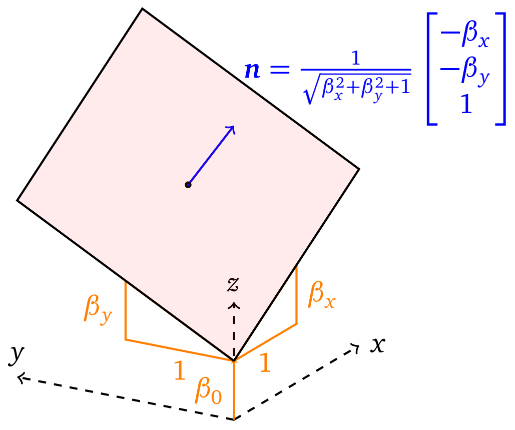
Where we have defined $\pi_B(i,\ j)$ as a function that maps array pixel coordinates (i, j) to the respective coordinate in the UTM coordinate system in which the polygons $P \in \mathcal{P}$ are specified. Given that we have a raster array of *resolution* height H and width W, which covers a geographic area represented by the bounding box $B = B(\mathbf{c}, w, h)$ centered at $\mathbf{c}$, with *geographic* width w and height h, and that $(i, j) = (0, 0)$ represents the upper left (northwestern) corner of the bounding box, then we have:

$$\pi_B(i,\ j) = \mathbf{c} + \frac{1}{2}\begin{bmatrix} -w \\ h \end{bmatrix} + \begin{bmatrix} \frac{w}{W} j \\ -\frac{h}{H} i \end{bmatrix}$$

When it comes to the rasterization of $\boldsymbol{\beta}(P) = [\beta_0, \beta_x, \beta_y]^T$, we start by noticing that the *normal vector* of the plane can be constructed from $\boldsymbol{\beta}(P)$ in the following manner:

$$\boldsymbol{\beta}(P) = \begin{bmatrix} \beta_0 \\ \beta_x \\ \beta_y \end{bmatrix} \iff \boldsymbol{n}(\boldsymbol{\beta}(P)) = \frac{1}{\sqrt{\beta_x^2 + \beta_y^2 + 1}} \begin{bmatrix} -\beta_x \\ -\beta_y \\ 1 \end{bmatrix} := \begin{bmatrix} n_x \\ n_y \\ n_z \end{bmatrix} \tag{Normal planar map}$$

The relationship between the equation of the plane and the associated surface normal vector is illustrated in Figure 2.18.



$\boldsymbol{n} = \frac{1}{\sqrt{\beta_x^2+\beta_y^2+1}} \begin{bmatrix} -\beta_x \\ -\beta_y \\ 1 \end{bmatrix}$
z
β_y
β_x
y
x
1
1
β_0

The plane is defined by $z = \beta_0 + \beta_x x + \beta_y y$. The elements of the parameter vector $\boldsymbol{\beta}$ are shown in orange, while the normal vector $\boldsymbol{n}$ is shown in blue.

Figure 2.18: The relationship between the equation of the plane, $\boldsymbol{\beta}(P)$, and the surface normal vector, $\boldsymbol{n}(\boldsymbol{\beta}(P))$.

The following restrictions on $\boldsymbol{n}\left(\boldsymbol{\beta}\left(P\right)\right)$ hold by construction,

$$\begin{aligned} \|\boldsymbol{n}\left(\boldsymbol{\beta}\left(P\right)\right)\|_2 = \sqrt{n_x^2 + n_y^2 + n_z^2} &\equiv 1, \\ n_z &\geq 0. \end{aligned}$$

With other words, the normal vector has been constructed such that it is of length 1 in the Euclidean norm, and that it *always* points upwards. Given that we can easily determine $\boldsymbol{\beta}\left(P\right)$ for any polygon, we now define the following raster format,

$$N_{i,j} = \begin{cases} \boldsymbol{n}\left(\boldsymbol{\beta}\left(P\right)\right), & \text{if } \pi_{\text{2D}}\left(P\right) \text{ covers } \pi_B\left(i,\ j\right). \\ \boldsymbol{0} := [0,0,0]^T, & \text{if no such } P \in \mathcal{P} \text{ exists.} \end{cases} \qquad \text{(surface normal raster)}$$

This is a raster format consisting of three raster channels, as each pixel location (i, j) in the raster "contains" a three-dimensional normal vector. It should now become apparent why we made assumption A2 earlier, that is, all polygons $P \in \mathcal{P}$ should be mutually non-overlapping after being projected by $\pi_{\text{2D}}\left(\cdot\right)$. If more than one polygon covers the coordinate $\pi_B\left(i,\ j\right)$, then the value for $N_{i,j}$ becomes ambiguous. You may now ask why we rasterize $\mathcal{P}$ into both a surface normal raster, N, *and* a semantic segmentation map, S, when S can be directly inferred from N. That is, we can formulate S in form of N,

$$S_{i,j} = \begin{cases} 0, & \text{if } N_{i,j} = [0,0,0]^T. \\ 1, & \text{otherwise.} \end{cases}$$

The reason is that this is a more targetable raster decomposition. We can now construct two relatively independent predictors, $\hat{f}_{\text{seg}}(X; \boldsymbol{\theta}_{\text{seg}})$ and $\hat{f}_{\text{norm}}(X; \boldsymbol{\theta}_{\text{norm}})$, which when combined form a main predictor $\hat{f}$ according to,

$$\hat{f}\left(X; \boldsymbol{\theta}_{\text{seg}}, \boldsymbol{\theta}_{\text{norm}}\right) = \begin{cases} \hat{f}_{\text{norm}}\left(X; \boldsymbol{\theta}_{\text{norm}}\right), & \text{if } \hat{f}_{\text{seg}}\left(X; \boldsymbol{\theta}_{\text{seg}}\right) \geq 0.5. \\ [0,0,0]^T, & \text{otherwise.} \end{cases}$$

Now an optimal value for $\boldsymbol{\theta}_{\text{seg}}$ can be found by training $\hat{f}_{\text{seg}}$ on S as the ground truth, using model architectures and loss functions from the semantic segmentation literature. A model architecture and loss function can be likewise be chosen for $\hat{f}_{\text{norm}}$ *entirely* independent of the segmentation problem at hand. The normal vector model architecture can for instance enforce $\|\hat{f}_{\text{norm}}\|_2 \equiv 1$, and the respective loss function can utilize this restriction of the model output and ground truth. Such a "separation of concerns" has been shown to be beneficial for several model architectures.

We can now rasterize any single polygon $P \in \mathcal{P}$ into two separate raster formats, S and N, as illustrated in Figure 2.19. When generalizing from a single three-dimensional polygon P to a set of polygons $\mathcal{P}$, we can iteratively fill in the values into a single pair of raster arrays, S and N, since the polygons are mutually non-overlapping. In order to map from this raster domain, represented by S and N, back to the original polygon domain, represented by $\mathcal{P}$, we propose the following pseudoinverse mapping, $m^\dagger$,

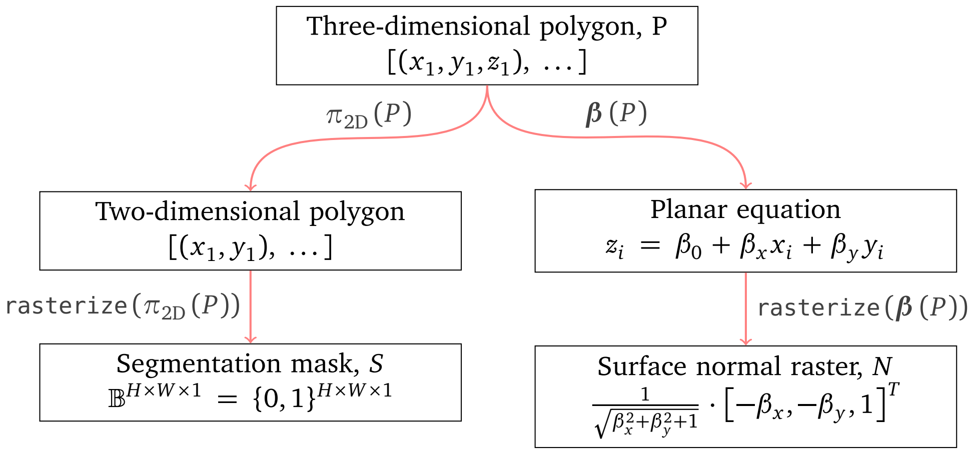

Three-dimensional polygon, P
$[(x_1, y_1, z_1),\ \dots]$
$\pi_{2D}(P)$
$\boldsymbol{\beta}(P)$
Two-dimensional polygon
$[(x_1, y_1),\ \dots]$
Planar equation
$z_i = \beta_0 + \beta_x x_i + \beta_y y_i$
$\texttt{rasterize}(\pi_{2D}(P))$
$\texttt{rasterize}(\boldsymbol{\beta}(P))$
Segmentation mask, S
$\mathbb{B}^{H\times W\times 1} = \{0,1\}^{H\times W\times 1}$
Surface normal raster, N
$\frac{1}{\sqrt{\beta_x^2+\beta_y^2+1}} \cdot [-\beta_x, -\beta_y, 1]^T$

Figure 2.19: The deconstruction of a three-dimensional polygon into two separate rasters formats.

Proposed m[†] – Partition semantic segmentation raster S such that all partitions form contiguous pixel areas which share the same value of N. Reconstruct β_0 by inference from LiDAR input data.

It should be noted that this is in fact a lossy decomposition, making the raster format not perfectly representative. The reason for this is that the parameter β_0 has been entirely discarded when we rasterize $\boldsymbol{\beta}(P)$. The exact conditions under which the surface raster format and associated inverse mapping becomes non-representative will be discussed in detail in Chapter 4. For now, suffice it to say that the following two conditions must be simultaneously satisfied.

1. There exists two polygons $P_1, P_2 \in \mathcal{P}$ such that $\pi_{2D}(P_1)$ and $\pi_{2D}(P_2)$ touch borders when rasterized. That is, P_1 and P_2 form one single, contiguous area in the semantic segmentation map S.
2. P_1 and P_2 share values for β_x and β_y, but *not* β_0. That is, the plane of P_1 and P_2 share the same *orientation* in space, but not the same *elevation*.

In order to correct for the lossy decomposition into S and N, we must introduce a third raster format into the mix. Given that the pixel coordinate $[i, j]^T$ maps to the geographic coordinate $[x, y]^T$, i.e. $\pi_B(i,\ j) = [x, y]^T$, then we define the *surface elevation raster* as:

$$Z_{i,j} = \begin{cases} \left[1, x, y\right] \boldsymbol{\beta}(P), & \text{if } \pi_{2D}(P) \text{ covers } \pi_B(i,\ j). \\ -\infty, & \text{if no such } P \in \mathcal{P} \text{ exists.} \end{cases} \qquad \text{(surface elevation raster)}$$

This *surface elevation* raster array enables us to define a perfectly representative raster format and associated reverse mapping. In practice however, it will be shown that the decomposition of $\mathcal{P}$ into S and N, discarding Z, is sufficient for accurate

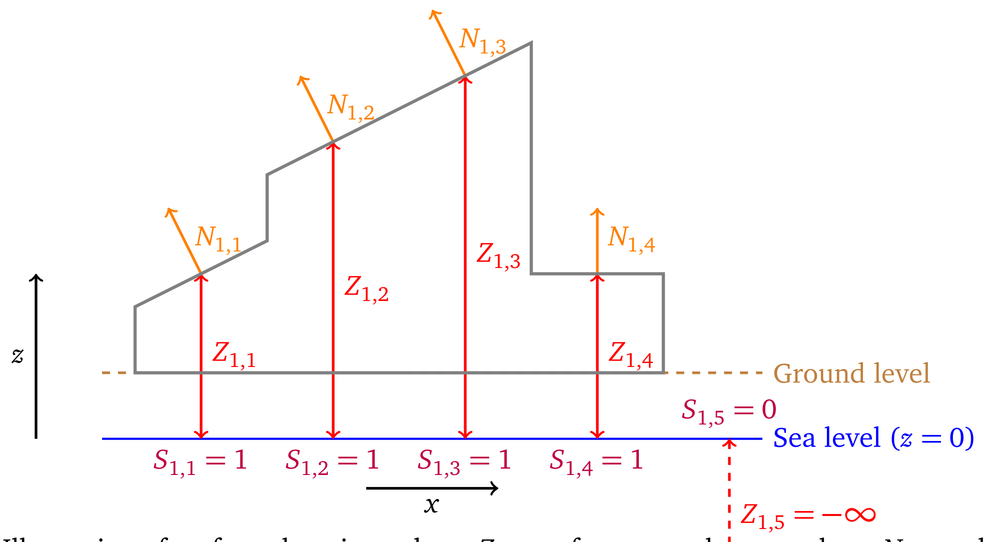


Illustration of surface elevation values, $Z_{i,j}$, surface normal array values, $N_{i,j}$, and segmentation mask, $S_{i,j}$. Slice for $i = 1$ and $1 \leq j \leq 5$.

Figure 2.20: Illustration of surface raster formats Z, N, and S.

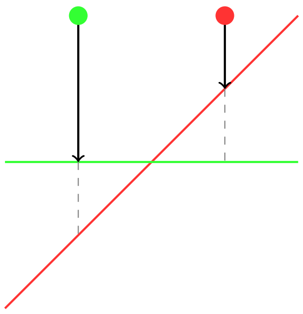
roof geometry inference. The three surface raster formats presented so far are all illustrated in Figure 2.20.

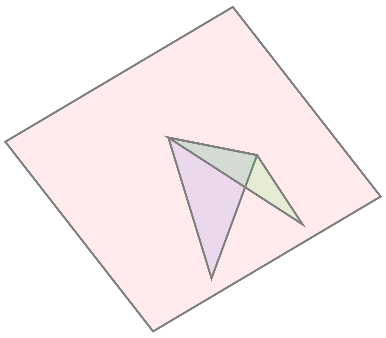
We will describe how to construct these three raster arrays from a practical implementation perspective. We start by summarizing the entire implementation in conceptual terms:

Given a set of bounding boxes $\mathcal{B}$ and a set of polygons $\mathcal{P}$:

- Construct R-tree index for polygon collection $\mathcal{P}$.
- Calculate and memoize $\boldsymbol{\beta}(P)$ for all $P \in \mathcal{P}$.
- For each bounding box $B \in \mathcal{B}$...
    - Determine the subset of polygons $\mathcal{P}_B \subset \mathcal{P}$ which is at least partially covered by the bounding box B. The aforementioned R-tree index is used in order to substantially speed up this spatial query.
    - Map all vertex coordinates of $P \in \mathcal{P}_B$ to the pixel coordinate system $[0, 255] \times [0, 255]$.
    - For each pixel coordinate (i, j)...
        - Determine subset of polygons $\mathcal{P}_{\text{cover}} \subset \mathcal{P}_B$ which covers the area represented by the pixel (i, j). If $|\mathcal{P}_{\text{cover}}| = 0$, set $Z_{i,j} = -\infty$, $N_{i,j} = [0, 0, 0]^T$, and $S_{i,j} = 0$, and continue onto next iteration of loop.
        - Fetch pre-calculated values for $\beta(P)$ for all $P \in \mathcal{P}_{\text{cover}}$.
        - Given that $\pi_B(i,\ j) = [x,\ y]^T$, calculate surface elevation $z_P = \beta_0 + \beta_x x + \beta_y y$ for all $P \in \mathcal{P}_{\text{cover}}$. Select polygon P_m with the greatest corresponding elevation value z_P.
        - Set $Z_{i,j} = z_{P_m}$, $N_{i,j} = \boldsymbol{n}(\boldsymbol{\beta}(P_m))$, and $S_{i,j} = 1$.

(a) Example of dormer roof causing overlapping polygons. This public domain image has been sourced from Wikimedia [15].


(b) Illustration of the “bird’s eye view decision rule”.


(c) Three-dimensional polygons which overlap when projected.

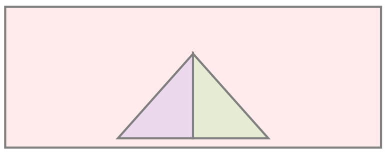

(d) Two-dimensional instance mask resulting from the bird’s view decision rule.

Figure 2.21: Illustration of overlapping surface polygons.

All of these steps will now be explained in more detail.

2.6.3 Handling overlapping surface polygons

Earlier we made the assumption that all polygons $P \in \mathcal{P}$ are mutually non-overlapping when projected into the xy-plane by $\pi_{2D}(P)$. In reality, this is *not* the case for our dataset. The non-overlapping assumption is often broken whenever small “sub-surfaces” are situated upon larger “main” surfaces. *Dormers*[8] often cause surfaces to overlap in two dimensions, for example. Figure 2.21c illustrates how the Trondheim surface polygon dataset $\mathcal{P}$ would represent a type of dormer, namely one main roof surface (shown in red), and a dormer represented by two separate flat surfaces (shown in blue and green). The key point is that the main roof surface is a rectangular polygon *without* any interior hull where the dormer surfaces are located. This results in the two dormer surfaces overlapping with the single main roof surface when they are projected into the horizontal plane. As

[8] *Dormer* — Roof surface protruding out from the main roof surface, often in order to protect a loft window. A dormer can be seen in Figure 2.21a

mentioned before, this causes problems with the definition of the surface normal vector raster, N, and surface elevation raster, Z, as the choice of polygon becomes ambiguous. This ambiguity needs to be resolved with a *decision rule*, a rule which determines which single polygon should be considered as applicable at locations covered by multiple polygons. For our dataset, the “bird’s eye view decision rule” produces sensible results in most cases, and is defined as follows.

Bird’s eye view decision rule — *The polygon with the greatest elevation at a given coordinate is the applicable polygon at that point.*
More formally, for a pixel location (i, j) corresponding to the geographic coordinate $\pi_B(i,\ j) = [x, y]^T$, we define the *polygon shadow subset* $\mathcal{P}(x, y)$ as

$$\mathcal{P}(x,y) := \{P \mid P \in \mathcal{P} \text{ and } \pi_{2\text{D}}(P) \text{ covers } (x,y)\}\,.$$

The *applicable polygon* at the pixel location (i, j), and the corresponding geographic coordinate (x, y), is then defined as

$$P(x,y) := \underset{P \in \mathcal{P}(x,y)}{\text{argmax}}\ [1, x, y]\,\boldsymbol{\beta}\,(P)\,.$$

In other terms, the *polygon shadow subset* $\mathcal{P}(x, y)$ consists of all polygons that are intersected by the line drawn from $(x, y, -\infty)$ to (x, y, ∞), hence casting a shadow on the point (x, y) when the sun stands at zenith (directly overhead). The *applicable polygon* is the single polygon which ends up being hit by direct sunlight at the given geographic point, thus casting a shadow upon all other polygons in the polygon shadow subset. The bird’s view decision rule has been illustrated in Figure 2.21b, and Figure 2.21d is an example of a two-dimensional instance segmentation map generated with this rule from the three-dimensional roof structure shown in Figure 2.21c. We can now update the definition of surface elevation raster Z and the normal vector raster N, taking this new decision rule into account,

$$\begin{aligned}
N_{i,j} &= \begin{cases} \boldsymbol{n}\left(\boldsymbol{\beta}\left(P\left(x,y\right)\right)\right), & \text{if } |\mathcal{P}(x,y)| \geq 1. \\ \mathbf{0} := [0,0,0]^T, & \text{otherwise.} \end{cases} \\
Z_{i,j} &= \begin{cases} \Big[1, x, y\Big]\,\boldsymbol{\beta}\left(P(x,y)\right), & \text{if } |\mathcal{P}(x,y)| \geq 1. \\ -\infty, & \text{otherwise.} \end{cases}
\end{aligned} \tag{2.1}$$

We will construct a computational procedure which is able to rasterize a given polygon $P \in \mathcal{P}$ relative to a given bounding box $B \in \mathcal{B}$. Denote this procedure as `Rasterize`$(P,\ B)$, and say that this procedure returns three rasters: a surface elevation raster Z_p, a surface normal raster N_p, and a two-dimensional segmentation mask S_p. We now want to generalize from the rasters Z_p, N_p, S_p which represent a *single* polygon $P \in \mathcal{P}$, to rasters Z, N, S representing an arbitrarily sized *set* of polygons $\mathcal{P}$. We start by initializing these arrays (Z, N, and S) with temporary placeholder values,

$$\begin{aligned} Z_{i,j} &= -\infty, & Z &\in \mathbb{R}^{H\times W\times 1}, \\ N_{i,j} &= [0,0,0]^T, & N &\in [0,1]^{H\times W\times 3}, \\ S_{i,j} &= 0, & S &\in \mathbb{B}^{H\times W\times 1}. \end{aligned}$$

This initialization is implemented by the procedure `ConstructPlaceholderArrays()`. Now, the idea is to iterate over the polygons $P \in \mathcal{P}$, rasterize them individually with `Rasterize()`, and fill these arrays into the global mutable arrays Z, N, and S by some decision rule implemented by `FillValues()`. The main loop will look like follows,

```
def PreprocessRaster($\mathcal{P}$, $\mathcal{B}$):
  for $B$ in $\mathcal{B}$:
    $Z, N, S \leftarrow$ ConstructPlaceHolderArrays()
    for $P \in \mathcal{P}$:
      $Z_p, N_p, S_p \leftarrow$ Rasterize($P$, $B$)
      FillValues(from $= [Z_p, N_p, S_p]$, into $= [Z, N, S]$)
    SaveToHashLookup(data $= [Z, N, S]$, hash $= B$).
```

`SaveToHashLookup()` is a procedure which persists the arrays Z, N, and S, to disk for later retrieval. The *bird's view decision rule* can therefore be implemented in the following procedural manner,

```
def FillValues(from $= [Z_p, N_p, S_p]$, into $= [Z, N, S]$):
  for $i \in \{0, 1, \ldots, 255\}$:
    for $j \in \{0, 1, \ldots, 255\}$:
      If $(S_p)_{i,j} = 0$ or $(Z_p)_{i,j} \leq Z_{i,j}$: continue onto next iteration of inner loop
      Insert $S_{i,j} \leftarrow 1$, $N_{i,j} \leftarrow (N_p)_{i,j}$, and $Z_{i,j} \leftarrow (Z_p)_{i,j}$.
```

2.6.4 Handling expensive spatial queries

The naive implementation of `PreprocessRaster`($\mathcal{P}$, $\mathcal{B}$) provided above invokes `Rasterize`(P, B) a total of $|\mathcal{P}||\mathcal{B}|$ times. In our case, using bounding boxes and roof surface polygons from Trondheim, we have $|\mathcal{P}| = 293\,336$ and $|\mathcal{B}| = 64\,146$, and thus $|\mathcal{P}||\mathcal{B}| = 18\,816\,331\,056$. This is obviously an infeasible number of iterations, no matter how efficiently `Rasterize`(P, B) is implemented. A better approach is to reduce the number of iterations performed by the inner loop over $\mathcal{P}$, utilizing the fact that only polygons $P \in \mathcal{P}$ for which $\pi_{2\text{D}}(P)$ intersects with B will ever contribute with values to the raster arrays. We implement such a filter procedure in `IntersectingFilter()`:

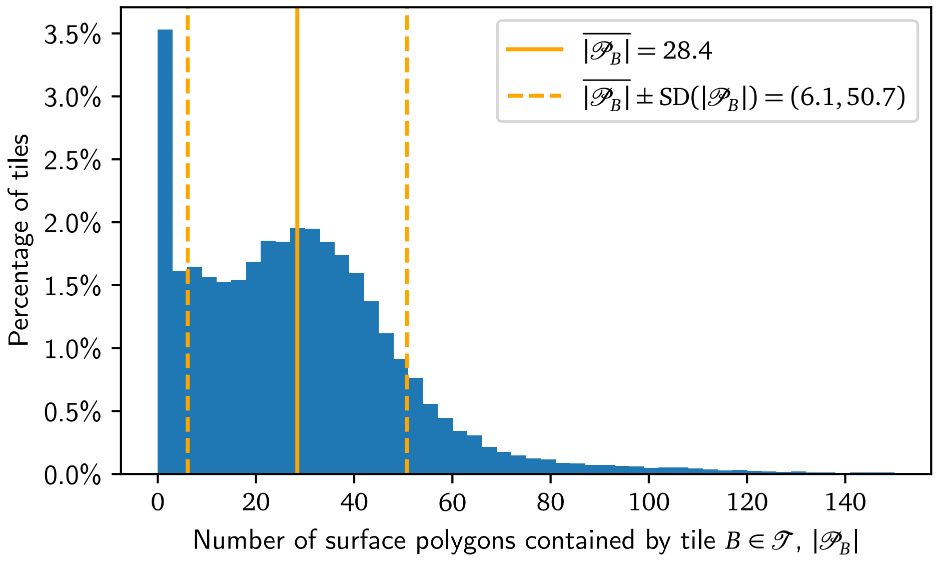


Figure 2.22: Trondheim distribution of number of intersecting surface polygons contained by each raster tile $B \in \mathcal{B}$.

```
def IntersectFilter($\mathcal{P}$, $B$):
    return $\{P \mid P \in \mathcal{P}$ and $\pi_{2D}(P)$ intersects with $B\}$
```

Using `IntersectingFilter()` we can improve upon `PreprocessRaster()` as follows:

```
def PreprocessRaster($\mathcal{P}$, $\mathcal{B}$):
    for $B$ in $\mathcal{B}$:
        $Z, N, S \leftarrow$ ConstructPlaceHolderArrays()
        $\mathcal{P}_B \leftarrow$ IntersectingFilter($\mathcal{P}$, $B$)
        for $P \in \mathcal{P}_B$:
            $Z_p, N_p, S_p \leftarrow$ Rasterize($P$, $B$)
            FillValues(from = $[Z_p, N_p, S_p]$, into = $[Z, N, S]$)
        SaveToHashLookup(data = $[Z, N, S]$, hash = $B$).
```

If we denote the *average* size of the set `IntersectFilter`($\mathcal{P}$, B) over $B \in \mathcal{B}$ as $\overline{|\mathcal{P}_B|}$, that is,

$$\overline{|\mathcal{P}_B|} = \frac{1}{|\mathcal{B}|} \sum_{B \in \mathcal{B}} \Big| \{P \mid P \in \mathcal{P} \text{ and } \pi_{2D}(P) \text{ intersects with } B\} \Big|,$$

then the number of invocations of `Rasterize`(P, B) is reduced to $|\mathcal{B}|\overline{|\mathcal{P}_B|}$. Specifically, for our dataset $\overline{|\mathcal{P}_B|} \approx 28.4$ and $|\mathcal{B}|\overline{|\mathcal{P}_B|} = 1\,822\,376$. The full empirical distribution of $|\mathcal{P}_B|$ for the Trondheim dataset is provided in Figure 2.22. Although the number of invocations of `Rasterize`(P, B) has been greatly reduced, there has been introduced an additional cost of invoking IntersectFilter($\mathcal{P}$, B) a total of $|\mathcal{B}|$ times. A naive implementation of IntersectFilter($\mathcal{P}$, B) has computational cost that is linearly proportional to $|\mathcal{P}|$, which makes the introduction of IntersectFilter() moot from an asymptotic perspective (it results in a substantial speedup for our finitely sized dataset, however). The solution is to pre-compute a so-called R-tree spatial index, which when first computed, allows for almost instantaneous intersection filtering[16].

```
def PreprocessRaster($\mathcal{P}$, $\mathcal{B}$):
  RTreeIndex $\leftarrow$ GenerateRTreeIndex($\mathcal{P}$)
  for $B$ in $\mathcal{B}$:
    $Z, N, S \leftarrow$ ConstructPlaceHolderArrays()
    $\hat{\mathcal{P}}_B \leftarrow$ RTreeIndex($B$)
    for $P \in \hat{\mathcal{P}}_B$:
      $Z_p, N_p, S_p \leftarrow$ Rasterize($P$, $B$)
      FillValues(from $= [Z_p, N_p, S_p]$, into $= [Z, N, S]$)
    SaveToHashLookup(data $= [Z, N, S]$, hash $= B$).
```

The computational cost of RTreeIndex(B) is $\mathcal{O}(\log|\mathcal{P}|)$, making the time complexity of PreprocessRaster($\mathcal{P}$, $\mathcal{B}$) as a whole $\mathcal{O}\left(|\mathcal{B}|\overline{|\mathcal{P}_B|}\log|\mathcal{P}|\right)$. It should also be mentioned that PreprocessRaster() is a highly parallelizable process. The introduction of an R-tree index and splitting the processing of $\mathcal{B}$ over 12 cores/24 threads results in a reduction in computation from approximately 24 hours to only just over 40 minutes for the Trondheim dataset.

