Technische Universität Berlin

INSTITUT
WERKZEUGMASCHINEN UND FABRIKBETRIEB
TECHNISCHE UNIVERSITÄT BERLIN

Fraunhofer IPK

Master Thesis

Matthias Weber

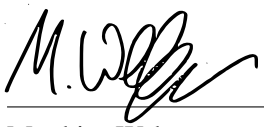
Edge detection in three-dimensional point clouds using a neural network performing generalised convolution

STUDENT NUMBER: 326 836
SUPERVISOR: Oliver Krumpek, M.Sc.
EXAMINER: Prof. Dr.-Ing. Jörg Krüger

Berlin, March 29, 2024

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ohne unerlaubte fremde Hilfe und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe.

Berlin, den 29. März 2024



Matthias Weber

Abstract

Point clouds are the most fundamental data type when dealing with 3D data. It is produced naturally by many different scanners, from LIDAR to structured light and TOF detectors. Edge detection is a fundamental technology that deals with the complexity of the data and identifies regions of interest for further processing (e.g., object detection and identification).

This thesis aimed to assess the effectiveness of the method proposed by Savchenkov et al. for generalising Convolutional Neural Networks (CNN) from 2D to 3D data. The aim was to determine if this approach could be used for edge detection in 3D point clouds and to identify a suitable model and training dataset. A Generalised Convolutional Edge Detector, and a Normal Clustering and Angular Gap approach were tested. The neural network was trained on two datasets and showed good classification results, indicating that the approach is well-suited for the task at hand. The neural network outperforms classical approaches due to its usability, as it eliminates the need for parameter optimisation. However, it still requires suitable training data, time, and computation resources.

Nonetheless, the results were deficient when applied to real-world scanner data taken from the scan of a real-world sample object using a TOF-sensor. This leads to the conclusion that the neural network is strongly influenced by the quality and density of the input clouds, necessitating future careful considerations of data preprocessing and augmentation strategies. Additionally, edges are ambiguous and depend on the use case and the expectation, so clearer and quantifiable definitions based on the intended use case are needed and will improve the results further.

Punktwolken sind der grundlegendste Datentyp im Umgang mit 3D-Daten. Sie werden von vielen verschiedenen Scannern erzeugt, von LIDAR bis zu strukturiertem Licht und TOF-Detektoren. Die Kantendetektion ist eine grundlegende Technologie, um der Komplexität der Daten Herr zu werden und Regionen zu identifizieren, die für die weitere Verarbeitung (z. B. Objekterkennung und -identifizierung) von Interesse sind.

Das Ziel dieser Arbeit war es, die Effektivität der von Savchenkov et al. vorgeschlagenen Methode zur Verallgemeinerung von CNN von 2D- auf 3D-Daten zu bewerten, insbesondere für die Kantendetektion in 3D-Punktwolken. Ein geeignetes Modell und Trainingsdatensatz wurden identifiziert. Der Ansatz wurde neben zwei anderen Methoden implementiert - einem modernen Ansatz für Normal Clustering und einem aktuellen Beispiel für Angular Gap. Das neuronale Netz wurde auf zwei Datensätzen trainiert und zeigte gute Klassifizierungsergebnisse, was auf seine Eignung für die Aufgabe hinweist. Im Vergleich zu klassischen Ansätzen erfordert das neuronale Netz keine Parameteroptimierung, ist jedoch ressourcen- und zeitintensiv in Bezug auf die Trainingsdaten.

Die Ergebnisse waren jedoch unzureichend, wenn sie auf reale Scannerdaten angewandt wurden, die vom Scan eines realen Musterobjekts mit einem TOF-Sensor stammen. Dies führt zu der Schlussfolgerung, dass das neuronale Netz stark von der Qualität und Dichte der Trainingsdaten beeinflusst wird, was in Zukunft sorgfältige Überlegungen zur Vorverarbeitung bzw. Anpassung der Daten erfordert. Außerdem sind Kanten mehrdeutig und hängen vom Anwendungsfall und den Erwartungen ab, weshalb klarere und quantifizierbare Definitionen auf der Grundlage des beabsichtigten Anwendungsfalls erforderlich sind um die Ergebnisse weiter zu verbessern.

Contents

1 Introduction

Vision is the primary way humans detect their surroundings. Natural scenes are projected through the lenses of the human eyes and onto the respective retinas. The retinas, in turn, act as two-dimensional detectors for complex intensity functions of illuminance across their surface area. For them to extract information from these raw intensity values, the brain needs to process and describe the data. This initial description should ideally contain primitive elements that are both complete (i.e., representing the full information in the image) and meaningful (i.e., capturing significant properties of the three-dimensional surfaces surrounding the viewer) [2]. Information is thus extracted through the segmentation of the projected scene. The underlying segmentation can either be based on finding homogeneity (i.e., finding homogeneous regions) or by finding discontinuity (i.e., finding the transitions between those homogeneous regions) within the intensity function of the image projected onto the retina [3]. To capture information, humans effortlessly extract a multitude of information about their surroundings (e.g., surface qualities/ finish, surface pigmentation, shadows, occlusion of one object by another, etc.) [4]. To do so, a multi-level segmentation process is carried out. Extensive research in cognitive science indicates that human visual perception is largely based on simple geometric objects [3]. These simple geometric objects need to be localised and bounded. Object boundaries and changes in surface orientation or material properties, in turn, are some of the most important discontinuities in the scene, all represented by a physical edge [2]. Edge detection, therefore, plays an important role in the human perception process and is elementary for our understanding of the physical world around us.

Based on the understanding of the visionary perception of humans, edge detection has been an integral part of computer vision since the very early days of the field. Edges frequently occur at the boundary of two regions within an image, providing a first overview of the parts of the image and the scene captured within. Edge detection is thus frequently used as a first information extraction algorithm to provide a solid basis for consecutive processing steps like segmentation, object identification, or face detection.

Similar to the process in humans, the computational edge detection process also focuses on detecting relatively abrupt spatial changes in the intensity function of the image. Change can be found using an approach based on detecting large gradients within the image (ideally, the large gradients come in the shape of discontinuities due to the infinite sharpness of the edge in the form of a step function). Thus, the two-dimensional edges have both a magnitude and a direction [5].

Some of the earliest examples of gradient-based edge detectors include the Sobel-Feldman operator in 1968, the Prewitt operator in 1970, and the Canny algorithm in 1986 [6], [7], [8]. The edge detectors mentioned are based on gradient detection in the first derivative of the image intensity function, which produces thick edges (also see Section 3.6.2). Ideally, an edge is represented by an infinitely small boundary. Therefore, the Laplacian approach searches for local extrema within the first derivative and tries to find places where the change in the gradient is highest. The more basic gradient-based algorithms are sensitive to noise. More elaborate algorithms, like the Canny algorithm, for example, can overcome some of these shortcomings, but at the cost of increased computation time due to their complexity [9].

In recent years, advancements in artificial intelligence have produced novel algorithms for edge detection in 2D images. The use of Neural Networks, which learn the required filters for edge detection based on labelled input data, have the great advantage of not requiring fine-tuning

of a parameter set.

Where edge detection in 2D images has been a research topic for quite some time, detecting edges in 3D point cloud data is a rather new endeavour. In 2D cameras, the data acquisition works fundamentally similar to the human eye; data acquisition in 3D, on the other hand, comprises two main detection methods. In laser-scanning-based TOF approaches, triangulation is used to deduce depth of field information directly from the acquired data (see Section 3.1.3). In stereo vision, on the other hand, two 2D cameras mounted at a set distance from each other are used to compute depth information by comparing scene information from two vantage points - similar to human depth perception (see Section 3.1.1) [10].

For a long time, 3D computer vision was a rather costly task since specialised hard- and software was unavailable or expensive. Only in the last decade have widely available and cheap solutions in 3D camera systems revolutionised data acquisition and shifted research focus into the field. Similar to the 2D case, edge detection in point clouds plays an important role in understanding the data and deducing information from it. Use cases range from quality assurance/ metrological applications, object identification and classification, reverse engineering and machining path planning, computer-assisted restoration of fragmented relics, over Simultaneous Localisation and Mapping (SLAM), and manipulation tasks in autonomous robotics to manipulation of unknown or diverse objects with robots [11], [12], [13]. Compared to 2D data, point cloud data has one major drawback in the fact that data in 3D data structures (like point clouds) has no intrinsically encoded spatial information: Within the point cloud, a point and its direct successor and predecessor have no direct relationship. Information about spatial vicinity is a costly computation that must be performed whenever the data changes. Classically, edge detection is approached using statistical and geometric properties or by estimating normals on the previously computed neighbourhoods around a point in the point cloud [11].

Ni et al. use a statistical and geometric approach evaluating the angular gap of connecting vectors between the candidate edge point and its neighbours in a statistically fitted Random Sample Consensus (RANSAC) plane through the neighbourhood (also see Section 4.2.2) [14].

Indirectly using normals Bazazian et al. have found an elegant and fast way to use an eigenvalue analysis of the covariance matrix of the neighbourhood around an edge candidate to evaluate an edge point candidate using surface variation (also see Section 4.2.1) [11].

Since data within point cloud data structures is generally unsorted, approaches using convolution have had a difficult standing. Only recently have Savchenkov et al. proposed an approach that uses a Generalised Convolutional Neural Network (gCNN) to tackle 3D data in unsorted point clouds and laid the foundation for the use of classical Convolutional Neural Networks in 3D-edge detection (see Section 4.3) [1]. However, compared to established 2D edge detection algorithms, 3D edge detection is still in its infancy, with most algorithms being comparably slow and deficient.

2 Problem Statement

This thesis investigates how neural networks can be used to automatically extract edge information directly from point clouds using the generalised convolution approach. After literature research, two classical (i.e., not based on neural networks) edge detection algorithms are implemented as benchmarks. Subsequently, an approach based on the generalised convolution approach is developed, implemented in a suitable programming language (e.g., Python) and trained based on a befitting training dataset. Finally, all three algorithms are to be applied to the same point cloud dataset, and the results are compared.

The following research questions shall be answered in this thesis:

- Is generalised convolution a suitable approach to edge detection in point clouds?
- What does an implementation of the principle of generalised convolution for edge detection in three-dimensional point clouds look like, and which model structure yields good results for edge detection?
- What does a suitable training data set look like?
- What challenges arise when the developed algorithm is applied to real scan data from a 3D scanner (e.g., 3D laser scanner)?
- What are the opportunities and limitations of the various algorithms implemented?

3 Background Research

This section lays the theoretical foundation for the work with edge detectors. It covers the acquisition of 3D data and the way in which that data is represented. To bridge the gap between two-dimensional data and three-dimensional data nearest neighbour search plays a crucial role. After a brief excursion into classification theory, the theoretical foundation for neural networks is laid out, only to be rounded off by an excursion into state-of-the-art edge detection in point clouds.

3.1 3D sensing and data acquisition

The process of acquiring 3D data has been of interest since the pioneer days of photography. From the middle of the 19th century, when Laussedat and Meydenbauer developed the first photogrammetric methods for reconstructing buildings from photographic images, the methods and topics have always attracted active research [15].

Generally, 3D data acquisition can be divided into active and passive techniques. On the one hand, passive techniques do not use a specific light source to enhance data acquisition [16]. The basic principle in very early methods was based on the reconstruction of the three-dimensional scene from the geometry of the image formation process and the underlying geometric perspective projection for a three-dimensional scene onto a two-dimensional image plane [17]. This approach is also known as stereo vision; see Section 3.1.1. Other methods not further discussed here include structure from motion, shape from shading, shape from texture, and shape from focus.

On the other hand, active techniques use a controlled source of active light emission. These emissions can be a projected light pattern, or a scanning laser [16]. Active range-finding techniques use the elapsed time until the return of a sent-out (light) signal in combination with the speed of light to compute depth information through triangulation, see Section 3.1.2.

More recent active and passive approaches have reached beyond the use of multiple images but use a wide variety of alternative visual cues, for example, shading and focus, as well as merging multiple range or depth images [18].

3.1.1 Stereo vision

Stereopsis, stereo image analysis or stereo vision is the term given to the method of acquiring images of the same scene from different perspectives (i.e., different positions and viewing directions) and thus reconstructing the three-dimensional scene - much like humans gather depth-of-field information from two distinct images provided by their two eyes [17]. Humans gain a strong sense of depth by exploiting the difference (or disparity) between the two images acquired by our two eyes. The underlying process can be broken down into two steps: a fusion step in which features observed in the presented images are matched and fused, and a reconstruction step in which the actual three-dimensional scene model is obtained [19]. Once the correct fusion has been established, i.e., the corresponding points in the two images have been found, the scene reconstruction is relatively simple: Given two cameras with the same focal length f, placed at a distance d between the centres of their lenses O_1 and O_2, as shown in Figure 3.1. Let the angle between the two optic axes be $2 \cdot \theta$. Two coordinate systems $X_1 - Y_1 - Z_1$ and $X_2 - Y_2 - Z_2$ are defined for those cameras, respectively, and a third coordinate system $X - Y - Z$ shall be placed so that the Z axis bisects the angle between the Z_1 and Z_2 axis as shown in Figure 3.1 [20].

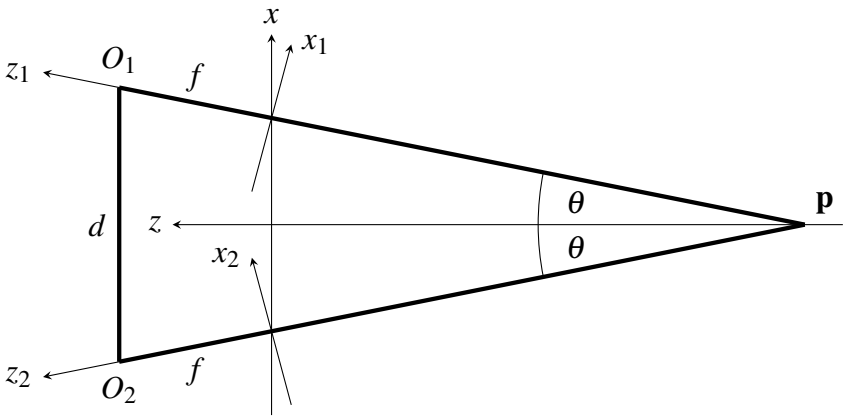


Figure 3.1: Geometry of stereo vision: Two cameras with the same focal length f, placed at distance d between the centres of their respective lenses O_1 and O_2.

According to Shirai the relations between these coordinate systems are given as follows [20]:

$$\begin{pmatrix} X_1 \\ Y_1 \\ Z_1 \end{pmatrix} = \begin{pmatrix} cos(\theta) & 0 & sin(\theta) \\ 0 & 1 & 0 \\ -sin(\theta) & 0 & cos(\theta) \end{pmatrix} \begin{pmatrix} X + d/2 - f \cdot sin(\theta) \\ Y \\ Z \end{pmatrix} \tag{3.1}$$

$$\begin{pmatrix} X_2 \\ Y_2 \\ Z_2 \end{pmatrix} = \begin{pmatrix} cos(\theta) & 0 & -sin(\theta) \\ 0 & 1 & 0 \\ sin(\theta) & 0 & cos(\theta) \end{pmatrix} \begin{pmatrix} X - d/2 + f \cdot sin(\theta) \\ Y \\ Z \end{pmatrix}. \tag{3.2}$$

Suppose a point $\mathbf{p} \in \mathbb{R}^3$ at (X,Y,Z) is observed by the two cameras on their respective image planes and their locations (x_1,y_1) and (x_2,y_2) respectively. Their relationship for $i = 1,2$ is given by [20]:

$$\begin{pmatrix} x_i \\ y_i \end{pmatrix} = \begin{pmatrix} f \cdot X_i / (f - Z_i) \\ f \cdot Y_i / (f - Z_i) \end{pmatrix}. \tag{3.3}$$

When two corresponding points at (x_1,y_1) and (x_2,y_2) in the corresponding image planes have been established by the fusion step, the three-dimensional position of the original point (X,Y,Z) is derived from Equations (3.1), (3.2), and (3.3). Note that there are ten equations but only nine variables. This is due to a constraint on the relation between two points in the image planes [20]. Given a point in one image plane, the projection on the other respective image plane is a line segment - the epipolar line segment l', see Figure 3.2. The resulting line segment is bounded by the projected of the original image viewing ray at infinity at the one end and by the epipole e' at the other end [18]. The epipole e' corresponds to the original camera centre O projection into the second camera. As shown previously, the scene - sometimes called preimage - is reconstructed through basic ray geometry at the intersection of the rays going from the corresponding points and through the associated pupil centres/ pinholes. Thus, the simple case is very straightforward (see Figure 3.3 left). However, the reconstruction process can become ambiguous when dealing with more realistic scenes that contain multiple candidates for each point, such as modern images consisting of millions of pixels and tens of thousands of image features like edge elements. This can result in incorrect reconstructions being produced. (see Figure 3.3 right, little grey discs) [19]. Thus, the most difficult problem of stereo vision is establishing correct correspondences between

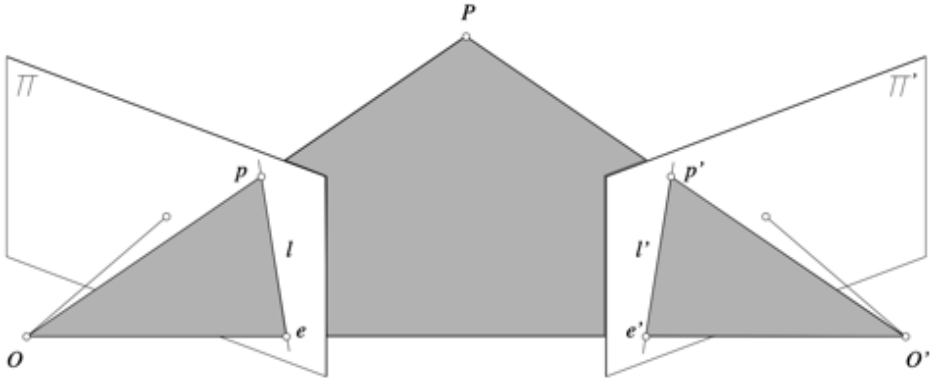


Figure 3.2: Epipolar geometry [19]

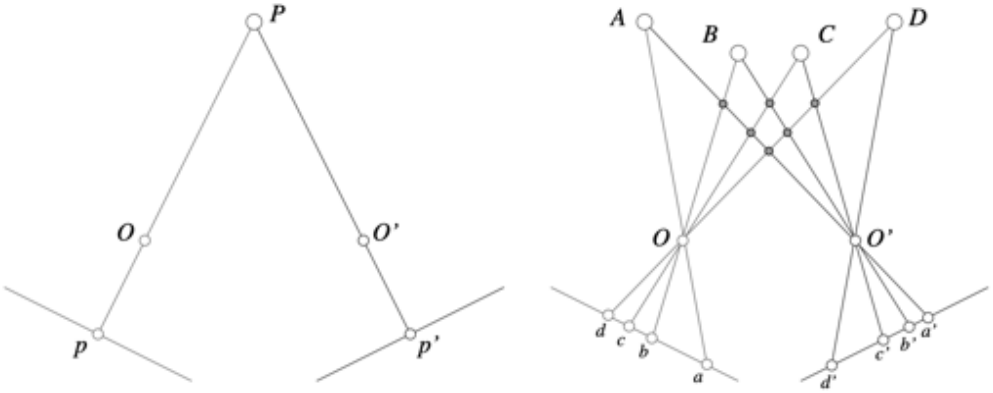


Figure 3.3: The problem of binocular fusion: Simple problem with a single point and no ambiguity in the stereo reconstruction on the **left**, and the more realistic scene on the **right** in which any of the initial points on the left could *a priori* match any of the points on the right, with incorrect correspondences indicated with little grey discs [19].

features and avoiding erroneous depth measurements.

As previously shown, only features on the corresponding epipolar line need to be considered when taking a feature from one image, not all existing features have to be considered as candidates. Through this approach, the possible correspondences can already be narrowed down.

A wide variety of stereo fusion algorithms have been proposed. Early algorithms were feature-based. Various filters were used to extract areas of interest in one of the images. These filters were usually based on either interest operators or edge detectors. The algorithm then proceeded to search for corresponding locations in both images as candidates for matching. The results were few matches with high certainty - thus, they are generally called sparse correspondence algorithms. At the time of their application, sparseness was partly desired due to limitations in computing power. This makes it understandable why it has taken considerable time for three-dimensional sensing to enter the general research focus. Today, due to the availability of higher computing power and the fact that for rendering or modelling, a much larger number of correspondences are required, the focus has shifted to dense correspondence algorithms [18].

3.1.2 Structured light approaches

As shown in the previous section, the biggest problem of stereo vision is establishing correspondence between the two recorded images. Triangulation-based active range-finding techniques try to avoid this problem by projecting some light pattern (structured light) onto the scene (see Figure 3.5). Active-ranging systems work in a very similar way to stereopsis systems. For active range finding, one of the cameras is replaced by a controlled light source to avoid the correspondence problem; see Figure 3.4. In this sense, a laser paired with small moveable mirrors could scan an object systematically. As was shown in Section 3.1.1, the point of the laser beam striking the object of interest in a particular spot can be found at the intersection of the beam with the projection ray joining the spot to its image [19]. Due to the laser beam's much higher brightness compared to the surrounding area in the scene, the correspondence can quickly be established [19]. The mechanical design of the mirror setup and the scanning time can be reduced by transforming the laser beam through a cylindrical lens from a spot to a plane of light; also see Figure 3.4. Note further that this system does not introduce matching ambiguities (compare Figure 3.3) since only a single ray is considered at every step. The corresponding image pixel can be retrieved as a unique intersection of the corresponding projection ray with the plane of light [19].

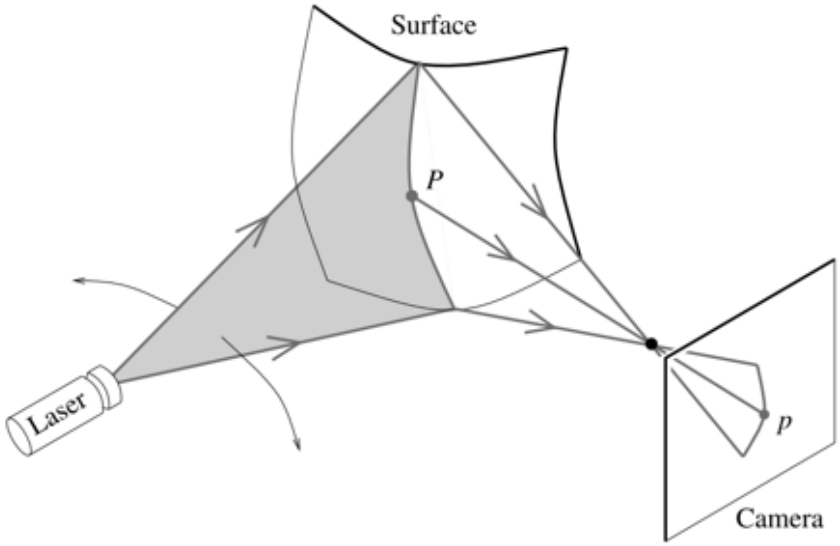


Figure 3.4: Active range finder: A plane of light is used to scan the surface of an object [19].

One of the main drawbacks of the technique is the time needed for scanning since the time it takes to scan an object is proportional to the number of depth planes used [18]. Proposed improvements include using multiple cameras for additional measurement accuracy and extending the laser beam to two-dimensional (possibly time-coded) light patterns (thus the name structured light) for improved acquisition speeds [19]. Even under the use of described light patterns, acquisition speed is one of the main drawbacks of the technique; other drawbacks include missing data points due to perspective (laser beam/ pattern hidden from camera view) and missing data due to specularities and reflections [19].

Figure 3.5: Real-time dense 3D face reconstruction using structured light: Stripe patterns (time-coded) projected onto a face in consecutive frames of a video setup. Images of one of the two stereo cameras taken at different times on the **left**, resulting 3D surface model (depth map visualised as a shaded rendering) on the **right** [21].

3.1.3 Time-of-flight approaches

Another main approach to active range finding is to use the time it takes a signal on its round trip to return to the emitter. The distance between the emitter and the reflecting object can easily be computed based on the speed of light. The previously described problem of missing data due to perspective is eliminated in this approach by aligning transmitter and receiver coaxially [19].

Three main classes of TOF laser range scanners can be distinguished: *Pulse time delay* range finders send out laser pulses and directly measure the time of flight of a single laser pulse, *Amplitude Modulation (AM) phase-shift* range finders only indirectly measure the TOF through the measurement of the proportionally to the TOF changed phase difference between an amplitude-modulated laser beam emitted and the reflected beam. Finally, FM beat sensors base the TOF on a frequency shift between a frequency-modulated laser beam and its reflection [19]. Compared to their triangulation counterparts, the main advantage of TOF-based systems is their much greater operating range. Therefore, in many outdoor applications, including autonomous robotics and autonomous driving, TOF sensors are widely established [19].

3.2 Data representations

3.2.1 2D Images

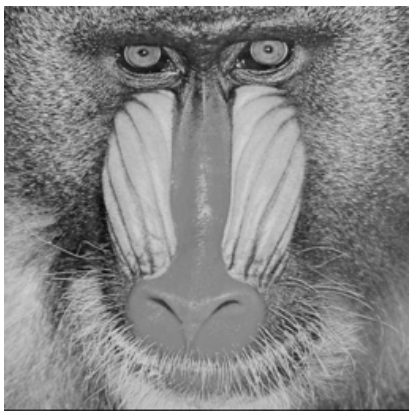
The projection of a scene onto the sensor of a digital camera is a two-dimensional, time-dependent and continuous distribution of energy introduced by the incoming light. According to Burger and Burge, three main steps have to be undertaken to create a digital image of the said scene [22]:

1. Spatial sampling of the light distribution across the sensor,
2. sampling of the distribution over time, and
3. quantification of the sample values into a finite number of possible values.

The first step of the sequence is determined by the geometrical distribution of sensor elements across the sensor plane. The second step is determined by when the sensor readings are fetched. Processing the data from the sensor readings is mainly dependent on how the camera equipment electronics handle it [22].

Application of the described process results in a regular two-dimensional matrix - more formally, according to Burger and Burge, the digital image I is therefore a two-dimensional function of integer coordinates $\mathbb{N} \times \mathbb{N}$ on a set (i.e., an interval) of image values $\mathbb{P}$:

$$I(u,v) \in \mathbb{P} \quad \text{and} \quad u,v \in \mathbb{N}. \tag{3.4}$$



149	57	50	92	140	94	61	33	72	77	…	182
121	104	40	70	91	55	65	47	97	60	…	131
78	118	45	47	102	61	60	47	115	110	…	82
74	141	99	49	85	126	53	47	72	94	…	73
88	128	136	49	79	88	76	49	57	127	…	89
84	78	164	60	74	98	43	45	73	99	…	190
43	52	139	134	74	127	54	63	67	111	…	168
33	51	118	158	88	120	47	58	102	169	…	145
56	48	56	121	154	131	60	34	66	138	…	126
38	32	51	67	153	82	100	60	50	158	…	141
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮		⋮
11	12	14	15	16	16	19	22	19	19	…	141

Figure 3.6: Continuous light distribution $F(x,y)$ on the **left** [23], and the resulting discretised digital image $I(u,v)$ with each value representing the corresponding grey-scale value of a single pixel on the **right**.

2D images can therefore be represented by a single matrix in the case of greyscale imagery (see Figure 3.6) or multiple matrices in the case of coloured images (e.g., three, one for each red, green and blue values in RGB).

