University of Passau
Faculty of Computer Science and Mathematics
Chair of Digital Image Processing

Master Thesis in Computer Science

A Similarity Measure for 3D Shape Retrieval using Deep Convolutional Autoencoder and Haar Wavelets

submitted by

Alexander Paßberger

1. Examiner: Univ.-Prof. Dr. Tomas Sauer
2. Examiner: Univ.-Prof. Dr. Michael Granitzer
Date: May 21, 2023

Abstract

Computer vision is an interdisciplinary field that focuses on developing algorithms and techniques that enable machines to interpret and understand visual information from the world around us. Within this field, 3D shape retrieval has emerged as an exciting area of research, that involves the search and retrieval of similar 3D shapes from a large database. The main challenge in 3D shape retrieval is designing an appropriate architecture for encoding and a similarity function for subsequent comparison. Both have to accurately capture the structural similarities between 3D shapes to enable effective retrieval. In this thesis, a method for 3D shape retrieval in regards to CT scans using a convolutional autoencoder is proposed. The models obtained demonstrate promising results and lay the foundation for further advancements in the field of shape retrieval.

To measure the similarity between the different latent encodings, a combined similarity function is proposed. Several similarity functions are evaluated for the given data in 2D, and the best function is chosen. This function combines the Euclidean distance and the tanh squash function, effectively shifting the weighting of outliers in comparison to inliers in regards to the basic Manhattan or Euclidean distance. This incorporates a trade-off between the Manhattan and Euclidean distances, which helps to address the curse of dimensionality.

The design of the architecture of the autoencoder plays a crucial role in the success of 3D shape retrieval. In the theoretical designing phase, various architectural choices are explored, including the number of layers, their distribution of convolutional and fully connected layers, the type of activation functions, the size of the neurons and many more. These choices are guided by the goal of capturing the essential structural features of the input shapes, while keeping the model complexity manageable. The most promising theoretical configurations are evaluated through pre-tests in 2D to identify the ones that yield good performance across different datasets and tasks.

Once the theoretical architecture is defined, the next step is to transfer it to a 3D model and evaluate its performance. To ensure the robustness and generalizability of the proposed method, a precise evaluation is conducted using various augmentation methods. Techniques such as rotation, scaling, translation, and noise addition are applied to the 3D shapes in the test set, simulating realistic variations that may occur in real-world scenarios. The performance of the retrieval system is then evaluated based on metrics such as precision, recall, and mean average precision. Through these steps, the proposed method undergoes rigorous testing and refinement, aiming to achieve high retrieval accuracy and robustness in 3D shape retrieval tasks.

The integration of wavelets into the proposed method resulted in comparable retrieval performance without significant advantages in calculation time or memory usage. However, the current wavelet integration version shows limitations in handling noisy images, requiring the development of more sophisticated integration techniques. These findings highlight the need

for further refinement and exploration to fully leverage the benefits of wavelets in neural network based 3D shape retrieval, including robustness to noise and improved integration for calculation advantages.

In summary, this thesis proposes a method for 3D shape retrieval using convolutional autoencoders in CT scans. The approach incorporates a combined similarity function and explores architectural choices for effective shape encoding. The integration of wavelets shows potential but requires further refinement to address noise challenges. Overall, this research lays the foundation for advancing 3D shape retrieval and holds promise for applications in various domains.

Contents

List of Acronyms iv

List of Figures vi

List of Tables vii

List of Python Code viii

1 Introduction 1
1.1 Related Work 2
1.1.1 Algorithmic Image Retrieval and 3D Shape Retrieval 2
1.1.2 Neural Network Image Compression and Image Retrieval 2
1.1.3 Neural Network Feature Learning and 3D Shape Retrieval 3
1.2 Research Questions 4

2 Background 5
2.1 Similarity Measures 5
2.1.1 Similarity Function, Distance and Metric 5
2.1.2 Pair-Wise Distances 6
2.2 Neural Networks 7
2.2.1 Perceptron 7
2.2.2 Feedforward Neural Networks 8
2.3 Convolutional Neural Networks 10
2.3.1 The Discrete Convolution Operation 10
2.3.2 The Convolution Layer 11
2.3.3 The Pooling Layer 11
2.4 Autoencoder 12
2.4.1 Feed-Forward Autoencoder 13
2.4.2 Convolutional Autoencoder 14
2.5 Activation Functions 15
2.5.1 Trivial Activation Functions 15
2.5.2 S-Shaped Activation Functions 16
2.5.3 Rectified Linear Units 18
2.5.4 Exponential Linear Units 20
2.6 Loss Functions 22
2.6.1 Error Measures 22
2.6.2 Sum Aggregated Scale-Dependent Error Measures 23
2.6.3 Mean Aggregated Scale-Dependent Error Measures 23

2.7 Optimization . . . 24
2.7.1 Gradient-Based Optimization . . . 24
2.7.2 Momentum-Based Optimization . . . 26
2.7.3 Adaptive Optimization . . . 27
2.8 Regularization . . . 28
2.8.1 Data Augmentation . . . 29
2.8.2 The Regularization Term . . . 29
2.8.3 Dropout . . . 30
2.9 Retrieval Performance Measures . . . 30
2.9.1 Recall . . . 30
2.9.2 Precision . . . 31
2.9.3 Average Precision . . . 31
2.9.4 Mean Average Precision . . . 31
2.10 Wavelets . . . 32
2.10.1 Analyzing Wavelet and Scaling Function . . . 32
2.10.2 Haar Wavelet . . . 33
2.10.3 Multidimensional Haar Wavelet . . . 33

3 Method 36
3.1 Similarity Measure . . . 37
3.2 2D Model Design . . . 40
3.2.1 Network Architecture . . . 40
3.2.2 Network Implementation . . . 42
3.2.3 Training Architecture . . . 45
3.2.4 Training Implementation . . . 46
3.2.5 2D Retrieval Implementation . . . 49
3.3 2D Layer Configuration Tests . . . 50
3.3.1 Fashion-MNIST . . . 50
3.3.2 dSprites . . . 58
3.4 3D Model Design . . . 62
3.4.1 Network Implementation . . . 62
3.4.2 Input Pipeline . . . 63
3.4.3 Training, Evaluation and Visualization . . . 65
3.4.4 Haar Wavelet Integration . . . 67

4 Results 68
4.1 Test Cases and Evaluation . . . 68
4.1.1 Shape Similarity Tests . . . 68
4.1.2 Translation Robustness Tests . . . 69
4.1.3 Rotation Robustness Tests . . . 70
4.1.4 Noise Robustness Tests . . . 70
4.2 Shapes3D . . . 72
4.2.1 Model Comparison: Performance Evaluation . . . 72
4.2.2 Model 2 "Translation": Retrieval Visualization . . . 74
4.3 Blended Multi-Shapes3D and Wavelet Integration . . . 75
4.3.1 Model Comparison: Performance Evaluation . . . 75
4.3.2 Exploring Edge Cases: Poor Retrieval Results . . . 77