2.6.5 Handling non-planar polygons

Let $|P|$ denote the number of unique vertices required in order to represent a polygon $P \in \mathcal{P}$, and assume that all polygons $P \in \mathcal{P}$ are simple polygons without any interior hulls (assumption A3). We can therefore denote the polygon P as

$$\begin{aligned} P &= [(x_1, y_1, z_1), (x_2, y_2, z_2), \ldots, (x_{|P|}, y_{|P|}, z_{|P|}), (x_1, y_1, z_1)] \\ &:= [\boldsymbol{p}_1, \boldsymbol{p}_2, \ldots, \boldsymbol{p}_{|P|}, \boldsymbol{p}_1]. \end{aligned}$$

In Section 2.6.2 we imposed assumption A1 on $\mathcal{P}$, namely that *all* the three-dimensional vertices (x, y, z) required in order to represent a given polygon $P \in \mathcal{P}$ must lie exactly flat on a three-dimensional plane. That is, there exists $\boldsymbol{\beta}(P) = [\beta_0, \beta_x, \beta_y]^T$ such that

$$z = \beta_0 + \beta_x x + \beta_y y \text{ for all vertices } (x, y, z) \text{ representing } P.$$

Three unique polygon vertices, $\boldsymbol{p}_i$, $\boldsymbol{p}_j$, and $\boldsymbol{p}_k$, which do *not* form a straight line is sufficient in order to perfectly represent the plane. It is therefore easy to implement a procedure for calculating $\boldsymbol{\beta}(P)$, the definition of which was provided in Equation (Planar mapping) on Page 26, and likewise for $\boldsymbol{n}(\boldsymbol{\beta}(P))$ as defined in Equation (Normal planar map) on Page 27. In reality assumption A1 does *not* hold for our dataset. The polygon vertices are more accurately described by the following relationship

$$z = \beta_0 + \beta_x x + \beta_y y + \varepsilon \text{ for all vertices } (x, y, z) \text{ representing } P,$$

for some error term ε caused by measurement errors, data entry errors, or other unknown causes. The accurate determination of $\boldsymbol{\beta}(P)$ depends on the behaviour of the error term ε and will heavily depend by the nature of the surface polygon dataset. We will argue that an ordinary least squares predictor, denoted as $\widehat{\boldsymbol{\beta}}(P)$, is sufficiently accurate for our use, where we define $\widehat{\boldsymbol{\beta}}(P)$ as

$$\widehat{\boldsymbol{\beta}}(P) := \left(X_{xy}^T X_{xy}\right)^{-1} X_{xy}^T \boldsymbol{z}, \quad \text{where } X_{xy} = \begin{bmatrix} 1 & x_1 & y_1 \\ 1 & x_2 & y_2 \\ \vdots & \vdots & \vdots \\ 1 & x_{|P|} & y_{|P|} \end{bmatrix}, \boldsymbol{z} = \begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_{|P|} \end{bmatrix}.$$

In order to investigate if $\widehat{\boldsymbol{\beta}}(P)$ is a sufficiently accurate estimator for $\boldsymbol{\beta}(P)$, we will compare the original vertex coordinates (x, y, z) with the fitted vertex coordinates $(x, y, \hat{z})$, where $\hat{z}$ is constructed from the linear predictor

$$\widehat{z} = \widehat{f}_z\left(x, y; \widehat{\boldsymbol{\beta}}(P)\right) = \widehat{\beta}_0 + \widehat{\beta}_x x + \widehat{\beta}_y y.$$

We can now define the *vertex residuals* as

$$\begin{aligned} e_i &:= z_i - \hat{z}_i \\ &= z_i - [1, x_i, y_i]\, \widehat{\boldsymbol{\beta}}(P), \quad \text{for } i \in \{1, 2, \dots, |P|\}. \end{aligned}$$

And also the coefficient of determination, R^2, for a given polygon regression fit.

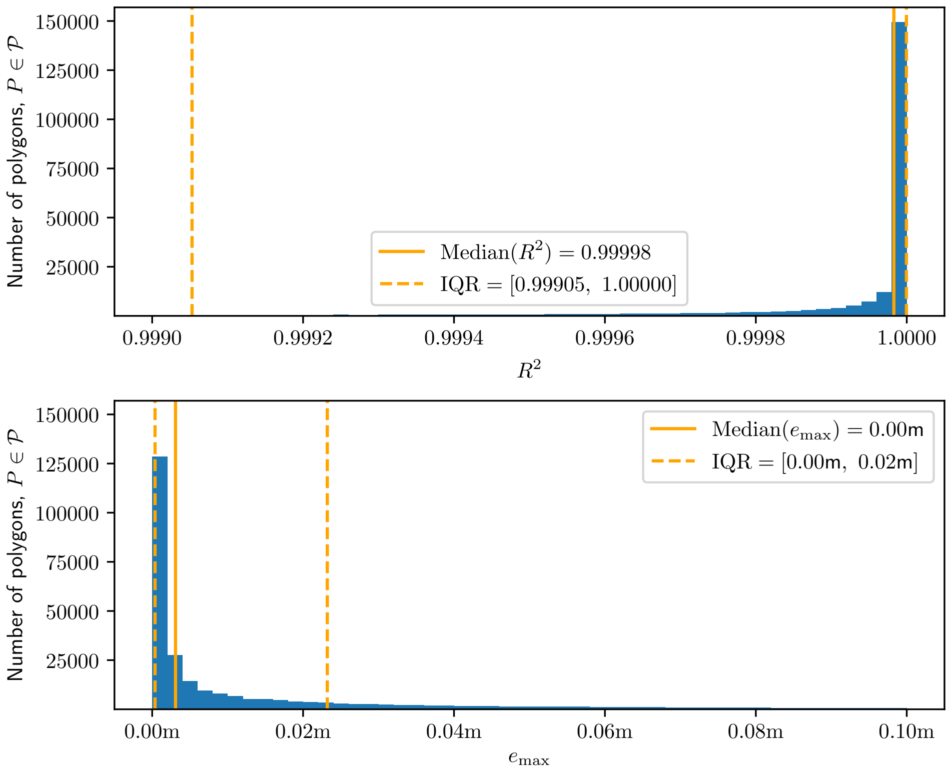
$$R^2 := 1 - \frac{\sum_{i=1}^{|P|} e_i^2}{\sum_{i=1}^{|P|} (z_i - \bar{z})^2}, \text{ where } \bar{z} := \frac{1}{|P|} \sum_{i=1}^{|P|} z_i.$$

The distribution of R^2 over all Trondheim polygons is shown in the top part of Figure 2.23, with a median R^2 value of 0.99998, which definitely should be considered a good fit. Of similar interest is the *greatest absolute vertex residual* of a given polygon, defined as

$$e_{\max} := \max_{i \in \{1, \dots, |P|\}} |e_i|,$$

the distribution of which is plotted in the bottom part of Figure 2.23. This "maximum vertex deviation" goodness of fit metric also shows favorable results. A final quality assurance metric can be inspected in order to confirm that the polygons have been correctly fitted; the *LiDAR residual* defined as

$$E_{i,j} := X_{i,j} - Z_{i,j}, \text{ for indices } (i, j) \text{ where } S_{i,j} = 1.$$



Top — The distribution of the coefficient of determination, R^2, for an ordinary least squares estimator for $\boldsymbol{\beta}(P)$ over all Trondheim polygons $P \in \mathcal{P}$.
Bottom — The distribution of the maximum distance between a vertex in a polygon and its fitted plane, $e_{\max}$.

Figure 2.23: Statistical summary of the regression fits of the Trondheim surface polygons.

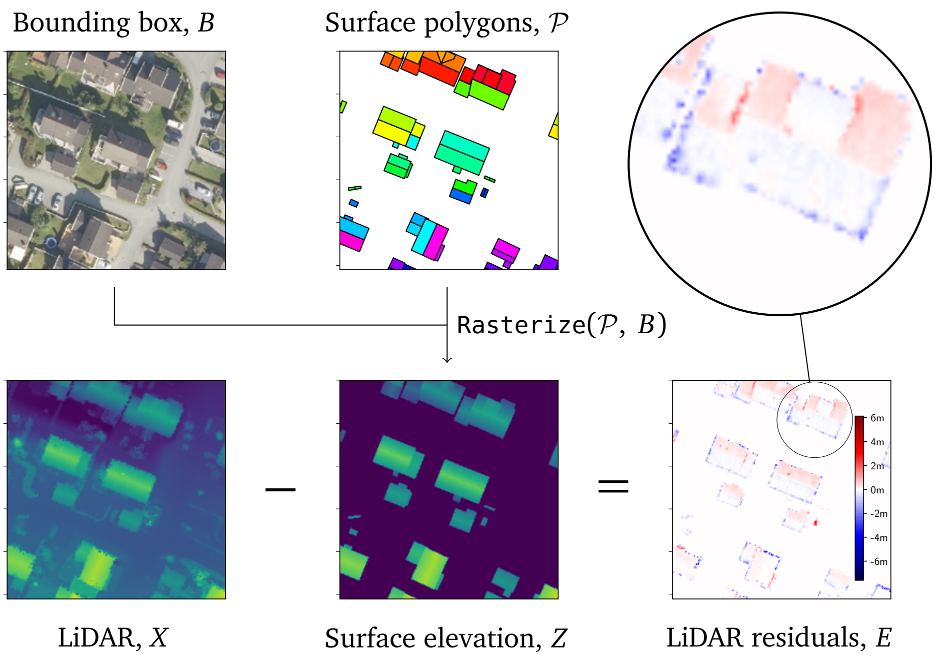


Figure 2.24: The construction of LiDAR residuals from a surface elevation raster.

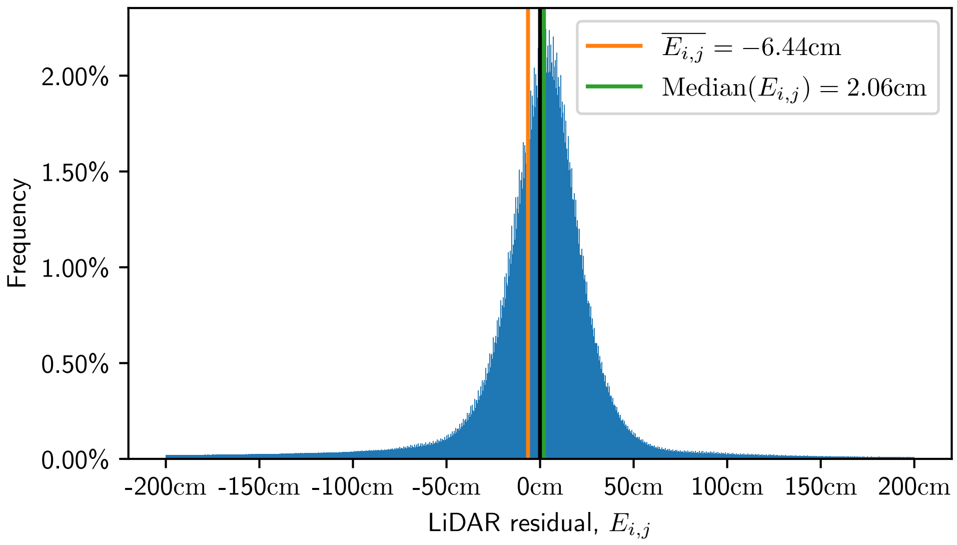


Figure 2.25: Distribution of LiDAR residuals calculated over all bounding boxes $B \in \mathcal{B}$ for the Trondheim dataset.

The LiDAR residual is with other words the difference between the roof height calculated from the three-dimensional roof surface polygon dataset at a given coordinate, and the LiDAR height measurement at the same coordinate. An illustration of the LiDAR residual and how it is constructed is provided in Figure 2.24, and the distribution of the Trondheim LiDAR residuals is provided in Figure 2.25. The discrepancy between the polygon surface elevation and LiDAR measurements can be attributed to three main mechanisms:

1. *Positive* residuals caused by objects and structures placed upon roof surfaces, e.g. chimneys, which are not part of the surface polygon dataset itself.
2. *Negative* residuals caused by LiDAR measurements taken close to the edges of roof surface situated adjacent to vertical drops, measuring the height of building walls or the nearby ground instead.
3. *Normally distributed* residuals caused by LiDAR measurement errors centered around zero but with non-zero variance.

The first two mechanisms can be observed within the focused area in Figure 2.24, while the zero-centered measurement error can be observed in the bell shaped distribution shown in Figure 2.25. Mechanism 1 is more common than mechanism 2, but smaller in absolute extent, causing the empirical mean of the LiDAR residuals to become negative ($-6.44\,\text{cm}$), while the empirical median is positive ($2.06\,\text{cm}$).

It is possible to discard any polygon $P \in \mathcal{P}$ that performs badly under a given goodness of fit metric, R^2 or e_{max} for instance, but this would also require us to discard all the respective bounding boxes $B \in \mathcal{B}$ that $\pi_{\text{2D}}(P)$ at least partially intersects with. This could be done in order to prevent training a machine learning model on false ground truth data. We do not consider this to be necessary for the Trondheim polygon dataset, but it may be necessary for other surface polygon datasets of lower quality.

2.7 Overview

The preprocessing pipeline responsible for transforming raw GIS data into a format suitable for machine learning is outlined in Figure 2.26.

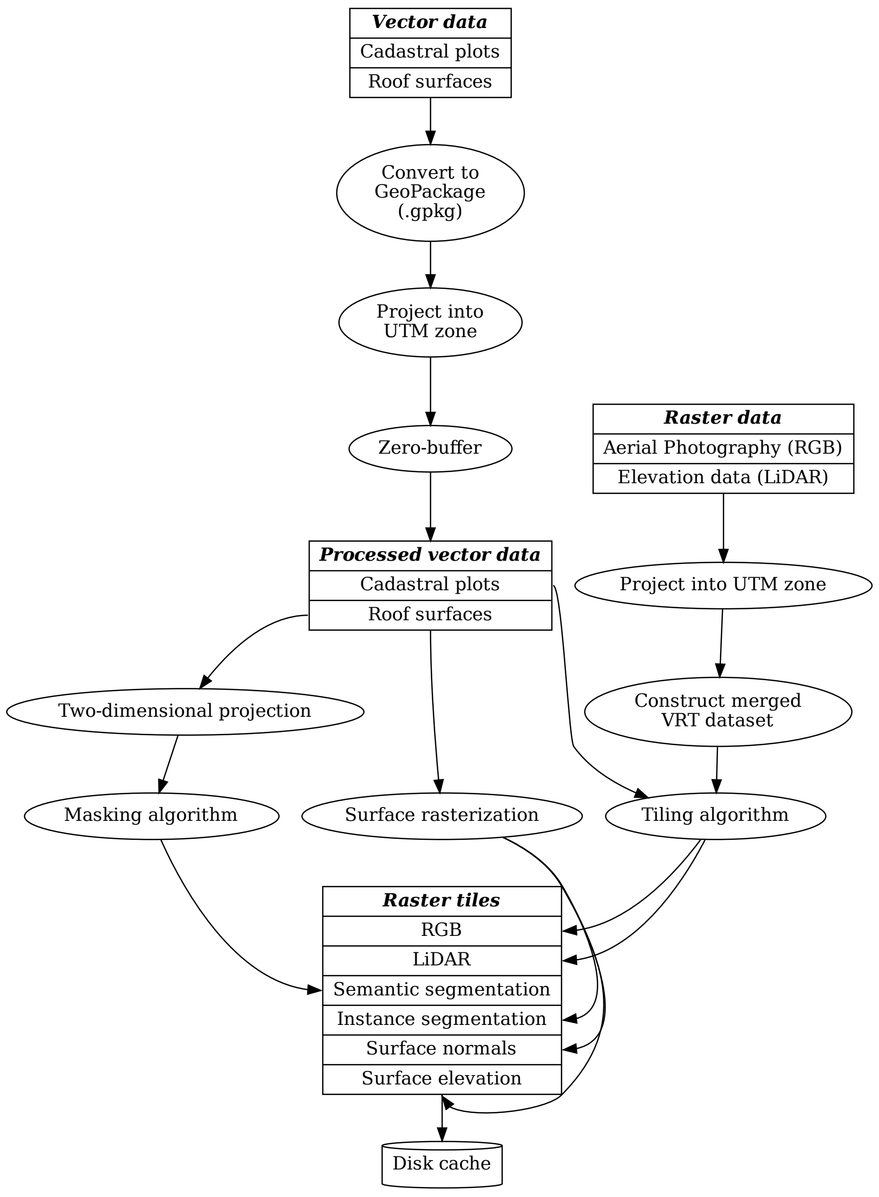


Figure 2.26: Overview of the GIS preprocessing pipeline developed in order to train machine learning models on geospatial data.

Chapter 3

Modeling

The field of *computer vision* got started in the early 1970s [17, p. 10], where one of the problems being tackled is the three-dimensional reconstruction of a scene from two-dimensional data. 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 [17, p. 15]. The statistical approach eventually morphed into the field of *machine learning*, where most of the research advances are made today [17, p. 17].

We have already provided an overview of the problem domains of interest in Section 1.1.1, but we will now provide a more in-depth description of all the relevant theoretical concepts. *Convolutional neural networks* (CNNs) have been applied to image segmentation problems with great success [6, p. 1], and Section 3.1 provides a theoretic overview of the elementary building blocks used to construct modern CNN architectures. Section 3.2 will go more in-depth into *image segmentation* specifically, treating topics such as popular evaluation metrics, optimization losses, and the current state-of-the-art. The section is finalized by a description of the CNN architecture used in this work, namely the U-Net architecture. Having explained the relevant concepts of semantic segmentation, Section 3.3 transitions to the problem of predicting rasterized surface normal vectors. The section starts by describing previous related work, followed by a description of a modified U-Net architecture for predicting such vectors. Finally, suitable loss functions for surface normal vector predictions are discussed. Sections 3.4 and 3.5 end this chapter by describing the general process of training CNN networks and how the remote sensing raster data should be normalized before being passed into the neural networks.

3.1 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 3.1 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.

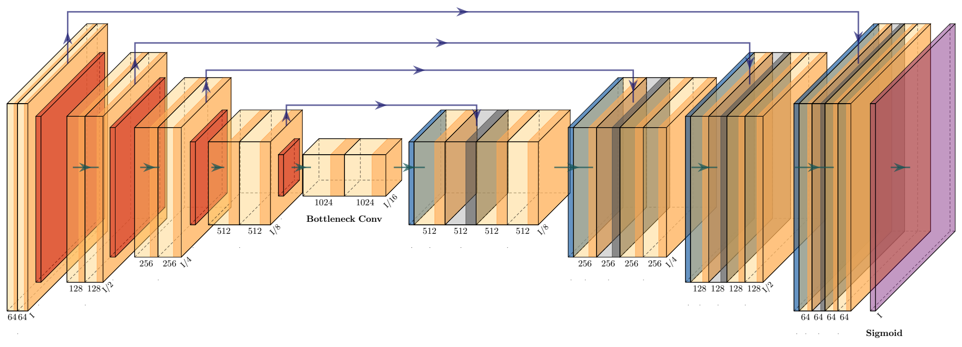


Figure 3.1: Illustration of the U-Net architecture for single-class segmentation, a typical example of an *encoder/decoder* structure. Convolution layers are shown in orange, and max pooling layers in red. Arrows indicates how data is forwarded through the network, top arrows being *skip connections*. 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 3.1. 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.

3.1.1 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 3.2.

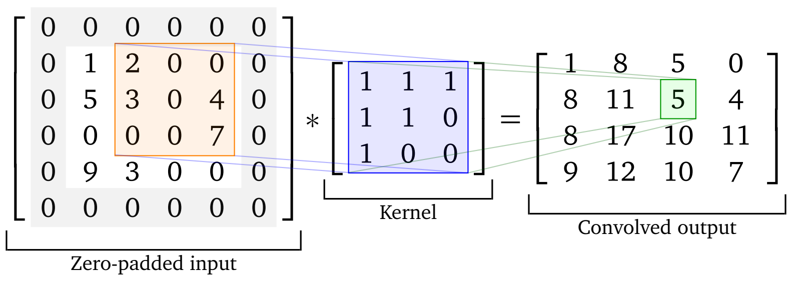


Figure 3.2: Visualization of a kernel convolution with a 3×3 kernel over an image of size 4×4 with additional zero-padding and stride size of 1×1. The *receptive field* is shown in orange, the respective kernel weights in blue, and the resulting convolution output in green. The zero padding of the input 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 [17, 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 such pixels. Secondly, a horizontal stride of $W_s > 1$ or a vertical stride of $H_s > 1$ 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

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

as shown by [18]. 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

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

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 $(H/H_s) \times (W/W_s)$ [18]. Such a padding is shown in gray in Figure 3.2.

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 [19]. 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.

3.1.2 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 [20, p. 165], but have since been shown to be a theoretical prerequisite of the *universal approximation* property of artificial neural networks [21, 22]. 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 [23], 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. A closely related activation function, with many of the same properties except that it maps into the range $(-1,1)$, is the *hyperbolic tangent* (tanh) function defined by

$$\text{TanH}(x) := \frac{e^x - e^{-x}}{e^x + e^{-x}}. \qquad \text{(Tanh activation function)}$$

Although the sigmoid activation function has strong biological [23] and theoretical [21] 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 [24] by Hahnloser *et al.* in year 2000. It is defined as

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

The ReLU activation function has become the dominant activation function for use in neural networks in recent years [25, p. 438] as it has been empirically shown to adapt well to deeper neural networks [26].

3.1.3 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 [19, 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 [27]. 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 3.3.

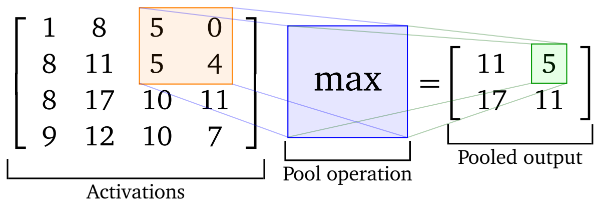


Figure 3.3: Example of a *max-pooling* operation with a receptive field of size 2×2 and an identical stride size. The receptive field is shown in orange and the respective pooled output is shown in green.

As can be seen in Figure 3.3, 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.

3.1.4 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 $\mathrm{Var}(a) = 1$ 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.

3.1.5 Dropout

Dropout is a regularization technique for neural networks intended to prevent “complex co-adaption of feature detectors” [29]. 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 [30]. 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 [31]. Alternatively, *stochastic depth dropout* randomly selects entire layers to be dropped, replacing them with identity functions instead [32]. Dropout can also be integrated into max pooling layers, ignoring values at random during the search for the maximum value in the receptive field [33]. This has become known as *max-pooling dropout* and is illustrated in Figure 3.4.

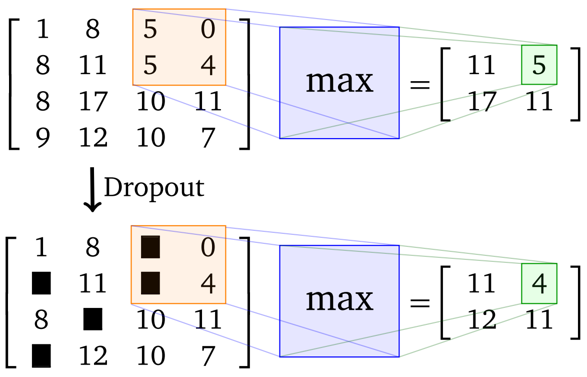


Figure 3.4: An example application of *max-pooling dropout* using a receptive field and stride of size 2×2. A dropout probability of $p = 0.25$ has been used. Dropped values are shown as black boxes.

3.2 Semantic Segmentation

Now that we have explained the generic building blocks of CNNs, we will go into more specific details related to *semantic image segmentation*. The current state-of-the-art, including the model architecture which we will use, will be discussed, but we will start by describing popular segmentation *metrics* and *losses*.

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 [20, 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.

3.2.1 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 3.5.

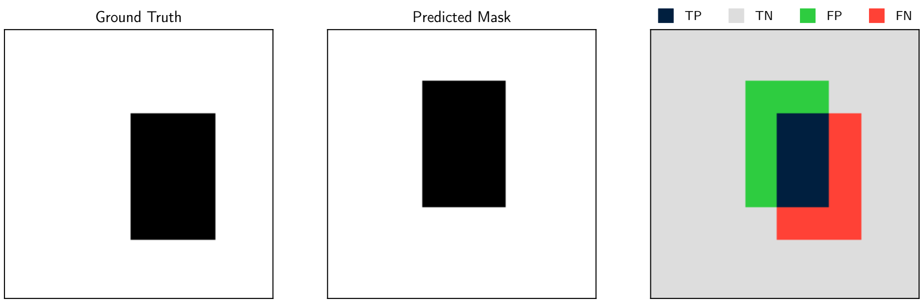


Figure 3.5: Binary segmentation problem of size 256 × 256. The ground truth, a rectangle of size 120 × 80 is shown on the left. The “predicted” mask, shown in the middle, is of the same size, but offset by (−30, −30). The right figure shows the visual equivalent of a confusion matrix. True positives are shown in dark blue, true negatives in light gray, false positives in green, and false negatives in red.

The 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:

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

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.

3.2.2 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 [34]. Notice how the IoU metric is bounded between 0 and 1; $\text{IoU} = 0$ represents a complete “predictive miss”, while $\text{IoU} = 1$ represents a prediction in perfect accordance with the ground truth. A visualization of this metric is given in Figure 3.6 below.

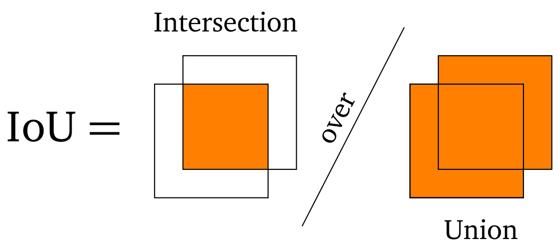


Figure 3.6: Visualization of single-class IoU metric.

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:

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

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 3.7.

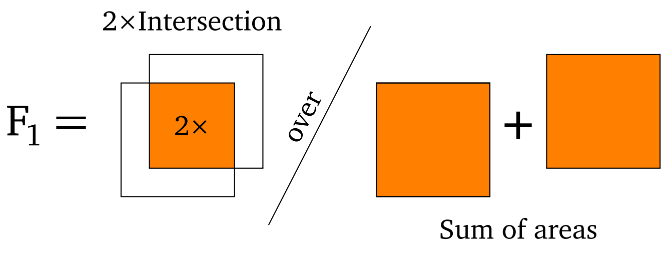


Figure 3.7: Visualization of the single-class dice coefficient metric, also known as the F_1 score.