3.2.2 Voxel or occupancy grids

Theoretically, the 2D approach could also be applied to three-dimensional data. The scene is divided into a three-dimensional grid, and data is stored in the appropriate elements (called voxels) (see Figure 3.7 in the middle). Similar to the advantages in the 2D case, the voxel grid grants access in constant time and yields a volumetric representation of the scanned scene. Most widely used vision scanning applications only scan object surfaces, resulting in sparsely populated real-world scenes. Thus, this approach of discretizing three-dimensional space leads to very little information being stored in considerable amounts of data.

This is why, in practice, very little three-dimensional application data is stored in so-called occupancy grids, and other representations are far more popular.

3.2.3 Point clouds

As was shown in previous sections, the scanning of three-dimensional objects produces a large amount of points lying on the surface of an object. This collection of points as a representation of the object surface is generally called a point cloud [24] (see Figure 3.7 on the left).

Thus, the point cloud dataset is an unsorted collection of individual tuples, each representing a single 3D point, where $\mathbf{p} = (x,y,z) \in \mathbb{R}^3$ can be seen as a 3D vector, let $\mathbf{P}$ be the corresponding

point cloud interpreted as a $3 \times N$ matrix, where N is the total number of points in the cloud [25]:

$$\mathbf{p} = \begin{bmatrix} x & y & z \end{bmatrix}^T \tag{3.5}$$

$$\mathbf{P} = \begin{bmatrix} \mathbf{p}_1 & \dots & \mathbf{p}_N \end{bmatrix} \tag{3.6}$$

This list can hypothetically be appended without limits, making the theoretical memory use unbounded. The most significant advantage of point clouds is that the data does not have to be discretised. Each point within the cloud can be represented with a very high accuracy in its coordinates. This also makes the represented area unlimited since the coordinates are also theoretically unbounded.

3.2.4 Triangle meshes

Triangle meshes consist of triangles representing an object's surface; see Figure 3.7 on the right. The object's surface is expressed in a piecewise linear fashion segmented along each triangle through its barycentric parametrisation [26]. Let $\mathbf{p}$ denote any point in the interior of the triangle $[a,b,c]$. According to Botsch et al., a unique representation can be established through the barycentric combination of the corner points [26]:

$$\mathbf{p} = \alpha \cdot a + \beta \cdot b + \gamma \cdot c, \tag{3.7}$$

with

$$\alpha + \beta + \gamma = 1, \quad \alpha, \beta, \gamma \geq 0. \tag{3.8}$$

Based on this a linear mapping $\mathbf{f} : \mathbb{R}^2 \to \mathbb{R}^3$ can be defined using an arbitrary triangle $[u,v,w]$ from the parameter domain:

$$\alpha \cdot u + \beta \cdot v + \gamma \cdot w \quad \mapsto \quad \alpha \cdot a + \beta \cdot b + \gamma \cdot c. \tag{3.9}$$

Therefore, an entire triangle mesh can be based upon a single 2D position for each vertex, resulting in the desired global parameterisation [26]. Triangle meshes were popular in the 90s to reconstruct surfaces from point clouds. According to Linsen, various algorithms were introduced based on spatial subdivision, distance functions, warping and incremental surface increase to compute the triangle mesh from the underlying point cloud [24].

3.3 Nearest neighbour search

Whether a single point lies on an edge is determined by its neighbourhood. The edge property is a property of the point to its neighbourhood. The definition of said neighbourhood around a given point is, therefore, integral for the correct classification. Generally speaking, the nearest neighbour search is used on a given set of points $\mathbf{P} = \mathbf{p_1}, \mathbf{p_2}, \dots, \mathbf{p_n}$ in a metric space $\mathbf{X}$. The goal of any nearest neighbour search algorithm is, given a new query point $\mathbf{q} \in \mathbf{X}$, to find the point in $\mathbf{P}$ that is closest to $\mathbf{q}$ and to do so quickly.

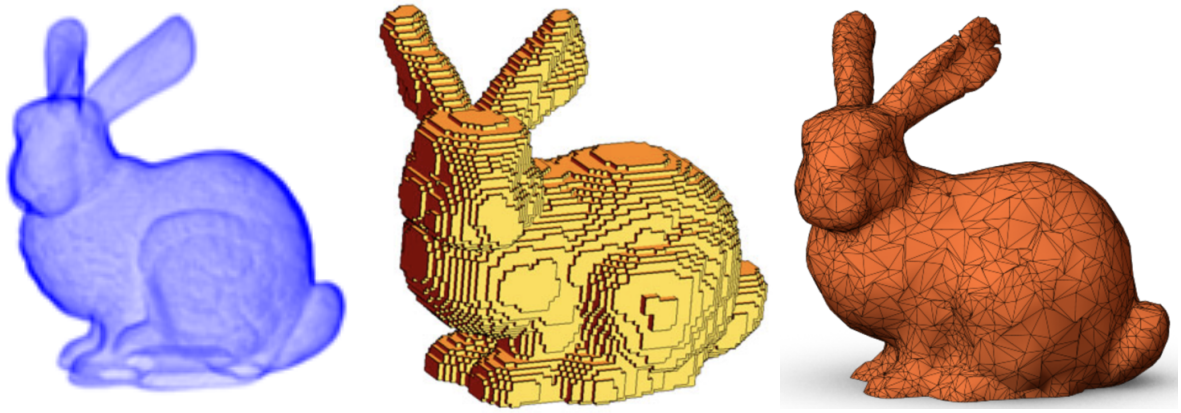


Figure 3.7: The iconic Stanford bunny shown in a point cloud representation on the **left**, Voxel grid representation in the **middle** [27], and a triangle mesh representation on the **right** [28].

3.3.1 Nearest neighbour search in 2D data

In two-dimensional images, spatial information is intrinsic to the data structure. Consider any given two-dimensional greyscale image represented by a matrix $\mathbf{I} \in \mathbb{R}^{mxn}$, $m,n \in \mathbb{N}^+$ of greyscale values and an arbitrary point within that image $\mathbf{p}(x,y) \in \mathbb{R}^2$, $x \in [0,m]$, $y \in [0,n]$ as shown in Equation (3.10).

$$\mathbf{I} = (i_{j,k}) = \begin{pmatrix} i_{1,1} & i_{1,2} & & & \cdots & & & i_{1,(n-1)} & i_{1,n} \\ i_{2,1} & i_{2,2} & & & & & & i_{2,(n-1)} & i_{2,n} \\ & & \ddots & & & & & & \\ & & & i_{(x-1),(y-1)} & i_{(x-1),y} & i_{(x-1),(y+1)} & & & \\ \vdots & & & i_{x,(y-1)} & i_{x,y} & i_{x,(y+1)} & & & \vdots \\ & & & i_{(x+1),(y-1)} & i_{(x+1),y} & i_{(x+1),(y+1)} & & & \\ & & & & & & \ddots & & \\ i_{m,1} & i_{m,2} & & & \cdots & & & & i_{m,n} \end{pmatrix}, \quad (3.10)$$

where m is the height of the image, n the width of the image and $j \in [0,m]$, $k \in [0,n]$. It is obvious that the neighbours of the point $\mathbf{p}$ are easily found through an increase or decrease of the spatial coordinates of the original point itself - where for example, its direct neighbour to the right can be found by keeping the x-coordinate constant and incrementing the y-coordinate once, yielding $i_{x,(y+1)}$. Conclusively, two-dimensional images are represented by dense matrices that contain spatial information and are thus, in a sense, sorted.

3.3.2 Nearest neighbour search in 3D data

Contrarily to the case in 2D data, spatial information is not generally embedded within the data structure in the 3D-case (also see Section 3.2). For any given query point, the nearest neighbours have to be computed. Earlier, the nearest neighbour search in three-dimensional data was (particularly in higher dimensions) performed as a linear search. Linear search is, however, a costly

algorithm that can take a lot of computation time to accomplish. According to Muja and Lowe, this has sparked a general interest in approximate nearest neighbour search, which can be orders of magnitude faster than optimal search while still providing near-optimal accuracy. However, it is important to remember that non-optimal neighbours are sometimes returned [29]. Many such algorithms had been suggested whose performance varied highly depending on the general properties of the dataset they were used on (e.g., dimensionality, correlations, clustering characteristics and size). Muja and Lowe introduced Fast Library for Approximate Nearest Neighbours (FLANN), which includes an automatic approach to select and configure the most suitable algorithm for any given dataset, which allows the best algorithm and parameter settings to be determined automatically [29]. For most datasets Muja and Lowe found that algorithms based on one of two strategies, namely the randomised kd-tree algorithm or the hierarchical k-means tree algorithm, proved to be most effective based on their precision.

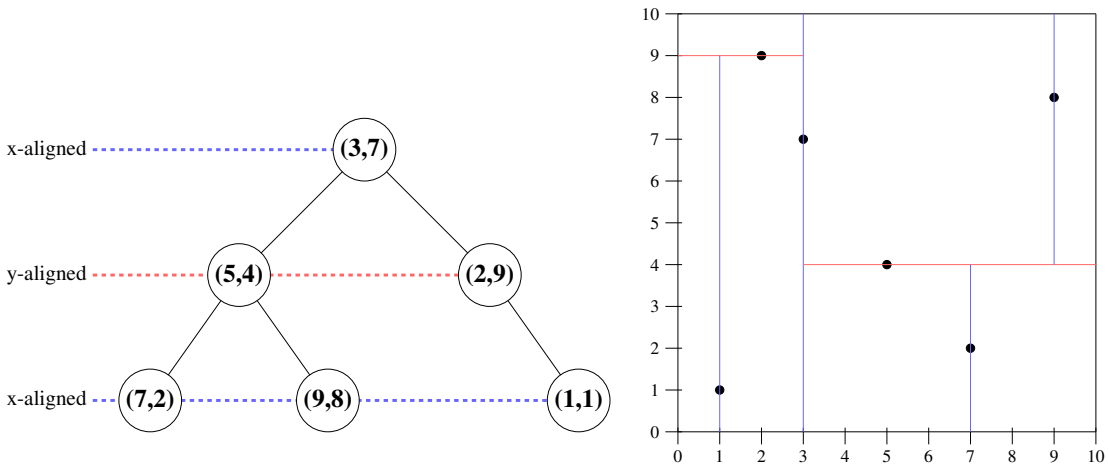


Figure 3.8: Example of a 2D kd-tree for $\{(3,7);(5,4);(2,9);(7,2);(9,8);(1,1)\}$ in the 2D-plane on the **left** and the corresponding decomposition on the **right**.

The kd-tree algorithm is a concept first introduced by Bentley in 1975 to organise points in a k-dimensional space using a binary tree data structure [30]. This data structure was then used by Friedman et al. for a constructing scheme that iteratively divides space along a feature dimension by the median of data points within the current subspace. Therefore, each node within the tree represents a point in k-dimensional space and can be considered a hyperplane dividing the data points within the current sub-tree into two distinct sets [31]. The great benefits of such a data structure lie in its ease of construction and ease of use for later classification tasks of a newly introduced query point. For constructing the tree, the dimension with the highest variance of the dataset is chosen. The median of all points along this dimension is selected as the tree's root, splitting the dataset into two sub-trees. Again, the dimension with the highest variance is chosen from the remaining dimensions to separate the two sub-trees along their median-hyperplane again - splitting the tree into further sub-trees. This process can be repeated any number of times, resulting in a higher fragmentation of the initial space with each step (see Figure 3.8). A query point's classification is obtained by going through the tree layer by layer and determining which sub-tree most likely contains it by comparing the values of the query point and the node along the current layer's splitting dimension. However, this classification scheme only yields approximate

results since distances are only considered along the main dimensions. A point could "diagonally" be closer to a point in an adjacent bin (i.e., a leaf). Modifications for the randomised kd-tree algorithms include the use of multiple randomised kd-trees, where the splitting dimension is not solely chosen along the dimension with the highest variance but along a dimension randomly selected from the D dimensions with the highest variance. This is done in multiple trees in parallel to decrease overall computation time [29].

For the hierarchical k-means tree algorithm, the main change to the original kd-tree algorithm lies in the splitting of the data points at each level into K distinct regions using a k-means clustering instead of just two regions as originally proposed [29].

3.4 Classification

Edge detection is a binary classification problem in which a classifier aims to assign an observation (i.e., a query point) to a category (i.e., edge or non-edge point). For this purpose, we must first define criteria based on which the classifier can root its decision, or - as in the case of neural-networks-based classifiers (see Section 3.5) - the classifier aims to learn the criteria from a training dataset provided. Signal detection theory provides the tools for evaluating the performance of a binary classifier when distinguishing between the two possible states: the presence of a signal (i.e., an edge) and the absence of a signal (i.e., noise). This theory was initially developed in psychology to explain human perception and decision-making but has since found wide applications in various fields, including image processing and machine learning.

3.4.1 Signal detection theory and operating characteristics

In the previously described binary classification process, signal refers to an actual edge or feature in the image (or point cloud). At the same time, noise denotes regions that do not contain edges but may still be classified as edges. While processing the image (or point cloud), the classifier produces responses or scores for each pixel or region. The responses can be considered the evidence the classifier bases its binary decision on. The distribution of these responses is critical since it informs the classification decision. Suppose the detector creates a signal whenever it is used on a region. The detector gives out a signal with mean μ_1, when an edge point is present, and a signal with a mean of μ_2, when it is a non-edge point. The assumption suggests itself that the distributions are normal with different means but the same variance ($p(x|\omega_i) \sim N(\mu_i, \sigma^2)$). To classify each pixel (or point), the classifier employs a decision criterion x^* (see Figure 3.9). The decision criterion is a threshold, and responses above this threshold are classified as signals, while responses below this threshold are classified as noise. Assuming we do not know the values of μ_1, μ_2, σ or x^*, but the correct answer and the system's decision. Four defining probabilities can be identified (also compare Figure 3.9 on the left and Figure 3.10) [32]:

- *true positive* (TP): $P(x > x^*|x \in \omega_2)$
  The probability that the internal signal is above x^* given that the external signal is present, i.e., a classification that correctly classifies an edge point as an edge point.

- *false positive* (TP): $P(x > x^*|x \in \omega_1)$
  The probability that the internal signal is above x^* despite no external signal present, i.e., a classification that wrongly classifies a non-edge point as an edge point.

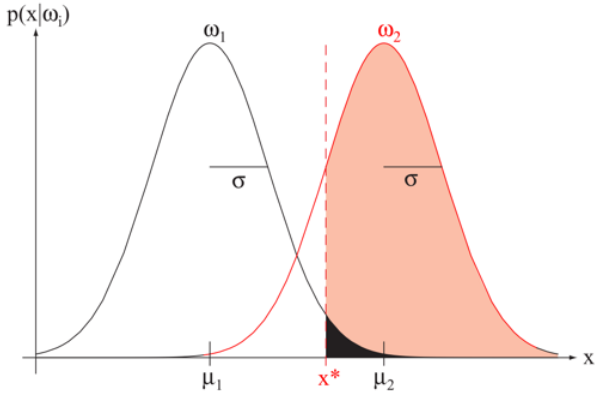


Figure 3.9: Probability density functions for signal detection. The signal of the detector if an edge point is present is given by ω_1 and the signal of the detector given a non-edge point is denoted by ω_2. The decision threshold x^* determines the probability of a true positive (the pink area under the ω_2 curve, above x^*) and of a true negative (the black area under the ω_1 curve, above x^*) [32].

- *false negative* (FN): $P(x < x^* | x \in \omega_2)$
  The probability that the internal signal is below x^* given that the external signal is present, i.e., a classification that falsely dismisses a point as a non-edge point even though it is an edge point.
- *true negative* (TN): $P(x < x^* | x \in \omega_1)$
  The probability that the internal signal is below x^* given that the external signal is not present, i.e., a correct dismissal of a point that really is a non-edge point.

		Predicted condition	
	Total number of signals (= P + N)	Positive (PP)	Negative (PN)
Actual condition	Positive (P)	True positive (TP)	False negative (FN)
	Negative (N)	False positive (FP)	True negative (TN)

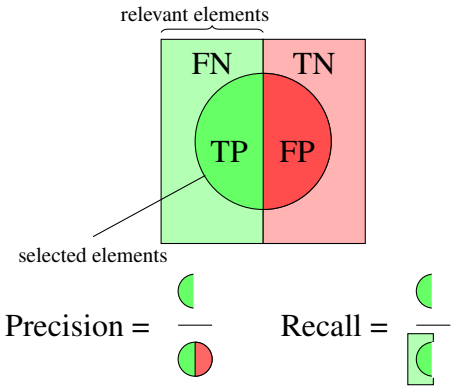


Figure 3.10: Confusion matrix for the signal detection example on the **left**, graphical representation of the different classification outcomes on the **right**.

3.4.2 Performance metrics in binary classifiers

Performance metrics for binary classifiers provide a comprehensive view of a classifier's abilities. Based on previously defined outcomes, a multitude of performance measures are defined [33]:

- *Precision* - also known as positive predictive value (PPV):
  Precision quantifies the classifier's ability to identify the positive class while minimising false positives correctly. It is crucial when the cost or impact of false positives is high. It evaluates the classifier's ability to minimise false alarms while correctly identifying the positive class. High precision means that when the classifier indicates a positive result, it is correct with a high probability.
  $$PPV = \frac{TP}{TP+FP}$$

- *Recall* - also known as sensitivity or true positive rate (TPR):
  The sensitivity represents the ability of the classifier to correctly identify signals - the true positive rate. The recall is particularly interesting when the cost of missing a positive (i.e., a false negative) is high, as it assesses the classifier's ability to minimize such misses.
  $$TPR = \frac{TP}{TP+FN}$$

- *Specificity* - also known as selectivity or true negative rate (TNR):
  The Specificity measures the ability of the classifier to reject noise correctly - the true negative rate. Specificity is of particular interest when the cost of false positives is high or precision is of primary concern.
  $$TNR = \frac{TN}{TN+FP}$$

- *ROC-Curve* - also known as Receiver Operating Characteristic:
  The ROC-curve is a graphical representation of a binary classifier's performance that illustrates its ability to discriminate between the positive and negative classes (see Figure 3.11, on the left). It provides insights into how a classifier's sensitivity (true positive rate) and specificity (true negative rate) change at different decision thresholds, thus visualising the trade-off between sensitivity and specificity. As the decision threshold is adjusted, sensitivity typically increases at the cost of reduced Specificity, and vice versa [34]. In ROC space, a perfect classifier's score is located in the top left corner of the graph ($TPR = 1$ and $FPR = 0$).

- *Precision-Recall Curve*:
  The Precision-Recall curve works similarly to the ROC curve and is a graphical representation of a binary classifier's performance that focuses on its ability to balance precision and recall across different decision thresholds (see Figure 3.11, on the right). In PR space, a perfect classifier's score is located in the top right corner of the graph ($Precision = 1$ and $Recall = 1$).

- *F1 Score*:
  The F1 score is a single metric combining precision and recall into one value, providing a balance (harmonic mean) between these performance measures. It is beneficial when, in binary classification, both precision and recall should be optimised simultaneously.
  $$F_1 = \frac{2 \cdot TP}{2 \cdot TP + FP + FN}$$

- *AUC* - also known as Area-Under-Curve:
  The AUC quantifies the overall performance of the classifier based either on the ROC (called ROC-AUC) or Precision-Recall (namely PR-AUC) curves. A perfect classifier has an AUC of 1, while a random classifier has an AUC of 0.5. The AUC provides a single summary metric for assessing classifier discrimination ability in the case of the ROC-AUC, or, in the case of PR-AUC, the ability to combine both Precision and Recall [34].

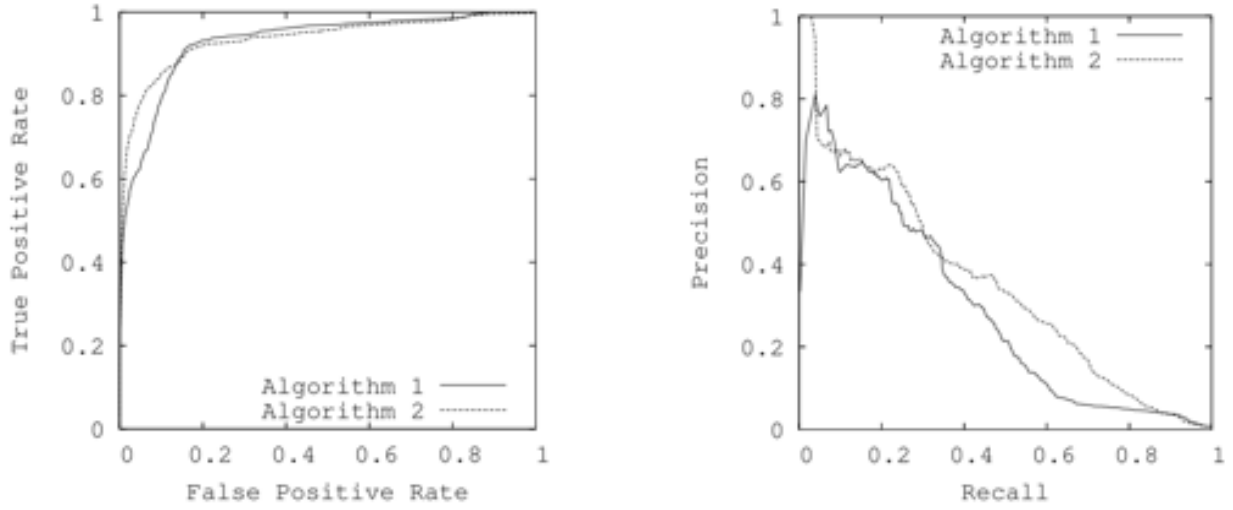


Figure 3.11: Comparison of two example algorithms in ROC-space on the **left** and PR-space on the **right** [35].

Most binary classifiers yield fuzzy classification results, e.g., probabilities that an item is part of the one class and not the other. A thresholding step is subsequently needed to assign elements to a specific class (i.e., generate crisp class labels). Both the ROC and Precision-Recall (PR) plot the classifier's performance across many different thresholds, giving an overview of the classifier's performance across the whole spectrum (see Figure 3.11).

The choice of the threshold value ultimately depends on different costs assigned to a wrongful classification of items into one or the other class, respectively. If no cost can be assigned to the different wrongful classifications, or if the different faults have equal cost, an ideal threshold can be obtained from the plots by finding the threshold that is closest to the ideal classifier (i.e., closest to the top left corner in the ROC and top right corner in the PR). In this respect, each metric has its strengths and shortcomings, making them useful for different purposes. When assessing a binary classifier, it is vital to consider the specific goals and trade-offs involved. Table 3.1 provides an overview of commonly used metrics, including ROC, Precision-Recall, F1 Score, and AUC, and their applicability to an edge detector.

3.5 Neural Networks

Computers are classically built using a single processor with significant processing power. On the other hand, the human brain is comprised of in the order of 10^{11} neurons. Each of these neurons is a highly connected processing unit, with every one connected to about 10^4 other neurons working

Metric	Strengths	Drawbacks
ROC	• Provides a graphical representation of performance trade-offs. • Allows visual assessment of sensitivity and specificity. • ROC-AUC quantifies overall classifier performance (i.e., over different thresholds).	• Less informative for imbalanced datasets.
Precision-Recall	• Focuses on the positive class, suitable for imbalanced datasets. • Balances precision and recall across different thresholds. • AUC-PR quantifies overall classifier performance (i.e., over different thresholds).	• May not be ideal when false negatives are not as critical. • May not adequately address specific trade-offs.
F1 Score	• Combines precision and recall into a single metric. • Useful when optimizing both precision and recall is necessary. • Particularly valuable when there is no distinct preference for false positives or false negatives and when both types of classification errors need to be minimised.	• Does not account for true negatives. • Requires crisp labels and, as such, a choice of a threshold value.
AUC	• Quantifies overall classifier performance. • Valuable for comparing different classifiers or thresholds. • Can be based on ROC (i.e., AUC-ROC) or Precision-Recall (i.e., AUC-PR).	• Less interpretable than other metrics. • May not directly address specific trade-offs.

Table 3.1: Comparison of the strengths and drawbacks of different performance metrics for a binary classifier [33].

in parallel to solve problems. The processing and memory capabilities are distributed across the network, making it a distributed system [36]. In vision, speech recognition, and learning - to name just a few domains - the brain's processing power still surpasses the processing capabilities of current state-of-the-art computer systems [36]. The basis for the introduction of brain modelling into the realm of computer science was established in 1949, when McCulloch and Pitts defined the McCulloch-Pitts neuron (MCP neuron), a highly simplified model of a human neuron [37]. Based on this very early model Minsky's revolutionary dissertation in 1954 evoked much interest in the field with his theoretical work on interconnected neurons - some of the first neural networks [38]. Since then, much research has been focused on understanding and modelling the human brain and artificial neural networks and applying them to real-life problems.

3.5.1 Artificial neurons and the perceptron

Since the early work of McCulloch and Pitts, more work was invested in developing the fundamental building blocks, the neurons. The MCP neuron behaves similarly to a biological neuron; it receives binary inputs and produces binary outputs based on a pre-defined threshold. Later, in 1957, Rosenblatt improved upon the MCP neuron and introduced the perceptron, which can also work on non-boolean input values [39]. To this day, the perceptron is seen as the fundamental building block of NNs.

As depicted in Figure 3.12, inputs into the perceptron may come from the environment or the out-

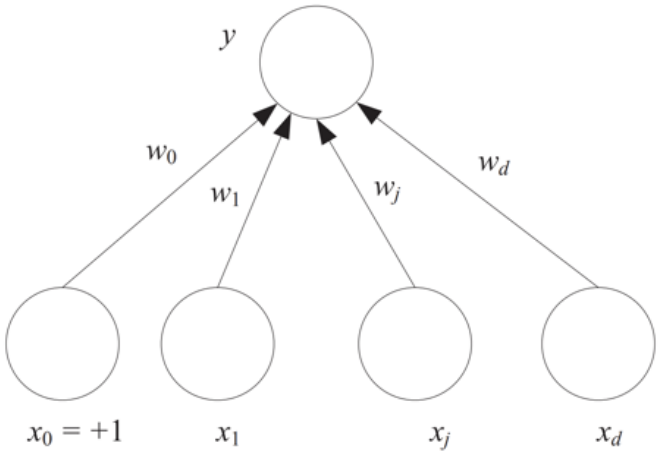


Figure 3.12: The simplest form of a perceptron: With the input units $x_j, j = 1,\ldots,d$, the bias unit that is always equal to 1 w_0, the with w_j weighted connections from input to output and the output unit y [36].

puts of other perceptrons. Each of the inputs $x_j \in \mathbb{R}, j = 1,...,d$ into the perceptron is associated with a weight $w_j \in \mathbb{R}$ [36]. In its simplest form, the output y is computed through a weighted sum of the inputs:

$$y = \sum_{j=1}^{d} w_j x_j + w_0, \tag{3.11}$$

where w_0 is sometimes called the intercept value and is most commonly modelled as a bias unit coming from an input x_0 that is always equal to 1 and is used to make the model more general [36]. The output of the perceptron can be rewritten using the dot product as

$$y = \mathbf{w}^T \mathbf{x}, \tag{3.12}$$

where $\mathbf{w}^T = [w_0, w_1, \ldots, w_d]^T$ and $\mathbf{x} = [1, x_1, \ldots, x_d]^T$.

To make the model of a perceptron useful for a given task, the weights $\mathbf{w}$ have to be learned such that correct outputs are generated, given the inputs.