4.3.3 Real Data: Retrieval Visualization 78
4.3.4 Differences in Noise Robustness 80

5 Discussion 82
5.1 Handling Isotropic versus Anisotropic Scaling 82
5.2 Model Size and Augmentation 82
5.3 Using Haar Wavelet Filters 83

6 Conclusion and Future Work 85

Bibliography 87

Declaration 94

List of Acronyms

1D one dimensional 32

2D two dimensional 1

3D three dimensional 1

Adagrad Adaptive Gradient Algorithm 27

Adam Adaptive Moment Estimation 28

AE Autoencoder 2

AI Artificial Intelligence 1

API Application Programming Interface 40

CT Computer Tomography 4

CAE Convolutional Autoencoder 2

CNN Convolutional Neural Network 1

DWT discrete Wavelet transform 32

ELU Exponential Linear Unit 20

FFT fast Fourier transform 32

Fashion-MNIST a MNIST-like fashion product database 36

GAN Generative Adversarial Network 3

GMAP Group Mean Average Precision 66

GPU Graphics Processing Unit 49

GZIP GNU zip 63

leaky ReLU leaky Rectified Linear Unit 19

MAE Mean Absolute Error 14

MAP Mean Average Precision 31

MBE Mean Bias Error 24

MNIST Modified National Institute of Standards and Technology 2

MSE Mean Squared Error 14

Nadam Nesterov-accelerated Adaptive Moment Estimation 28

NN Neural Network 3

PCA Principal Component Analysis 2

PELU Parametric Exponential Linear Unit 21

PReLU Parametric Rectified Linear Unit 19

RBM Restricted Boltzmann Machine 3

ReLU Rectified Linear Unit 18

SELU Scaled Exponential Linear Unit 21

tanh Hyperbolic Tangent 16

List of Figures

2.1 Graph representation of a fully connected neural network 9
2.2 Graph representation of a fully connected autoencoder 13
2.3 Plots of common activation functions and their derivatives 17
2.4 Plots of advanced ReLU based activation functions and their derivative . . . 20
2.5 Plots of the Haar wavelet $\psi(t)$ and it's scaling function $\varphi(t)$ 34

3.1 Example graphic on different similarity types 37
3.2 Comparison of 2D retrieval results for Manhattan and Euclidean distances . . 38
3.3 2D retrieval improvements by tanh normalized similarity measure 39
3.4 Convolutional autoencoder encoder visualization 41
3.5 Example images from the Fashion-Mnist dataset 51
3.6 Processing of example images by fm_1 53
3.7 Processing of example images by fm_2 55
3.8 Retrieval results for Fashion-MNIST shoe query 56
3.9 Retrieval results for Fashion-MNIST pullover query 56
3.10 Retrieval results for Fashion-MNIST set restricted shoe query 57
3.11 Example images of the dSprites dataset 58
3.12 Retrieval results for the dSprites heart query 61
3.13 Retrieval results for the dSprites ellipsoid query 61

4.1 Example volumes from the Shapes3D dataset 72
4.2 Volumes of three example test queries from Shapes3D 74
4.3 Retrieval results of Model 2 on the example test queries from Shapes3D . . . 74
4.4 Example volumes of the Multi-Shapes3D dataset 75
4.5 The loss curves of Model 2 and Model 3 75
4.6 Easy test queries of Multi-Shapes3D with bad retrieval results 77
4.7 Bad retrieval results for easy test queries on Multi-Shapes3D of Model 2 . . . 77
4.8 Bad retrieval results for easy test queries on Multi-Shapes3D of Model 3 . . . 78
4.9 Real data test queries of Multi-Shapes3D 78
4.10 Retrieval results for real data test queries on Multi-Shapes3D of Model 2 . . 79
4.11 Retrieval results for real data test queries on Multi-Shapes3D of Model 2 . . 79
4.12 Speckle Noise processing comparison 80
4.13 Salt & pepper noise processing comparison 81
4.14 Gauss noise processing comparison 81

List of Tables

3.1 Results of different Dense layer configurations of fm_1 51
3.2 Results of different regularization techniques on fm_1 52
3.3 Results of different alpha values for leaky ReLU on fm_1 52
3.4 Results of fm_1 without sigmoid activation at the latent layer 53
3.5 Results of different Dense layer configurations on fm_2 54
3.6 Results of the architectures fm_1 and fm_2 after 100 training epochs 55
3.7 Precision and modified average precision of different convolution kernel sizes . 60

4.1 Retrieval scores for Model 1 and variously trained Model 2 on Shapes3D . . . 73
4.2 Retrieval scores for Model 2 "Translation" and Model 3 "Haar Wavelet" on Multi-Shapes3D 76

List of Python Code

3.1 The Autoencoder base class 42
3.2 An implementation of a chained Convolution-Activation-Pooling layer 44
3.3 An example AE implementation 44
3.4 An example implementation of an encoder 45
3.5 An example implementation of a decoder 45
3.6 Interface of the CAEManager Class 47
3.7 Loading and preprocessing of Fashion-MNIST 48
3.8 dSprites input pipeline for the train set 49
3.9 The calculation of the similarity list 49
3.10 Layers and configuration of architecture 1 62
3.11 Layers and configuration of architecture 2 63
3.12 Definition of the read_n_prepare_dataset method 64
3.13 The implementation of the translation augmentation. Only the most significant code lines are given. 64
3.14 The relevant code lines of the implementation of gauss noise augmentation . . 64
3.15 The relevant code lines of the implementation of salt&pepper noise augmentation 65
3.16 The significant lines of implementation of the speckle noise augmentation . . 65
3.17 The calculation of the average precision in the util module 66
3.18 The convenience function evaluate_group 66
3.19 The self-defined layer ConvolveFilters for an arbitrary fixed convolution, which is used to apply the wavelet filters to the input. 67
3.20 The calculation of the Haar wavelet filters in numpy 67

1 Introduction

Computer vision is an interdisciplinary field of study that involves the development of algorithms and techniques to enable machines to interpret and understand visual information from the world around us. This technology has seen widespread use in numerous applications such as surveillance, autonomous vehicles, face recognition, medical imaging, and many more [1–4].

Research in Artificial Intelligence (AI) and computer vision has led to significant advances in recent years, including the development of deep learning models such as Convolutional Neural Networks (CNNs), which have revolutionized the field. These models have achieved remarkable results in tasks such as object detection, image classification, and semantic segmentation, and have become a cornerstone of modern computer vision [5–7]. The success of these models can be attributed to their ability to automatically learn relevant features from raw data and capture complex patterns and relationships within images. In addition, the availability of large datasets and powerful computational resources has enabled researchers to train increasingly larger and more complex models, leading to continued advances in the state of the art [8, 9].