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]

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

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:

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

[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`.

The IoU is *always* less than or equal to the dice coefficient, but never smaller than half the value. The fraction IoU/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.

3.2.3 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 $S \in \mathbb{B}^{H \times W}$ 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 S, the probability estimate denoted as $\widehat{S} = \hat{f}(X; \boldsymbol{\theta}) \in [0,1]^{H \times W}$. For notational convenience we will use a linear index in order to denote single matrix elements, for instance $\widehat{S}_i \in [0,1]$ for $i = 1, \dots, HW$.

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

$$\mathcal{L}_{\text{BCE}}(\widehat{S}; S) = -\sum_{i=1}^{HW} S_i \log(\widehat{S}_i) + (1 - S_i) \log(1 - \widehat{S}_i). \tag{3.1}$$

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 [35, 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* [36], defined by

$$\mathcal{L}_{\text{SJL}}(\widehat{S}; S) = 1 - \frac{\sum\limits_{i=1}^{HW} \widehat{S}_i S_i}{\sum\limits_{i=1}^{HW} \left(\widehat{S}_i + S_i - \widehat{S}_i S_i\right)} \approx 1 - \text{IoU}. \tag{3.2}$$

Notice that if $\widehat{S}_i$ is restricted to only take values in $\{0, 1\}$ then $\mathcal{L}_{\text{SJL}}$ becomes equal to $1 - \text{IoU}$. 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 [37].

$$\mathcal{L}_{\text{SDL}}(\widehat{S};S)=\frac{2\sum_{i=1}^{HW}\widehat{S}_iS_i}{\sum_{i=1}^{HW}\widehat{S}_i^2+\sum_{i=1}^{HW}S_i^2}\approx 1-F_1. \tag{3.3}$$

Optimizing these two metric-sensitive losses have been shown theoretically and empirically to indirectly maximize their respective surrogate metrics [35]. 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 [35] that these two metric-sensitive losses are equally good surrogates for each others metrics, and the choice is therefore mainly a preferential one.

3.2.4 State-of-the-art

At the time of this writing, CNNs have largely surpassed all previous methods for performing image segmentation [34], 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* [34]. 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 [38] won several image classification competitions when it was first published in 2012, including the ILSVRC-2012 competition [34]. By employing five convolutional layers, max-pooling layers, ReLU activation functions, and dropout, followed up by a fully connected feedforward classification network, it outperformed the 2nd place contender by a relatively large margin.

The **VGG-16** architecture [39] published in 2014 distinguished 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 [34].

Substantially deep networks are prone to overfitting and are subject to additional computational overhead. The **GoogLeNet** architecture [40] 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 [41] 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 [34]. 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 *et al.* in 2015 [42]. 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** [44], 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 [45], and the subsequent improvements **Fast R-CNN** [46] and **Faster R-CNN** [47], 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** [48], 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 [49] by Sabour *et al.*, it has since been applied to segmentation tasks as well. One such adaption is the **SegCaps** architecture [50]. 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.

3.2.5 The U-Net model architecture

We have chosen the U-Net architecture for segmenting roof structures and we will present numerical experiments in Chapter 5. The U-Net model has already been briefly described in the previous section and the architecture has been illustrated in Figure 3.1, 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 3.8. 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 3.8. Each decoder block utilizes batch normalization and max-pooling dropout. 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 v2.1, yielding a final network with 7 025 329 trainable parameters.

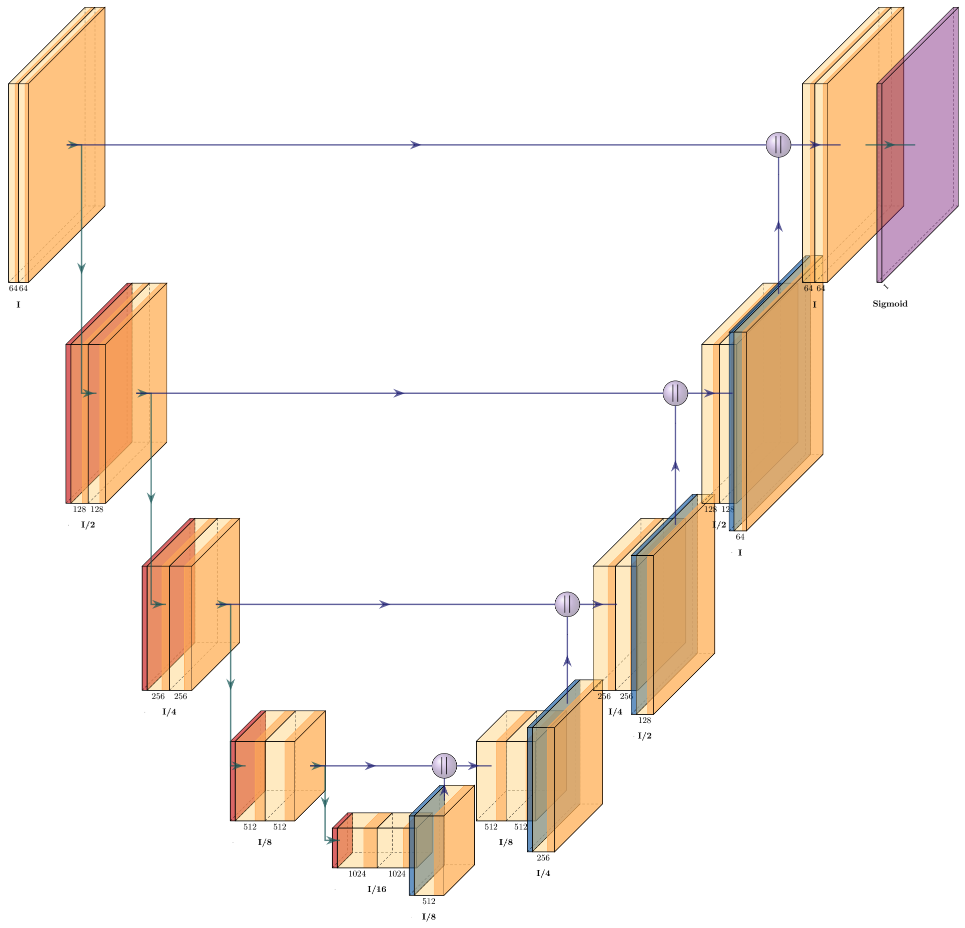


Figure 3.8: U-Net model architecture. The vertical axis denotes the resolution of the features, resulting in the U-shape of U-Net. Figure has been generated by modifying a `tikz` example provided in the MIT licenced `PlotNeuralNet` library available at this URL: `https://github.com/HarisIqbal88/PlotNeuralNet`.

3.3 Surface Normal Vector Prediction

The previous section described the deep learning approach for predicting *semantic* segmentation masks. In our specific case, these semantic segmentation masks denote which pixels contain roof structures, and which do not. Such semantic segmentation masks are not sufficient for achieving our stated goal of reconstructing three-dimensional roof surface geometries from remote sensing data. In order to achieve this goal we must produce *instance* segmentation masks instead, that is, not only determining if a given pixel contains a roof structure, but also determining which specific flat roof surface the pixel is part of.

In the upcoming Chapter 4 - “Post-processing”, we will present a procedure for partitioning semantic segmentation maps into instance segmentation maps by clustering normal vectors. Subsequent post-processing steps can be applied in order to convert these instance segmentation maps into three-dimensional vector polygons representing flat roof surfaces. This section will describe the prediction of these surface normal vector rasters which are used by these post-processing steps. Since this is a relatively novel approach to producing instance segmentation masks, there exists relatively little existing work in the academic literature which is directly applicable. We will, however, present some existing work which is tangentially relevant to our task at hand in Section 3.3.1. A U-Net derived CNN architecture for predicting rasterized surface normal vectors is presented in Section 3.3.2.

3.3.1 Related work

ESRI, in cooperation with Miami-Dade County and Nvidia, constructed a machine learning pipeline for the reconstruction of three-dimensional building models from aerial LiDAR data [51]. The project used human-annotated building outline polygons in conjunction with a roof type classification label assigned to each polygon; either *flat*, *gable*, *hip*, *shed*, *dome*, *vault*, or *mansard*. This was a costly procedure as the average human annotator only managed to annotate about 70 roof segments per hour. Another drawback is that the ensuing model is restricted to reconstructing polygons of one of the seven pre-defined classifications. The digital surface model (DSM) was normalized by subtracting the digital terrain model (DTM), producing a raster which indicates the height above ground level, a so-called *normalized digital surface model* (nDSM). These normalized heights were converted into an 8-bit unsigned integer raster channel, and two additional 8-bit channels where constructed from the normalized x- and y-gradient from Sobel–Feldman operator applied upon the nDSM. The resulting three-channel raster images were fed into a Mask R-CNN architecture with minor modifications, requiring training for several weeks on 18 200 training samples before reaching convergence. Proprietary ArcGis geoprocessing tools were finally applied in order to convert the classified instance masks into three-dimensional vector polygons.

Wang *et al.* presents a robust and efficient CNN architecture for predicting surface normals from RGB images in [52], but under the strong assumption of a

"Manhattan world", that is, each normal vector can only point in one of exactly three directions (the x-, y-, z-direction). This results in a three-class pixelwise classification model rather than a continuous normal vector regression model.

The inference of surface normal vectors is closely related to the field of *depth estimation*. Qi *et al.* proposes the **GeoNet** architecture in [53] for predicting joint depth and surface normal vector rasters, utilizing a least square solution of the surface normal using the predicted depths, and improves upon this normal prediction with a residual module.

Nesti-net presented in [54] in 2019 is able to predict surface normals from unstructured LiDAR *point cloud data*, rather than LiDAR rasters. A local point cloud representation using multi-scale point statistics (MuPS) is used in order to pre-process the raster data into a format suitable for CNN architectures.

3.3.2 CNN architecture for predicting surface normal vectors

In Equation (2.1) on Page 32 we defined the *normal vector* raster, which we will provide again here in a somewhat simplified form:

$$N_{i,j} = \begin{cases} \boldsymbol{n}_{i,j} = \left[n_{i,j,x},\ n_{i,j,y},\ n_{i,j,z}\right]^T, & \text{if polygon exists at } \pi_B(i,\ j), \\ \boldsymbol{0} = [0,\ 0,\ 0]^T, & \text{otherwise.} \end{cases}$$

We now intend to construct a model, which we will denote as $\widehat{f}_{\text{norm}}$, which targets this ground truth raster. Denote the parametrization of this model as $\boldsymbol{\theta}_{\text{seg}}$, such that for a given input raster X, the model produces a prediction $\widehat{N}$ according to $\widehat{N} := \widehat{f}_{\text{norm}}(X; \boldsymbol{\theta}_{\text{norm}})$. Additionally, assume that we have a semantic segmentation model, $\widehat{f}_{\text{seg}}$, parametrized according to $\boldsymbol{\theta}_{\text{seg}}$, which produces a semantic segmentation map over the same geographic area, $\widehat{S} := \widehat{f}_{\text{seg}}(X; \boldsymbol{\theta}_{\text{seg}})$.

We intend to construct $\widehat{f}_{\text{norm}}$ such that it is able to accurately predict $N_{i,j}$ whenever there exists a polygon at $\pi_B(i,\ j)$, but not necessarily otherwise. With other words, $\widehat{f}_{\text{norm}}$ will not be required to predict zero-vectors for pixel locations where no polygons are located. This will drastically reduce the task-complexity of $\widehat{f}_{\text{norm}}$ as it is no longer required to semantically segment the input raster *in addition to* predicting surface normal vectors. The simplification is made possible by ignoring the normal vector predictions $\widehat{N}_{i,j}$ whenever $\widehat{S}_{i,j} < \texttt{TOL}$, for some model confidence threshold `TOL` (we will use $\texttt{TOL} = 0.5$). This process is described in much more detail in Section 4.1.

Notice that the following must hold for any ground truth normal vector $\boldsymbol{n}_{i,j}$ by construction:

$$\begin{aligned} \left\|\boldsymbol{n}_{i,j}\right\|_2 = \sqrt{n_{i,j,x}^2 + n_{i,j,y}^2 + n_{i,j,z}^2} &\equiv 1, \\ \implies -1 \leq n_{i,j,d} \leq 1, &\qquad \text{for } d \in \{x, y, z\}. \end{aligned}$$

A three-channeled CNN layer which uses the tanh activation function will produce a raster belonging to the domain $[-1, 1]^{H \times W \times 3}$. An implementation of such a layer is provided in Code listing 3.1.

Code listing 3.1: Keras layer producing raster values of absolute magnitude less than or equal to 1. Python code using the Tensorflow v2.1 library. The variable `fanal_decoder` is the last decoder output of U-Net of size $256 \times 256 \times 64$.

```python
import tensorflow as tf
...
...
surface_vectors = tf.keras.layers.Conv2D(
    filters=3,
    kernel_size=(1, 1),
    activation=tf.keras.activations.tanh,
    name="surface_vectors"
)(final_decoder)
```

The three-channeled tanh output layer now produces a three-dimensional vector for each pixel (i, j), thus the model has $H \times W \times 3$ degrees of freedom in its output layer. We can now utilize the fact that $\left\|\boldsymbol{n}_{i,j}\right\|_2 \equiv 1$ in order to reduce the degrees of freedom of each predicted vector from three to two. This is performed by "forcing" the model output into the correct domain by applying a ℓ_2-normalization on each predicted vector. The ℓ_2-normalization procedure is formally defined as:

$$\text{L2Normalize}(\boldsymbol{x}) := \frac{1}{\|\boldsymbol{x}\|_2} \cdot \boldsymbol{x} = \frac{1}{\sqrt{x_1^2 + x_2^2 + x_3^2}} \cdot \boldsymbol{x}, \quad \boldsymbol{x} \in \mathbb{R}^3$$

The ensuing ℓ_2-normalized model is no longer required to produce vectors of correct *magnitude*, only vectors of correct *direction*. An implementation of L2Normalize in the form of a Keras modeling layer, and an example application of this Keras layer to the three-channeled output layer, is provided in Code listing 3.2.

Code listing 3.2: A Keras modeling layer implementing ℓ_2-normalization. Python code using the Tensorflow v2.1 library.

```python
class NormalVectorization(tf.keras.layers.Layer):
    def __init__(self, *args, **kwargs):
        kwargs["trainable"] = False
        kwargs["dynamic"] = False
        super(NormalVectorization, self).__init__(*args, **kwargs)

    def call(self, inputs: tf.Tensor):
        return tf.math.l2_normalize(
            x=inputs,
            axis=-1,
            epsilon=1e-12,
        )
...
...
normal_surface_vectors = NormalVectorization(
    name="surface_normal_vectors"
)(surface_vectors)
```

Now it remains to construct a loss function for this ℓ_2-normalized vector model $\widehat{f}_{\text{norm}}$ which does *not* penalize "bad" behaviour at pixel locations (i, j) where $N_{i,j} = \mathbf{0}$. Start by noticing that since $\|\widehat{N}_{i,j}\| \equiv 1$ and $\|N_{i,j}\|_2 \in \{0, 1\}$, we have

$$N_{i,j}^T\,\widehat{N}_{i,j} = \|N_{i,j}\|_2\,\|\widehat{N}_{i,j}\|_2\cos(\theta) = \begin{cases}\cos(\theta), & \text{if } \|N_{i,j}\|_2 = 1,\\ 0, & \text{if } \|N_{i,j}\|_2 = 0.\end{cases},$$

where $N_{i,j}^T\,\widehat{N}_{i,j}$ is the vector dot product between $N_{i,j}$ and $\widehat{N}_{i,j}$, and θ is the angle formed between these two normal vectors. We intend to minimize θ, and thus indirectly maximize $\cos(\theta)$, which inspires the *cosine similarity* loss function defined by

$$\mathcal{L}_{\cos}\left(\widehat{N}_{i,j}; N_{i,j}\right) = 1 - N_{i,j}^T\,\widehat{N}_{i,j} = \begin{cases}1 - \cos(\theta), & \text{if } \|N_{i,j}\|_2 = 1,\\ 1, & \text{if } \|N_{i,j}\|_2 = 0.\end{cases}.$$

Notice how the cosine similarity loss is constantly equal to 1 whenever $\|N_{i,j}\|_2 = 0$, thus having zero-derivative no matter the input. The behaviour of the normal vector model $\widehat{f}_{\text{norm}}$ is therefore completely ignored at such pixel locations during training, just as intended. When generalizing this loss function for an entire surface normal raster, we construct the *pixel-averaged cosine similarity* loss defined as

$$\begin{aligned}\mathcal{L}_{\text{norm}}\left(\widehat{N}; N\right) &= \frac{1}{HW}\sum_{i=1}^{H}\sum_{j=1}^{W} 1 - N_{i,j}^T\,\widehat{N}_{i,j}\\ &= 1 - \frac{1}{HW}\sum_{i=1}^{H}\sum_{j=1}^{W} N_{i,j,x}\widehat{N}_{i,j,x} + N_{i,j,y}\widehat{N}_{i,j,y} + N_{i,j,z}\widehat{N}_{i,j,z}\end{aligned} \tag{3.4}$$

Notice that for pixel locations where $N_{i,j} = \mathbf{0}$, each component in the predicted normal vector $\widehat{N}_{i,j}$ becomes multiplied by zero when taking the dot product $N_{i,j}^T\widehat{N}_{i,j}$. These terms are therefore completely disregarded by the loss function, and the model is allowed to predict completely garbage data for such pixel locations. How such garbage data is ignored by the loss function is demonstrated in Figure 3.9.

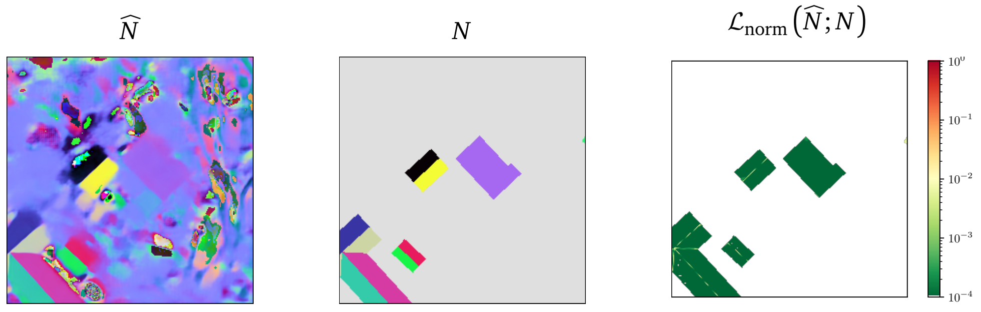


Figure 3.9: Demonstration of pixel-wise cosine similarity loss. The left most tile shows a normal vector raster prediction, $\widehat{N}$, while the ground truth is shown in the middle tile, N. The pixel wise cosine similarity, before being averaged, is illustrated in the right-most tile.

An implementation of this loss function written in python using Tensorflow v2.1 is provided in Code listing 3.3.

Code listing 3.3: Cosine similarity loss function implemented in Tensorflow v2.1.

```python
@tf.function
def cosine_similarity(
    y_true: tf.Tensor,
    y_pred: tf.Tensor,
):
    dot_products = tf.math.reduce_sum(y_true * y_pred, axis=-1)
    similarites = 1 - dot_products
    average_similarities = tf.math.reduce_mean(similarites, axis=(-1, -2))
    return tf.math.reduce_sum(average_similarities, axis=0)
```

To summarize, our model for targeting normal vector rasters is constructed by taking the well-known U-Net semantic segmentation CNN architecture and replacing the single-channeled sigmoid output layer with a ℓ_2-normalized three-channeled tanh output layer. This CNN architecture is trained by minimizing the average cosine similarity between the predicted normal vectors and the ground truth normal vectors, ignoring all pixels where no proper ground truth normal vectors are defined.

3.4 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 details.

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

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

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

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

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

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

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* [55]. 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 uniformly 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 [56].

A common modification to this scheme is the so-called *mini-batch gradient descent* algorithm [57]. 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* [58]. 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 [59] 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 Chapter 5.

3.4.1 Multitask learning

So far we have only discussed the construction and optimization of machine learning models with one single task at hand. Now we pose the question, if we construct a *single* model, $\widehat{f}_{\mathrm{A,B}}$, which is able to produce predictions for *two* different tasks A and B simultaneously, how should the model be trained? Assume that $\widehat{f}_{\mathrm{A,B}}$ is parametrized according to $\boldsymbol{\theta}_{\mathrm{A,B}}$, and that the model produces two distinct predictions $\widehat{Y}_A$ and $\widehat{Y}_B$ for any given input X, that is, $(\widehat{Y}_A,\ \widehat{Y}_B) := \widehat{f}_{\mathrm{A,B}}(X; \boldsymbol{\theta}_{\mathrm{A,B}})$. Finally, assume that the model targets the ground truths Y_A and Y_B. The task of solving two different tasks with a single model parametrization is an problem instance of *multitask learning* (MTL) [60].

You might think that any *multitask model* $\widehat{f}_{\mathrm{A,B}}$ will perform strictly worse than the combination of two independently trained single-task models $\widehat{f}_{\mathrm{A}}$ and $\widehat{f}_{\mathrm{B}}$, especially if the three models have the same parametric complexity: $\left\|\boldsymbol{\theta}_{\mathrm{A,B}}\right\| = \|\boldsymbol{\theta}_{\mathrm{A}}\| = \|\boldsymbol{\theta}_{\mathrm{B}}\|$. The multitask model $\widehat{f}_{\mathrm{A,B}}$ does after all solve two different tasks simultaneously, while the single-task models $\widehat{f}_{\mathrm{A}}$ and $\widehat{f}_{\mathrm{B}}$ can focus on just one single task at a time. In practice, however, multitask models are often observed outperforming respective sets of single-task models [60, Section 2]. The reason for this enhanced predictive power of multitask models is explained by Caruana:

> "MTL improves generalization by leveraging the domain-specific information contained in the training signals of *related* tasks. It does this by training tasks in parallel while using a shared representation. In effect, the training signals for the extra tasks serve as an inductive bias." [60, p. 41]

Assume that there exists well-defined loss functions for each task *independently*, $\mathcal{L}_A(\widehat{Y}_A; Y_A)$ and $\mathcal{L}_B(\widehat{Y}_B; Y_B)$. One way to perform multitask learning is to construct a linear combination of these two loss functions,

$$\mathcal{L}_{A,B}\left(\widehat{Y}_A, \widehat{Y}_B; Y_A, Y_B\right) = \alpha \cdot \mathcal{L}_A(\widehat{Y}_A; Y_A) + (1-\alpha) \cdot \mathcal{L}_B(\widehat{Y}_B; Y_B), \quad \alpha \in (0,1)$$

and then finding a suitable value for $\boldsymbol{\theta}_{\mathrm{A,B}}$ by optimizing $\mathcal{L}_{A,B}$ over Y_A and Y_B in conjunction. We will in fact use this approach in order to construct a multitask model for predicting semantic segmentation masks *and* surface normal vectors simultaneously. The results of this will be presented in Section 5.4.

3.5 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 [61]. How to normalize input data depends on distribution of the feature space, which will be investigated here.

3.5.1 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 3.10, and aggregate statistics for each channel are listed in Table 3.1.

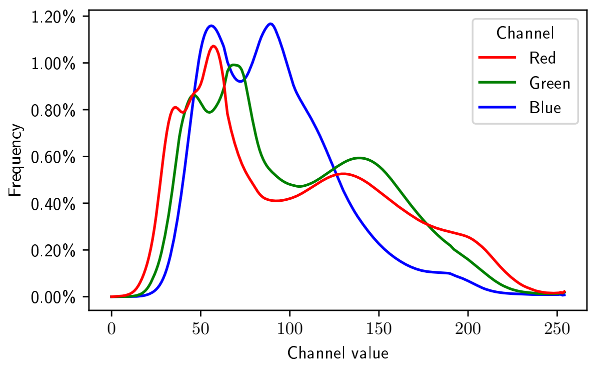


Figure 3.10: Distribution density for all three color channels in the aerial photography data set covering Trondheim municipality (2017).

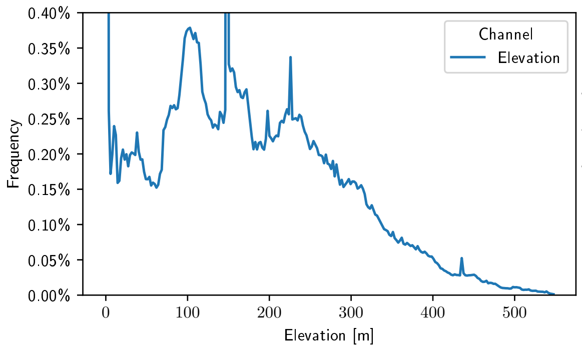
Channel	Mean [1]	SD [1]
Red	101.6	55.0
Green	102.7	48.7
Blue	91.3	37.2

Table 3.1: 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.

3.5.2 LiDAR rasters

The LiDAR raster E 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 $E_{i,j} \in (-3.4 \times 10^{38}\,\mathrm{m}, 3.4 \times 10^{8}\,\mathrm{m})$. 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 3.11, and aggregate statistics are listed in Table 3.2.



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

Figure 3.11: Distribution density for elevation data set covering Trondheim municipality (2017). Outlier values (0 m, 2.74 %) and (148 m, 1.93 %) have been cropped.

Table 3.2: Aggregate statistics for elevation data set covering municipality of Trondheim (2017).

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

$$\hat{E}^{(k)}_{i,j} = \frac{E^{(k)}_{i,j} - \min\limits_{t \in \mathrm{TRD}} E^{(t)}}{\max\limits_{t \in \mathrm{TRD}} E^{(t)} - \min\limits_{t \in \mathrm{TRD}} E^{(t)}} \qquad \text{(Global min-max normalization)}$$

$$= \frac{E^{(k)}_{i,j} + 9\,\mathrm{m}}{578\,\mathrm{m}},$$

where TRD is the index set of all tiles belonging to the Trondheim region. The normalized raster elevation values in $\hat{E}^{(k)}$ 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 LiDAR rasters across geographic tiles, we plot the "tile-by-tile" elevation statistics in Figure 3.12.

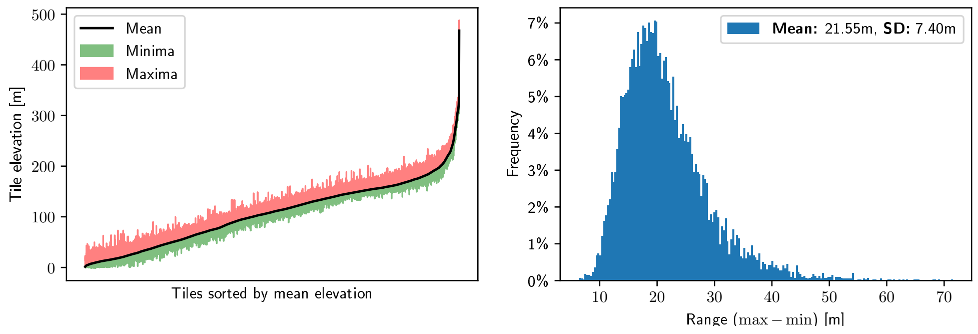


Figure 3.12: Elevation value statistics for a tile subset of sample size $n = 10\,000$. The left figure shows the minimum, mean, and maximum elevation, sorted by increasing mean from left to right. The right figure shows the histogram of the tile elevation *ranges* (difference between maximum elevation and minimum elevation within tile).

While the global elevation range is $569\,\mathrm{m} - (-9\,\mathrm{m}) = 578\,\mathrm{m}$, the elevation range within each respective tile is on average approximately $22\,\mathrm{m} \pm 8\,\mathrm{m}(\mathrm{SD})$, 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".