It is useful to look at the simplest case to understand that the perceptron defined in Equation 3.11 defines a hyperplane. Given $d = 1$, the equation collapses to

$$y = wx + w_0, \tag{3.13}$$

the equation of a linear function with a slope of w and an intercept of w_0 (hence the name). Therefore, a perceptron in this fashion can make a linear fit. Perceptrons with more than one input can implement multivariate linear fit, represented by a hyperplane. Regression can be used to find the parameters w_j given a sample, and this will be elaborated in a later section (see Section 3.5.4) [36]. The fact that the perceptron defined in Equation 3.11 defines a hyperplane can be used to divide the input space in two. The two resulting half spaces - one where the output is positive and the other where the output is negative - can be used to implement a linear discriminant function. This can be used to separate two classes by checking the output sign. In accordance with Alpaydin a threshold function $s(\cdot)$ can be defined such that

$$s(a) = \begin{cases} 1 & \text{if } a > 0 \\ 0 & \text{otherwise} \end{cases} \quad \text{and then choose} \quad \begin{cases} C_1 & \text{if } s(\mathbf{w}^T\mathbf{x}) > 0 \\ C_2 & \text{otherwise} \end{cases} \tag{3.14}$$

that effectively sorts input values into one of two classes, namely C_1 and C_2 [36]. The described approach using a linear discriminant assumes that the given classes C_1 and C_2 are linearly separable, i.e., that a hyperplane $\mathbf{w}^T\mathbf{x} = 0$ exists, so that a clear separation between $\mathbf{x}^t \in C_1$ and $\mathbf{x}^t \in C_2$ can be found [36]. Note that $\mathbf{x}^t$ represents a specific data point in the dataset. As previously men-

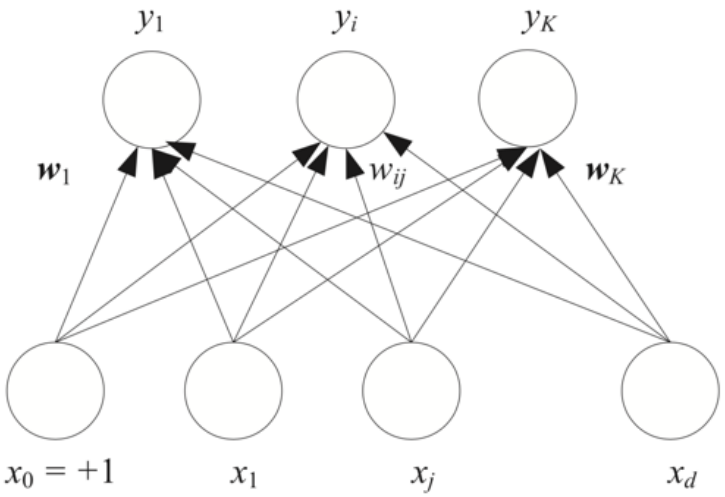


Figure 3.13: Setup of K parallel perceptrons with the inputs $x_j, j = 1,\ldots,d$, the outputs $y_i, i = 1,\ldots,K$ and the weighted connections w_{ij} [36].

tioned, the inputs to a perceptron may come either from the environment or another perceptron. Perceptrons may be used in parallel setups as depicted in Figure 3.13. In this case, the output of the network is computed using

$$y_i = \sum_{j=1}^{d} w_{ij}x_j + w_{i0} = \mathbf{w}_i^T\mathbf{x} \quad \text{with} \quad i = 1,\ldots,K \tag{3.15}$$

$$\mathbf{y} = \mathbf{W}\mathbf{x}, \tag{3.16}$$

note that now $W \in \mathbb{R}^{K\times(d+1)}$ is a weight matrix [36]. The classification is then done using the maximum resulting value:

$$\text{choose } C_i \text{ if } y_i = \max_k y_k. \tag{3.17}$$

3.5.2 Multilayer Perceptrons and Deep Neural Networks

As demonstrated in the preceding section, a perceptron with a single layer of weights can only be utilised to model linear functions in its outputs, and it cannot effectively model relationships that are nonlinear, such as those encountered in nonlinear regression tasks. To eliminate this limitation, perceptrons can be built with multiple parallel units and stacked into so-called layers. These Multilayer Perceptrons (MLP) use hidden layers of neurons to enable them to approximate nonlinear functions of the input; see Figure 3.14.

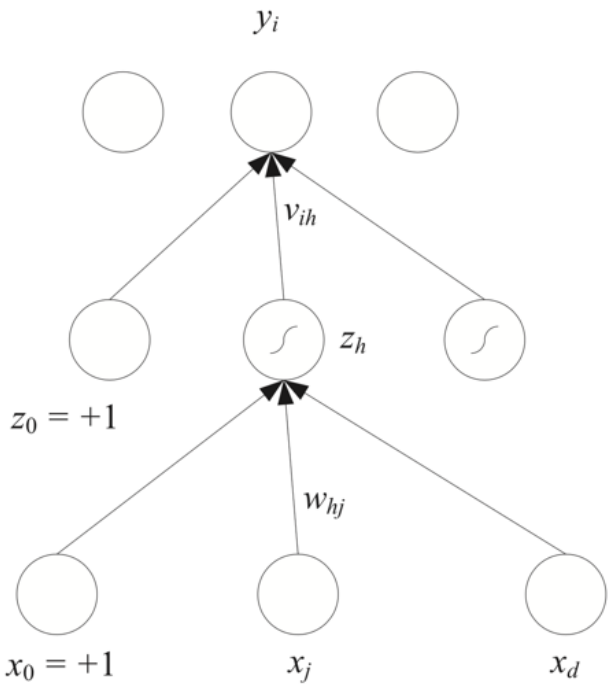


Figure 3.14: A MLP with the inputs $x_j, j = 1,\ldots,d$, the hidden units $z_h, h = 1,\ldots,H$, the dimensionality of the hidden space H, the bias of the hidden layer z_0, the output units $y_i, i = 1,\ldots,K$, the weighted connections w_{hj} in the first layer and the weighted connections in the second (hidden) layer v_{ih} [36].

The stacking of layers can be pushed even further, and multiple hidden layers can be utilised, with each consecutive hidden layer taking the inputs of the previous layer. The MLP has, compared to other machine learning approaches (e.g., Principal Component Analysis (PCA)) the advantage that it is also trained on data, but that feature extraction (first layer) and the combining of these features are learned together, and that the learning is done in a supervised manner [36]. It has been demonstrated that a Multilayer Perceptron (MLP) with a single hidden layer can act as a universal approximator, meaning that it can approximate any function with arbitrary accuracy depending on the number of hidden neurons. However, networks with deeper layers and fewer parallel neurons not only have fewer parameters and, therefore, lower complexity but also exhibit better generalisation in practice [36].

In many, if not most, real-world applications, the structure of the input and its dependencies are unknown and should be learned in training. Therefore, deep learning networks aim to "learn feature levels of increasing abstraction with minimum human contribution" [36].

Deep learning has many practical upsides. However, some caveats include increased memory and computing power requirements due to more weights and processing units and more difficult training due to more free parameters and a more complex error surface [36].

3.5.3 Gradient descent learning

So far, the shape has been described, but how the weights can be found has not yet been discussed. The most widely used technique for training neural networks, supervised learning, uses labelled training data to adapt the parameters of the network. Before this background, labelled training data is data for which the expected output of the neural network is well known—the data points within the set are provided with ground truth labels, providing the benchmark for the network's output. Generally, online learning is used, where the network is not given the whole dataset at once, but instances are presented individually to the network. After each presentation of a data instance in the forward direction, the network performance is rated using an error function computed from the network output compared to the current instance's ground truth label. Starting from random initial weights, the parameters are adjusted gradually to minimise the error. Given the error function is differentiable, gradient descent can be used to minimise the error and find a local minimum [36].

Suppose a data instance has been presented to the network's inputs. The training process involves running the training instance in both directions. First, the data point $\mathbf{x}$ is passed through the network in the forward direction computing a prediction $\mathbf{y}$ based on the current parameters. This prediction is now compared to the ground truth label of the current data point using the error function. The error E serves as a supervision signal, indicating how to modify the model's parameters. Using gradient descent, the parameters are updated:

$$\mathbf{w}_i \leftarrow \mathbf{w}_i - \eta \frac{\partial E}{\partial \mathbf{w_i}}, \tag{3.18}$$

where η is the learning rate or learning factor, which is gradually decreased over time for convergence [40]. Mathematically, the partial derivative $\frac{\partial E}{\partial \mathbf{w_i}}$ in Equation 3.18 is called a gradient, denoting the rate of increase of E respective to the changes in the different dimensions of w_i. Thus, the gradient indicates - within a small region around the current value of w_i in weight space - the direction of change for w_i to increase E and vice versa for the negative gradient, the direction of change for w_i to decrease E, i.e., minimise the loss function. To limit the change to the small region around the current value of w_i, the learning rate η is additionally applied - generally set to a tiny number [40].

After presenting a single data instance and updating the weights using gradient descent, the next data instance is selected, and the process is repeated. An epoch is when each data instance in the training set is presented once.

Generally, updating the parameters using the gradient after every training example can lead to a very unstable loss function due to the gradient's instability. Using small subsets of the training set to update the parameters is much more practical. The process of updating the parameters over such a subset is called Stochastic Gradient Descent [40].

3.5.4 Error Backpropagation

The previous section detailed how the parameters of a neural network can be trained. However, in practice, it is not yet clear how to compute the gradient to modify the weights.

As was shown in Equation 3.16, each unit computes a weighted sum of its inputs. A non-linear activation function $g(\cdot)$ is applied to compute the activation z_j of unit j in the form

$$z_j = g(a_j). \tag{3.19}$$

Note that any number of variables z_i in this equation could be inputs, in which case they shall be denoted by x_i, and the unit j could be an output unit, in which case y_k shall denote them [41]. Two assumptions have to be made to derive a general approach for backpropagation. First, we must assume that the cost function can be written as an average over individual cost functions E^n for individual training samples:

$$E = \sum_n E^n. \tag{3.20}$$

This assumption is needed because the backpropagation algorithm will compute partial derivatives for individual training samples. The total partial derivative is subsequently attained through averaging over the individual derivatives [41].

The second assumption to make is that the error shall be expressed as a differentiable function of the network output variables:

$$E^n = E^n(y1, \ldots, y_k). \tag{3.21}$$

The output of the various output units depends on which training sample has been fed into the network. However, since according to Equation 3.20, there is a way to compute the total from the individual data points, the subscript will be omitted. If we consider the evaluation of the derivative of E_n for a weight w_{ij}, it is important to note that E_n depends on the weight w_{ji} solely through the summed input a_j to unit j [41]. The chain rule for partial derivatives can be applied with the second term in the sum cancelling out:

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial a_j}\frac{\partial a_j}{\partial w_{ji}}. \tag{3.22}$$

Introducing a new notation for the errors δ_j, with

$$\delta_j \equiv \frac{\partial E_n}{\partial a_j}, \tag{3.23}$$

and using Equation 3.16 generalised to multiple layers (replace the output $\mathbf{y}$ by the activations $\mathbf{a}$) to obtain:

$$\frac{\partial a_j}{\partial w_{ij}} = z_i. \tag{3.24}$$

Substitution of this term and δ in Equation 3.22 results in

$$\frac{\partial E_n}{\partial w_{ji}} = \delta_j z_i. \tag{3.25}$$

Through the use of Equation 3.25, the required derivative is obtained with only the value of δ for the unit at the output end of the weight and the value of z at the input end of the weight as prerequisites. Therefore, the derivatives can be evaluated by calculating δ_j for each hidden and output unit in the network in conjunction with Equation 3.25 [41]. The δ's for output units are

easily obtained through comparison with the ground truth labels. For hidden units, the chain rule for partial derivatives is applied again, yielding

$$\delta_j \equiv \frac{\partial E_n}{\partial a_j} = \sum_k \frac{\partial E_n}{\partial a_k} \frac{\partial a_k}{\partial a_j}, \tag{3.26}$$

with k being the number of units to which unit j sends connections [41].

A complete backpropagation formula can now be obtained

$$\delta_j = h'(a_j) \sum_k w_{jk} \delta_k, \tag{3.27}$$

showing that backpropagating the δ's from higher up units in the network, the value of δ for any particular hidden unit can be obtained [41].

3.5.5 Activation functions

In Section 3.5.1, it was shown that a perceptron can be used to divide its input space into two half spaces using a threshold function. In Equation 3.14, a simple step function was proposed. However, the backpropagation algorithm, detailed in the previous section, tries to minimise the error function in the weight space using gradient descent. At the core of gradient descent lies the computation of the gradient of the error function at each iteration step. For the gradient to exist, however, the continuity and differentiability of the error function must be guaranteed. The step function does not meet these requirements, making the composite function produced by interconnected perceptrons and the error function discontinuous [42].

The most popular proposed activation function for backpropagation networks is the sigmoid function defined as:

$$s_c(x) = \frac{1}{1 + e^{-cx}}, \tag{3.28}$$

with $s_c(x) : \mathbb{R} \to (0,1)$ a real function over the constant c (see Figure 3.15, top left). The sigmoid function gets closer to the shape of the step function with higher values of c, and in the limit $c \to \infty$ converges to a step function in the origin [42]. Other types of activation functions suitable for backpropagation networks include the hyperbolic tangent, Rectifier Linear Unit and the Leaky Rectifier Linear Unit; see Figure 3.15 for a comparison.

3.5.6 Cost functions

The cost function measures the performance of a neural network for its given training sample and expected output. The cost function is represented by a single value (not a vector), rating how well the network did. A cost function comes in the form

$$C(\mathbf{W}, \mathbf{B}, \mathbf{S}^r, \mathbf{E}^r), \tag{3.29}$$

where $\mathbf{W}$ is the neural networks' weights, $\mathbf{B}$ is the neural networks' biases, $\mathbf{S}^r$ is the input of a single training sample, and $\mathbf{E}^r$ is the desired output associated with the training sample [43].

In Section 3.5.4 it was shown that the requirements towards a suitable cost function are twofold: First, it must be possible to write it as an average over individual cost functions (see Equation

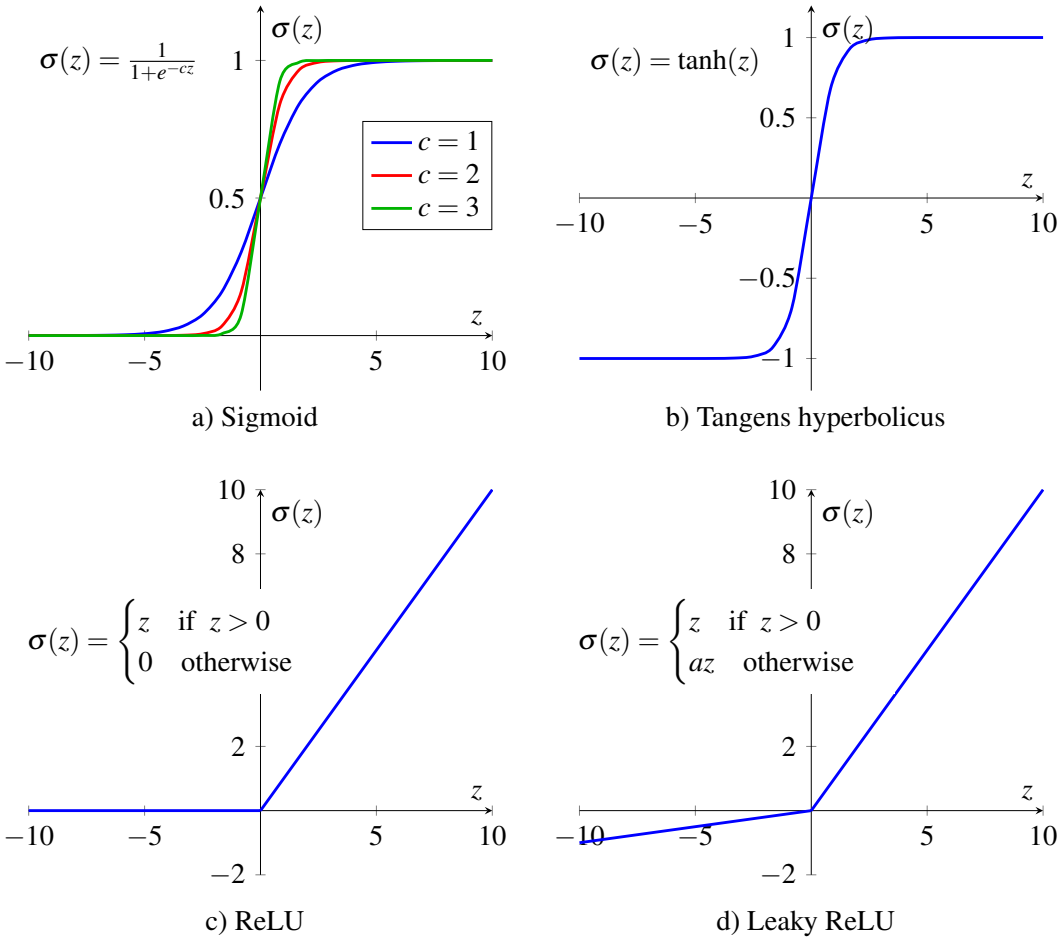


Figure 3.15: Comparison of different activation functions: Sigmoid functions with different values of c on the **top left**, hyperbolic tangent on the **top right**, Rectifier Linear Unit on the **bottom left** and the Leaky Rectifier Linear Unit on the **bottom left**.

3.20); secondly it must be able to be represented as a differentiable function of the network output variables (see Equation 3.21).
The loss function shall check for equality in class learning (where the neural network outputs are 0 0r 1). In Regression, however, because the neural network's output is a numeric value, the network provides a piece of ordering information for distance. In the second case, many different distance metrics are used [36].

3.5.7 Convolutional Neural Networks

Convolutional Neural Networks (CNN) are a special form of neural networks for the processing of input data with a local structure (e.g., image data - see Section 3.2.1). In such a case, not all inputs are correlated since only a certain neighbourhood around an individual entry influences it. This means that during MLP design, not all hidden units need to be connected to all inputs; instead, hidden units are designed to define a window over the input space. Only a small subset of all the inputs are connected to the hidden units [36].

As with an ordinary MLP setup, this process can be repeated in successive layers. Later layers

shrink and represent more complicated features than previous layers. In a practical example, the input might be an image, the first layer might check for edges in various orientations, and the proceeding layer might combine the edges into corners, arches, and other simple shapes. Later layers might build on these simple shapes and learn to look for rectangles, semicircles, etc. The underlying principle is often called a *hierarchical cone* where features get more complex, more abstract and less local the deeper the transgression into the network [36].

In CNNs, the neural network learns the features by itself. Since the features shall be the same over the entire input space (the detection of an edge shall work no matter where on the image plane the edge is located), the same input units need to look at different subsets of the input space. This process can be pictured as if a small frame - the kernel - is moved over the input data, and the results are computed in all different possible positions (see Figure 3.16, left). The results from

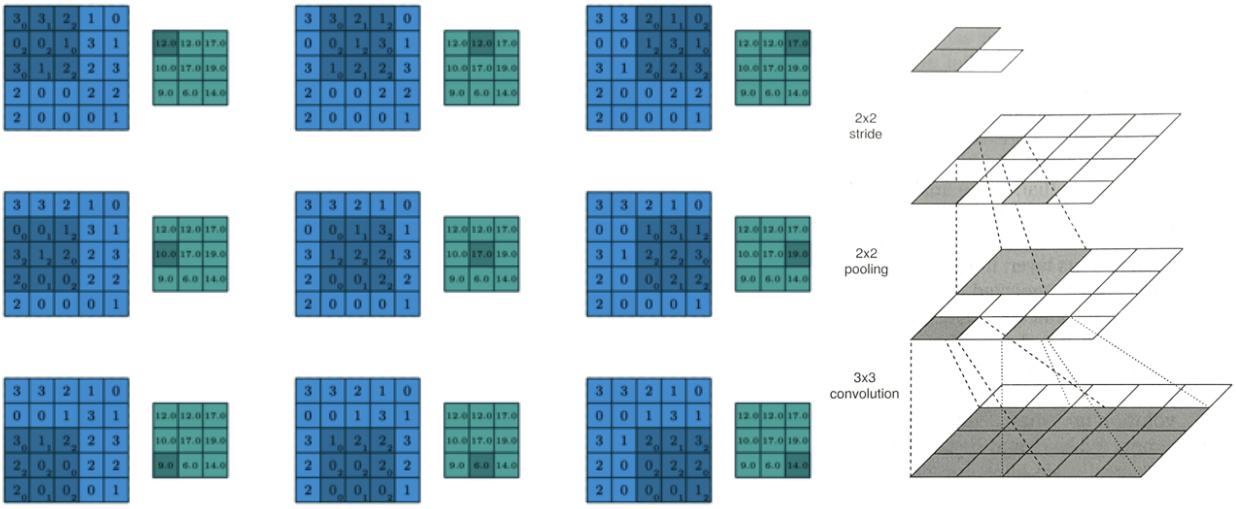


Figure 3.16: Convolving a 3 x 3 kernel over a 5 x 5 input without padding and unit strides on the **left** [44], a 3 x 3 convolutional layer with subsequent pooling layer (2 x 2) and a stride of 2 reduces the resolution by half on the **right** [36].

the convolution computation are generally completed with a bias. After applying a non-linearity (see Section 3.5.6), the feature map output is generated. A convolution is, therefore, essentially a hidden unit whose input is restricted to a subset of the input [36].

To accelerate the resolution decrease in networks with multiple convolutional layers, *striding* and *pooling* are used. In pooling, a neighbourhood size is set, and within this neighbourhood, either the maximum or average value is used, in striding on the other hand, instead of using values at every location, some are skipped (see Figure 3.16, right). Note that both of these operations are fixed, and there are no learned parameters involved [36].

Inputs into convolutional layers can consist of multiple parallel channels. Classical two-dimensional colour images, for example, come with three channels (Red, Green and Blue). Channels can also be created within the network by applying multiple convolutional layers in parallel. Therefore, data inputs into the convolutional layers come as tensors. Tensors can be viewed as generalised, higher-order matrices - e.g., $\mathbf{x} \in \mathbb{R}^{HxWxD}$ is a third-order tensor. Classical two-dimensional colour images are third-order tensors consisting of stacked second-order tensors of the same image in the RGB channels.

Within the architecture of the CNN, typically, with each step, the resolution decreases, but the set of abstract features increases. Once the compressed representation of the image is expected to be good enough, a fully connected layer is used to produce an output. All units in the fully connected layer are connected to all the units over all the channels of the output of the previous convolutional layer. In principle, the output is flattened before being input into the fully connected layer. Refer to Figure 3.17 for an example of the architecture. The fully connected layer drops the explicit position information and detects abstract features over the whole image. Alpaydin fittingly compares the convolutional layers to analysers that break down the image, where the fully connected layers are synthesizers that generate the outputs [36].

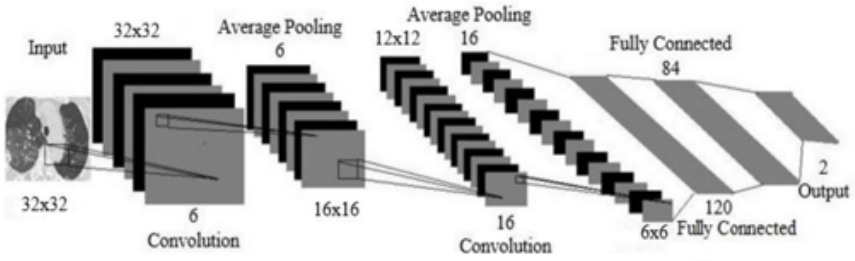


Figure 3.17: Example of a CNN- Architecture based on LeNet-5 and used for the detection of COVID-19 in CT images [45].

All the algorithms introduced in previous sections for the process of supervised learning in neural networks (i.e., MLP) can be generalised to be used on CNN. CNNs are thus trained in a very similar fashion to MLPs [40].

3.6 Edge detection

Edge detection is a classification problem intended to assign the edge or non-edge property to a query point considering its immediate neighbourhood. To achieve this, most algorithms use a two-stage process, first detecting various features and then classifying the points. The features used for the classification differ widely depending on the algorithm used.

3.6.1 Edge features in different dimensions

Traditionally, Rosenfeld and Thurston distinguished edge features in 2D data into two types:

1. Abrupt changes in average grey level define grey level edges and usually represent visible edges in the scene (see Figure 3.18 on the left) [46].

2. Texture edges represent abrupt coarseness changes between adjacent regions containing the same texture, but seen at two different scales (see Figure 3.18 on the right) [46].

However, 2D images represent fundamentally different information compared to 3D point clouds - where the images contain mostly spectral information, the point cloud contains mostly spatial information. This led Ni et al. to visually define 3D edges as "3D discontinuities of the geometric properties in the underlying 3D-scene" [14]. Mathematically, this leads to two different types of edge features (also see Figure 3.19):

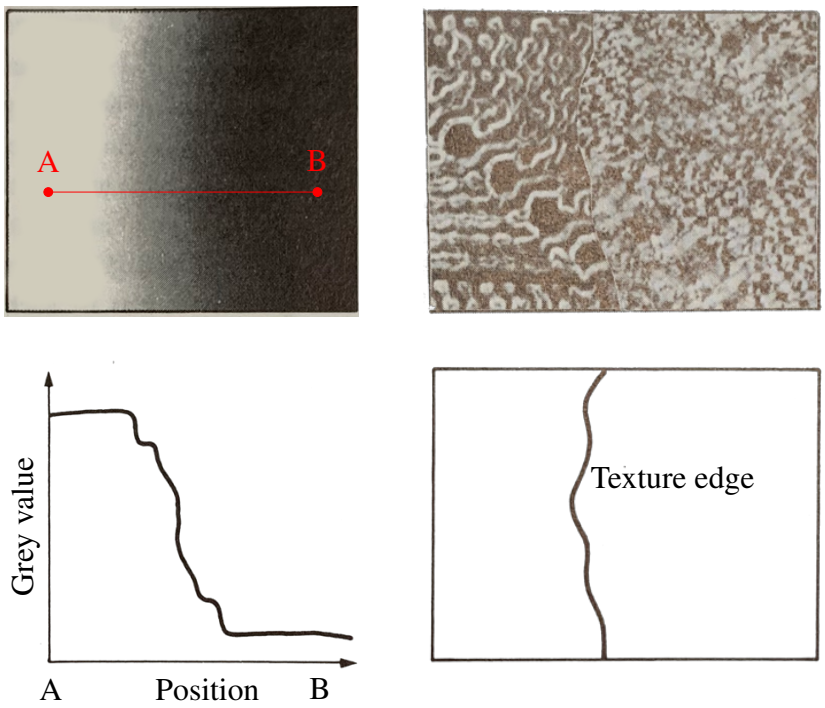


Figure 3.18: Grey value edge example (**top left**) with grey value representation along the connection $\overline{AB}$ below on the **bottom left** and texture edge example on the **right** [47].

1. Boundary elements lie on a neighbourhood boundary that a single curve or planar surface can represent [14].
2. Fold edges exist in a neighbourhood where two or more curve or planar surfaces intersect [14].

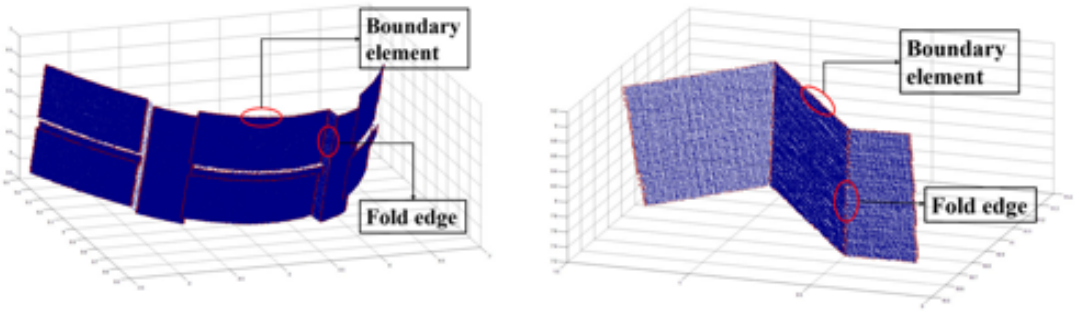


Figure 3.19: Types of edges in 3D data [14].