Another crucial task in modern day vision-based AI systems is the ability to compare pictures or volumes, known as matching. Matching serves as the foundation for more complex problems such as high-dimensional structure recovery, three dimensional (3D) reconstruction, visual simultaneous localization, mapping, image mosaic, image fusion, image retrieval, target recognition, and more [10]. It is also an essential technique for 3D shape retrieval, which is the focus of this thesis. 3D shape retrieval is the process of searching and retrieving 3D models from a database based on their shape characteristics and features.

The field of image matching can be divided into four different approaches. The first approach, known as area-based methods, uses similarity measurements on the original image pixels' intensity or information without detecting any salient structures in the image. The second and most important approach is feature-based methods, which extract key features and descriptors and apply the matching task to them. The third and fourth approaches use classical machine learning or deep learning techniques to solve the problem [10].

In object retrieval, the focus is on the shape rather than the whole image, and feature-based solutions have been the state-of-the-art approach for algorithmic solutions [11]. While these approaches have achieved decent results in various fields, a superior approach is still missing. An overview of algorithmic object retrieval is presented in subsection 1.1.1. Machine learning research has been predominantly focused on the deep learning field, with CNNs being at the forefront of development due to their promising results [12]. In subsection 1.1.2 an overview of machine learning approaches for two dimensional (2D) image retrieval is provided, while subsection 1.1.3 discusses the same for 3D shape retrieval.

In this thesis, the focus is on developing a method applicable for 3D shape retrieval using Convolutional Autoencoders (CAEs). The key contribution is the design of an appropriate Autoencoder (AE) architecture and similarity function that can accurately capture the structural similarities between 3D shapes, allowing for effective retrieval. Furthermore, wavelet filters are integrated into the approach to hopefully enhance the feature extraction process, leading to a more robust, accurate, and efficient retrieval system. Through experiments and evaluations, the proposed method demonstrates promising results and lays the foundation for further advancements in the field of shape retrieval.

1.1 Related Work

1.1.1 Algorithmic Image Retrieval and 3D Shape Retrieval

According to A. Goodrum [13], research on algorithmic retrieval started more than 50 years ago, in the form of systems for text retrieval. As images include information on what the image is about as well as what is actually depicted in the image, textual representation is problematic. Therefore, research shifted towards content-based image retrieval techniques, which rely on the extraction of primitive features. Quite a few of these content-based information retrieval systems were used commercially from the late 1990s onward. For more detailed information, the reader is referred to the overview work of A. Goodrum [13].

At the time content-based information retrieval systems became available for images, research for 3D images began. In 1993, Humblet & Dunbar [14] described a similarity searching method for structure-activity and molecular design in medicine. Ankerst *et al.* [15] used 3D shape histograms based on a flexible similarity distance function as a similarity model for 3D objects. A lot of different approaches were presented in the early 2000s and onward as the field got a lot of attention: Saupe, Vranić and Richter used spherical harmonics and moments on polygonal meshes based on Principal Component Analysis (PCA) for normalization [16], [17]. In 2002, Osada *et al.* [18] proposed a method based on shape signatures 3D polygonal models. In contrast to the mentioned geometrical approaches, Chen *et al.* [19] proposed a novel method based on visual similarities measured with image differences in the light field. Further methods include spin image signatures [20], spectral embedding using eigenvectors of an appropriately defined affinity matrix [21], a hybrid descriptor composed of 2D features based on depth buffers and spherical harmonics 3D features [22], triangulated meshes [23], a graph based representation after mesh segmentation [24], bag-of-words descriptors [25], covariance based descriptors [26], multiscale fourier descriptor [27] and many more. For a precise overview, see [28], [29] or the most recent[30].

1.1.2 Neural Network Image Compression and Image Retrieval

In 1989, Le Cun *et al.* [31] introduced a neural network with backpropagation for processing images directly without the need for feature vectors as input. The network was trained for recognizing zip-code digits and classifying them using a database that is now widely known as the Modified National Institute of Standards and Technology (MNIST) database of handwritten digits. An updated version of the database is still frequently used to evaluate learning techniques and is freely available [32].

Research on Neural Networks (NNs) for classification continued, but they were not applicable for retrieval or even compression for a long time. However, this changed when Hinton & Salakhutdinov [33] introduced the concept of AEs capable of reducing the dimensionality of an 2D image in "Reducing the Dimensionality of Data with Neural Networks". Afterwards, approaches to these problems began to shift towards machine learning.

For instance, Krizhevsky & Hinton [34] used deep AEs to compress 28x28-pixel images into semantically hashed binary codes, according to Salakhutdinov & Hinton [35]. Xu & Fang [36] presented a deep AE for image shape retrieval that directly worked on raw 2D images. Their AE produced a 40-dimensional descriptor using a stack of Restricted Boltzmann Machines (RBMs) for pre-training. Later Cai *et al.* [37] presented a triplet CNN for content-based image retrieval. Today, deep learning approaches dominate research on image compression. "According to experimental results, [CAE] CAEs achieve better coding efficiency than JPEG by extracting compact features. [Generative Adversarial Network (GAN)] GANs show potential advantages on large compression ratio and high subjective quality reconstruction. Super-resolution achieves the best rate-distortion (RD) performance among them, which is comparable to BPG [...] Deep learning based approaches not only achieve better coding efficiency, but also can adapt much quicker to new media contents and new media formats." [38, p. 1].

1.1.3 Neural Network Feature Learning and 3D Shape Retrieval

Following the successes of deep learning approaches in image classification, object detection, and more, researchers turned their attention towards 3D shape retrieval. Early approaches, such as that of Zhu *et al.* [39], simply projected 3D shapes into 2D space as multiple views and then aggregated the learned 2D features. Similar methods were proposed by Liu *et al.* [40], Zhou & Jia [41], and Leng *et al.* [42].

Other works have used graph-based 3D models, such as that presented by Xie *et al.* [43], which converted them into multiple one-dimensional feature vectors using the heat kernel signature at different scales based on eigenfunction expansion with the Laplace-Beltrami operator. Afterwards, a discriminative AE is employed to obtain a shape descriptor by combining these feature vectors with the latent dimension. In a similar vein, Bu *et al.* [44] combined a geometric bag-of-words descriptor with deep belief networks for shape analysis.

More recently, fully automated approaches have emerged, such as that of Wang *et al.* [45], who used a combination of CAEs, AEs, and extreme learning for rapid 3D feature learning. Another recent work, by Yu & Sabuncu [46], combined a spatial transformer network with a CAE to obtain a rotation-, translation-, and axis-independent scaling-invariant descriptor for instance retrieval. These and other deep learning approaches promise to greatly enhance 3D shape retrieval, enabling more accurate and efficient search in 3D model databases.