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

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 $\mathcal{V} = \{(i, j) : E_{i,j} \neq \texttt{nodata}\}$.
2 Calculate $\alpha = \max_{(i,j)\in\mathcal{V}} E_{i,j}$ and $\beta = \min_{(i,j)\in\mathcal{V}} E_{i,j}$.
3 Construct normalized raster defined by
      $$\hat{E}_{i,j} = \begin{cases} \frac{E_{i,j}-\beta}{\alpha-\beta}, & \text{if } (i,j) \in \mathcal{V}, \\ 0, & \text{otherwise.} \end{cases}$$
```

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:

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

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 $\mathcal{V} = \{(i,j) : E_{i,j} \neq \texttt{nodata}\}$.
2 Calculate $\beta = \min_{(i,j)\in\mathcal{V}} E_{i,j}$ and define a global scaler $\gamma > 0$.
3 Construct normalized raster defined by
      $\hat{E}_{i,j,z} = \begin{cases} \frac{E_{i,j} - \beta}{\gamma}, & \text{if } (i,j) \in \mathcal{V}, \\ 0, & \text{otherwise.} \end{cases}$
```

Chapter 4

Post-processing

Assume that we have two machine learning models; firstly $\widehat{f}_{\text{seg}}$ which is able to predict *semantic* segmentation masks $\widehat{S}$, and secondly $\widehat{f}_{\text{norm}}$ which is able to predict surface normal rasters $\widehat{N}$. These two models could conceivably be sub-models of a *single* multi-task model, both accepting remote sensing data in the form of aerial photography and/or LiDAR measurements as illustrated in Figure 4.1.

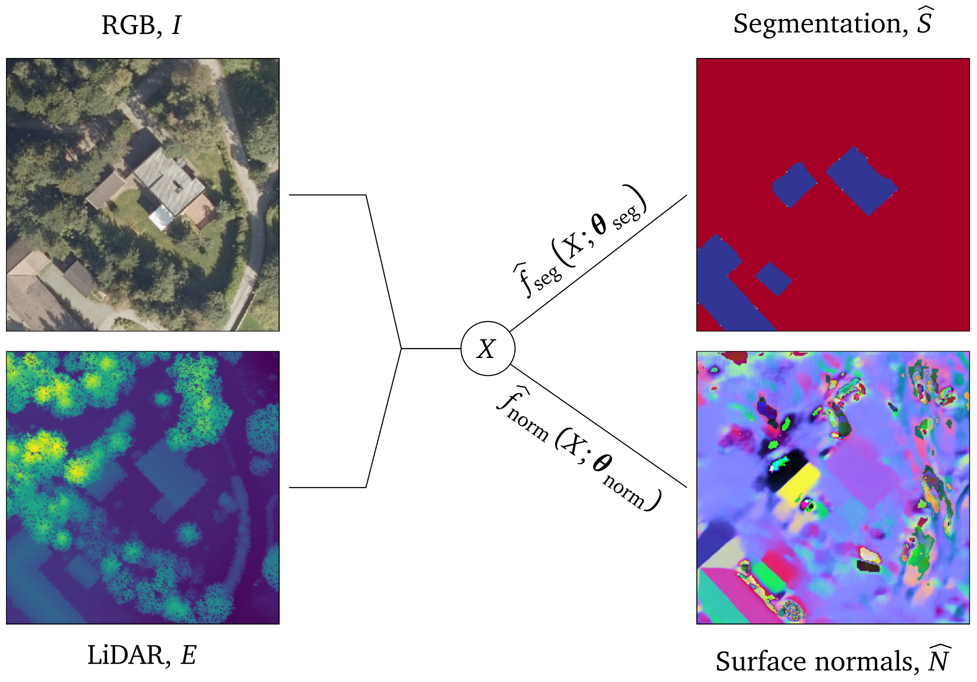

RGB, I
Segmentation, $\widehat{S}$
$\widehat{f}_{\text{seg}}(X; \boldsymbol{\theta}_{\text{seg}})$
X
$\widehat{f}_{\text{norm}}(X; \boldsymbol{\theta}_{\text{norm}})$
LiDAR, E
Surface normals, $\widehat{N}$

Figure 4.1: Demonstration of unprocessed output from a surface raster machine learning pipeline. The surface normal vector output, $\widehat{N}$, is scaled and translated into the value domain $[0, 255]^{H \times W}$ and visualized as an RGB image.

As discussed in Section 2.6 – "Surface Rasterization Algorithm", the goal is to convert these two model predictions belonging to the surface *raster* domain R back to the *vector* polygon domain V by constructing a suitable pseudoinverse mapping $m^\dagger : R \to V$ as illustrated in Figure 4.2.

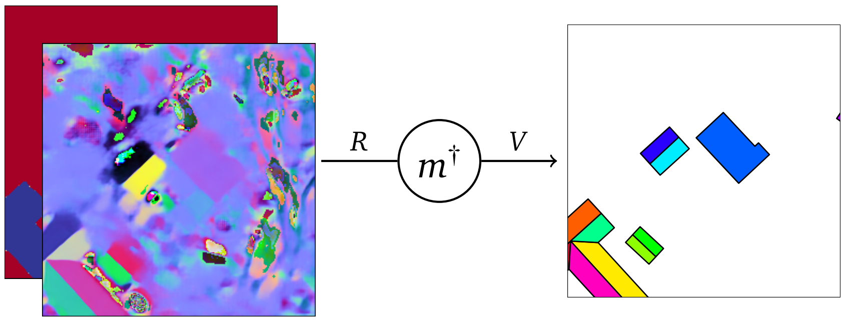


Figure 4.2: Pseudoinverse mapping, $m^\dagger$, mapping from the raster domain to the vector domain.

More specifically, we will construct a pseudoinverse mapping which accepts two inputs, a predicted surface normal raster and a semantic segmentation map, and produces three-dimensional vector polygons, which we will denote as $\widehat{\mathcal{P}}$,

$$\widehat{\mathcal{P}} := m^\dagger\left(\widehat{S}, \widehat{N}\right).$$

This pseudoinverse mapping should produce three-dimensional polygons which can be considered close to the ground truth polygons $\mathcal{P}$, where the notion of "closeness" between $\widehat{\mathcal{P}}$ and $\mathcal{P}$ is expressed in a *polygon distance metric* $d : V \times V \to \mathbb{R}$. The main objective of the pseudoinverse mapping is therefore to minimize

$$d\left(\widehat{\mathcal{P}}, \mathcal{P}\right) = d\left(m^\dagger\left(\widehat{S}, \widehat{N}\right), \mathcal{P}\right).$$

The model predictions, $\widehat{S}$ and $\widehat{N}$, are not perfect, and the pseudoinverse should therefore be *robust* relative to the types of errors commonly produced by the models, $\widehat{f}_{\text{seg}}$ and $\widehat{f}_{\text{norm}}$, producing minimal error as defined by the distance metric d. Constructing a suitable pseudoinverse is therefore not only highly dependent on the surface raster format itself, but also the specific behaviour of the models used to produce the raster format. In the evaluation of a pseudoinverse mapping it may still be beneficial to take the models themselves out of the equation by replacing the predicted rasters with the ground truth rasters $m(\mathcal{P})$, thus verifying that the pseudoinverse mapping also minimizes

$$d\left(m^\dagger\left(m\left(\mathcal{P}\right)\right), \mathcal{P}\right).$$

A pseudoinverse mapping which performs badly under this "round-trip metric" will likely perform badly under the pseudoinverse mapping of model predictions as well.

In the following sections we will present a pseudoinverse mapping which is considered highly suitable for producing roof surface polygons, a mapping which encodes our specific domain knowledge about the general geometric properties of roof surfaces. Our main idea is to use the predicted surface normal raster in order to *partition* the *semantic* segmentation map into an *instance* segmentation map, where the instance segmentation map can subsequently be used in order to construct enclosing vector polygons for each instance. The three-dimensional orientation of each polygon can finally be inferred from the predicted surface normals in conjunction with the original LiDAR data.

Our implementation applies a divide-and-conquer approach to the problem by dividing the semantic segmentation map into mutually disjunct sub-regions and partitioning these sub-regions entirely independent of each other. This first-pass partitioning is described in Section 4.1 – "Connected region labeling". A second-pass partitioning is performed by grouping together connected regions which share the same normal vector orientation. The task of determining which pixels (i, j) which share the same values for $N_{i,j}$ is formulated as a *clustering problem*, the solution of which will be presented in Section 4.2. A final conversion of instance segmentation maps to vector polygons will be described in Section 4.3.

Before delving into the details of the implementation, we begin by summarizing the entire post-processing algorithm in conceptual terms:

Given a predicted segmentation map $\widehat{S}$ and surface normal raster $\widehat{N}$:

- Threshold segmentation activations $\widehat{S}$ according to some tolerance `TOL` in order to construct *binary* segmentation raster $\widetilde{S}$.
- Identify connected sub-regions of the binary segmentation raster $\widetilde{S}$, and sub-divide the following processing independently across these sub-regions.
- Apply a conservative clustering algorithm which does *not* require the specification of the number of clusters *a priori*.
- Apply a second clustering algorithm which encodes suitable domain knowledge in order to classify the remaining points, producing a labeled partition of the original binary segmentation raster.
- Construct two-dimensional vector polygons which enclose each instance region.
- Simplify these vector polygons.
- Reconstruct z-coordinates of each polygon vertex.

4.1 Connected region labeling

The predicted segmentation map $\widehat{S}$ produced by $\widehat{f}_{\text{seg}}$ is a $H \times W \times 1$ raster array where the values $0 \leq \widehat{S}_{i,j} \leq 1$ can be interpreted as the model's confidence in there being a roof surface at pixel location (i, j). In order to make an actual binary prediction for each pixel, we *threshold* the segmentation map, creating a binary segmentation map $\widetilde{S}$ defined by

$$\widetilde{S} := \widehat{S} > \texttt{TOL} \iff \widetilde{S}_{i,j} = \begin{cases} 1, & \text{if } \widehat{S}_{i,j} > \texttt{TOL}. \\ 0, & \text{otherwise.} \end{cases}$$

We will use the most commonly used threshold value, namely $\texttt{TOL} = 0.5$. The process of thresholding has been illustrated in Figure 4.3.

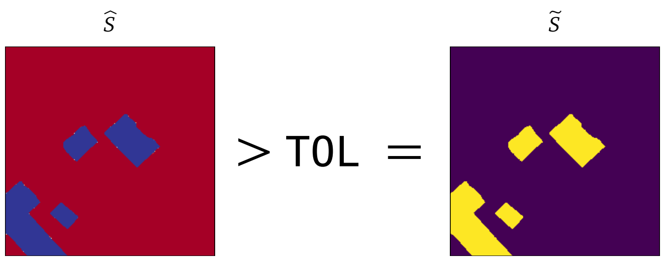


Figure 4.3: Thresholding a probabilistic segmentation prediction in order to create a *binary* segmentation map. Here $\texttt{TOL} = 0.5$ has been used.

This binary segmentation map can now be used in order to segment the predicted surface normal raster as well, producing what we will refer to as *segmented surface normals*. The segmented surface normal raster, $\widetilde{N}$, is constructed by taking the element-wise product of the predicted surface normal raster and the thresholded binary segmentation map, which we will denote as $\widetilde{S} \odot \widehat{N}$ and formally define as

$$\widetilde{N} := \widetilde{S} \odot \widehat{N} \iff \widetilde{N}_{i,j,d} = \widetilde{S}_{i,j} \cdot \widehat{N}_{i,j,d}, \text{ for } d \in \{x, y, z\}.$$

The segmentation of the predicted surface normal raster is illustrated in Figure 4.4.

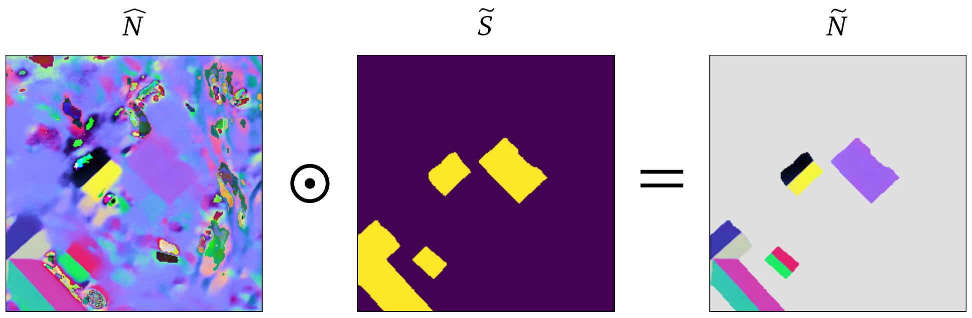


Figure 4.4: Segmentation of a predicted surface normal raster using a binary segmentation map.

These segmented normals $\widetilde{N}$ are intended to be used in order to partition the semantic segmentation map $\widetilde{S}$ into an instance segmentation map, where each instance is considered to be a two-dimensionally projected roof surface polygon. A simple first-pass partitioning of the segmentation map into regions which are completely connected can be performed, with other words the partitioning into entire roofs rather than single roof surfaces. The identification of connected sub-regions can be performed by a so-called *connected region labeling* algorithm as described in [62, 63][1].

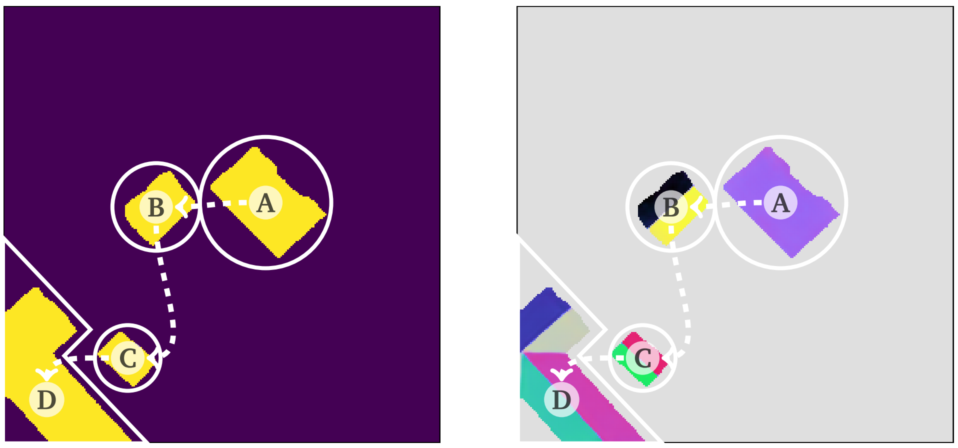


Figure 4.5: The result of `skimage.measure.label` applied on a binary segmentation map. Four separate sub-regions, marked as A, B, C, and D, have been identified.

A dynamic programming programming approach can be subsequently applied in order to independently subdivide these *roof partitions* into *surface partitions*. The example presented in Figure 4.5 shows that there exists four separate connected regions (A, B, C, and D) in the predicted semantic segmentation raster. Provided that the predicted segmentation map is sufficiently correct, we can conclude that any polygon belonging to any one of these regions will not belong to any other region. We can therefore formulate the partitioning of each region as entirely independent tasks, the solution of which will be presented in the next section.

[1]A connected region labeling algorithm is implemented by the `skimage` python package in `skimage.measure.label`

4.2 Instance Clustering

We now intend to subdivide each roof partition into constituent surface partitions, creating an instance segmentation map as a result. Each partition should be constructed such that all pixels (i, j) in a given surface partition share the same normal vector orientation, $N_{i,j}$. A complicating factor is that the predicted surface normal raster $\widetilde{N}$ is not perfect, that is, the predicted normal vectors $\widetilde{N}_{i,j}$ are spatially distributed across some neighbourhood around the ground truth surface normal vector $N_{i,j}$. The distribution of the cosine similarity between the predicted normal vectors and the ground truth normal vectors, $1 - \widetilde{N}_{i,j} \cdot N_{i,j}$, for one of the trained surface normal model $\widehat{f}_{\text{norm}}$ is shown in Figure 4.6.

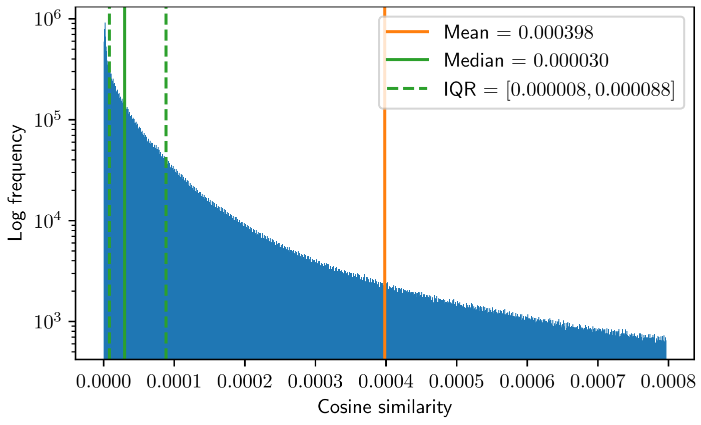


Figure 4.6: The distribution of the cosine similarities between the ground truth normal vectors and the predicted normal vectors of a surface normal model over 9537 test tiles.

Denote the index set of a given roof partition r as $\mathcal{I}_r = \{(i_{r,1}, j_{r,1}), (i_{r,2}, j_{r,2}), \dots\}$. We will specifically focus on roof partition $r = D$ as labeled in Figure 4.5 in the following example plots. Start by constructing a set of surface normal vector observations defined over the given roof partition, $\mathcal{N}_r := \{\widetilde{N}_{i,j} \mid (i, j) \in \mathcal{I}_r\}$. The spatial distribution of $\mathcal{N}_D$ is shown in Figure 4.7, each normal vector being represented as a scatter point on the projected xy-plane. Now, the region defined by roof partition D contains four ground truth roof surface polygons, and each scatter point $\widetilde{N}_{i,j}$ has been color annotated according to which polygon that overlaps at pixel location (i, j). Each ground truth roof surface polygon has an associated ground truth normal vector as well, and these are annotated as black crosses

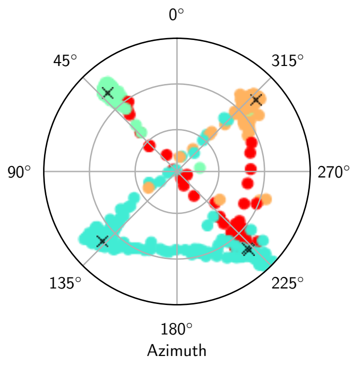


Figure 4.7: Ground truth labeling of predicted normal vectors. Azimuth = 0° indicates true north.

($\times$) in Figure 4.7. The task is now to assign a cluster label $l \in \{1, 2, \dots, C\}$ to each observation in $\mathcal{N}_D$ such that those observations which share the same label also can be considered to share the same normal vector. In our example, where the ground truth is available, we know that the number of clusters C should ideally be equal to four, i.e. the number of unique roof surfaces. In practice, however, it is important to notice that the number of roof surfaces in a given roof partition is not known *a priori*, and C therefore needs to be inferred from the spatial distribution of $\mathcal{N}_D$. It is therefore important that the clustering algorithm we chose does not require C to be specified.

Our clustering procedure utilizes two different clustering algorithms applied in succession, first *density-based spatial clustering of applications with noise* (DBSCAN) and then *k-nearest neighbours* (k-NN). DBSCAN has the benefit of being able to infer C from $\mathcal{N}_D$, a value which can then be passed forwards to the k-NN clustering algorithm which requires it *a priori*. Additionally, DBSCAN has the ability to label observations as "noise" whenever the cluster they belong to is not entirely clear. This allows us to use the conservative non-noisy labels produced by DBSCAN to estimate the ground truth normal vectors, while applying k-NN (which encodes a greater degree of our domain knowledge about roof surfaces) on the remaining noisy observations. The application of these two clustering algorithms will now be explained in more detail.

4.2.1 DBSCAN

DBSCAN is a non-parametric clustering algorithm (that is C does not need to be specified) first proposed by Ester *et al.* in 1996[64]. Consider the normal vector observations $\widetilde{N}_{i,j} \in \mathcal{N}_r$ to be clustered, and define some distance metric between two normal vectors N_1 and N_2, $d(N_1, N_2)$. The distance metric could be chosen to be the cosine distance between two normalized vectors of length 1: $d(N_1, N_2) = 1 - N_1 \cdot N_2$[2]. Given parameters $\varepsilon > 0$ and `MinPts` $\in \mathbb{Z}^+$, DBSCAN annotates each normal vector in $\widetilde{N} \in \mathcal{N}_r$ as either a *core point*, *reachable point*, or as a noisy *outlier*. A normal vector $\widetilde{N} \in \mathcal{N}_r$ is considered to be a core point if there exists at least `MinPts` normal vectors $\widetilde{N}^* \in (\mathcal{N}_r \setminus \widetilde{N})$ where $d(\widetilde{N}, \widetilde{N}^*) < \varepsilon$. DBSCAN then creates a graph of all identified core points and draws edges between core point nodes $\widetilde{N}_1$ and $\widetilde{N}_2$ whenever $d(\widetilde{N}_1, \widetilde{N}_2) < \varepsilon$. All *connected components* of this core point neighbour-

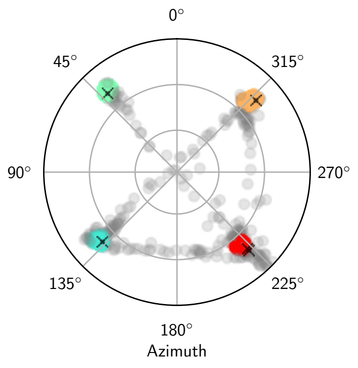


Figure 4.8: DBSCAN labeling of predicted normal vectors. $\varepsilon = 2.5 \times 10^{-6}$, `MinPts` $= 25$, $d =$ cosine distance.

[2]Using cosine distance as the distance metric during clustering is especially suitable when $\widehat{f}_{\text{norm}}$ is trained by using cosine distance as the loss function. The same can be said of other cluster distance metric / model loss combinations.

hood graph are considered to be clusters, thus automatically inferring the total number of clusters C. Any non-core normal vector is assigned to a nearby cluster as long as it is within ε distance of a normal vector within that cluster. All remaining normal vectors are finally considered noisy outliers.

Figure 4.8 on the previous page shows the result of DBSCAN when applied on the predicted surface normal vectors $\mathcal{N}_D$ belonging to roof partition D as annotated in Figure 4.5 (Page 73). The parametrization of DBSCAN is set as $\varepsilon = 2.5 \times 10^{-6}$ and `MinPts` $= 25$, where the cosine distance metric has been used. Of the non-noisy labels produced by DBSCAN in Figure 4.8, 99.8 % are labeled correctly. The remaining noisy normal vectors are shown as gray scatter points.

Figure 4.9 shows the result of applying DBSCAN on all four roof partitions (A, B, C, and D as presented in Figure 4.5). The normal vectors labeled as noise by DBSCAN, marked as black pixels, are frequently situated along the border between roof surfaces. Protruding roof segments have also been observed as a common location for noisy normal vectors.

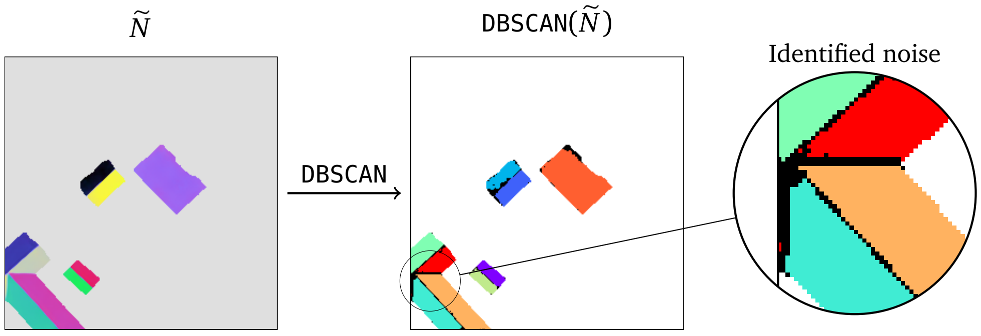


Figure 4.9: Application of DBSCAN on predicted surface normal raster. Normal vectors considered noise by DBSCAN have been colored in black. DBSCAN parameters: $\varepsilon = 2.5 \times 10^{-6}$, `MinPts` $= 25$, $d =$ cosine distance.

The labeling of the remaining noisy normal vectors will now be discussed.

4.2.2 k-nearest neighbour noise classification

Denote the index set of the DBSCAN-identified noise for roof partition r as $\mathcal{I}_r^*$. The set of noisy normal vectors $\mathcal{N}_r^* = \{\widetilde{N}_{i,j} \mid (i,j) \in \mathcal{I}_r^*\}$ is now considered insufficient for further clustering, as they are likely insufficiently close to their ground truth normal vectors in addition to being spuriously distributed in the normal vector feature space. Until now, the geographic location $[x, y]^T = \pi_B(i,\ j)$ corresponding to each predicted normal vector $\widetilde{N}_{i,j}$ has been entirely ignored. The idea is to use this spatial information in order to assign labels to the remaining

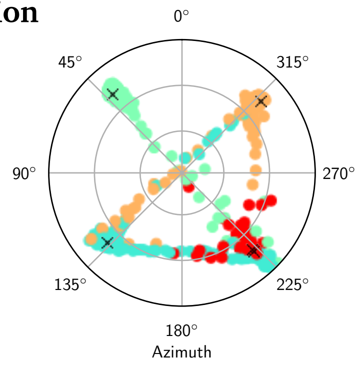


Figure 4.10: k-NN labeling of remaining DBSCAN noise. $k = 1$ with ℓ^2 distance norm.

noise, while utilizing the clustering already performed by DBSCAN. The *k-nearest neighbours* (k-NN) algorithm [65] is a conceptually simple algorithm highly suitable for this task. Start by constructing two geographic location sets, first one for the noisy coordinates $\mathcal{R}_b^* := \{\pi_B(i,\ j) \mid (i,j) \in \mathcal{I}_r^*\}$, and a second one for the successfully labeled coordinates $\mathcal{R}_b := \{\pi_B(i,\ j) \mid (i,j) \in (\mathcal{I}_r \setminus \mathcal{I}_r^*)\}$. For each geographic point marked as noise by DBSCAN, $p^* \in \mathcal{R}_b^*$, find the k nearest neighbours $\{p_1, p_2, \dots, p_k\}$ in the labeled set $\mathcal{R}_b$ according to some distance metric. The label assigned to p^* is then determined by a plurality vote of the labels of the neighbours $\{p_1, p_2, \dots, p_k\}$. We will specifically set $k = 1$ and use the Euclidean norm (ℓ^2-norm), that is, assigning the label of the geographically closest DBSCAN label, the result of which is presented in Figure 4.10. Of the noisy normal vectors labeled by k-NN in Figure 4.10, 76.9 % end up being labeled correctly. This yields a final clustering accuracy of roof partition D of 97.9 %. We should note that Figure 4.10 does not plot the features actually used by the k-NN clustering method, as it shows $\widetilde{N}_{i,j}$ and not $\pi_B(i,\ j)$. The figure is intended to be compared against Figure 4.8. The geographic features of the k-NN processed noise labels are shown in Figure 4.11.