3.6.2 Classical approaches on two-dimensional data

As previously described (see Section 3.6.1), edge features within 2D data are either changes in grey levels or differences in textures. In both cases if a pixel is an edge pixel a derivative of some

form has a large magnitude in the place of the pixel [48].

The most common method to find edges is to start by computing the gradient magnitude in each point. Since we are operating on two-dimensional data, it is important to consider level changes in many directions. This can be done using partial derivatives along the principal directions of the image x and y [49]. Representing the image as a function of two variables $\mathbf{A}(x,y)$, the gradient is defined as a two-dimensional vector as follows:

$$\nabla \mathbf{A}(x,y) = \left(\frac{\partial \mathbf{A}}{\partial x}, \frac{\partial \mathbf{A}}{\partial y} \right). \tag{3.30}$$

The edge response is finally obtained by:

$$\mathbf{G}_{mag} = \sqrt{\left(\frac{\partial \mathbf{A}}{\partial x} \right)^2 + \left(\frac{\partial \mathbf{A}}{\partial y} \right)^2}. \tag{3.31}$$

The magnitude obtained using Equation 3.31 is usually thresholded to obtain edge pixels.
Since an image sensor discretises the physical (continuous) world into a matrix of individual pixels, an image cannot be differentiated in the usual way (as suggested in Equation 3.30). Thus the derivative in a pixel has to be approximated by the difference of grey levels over some local region. Many different approximations of the gradient have been proposed, some of the most widely accomplished ones being the Sobel and Prewitt edge detectors (see Table 3.2).

Operator	Gradient, x-Direction	Gradient, y-Direction
Prewitt	$\begin{bmatrix} -1 & 0 & 1 \\ -1 & \mathbf{0} & 1 \\ -1 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} -1 & -1 & -1 \\ 0 & \mathbf{0} & 0 \\ 1 & 1 & 1 \end{bmatrix}$
Sobel	$\begin{bmatrix} -1 & 0 & 1 \\ -2 & \mathbf{0} & 2 \\ -1 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} -1 & -2 & -1 \\ 0 & \mathbf{0} & 0 \\ 1 & 2 & 1 \end{bmatrix}$

Table 3.2: Typical gradient operator masks for approximating first derivatives over a local area (i.e., 3x3). The element at the origin is shown in boldface [50].

While using the first derivative, the gradient peaks are indicators of edges, and zero-crossings are indicators of edges in the second derivative of the image intensity. The second derivative of the intensity function of the image can be written as:

$$\Delta \mathbf{A}(x,y) = \frac{\partial^2 \mathbf{A}}{\partial x^2} + \frac{\partial^2 \mathbf{A}}{\partial y^2}. \tag{3.32}$$

The second derivative must also be approximated in discrete space when dealing with the first derivative. For this purpose, the Laplace operator has been proposed:

$$\text{Laplacian} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -\mathbf{4} & 1 \\ 0 & 1 & 0 \end{bmatrix}. \tag{3.33}$$

Both approaches are utilised similarly based on the first or second derivative. The gradient masks (kernels) are evaluated in every pixel of the image of interest. The output is a new image of the same size as the input image. The information stored in the individual pixels, however, has changed. Where before, they stored information on the light intensity in this pixel (grey value), they now represent the gradient magnitude of the respective derivatives (first or second). This information can be visualised as shown in Figure 3.20.

Figure 3.20: Comparison of kernel-based edge detectors. The original image on the **left**, gradient magnitude computed with a Sobel filter (based on the first derivative) in the **middle** and gradient magnitude computed on the Laplacian of Gaussian (based on the second derivative) on the **right** [51].

3.6.3 Convolutional Neural Networks for two-dimensional data

Two-dimensional vision data has a local structure: Nearby pixels are correlated, and there are local features like edges and corners (see Section 3.2.1). Therefore, applying CNN for detecting edges is a logical endeavour and is performed routinely today. The neural network learns kernels for the detection based on training data - it is a data-driven approach instead of the analytical or heuristics-driven classical approaches. As with most machine learning applications, the achieved performance is highly dependent on the quality of the training data available.

3.6.4 Neural Networks for three-dimensional data

Where CNNs in 2D images can exploit the fact effectively that spatial relationships between pixels are encoded into the data structure (see Section 3.2), their application on 3D point cloud data is much more complex.

For a long time, solutions in three-dimensional space depended on restructuring the input data into a more convenient data structure for the conventional CNNs. However, the proposed solutions' downsides (apart from the fact that they could not process point clouds natively) included extracting low-level geometric features as inputs to CNNs [52], densifying sparse data (e.g., converting the 3D point cloud data into a three-dimensional grid, with the grid values representing the number of points present within that respective volume) and rapidly growing data requirements based on the required granularity of the network (e.g., by the cube when dealing with three-dimensional grids) [1].

Only recently have point clouds drawn more attention, and the focus lay on utilising neural networks to directly process point clouds to eliminate the previously necessary conversion steps and their downsides. One of the first proposals came from PointNet, a network used for object classification and segmentation based on features directly learned from unordered 3D point clouds [53]. PointNet achieves this by employing a symmetric function to capture permutation invariance, allowing the network to process unordered points. Some of the key characteristics of PointNet include the aforementioned permutation invariance (based on symmetric functions such as max pooling) to aggregate local features of each point (ensuring that the network's predictions are invariant to the order of points in the input), the use of MLPs to extract local features for each point (capturing both local and global information), and the aggregation of the local features of each point into a global feature vector (representing the entire point cloud). This global feature vector is finally fed into a fully connected network for further processing or classification [53].

Based on the pioneering work done with PointNet (and its successor PointNet++ [54]), Yu et al. were one of the very first who used neural networks to natively consume three-dimensional point clouds for edge detection through their implementation of EC-Net [52]. EC-Net's main purpose is to be a deep edge-aware point cloud consolidation network (see Figure 3.21). However, EC-Net's first phase consists of upscaling the point cloud, an edge point classifier and a regression of per-point distances to the edge. This information is valuable for precise edge detection and localisation. In the second step, edge points can be detected through their zero distance from the edge. Despite its innovative approach and promising results, EC-Net suffers from certain limitations: It utilises very deep network architecture, which demands significant computational resources during the training and inference phases, which may pose challenges for real-time applications. Additionally, the upscaling process in EC-Net may limit its scalability, particularly when dealing with large-scale point cloud data. This could restrict its applicability to specific use cases or scenarios where scalability is a concern. Addressing these limitations would enhance the practical utility and widespread adoption of EC-Net in domains requiring edge detection in three-dimensional point cloud data. Strategies such as network optimisation, resource-efficient architectures, and scalability enhancements could be explored to mitigate these challenges.

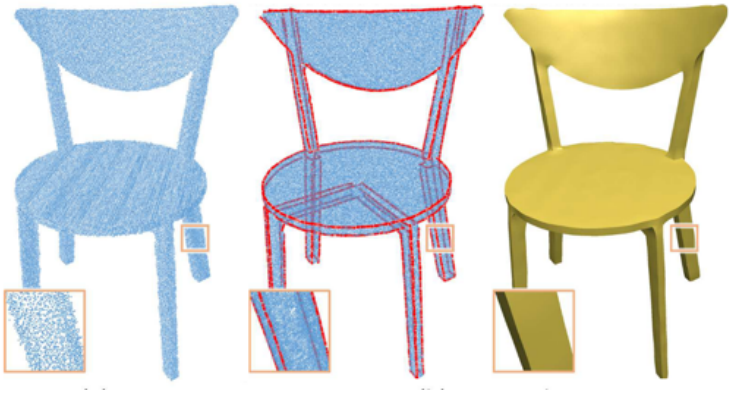


Figure 3.21: EC-Net: Consolidation of a point cloud [52].

Shortly after EC-Net, Wang et al. created another edge detection approach based on the seminal work with PointNet: PIE-NET, a deep neural network for the parameter interference of feature edges in three-dimensional point clouds. PIE-NET aimed to output parametric curves representing

edges within the input cloud [55]. The first phase of their curve inference trains two networks based on PointNet++ to classify both edge and corner points. A non-maximal suppression is then used, and the points are clustered by feature using a third instance of a PointNet++ network. A final two-headed instance of a PointNet is ultimately used to generate a resulting set of curves. As a concatenation of several networks with deep architectures, PIE-NET is also prone to high resource requirements. However, as direct descendants of PointNet, EC-Net and PIE-NET base their classification solely on local features learned through supervised learning based on the training data and do not consider any spatial relationships between the points [55].

More recently, Bazazian and Parés proposed another approach using a neural network, EDC-net, a capsule network that naturally consumes three-dimensional point clouds. Capsule networks are a special type of neural network in which groups of neurons (capsules) act together to store information at the vector level instead of as a scalar [56]. The inherent structure of capsules enables EDC-net to capture intricate patterns and hierarchies present in 3D data, thereby enhancing its interpretability and generalisation capabilities. However, its reliance on capsule networks may incur increased computational complexity compared to traditional architectures, potentially hindering its scalability to larger datasets or real-time applications.

Only very recently has another approach using a CNN been proposed: Himeur et al. proposed, PCEDNet [57]. Their approach is closely related to PIE-NET. However, Himeur et al. use a different parameterisation where they add a set of differential information to each dataset point. The approach builds upon the Growing Least Squares (GLS) methodology, which is, in turn, inspired by the concept of Scale-Space analysis. The parameterisation contains differential information on its surrounding shape reconstructed at different scales. It employs differential properties (such as derivatives) of surfaces to understand geometric structures and scales around the points in the point cloud. A Scale-Space Matrix (SSM) encapsulates these properties across different scales, helping to distinguish between noise and meaningful features. It is fed into their neural network, which learns the description of edges. The focus is on stable points in scale space, where geometric characteristics are consistent, aiding in accurate point-based shape analysis. While the SSM approach introduces a new parameterisation technique, its effectiveness heavily relies on the quality of the Scale-Space Matrices and the accompanying neural network architecture. Designing an effective SSM and network might require significant trial and error to achieve optimal performance.

4 Approach

This section describes two classical approaches to edge detection in point cloud data and a novel neural network-based approach. The two approaches will later serve as performance baselines to evaluate the neural network-based approach's performance.

4.1 Edge detection in point cloud data

Approaches from 2D images cannot directly be applied to 3D point cloud edge detection in point clouds. As was shown in the previous sections and summarised by Ni et al., this is mainly due to the following [14] (also compare Section 3.2):

- The differences in data representation (i.e., an image being a matrix, whereas a 3D-point cloud being an unorganised and irregularly distributed point set).
- The difference in represented information type (i.e., an image containing cryptic spatial information and abundant spectral information, whereas a point cloud contains explicit spatial data and only sometimes the reflected intensities).
- The differences in spatial information (i.e., a grid-like pattern in images, whereas the points are unorganised in 3D point clouds).

4.2 Classical approaches to edge detection in 3D data

Based on the described differences in edge detection in 3D data (see Section 4.1), new approaches were proposed by various authors. Two approaches by Weber et al. and Ni et al. were chosen as respective representatives of the popular direct approaches of normal clustering and angular gap detection.

4.2.1 Surface variation approach on point cloud data

The general approach of feature detection algorithms based on normal vector clustering is fairly simple. A candidate point **p** is selected from the data, and a neighbourhood around **p** is computed. Normal vectors are then computed based on point **p** and some combination of points in the given neighbourhood. Clustering of these normal vectors is consequently used to extract a feature. Clustering of all normal vectors in the neighbourhood into a single cluster indicates the presence of a flat surface around **p**, the presence of no clusters indicates a non-sharp feature (obtuse angle or rounded feature), and the presence of two distinct clusters indicates an edge feature (see figure 4.1, left, middle and right respectively) [58]. Many sharp feature detection approaches have been proposed based on normal vector estimation and clustering. Differences lie mainly in the way the normals are computed. One of the more recent ones has been the approach of Weber et al., which bases the computation of the normals on triangles consisting of a candidate point **p** and two of its neighbours as vertices. In the given neighbourhood of **p**, with its k-nearest neighbours, exist $k \cdot (k-1)$ triangles, i.e., normal vectors. Using Gauss map clustering, where the computed normal vectors are projected onto a unit sphere around **p** and the application of a distance measure between the projected points representing the normal vectors, clusters can be identified [58].

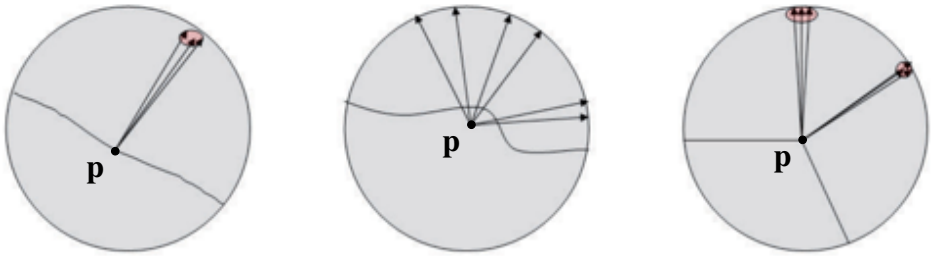


Figure 4.1: 2D examples of clustering of normal vectors on different features: A flat feature on the **left**, an obtuse angle/ round feature in the **middle**, and an edge feature on the **right** [58].

In figure 4.2, a model of a sharp feature is shown on the left. Points on either side of the edge are shown in blue and red. The projected normal vectors are shown as the same coloured points on the Gauss map on the right. Note that the direction of the normal vectors indicates no difference in the feature itself. Normal vectors are, therefore, projected in their original and opposite direction onto the Gauss map. Vectors with little difference in their angles shall now be sorted into clusters. According to Weber et al., it can be shown that within the Gauss map, the geodesic distance between two points on the sphere is equivalent to the angle between the two vectors [58]. Using geodesic distances, a hierarchical agglomerative clustering method condenses normal vectors into larger and larger clusters, initially alone in their respective clusters. As a merging criterion, the mean distance in each cluster is used, as defined in Equation (4.1):

$$D_c(S_1, S_2) = \frac{1}{|S_1| \cdot |S_2|} \sum_{\mathbf{x} \in S_1} \sum_{\mathbf{y} \in S_2} d(\mathbf{x}, \mathbf{y}), \tag{4.1}$$

where S_1 and S_2 are the two clusters, and d is the distance measure for the Gauss map (based on the geodesic distance) [58].

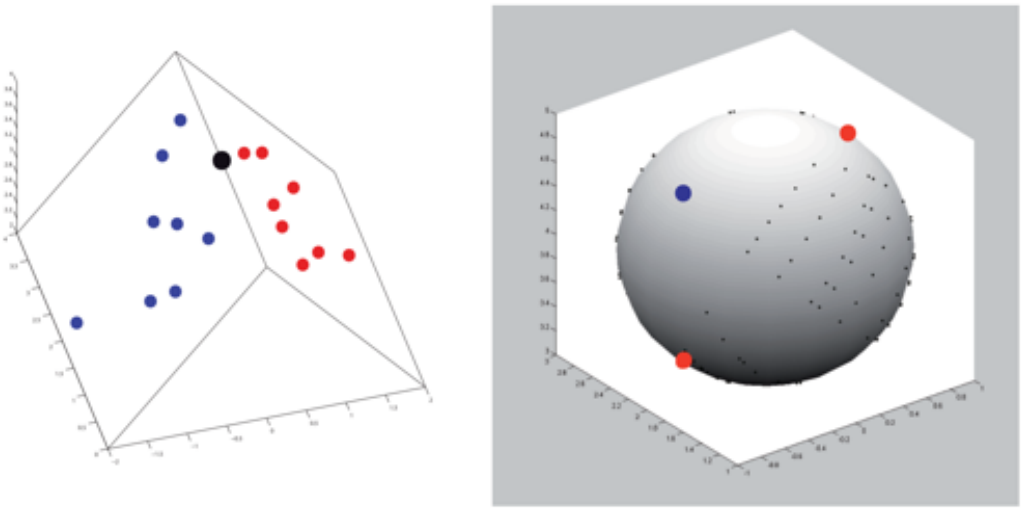


Figure 4.2: Projection of the normal vectors onto the Gauss map in a local neighbourhood [58].

However, as Bazazian et al. have pointed out, the computed normals are based on three points only - the candidate point **p** and two neighbours - the proposed normal computation is therefore highly sensitive to measurement noise. Additionally, if the point **p** and the two neighbours lie on a straight line, no triangle can be formed, and no normal vector exists. Sharp features result in unreliable and inaccurate normals, and for large-scale point clouds, the process is computationally expensive [11].

Based on these observations Bazazian et al. have proposed changes to the existing algorithm introduced by Weber et al.: Their first proposal aimed at replacing the normal estimation method. Where previously the normal estimation was based on a costly triangulation of each point with two of its neighbours, Bazazian et al. propose to replace the process with a much faster PCA-based approach. Each point and its given neighbourhood are fitted with a plane of least squares for this purpose. The normal vector is then computed as the eigenvector associated with the smallest eigenvalue of the covariance matrix **C** [11]:

$$\mathbf{C} = \begin{bmatrix} Cov(x,x) & Cov(x,y) & Cov(x,z) \\ Cov(y,x) & Cov(y,y) & Cov(y,z) \\ Cov(z,x) & Cov(z,y) & Cov(z,z) \end{bmatrix}, \tag{4.2}$$

where **C** is the covariance matrix for a specific point $\mathbf{p}\,(x,y,z)$ in a given point cloud dataset. The components of the 3x3 covariance matrix are given by, for example:

$$Cov(x,y) = \frac{\sum_{i=1}^{k} (x_i - \bar{x})\,(y_i - \bar{y})}{n-1}. \tag{4.3}$$

Bazazian et al.'s second proposal was to change the process from a two-step process, where first normal vectors had to be computed to consecutively be clustered, to a single-step process where the edge features could be extracted directly from the eigenvalues of the covariance matrix through a concept called surface variation introduced by Pauly et al. :

$$\sigma_k(\mathbf{p}) = \frac{\lambda_0}{\lambda_0 + \lambda_1 + \lambda_2}, \tag{4.4}$$

where $\sigma_k(\mathbf{p})$ is the surface variation at point **p** in a neighbourhood of size k and where λ_n are the eigenvalues of the covariance matrix with $\lambda_0 \leq \lambda_1 \leq \lambda_2$ [59]. If all points lie in a plane, the surface variation is minimal $\sigma_k(\mathbf{p}) = 0$. If all points are, however, distributed completely isotropically, the surface variation is at its maximum at $\sigma_k(\mathbf{p}) = 1/3$ [59]. Based on this surface variation, metric edge features are extracted based on the variation of the eigenvalues for each point using a simple threshold.

4.2.2 Angular gap approach on point cloud data

Ni et al. proposed an algorithm for edge detection in 2016 that uses an Analysis of Geometric Properties of Neighborhoods (AGPN) method to classify points in a 3D point cloud. As the name suggests, AGPN uses geometric properties of the neighbourhood of a query point in the spatial domain to use both a RANSAC, a normal optimisation, and finally, an angular gap metric to classify the point.

Let **p** be an unclassified point in the point cloud. First, the neighbourhood around **p** is computed based on the distance to **p**. A plane is now statistically fitted using a RANSAC-algorithm through the points of the neighbourhood. Points within the neighbourhood are subsequently divided into n inliers $\mathbf{p}_i$ with $i \in n$ (i.e., lying within the RANSAC plane) and outliers (i.e., not lying in the plane). If **p** is not part of the inliers, it is classified as a non-edge point. If, however, **p** is part of the inliers, spatial vectors are constructed from **p** to all the inliers. The point is classified as an edge point if a substantial angular gap θ exists between the constructed vectors on the plane (see Figure 4.3).

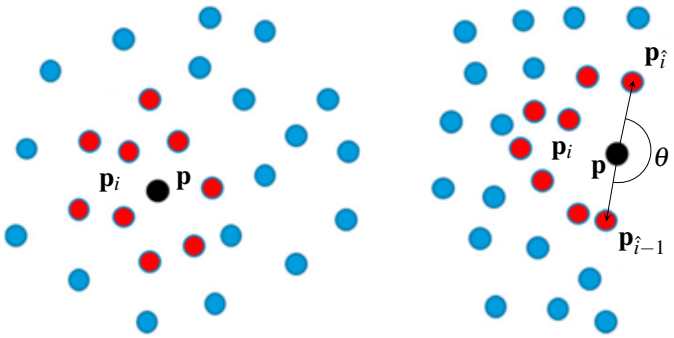


Figure 4.3: Unclassified point **p**, its neighbourhood and the RANSAC-plane inliers $\mathbf{p}_i$ (red) and outliers (blue), vectors between the inliers and the query point and the angular gap θ for an interior point on the **left** and a boundary point on the **right** [14].

4.3 Generalised Convolutional Neural Network in 3D data

Based on the advances in the field of the application of neural networks directly on three-dimensional point clouds (detailed in section 3.6.4), Savchenkov et al. proposed a novel framework using convolution in a traditional sense, but defined over a continuous input space allowing it to process both point clouds and images natively [1].

4.3.1 Theoretical approach to Generalised Convolutional Layers

A traditional convolution uses the points within a certain static kernel and outputs the sum of a function applied once to each point within the kernel. Since the underlying datatype is arranged in a grid-like fashion, and thus each point is an integer $(\Delta x, \Delta y)$ away for which the convolution is evaluated, the applied function

$$f(\Delta x, \Delta y, a_{\Delta x,\Delta y}) = a_{\Delta x,\Delta y} \cdot \theta_{\Delta x,\Delta y} \tag{4.5}$$

is defined piecewise for each possible spatial relationship within the kernel (note that this piecewise function is what is learned during training). Unlike regular grid-like input data (such as images), point clouds represent continuous space (i.e., habe not discretised space and lack a fixed structure) and require adaptations to exploit the convolutional operations - an assumption of integer distances to the query point cannot be made. A continuous function must replace the previously learnable piecewise function. The Generalised Convolution approach aims to extend CNNs to handle point

cloud data effectively [1]. The trick is to not compute the function in itself but to use a small neural network to approximate the new continuous function $f(\Delta x, \Delta y, a_{\Delta x,\Delta y})$ since evaluating a function is much faster than fetching the values of the weights and applying an activation function to them. An activation function is ultimately applied to the sum along the neighbour dimension. Concatenating the computed activations to the absolute spatial coordinates of the input points results in a new point cloud storing information in the same places as the original point cloud. The result is an architecture that consumes point clouds natively (without the need for conversion), produces point clouds in the same format and also closely mirrors conventional CNN layers in that it also utilises a shared filter that is applied to a local neighbourhood [1].

Similarly to CNNs in 2D, multiple of these Generalised Convolutional Layers can be used in succession, learning more complex features as the network progresses. The approach proposed by Savchenkov et al. can be trained on various relative spatial features describing the relationship between a point and its neighbours (we will later use differences in the three spatial dimensions Δx, Δy, Δz and the Euclidean distance). These relative features replace the absolute spatial features used in a 2D application.

4.3.2 Properties of Generalised Convolutional Networks

- Structural information encoding: Generalised Convolution introduces an encoding scheme to represent the local structural information of points in a point cloud. This encoding is derived from the k-nearest neighbours of each point, enabling convolutional operations on point clouds [1].
- Graph-based convolution: The approach treats the point cloud as a graph, with points as nodes and their connections as edges. Convolutional operations are then applied to this graph representation, considering the local structure of points [1].
- Hierarchical feature learning: Generalised Convolution employs a hierarchical feature learning process, where local features are extracted and aggregated at different levels to capture the point cloud's local and global information [1].
- Adapted pooling operations: Generalised Convolution introduces specialised pooling operations to handle point cloud data, such as adaptive maximum pooling, which adapts to the irregularity of the point distribution [1].

4.4 Datasets for training, testing and performance comparison

The neural networks were trained on different input training data. For one, a synthetic training dataset containing edge labels was produced. An extensive publicly available dataset - the ABC dataset - was also used to investigate the influence of the training data further.

4.4.1 Synthetic data

The synthetic training data is mainly based on cuboids and elliptical cylinders as base bodies. These bodies were cut with another cuboid or elliptical cylinder shape to generate different edge geometries on the inside and the outside. The resulting bodies were randomly sheared and rotated

to increase data diversity for edge angles (i.e., sharpness) and edge shapes. Gaussian noise ($\mu = 0$, $\sigma = 0 \ldots 0.5$) was finally applied to each point to make the dataset more diverse and help the NN to become more robust towards inaccuracies or noise in real-world scanning data. A select few examples are depicted in Figure 4.4.

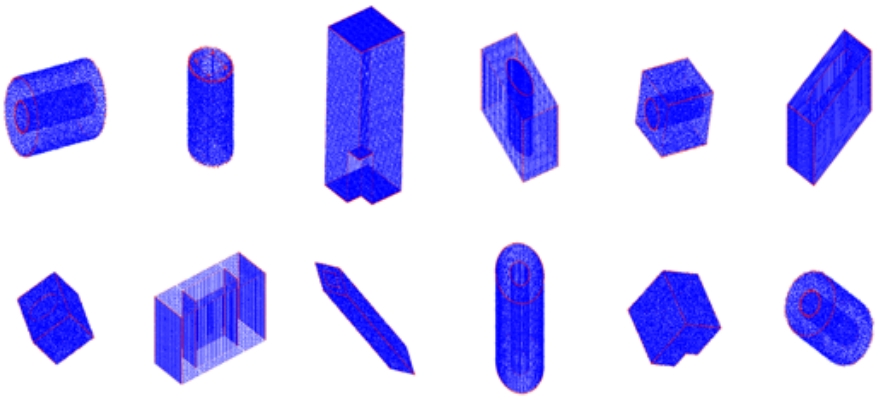


Figure 4.4: Examples of point clouds from the synthetic dataset comprised of different base shapes (cylinders and cubes) with different cutouts (cylindrical and cuboid) and different levels of Gaussian noise applied. Non-edge points are shown in blue, and edge points are shown in red.

4.4.2 The ABC dataset

The ABC-Dataset is a large collection of Computer Aided Design (CAD) data files published by Koch et al. in 2019 to provide an extensive dataset for geometric deep learning applications [60]. The objects within the dataset are taken from an online CAD editor and are thus comprised of diverse objects; see Figure 4.5 for some examples.

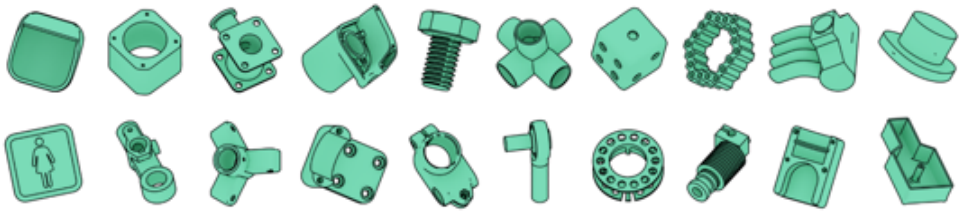


Figure 4.5: Random examples of models from the ABC-dataset [60].

The dataset files are published as parametric surfaces in various file formats (e.g., .step-, .para-, and .obj- files). Each file has an annotative features file containing a list of surface patches and curves. The information about the curves is used to obtain edge information from the parametric file formats, yielding ground truth information for the training of our edge applications. Note that this approach is not ideal (see Section 7.1.3). Additionally, point density in some files is low, so files were densified using the provided triangle meshes (see Section 3.2). Gaussian noise was added to diversify the dataset and prepare the NN for scanner noise. Due to computing power limitations, the training of the Neural Network was conducted only on the first chunk of the ABC dataset (roughly 650 objects with ca. 28 M points).