1.2 Research Questions

Using CAE seems to be the go-to approach for solving the problem of 3D shape retrieval. A few different architectures were proposed and used with promising results for the task, combining feed-forward AEs and CAEs. Hence, most of those papers lack a previous comparison of the exact layer design of deep AE architectures and therefore an explanation for the proposed solution. Therefore, the first research question of this thesis consists of finding an appropriate architecture to combine the various layers in CAEs.

Research Question 1. *Which combination of layers, activation, and regularization is suitable for an CAE in the case of shape retrieval, and how to set their hyper-parameter using state-of-the-art optimization?*

AEs only performs the compression of the data; for retrieval, one still has to find a suitable similarity metric. The second research question therefore addresses the field of vector comparison:

Research Question 2. *Which similarity metric provides useful results in the case of shape retrieval on latent vectors retrieved from CAEs?*

In section 2.10 the Haar wavelet, a well-known and well-working algorithmic edge detector, is presented. The first convolutional layer of deep architectures should basically learn edge-detecting filters as well. In some architectures, like the one from Kausar *et al.* [47], Haar wavelet-decomposed images are used as input to a convolutional NN for decreasing the required GPU memory for breast cancer classification in histological images. As Computer Tomography (CT) scans deliver huge volumetric images, further optimizing is desirable in this type of volumetric shape retrieval, too. Therefore, the third research question arriving is whether the Haar wavelet filters can also optimize CAEs in regards to the proposed method.

Research Question 3. *Can Haar wavelet filters be used in CAEs for fast edge-detection to reduce the convolutional layers of the proposed architecture and hence the overall computation time or memory required?*

2 Background

2.1 Similarity Measures

In deep learning approaches for retrieval, the calculation of features and the comparison of their descriptions are the two main components. For the second part, the measure of the similarity of the received encodings of different inputs of an AE, it is necessary to employ suitable similarity measures.

In subsection 2.1.1, the different concepts related to measuring similarity, including similarity and dissimilarity functions, distance functions, and metrics, are presented. Since the encoding of an AE is a multi-dimensional n-vector $x = (x_1, ..., x_n)$, and generally has no distribution, a certain dimension can encode a specific feature that can be compared pair-wise. In subsection 2.1.2 an overview of suitable distances, especially combined distances based on numerical distances between points, is given.

Only a limited overview of suitable distances is presented here, precisely combined distances based on numerical distances between points. For further information on distances and metrics, the reader is referred to [48], [49] or especially for image retrieval, [50].

2.1.1 Similarity Function, Distance and Metric

A similarity function or a dissimilarity function are loosely defined concepts used to measure the similarity or dissimilarity between two arbitrary objects.

Definition 2.1. ***Similarity Function***
A function s is called a similarity function if it satisfies the following three properties [48]:

(i) non-negativity $s(x, y) \geq 0$*;*

(ii) symmetry $s(x, y) = s(y, x)$*;*

(iii) $s(x, y)$ *monotone increasing for x and y being more similar.*

In contrast, a function S is called a dissimilarity function if for property (iii) it is monotone increasing for x and y being more dissimilar.

Distances and metrics are more strictly defined concepts that share the first two properties with similarity functions [48].

Definition 2.2. ***Distance and Metric***
A function d is called a distance function or short distance if properties (i)-(iii) are satisfied. If d also satisfies properties (iv) and (v) it is called a metric.

(i) *non-negativity* $d(x, y) \geq 0$;

(ii) *symmetry* $d(x, y) = d(y, x)$;

(iii) *identification mark* $d(x, x) = 0$;

(iv) *definiteness* $d(x, y) = 0$ *if and only if* $x = y$;

(v) *triangle inequality* $d(x, y) + d(y, z) \geq d(x, z)$.

Sometimes the identification mark definition is omitted and definiteness, which implies an identification mark, is directly used as a condition for a distance, for example in [49]. Gentleman *et al.* [48] also states that there is no need to require symmetry with some adjustments, mentioning air plane flight times as non-symmetric examples.

2.1.2 Pair-Wise Distances

As mentioned, the latent encoding of an AE is a n-vector $x = (x_1, ..., x_n)$. Given another encoding $y = (y_1, ..., y_n)$, it is only meaningful to compare $x_k, k \in (0, n)$ with $y_k, k \in (0, n)$, hence pair-wise between points.

Definition 2.3. ***pairwise distance [48]***
Given two n-vectors $x = (x_1, ..., x_n)$ *and* $y = (y_1, ..., y_n)$. *A distance function d(x,y) is called pairwise if and only if*

$$d(x, y) = F[d_1(x_1, y_1), ..., d_n(x_n, y_n)], \tag{2.1}$$

for $d_1, ..., d_n$ *each being distances.*

The definition of pairwise distances is general for arbitrary distances $d_k, k \in (1, n)$. The most common, non-correlation-based pairwise distances are all derived from the parametric Minkowski metric. The Minkowski metric itself is based on the L_p norm substituted with the difference vector.

Definition 2.4. ***Lp Norm***
For $p \in \mathbb{R}, p \geq 1$, *the* L_p *norm is given by:*

$$||x||_p = \left(\sum_{k=1}^{n} |x_k|^p \right)^{\frac{1}{p}}. \tag{2.2}$$

Definition 2.5. ***Minkowski metric***
In accordance to the L_p norm, the Minkowski metric is defined by:

$$d_{min}(x,y) = F(z_1,,...,z_n) = \left(\sum_{k=1}^{n} z_k^p\right)^{\frac{1}{p}}, z_k = d_k(x_k,y_k) = |x_k - y_k|. \tag{2.3}$$

As the name implies, the distance also satisfies the properties of a metric. The most important cases to mention are for $p = 1, 2$, namely the Manhattan metric and the Euclidean metric. For simplicity, they are often just called Manhattan or Euclidean distances, to better blend in with a group of choose-able distances in a framework.

Definition 2.6. ***Manhattan Metric***

$$d_{man}(x,y) = \sum_{k=1}^{n} |x_k - y_k| \tag{2.4}$$

Definition 2.7. ***Euclidean Metric***

$$d_{euc}(x,y) = \sqrt{\sum_{k=1}^{n}(x_k - y_k)^2} \tag{2.5}$$

2.2 Neural Networks

NNs are a class of machine learning models inspired by the structure and function of the human brain. The simplest and earliest form of a NN is the Perceptron [51], which can only handle linearly separable problems. To overcome this limitation, the feedforward NN was developed, consisting of multiple layers of interconnected neurons that use non-linear activation functions and more powerful optimization algorithms [52]. These networks are able to model highly non-linear and complex relationships between inputs and outputs and have been successfully applied in many computer vision domains.

In the following sections, these two network models are presented in more detail. The basic concept of the perceptron is introduced first in subsection 2.2.1 before moving on to the more complex feedforward neural network in subsection 2.2.2.