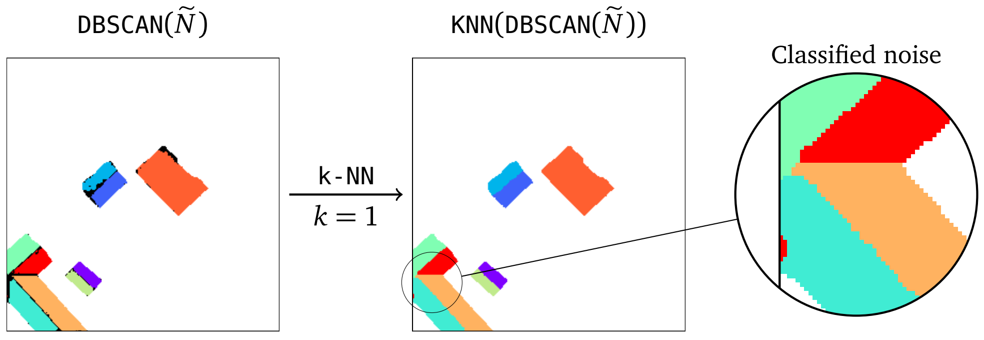


Figure 4.11: Application of k-NN on DBSCAN-identified noise, with $k = 1$ and using the ℓ^2-norm.

As shown in Figure 4.11 the k-NN clustering produces relative straight decision boundaries, which is important for roof surfaces. The result is a rasterized instance segmentation map which must be processed further in order to convert each instance to a vectorized polygon. This is the topic of the upcoming section.

4.3 Vectorization

The clustering described in the previous section results in a rasterized instance segmentation map which we will denote as $\widehat{L}$. Denote the total number of identified clusters across all roof partitions as C such that $\widehat{L} \in \{0, 1, 2, \ldots, C\}^{H \times W \times 1}$. Here $\widehat{L}_{i,j} = 0$ indicates that the pixel location (i, j) does not contain any surface polygon, i.e. $\widehat{S}_{i,j} = 0$. This section will describe how to convert this two-dimensional instance segmentation raster map into a set of vectorized three-dimensional polygons $\widehat{\mathcal{P}}$.

4.3.1 Two-dimensional polygonization

Connected region labeling can be used in order to group together all neighbouring pixels with equal values for $\widehat{L}_{i,j}$ as previously described in Section 4.1. Each connected sub-region can be converted to polygons by simply drawing line segments along the pixel borders, resulting in a fine-grained polygon for each instance[3]. Denote the resulting set of two-dimensional polygons as $\widehat{\mathcal{P}}$, an example of which is shown in Figure 4.12.

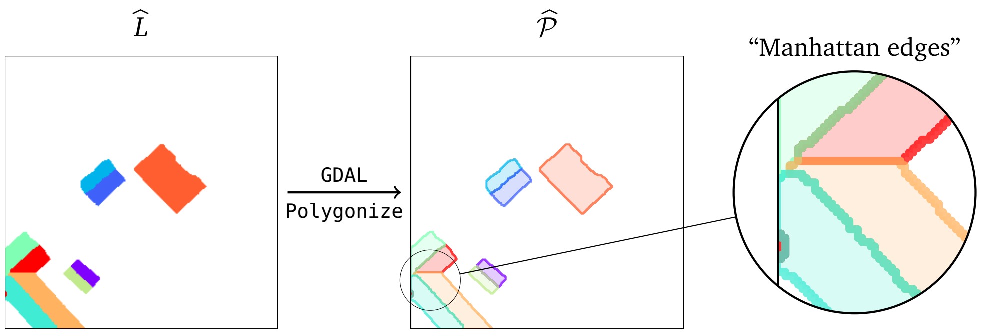


Figure 4.12: The result of applying `GDALPolygonize` on the clustered instance map $\widehat{L}$.

As can be seen in Figure 4.12, the resulting polygons show a high degree of "rasterization", having edges which are exclusively oriented along the east-west or north-south axis ("Manhattan edges" if you will). Depending on the application of the predicted polygons, it may be preferable to smooth these edges in order to represent diagonal edges more accurately. The following section will describe such a simplification.

[3]This functionality is implemented in `GDALPolygonize` in the GDAL C-library. The `rasterio` python package wraps this GDAL function in `rasterio.features.shapes`.

4.3.2 Simplification

A vectorized polygon is, as previously described, simply a collection of line segments which start and end at the same point (a so-called linear ring). The simplification of vectorized line segments is a well researched area with many proposed algorithms. The most common approach is to select a subsequence of the original linear ring while minimizing some loss criterion. Methods for preserving areas [66], distances [67], and angles [68] have been proposed. Visvalingam and Whyatt proposes an area-based progressive algorithm in [69][4]. Another common simplification guideline is the *bandwidth criterion*, stating that a simplified line is acceptable as long as it is within an ε-neighbourhood of the original line segment. The task is then to select the minimum number of vertices which still satisfies this criterion. The *Ramer-Douglas-Peucker* (RDP) algorithm[70] is one such method, a recursive algorithm which is implemented along the following lines:

```
def RDP($[p_1, p_2, \dots, p_n];\ \varepsilon$):
    Find $p_\perp \in [p_1, \dots, p_n]$ furthest away from the line between $p_1$ and $p_n$.
    If the distance from $p_\perp$ to the line is less than $\varepsilon$, return $[p_1, p_n]$.
    Otherwise, merge result of RDP($[p_1, \dots, p_\perp]$) and RDP($[p_\perp, \dots, p_n]$).
```

An illustration of the RDP algorithm is provided in Figure 4.14 on Page 80. We will use a slightly modified version of RDP implemented in the GEOS library which preserves certain topological properties of the simplified polygons[5], specifically the number of exterior and interior rings. The result of applying RDP on the vectorized polygons $\widehat{\mathcal{P}}$ with $\varepsilon = 0.75\,\text{m}$, producing simplified polygons $\widetilde{\mathcal{P}}$, is shown in Figure 4.13.

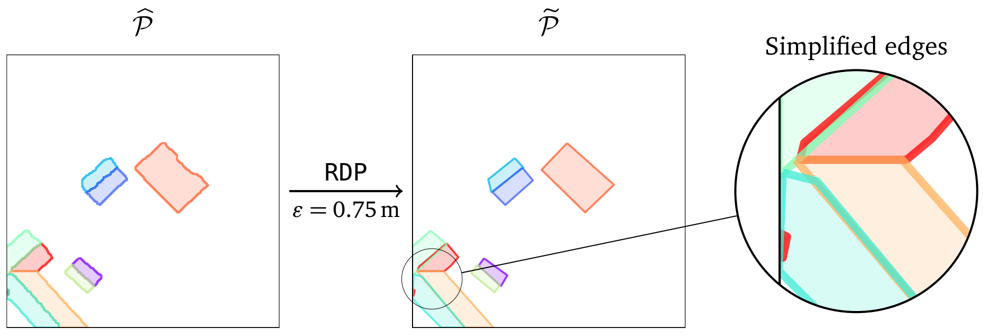


Figure 4.13: Polygons before and after RDP has been applied with bandwidth tolerance $\varepsilon = 0.75\,\text{m}$.

[4]The Visvalingam-Whyatt polygon simplification method has been implemented and newly released on the Python Package Index (PyPI) under the name `visvalingamwyatt`.

[5]Documentation for `TopologyPreservingSimplifier` available at: `https://geos.osgeo.org/doxygen/classgeos_1_1simplify_1_1TopologyPreservingSimplifier.html`.
A python wrapper is available in the `simplify()` function implemented in the `shapely` library when parametrized with “`preserve_topology=True`”: `https://shapely.readthedocs.io/en/latest/manual.html#object.simplify`.

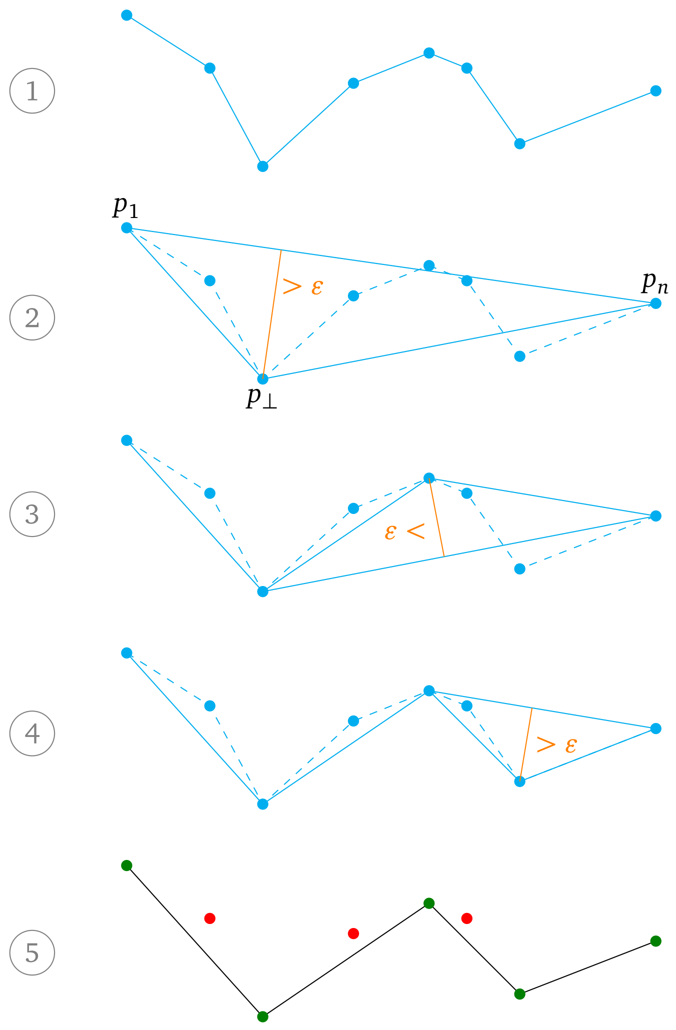


Figure 4.14: Illustration of the Ramer-Douglas-Peucker algorithm. The perpendicular distances between the points $p_\perp$ and the lines formed by the endpoints are shown in orange. Row 1 shows the original line, while rows 2 to 4 show the vertices determined to be included since they are further away than ε. Row 5 shows the final resulting line, where the green points indicate the vertices to be kept, while the red points indicate the discarded vertices. This specific example has been recreated from Figure 5 in [71].

4.3.3 Three-dimensional reconstruction

As discussed in Section 2.6.2, any planar three-dimensional polygon P can be decomposed into, and reconstructed from, two sub-components

1. the two-dimensional projection $\pi_{2\mathrm{D}}(P)$, …
2. and the parametric equation of the plane $\boldsymbol{\beta}(P) = [\beta_0, \beta_x, \beta_y]$.

The first sub-component has now been approximated by the simplified polygons $\widetilde{\mathcal{P}}$. We must now try to approximate $\boldsymbol{\beta}(P)$ before we can reconstruct the three-dimensional polygons. Start by noticing that β_x and β_y can be expressed in terms of the normal vector of the plane, $\boldsymbol{n} = [n_x, n_y, n_z]^T$, by construction

$$\begin{bmatrix} n_x \\ n_y \\ n_z \end{bmatrix} := \frac{1}{\sqrt{\beta_x^2 + \beta_y^2 + 1}} \begin{bmatrix} -\beta_x \\ -\beta_y \\ 1 \end{bmatrix} \iff \begin{bmatrix} \beta_x \\ \beta_y \end{bmatrix} = -\frac{1}{n_z} \begin{bmatrix} n_x \\ n_y \end{bmatrix}.$$

With other words, if we are able to construct a good estimate for the normal vector of the plane $\boldsymbol{n}(\boldsymbol{\beta}(P))$, we can also estimate β_x and β_y. Now, each polygon $P \in \widetilde{\mathcal{P}}$ has an associated cluster label value l in the instance segmentation map $\widehat{L}$. The corresponding raster index for the given polygon P is then defined as $\mathcal{I}_P := \{(i,j) \mid \widehat{L}_{i,j} = l\}$, and the predicted normal vectors associated to the polygon P can therefore be retrieved from the predicted surface normal vector raster as $\mathcal{N}_P := \{\widetilde{N}_{i,j} \mid (i,j) \in \mathcal{I}_P\}$. We can now construct a predictor for $\boldsymbol{n}(\boldsymbol{\beta}(P))$ from the associated normal vector set $\mathcal{N}_P$, for example the *normalized vector average*, defined as

$$\overline{\boldsymbol{n}} = \begin{bmatrix} \overline{n}_x \\ \overline{n}_y \\ \overline{n}_z \end{bmatrix} := \frac{\sum_{\widetilde{N}_{i,j} \in \mathcal{N}_P} \widetilde{N}_{i,j}}{\left\| \sum_{\widetilde{N}_{i,j} \in \mathcal{N}_P} \widetilde{N}_{i,j} \right\|_2}.$$

It is also possible to remove the DBSCAN-identified noise from the normal vector set in order to construct an even more robust estimator, that is, if the polygon P belongs to roof partition r, we use $\mathcal{N}_P^* := \{\widetilde{N}_{i,j} \mid (i,j) \in (\mathcal{I}_P \setminus \mathcal{I}_r^*)\}$ instead. We can now construct estimators for β_x, $\hat{\beta}_x := -\overline{n}_x / \overline{n}_z$, and β_y, $\hat{\beta}_y := -\overline{n}_y / \overline{n}_z$. The remaining task is now to construct an estimator for the final planar parameter β_0. Since the surface normal raster is entirely independent of the elevation of the plane, we must use the LiDAR input data in order to construct an estimator $\hat{\beta}_0$ for β_0. The idea is to formulate the search for $\hat{\beta}_0$ such that the *LiDAR residuals* (see Section 2.6.5) are minimized. Given a distance metric d, we can construct the predictor as

$$\hat{\beta}_0 = \underset{\beta_0 \in \mathbb{R}}{\operatorname{argmin}} \sum_{(i,j) \in \mathcal{I}_P} d\left(E_{i,j},\ \beta_0 + \hat{\beta}_x x + \hat{\beta}_y y\right),$$

where $[x, y]^T := \pi_B(i,\ j)$ and $E_{i,j}$ is the LiDAR measurement at pixel location (i,j). If we specify the square distance metric $d(x,y) = (x-y)^2$, then we have an ordinary least squares regression with just one regression parameter. Due to the nature of LiDAR measurement errors, the absolute distance $d(x,y) = |x - y|$ has been shown to produce more robust estimators for β_0.

We can now reconstruct the predicted set of simplified, three-dimensional polygons:

$$P_{3\text{D}} = \left[(x, y, \hat{\beta}_0 + \hat{\beta}_x x + \hat{\beta}_y y) \mid \text{ for vertex } (x,y) \text{ representing } P \right], \text{ for } P \in \widetilde{\mathcal{P}}.$$

Chapter 5

Experiments

This section will investigate the inference of flat roof surfaces using the raster data produced by the pipeline outlined in Chapter 2. The U-Net model architecture presented in Section 3.2.5 is used for semantic segmentation, while the slightly modified U-Net architecture presented in Section 3.3.2 will be used for predicting surface normal vectors. We start by describing the general experimental setup in Section 5.1. Section 5.2 provides a summary of all the relevant experimental results of my specialization project [5] related to semantic segmentation of roof structures. A comparative investigation into the suitability of the different raster data types (aerial photography and/or LiDAR DSMs) for surface normal prediction is presented in Section 5.3. Lastly, a multitask learning model for predicting semantic segmentation masks *and* surface normal vectors simultaneously is tested in Section 5.4.

5.1 Experimental Setup

5.1.1 Training procedure

The Trondheim dataset produces 64 146geographic tiles after being processed, each tile including aerial photography (RGB) data, elevation data (LiDAR elevation), ground truth semantic roof segmentation masks and surface normal vector rasters. This sample space is split into a customary 70% / 15% / 15% training–validation–testing split. The training data is randomly shuffled and subsequently grouped into batches of size 16 before applying the Adam optimizer. Training is continued until observed convergence by the use of the loss evaluated over the validation split. The weights corresponding to the epoch yielding the best validation loss is used as the final model parametrization.

Training summary

- 58 559 geographic tiles: $256\,\text{px} \times 256\,\text{px} = 64\,\text{m} \times 64\,\text{m}$.
- 44 939/ 9670/ 9537 training / validation / test.
- Random shuffling.
- Adam optimizer.
- Validation loss early stopping.

5.1.2 Software

The majority of the source code written in order to produce and present the results in this paper 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) [72] has been extensively used in order to process GIS data, and the python wrappers for GDAL, Rasterio [73] for raster data and Fiona [74] for vector data, are central building blocks of the data processing pipeline. The machine learning framework of choice has been the new 2.1 release of TensorFlow [75], most of the modelling code having been written with the declarative Keras API.

5.1.3 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.
- **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
  $\implies \approx$ 18.3 h per experiment.
- 8 ms per prediction (batch 1)
  $\implies$ 125 predictions per s.

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 2809 training batches of size 16, validating the model on 605 additional validation batches, and executing various Keras callbacks, each epoch lasts for 11 minutes from end to end. Most experiments have been trained for 100 epochs, hence requiring altogether 18 hours and 20 minutes of training. The final models are able to produce 125 predictions of size 256 px × 256 px per second.

5.2 Semantic Segmentation

The research questions posed in this thesis is a strict superset of the problems solved in my previous specialization project [5], a project which solely focuses on the semantic segmentation of roof structures from remote sensing data. We will therefore provide a brief summary of the conclusions made in [5] before continuing onto the experiments which are new to this thesis. We refer to the original paper[1] for a more detailed presentation and analysis of these results.

- LiDAR raster data is far more suitable for the prediction of semantic roof segmentation masks than aerial RGB photography.
- A model which consumes both LiDAR *and* RGB photography has a small, but not negligible, performance advantage over a model which only uses LiDAR.
- Regularization techniques such as batch normalization and dropout have been shown to significantly improve the resulting test performance of the trained networks, while data augmentation had an insignificant effect.
- When it comes to the normalization of LiDAR input rasters, Algorithm 3 – "`Nodata-aware metric normalization`" was shown to perform slightly better than Algorithm 2 – "`Nodata-aware local min-max normalization`".
- While semantic segmentation models trained with binary cross-entropy loss (BCEL) showed a purely quantitative advantage on almost all test metrics compared to alternative losses, the soft variant losses still showed a greater degree of "common sense". The soft loss models also showed a greater propensity for ignoring wrong ground truth data during training.

Based on these results, we will train all models with both dropout and batch normalization. LiDAR input raster data will be normalized according to Algorithm 3 – "`Nodata-aware metric normalization`" with parametrization $\gamma = 30$. The semantic segmentation loss function that will be used is the soft Jaccard loss as presented in Equation (3.2).

[1]The original specialization project PDF file can be downloaded from: `https://jakobgm.com/pdf/ntnu_reports/9-project-thesis.pdf`

5.3 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 surface normal rasters. 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 we hypothesize that models based on LiDAR will fare better than RGB models due to the importance of the spatial information when it comes to surface normals. 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.

5.3.1 LiDAR data

We start by training a model based solely on LiDAR data normalized according to Algorithm 3. The training procedure is presented in Figure 5.1.

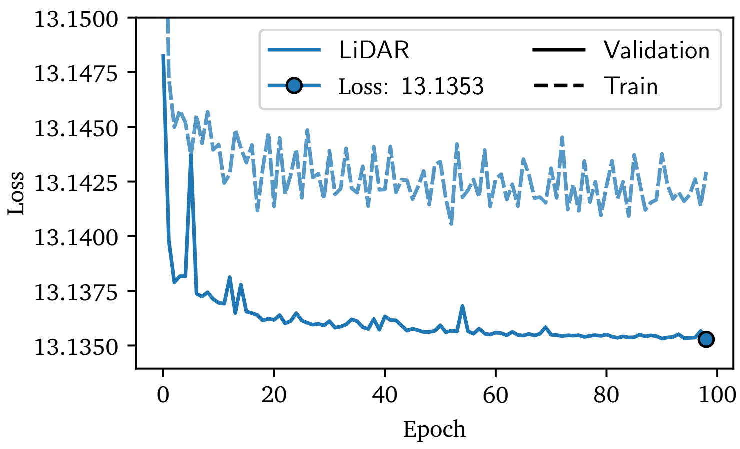


Figure 5.1: Training procedure of LiDAR-only U-Net-derived architecture for predicting surface normals over 100 training epochs. The training epochs are given along the horizontal axis, while the end-of-epoch cosine similarity loss evaluations are given along the vertical axis. Validation split loss is shown as a blue solid line, while the training split loss is shown as a blue dashed line. The epoch yielding the best validation loss is annotated as a solid blue circle, in this case the 99th epoch with a validation loss of 13.1353.

Notice how Figure 5.1 shows that the model has a validation loss which is consistently lower than the training loss. The *tile-averaged*, *batch-summed* cosine similarity loss used during training is highly data-dependent, portraying great variance across tiles, and consequently the data splits, as shown in this figure. We can therefore not necessarily conclude that the model performs better on the validation split than the training split.

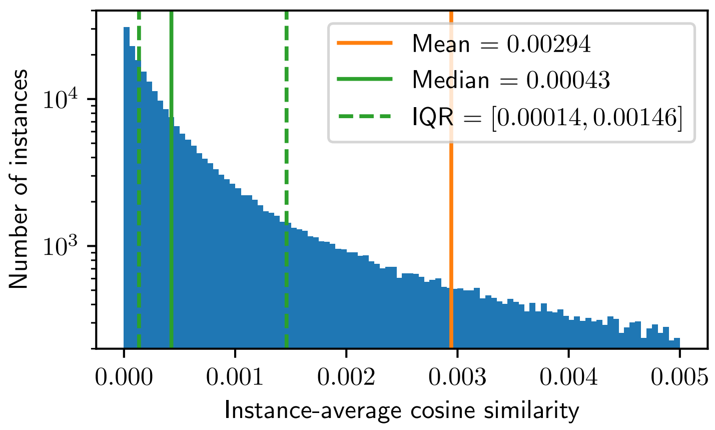


Figure 5.2: Distribution of instance-averaged cosine similarities (IACS) of the LiDAR-only surface normal model over the test split. The instance-average cosine similarity is given along the horizontal axis (truncated to the domain $[0, 0.005]$), while the frequency of each instance-averaged cosine similarity is given along the log-scaled vertical axis. The mean of the instance-averaged cosine similarities is shown as an solid orange line, while the median is annotated as a solid green line. The interquartile range is annotated as dashed green lines. Specific statistical values are provided in the figure legend.

Figure 5.2 shows the distribution of the *instance-averaged cosine similarities* (IACS) over the test split. The instance-averaged cosine similarity for a given instance is defined as the average cosine similarity between the predicted normal vectors (over the respective ground truth *instance* mask) and the ground truth normal vector of the given surface instance.

The median-performing prediction with respect to the average cosine similarity is presented in Figure 5.3 as a representative model prediction. That is, half of the model predictions over the test set perform *worse* than the prediction presented in Figure 5.3, while the other half perform *better*. The result of post-processing the median test prediction is presented in Figure 5.4.

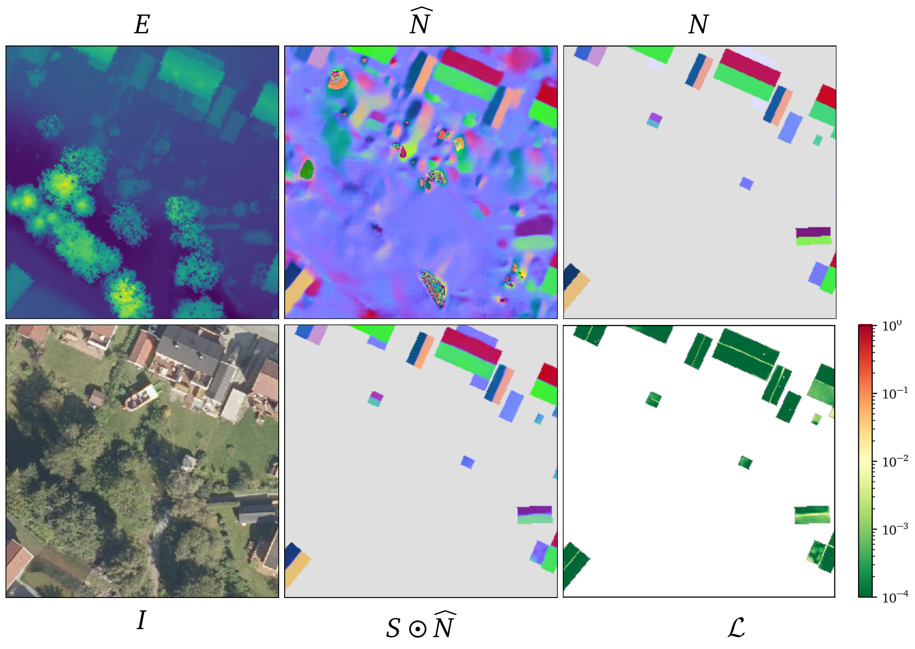


Figure 5.3: Median test prediction for LiDAR-only normal vector model. Each image tile represents the following:

E (upper left) – LiDAR DSM raster.
I (lower left) – Aerial RGB photography.
$\widehat{N}$ (upper middle) – Predicted surface normal vector raster, scaled and translated into value domain $[0, 255]^{H \times W}$ in order to be visualized as an RGB image.
$S \odot \widehat{N}$ (lower middle) – Segmented predicted surface normal vector raster using the ground truth semantic segmentation mask.
N (upper right) – Ground truth surface normal vector raster.
$\mathcal{L}$ (lower right) – Pixel-wise cosine similarity.

The aerial RGB photography input, I, has not been used by the model, but is provided for better visual insight.

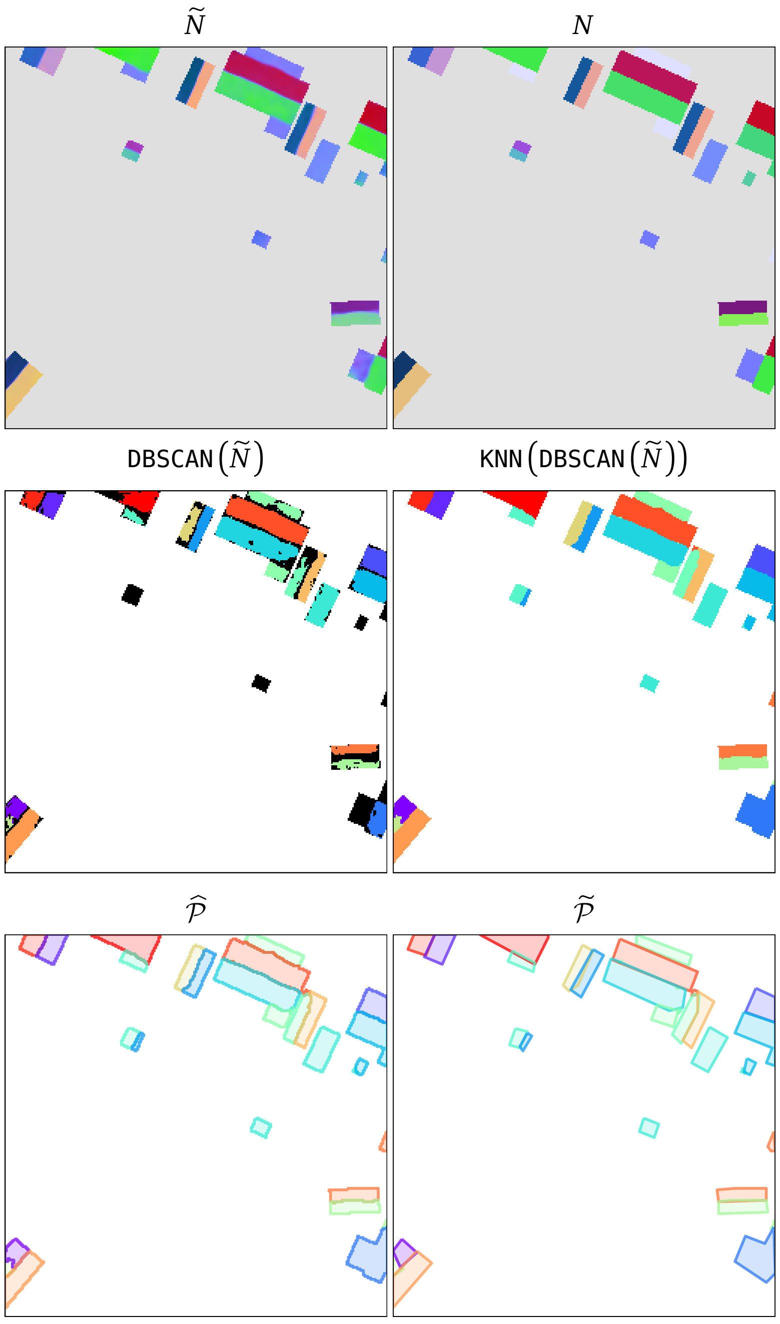


Figure 5.4: Post-processing of median test prediction for LiDAR-only model with respect to the cosine similarity loss component. Each image tile represents the following:

$\widetilde{N}$ (upper left) – Predicted segmented normal vector raster, using the ground truth segmentation mask, $S \odot \widehat{N}$.
N (upper right) – Ground truth normal vector raster.
$\texttt{DBSCAN}\left(\widetilde{N}\right)$ (middle left) – DBSCAN-clustered normal vectors, black indicating noisy outliers.
$\texttt{KNN}\left(\texttt{DBSCAN}\left(\widetilde{N}\right)\right)$ (middle right) – Result of applying KNN-clustering to the DBSCAN-identified noise.
$\widehat{\mathcal{P}}$ (lower left) – Unsimplified predicted polygons.
$\widetilde{\mathcal{P}}$ (lower right) – Simplified predicted polygons.