4.4.3 Real world evaluation data from a 3D laser scanner

Real-world scanning data is used to evaluate the different edge detector implementations. Different objects were scanned using a *Hexagon RS5* laser scanner mounted on a *Hexagon Absolute Arm 7-axis*. The *RS5* is a TOF-scanner (see Section 3.1.3) that uses a scanning range of up to 115 mm width to digitise real-world surfaces and produces point cloud data natively. The object is placed on a surface, and all the faces are scanned from different angles. To obtain a complete model with all faces included, two separate scans are taken with different object placements, and the hidden faces (i.e., the face in contact with the table) are obtained by merging the two respective point clouds using a least squares approach. Finally, the points are manually labelled to obtain edge and non-edge information for each point so that the scan data can be used to evaluate the different edge detectors.

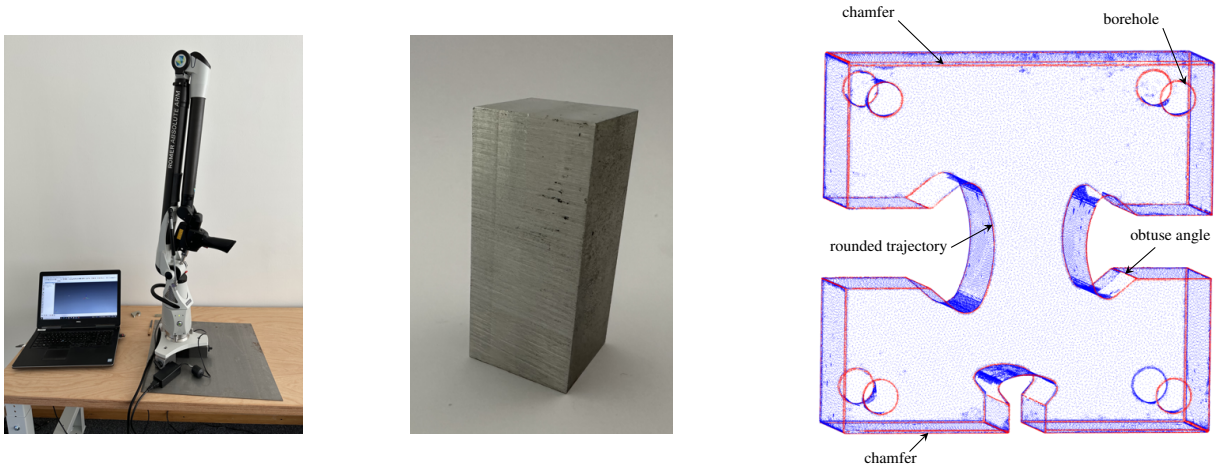


Figure 4.6: The *Hexagon RS5* TOF laser scanner mounted on a *Hexagon Absolute Arm 7-axis* on the **left**, a scanning object in the **middle**, and an already manually labelled scan (edge points in red, non-edge points in grey) in the form of a point cloud and with labels for the distinct features on the **right**.

The scanned shape (see Figure 4.6, right) is ideal for evaluating an edge detector. With its unique shape, similar to that of a jigsaw puzzle piece with only female sides of different sizes, and additionally fitted with boreholes in every corner and chamfers on the top and bottom edges of the front, it provides a multitude of different edge and body geometries. Edge features present in the body include straight edges with rectangular angles (long and short), straight edges with obtuse angles (inside the jigsaw cutouts), rounded edge trajectories with rectangular angles (along jigsaw cutouts, with different curvatures), corners with three meeting edges, round edge trajectories with obtuse angles (sunk boreholes), and parallel edges in close vicinity with obtuse angles (chamfers on the top and bottom edges on the front); also see Figure 4.6 on the right.

4.5 Complexity analysis of the different algorithms

Many different metrics can be used to compare algorithms that perform the same task. The quality of their outputs is the obvious choice, but in many (and most) applications, the time it takes to

achieve a certain result also plays an important role. Since, for most algorithms, the running time depends on the size of the input, theoretical computer science has developed the concept of the complexity of an algorithm, which connects the size of an input with the running time until an output is produced. This section will analyse the complexities of the different edge detection algorithms regarding their maximum running times. The O-notation characterises the upper bound on the asymptotic behaviour of a function, giving an upper limit for the growth of the running time [61].

To compare the edge detection algorithms, we will examine the time complexity of applying them to new data exclusively. This is based on the assumption that in most real-world applications, a neural network can be trained before the algorithm is used and should, therefore, not be considered for the performance evaluation of an application in the situation that it is needed.

4.5.1 Complexity analysis of the Angular Gap Edge Detector

The angular gap edge detector starts with a k-nearest neighbour search. Our implementation uses the Point Cloud Library (PCL)'s KdTreeFLANN to extract the neighbours [62]. Since this is the implementation of a KD-tree (K-dimensional tree) using FLANN, the algorithm is logarithmic because, at each level of the tree, the search space is divided in half based on a chosen splitting dimension until the nearest neighbour is found or a specified termination condition is met (also see Section 3.3.2). The average time complexity of the nearest neighbour search is $O(log(N+k))$, where N is the dataset size and k is the number of neighbours found around each query point. However, in the worst case, such as when the tree is unbalanced, the time complexity might approach O(n), making the performance less efficient. This information is important, but the average time complexity will be considered as we proceed since the number of neighbour searches is large enough to apply the law of large numbers.

The next step after the neighbours have been found is fitting a plane through the neighbours. A RANSAC algorithm is applied, which is an iterative algorithm that randomly selects minimal subsets of the input data to fit a model (in our case, a plane). The quality of the model is evaluated by counting the number of inliers (points that fit the model within a certain threshold). The algorithm, therefore, highly depends on the chosen optimisation criteria. The algorithm's time complexity is thus often represented in terms of the iterations required to achieve a certain probability of finding a suitable model given the input data and the model parameters. Simplified, the time complexity for a RANSAC-based approach is given as $N \cdot O(m \cdot k)$, where m is the number of iterations (can be constant or related to the probability of finding a good model), k is the number of neighbours in the neighbourhood, and N is the total number of data points [63].

The following computation of the vectors connecting the query point with the inliers runs in constant time $O(1)$ but is looped until all inliers have been dealt with, so $O(k)$, where k is the number of neighbours around a point. This is the worst-case scenario, where all neighbours are inliers of the fitted RANSAC plane, which can happen if all the points are located on a flat surface.

Finally, the vectors are sorted, and their angular gaps are computed. Whilst the computation runs in constant time and is repeated for all vector pairs around the query point $O(k)$, the sorting is based on C++'s `std::sort` and does not run in constant time. The Standard Library's implementation typically uses an efficient implementation of the introsort algorithm - a hybrid sorting algorithm that combines quicksort, heapsort, and insertion sort [64]. The average-case time complexity of `std::sort` in C++ is $O(N \cdot log(N))$, where N is the number of elements to be sorted.

This is the same as the average-case time complexity of quicksort, which dominates the performance of std::sort for larger datasets [61]. In our case, this means that we have a maximum time complexity of the sorting algorithm of $O(k \cdot log(k))$, where k is the number of neighbours around each point (which, again, is the worst-case scenario).

The overall time complexity of the Angular Gap Edge Detector is therefore:

$$T_{AG}(N,k,m) \approx N \cdot [O(log(N+k)) + O(m \cdot k) + k \cdot O(k) + O(k \cdot log(k))] \tag{4.6}$$
$$\approx O(N \cdot log(N+k)), \tag{4.7}$$

where N is the overall number of points in the cloud, k is the number of neighbours to be considered, and m is the number of iterations for fitting the RANSAC-plane. The simplification from lines one to two is justified under the presumptions that the dataset size is much larger compared to the size of the neighbourhood ($N \gg k$) and compared to the number of iterations required to fit the plane ($N \gg m$).

4.5.2 Complexity analysis of the Surface Variation Edge Detector

The Surface Variation Edge detector must estimate the normals for each point in the point cloud based on its immediate neighbourhood. Our implementation uses the PCL's normal estimation, which is based on an analysis of the eigenvalues and eigenvectors (essentially a PCA) of a covariance matrix created from the nearest neighbours of the query point [62]. Since the time complexity of the computation steps for the covariance matrix is time constant, the time complexity of the normal estimation mainly depends on the computation of the neighbours, which was already estimated in the previous section (see Section 4.5.1) to be $O(log(N+k))$ where N is the size of the dataset and k is the number of neighbours to be found around each query point. After the normals have been estimated for each point, the principal curvatures are computed. This procedure consists of very similar steps as normal estimation in the sense that the eigenvectors and eigenvalues of the covariance matrix of a neighbourhood are evaluated for each point and its neighbourhood. These computations take place in constant time, and the reasoning follows the same lines as normal estimation. The overall time complexity of the Surface Variation Edge Detector is therefore:

$$T_{SV}(N,k,m) \approx N \cdot [O(log(N+k)) + O(1)] \tag{4.8}$$
$$\approx O(N \cdot log(N+k)), \tag{4.9}$$

where N is the overall number of points in the cloud, and k is the number of neighbours to be considered. The simplification is another time justified under the assumption of a large dataset compared to the size of the considered neighbourhood ($N \gg k$).

4.5.3 Complexity analysis of the Generalised Convolution Edge Detector

Once the gCNN has been trained, the application of the algorithm is fairly simple and consists of the computation of the features, subsequently fed into the neural network. As with the other two edge detection algorithms, the time complexity of the computation of the features is dominated by the extraction of the neighbours. Since the implementation of the gCNN edge detector was done in Python, SciKit's nearest neighbour search and `pyflann` were used instead of the PCL's (which is available for C++ only). However, the SciKit's implementation also uses a k-d tree

neighbour search which is essentially the same as the implementation in the PCL and `pyflann` is a Python wrapper for the C++ implementation of FLANN [65]. The time complexity of the nearest neighbour search is again given by $O(log(N+k))$ where N is the size of the dataset, and k is the number of neighbours found around each query point.

Finally, the features are fed into the pre-trained gCNN layers, where the input matrix (consisting of a query point and its neighbourhood distances within the spatial coordinates and their Euclidean distance) is multiplied by the layer's weight matrix, the biases are added, the activation function is applied, and the results are normalised and reshaped.

The complexity of the bias addition is linear and can be approximated to $O(B_i)$. The activation function is applied element-wise and has a $O(B_i)$ complexity. Operations like normalisation and tensor reshaping are typically linear in complexity and less significant compared to matrix multiplications and activations. The sizes of the matrices determine the matrix multiplication time complexity for each layer's weight matrix and input feature matrix.

If we denote the number of neurons in the input dimension of layer i with A_i and the number of neurons in the output dimension of layer i with B_i, the complexity of the first layers can be computed as:

$$O_1(k) \approx B_0 \cdot A_1 \cdot B_1 \tag{4.10}$$

$$O_2(k) \approx B_1 \cdot A_2 \cdot B_2 \tag{4.11}$$

$$\dots$$

$$O_i(k) \approx B_{i-1} \cdot A_i \cdot B_i. \tag{4.12}$$

Since the layers are applied in succession, their time complexities can be summed. Disregarding the comparably small time complexities for the bias additions, activation functions, normalisations and reshapings, the overall time complexity for the neural network dependent on its number of layers L can be given as:

$$O(N,k) \approx N \cdot \sum_{i=1}^{L} B_{i-1} \cdot A_i \cdot B_i, \tag{4.13}$$

where $B_0 = 4 \cdot k$, k the number of neighbours, and N the size of the dataset. The total time complexity of the generalised convolution edge detector is therefore:

$$T_{gCNN}(N,k,L,A_i,B_i) \approx O(N \cdot log(N+k)) + O(N \cdot \sum_{i=1}^{L} B_{i-1} \cdot A_i \cdot B_i) \tag{4.14}$$

For large datasets and comparably small neural networks, this solution falls back to the solution we also obtained for the classical approaches, which again is dominated by the computation of the neighbours:

$$T_{gCNN}(N,k) \approx O(N \cdot log(N+k)) \tag{4.15}$$

4.5.4 Comparison of the computed complexities

All three algorithms show the same time complexities concerning the size of their inputs - all dominated by the computation time needed to find the neighbourhood around their points. Given the same (size) dataset, the algorithms should take roughly the same computation time. Note that the computation time for training a classifier and the time consumed for parameter optimisation have not been considered.

5 Implementation

This section explains the implementation of the three previously described edge detectors. Relevant algorithms are pictured in flow graphs, and their significant sections are described in detail.

5.1 Implementation of the classical approaches

The classical approaches were implemented in C++ using the PCL [62]. An edge detector class was created to consolidate both edge detectors in a concise framework that defines the basic methods for the Angular Gap Edge Detector and the Surface Variation Edge Detector. The different edge detectors inherit these methods from the parent class. The framework can be built and included as a library, ensuring ease of use.

5.1.1 The Angular Gap Edge Detector

The Angular Gap Edge Detector uses the in Section 4.2.2 described approach to classify a query point by the angular gap between connecting vectors of the query point and all its neighbours lying on a RANSAC-plane.

For this purpose, the algorithm runs through the points in the point cloud (see Figure 5.1), selecting a single query point after another. The k-nearest neighbours or the neighbours found within a search radius around the query point are computed for the point in question, and the RANSAC-plane is then fitted through the query point and its neighbours. If the query point does not lie on the fitted plane, the neighbourhood refinement is triggered, which removes the current inliers of the RANSAC-plane from the neighbours and a new RANSAC-plane is fitted. After a certain amount of iterations, or if no more neighbour points are left and the point is still no inlier to the RANSAC-plane, the query point is classified as a non-edge point. Note that in Figure 5.1, the maximum iterations check is omitted to make it not overload the figure.

However, the angular gap is computed if the query point is contained in the RANSAC-plane at any of the iterations. This is done by fitting vectors between the query point and all the neighbours in the plane, sorting the vectors by their angle to an arbitrary reference to find neighbouring vectors, computing the angular gap between neighbouring vectors and finding the maximum of the angular gap for all the neighbouring vector pairs. Ultimately the query point is classified by a simple threshold applied to the computed angular gap before continuing to the next query point until all the points within a point cloud have been classified.

5.1.2 The Surface Variation Edge Detector

Similarly to the Angular Gap Edge Detector, the Surface Variation Edge Detector uses the in Section 4.2.1 described approach to classify a query point by the surface variation of a point and its neighbourhood.

The algorithm takes the point cloud as an input and computes the normals for all its points based on a set search radius or a number of k nearest neighbours to be found. Based on this neighbourhood, the principal curvatures for all points are computed (see Figure 5.2 on the left). The computation of the principal curvatures is based on the eigenvalues of the Covariance matrix of

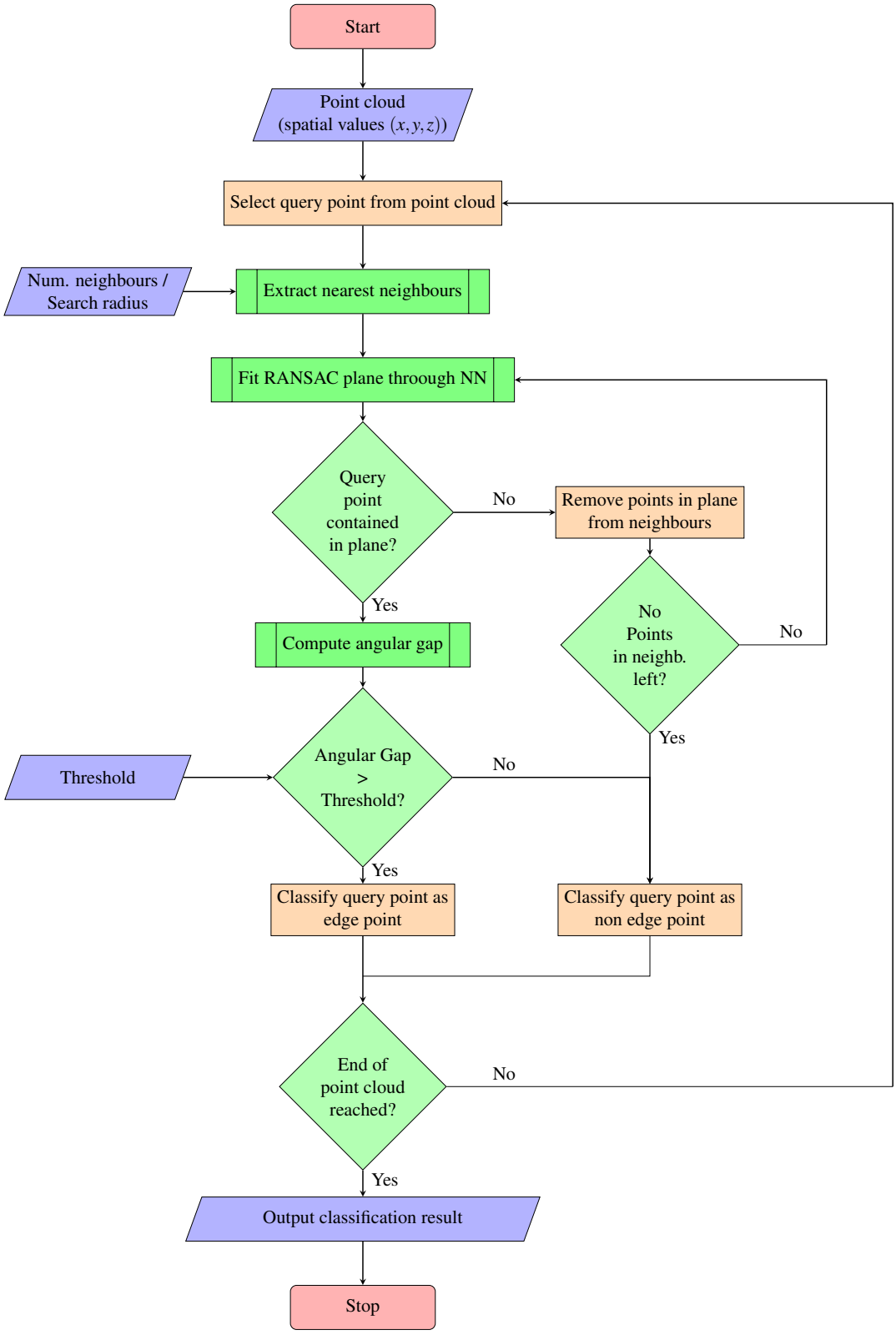


Figure 5.1: Angular Gap Edge Detector with neighbourhood refinement.

a point and its neighbourhood. Thus, the algorithm for the computation of the principal curvatures (see Figure 5.2 on the right) goes through all the points within the point cloud, computes the covariance matrix for each query points' neighbourhood and extracts the eigenvectors and eigenvalues. The surface variation in each query point can then be computed using the eigenvalues and plugging them into Equation 4.4 (see. Section 4.2.1). After the values for the surface variations of all points have been found, a simple threshold can be applied to classify them as edge points or non-edge points.

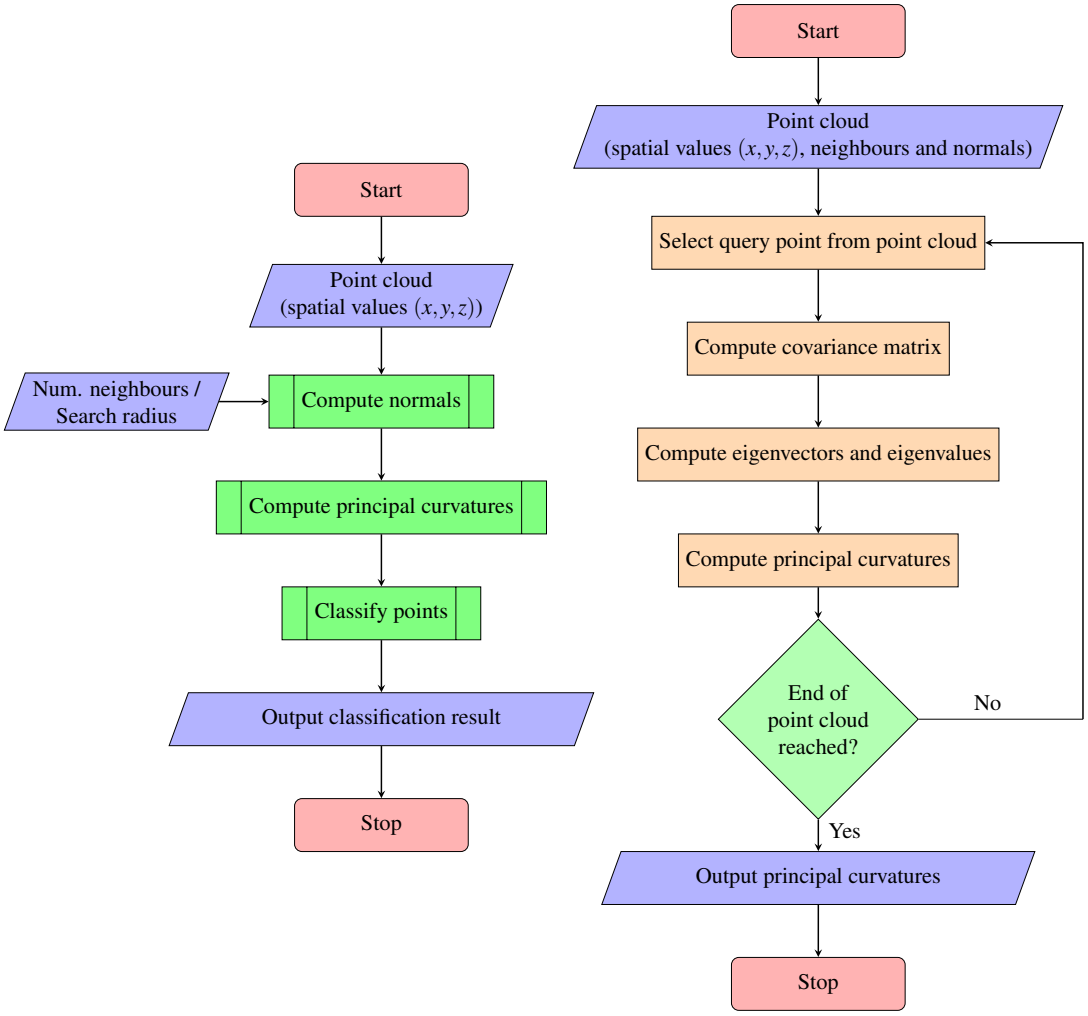


Figure 5.2: Surface variation edge detector module on the **left**, and Principal curvature computation module on the **right**.

5.2 Implementation of the Generalised Convolution Detector

As shown in previous sections, the assignment of the edge property to a certain point depends on the query point itself and its immediate neighbourhood. In 2D applications, CNNs processes a point and its neighbourhood (encoded into their respective positions within the data structure). Similarly to the classical approaches, in the 3D-case, the neighbours must first be found, and

features must be computed before the classification operation of the CNN. Since neighbourhoods and their corresponding features do not change over time, an approach was chosen in which the neighbourhoods and corresponding features are pre-computed before the training is started. This leads to much shorter training times over successive epochs since the computations do not have to be repeated in each epoch. The algorithm was implemented in Python using TensorFlow, falling back on its extensive functionalities to set up machine learning algorithms.

5.2.1 Edge detection module

The edge detection module (see Figure 5.3) is the top-level algorithm for detecting edges in point clouds. Its goal is to process a point cloud input and classify points into edge and non-edge points. Multiple General Convolution Modules are used in succession, each learning new and more complex features based on the activations provided by the previous module. After the desired number of General Convolution modules has been reached, and to get a classification result for a query point, the activations from the final module need to be averaged along the neighbour dimension.

5.2.1.1 Feature extraction

The feature extraction module (see Figure 5.4, on the left) generates the features on which the convolutional neural network bases its classification. Once the input data has been defined, the feature extraction module computes an input pipeline containing all relevant features. Making this module separate is especially useful in the exploration phase of different models and parameter sets since the input features for the neural network are only computed once and can be reused for however many training runs are necessary until a satisfactory parameter set/ architecture is found. To prepare the data for the computation of the input pipeline, the individual point clouds of the dataset are loaded, centred, normalised, and rotated at a random angle. This process ensures that the data input is consistent but yields the maximum possible entropy concerning the positioning of the edges. Spatial features are used to represent the query-point-neighbour relationship. For this purpose, k nearest neighbours are first computed for all the points in the point cloud. Next, each individual query-point-neighbour relationship is represented by computation of the spatial features - the normalised distances along the Cartesian axis (Δx,Δy,Δz), and the Euclidean distance (d) (compare Figure 5.4, left). The point data (x, y, z), the features $(\Delta x, \Delta y, \Delta z, d)$, and if available, the labels (edge point / non-edge point) are finally written to the pipeline. The creation of the pipelines is very similar whether it is to be used for training, testing, or in the classification process for unlabelled data (application). The only exception is that during the creation of the training pipeline, a ratio is applied that limits the number of non-edge points (generally overrepresented in edge detection) to the number of edge points present in the point cloud. This ensures that both classes are represented equally during training and prevents bias.

5.2.1.2 General Convolution Module

The General Convolution Module needs the input of the previously computed features. Those can originate from a spatial feature extractor module and/ or from previous General Convolution Modules (i.e., activations from previous layers performing general convolution). These features are

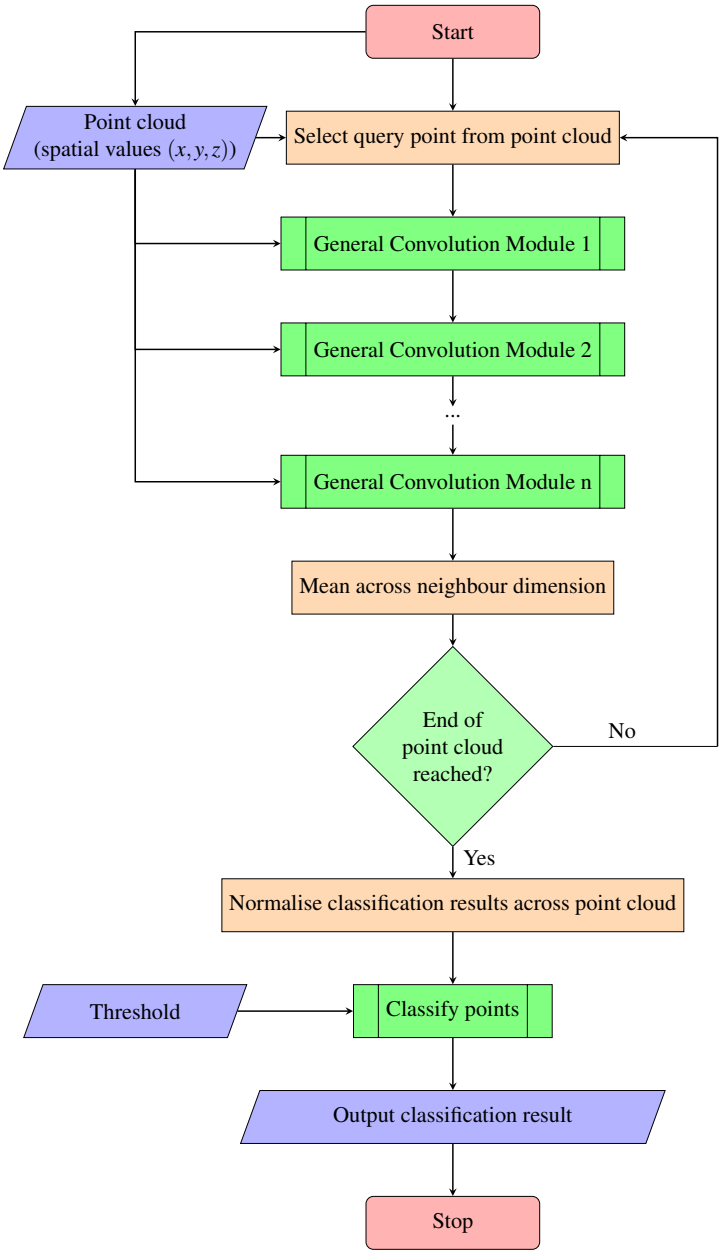


Figure 5.3: Edge detection module

first concatenated to yield a single feature tensor (see Figure 5.4, on the right). A fully connected layer is used to process the data. As described in Section 4.3.1, the neural network's task is to estimate the continuous function equivalent to the kernel in a 2D CNN. A batch normalisation step stabilises the neural network internally and makes it robust against big differences in pre-activations. Finally, an activation function is applied, and the absolute spatial values (x, y, z) are concatenated so that the new data lives in the same place as the initial point cloud.