2.2.1 Perceptron

Feedforward NNs are also referred to as multilayer perceptrons based on Frank Rosenblatt's 1957 presented perceptron algorithm [51]. The Perceptron algorithm is a type of feedforward NN that can be used for binary classification problems.

At each iteration, the perceptron calculates a linear combination of the input features x, the weight vector w and a bias w_b. For simplicity of notation, the vectors x and w are expanded such that $x_0 = 1$ and $w_0 = w_b$. After the calculation of z, a step function ϕ (such as the Heaviside function) is applied to produce a binary output y. The perceptron then updates its weights w based on the difference between the predicted output P_y and the actual output A_y, multiplied by the input features and a learning rate η. This update rule aims to minimize the error between the predicted and actual outputs over the training examples. The calculation of the basic perceptron is presented in Definition 2.8.

Definition 2.8. ***Perceptron [51]***
Let x be the expanded input vector of length n+1, w the expanded weight vector of length n+1 and ϕ a Heaviside mapping function. For each training object at each iteration the Perceptron calculates the mapping

$$y = \phi(z) = \phi(\sum_{k=0}^{n} w_k x_k) = \phi(w_b + w_1 x_1 ... + w_n x_n) = \phi(w^T x). \tag{2.6}$$

It then updates the weights vector w accordingly to the learning rate η and the difference between the predicted output and actual output [53].

$$w_k = w_k + \Delta w_k, \tag{2.7}$$
$$\Delta w_k = \eta(A_y - P_y)x_k \tag{2.8}$$

The perceptron is a simple but powerful algorithm that has inspired many modern NN architectures. However, it has limitations in its ability to handle non-linearly separable problems [52].

2.2.2 Feedforward Neural Networks

Feedforward NNs are a powerful class of NNs that build on the concepts introduced by the Perceptron. In modern deep feedforward networks, many neurons are combined in parallel and in chains to form complex computations. Each neuron is based on the perceptron but replaces the heavyside step function ϕ with a non-linear activation function. To learn the parameters of the network, more powerful optimization algorithms based on the gradient are used [53]. These optimization algorithms are presented in detail in section 2.7.

Definition 2.9. ***Feedforward Network***
Let x be the input vector, θ be the parameters of the whole network, and y be the desired output given by a function $y = f^*(x)$. Then the feedforward network defines the mapping $y = f(x;\theta)$ and learns the values of θ such that f approximates f^* as well as possible [52].

A feedforward NN maps an input vector x to an output vector y by applying a composition of d multiple functions chained together like $f(x) = f^d(...(f^2(f^1(x))))$ to the input. The

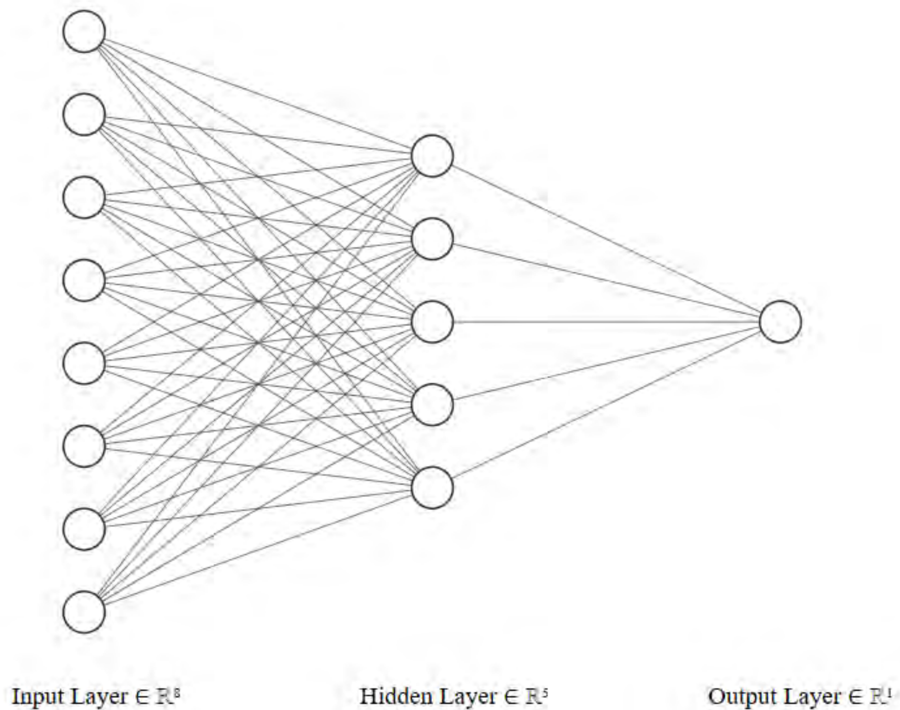

Figure 2.1: The graphic shows a fully connected NN with a depth of 3 and a configuration of 8-5-1 neurons at the layers. Each of the circles corresponds to a neuron in the network

chaining of these functions proposes the term network, whereas each function $f^l, l \in (1, d)$ proposes a layer of the network. Each layer is furthermore composed of single neurons, and the number of neurons matches the dimensionality of the layer function f^l. The number of nodes in each layer determines the width of the network, while the number of layers determines its depth [52].

The goal of training is to find the values of θ that minimize a specified loss function, which measures the discrepancy between the predicted output and the desired output [52].

The mapping f can be presented by an acyclic graph, either using the layer functions as nodes or all single neurons [52]. An example graph with single neurons as nodes is shown in Figure 2.1

2.3 Convolutional Neural Networks

CNNs are a specialized type of NNs designed to process grid-like data, such as time-series, images, or volumes, by training convolutional kernels [52]. Instead of the general matrix multiplication used in traditional NNs, CNNs employ the linear mathematical convolution operation, which is also used in algorithmic image and signal processing [54]. This leads to Goodfellow's definition: "Convolutional networks are simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers." [52, p. 330].

Using filters and masks in algorithmic digital image processing is inspired by the way the human visual cortex processes information. The convolution of a signal with a kernel as a masking operation is similar to how the receptive field processes stimuli [54]. Similar to these hand-designed algorithms, convolutional layers extract hierarchical features, which are then used in fully connected layers for classification or encoding in the case of AEs [53].

The ground-breaking work of Le Cun *et al.* [31] in 1989 introduced such CNNs and revolutionized machine vision, inspiring a generation of researchers and enabling them to achieve excellent results. In 2019, Yann LeCun, Yoshua Bengio, and Geoffrey Hinton were awarded the Turing Award in the field of artificial intelligence for their contributions [53]. Today, CNNs are the go-to approach for image- and signal-related tasks.

2.3.1 The Discrete Convolution Operation

The convolution operation is a mathematical operation on two real-valued functions that is widely used in various fields, including signal processing, image processing, and machine learning [52]. In the context of CNNs, the convolution operation is used to process grid-like data such as images, time series, or volumes.

Images or volumes and their convolution kernels used during processing are multidimensional arrays and therefore discrete. The discrete convolution of two one-dimensional signals is defined in the following.