5.3.2 RGB data

A model using only RGB data is trained, and the training procedure is summarized in Figure 5.5 side-by-side the LiDAR-only model for comparison purposes.

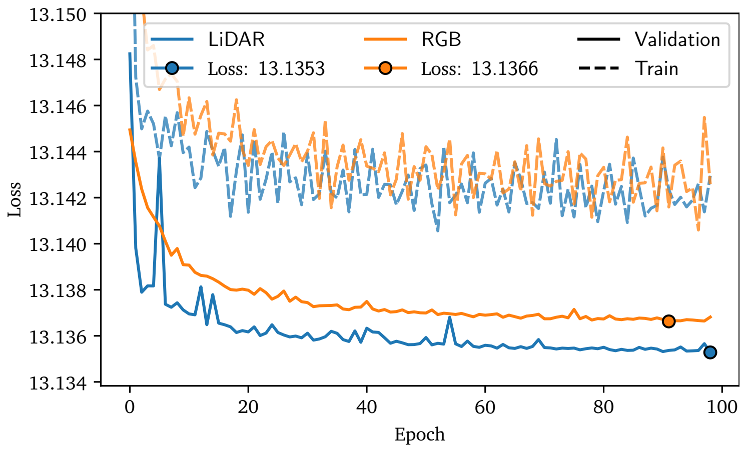


Figure 5.5: Training procedure for surface normal vector models using *either* LiDAR or RGB as input. The training epochs are given along the horizontal axis, while the end-of-epoch cosine similarity loss evaluations are given along the vertical axis. The LiDAR-only model is shown in blue, while the RGB-only model is shown in orange. The training split loss is shown with dashed lines, while the validation split is shown with solid lines. The best validation loss epochs are annotated with solid circles, with specific validation loss values provided in the figure legend.

From Figure 5.5 it is immediately obvious that a RGB-only model performs worse than a LiDAR-only model, as hypothesized. In order to confirm this, the instance-averaged cosine similarity distribution over the test set is yet again plotted in Figure 5.6, this time for the RGB-only model instead. As can be seen in Figure 5.6 the mean instance-averaged cosine similarity evaluated over the test set increases from 0.002 94 to 0.004 02, and the median increases from 0.000 43 to 0.000 78. The RGB-only surface normal vector model can therefore be concluded to be strictly worse than the LiDAR-only equivalent model.

Yet again we plot the median test prediction of the RGB-only model in Figure 5.7 as a representative prediction. The result of post-processing the median test prediction is presented in Figure 5.8.

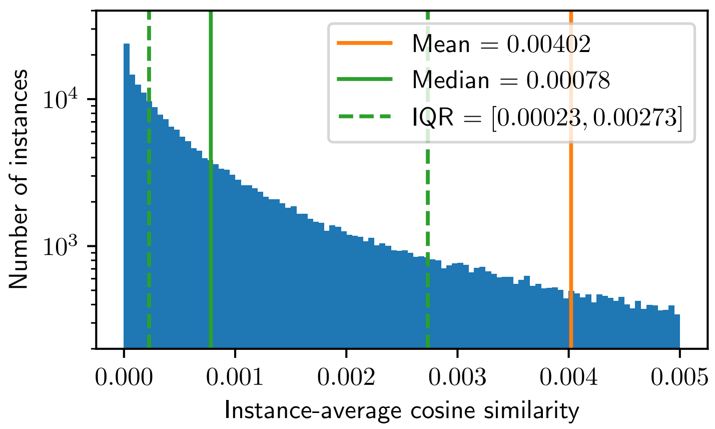


Figure 5.6: Distribution of instance-averaged cosine similarities of the RGB-only surface normal model over the test split. See Figure 5.2 for detailed figure description.

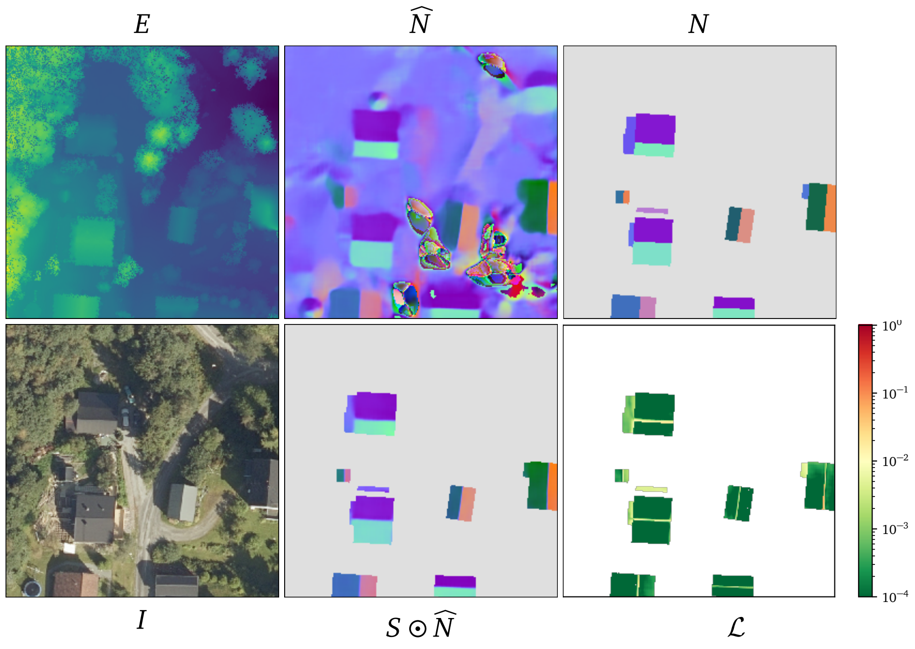


Figure 5.7: Median test prediction for RGB-only normal vector model. The LiDAR DSM input, E, has not been used by the model, but is provided for better visual insight. See Figure 5.3 for detailed figure description.

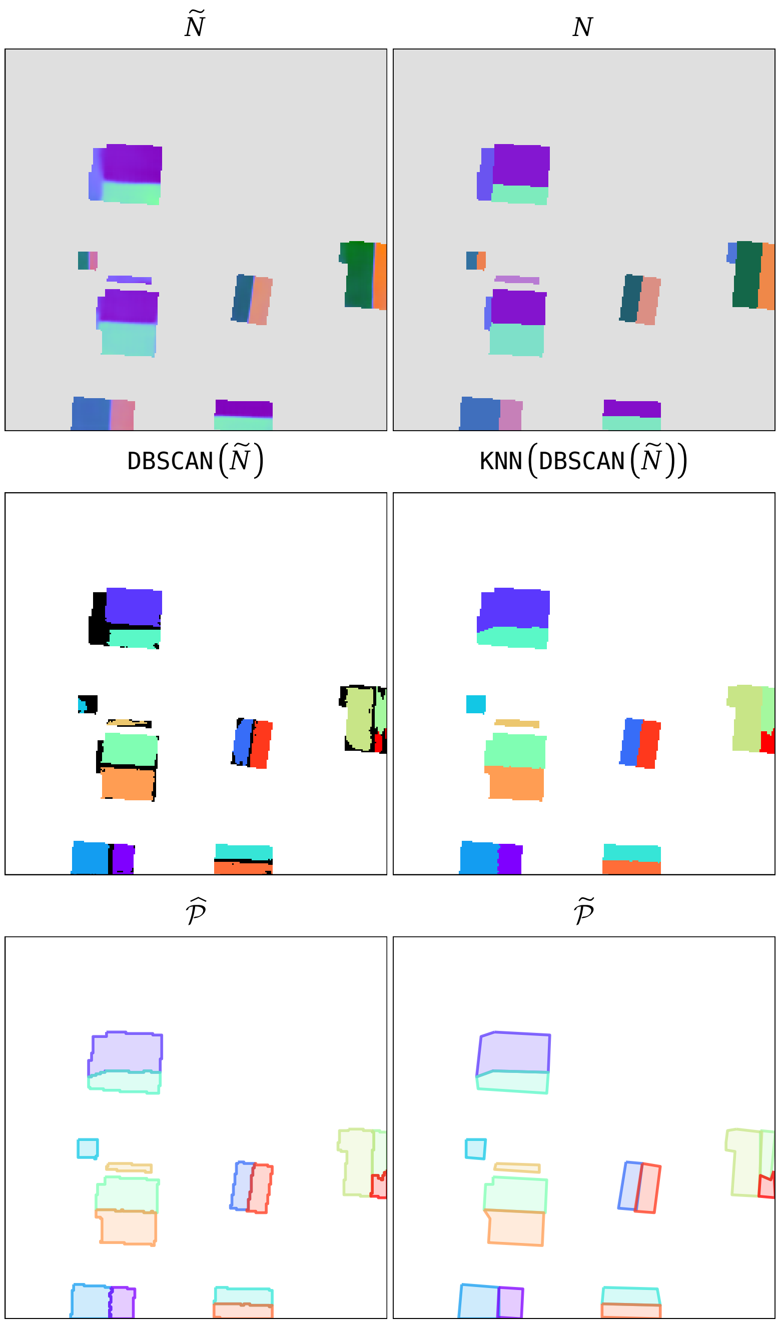

$\widetilde{N}$
N
$\texttt{DBSCAN}(\widetilde{N})$
$\texttt{KNN}(\texttt{DBSCAN}(\widetilde{N}))$
$\widehat{\mathcal{P}}$
$\widetilde{\mathcal{P}}$

Figure 5.8: Post-processing of median test prediction for RGB-only model with respect to the cosine similarity loss component. See Figure 5.4 for a detailed figure description.

5.3.3 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 5.9, and the training procedures of the two single-input models have been included for comparison.

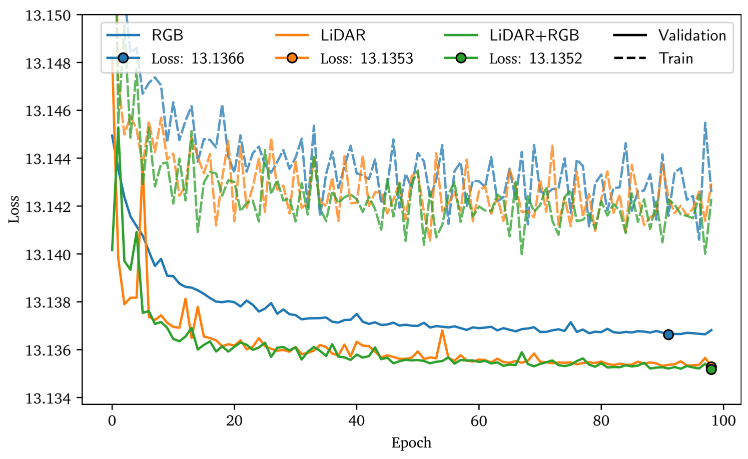


Figure 5.9: Training procedure for surface normal vector models using LiDAR and/or RGB as input. The training epochs are given along the horizontal axis, while the end-of-epoch cosine similarity loss evaluations are given along the vertical axis. The LiDAR-only model is shown in blue, the RGB-only model in orange, and finally the combined input model in green. The training split loss is shown with dashed lines, while the validation split is shown with solid lines. The best validation loss epochs are annotated with solid circles, with specific validation loss values provided in the figure legend.

Figure 5.9 does not show any immediately obvious in predictive performance between the LiDAR-only model and the combined input model. The distribution of the instance-averaged cosine similarities over the test set have been plotted in Figure 5.10.

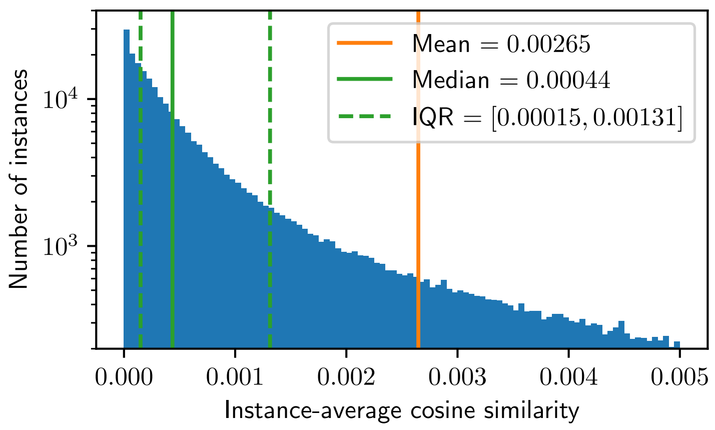


Figure 5.10: Distribution of instance-averaged cosine similarities of the combined input surface normal model over the test split. See Figure 5.2 for detailed figure description.

Compared to the LiDAR-only model, the combined input model has decreased the mean instance-averaged cosine similarity from 0.002 94 to 0.002 65, and negligible increased the median from 0.000 43 to 0.000 44. With other words, the combined input model has not increased the predictive performance as much as the LiDAR-only model improved upon the RGB-only model, but it still has somewhat better test performance.

The median test prediction of the combined input model is provided in Figure 5.11. The result of post-processing the median test prediction is presented in Figure 5.12.

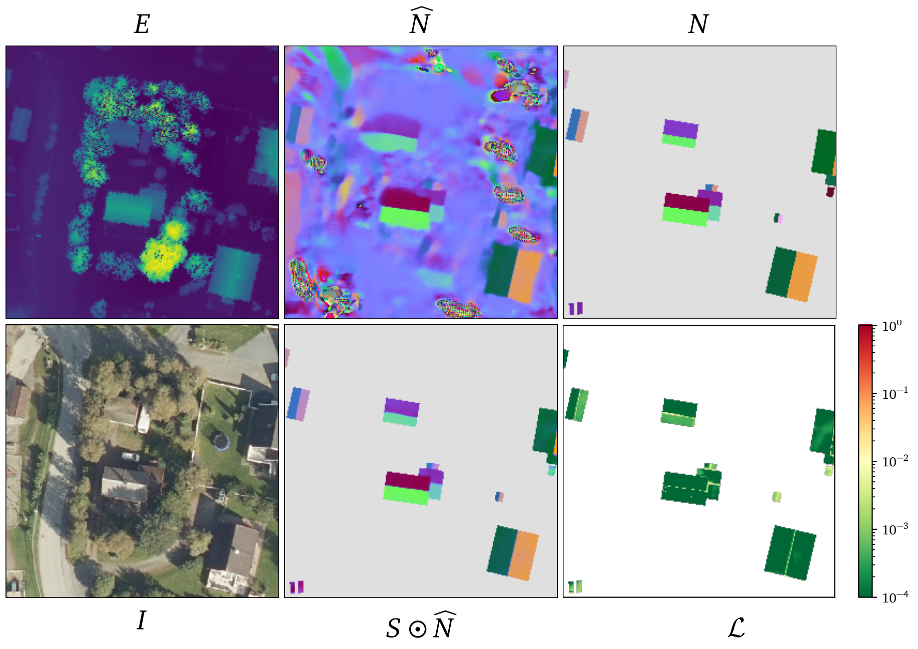


Figure 5.11: Median test prediction for the combined input normal vector model. See Figure 5.3 for detailed figure description.

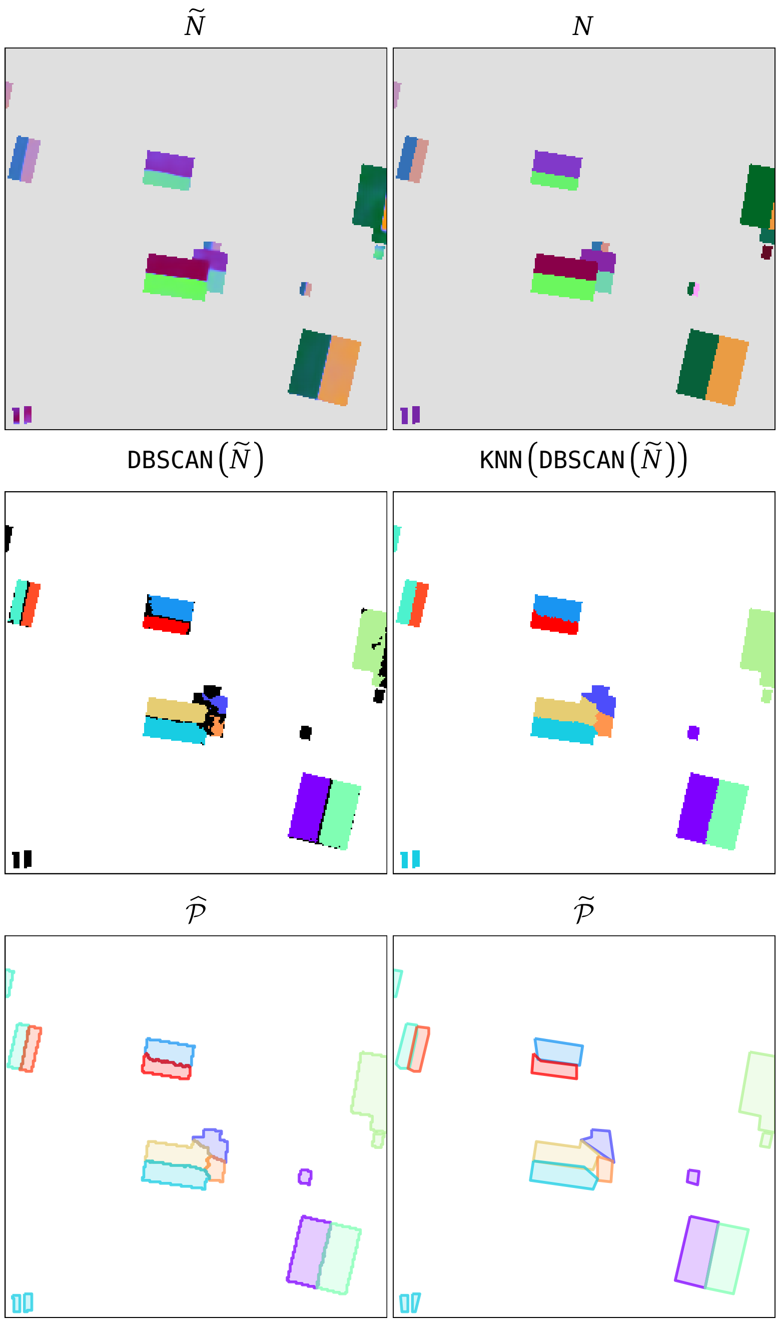


Figure 5.12: Post-processing of median test prediction for combined-input model with respect to the cosine similarity loss component. See Figure 5.4 for a detailed figure description.

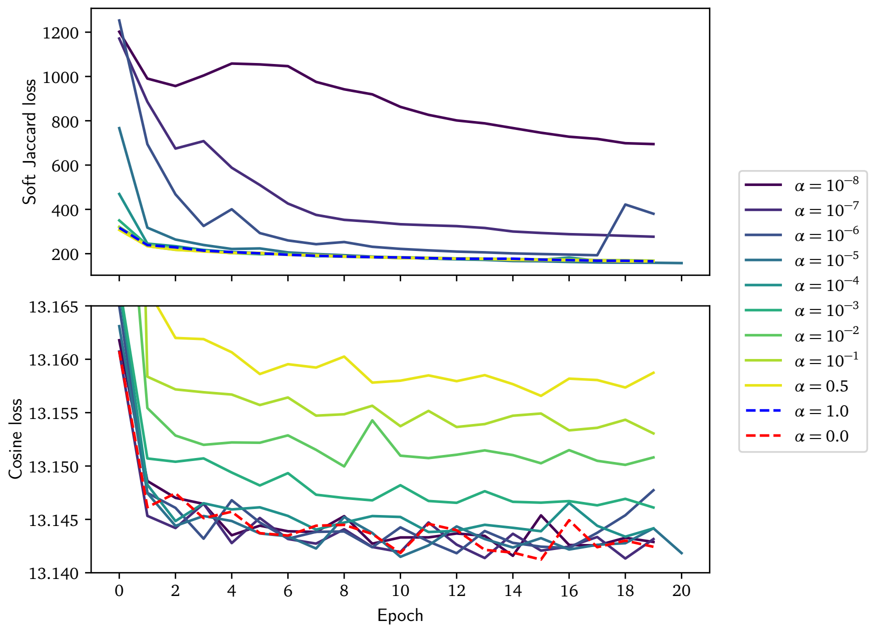
5.4 Multitask learning

We now intend to train a multitask learning (MTL) model which simultaneously predicts semantic roof segmentation masks *and* surface normal vector rasters. Both types of remote sensing data, LiDAR and aerial RGB photography, will be used by the multitask model. The main difficulty lies in constructing a proper loss function which can be optimized such that *both* tasks are solved simultaneously to a satisfactory degree. We will use a conceptually simple multitask loss function, denoted as $\mathcal{L}_{\text{MT}}$, which is a simply a linear combination of the soft Jaccard loss, $\mathcal{L}_{\text{SJL}}$ as defined in Equation (3.2), and the pixel-averaged cosine similarity loss, $\mathcal{L}_{\text{norm}}$ as defined in Equation (3.4),

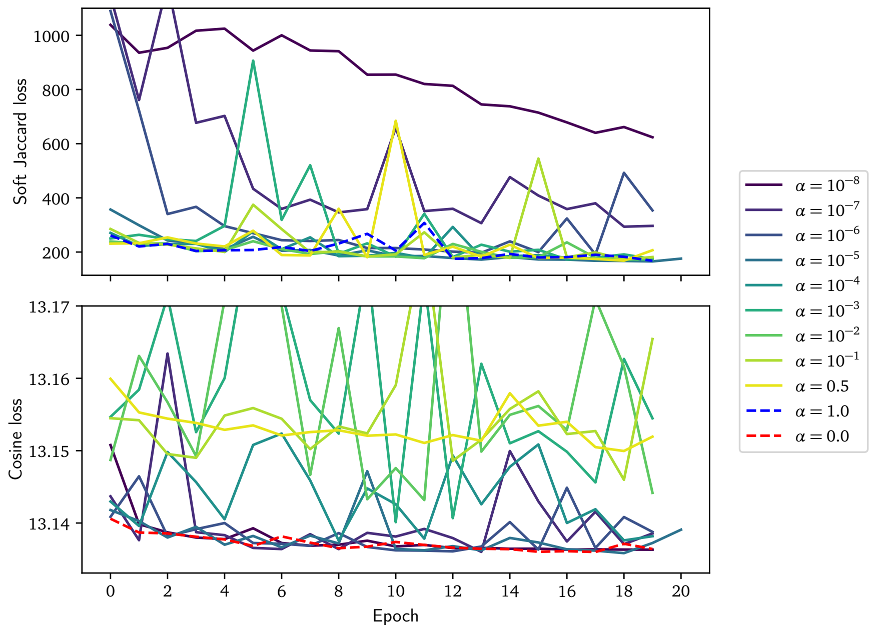
$$\begin{aligned}\mathcal{L}_{\text{MT}}\left(\widehat{S},\widehat{N};S,N\right) &:= \alpha\cdot\mathcal{L}_{\text{SJL}}\left(\widehat{S};S\right)+(1-\alpha)\cdot\mathcal{L}_{\text{norm}}\left(\widehat{N};N\right)\\ &= \alpha-\frac{\alpha\cdot\sum\limits_{i=1}^{HW}\widehat{S}_iS_i}{\sum\limits_{i=1}^{HW}\left(\widehat{S}_i+S_i-\widehat{S}_iS_i\right)}+\frac{1-\alpha}{HW}\sum_{i=1}^{H}\sum_{j=1}^{W}\left(1-N_{i,j}^T\,\widehat{N}_{i,j}\right).\end{aligned}$$

where $0\leq\alpha\leq 1$ determines the weighting of the linear combination. Now, the main challenge is to select a suitable value for the hyperparameter α; too small and the semantic segmentation task is neglected in favor of predicting surface normal vectors during training, and vice versa if α is set too close to 1. In order to find the most suitable value for α, we train the same model architecture several times from scratch for 20 epochs, but with different parametrizations of α. The resulting models are then evaluated in order to determine which α value that should be used for a full 100 epochs of training. We will specifically perform a hyperparameter search over the following loss weightings: $\alpha\in\{0,10^{-8},10^{-7},10^{-6},10^{-5},10^{-4},10^{-3},10^{-2},10^{-1},0.5,1\}$. Here $\alpha=0$ implies using a single-task surface normal vector architecture, trained solely with the pixel-averaged cosine similarity loss function, $\mathcal{L}_{\text{norm}}$. Likewise, $\alpha=1$ implies using a single-task semantic segmentation architecture trained with the soft Jaccard loss function, $\mathcal{L}_{\text{SJL}}$. These two single-task models are included for comparison purposes.

The end-of-epoch value decompositions of the multitask loss function, $\mathcal{L}_{\text{MT}}$, into the two *unweighted* linear components, $\mathcal{L}_{\text{SJL}}$ and $\mathcal{L}_{\text{norm}}$, are presented in Figures 5.13a and 5.13b. As can be observed from these figures, both loss components are affected by the specific value of α when it comes to optimization. The general training and validation loss trends are as expected; the greater the value for α, the better the network performs at producing semantic roof segmentation masks, and likewise for small values for α resulting in more accurate surface normal vector rasters. More interestingly, however, is that there seems to be a “saturation effect” when α approaches 0 and 1, yielding diminishing performance increases in the respective task efficiencies. Is it possible to choose a value for α which yields optimal model performance in both tasks simultaneously?



(a) End-of-epoch *training* loss decomposition for different values of α over 20 training epochs.



(b) End-of-epoch *validation* loss decomposition for different values of α over 20 training epochs.

Figure 5.13: The training epochs are given along the horizontal axis, while the end-of-epoch loss values are given along the vertical axis. The top plots show the *unweighted* soft Jaccard loss, $\mathcal{L}_{\text{SJL}}$, while the bottom plots show the *unweighted* batch-summed, pixel-averaged cosine similarity loss, $\mathcal{L}_{\text{norm}}$. The value for α used during training is indicated by the color of the line as described in the right-hand color legend. The single-task network losses are shown with dashed lines.