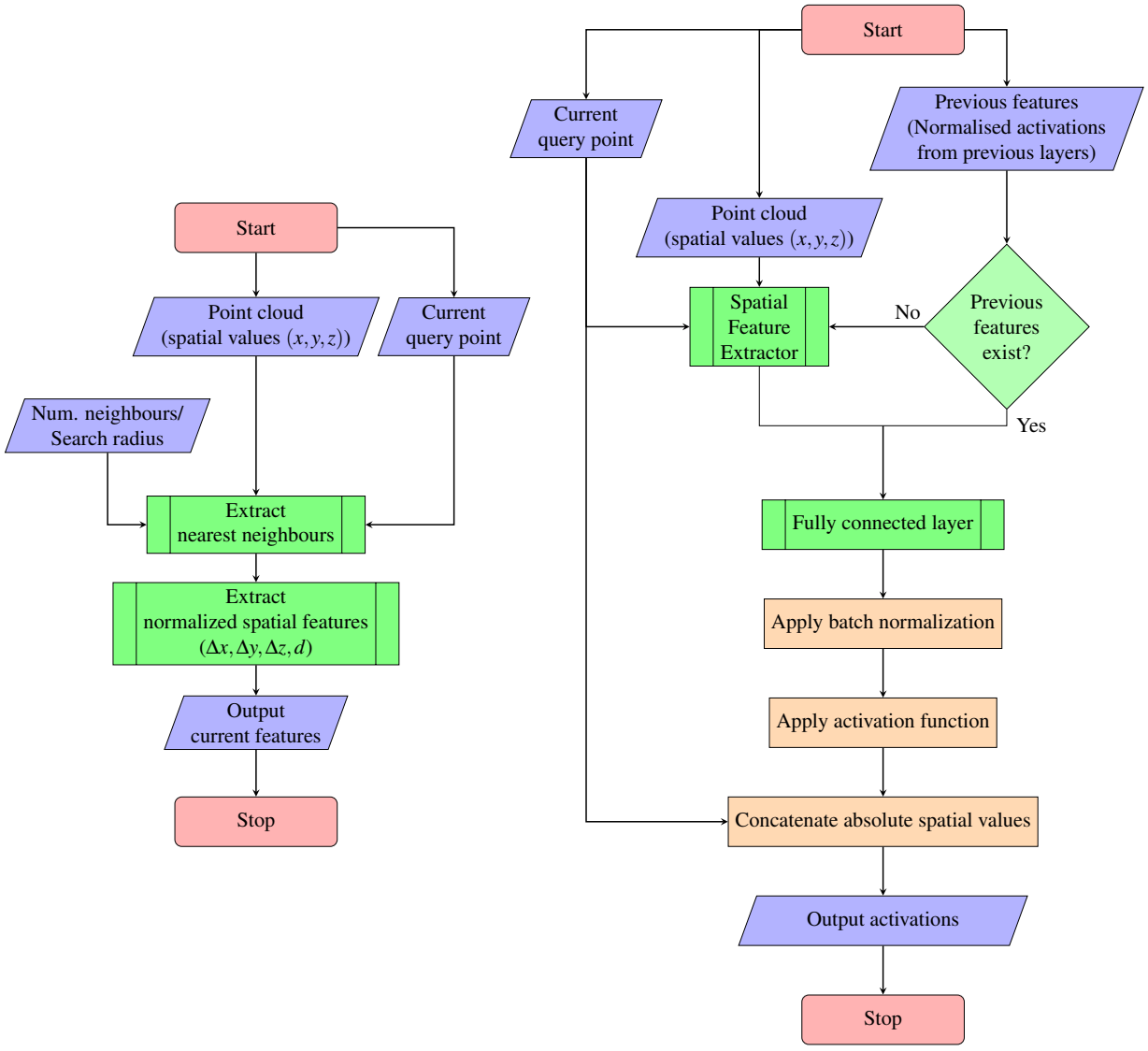


Figure 5.4: Feature extraction module on the **left**, and General Convolution module on the **right**.

5.2.1.3 The General Convolutional layer

The implementation of the generalised convolutional layer is following Savchenkov et al.'s implementation [1]. As already described in Section 4.3.1, this layer differs from traditional layers designed for regular grids (like images). Remember that the key idea is to adapt convolutional operations for point cloud data, where the relationship between points is not fixed as in regular grids. Due to point clouds having irregular distances to neighbouring points, each query point receives different information due to the relative distances between points in the cloud. Standard 1x1 convolutions (pointwise convolutions) are commonly used in CNNs for various reasons, including dimensionality expansion/ reduction, feature extraction, network inception and architecture flexibility and non-linearity and model capacity.

Since for each neighbouring point around the current query point, the convolutional layer receives a multidimensional feature tensor (i.e., in the current implementation, the first layer receives

Δx, Δy, Δz and the distance d) a 1x1 convolution needs to be applied to reduce the dimension. This enables the Generalised Convolution Module to determine which dimensions are useful for the task. However, due to the peculiarities of a point cloud described previously, a fixed kernel size may not capture the inherent structure of the point cloud and might result in unexpected behaviour. Therefore, the 1x1 convolutions cannot be applied before the actual generalised convolution but must be applied independently within the application of the generalised convolution and for each candidate in each query. This means that there cannot be a single weight tensor anymore but a multitude of independent weight tensors for each candidate for each query (i.e., a list of weight tensors).

5.2.2 Training procedure of the Neural Network

All three edge detectors apply a criterion to the input point cloud to classify the points as edge- or non-edge-points. The classical edge detectors have the criterion coded into them (see sections 4.2.2 and 4.2.1). The proposed Generalised Convolution network learns the metric from the data. For application, this means that the classifier needs to be trained on the training data before applying the neural network, meaning extra computation time is needed before application. The training can be performed on different datasets. Before starting the training, the dataset is split into a training and a testing portion (generally, two-thirds are used for training, leaving one-third for testing). The training data teaches the neural network relevant features by changing the weights and biases, and the testing data is used to scrutinize the neural network with unknown data and prevent overfitting. Precomputed pipelines are used as inputs (see Section 5.2.1.1). Since most of the preparation work has already been done in the computation of the input pipelines, the machine can focus on the actual training/testing of the neural network during the training run. For this purpose, the model is first set up according to the chosen architecture and the network weights and biases are initialised randomly. During training, the algorithm uses backpropagation (see Section 3.5.4) to change the weights and biases of the neural network based on the associated loss of the outcome of the current forward pass compared with the desired outcome defined by the pre-labelled input data, where the rate of change is determined by the set learning rate (which decays over time). For this purpose, the network is fed a single point at a time (i.e., batch size is 1), so the number of iterations is equal to the number of points in the training set. After a whole point cloud has been passed through the network, the average performance metrics (cross-entropy, ROC-AUC, F1-score) are computed for the file and stored for later use. After a whole training set (i.e., an epoch) has been passed through the network, the testing set is used to evaluate the past training epoch with only a forward pass and no backpropagation, i.e., no change of the weights and biases. Testing is important to detect overfitting, where the model is fitted to the intricacies and noise of the dataset but can not generalise well anymore. The testing algorithm also computes performance metrics and stores them for later use. At the end of a training run, the performance metrics are evaluated to determine which epoch model yielded the best performance and shall be used as an edge detector for the application.

5.2.2.1 Choice of loss function

The goal of the CNN is to process the point and its neighbours in a way that yields a binary output - namely 0 for it not lying on an edge and 1 for it being an edge point (or vice versa). The edge classification is a binary problem with only two viable outputs. CNNs use stochastic

gradient descent to optimise and learn the classification objective and close in on the preferred result. The learning of the network is based on a mathematical representation of the objective - the loss function. Loss functions are derived from the underlying distribution of the labels; for binary cases, the labels are distributed under a Bernoulli- distribution. The most popular loss function based on the Bernoulli distribution is the Binary Cross Entropy (BCE) loss function [66]:

$$L_{BCE}(y,\hat{y}) = -\left(y \cdot log\left(\hat{y}\right) + (1-y)\, log\left(1-\hat{y}\right)\right), \tag{5.1}$$

where L is the Loss for a certain classification result, y is the value assigned to the current output (i.e., classification result), and $\hat{y}$ is the correct output (i.e., ground truth label). The BCE can be modified by a factor β (called a punishment factor) to accommodate for skewed data or emphasis on the detection of a certain class, yielding the Weighted Binary Cross Entropy (WCE)- loss function [66]:

$$L_{WCE}(y,\hat{y}) = -\left(\beta \cdot y \cdot log\left(\hat{y}\right) + (1-y)\, log\left(1-\hat{y}\right)\right). \tag{5.2}$$

Note that for $\beta = 1$ L_{WCE} simplifies into the L_{BCE}.

For the edge detection problem using the generalised convolutional neural network, the WCE was used to be able to punish not finding an edge point harder than yielding a false positive.

5.2.2.2 Choice of activation functions

The leaky ReLu activation function is very well suited for many machine learning applications, including CNNs. It's characteristics of mitigation of the vanishing gradient problem and fast convergence make it a popular choice. The leaky ReLu function prevents the vanishing of the gradient in backpropagation applications by always allowing a small gradient to flow through the neurons with negative inputs, enabling the network to learn even when some units are inactive and making it prone to faster convergence during training (see Section 3.5.5). For these reasons, the leaky ReLu activation function was chosen for all but the final layer of the neural network.

A sigmoid activation function is used for the final layer of the neural network. Since the edge detection algorithm aims to make a binary prediction (edge vs. non-edge point), the sigmoid activation function is beneficial, yielding outputs in the range $(0..1)$, which can be interpreted as probabilities.

6 Results

This section justifies the choice of performance metrics used to compare and evaluate the different edge detectors previously implemented. The implemented edge detectors' achieved results are consecutively described after introducing the chosen hyperparameters for clarity. Distinct points are mentioned while analyzing the progression of chosen performance metrics throughout the whole training run. Finally, computation times are compared and discussed in the context of the different detectors and their implementations.

6.1 Choice of performance metrics

It was shown in Section 3.4.2 that an array of various metrics can be used to evaluate the performance of an edge detector. The proposed neural network-based edge detector assigns a normalised probability ($[0..1]$) to each point to indicate whether it is an edge or a non-edge point. Classification results (i.e., crisp class labels) are acquired from the probabilities by thresholding the values. However, the choice of the threshold value mainly depends on constraints put forward by the desired application of the edge detector. The threshold value is ultimately chosen as a trade-off between false positives or false negatives and their associated costs on the pretext of the desired application of the classification. Different performance measures must be considered to propose and evaluate different edge detectors without inferring an application.

The AUC under the ROC curve is a widely used performance metric. The AUC's biggest advantage is that it provides a holistic view of a classifier's performance across different decision thresholds, helping to assess its ability to discriminate between the two classes effectively (since it evaluates how well a model can distinguish between true positive and true negative cases). In this respect, it is also the most useful measure to compare the performance of different classifiers or models. It can help directly compare their ability to separate the positive and negative classes, making it particularly valuable in model selection. The AUC under the ROC curve is preferable for edge detectors over the AUC under the PR curve since it is less sensitive to class imbalance (which is very pronounced in most edge detection data sets). The AUC's lack of interpretability is less concerning since it is primarily used as a comparative metric.

The other chosen performance metric, the F1-score, on the other hand, requires crisp labels and is thus not threshold agnostic. To leave the application open and not infer too much about the chosen threshold, we are choosing the threshold based on the maximally achievable F1 score, thus maximising it. Its ability to optimise both precision (*Are detected edges genuine?*) and recall (*Are most actual edges detected?*) within a single metric and its freedom from a distinct preference for false positives and false negatives make it ideal for a broad spectrum of applications and the general comparison of different edge detectors.

Both metrics are used to ensure maximum comparability whilst reliably evaluating the performance of the edge detectors.

6.2 Training of the Neural Network

The training of the NN was performed on a *Macbook Pro 2017* with a 3.5 GHz *Intel Core i7* CPU, *Intel Iris Plus Graphics 650* 1536 MB GPU and 16 GB of 2133 MHz *LPDDR3* memory.

6.2.1 Influence of hyperparameters

The proposed neural network was trained with a choice of different hyperparameters. The different training runs' parameters are shown in Table 6.1. The parameters that can be changed to influence the neural network's training include the chosen input data, the overall architecture (i.e., the number of layers), the learning rate, its decay (and when the decay begins), the punishment factor, and the number of epochs for training and testing.

The training was performed on different datasets that vary in size, composition, and diversity (compare Section 4.4). However, since training results were very similar for both the synthetic data and the ABC datasets, the ABC dataset was chosen as a valid representative of the two, because it leaves better scalability for future extensions of the training algorithm.

The architecture can be modified by adjusting the number of General Convolution Modules applied in sequence (compare Figure 5.3). Since the activations from one layer are fed into the inputs of the subsequent layer, the presumption is that with greater depth, more complex features are learned from the data.

Another parameter that was changed during training is the punishment factor (note Equation 5.2 and Section 5.2.2.1). Applying the factor is a way to weigh errors differently during training. The term that punishes the miss (false negative) of an edge point is multiplied by the punishment factor during the computation of the loss after each forward propagation of the training input. Therefore, a punishment factor greater than one disproportionately punishes the miss of an edge point.

Changes in the various parameters and their resulting testing scores are shown in Table 6.1. Note that, the number of considered neighbours remained constant over the training runs. Two different edge detection algorithms have been tried, namely `pyflann`, a Python wrapper around the C++ implementation of FLANN (see Section 3.3.2, and Scikit's nearest neighbour search, which is entirely implemented in Python. The quality of the neighbours is comparable; however, `pyflann` is an order of magnitude faster when applied to the scanner data. Figure 6.1 depicts a point and its surrounding 30 neighbours for exemplary objects taken from the different datasets. From the visualisation, it is obvious that the neighbourhood is adequately represented and provides a solid basis on which the classifier can inform its classification.

	Data set	Number of neighbours	Number of Layers	Learning rate	Decay rate	Decay starting epoch	Punishment factor	Number of completed epochs	Best average testing ROC-AUC	Best average testing F1-score
1	ABC	30	4	1.00E-05	0.96	5	1	20	0.874	0.579
2	ABC	30	2	1.00E-05	0.96	5	1	20	0.804	0.419
3	ABC	30	3	1.00E-05	0.96	5	1	20	0.844	0.527
4	ABC	30	4	5.00E-06	0.96	5	1	20	0.834	0.456
5	ABC	30	4	1.00E-05	0.96	5	10	18	**0.879**	0.546
6	ABC	30	4	1.00E-05	0.9	5	10	20	0.834	0.406
7	ABC	30	4	1.00E-05	0.9	3	10	20	0.848	0.434
8	ABC	30	4	1.00E-05	0.96	5	1	40	0.877	**0.598**

Table 6.1: Comparison of best average performance metric scores over a whole testing epoch to different hyperparameters used during training on the ABC dataset.

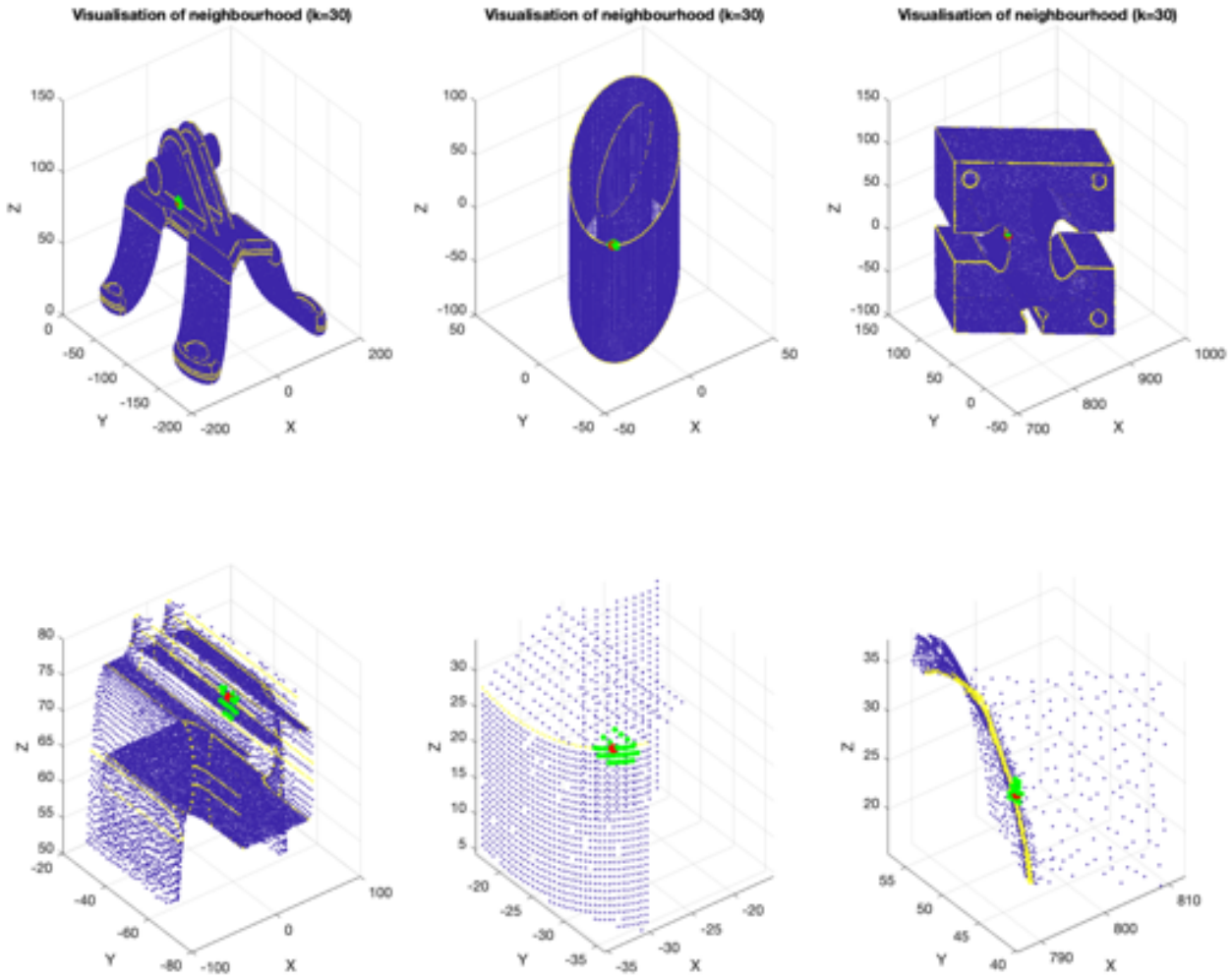


Figure 6.1: Visualisation of the detected neighbourhood (k=30) around an example query point. The query point is shown in red, associated neighbours in green, other points in blue, and, for perspective reasons, edge points in yellow. Example objects were taken from the ABC dataset on the **left**, the synthetic dataset in the **middle**, and the scanner data object on the **right**.

6.2.2 Description of the training progress

A detailed comparison of the influence of the different hyperparameters on the training of the Neural Network over the whole training run and the trends of the different performance metrics is shown in Figure 6.2. Many of the higher performance metric scores were achieved in later epochs, and some of the trends were still increasing, which is why a longer training (40 instead of 20 epochs) was performed for parameter set 8. The results are depicted separately in Figure 6.3. It is important to note that a moving average has been applied to all graphs for smoothing. Testing starts in epoch zero with a randomly initialised classifier to establish a benchmark and to enable the assessment of training improvements of the first epoch.

From the uppermost graph in both Figures 6.2 and 6.3, it is obvious that the backpropagation algorithm effectively reduces the entropy of the training dataset over the training epochs. All depicted trends are decreasing and show convergence towards their respective limits - the algorithm

changes the weights and biases to decrease the computed loss. Note that some trends (namely 5, 6, and 7 in Figure 6.2) decrease faster. This is because, in those training runs, a punishment factor punishes missing edge points disproportionally; the algorithm seems to compensate for this quite quickly, which is, however, not evident in the performance metrics and does not, therefore, be beneficial to the overall performance.

It is evident from both figures that the decrease is not smooth but oscillates around the trendline. Training a machine learning model can sometimes result in oscillations in performance measures, which can be caused by various factors such as learning rate fluctuations, overfitting, model complexity, and data variability. However, even under stable learning rates and non-overfitting conditions and occurring across multiple training runs and periodically across all epochs, suppose these oscillations persist (as in the case in the shown situation). In that case, the intrinsic complexity fluctuations of the training data itself likely cause the variations.

From the fifth epoch, the learning rate decay starts (except for trend 7, whose decay begins in epoch 3 already), slowly decreasing the learning rate. This seems to positively influence development because the trends keep decreasing after the decay is introduced and long after. The development of the two different chosen performance metrics (see Section 6.1) over the training epochs is shown in the two bottom graphs of Figures 6.2 and 6.3. The results improve significantly from a randomly initialised classifier (epoch 0), especially over the first epoch. Starting from randomly initialised values in epoch 0, the AUC improves for all classifiers over time (with some noise) but at a decreasing rate. The model's ability to distinguish between different classes is improving gradually. However, overfitting can be observed in the last two to three training episodes when the punishment factor is applied. This is manifested by a decrease in the entropy of the training data and a decline in the performance of the model on testing data. It is worth noting that training run 5 ended prematurely after the completion of epoch 18.

The F1 score results provide a more mixed picture: During the first six or seven epochs, all parameter sets show similar scores, but the trends diverge later on. Some trends display a significant and rapid (but locally limited) decline in F1 score development at different stages during the training process. Epochs seven and eight show such behaviour in trends 5, 6, and 7. Trends 6 and 7 in epoch 12 also exhibit similar behaviour, as do trends 1, 3, and 5 in epoch 16. This drop occurs despite the training entropies continuing to decrease over the same period. These drops indicate one of two possibilities: overfitting or effective utilisation of regularisation. Overfitting occurs when the model memorizes the training dataset's intricacies, including noise, making it less effective at generalizing to new data. The declining F1 scores may also indicate that the implemented regularisation works effectively. During the training of the model, a dropout is applied. A fraction of the neurons are set to zero. This means that a random subset of neurons is ignored during each training iteration, forcing the network to learn redundant data representations. The initial drop in the F1 score may be due to regularisation: dropout effectively regularises the model, causing it to generalise better but initially perform worse on training data. The fact that the F1 score bounces back a few epochs later supports this conclusion: Over time, as the model learns more robust features, it becomes better at generalising to unseen data. This improvement in generalisation is reflected in the rebound of the F1 score.

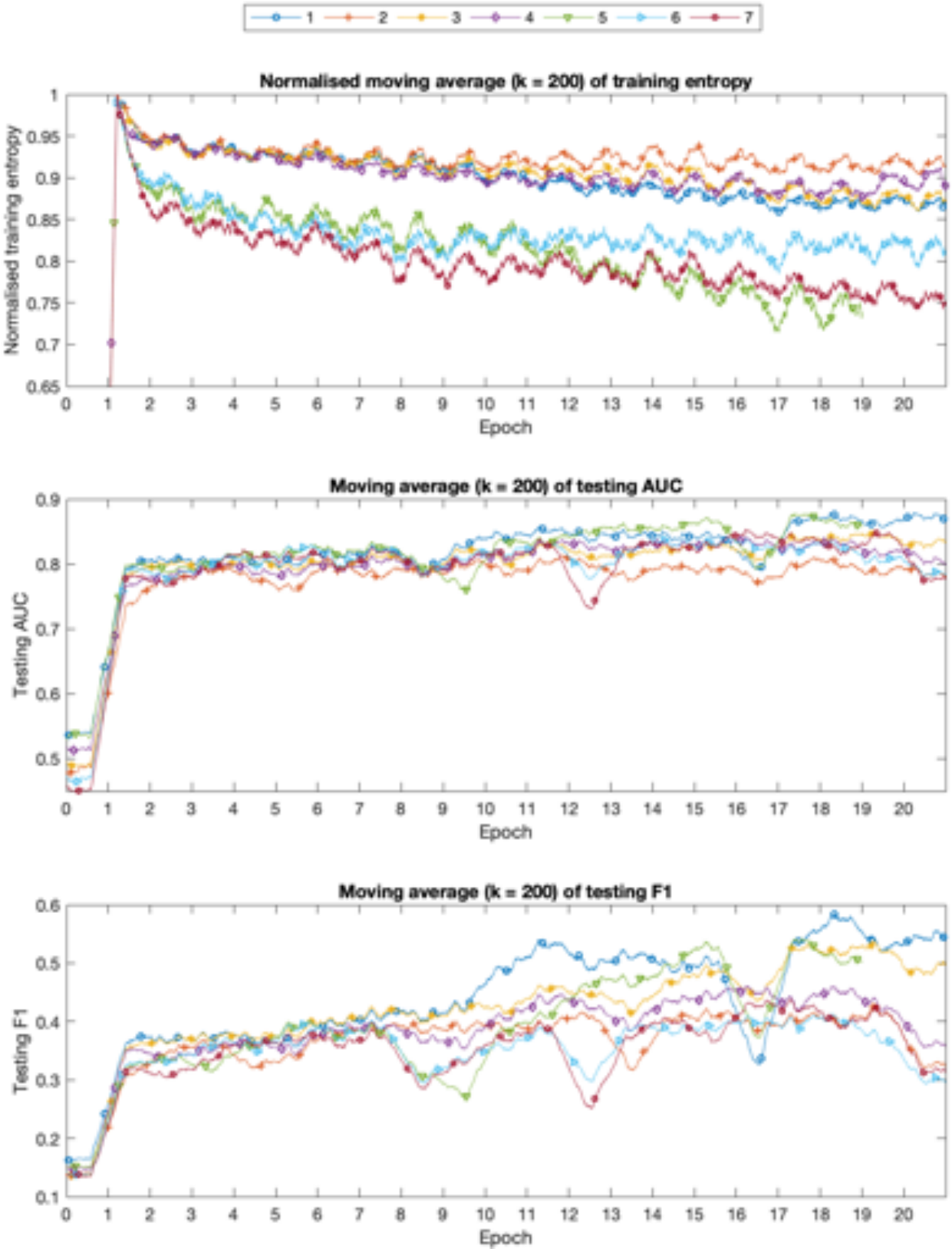


Figure 6.2: Comparison of the moving averages over the normalised training entropies (**top**), testing AUCs (**middle**) and testing F1s (**bottom**) during training with different parameters. See Table 6.1 for details on the used parameters.