Definition 2.10. ***Discrete one-dimensional Convolution [54]***
Let x1, x2 be two discrete one-dimensional real-valued signals. Then the convolution operation is given by:

$$x_1[n_1, n_2] * *h[n_1, n_2] = \sum_{k_1=-\infty}^{+\infty} \sum_{k_2=-\infty}^{+\infty} x_2[k_1, k_2] \cdot x_1[n_1 - k_1, n_2 - k_2]. \tag{2.9}$$

In practice, assuming every point outside the finite set is zero, the infinite summation can be implemented as a summation over a finite set [52]. Thus leading to the definition of the discrete convolution for an n-dimensional signal as:

Definition 2.11. ***Discrete n-dimensional Convolution***
Let x1, x2 be two discrete n-dimensional real-valued signals. Then the convolution operation is given by:

$$x_1[n_1, ..., n_m] * *x_2[n_1, ..., n_m] = \sum_{k_1} ... \sum_{k_m} x_2[k_1, ..., k_m] \cdot x_1[n_1 - k_1, ..., n_m - k_m]. \tag{2.10}$$

By applying the convolution operation with different convolution kernels at the layer level, a CNN learns to extract various feature maps of the input, leading to hierarchical features in subsequent layers. The convolution operation is a key component in the design of CNNs and is responsible for their ability to process grid-like data effectively [53].

2.3.2 The Convolution Layer

The convolution layer is the most important building block of CNNs. The filters learn to detect specific patterns or features in the input data, such as edges, corners, textures, or shapes. The convolution layer thus learns to extract hierarchical features from the input, with the lower layers detecting low-level features and the higher layers detecting more complex features that are combinations of the lower-level features [52, 53].

In such a layer, the matrix multiplication of the perceptron is replaced with a convolution operation, resulting in a trainable filter kernel. Multiple kernels are applied by convolutions to the layers input, outputting a feature map for each kernel. The trainable weights in the kernel correspond to the weight vector in feed-forward networks or perceptrons, respectively. To better see the similarities, the multiple sums can be transferred into one big sum using new one-dimensional indices. As the second signal h becomes the trainable filter kernel, its values are furthermore simply denoted as w alike. The n-dimensional convolution can then be rewritten like this:

$$\sum_{k_1} ... \sum_{k_m} x_2[k_1, ..., k_m] \cdot x_1[n_1 - k_1, ..., n_m - k_m] = \sum_{k} w_k x_k. \tag{2.11}$$

By adding bias to the calculation and wrapping everything up with an activation function, the basic convolution layer with no padding and stride one is obtained. For practical applications, mostly the same padding is used, which simply puts a padding around the input input such that the output feature maps are of the same size as the input images. Strides, on the other hand, correspond to the step size of the summation indices or simply the step size of the performed filtering and can act as an alternative to using pooling layers. For more information on padding and strides, readers are referred to [55].

2.3.3 The Pooling Layer

The convolution operation is complex and both heavy in calculation time as well as in memory usage. To reduce the amount of processing data, a form of sub-sampling is needed in practice [54]. As the basic sub-sampling algorithm (simply omitting every n^{th} value) leads to poor

performance, so-called pooling algorithms are mostly used. These algorithms calculate a new value based on their local neighborhood [54].
An alternative to mention are all convolutional networks. They rather hide the sub-sampling directly in the convolution by iterating with step sizes of more than oner over the input indices than using explicit pooling layers (see for example [56]).

Definition 2.12. ***The 2D Pooling Operation***
Let the step size be s and the input feature map m be rectangular of size $n \times n$. Then m is divided into $k = i^2, i = n/s$ rectangle pooling regions $r_{0,0}, ..., r_{i-1,i-1}$. Each of these pooling regions builds a group with $|r_{l,j}| = n^2, l, j \in (0, i-1)$ input activations. Then pooling of the specific operator ϕ calculates

$$\forall l, j \in (0, i-1) : p_{l,j} = \phi(r_{l,j}) = \phi \begin{bmatrix} m_{sl,sj} & \cdots & m_{sl,sj+s-1} \\ \cdots & \cdots & \cdots \\ m_{sl+s-1,sj} & \cdots & m_{sl+s-1,sj+s-1} \end{bmatrix}, \tag{2.12}$$

where sl and sj denote the indices of the top-left corner of each pooling region.

The result is a sub-sampled $i \times i$ matrix p. The operation can also be seen as shifting a pooling kernel across the input feature map, comparable to filtering with masks. Higher-dimensional pooling is performed accordingly. The most common pooling algorithms are:

- average or mean pooling [31]: $\phi(r) = mean(r)$
- max pooling [57]: $\phi(r) = max(r)$
- Lp pooling [58]: $\phi(r) = \left(\frac{1}{k} \sum_{l \in r} a_l^p\right)^{\frac{1}{p}}$, a_l being the feature value at position l in r

2.4 Autoencoder

AEs are a type of NN that have gained a lot of attention in recent years due to their ability to convert high-dimensional data into low-dimensional codes. Basically, AEs are just a special case of NNs consisting of two symmetrically connected networks and can be trained alike [52].

The concept of an AE can be traced back to the early days of perceptron-based NNs. Rumelhart *et al.* [59] describes an AE like NN as early as 1985, but without a small central layer. Goodfellow *et al.* [52] in contrast to their standard literature book "deep learning" Ballard [60] as first theoretical contributions [61]. However, it wasn't until 2006 that AEs was implemented as a symmetric NNs with a small central layer by Hinton & Salakhutdinov [33].

Since then, AEs has become the go-to approach for dimensionality reduction, feature learning, and representation learning, outperforming previous hand-designed algorithms like JPEG for image compression [38].