Figure 5.14 presents the *best* end-of-epoch loss component values across *all* 20 training epochs for all values of α. Both loss components ($\mathcal{L}_{\text{SJL}}$ and $\mathcal{L}_{\text{norm}}$) and both data splits (training and validation) are included.

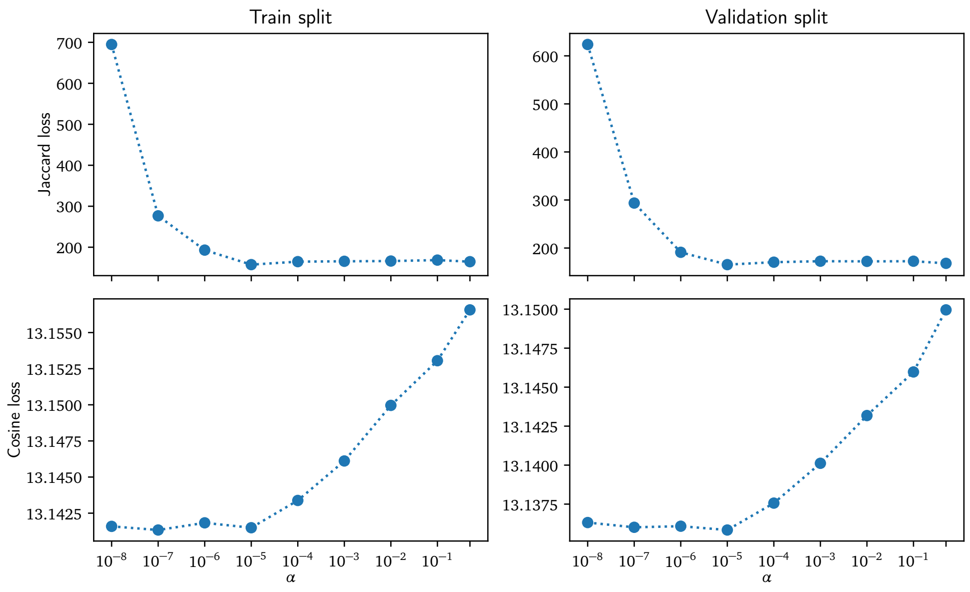


Figure 5.14: Best loss component value across all 20 training epochs for each value of α. The value of α is provided along the log-scaled horizontal axis, while the best loss values are provided along the vertical axis. The top row shows the soft Jaccard loss, $\mathcal{L}_{\text{SJL}}$, while the bottom row shows the pixel-averages cosine similarity loss, $\mathcal{L}_{\text{norm}}$. The left column is evaluated on the train split, while the right column is evaluated on the validation split.

We can conclude from Figure 5.14 that $\alpha = 10^{-5}$ is the obvious choice since the resulting model uncompromisingly performs at least as well as *all* other models on *both* tasks simultaneously. For this reason, going forwards, we will exclusively parametrize the multitask loss function with $\alpha = 10^{-5}$.

We now train a multitask model for 100 full epochs, using the multitask loss function $\mathcal{L}_{\text{MT}}$ with $\alpha = 10^{-5}$. The full training procedure is provided in Figure 5.15.

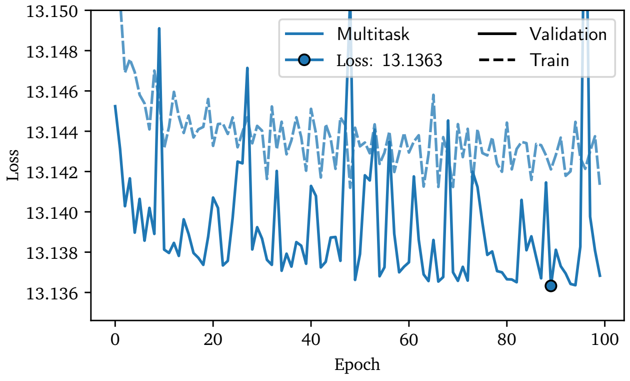
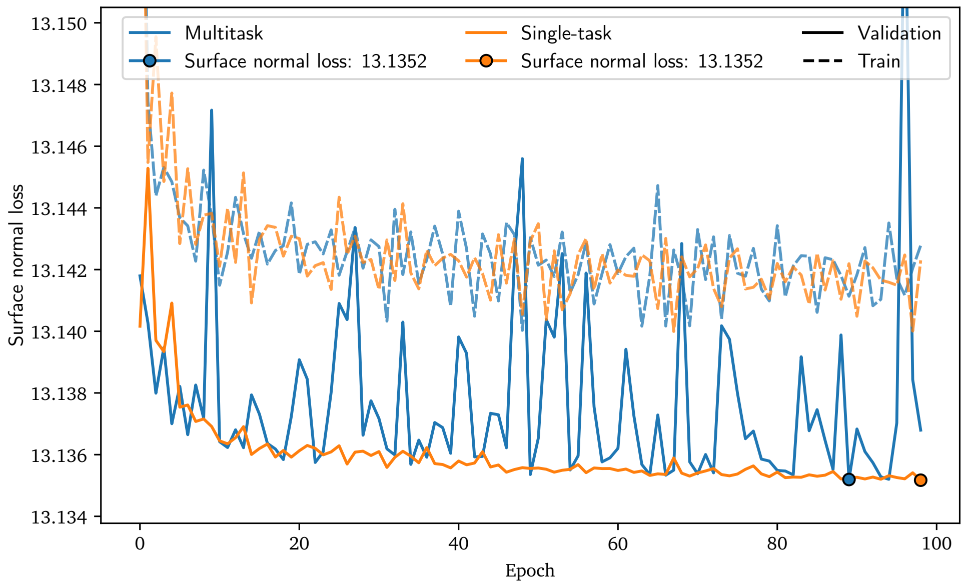


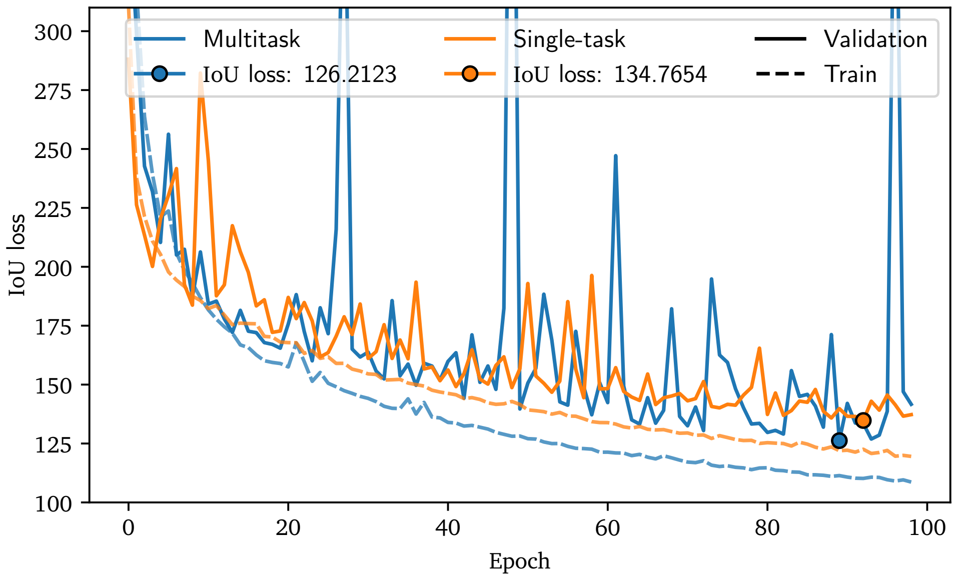
Figure 5.15: Training procedure of combined input, U-Net-derived multitask architecture ($\alpha = 10^{-5}$) for predicting semantic segmentation masks *and* surface normals over 100 training epochs. The training epochs are given along the horizontal axis, while the end-of-epoch multitask loss evaluations, $\mathcal{L}_{\text{MT}}$, are given along the vertical axis. The validation split loss is shown as a blue solid line, while the training split loss is shown as a blue dashed line. The epoch yielding the best validation loss is annotated as a solid blue circle, in this case a validation loss of 13.1363.

As can be seen in Figure 5.15, the validation loss shows a large degree of variance from epoch to epoch. In order to investigate the behaviour of the loss function over each training epoch, we plot the decomposition of $\mathcal{L}_{\text{MT}}$ into $\mathcal{L}_{\text{SJL}}$ and $\mathcal{L}_{\text{norm}}$ in Figure 5.16. For comparison purposes we include the losses of the respective single-task architectures as well. As can be seen in Figure 5.16, the multitask network portrays a greater degree of instability in its validation loss compared to the respective single-task networks, especially for the pixel-averaged cosine similarity loss. Although the multitask network portrays a greater degree of instability during training, it still manages to converge to a general performance level which is comparable to the respective single-task models. For the semantic segmentation task, specifically, the multitask network even performs *better* than the respective single-task roof segmentation model. It can also be observed that the semantic segmentation loss generally converges slower than the surface normal vector loss.

The end-of-epoch model parametrization which yields the best loss value when evaluated over the validation split is chosen as the final model parametrization. Figure 5.17 shows the distribution of the instance-averaged cosine similarities (IACS) over the test split. Compared to the single-task surface normal vector model, the multitask model has *increased* the mean test IACS from 0.002 65 to 0.002 84, while negligible decreased the median from 0.000 43 to 0.000 42. Based on these results, we consider the multitask model to be approximately equally accurate as the respective single-task model.



(a) Batch-summed, pixel-averaged cosine similarity loss component, $\mathcal{L}_{\text{norm}}$, for both the multitask network ($\alpha = 10^{-5}$) and single-task surface normal vector network.



(b) Soft Jaccard loss component, $\mathcal{L}_{\text{SJL}}$, for both the multitask network ($\alpha = 10^{-5}$) and single-task semantic roof surface segmentation network.

Figure 5.16: End-of-epoch loss components of multitask network compared to respective single-task losses. The multitask network losses are annotated in blue, while the respective single-task network losses are annotated in orange. Training losses are shown as dashed lines, while validation losses are shown as solid lines. The best validation losses across all 100 epochs are annotated as solid circles.

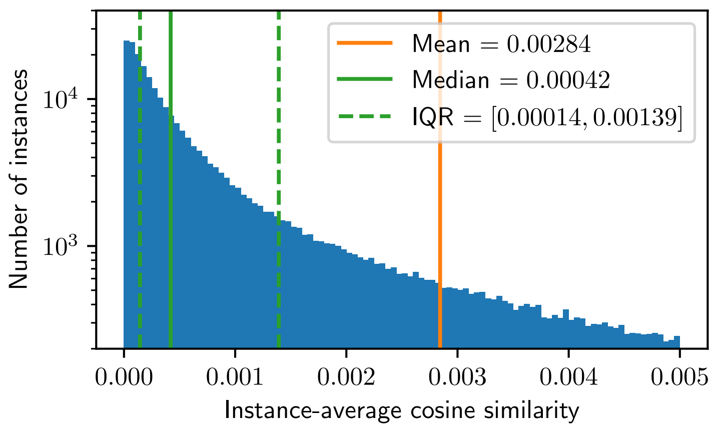
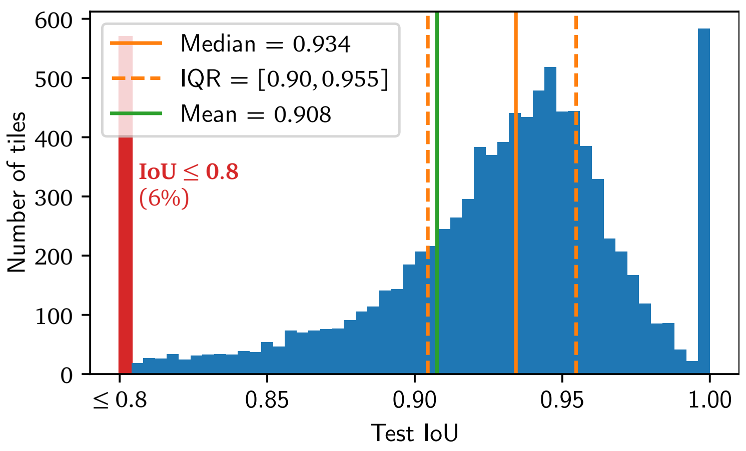


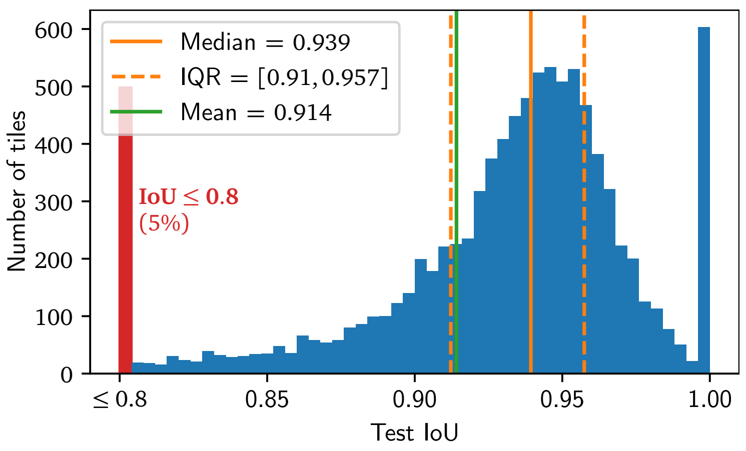
Figure 5.17: Distribution of instance-averaged cosine similarities of the surface normal vector rasters produced by the multitask network over the test split. See Figure 5.2 for detailed figure description.

Now that we have concluded that the multitask model performs accurately on the task of predicting surface normal vector rasters, we investigate the predictive accuracy of the multitask model when it comes to semantic roof segmentation. The distribution of the test IoU evaluations for the multitask model is provided in Figure 5.18b, while the test IoU distribution of a respective single-task semantic roof segmentation model is provided in Figure 5.18a for comparison purposes. As can be seen in Figure 5.18, the multitask model increases the mean test IoU from 0.934 to 0.939, and the median test IoU from 0.908 to 0.914, compared to the single-task model. The number of test performance outliers, that is IoU ≤ 0.8, also decreases from 6 % to 5 %. With other words, we can conclude that a multitask model which produces both semantic roof surface segmentation maps *and* surface normal vector rasters simultaneously ends up performing better than the set of respective single-task models.

While it has been established that the multitask model outperforms the single-task segmentation 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 5.19. If the 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 5.19, which is clearly not the case here due to the multitask model outperforming the single-task model. While the multitask model is on average better than the single-task model, single-task still outperforms multitask in about 40.4 % of the test cases. This may be partly caused by the randomness introduced into the training procedure, and thus the final model parametrization.



(a) Test IoU distribution for single-task semantic roof surface segmentation model.



(b) Test IoU distribution for semantic segmentation masks produced by multitask learning model.

Figure 5.18: Distribution of IoU evaluations of the combined-input models (single-task and multitask) over tiles from the test set. The left tail of the distribution (IoU ≤ 0.8) of the IoU 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.

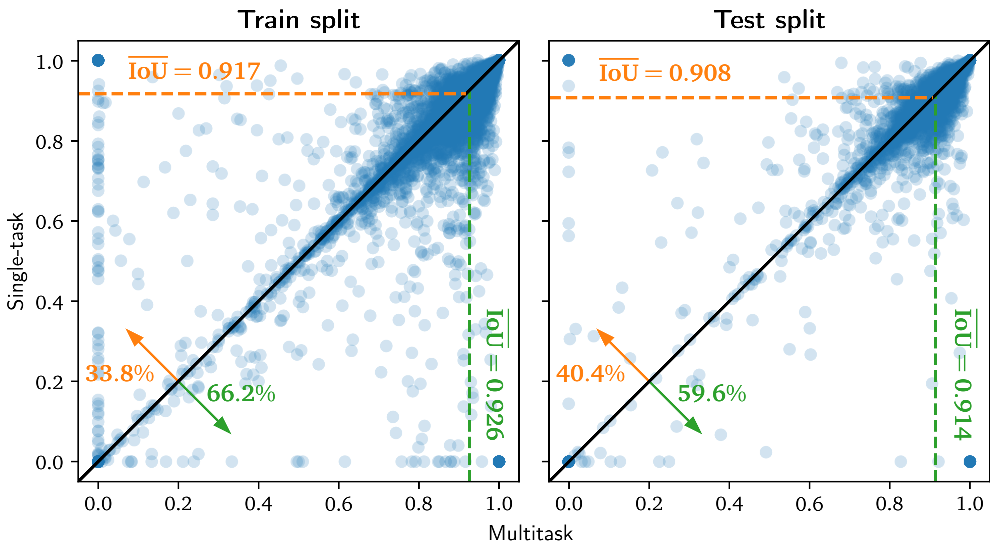


Figure 5.19: Scatter plot showing the correlation between the IoU evaluation metric performance of two surface segmentation models, single-task vs. multitask. Each blue scatter point (x_i, y_i) corresponds to a given tile, i, where the x-coordinate is the IoU metric of the multitask model prediction and the y-coordinate is the IoU metric of the single-task 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 single-task model for the respective splits, while the vertical dashed lines in green indicate the *mean* IoU for the multitask 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 single-task model performs better than the multitask model, while scatter points located *below* the diagonal represent tiles where the multitask model performs better than the single-task model.

The median test prediction for the multitask model with respect to the cosine similarity loss component is presented in Figure 5.20. The *predicted* segmentation mask is used in order to segment the predicted surface normals instead of the *ground truth* segmentation mask as in earlier prediction plots.

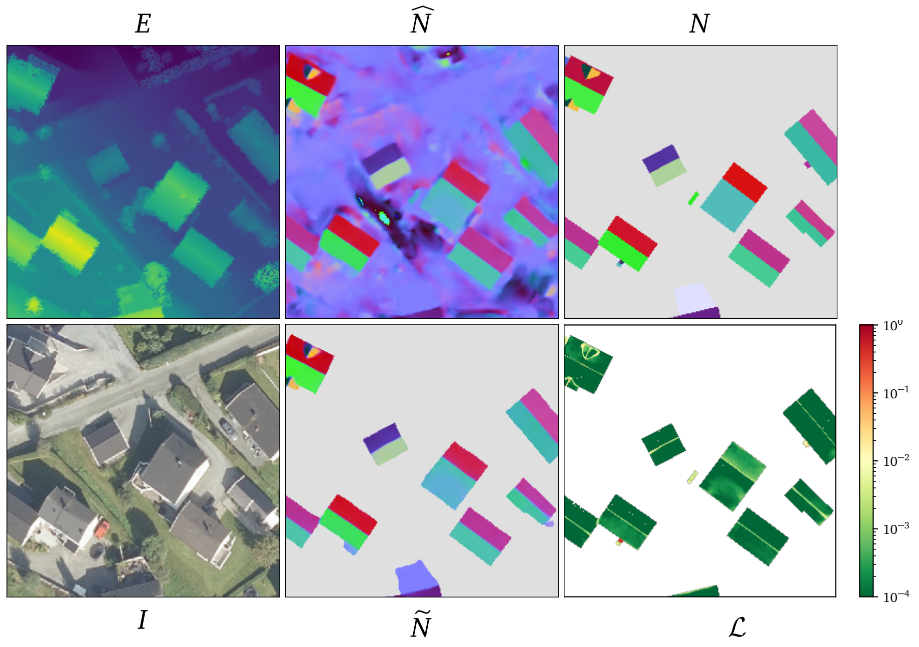


Figure 5.20: Median test prediction for multitask model with respect to the cosine similarity loss component. Each image tile represents the following:

E (upper left) – LiDAR DSM raster.
I (lower left) – Aerial RGB photography.
$\widehat{N}$ (upper middle) – Predicted surface normal vector raster.
$\widetilde{N}$ (lower middle) – Predicted *segmented* surface normal vector raster, $\widehat{S} \odot \widehat{N}$.
N (upper right) – Ground truth surface normal vector raster.
$\mathcal{L}$ (lower right) – Pixel-wise cosine similarity.

The result of post-processing the median test prediction is presented in Figure 5.21.

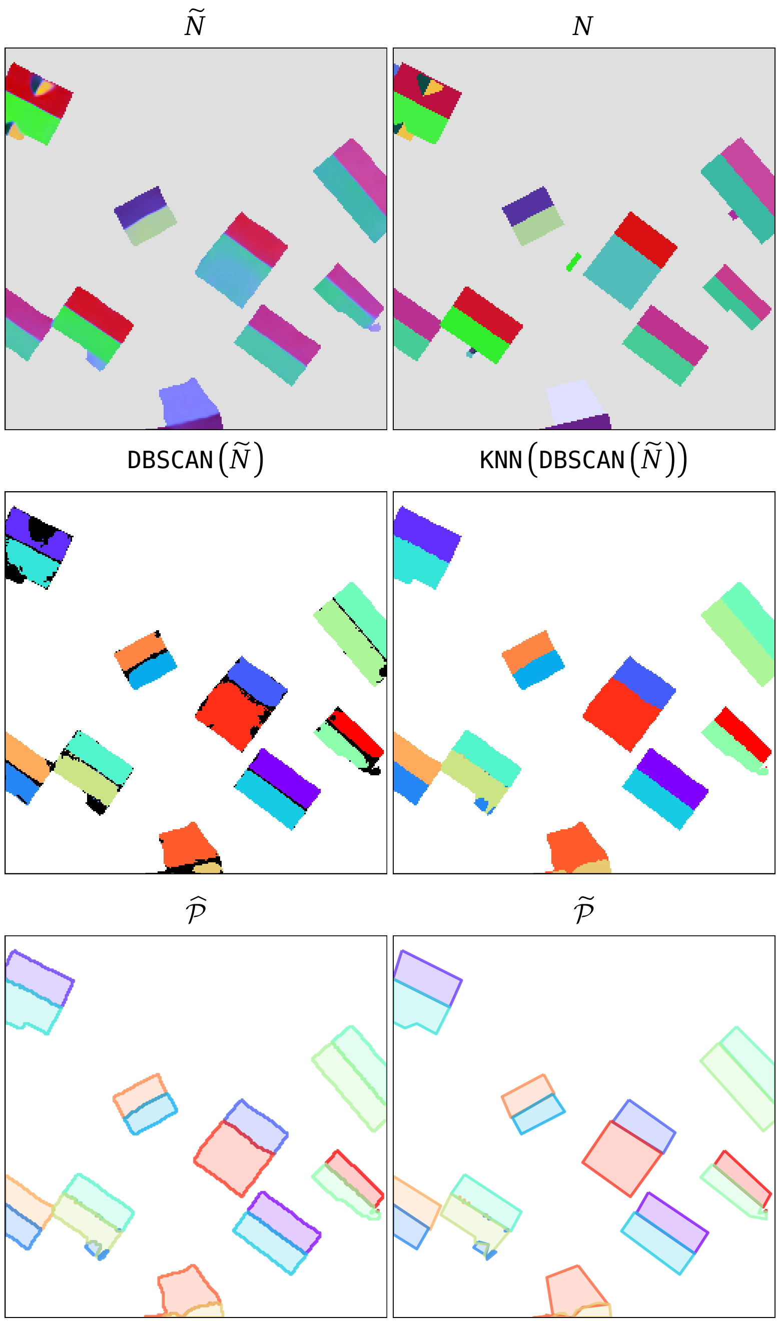


Figure 5.21: Post-processing of median test prediction for multitask model with respect to the cosine similarity loss component. Each image tile represents the following:

$\widetilde{N}$ (upper left) – Predicted segmented normal vector raster, $\widehat{S} \odot \widehat{N}$.
N (upper right) – Ground truth normal vector raster.
DBSCAN$(\widetilde{N})$ (middle left) – DBSCAN-clustered normal vectors, black indicating noisy outliers.
KNN$(\text{DBSCAN}(\widetilde{N}))$ (middle right) – Result of applying KNN-clustering to the DBSCAN-identified noise.
$\widehat{\mathcal{P}}$ (lower left) – Unsimplified predicted polygons.
$\widetilde{\mathcal{P}}$ (lower right) – Simplified predicted polygons.

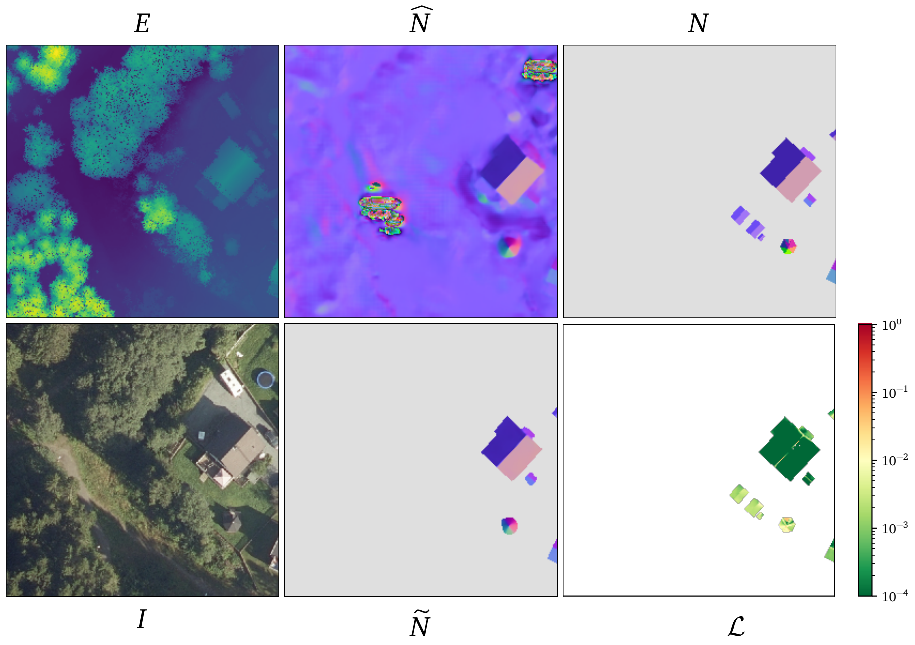


Figure 5.22: 95th percentile test prediction (bad prediction outlier) for multitask model with respect to the cosine similarity loss component. See Figure 5.20 for detailed figure description.

So far, we have only shown median test predictions with respect to the loss. In Figures 5.22 and 5.23 we show the 95th percentile test prediction and respective post-processing in order to demonstrate a negative prediction outlier. A common trait often observed in negative outliers is the presence of many small roof surfaces adjacent to each other, which can also be observed in Figure 5.22 with the circular construction in the lower right quadrant.

Figures 5.24 and 5.25, on the other hand, show the 5th percentile test prediction in order to demonstrate an exceptionally good surface normal prediction. Such positive outliers are often dominated by large rectangular surfaces and cleanly delineated boundaries.

The polygon simplification method, $\widehat{\mathcal{P}} \rightarrow \widetilde{\mathcal{P}}$, as exemplified in Figures 5.21, 5.23 and 5.25, can be considered as moderately successful, depending on the intended application of the simplified polygons. Simplification of polygons generally results in longer, straight edges where the ground truth polygons also have long edges, but the simplified polygons often wrongly represent protruding roof sections. The non-simplified polygons usually overlap the ground truth polygons to a greater degree, but the complexity of the simplified polygons is greatly reduced. This must be considered as a trade-off which is highly dependent on the given application of the roof surface polygons.