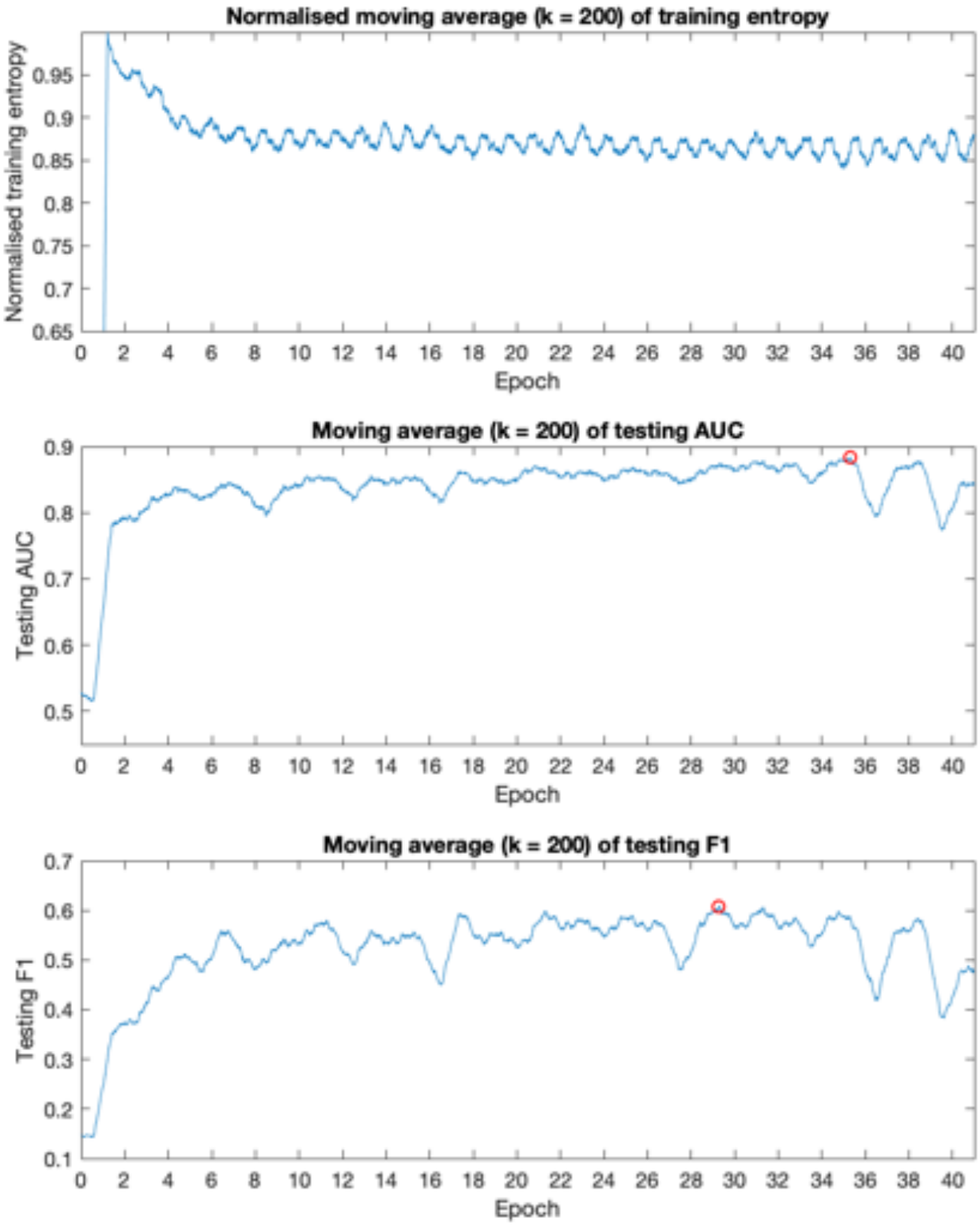


Figure 6.3: Moving average over the normalised training entropies (**top**), testing AUC (**middle**) and testing F1 (**bottom**) for training run 8(40 epochs). The highest achieved values for the averages of AUC and F1 are marked with a red circle. For details on the parameters, see entry 8 in Table 6.1.

6.3 Computation times

Computation times are given for the training of the gCNN and the evaluation times of the different edge detectors on the scanner data. Note that all absolute times refer to computation on the described machine in the beginning of the section.

6.3.1 Computation times for the training

The average training times are shown in table 6.2. The table compares the training times for testing and training of the NN on the different datasets used (ABC and synthetic, see Section 4.4). Computation times are also given per point for easier comparison because the datasets have different sizes. The training times per point are almost identical for both the synthetic and the ABC datasets. This is expected since the complexity of the features and, thus, the complexity of the computation is the same.

When comparing the computation times of the training with the computation times of the testing, however, it is surprising that training times per point are much larger (more than double) as compared to the training, even though the complexity of the computation is much lower - only a forward pass is executed, whereas during training the forward pass and the backpropagation for gradient computation and parameter updates are executed.

Differences in implementation, data loading and preprocessing, hardware utilisation, model size and complexity, and memory usage and caching could cause testing to be slower than training. Since training and testing loops are implemented the same, data is presented with precomputed features equivalent to testing, and the model size and complexity do not differ; the most probable reasons lie in hardware utilisation and memory usage. One reason for the elevated testing times could be that memory usage is much larger because whole files are tested, whereas, during training, only subsets of files are presented to the NN (with as many non-edge points as edge points). Dealing with large amounts of data could lead to the machine running out of working memory. If the machine's memory is full, memory swapping is executed, which extends the machine's physical memory with virtual memory on the hard drive by extending the RAM address space to the hard disk - a time and resource-consuming process - slowing down the computation.

	Dataset	Mode	Number of points per epoch	Computation time per epoch [hh:mm]	Computation time per point [ms]	Normalised computation time
1	Synthetic data	Training	1.7 M	01:10	2.47	0.39
		Testing	7.7 M	13:11	6.16	0.98
2	ABC	Training	3.5 M	02:25	2.48	0.38
		Testing	9.9 M	17:20	6.30	1.00

Table 6.2: Comparison of computation times per dataset and mode. Note: The compared times are all based on a model using 4 layers and considering 30 nearest neighbours.

6.3.2 Computation times for the application of the different detectors

The application times for the different edge detectors on the scanner data are shown in Table 6.3. Next to the absolute data for the computation on the previously described machine are also given as normalised values. Both classical edge detectors show similar computation times. This is expected since we established in Section 4.5 that all three edge detectors have the same time complexity.

Differences between the two classical approaches are minor and can be attributed to differences in the efficiency of the implementation - both implementations leave room for optimisation.

Surprisingly, the NN is much slower than the other two edge detectors, even though all three should have the same time complexity. Some of the differences between the two software systems can be attributed to differences in their implementation. The NN precomputes the features, which are then saved to a pipeline. During the classification process, the pipeline is loaded back into the algorithm, which adds an extra step to the computation and slows down the process. This additional step contributes to some extra time required for the algorithm to complete its task.

Additionally, the CNN has been implemented in Python, whereas the classical approaches have been implemented in C++. Computation times for different programming languages are difficult to compare because languages handle problems differently and deal with different overheads. High-quality studies are lacking, but simple comparisons show that C++ is substantially faster than Python (depending on the compared algorithms C++ is up to one order of magnitude faster) [67].

	Detector type	**Best evaluation ROC-AUC**	**Best evaluation F1-score**	**Computation time [s]**	**Computation time per point [ms]**	**Normalised computation time**
1	Angular gap	0.563	0.441	105	0.54	0.24
2	gCNN	0.608	0.443	430	2.21	1.00
3	Surface variation	**0.696**	**0.562**	38	0.19	0.09

Table 6.3: Comparison of best performance metric scores for different edge detectors over the evaluation data and their associated computation times.

7 Discussion

This section analyses sample files from the best-achieved training runs of the gCNN and compares them to the classical edge detectors. We will also evaluate the real-world scanner data classification results based on the evaluation shape's edge features.

7.1 Discussion of ABC testing results

7.1.1 Results of the Generalised Convolution Edge detector on the ABC data

Exemplary objects from the testing of the Neural Network are shown in Figures 7.1 and 7.2. Based on their complexity, the classifier achieves different results. The object in the top-left corner of Figure 7.1 has excellent classification results with an AUC of 98 and an F1 score of 81. The shape analysis reveals why edge detection works well on this object; all edges have a right angle and are not too short, and all body parts are thick enough to define a large enough neighbourhood for the edge detector to produce clear results. The object also contains no rounded edges or obtuse angles.

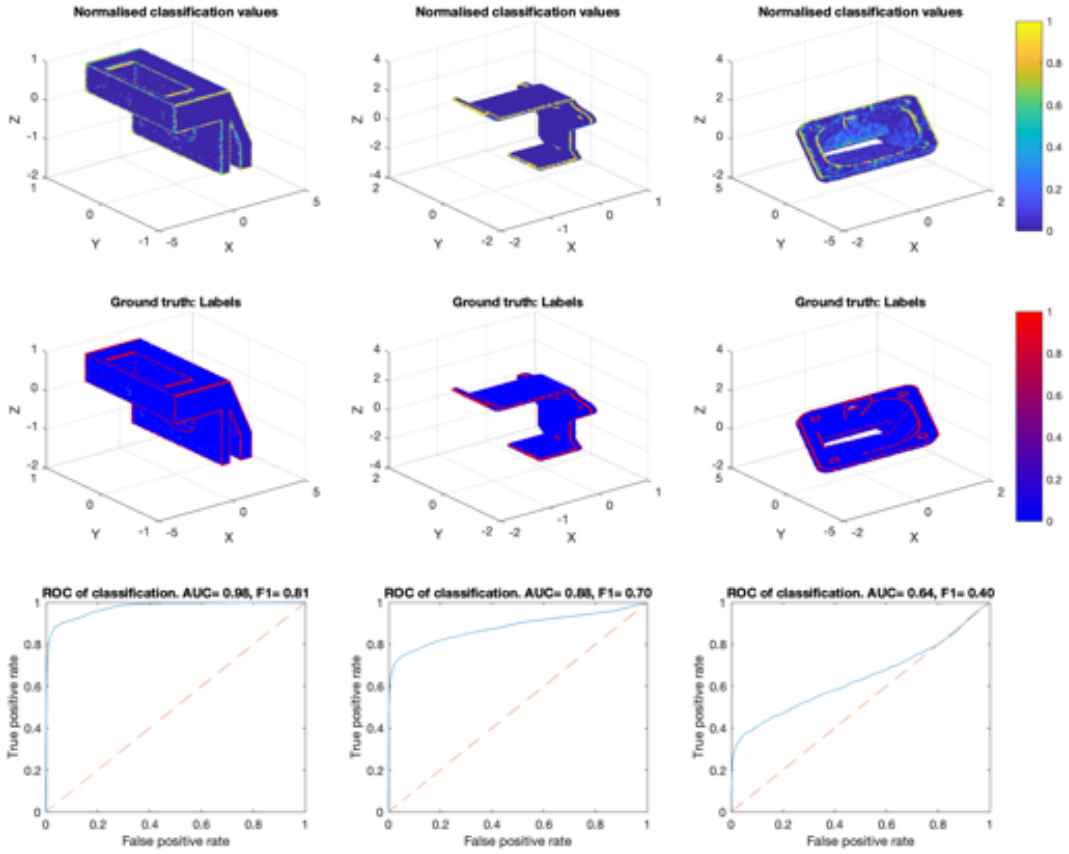


Figure 7.1: Exemplary objects from testing using the ABC dataset. Normalised classification results are shown in the **top row**, ground truths in the **middle row**) and testing ROC in the **bottom row**. All results are taken from the testing results of training run number 8 (refer to Table 6.1 for the parameters used).

When classifying more complex objects with rounded edges, multiple edges close to one another, changeable/ round edge contours, or slim object parts, such as the two objects in the top row on the right, the classifier achieves less accurate results. Edges in close proximity to one another (e.g., the central bottom edge of the top right object) and rounded contours (like the corner hole contours of the top right object) seem difficult for the detector and are not correctly classified.

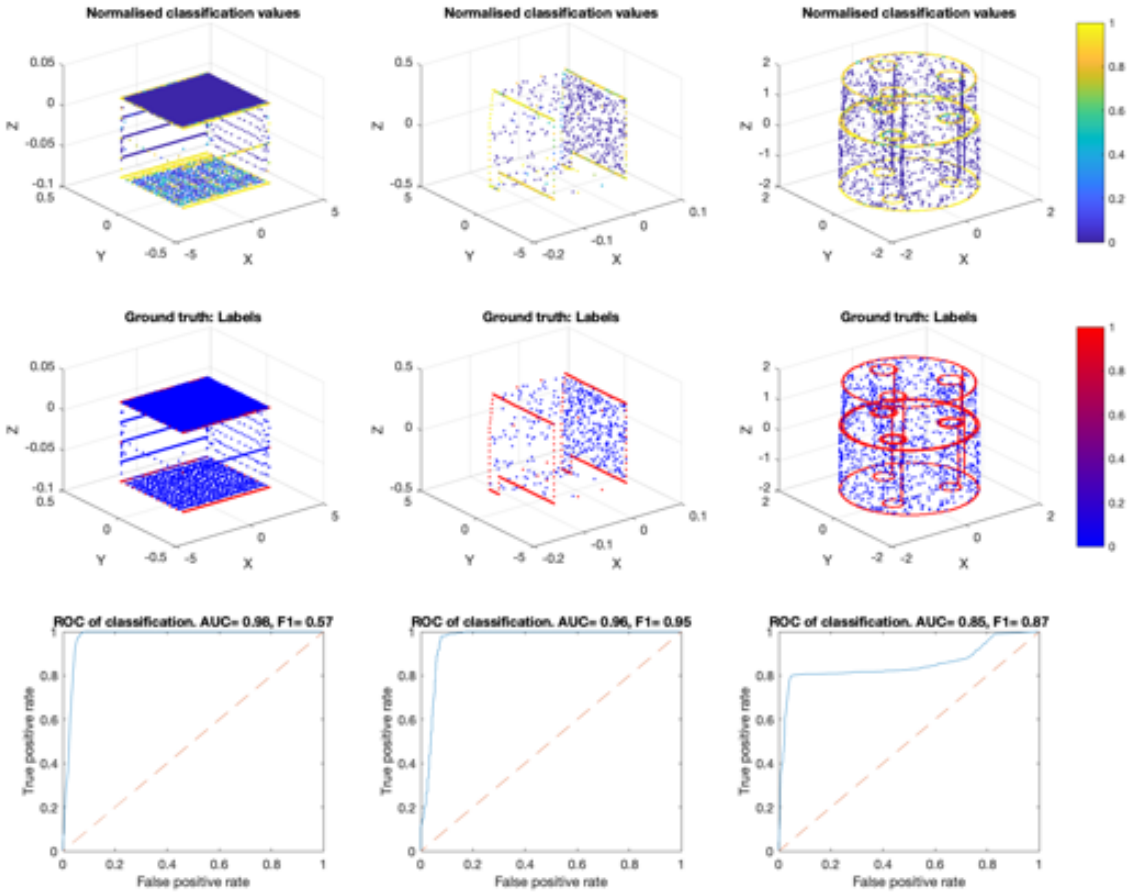


Figure 7.2: Exemplary objects from testing of a training run using the ABC dataset. Their respective normalised classification results are shown in the **top row**, their ground truths in the**middle row**) and their testing ROC in the **bottom row**. All results are taken from the testing results of training run number 8 (refer to Table 6.1 for the parameters used).

More objects are depicted in Figure 7.2. This time, objects that exhibit low point densities were chosen. The ABC dataset contains a significant number of objects that have a similar point density. The classifier does not seem influenced by the low density and achieves good classification results on most of these objects despite the low density.

Apart from the object at the top right of Figure 7.1, the ROC-plots support the overall good performance of the detector on the testing data. In all shown ROC plots (except the object in the top right corner), the classifier achieves AUC-scores of more than 0.85. AUC scores greater than 0.8 indicate a strong discrimination ability of the classifier. It suggests that the classifier is highly effective in distinguishing between positive and negative instances across various thresholds. However, the object in the top right corner shows suboptimal classification results, with an

AUC of only 0.64. This supports the visual analysis of the results and suggests that the classifier's ability to discriminate between positive and negative instances in this object is moderate, meaning there is only some ability to distinguish between the classes.

The classifier achieves a high balance between Precision and Recall on some objects (top left, bottom middle, and bottom right) exhibiting F1 scores greater than 0.8, meaning that the classifier correctly identifies positive instances (high Recall) and minimises false positives (high Precision). In one of the cases, the classifier is doing reasonably well with an F1 score of 0.7 (top middle). This means it balances Precision and Recall moderately well for the data related to this particular object. However, there is room for improvement in identifying positive instances and minimizing false positives for other objects (bottom left and top right). This suggests that the classifier is struggling to accurately identify positive instances in these cases, which can be ascribed to the classifier's struggle with the previously discussed features.

The object in the bottom left corner shows the challenges with the different performance metrics and why evaluating more than one simultaneously is advisable. The object shows a very high AUC of 0.98 paired with a moderate F1 of only 0.57. If only the AUC was considered, the classification's performance on the object was overstated. The discrepancy comes from the bottom of the object, where some rough surface structure confuses the detector. The high AUC indicates that the classifier is good at discriminating between the two classes. The F1, however, tells us that simultaneously, the classifier scores poorly in Precision (proportion of true positive predictions among all positive predictions), Recall (proportion of true positive predictions among all actual positives), or both. Because of a large class imbalance (the object contains many more non-edge than edge points), the classifier correctly identifies most points. Still, it performs poorly on the F1 because it fails to classify instances of the minority class correctly, yielding poor Precision due to many false positives.

7.1.2 Discussion of the results of the classical edge detectors on the ABC data

The results of the Angular Gap Detector on denser of the previously discussed objects in Section 7.1.1 are shown in Figure 7.3. Similar to the results of the gCNN, the Angular Gap Detector has the best results on the object previously identified as the easiest object for an edge detector (see Figure 7.3, left). However, the detector shows mediocre results that are significantly worse than what the neural network is showing. The same mechanics are observed where the AUC is high for all objects, but the F1 is low, this time due to low Recall additionally.

Surprisingly, the detector performs better on the allegedly most difficult object on the right than the easier object in the middle. The Angular Gap Detector shows good detectability of round edge trajectories and even the edges of the holes with a comparably small trajectory radius.

Overall, the Angular Gap edge detector performs significantly worse on the testing objects than the neural network - by visual standards and based on the chosen performance metrics.

Classification results of the Surface Variation Detector applied to the previously discussed objects from the ABC dataset are shown in Figure 7.4. Overall, the classification performance of the surface variation detector on the testing data is disappointing. The discriminability, quantified by the AUC, is only gradually above a chance classifier. The F1 is so low that it can't even be considered classification because both Precision and Recall are very low.

On the first object on the left (see Figure 7.4), the classifier shows a clear detection ability for corners. The corners and the area around them stand out from the other classification results and

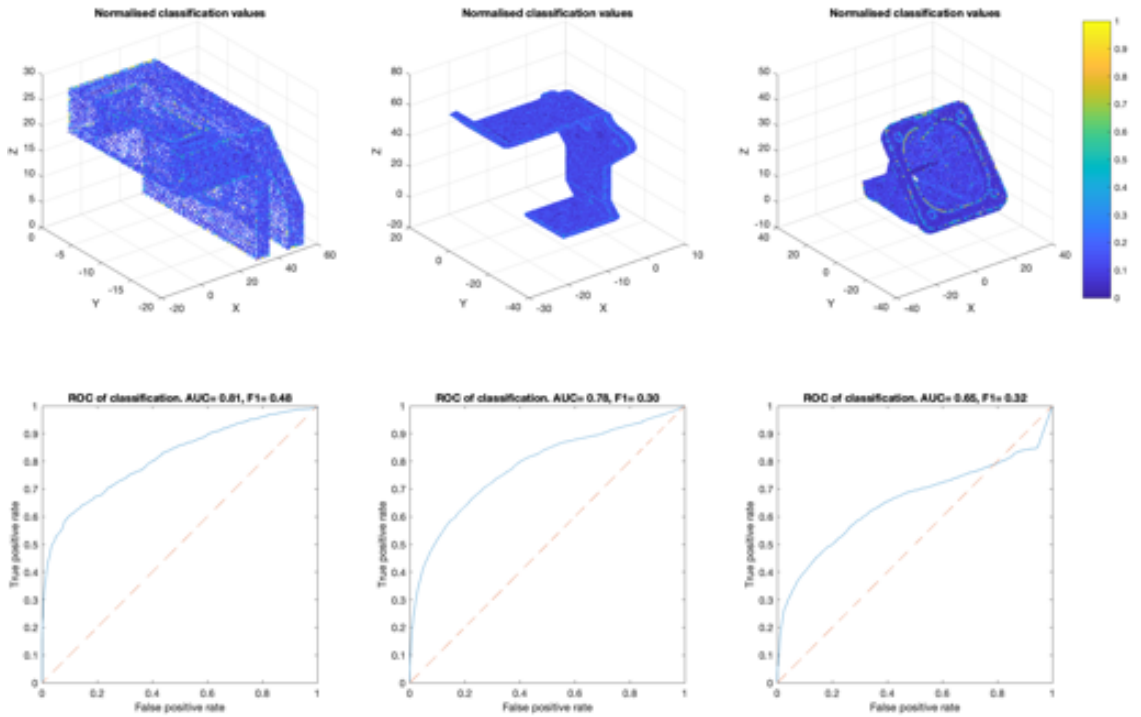


Figure 7.3: Angular Gap Detector classification of the exemplary objects from the ABC dataset. The normalised classification results are shown in the **top row** and the classification ROC in the **bottom row**. Note that a classification value/ label of 1 signifies an edge point, whereas 0 signifies a non-edge point.

are visually easily discernable.

However, overall, the classifier performs poorly on the testing data and highlights its main shortcoming: finding a fitting parameter set for the detection is very difficult. For each object, a new set of working parameters needs to be found, which is time-consuming and cannot be easily automated because, for unknown data, there is no optimisation problem to be formulated - the decision on which parameters to use is based solely on the visual assessment of the human operator.

7.1.3 Discussion of the shortcomings of the ABC dataset

From the previous sections, it becomes obvious where the shortcomings of the ABC dataset lie. The dataset contains a large number of objects taken from an online CAD-website (see Section 4.4). They were not originally intended for edge detection and contain only feature descriptors, not dedicated edge labels. The difference becomes obvious when looking at the left object depicted in Figure 6.1 taken from the dataset. The ground truth labels are depicted in yellow, and the non-edge labels are in blue. The object shows many rounded free-form surfaces, slanted shapes, and rounded edges. The feature descriptors provided with the data provide boundaries for the different features. Due to the lack of a better label, these feature boundary descriptors are converted into edge labels. In most cases, when the object's overall shape is simple (as in the object in Figure 7.1, left), the chosen path yields good results because the feature boundaries correspond with edges. However, in many other cases, said boundaries do not represent edges in our sense but are only where one CAD

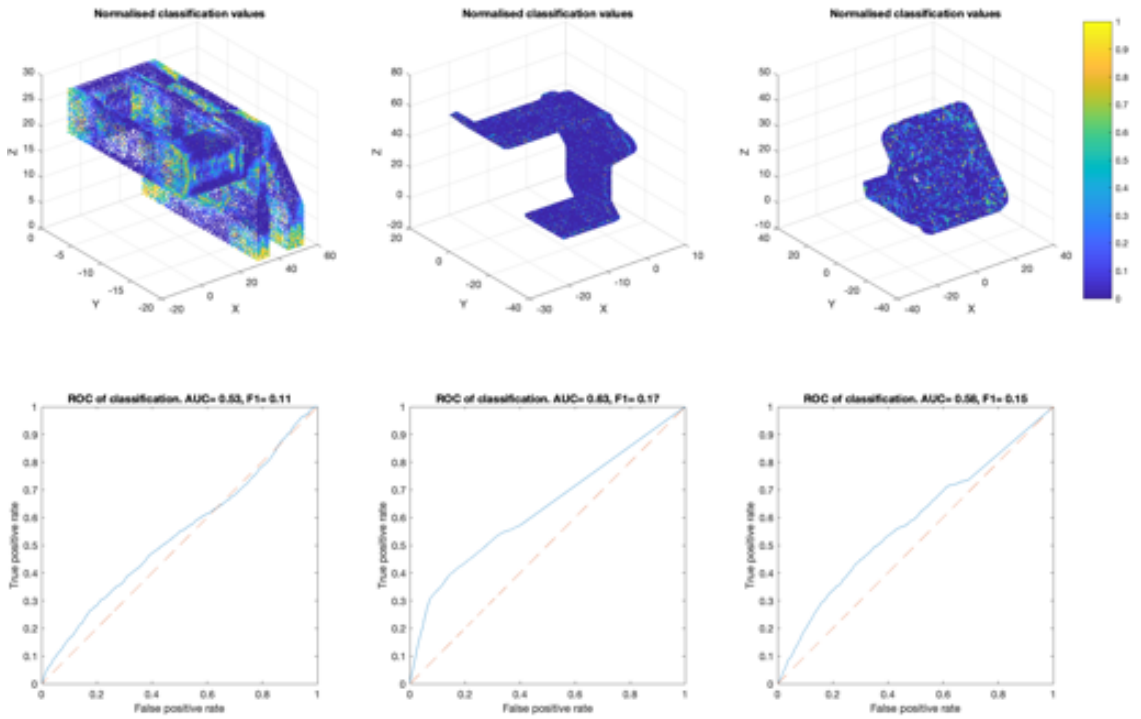


Figure 7.4: Surface Variation Detector classification of the exemplary objects from the ABC dataset. The normalised classification results are shown in the **top row** and the classification ROC in the **bottom row**. Note that a classification value/ label of 1 signifies an edge point, whereas 0 signifies a non-edge point.

body feature flows into the neighbouring feature. In modern times, where through the use of 5-axis CNC milling and 3D printing, free-form shapes are transferred straight from the digital realm into the physical world CAD-systems do a very good job in joining these neighbouring features and to prevent a detectable edge. However, these boundaries present mislabeled data to our detector.

Another major problem the dataset shows is the number of objects with low point densities, compare Figure 7.2. The problem can again be traced back to the source of the dataset. CAD data is saved as a set of features that defines the final shape. In this way, data requirements are shape-dependent and not size-dependent. Simple shapes in a CAD-system sense show very low point density in surfaces (e.g., in the non-extrusion directions, see Figure 7.2, left, surfaces in the X-Y and Y-Z planes).

7.2 Discussion of the evaluation results

In this section, we will discuss the results of various edge detectors on the scanner data, before discussing the limitations of said dataset.

7.2.1 Comparison of the performance of the edge detectors on the scanner data

The results of applying the various edge detectors are shown in Figure 7.5 (also compare Table 6.3 for an overview). The classification results were thresholded using the threshold that maximises the F1, yielding the depicted crisp labels. All three implemented edge detectors yielded visually

plausible results. With edges detected in the right regions but with different amounts of false positives in the rest of the point clouds. The edges of the Surface Variation edge detector appear the cleanest but also show some very obvious misclassifications (e.g., in the rounded sections of the jigsaws). The edges of the gCNN detector seem more frayed and, in places, missing some of the edges altogether. The Angular Gap detector shows low Precision (strong tendency to misclassify points on flat surfaces), but edges are distinguishable but too broad.

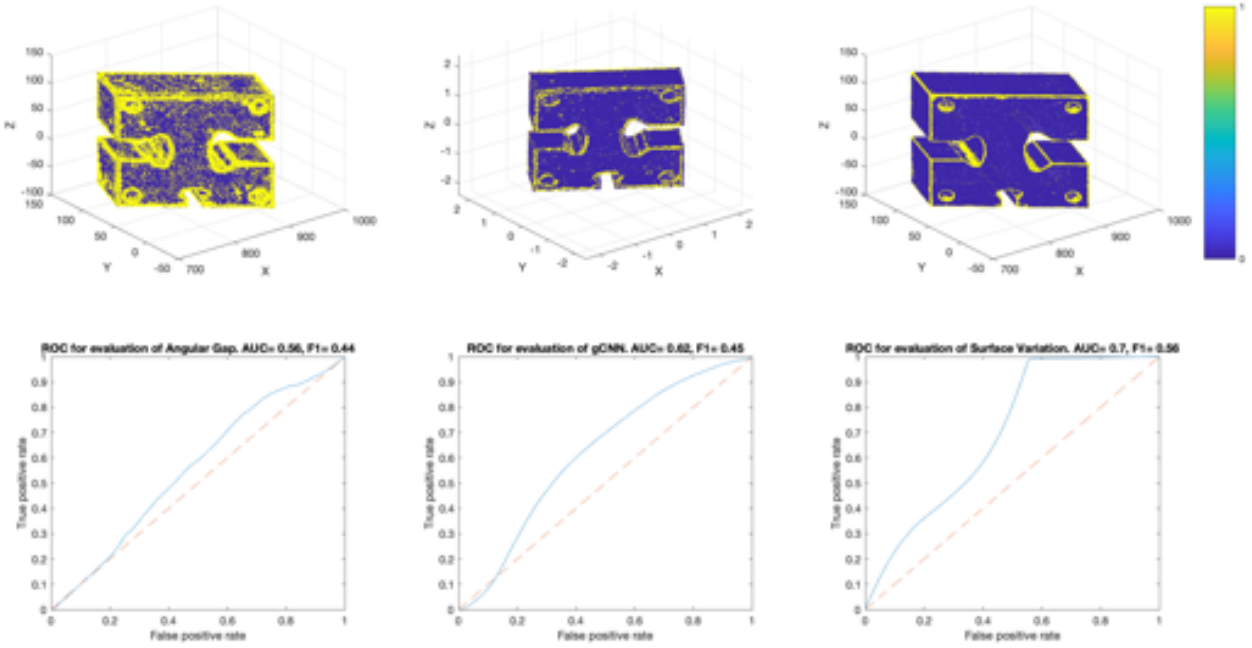


Figure 7.5: Evaluation results of the different edge detectors on the scanner data. With the classification results in the **top row** and their respective ROCs in the **bottom row**. The results of the Angular Gap Detector are shown on the **left**, the results of the Generalised Convolution Detector in the **middle**, and the results of the Surface Variation Detector on the **right**.