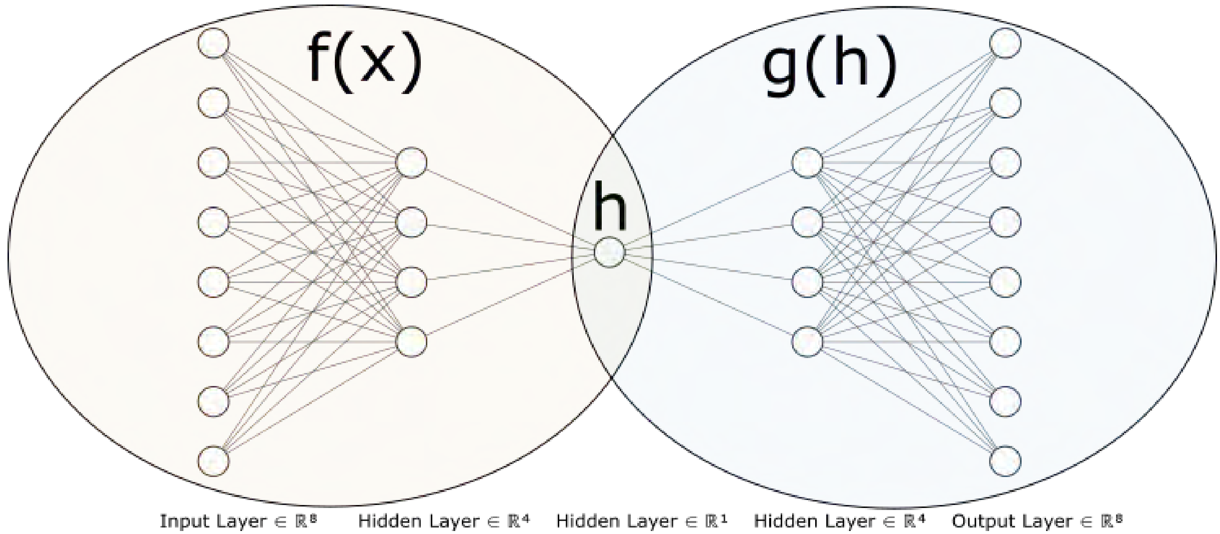

Figure 2.2: The graphic shows a deep fully connected undercomplete autoencoder with a total of five layers. The configuration of 8-4-1-4-8 neurons at the layers is chosen randomly. The marked left circle defines the encoder $f(x)$, the right circle the decoder $g(h)$.

2.4.1 Feed-Forward Autoencoder

An AE is comprised of two connected feed-forward networks that build a larger network. As shown in Figure 2.2, a feed-forward AE consists of an encoder function $h = f(x)$ and a decoder function $r = g(h)$. The encoder function compresses the input x into a lower-dimensional representation called the encoding or code h. This last layer of the encoder, the latent layer, acts as the input layer for the decoder function. The decoder function then tries to reconstruct the original input x from the code h. The decoder, generally (it is common practice but not mandatory), is a network symmetric to the encoder. For compression-related problems, one is generally interested in the received encoding h, not the reconstruction [52].

The overall computation of an AE is therefore given by $y = g(f(x))$, where y is the output of the network. However, this implies the problem of an AE learning to just copy the input. To address this problem, it is mandatory to restrict the AE during training to only copy approximately and only copy input that resembles training data [52]. A variety of restrictions exist for feed-forward AE, however, all of them force the model to prioritize which aspects of the input should be copied, leading to the learning of useful properties of the data [52].

The easiest way to prevent the AE from learning the identical function is to restrict the dimensionality of the encoding h, known as Undercomplete AE. "Learning an undercomplete representation forces the autoencoder to capture the most salient features of the training data" [52, p. 503].

Definition 2.13. ***Undercomplete Autoencoder***
An Undercomplete AE is an AE that fulfils the property:

(i) $dim(h) < dim(x)$,

where $dim(\cdot)$ *denotes the dimensionality of a vector.*

The learning process of an AE involves minimizing a loss function. As AE try to reconstruct the input, the loss function penalizes $g(f(x))$ for being dissimilar from x [52]. The weights are then updated during back-processing to minimize $L(x, g(f(x)))$.

Common loss functions such as Mean Squared Error (MSE) or Mean Absolute Error (MAE) can be used for L. An overview of loss functions is provided in section 2.6. Since the input is used as a reference for the loss, AEs are a form of unsupervised learning that can be applied to unlabeled data [52].

The AE is capable of learning a powerful non-linear generalization of PCA, especially with non-linear encoder and decoder functions [52]. However, to achieve proper functioning, it is essential to set suitable dimensions for all hidden layers. When the AE has too much capacity, it may simply copy the input data and fail to extract useful information. In contrast, an AE with insufficient capacity may not be able to reconstruct the input well and is unable to learn a good encoding function [52]. Even an AE with one latent dimension and a powerful non-linear encoder and decoder could theoretically learn the function:

$$f(g(x)) = \begin{cases} f(x) = i & \forall x^{(i)} \\ g(i) = x^{(i)} & \forall i. \end{cases} \tag{2.13}$$

Such an AE thus performs a copying [52]. On the other side, an AE with too little capacity fails to reconstruct the input well and thus fails to learn a good encoding function.

2.4.2 Convolutional Autoencoder

Convolutional Autoencoders (CAEs) combine the two concepts of AEs and CNNs. While the encoder part of a standard AE is a feed-forward network, the encoder of an CAE is a CNN, which is better suited for image data. To reconstruct the input data, the decoder part of the CAE requires transposed operations for the convolution and pooling layers used in the encoder. This can be achieved through the use of transposed convolution layers, and unpooling layers, respectively.

The transposed convolution is basically a convolution operation, but with the forward and backward passes switched. Depending on the settings of the convolutions in the encoder, different paddings might be needed for implementation. For more detailed information on the transposed convolution, the reader is again directly referred to [55]. The unpooling operation also depends on the pooling algorithm used in the encoder. For average pooling, the basic upsampling algorithm of duplicating each value by the desired amount, also known as the nearest neighbor, is sufficient [62]. For max pooling, the value is usually set at a fixed place in the output image and filled with zeros around it. Sometimes the place is also chosen randomly. The version of always using the upper left corner is known as bed of nails [62]. To restore more spatial information, a more advanced algorithm called what-where pooling was proposed in [63]. It restores the value at the right position during unpooling, but it needs a special max pooling to also store the position information in the first place.

The combination of convolution and deconvolution layers in the CAE allows for the learning of compressed representations of image data while preserving important spatial information [52]. Since the encoder and decoder of CAEs are CNNs, they can be trained end-to-end using backpropagation, making them suitable for large-scale image processing tasks [64].

2.5 Activation Functions

Having presented the abstract, high-level concepts used in this thesis, it is time to dive deeper. The upcoming sections take a detailed look at the building blocks of NNs and especially of CAEs, which are crucial for their performance. As a start, the absolute essential activation functions are presented. Without an activation function, the neurons of an NN would output a linear function. This would limit the capability of NNs to that of a linear regression model, which further implicates the need for non-linear activation functions [65].

An activation function takes the weighted sum of inputs and biases to produce an output, which decides the activation of the neuron. Mostly simple functions are used, which have a well-defined suitable derivation to aid the gradient processing during back-propagation [66]. Over time, a lot of different activation functions have been proposed. The most commonly used and most promising ones for the task are presented starting at section subsection 2.5.3. Furthermore, the developments that led to the state of the art are presented in the upcoming sections. For further information and more activation functions like the very common Softmax function in classification tasks, the reader is referred to Nwankpa *et al.* [67].

2.5.1 Trivial Activation Functions

Binary Step Activation

A mandatory requirement for an activation function is to either produce an activated or deactivated neuron. The simplest activation function is therefore a binary threshold function, namely the binary step function.