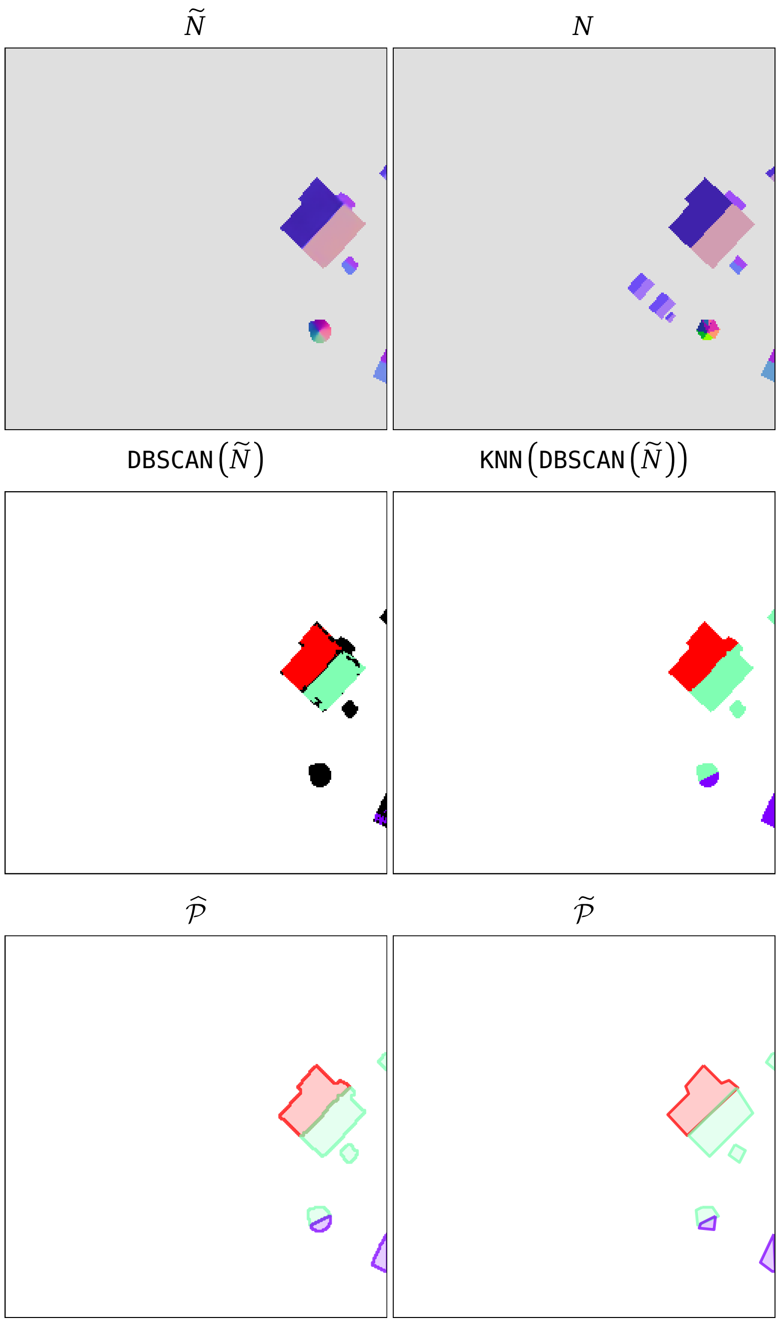

$\widetilde{N}$
N
$\texttt{DBSCAN}\left(\widetilde{N}\right)$
$\texttt{KNN}\left(\texttt{DBSCAN}\left(\widetilde{N}\right)\right)$
$\widehat{\mathcal{P}}$
$\widetilde{\mathcal{P}}$

Figure 5.23: Post-processing of 95th percentile test prediction (bad prediction outlier) for multitask model with respect to the cosine similarity loss component. See Figure 5.21 for detailed figure description.

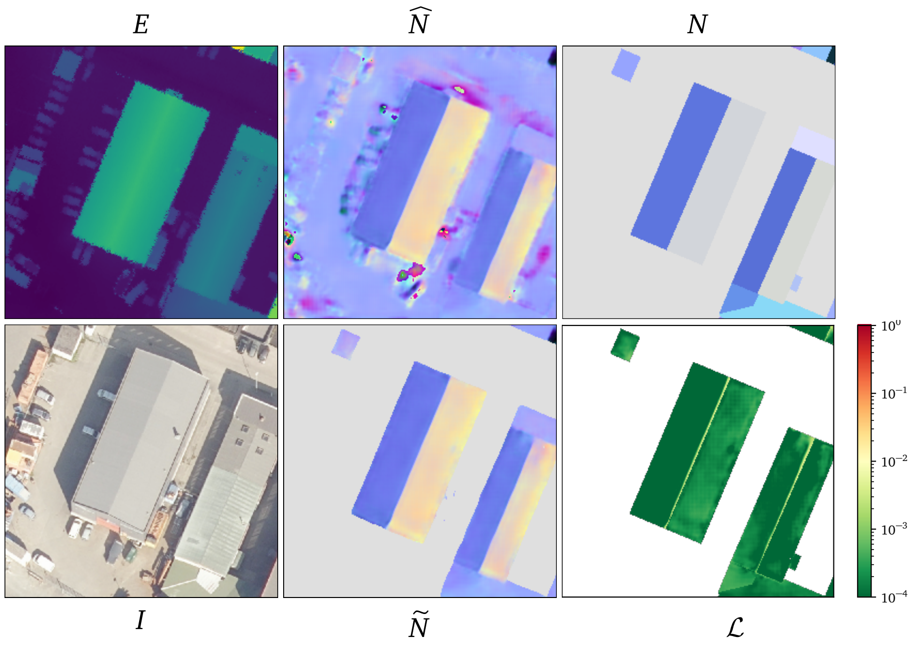


Figure 5.24: 5th percentile test prediction (good prediction outlier) for multitask model with respect to the cosine similarity loss component. See Figure 5.20 for detailed figure description.

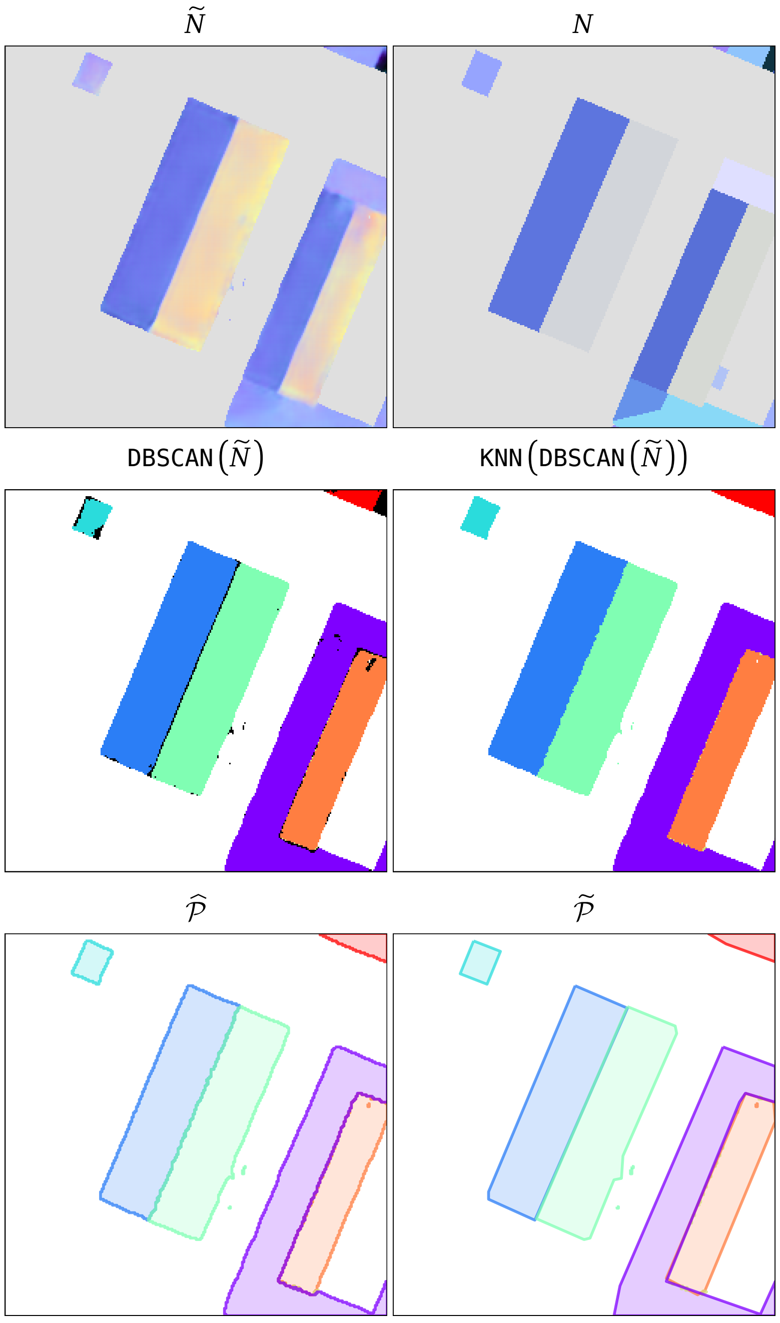

$\widetilde{N}$
N
$\texttt{DBSCAN}(\widetilde{N})$
$\texttt{KNN}(\texttt{DBSCAN}(\widetilde{N}))$
$\widehat{\mathcal{P}}$
$\widetilde{\mathcal{P}}$

Figure 5.25: Post-processing of 5th percentile test prediction (good prediction outlier) for multitask model with respect to the cosine similarity loss component. See Figure 5.21 for detailed figure description.

Chapter 6

Conclusion and Further Work

We consider there to be two main conclusions to be drawn from the experiments presented in Chapter 5:

- LiDAR input data is essential in order to predict accurate surface normal vector rasters, but RGB can be used in *addition* in order to marginally increase predictive performance.
- A multitask model which predicts *both* semantic segmentation masks *and* surface normal vectors ends up being more accurate at segmentation than a respective single-task segmentation model.

We have now presented a complete and novel end-to-end pipeline for predicting three-dimensional flat roof surface polygons. The pipeline can be considered to consist of four sequential components:

1. **Feature engineering** – Construct semantic segmentation masks and surface normal rasters from ground truth roof surface GIS data.
2. **Prediction** – Construct and train CNN architecture(s) for the prediction of semantic segmentation masks and surface normal rasters.
3. **Instance clustering** – Use predicted surface normal rasters in order to partition the semantic segmentation masks.
4. **Vectorization** – Reconstruct three-dimensional polygons from original LiDAR data in combination with the processed instance segmentation masks.

The first component, “feature engineering”, can be considered as the problem formulation, while the remaining three components are constructed in order to solve the problem. The three “solution components” are highly modular and composable, and different implementations can easily replace any of them. We consider this thesis a proof-of-concept of sorts, establishing a rough sketch of the types of building blocks required in order to solve the problem at hand. Here are some examples of how these building blocks can be improved upon in further work:

- Implement methods for performing image augmentation during training when surface normal vector rasters are involved. Any rotation of the input raster data must be performed on each individual ground truth normal vector.
- Utilize newer neural network architectures for the prediction and refinement of semantic segmentation masks and surface normal rasters, e.g. GANs.
- Investigate the application of different activation functions in the surface normal vector raster output layer. Preferably utilize the fact that $\|\boldsymbol{n}\| \equiv 1$ and $n_z \geq 0$ to a greater degree.
- Implement other polygon simplification methods which uses more domain-specific knowledge. Simplified polygons which share a common border *before* simplification should preferably still share the same border *after* simplification, for instance.
- Implement metrics for evaluating the accuracy of the "instance clustering" and "vectorization" components. Use these metrics in order to perform a proper hyperparameter search for any parameters used by these clustering and vectorization algorithms. These metrics can also be used in order to compare different algorithms all together.

As far as we know, the prediction and subsequent post-processing of surface normal vectors is an entirely new and novel machine learning task, and the applications of such surface normal vector rasters are manifold. Our specific pipeline utilizes these normal rasters in order to partition semantic segmentation masks and reconstruct three-dimensional surface polygons, but there exists several other real world applications. One such alternative application of (segmented) surface normal rasters is the calculation of solar irradiance[1] for any given location covered by remote sensing data.

[1]*Solar irradiance* $[\mathrm{W\,m^{-2}}]$ is the radiant solar flux received by a surface per unit area.

Bibliography

[1] C. A. Northend, R. C. Honey, and W. E. Evans, “Laser radar (lidar) for meteorological observations,” *Review of Scientific Instruments*, vol. 37, no. 4, pp. 393–400, 1966. DOI: 10.1063/1.1720199. eprint: https://doi.org/10.1063/1.1720199. [Online]. Available: https://doi.org/10.1063/1.1720199.

[2] R. O. Dubayah and J. B. Drake, “Lidar Remote Sensing for Forestry,” *Journal of Forestry*, vol. 98, no. 6, pp. 44–46, Jun. 2000, 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. [Online]. Available: https://doi.org/10.1093/jof/98.6.44.

[3] H. Ozdemir, C. Sampson, G. A. de Almeida, and P. Bates, “Evaluating scale and roughness effects in urban flood modelling using terrestrial lidar data,” *Hydrology and Earth System Sciences*, vol. 10, pp. 5903–5942, 2013.

[4] J. Hecht, “Lidar for self-driving cars,” *Optics and Photonics News*, vol. 29, no. 1, pp. 26–33, 2018.

[5] J. G. Martinussen, “Building footprint detection using remote sensing data,” Norwegian University of Science and Technology, 2019. [Online]. Available: https://jakobgm.com/pdf/ntnu_reports/9-project-thesis.pdf.

[6] Y. H. Liu, “Feature extraction and image recognition with convolutional neural networks,” in *Journal of Physics: Conference Series*, IOP Publishing, vol. 1087, 2018, p. 062 032.

[7] K. Li, X. Wu, D. Z. Chen, and M. Sonka, “Optimal surface segmentation in volumetric images-a graph-theoretic approach,” *IEEE transactions on pattern analysis and machine intelligence*, vol. 28, no. 1, pp. 119–134, 2005.

[8] Q. Song, J. Bai, M. K. Garvin, M. Sonka, J. M. Buatti, and X. Wu, “Optimal multiple surface segmentation with shape and context priors,” *IEEE transactions on medical imaging*, vol. 32, no. 2, pp. 376–386, 2012.

[9] B. L. Decker, “World geodetic system 1984,” Defense Mapping Agency Aerospace Center St Louis Afs Mo, Tech. Rep., 1986.

[10] W. Commons. (2015). “File:la2-europe-utm-zones.png — wikimedia commons, the free media repository,” [Online]. Available: `https://commons.wikimedia.org/w/index.php?title=File:LA2-Europe-UTM-zones.png&oldid=146057239` (visited on 11/04/2019).

[11] J. P. Snyder, *Map projections–A working manual*. US Government Printing Office, 1987, vol. 1395.

[12] S. kartverk. (May 2018). “Ortofoto trondheim 2017,” [Online]. Available: `https://kartkatalog.geonorge.no/metadata/cd105955-6507-416f-86d2-6d95c1b74278` (visited on 11/07/2019).

[13] S. kartverk, *Produktspesifikasjon for ortofoto i norge*, 2003. [Online]. Available: `https://register.geonorge.no/data/documents/produktspesifikasjoner_Digitale%20ortofoto_v1_ortofoto-spesifikasjon-v1-2003_.pdf`.

[14] T. AS, *Rapport for laserskanning*, Comissioned by Trondheim kommune, 2017.

[15] Ø. Holmstad. (2011). “Fil:roof window at evanger.jpg,” [Online]. Available: `https://no.m.wikipedia.org/wiki/Fil:Roof_window_at_Evanger.jpg` (visited on 06/03/2020).

[16] A. Guttman, “R-trees: A dynamic index structure for spatial searching,” in *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, 1984, pp. 47–57.

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

[18] A. Zhang, Z. C. Lipton, M. Li, and A. J. Smola, *Dive into Deep Learning*, Release 0.7.0. [Online]. Available: `https://d2l.ai` (visited on 11/28/2019).

[19] R. C. Gonzalez, *Digital image processing*, eng, New York, 2018.

[20] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, `http://www.deeplearningbook.org`.

[21] G. Cybenko, “Approximations by superpositions of a sigmoidal function,” *Mathematics of Control, Signals and Systems*, vol. 2, pp. 183–192, 1989.

[22] M. Leshno, V. Y. Lin, A. Pinkus, and S. Schocken, “Multilayer feedforward networks with a nonpolynomial activation function can approximate any function,” *Neural networks*, vol. 6, no. 6, pp. 861–867, 1993.

[23] F. Rosenblatt, “The perceptron: A probabilistic model for information storage and organization in the brain.,” *Psychological review*, vol. 65, no. 6, p. 386, 1958.

[24] R. H. Hahnloser, R. Sarpeshkar, M. A. Mahowald, R. J. Douglas, and H. S. Seung, “Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit,” *Nature*, vol. 405, no. 6789, p. 947, 2000.

[25] Y. LeCun, Y. Bengio, and G. Hinton, “Deep learning,” *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[26] X. Glorot, A. Bordes, and Y. Bengio, “Deep sparse rectifier neural networks,” in *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, 2011, pp. 315–323.

[27] E. Kauderer-Abrams, *Quantifying translation-invariance in convolutional neural networks*, 2017. arXiv: `1801.01450 [cs.CV]`.

[28] S. Ioffe and C. Szegedy, *Batch normalization: Accelerating deep network training by reducing internal covariate shift*, 2015. arXiv: `1502.03167 [cs.LG]`.

[29] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov, “Improving neural networks by preventing co-adaptation of feature detectors,” *arXiv preprint arXiv:1207.0580*, 2012. arXiv: `1207.0580 [cs.NE]`.

[30] A. Labach, H. Salehinejad, and S. Valaee, “Survey of dropout methods for deep neural networks,” *arXiv preprint arXiv:1904.13310*, 2019. arXiv: `1904.13310 [cs.NE]`.

[31] T. DeVries and G. W. Taylor, “Improved regularization of convolutional neural networks with cutout,” 2017. arXiv: `1708.04552 [cs.CV]`.

[32] G. Huang, Y. Sun, Z. Liu, D. Sedra, and K. Q. Weinberger, “Deep networks with stochastic depth,” in *Computer Vision – ECCV 2016*, B. Leibe, J. Matas, N. Sebe, and M. Welling, Eds., Cham: Springer International Publishing, 2016, pp. 646–661, ISBN: 978-3-319-46493-0.

[33] H. Wu and X. Gu, “Towards dropout training for convolutional neural networks,” *Neural Networks*, vol. 71, pp. 1–10, 2015, ISSN: 0893-6080. DOI: `https://doi.org/10.1016/j.neunet.2015.07.007`. [Online]. Available: `http://www.sciencedirect.com/science/article/pii/S0893608015001446`.

[34] A. Garcia-Garcia, S. Orts-Escolano, S. Oprea, V. Villena-Martinez, and J. Garcia-Rodriguez, “A review on deep learning techniques applied to semantic segmentation,” 2017. arXiv: `1704.06857 [cs.CV]`.

[35] J. Bertels, T. Eelbode, M. Berman, D. Vandermeulen, F. Maes, R. Bisschops, and M. B. Blaschko, “Optimizing the dice score and jaccard index for medical image segmentation: Theory and practice,” in *Medical Image Computing and Computer Assisted Intervention – MICCAI 2019*, D. Shen, T. Liu, T. M. Peters, L. H. Staib, C. Essert, S. Zhou, P.-T. Yap, and A. Khan, Eds., Cham: Springer International Publishing, 2019, pp. 92–100, ISBN: 978-3-030-32245-8.

[36] G. Csurka, D. Larlus, F. Perronnin, and F. Meylan, “What is a good evaluation measure for semantic segmentation?.,” in *BMVC*, Citeseer, vol. 27, 2013, p. 2013.

[37] 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`.

[38] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds., Curran Associates, Inc., 2012, pp. 1097–1105. [Online]. Available: `http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf`.

[39] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2014. arXiv: `1409.1556 [cs.CV]`.

[40] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," 2014. arXiv: `1409.4842 [cs.CV]`.

[41] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2016.

[42] J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2015.

[43] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," in *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*, N. Navab, J. Hornegger, W. M. Wells, and A. F. Frangi, Eds., Cham: Springer International Publishing, 2015, pp. 234–241, ISBN: 978-3-319-24574-4.

[44] V. Badrinarayanan, A. Kendall, and R. Cipolla, "Segnet: A deep convolutional encoder-decoder architecture for image segmentation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 39, no. 12, pp. 2481–2495, Dec. 2017, ISSN: 1939-3539. DOI: `10.1109/TPAMI.2016.2644615`.

[45] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2014.

[46] R. Girshick, "Fast r-cnn," 2015. arXiv: `1504.08083 [cs.CV]`.

[47] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-CNN: Towards real-time object detection with region proposal networks," in *Advances in Neural Information Processing Systems 28*, C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, Eds., Curran Associates, Inc., 2015, pp. 91–99. [Online]. Available: `http://papers.nips.cc/paper/5638-faster-r-cnn-towards-real-time-object-detection-with-region-proposal-networks.pdf`.

[48] K. He, G. Gkioxari, P. Dollar, and R. Girshick, “Mask r-cnn,” in *The IEEE International Conference on Computer Vision (ICCV)*, Oct. 2017.

[49] S. Sabour, N. Frosst, and G. E. Hinton, “Dynamic routing between capsules,” in *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., Curran Associates, Inc., 2017, pp. 3856–3866. [Online]. Available: `http://papers.nips.cc/paper/6975-dynamic-routing-between-capsules.pdf`.

[50] R. LaLonde and U. Bagci, “Capsules for object segmentation,” 2018. arXiv: `1804.04241 [stat.ML]`.

[51] D. Kudinov, D. Hedges, and O. Maher, *Reconstructing 3d buildings from aerial lidar with ai: Details*. [Online]. Available: `https://medium.com/geoai/reconstructing-3d-buildings-from-aerial-lidar-with-ai-details-6a81cb3079c0` (visited on 06/23/2020).

[52] X. Wang, D. Fouhey, and A. Gupta, “Designing deep networks for surface normal estimation,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 539–547.

[53] X. Qi, R. Liao, Z. Liu, R. Urtasun, and J. Jia, “Geonet: Geometric neural network for joint depth and surface normal estimation,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 283–291.

[54] Y. Ben-Shabat, M. Lindenbaum, and A. Fischer, “Nesti-net: Normal estimation for unstructured 3d point clouds using convolutional neural networks,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019, pp. 10 112–10 120.

[55] D. E. Rumelhart, R. Durbin, R. Golden, and Y. Chauvin, “Backpropagation: The basic theory,” *Backpropagation: Theory, architectures and applications*, pp. 1–34, 1995.

[56] K. He, X. Zhang, S. Ren, and J. Sun, “Delving deep into rectifiers: Surpassing human-level performance on imagenet classification,” in *The IEEE International Conference on Computer Vision (ICCV)*, Dec. 2015.

[57] S. Ruder, *An overview of gradient descent optimization algorithms*, 2016. arXiv: `1609.04747 [cs.LG]`.

[58] R. Caruana, S. Lawrence, and C. L. Giles, “Overfitting in neural nets: Backpropagation, conjugate gradient, and early stopping,” in *Advances in neural information processing systems*, 2001, pp. 402–408.

[59] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, 2014. arXiv: `1412.6980 [cs.LG]`.

[60] R. Caruana, “Multitask learning,” *Machine learning*, vol. 28, no. 1, pp. 41–75, 1997.

[61] J. Sola and J. Sevilla, "Importance of input data normalization for the application of neural networks to complex industrial problems," *IEEE Transactions on Nuclear Science*, vol. 44, no. 3, pp. 1464–1468, Jun. 1997. DOI: `10.1109/23.589532`.

[62] C. Fiorio and J. Gustedt, "Two linear time union-find strategies for image processing," *Theoretical Computer Science*, vol. 154, no. 2, pp. 165–181, 1996.

[63] K. Wu, E. Otoo, and A. Shoshani, "Optimizing connected component labeling algorithms," in *Medical Imaging 2005: Image Processing*, International Society for Optics and Photonics, vol. 5747, 2005, pp. 1965–1976.

[64] M. Ester, H.-P. Kriegel, J. Sander, X. Xu, *et al.*, "A density-based algorithm for discovering clusters in large spatial databases with noise.," in *Kdd*, vol. 96, 1996, pp. 226–231.

[65] N. S. Altman, "An introduction to kernel and nearest-neighbor nonparametric regression," *The American Statistician*, vol. 46, no. 3, pp. 175–185, 1992.

[66] P. Bose, S. Cabello, O. Cheong, J. Gudmundsson, M. Van Kreveld, and B. Speckmann, "Area-preserving approximations of polygonal paths," *Journal of Discrete Algorithms*, vol. 4, no. 4, pp. 554–566, 2006.

[67] J. Gudmundsson, G. Narasimhan, and M. Smid, "Distance-preserving approximations of polygonal paths," *Computational Geometry*, vol. 36, no. 3, pp. 183–196, 2007.

[68] D. Z. Chen, O. Daescu, J. Hershberger, P. M. Kogge, N. Mi, and J. Snoeyink, "Polygonal path simplification with angle constraints," *Computational Geometry*, vol. 32, no. 3, pp. 173–187, 2005.

[69] M. Visvalingam and J. D. Whyatt, "Line generalisation by repeated elimination of points," *The cartographic journal*, vol. 30, no. 1, pp. 46–51, 1993.

[70] D. H. Douglas and T. K. Peucker, "Algorithms for the reduction of the number of points required to represent a digitized line or its caricature," *Cartographica: the international journal for geographic information and geovisualization*, vol. 10, no. 2, pp. 112–122, 1973.

[71] B. Farjad, A. Gupta, H. Sartipizadeh, and A. Cannon, "A novel approach for selecting extreme climate change scenarios for climate change impact studies," *Science of The Total Environment*, vol. 678, Apr. 2019. DOI: `10.1016/j.scitotenv.2019.04.218`.

[72] GDAL/OGR contributors, *GDAL/OGR geospatial data abstraction software library*, Open Source Geospatial Foundation, 2019. [Online]. Available: `https://gdal.org`.

[73] S. Gillies *et al.*, *Rasterio: Geospatial raster i/o for Python programmers*, Mapbox, 2013–. [Online]. Available: `https://github.com/mapbox/rasterio`.

[74] S. Gillies *et al.*, *Fiona is ogr's neat, nimble, no-nonsense api*, Toblerity, 2011–. [Online]. Available: `https://github.com/Toblerity/Fiona`.

[75] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 265–283. [Online]. Available: `https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf`.

[76] J. Häme. (Jul. 3, 2019). "Shapefile vs. GeoJSON vs. GeoPackage," Terramonitor Feed, [Online]. Available: `https://feed.terramonitor.com/shapefile-vs-geopackage-vs-geojson/` (visited on 08/29/2019).

[77] E. Rouault. (2015). "File:gdalvrt.xsd — xml schema for gdal vrt files.," [Online]. Available: `https://raw.githubusercontent.com/OSGeo/gdal/master/gdal/data/gdalvrt.xsd` (visited on 11/07/2019).

Appendix A

GIS pre-processing

A.1 Mapping between coordinate systems

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

Code listing A.1: Coordinate system transformation using gdaltransform.

```
$ 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.2 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:

Code listing A.2: Zero-buffering vector dataset using ogr2ogr.

```
$ 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 [76]. `ogr2ogr` supports file conversions between most common vector file formats, which makes the data pipeline generalizable data sourced from different providers.

A.3 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.

Code listing A.3: Virtual merger of raster tiles using `gdalbuildvrt`.

```
$ 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,

Code listing A.4: Merging overlapping raster datasets using `gdalbuildvrt`.

```
$ 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 [77] 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:

Code listing A.5: Setting color interpretation of multi-channel VRT rasters.

```xml
<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.

Code listing A.6: Specifying source bands for multi-channel VRT rasters.

```xml
<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>
```

Jakob Gerhard Martinussen

Three-dimensional Roof Surface Geometry Inference Using Remote Sensing Data

NTNU

Norwegian University of
Science and Technology