The AUCs of the different detectors are quite similar in their values. All three detectors yield only moderate results in the range of 0.56, 0.62 and 0.7. Surprisingly, considering the results on the testing data, the Surface Variation Detector yielded the best results with the highest scores, followed by the gCNN, but by a significant margin. The gCNN, in turn, is very closely followed by the Angular Gap Detector, which scores very similarly - both scoring only barely over the chance detector.

The F1 score of the three different results supports the observations taken based on the AUCs. Again, the Surface Variation Detector yields the best results with an F1 of 0.56, followed by the gCNN and the Angular Gap Detector, scoring 0.45 and 0.44, respectively.

With the help of the details depicted in Figure 7.6, and the help of the features defined in Section 4.4.3 (see Figure 4.6, right) we can analyse the results further.

Upon examining Figure 7.6 (top row), the top right corner of the scanned object is visible, with a borehole located close to it. The performance around the borehole is very different depending on which edge detector was used. The Angular Gap Detector detected the borehole's edge but misclassified many points around as edge points (false positives), showing low Precision again.

The gCNN only found some edge points around the borehole and added misclassifications. The Surface Variation detector classified the hole's edge the best, with the whole edge distinguishable. The main straight edges (top edge in the front, vertical right side edge, and the edge going away into the depth (y-direction) of the figure from the corner) depicted in the detail also show differing results. The Angular Gap Detector seems to miss the vertex of the edge. Still, it seems to misclassify points around the vertex as edge points, which could be due to the neighbourhood refinement not functioning properly, and no plane to be found containing the query point. An increased point density is visible around the front edge of the top of the object with some frayed-out feature (a scanner artefact), which the gCNN happily picks up on - the boundary of said features is misclassified as edge points. The detector's edges are wider than the actual edges, but it performs well on straight edges, with all three edges distinguishable in the figure.

The detail depicted in the middle row of Figure 7.6 shows the largest of the jigsaw cutouts with the rounded edge trajectory and the inside edge with the obtuse angle (compare Figure 4.6, right). All three edge detectors pick up on the rounded edge trajectory reliably and correctly classify most of the edge. The Angular Gap and the Surface Variation detectors also show good results on the edge with the obtuse angle, which is missed entirely by the gCNN. The gCNN is the only detector that does not misclassify the inside of the jigsaw besides its rounded shape.

In Figure 7.6, the bottom row displays the corner of the object with the borehole, the smallest jigsaw cut-out, and on the backside of the object, we can observe the double edge, which is visible through the object. The borehole is detected similarly to the one in the top right corner, with the Surface Variation Detector classifying it correctly, the Angular Gap Detector picking up on it but also misclassifying many points around it, and the gCNN missing most of it. The jigsaw cutout's rounded edge trajectory also shows a mixed picture this time, with the gCNN showing difficulties following it. All three edge detectors correctly classify the double edge in the back.

Overall, the Angular Gap Detector seems to be able to pick up on most edge features (high Recall). Unfortunately, this is paired with many misclassifications in the form of false positives (low Precision), which explains the low scores in F1. Surprisingly, it also struggles with the vertex of the edge and only finds the areas around it. This may be related to the ambiguity about the plane used to calculate the vectors between the query point and its neighbours. This can be adjusted by altering the distance tolerance for valid points within the plane.

The gCNN shows a combination of both medium Recall, but at the same time higher Precision - not all edge points are found, but the ones found are mostly correct. The network seems to also struggle with artefacts in the data, e.g., the boundary of the are with increased density. However, especially when difficult contours or shapes are present, the gCNN performs best (e.g., inside the jigsaw).

Surprisingly, considering the detector's performance on the testing data, the Surface Variation detector shows the best results on the scanner data. It shows good Recall and classifies the edges reliably. However, the edges found are quite thick (especially compared to the ground truth labels). This leads to lower Precision and therefore F1 than the visual results would make one anticipate.

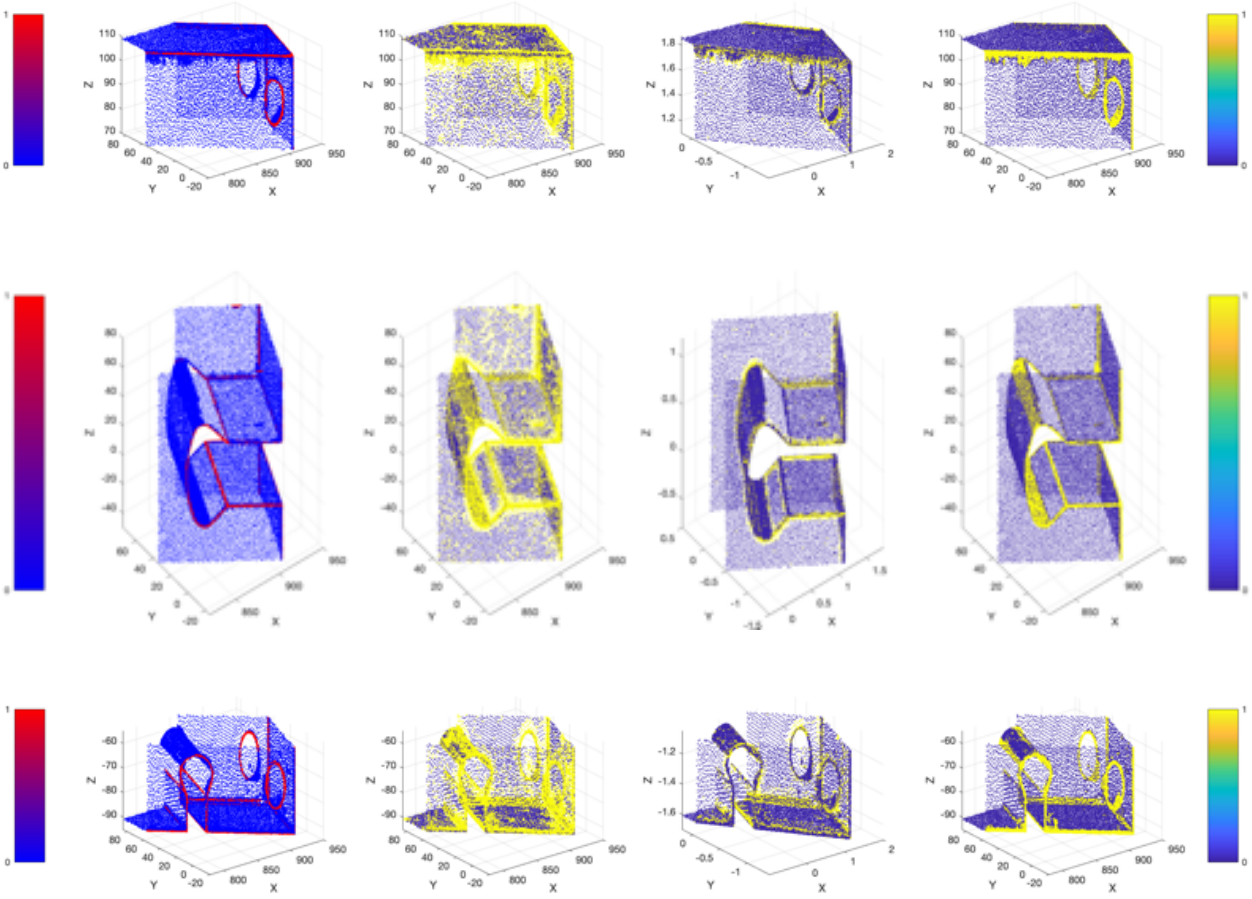


Figure 7.6: Details of the classification results on the scanner data from the classification results depicted in Figure 7.5. The top right corner of the object with a borehole **top**, the larger of the two opposite jigsaw shapes in the **middle**, and the bottom edge with the small jigsaw shape, the borehole and the double edge in the back on the **bottom**. After the depiction of the ground truth labels on the very left, results of the Angular Gap Edge Detector are again shown on the **left**, the results of the Generalised Convolution Edge Detector in the **middle**, and the results of the Surface Variation edge detector on the **right**.

7.2.2 Discussion of the shortcomings of the scanner data

After showing some of the results from the application of the edge detectors, we have come across some very obvious shortcomings of the scanner data. The data has been labelled manually, which on top of the human error involved, leads to inconsistent edges based on subjective visual decisions. It is difficult to make sense of a 3D scene presented in 2D on the screen, know where one is at any given point in time, and then select the correct point from a set perspective. Additionally, when the edge is not sharp the decision of where the edge begins and ends is rather philosophical without a defined metric - there are multiple points that could still be part of the edge, but also of the neighbouring surface plane. Not only does this influence the performance parameters of the edge detectors, but it also makes it difficult to compare them since one could coincidentally be more in line with our subjective definition of an edge in a certain area than the others.

8 Conclusion

This thesis presents a Generalised Convolution Edge Detector that performs well on artificial datasets but requires improvement when applied to real-world scanner data. The Generalised Convolutional Neural Network is a promising method for detecting edges in point clouds. The implemented detector exhibits superior performance on artificial data, including more complex objects, compared to classical approaches used as benchmarks. Additionally, the detector has the advantage of not needing new parameters to be optimized for each new dataset. Additionally, it was shown that it is similar in time complexity compared to the classical approaches because the computation of features in 3D is dominated by the computation of neighbours. Different architectures of gCNN-systems were tested, and the working architecture with four consecutive layers of General Convolution modules turned out favourable. A suitable training set was created, and an existing dataset was identified and utilised for training. The gCNN showed good training progress and testing ability on both training sets and surpassed the classical edge detectors on both datasets in comparison.

However, the shortcomings become obvious when applying the gCNN to real-world scanner data. Not only was the gCNN outperformed by the Surface Variation Detector, but also only showed slightly better results than the Angular Gap Detector. It has become evident that the training datasets are not diverse enough and do not make the gCNN robust enough to apply to real-world scanner data. However, the results are still promising since they are on par with the current state-of-the-art approaches despite the suboptimal training data.

The Generalised Convolutional Neural Network is unique because it doesn't require any data parameters to be applied. Once features are identified, computations are quick, making it ideal for live use, similar to classical edge detectors. Their similar time complexity also makes them equally scalable. In addition, it is suited for online learning, where it can improve during application and over time. However, compared to classical edge detectors, it also has some disadvantages, including the need for diverse enough training data to prepare the detector for the planned application, the time needed for training, and the associated computing power and memory usage.

There are several possible directions for future research:

- The inability of the gCNN to generalise to the scanner data most likely stems from the training data - the ABC dataset is not the most suitable for training an edge detector. To make the gCNN more resilient, it is necessary to augment the data by increasing its density, applying shearing, scaling, and more noise. Furthermore, curation is required to remove objects with low point densities that are unsuitable, and the ground truth labels should be checked for feasibility.

- Implementation of suitable data preprocessing before scanner data is input into the detectors could prevent them from misclassifying artefacts in the data.

- Batching in the presentation of the data during training, can speed up the learning process and stabilise the error function - yielding better training results in shorter amounts of time.

- Pooling and striding yield more complex features earlier with respect to the network depth and can also speed up training by decreasing network complexity and decreasing feature space complexity faster.

- If a network is optimised for a specific application, the loss and associated punishment factor can be optimised to improve network performance with a specific goal in mind.
- Improving the definition of a quantifiable edge can enhance the quality of labelling scanner data and make it easier to compare and assess the performance of detectors. Another approach is to define a better performance measure that considers the distance to an actual edge, thereby accounting for inaccuracies in edge definition, further improving comparability.
- Using a more capable machine for the training improves the ability to find better architectures/ models and training parameters and improves the application performance of the network.

List of Abbreviations

2D two-dimensional

3D three-dimensional

AGPN Analysis of Geometric Properties of Neighborhoods

AUC Area-Under-Curve

BCE Binary Cross Entropy

CAD Computer Aided Design

CNN Convolutional Neural Network

FLANN Fast Library for Approximate Nearest Neighbours

FN false negative

TP true positive

gCNN Generalised Convolutional Neural Network

LIDAR Light Detection and Ranging

MCP neuron McCulloch-Pitts neuron

MLP Multilayer Perceptron

NN Neural Network

PCA Principal Component Analysis

PCL Point Cloud Library

PR Precision-Recall

RANSAC Random Sample Consensus

ReLU Rectifier Linear Unit

ROC Receiver Operating Characteristic

SGD Stochastic Gradient Descent

SLAM Simultaneous Localisation and Mapping

TOF Time-of-flight

WCE Weighted Binary Cross Entropy

List of Figures

List of Tables

Bibliography

[1] Aleksandr Savchenkov, Andrew Davis, and Xuan Zhao. Generalized convolutional neural networks for point cloud data. Proceedings - 16th IEEE International Conference on Machine Learning and Applications, ICMLA 2017, pages 930–935, 2018. doi: 10.1109/ICMLA.2017.00-35.

[2] Vincent Torre and Tomaso A Poggio. On Edge Detection. IEEE Transactions on Pattern Analysis and Machine Intelligence, (2):147–163, 1986.

[3] Dietrich W. R. Paulus and Joachim Hornegger. Pattern Recognition and Image Processing in C++. Vieweg+Teubner Verlag, Wiesbaden, 1995. ISBN 9783528054915.

[4] Mark A. Georgeson. Human vision combines oriented filters to compute edges. Proceedings of the Royal Society B: Biological Sciences, 249(1326):235–245, 1992. ISSN 14712970. doi: 10.1098/rspb.1992.0110.

[5] Tzay Y. Young and King Sun Fu. Handbook of Pattern Recognition and Image Processing. Handbook of Pattern Recognition and Image Processing. Academic Press, 1986.

[6] Irwin Sobel and Gary Feldman. A 3x3 isotropic gradient operator for image processing. a talk at the Stanford Artificial Project, pages 271–272, 1968.

[7] Judith M. S. Prewitt. Object enhancement and extraction. In Picture processing and Psychopictorics, volume 10. 1970.

[8] John Canny. A Computational Apprroach to Edge Detection. IEEE Transactions on Pattern Analysis and Machine Intelligence, PAMI-8(6):679 – 698, 1986.

[9] G. T. Shrivakshan and Chandramouli Chandrasekar. A Comparison of various Edge Detection Techniques used in Image Processing. International Journal of Computer Science Issues (IJCSI), 9(5):269–276, 2012. ISSN 1694-0814; 1694-0784.

[10] Roelof Hamberg and Jacques Verriet. Automation in warehouse development. 2014. doi: 10.1007/978-0-85729-968-0.

[11] Dena Bazazian, Josep R. Casas, and Javier Ruiz-Hidalgo. Fast and Robust Edge Extraction in Unorganized Point Clouds. In 2015 International Conference on Digital Image Computing: Techniques and Applications (DICTA), pages 1–8, 2015.

[12] Kris Demarsin, Denis Vanderstraeten, Tim Volodine, and Dirk Roose. Detection of closed sharp edges in point clouds using normal estimation and graph theory. Computer-Aided Design, 39:276–283, 2007. doi: 10.1016/j.cad.2006.12.005.

[13] Jiangyong Xu, Mingquan Zhou, Zhongke Wu, Wuyang Shui, and Sajid Ali. Robust surface segmentation and edge feature lines extraction from fractured fragments of relics. Journal of Computational Design and Engineering, 2:79–87, 2015. doi: 10.1016/j.jcde.2014.12.002.

[14] Huan Ni, Xiangguo Lin, Xiaogang Ning, and Jixian Zhang. Edge detection and feature line tracing in 3D-point clouds by analyzing geometric properties of neighborhoods. Remote Sensing, 8(9), 2016. ISSN 20724292. doi: 10.3390/rs8090710.

[15] Thomas Luhmann. Nahbereichsphotogrammetrie. Wichmann, Heidelberg, 4 edition, 2018.

[16] Diana Beltran and Luis Basanez. A Comparison between Active and Passive 3D Vision Sensors: BumblebeeXB3 and Microsoft Kinect. Advances in Intelligent Systems and Computing, 252:725–734, 2014. ISSN 21945357. doi: 10.1007/978-3-319-03413-3.

[17] Christian Wöhler. 3D Computer Vision: Efficient Methods and Applications. Springer-Verlag, Berlin, 1 edition, 2009.

[18] Richard Szeliski. Computer vision: algorithms and applications. Springer, London, 1 edition, 2011. doi: 10.5860/choice.48-5140.

[19] David Forsyth and Jean Ponce. Computer vision: a modern approach. Pearson, Essed, 2 edition, 2012.

[20] Yoshiaki Shirai. Three-Dimensional Computer Vision. Springer- Verlag, 1 edition, 1986. doi: 10.1007/978-1-4613-3027-1-11.

[21] Li Zhang, Noah Snavely, Brian Curless, and Steven M. Seitz. Spacetime faces: High-resolution capture for modeling and animation. Data-Driven 3D Facial Animation, pages 248–276, 2007. doi: 10.1007/978-1-84628-907-1-13.

[22] Wilhelm Burger and Mark James Burge. Digitale Bildverarbeitung. Springer- Verlag, Berlin Heidelberg, 1. edition, 2005.

[23] Allan G. Weber. The USC-SIPI Image Database : Version 6. Ming Hsieh Department of Electrical Engineering: Signal and Image Processing Institute, pages 1–24, 2018.

[24] Lars Linsen. Point cloud representation. Technical Report, Faculty of Computer Science, University of Karlsruhe, Germany, pages 1–18, 2001.

[25] Mohamed Daoudi, Anuj Srivastava, and Remco Veltkamp. 3D Face Modeling, Analysis and Recognition. John Wiley & Sons Ltd, West Sussex, 1 edition, 2013.

[26] Mario Botsch, Leif Kobbelt, Mark Pauly, Pierre Alliez, and Bruno Levy. Polygon Mesh Processing. CRC Press, 2010. doi: 10.1201/b10688.

[27] Nilanjana Karmakar, Arindam Biswas, Partha Bhowmick, and Bhargab B. Bhattacharya. Construction of 3D Orthogonal Cover of Digital Object. In Aggarwal, J.K., Barneva, R.P., Brimkov, V.E., Koroutchev, K.N., Korutcheva, E.R. (eds) Combinatorial Image Analysis., volume 6636 Lectu, pages 70 – 83. Springer, Berlin Heidelberg, 2011. ISBN 9783642210723. doi: 10.1007/978-3-642-21073-0.

[28] Tom Vierjahn, Guido Lorenz, Sina Mostafawy, and Klaus Hinrichs. Growing Cell Structures Learning a Progressive Mesh During Surface Reconstruction - A Top-Down Approach. EUROGRAPHICS 2012 – Short Papers, (May):29–32, 2012. ISSN 1017-4656. doi: 10.2312/conf/EG2012/short/029-032. URL http://diglib.eg.org/EG/DL/conf/EG2012/short/029-032.pdf.

[29] Marius Muja and David G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. VISAPP 2009 - Proceedings of the 4th International Conference on Computer Vision Theory and Applications, 1:331–340, 2009. doi: 10.5220/0001787803310340.

[30] Jon Louis Bentley. Multidimensional Binary Search Trees Used for Associative Searching. Communications of the ACM, 18(9):509–517, 1975. ISSN 00010782.

[31] Jerome H. Friedman, Jon Louis Bentley, and Raphael Ari Finkel. An Algorithm for Finding Best Matches in Logarithmic Expected Time. ACM Transactions on Mathematical Software (TOMS), 3(3):209–226, 1977. ISSN 15577295. doi: 10.1145/355744.355745.

[32] Richard O. Duda, Peter E. Hart, and David G. Stork. Pattern Classification. John Wiley & Sons Ltd, second edi edition, 2001.

[33] David M. W. Powers. Evaluation: from precision, recall and F-measure to ROC, informedness, markedness and correlation. International Journal of Machine Learning Technology, 2(1):37–63, 2011. URL http://arxiv.org/abs/2010.16061.

[34] Tom Fawcett. An introduction to ROC analysis. Pattern Recognition Letters, 27(8): 861–874, 2006. ISSN 01678655. doi: 10.1016/j.patrec.2005.10.010.

[35] Jesse Davis and Mark Goadrich. The relationship between precision-recall and ROC curves. ACM International Conference Proceeding Series, 148:233–240, 2006. doi: 10.1145/1143844.1143874.

[36] Ethem Alpaydin. Introduction to Machine Learning. MIT Press, Cambridge, 4 edition, 2020.

[37] Warren S. McCulloch and Walter H. Pitts. A logical Calculus of the Ideas immanent in Nervous Activity. Bulletin of Mathematical Biophysics, 5:115–133, 1943.

[38] Marvin Lee Minsky. Theory of Neural-Analog Reinforcement Systems and its Application to the Brain-Model Problem. Princeton, 1954.

[39] Frank Rosenblatt. The perceptron, a perceiving and recognizing automaton. Cornell Aeronautical Laboratory, (Project Para), 1957.

[40] Jianxin Wu. Essentials of Pattern Recognition. Cambridge University Press, Cambridge, 1 edition, 2021.

[41] Christopher M. Bishop. Neural Networks for Pattern Recognition. Oxford University Press, Oxford, 1 edition, 1995.

[42] R. Rojas. Neural Networks. Springer-Verlag, Berlin, 1996. doi: 10.1109/78.127967.

[43] Michael Nielsen. Neural Networks and Deep Learning. Determination Press, 2015. URL http://neuralnetworksanddeeplearning.com/index.html.

[44] Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning. arXiv preprint arXiv:1603.07285, 2016. URL http://arxiv.org/abs/1603.07285.

[45] Md Rakibul Islam and Abdul Matin. Detection of COVID 19 from CT Image by the Novel LeNet-5 CNN Architecture. ICCIT 2020 - 23rd International Conference on Computer and Information Technology, Proceedings, (July 2021):19–24, 2020. doi: 10.1109/ICCIT51783.2020.9392723.

[46] Azriel Rosenfeld and Mark Thurston. Edge and Curve Detection for Visual Scene AnaIysis. IEEE Transactions on Computers, C-20(5):562–569, 1971. ISSN 00189340. doi: 10.1109/T-C.1971.223290.

[47] Peter Haberäcker. Digitale Bildverarbeitung - Grundlagen und Anwendungen. Carl Hanser Verlag, München Wien, 4 edition, 1991.

[48] Wesley E. Snyder and Hairong Qi. Fundamentals of Computer Vision. Cambridge University Press, Cambridge, 1 edition, 2017.

[49] James R. Parker. Algorithms for Image Processing and Computer Vision. Wiley Publishing, Inc., Indianapolis, 2 edition, 2011.

[50] D. Sundararajan. Digital Image Processing - A Signal Processing and Algorithmic Approach. Springer Nature Singapore Pte Ltd., Singapore, 2017.

[51] Raman Maini and Himanshu Aggarwal. Study and comparison of various image edge detection techniques. International Journal of Image Processing, 3(1), 2009.

[52] Lequan Yu, Xianzhi Li, Chi Wing Fu, Daniel Cohen-Or, and Pheng Ann Heng. EC-Net: An edge-aware point set consolidation network. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 11211 LNCS:398–414, 2018. ISSN 16113349. doi: 10.1007/978-3-030-01234-2{_}24.

[53] Charles R. Qi, Hao Su, Kaichun Mo, and Leonidas J. Guibas. PointNet: Deep learning on point sets for 3D classification and segmentation. Proceedings - 30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, 2017-Janua:77–85, 2017. doi: 10.1109/CVPR.2017.16.

[54] Charles R. Qi, Li Yi, Hao Su, and Leonidas J. Guibas. PointNet++: Deep hierarchical feature learning on point sets in a metric space. Advances in Neural Information Processing Systems, 2017-Decem:5100–5109, 2017. ISSN 10495258.

[55] Xiaogang Wang, Yuelang Xu, Kai Xu, Andrea Tagliasacchi, Bin Zhou, Ali Mahdavi-Amiri, and Hao Zhang. PIE-NET: Parametric inference of point cloud edges. Advances in Neural Information Processing Systems, 2020-Decem(NeurIPS):1–12, 2020. ISSN 10495258.

[56] Dena Bazazian and M. Eulàlia Parés. EDC-net: Edge detection capsule network for 3D point clouds. Applied Sciences (Switzerland), 11(4):1–16, 2021. ISSN 20763417. doi: 10.3390/app11041833.

[57] Chems Eddine Himeur, Thibault Lejemble, Thomas Pellegrini, Mathias Paulin, Loic Barthe, and Nicolas Mellado. PCEDNet: A Lightweight Neural Network for Fast and Interactive Edge Detection in 3D Point Clouds. ACM Transactions on Graphics, 41(1), 2022. ISSN 15577368. doi: 10.1145/3481804.

[58] Christopher Weber, Stefanie Hahmann, and Hans Hagen. Sharp feature detection in point clouds. SMI 2010 - International Conference on Shape Modeling and Applications, Proceedings, pages 175–186, 2010. doi: 10.1109/SMI.2010.32.

[59] Mark Pauly, Markus Gross, and L.P. Kobbelt. Efficient simplification of point-sampled surfaces. In IEEE Visualization, 2002. VIS 2002., pages 163–170, 2002. ISBN 0-7803-7498-3. doi: 10.1109/VISUAL.2002.1183771.

[60] Sebastian Koch, Albert Matveev, Zhongshi Jiang, Francis Williams, Alexey Artemov, Evgeny Burnaev, Marc Alexa, Denis Zorin, and Daniele Panozzo. ABC: A big cad model dataset for geometric deep learning. Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2019-June:9593–9603, 2019. ISSN 10636919. doi: 10.1109/CVPR.2019.00983.

[61] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms. The MIT Press, Cambridge, 4 edition, 2022.

[62] Radu Bogdan Rusu and Steve Cousins. 3D is here: Point Cloud Library (PCL). IEEE International Conference on Robotics and Automation (ICRA), 2011.

[63] Martin A. Fischler and Robert C. Bolles. Random sample consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography. Communications of the ACM, 24(6):381–395, 1981. ISSN 15577317. doi: 10.1145/358669.358692.

[64] ISO - International Organization for Standardization. ISO/IEC 14882:2020, 2020.

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

[66] Shruti Jadon. A survey of loss functions for semantic segmentation. 2020 IEEE Conference on Computational Intelligence in Bioinformatics and Computational Biology, CIBCB 2020, 2020. doi: 10.1109/CIBCB48159.2020.9277638.

[67] Farzeen Zehra, Darakhshan Khan, Maha Javed, and Maria Pasha. Comparative Analysis of C++ and Python in Terms of Memory and Time. (December):11, 2020. doi: 10.20944/preprints202012.0516.v1. URL https://www.preprints.org/manuscript/202012.0516/v1.