Definition 2.14. ***Binary Step Function [65]***

$$binary(x_k) = \begin{cases} 1, & x_k \geq 0 \\ 0, & x_k < 0 \end{cases} \tag{2.14}$$

The threshold of 0 can be set to a more suitable value. It is easy to implement and suitable for a simple linear binary classifier. It is not suited for multi-class classification, non-linear classification, or back propagation as the gradient is zero [65].

Linear Activation

According to Sharma et al. [65]: "The main drawback of the binary step function was that it had zero gradient because there is no component of x in binary step function". Again, just using the next simplest function available gives the linear activation function.

Definition 2.15. ***Linear Function***

$$lin(x_k) = \alpha x, \tag{2.15}$$

with the chosen constant scalar α. The linear function has a non-zero gradient; however, the value of the gradient is the same for every value. The weights and biases will be updated, but not the errors. It is therefore only suitable for simple tasks or when interpretability is required [65].

2.5.2 S-Shaped Activation Functions

The linear function made clear that a good activation function needs to update the weights and biases such that learning happens and the error reduces. Back propagation uses the derivative; big differences should result in bigger updates, and complex problems should be possible to solve. Therefore, the activation function should be non-linear but fully differentiable, with a derivative centered at null.
S-shaped functions like the sigmoid function or the Hyperbolic Tangent (tanh) match those wanted properties quite well, with their derivative being Gaussian-shaped. Even more, they transform the values into a restricted range while deriving. This resulted in S-shaped functions being the dominant activation functions in NNs for a long time [65]. There exist more improved variants of the presented basic functions, for which the reader is referred to [67] if interested.

Sigmoid

The classic s-shaped functions in machine learning are the sigmoid functions. Mostly the logistic sigmoid activation function, also referred to as the squashing function or Expit, is used [67]. It differs slightly from the basic sigmoid function by a scaling factor to make it more symmetric about zero [65]. Figure 2.3 shows the graph of the function and its derivative.

Definition 2.16. ***Logistic Sigmoid***

$$\sigma(x_k) = \left(\frac{1}{1+e^{-x_k}}\right) \tag{2.16}$$

The derivative of the Logistic Sigmoid is ([68] for detailed steps):

$$\sigma'(x_k) = \frac{e^{-x_k}}{(1+e^{-x_k})^2} = \sigma(x_k)(1-\sigma(x_k)). \tag{2.17}$$

The logistic sigmoid function and its derivative are easy to understand and apply. It delivers good results in shallow architectures, but the gradient is only updated in one direction due to the value transformation between zero and one. This leads to slow convergence and saturated neurons in deep architectures with problems like the "vanishing gradient problem": activation gradients die, hence stopping the training process itself [66]. The sigmoid activation function is still sometimes used in deep learning today, but it appears only at the output layer[67].

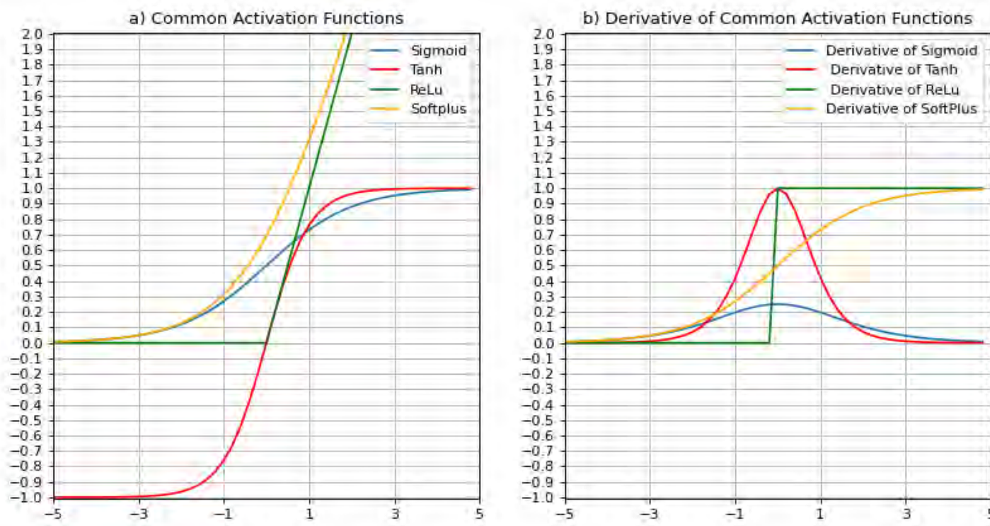

Figure 2.3: Graphic a) shows the plot of the common activation functions Sigmoid, tanh, ReLu, and Softplus. The shifting relation between the sigmoid and the tanh function can easily be seen in it. It also clearly shows that ReLu is a combination of a binary step function and a linear function, and SoftPlus is a smoothed version of Relu. Graphic b) shows their corresponding derivatives. The S-Shape function shows its expected Gaussian-shape form. For the not fully differentiable ReLu function, relu'(0)=0 is used, resulting in a binary step function.

Hyperbolic Tangent

A major problem with the sigmoid function is its transformation into zero-one space. To address the problem, a new function is formed by shifting the sigmoid function. The newly created function Hyperbolic Tangent (tanh) ranges between negative one and one. The graphic Figure 2.3 shows the tanh function in red. It can be written in a few different forms, of which the last one is the most commonly used.

Definition 2.17. ***Hyperbolic Tangent***

$$tanh(x_k) = 2 \cdot \sigma(2x_k) - 1 = \frac{1 - e^{-2x_k}}{1 + e^{-2x_k}} = \left(\frac{e^{x_k} - e^{-x_k}}{e^{x_k} + e^{-x_k}} \right) \tag{2.18}$$

The derivative is ([69] for detailed steps):

$$tanh'(x_k) = 1 - tanh(x_k)^2. \tag{2.19}$$

The tanh function is non-linear and bounded. The gradient is steeper than the sigmoid gradient. But most important, it is symmetrically centered around zero [70]. This provides better training performance in deep architectures, as it is less complex in the back-propagation process. The function was broadly used in recurrent NNs for natural language processing and

speech recognition [67].
It still suffers from the vanishing gradient problem and produces some dead neurons, as it can only attain a gradient of one when the input value is zero.

2.5.3 Rectified Linear Units

Rectified Linear Unit (ReLU)

The ReLU function itself was first used for NNs in 1975 by Fukushima [71]. The name "Rectified Linear Unit" however, derives from the 2010 paper of Nair & Hinton [72]. The function successfully eliminates the vanishing gradient problem of the earlier sigmoid and tanh functions. Until now, it has been the most commonly used activation function and absolutely dominates as an activation function in the hidden layers of practical deep learning applications [67].
The function is a very simple one and is basically a combination of a binary step function and a linear function. It rectifies inputs less than zero by simply setting them to zero and keeps values greater than zero.

Definition 2.18. ***Rectified Linear Unit***

$$relu(x_k) = \max(0, x_k) = \begin{cases} x_k, & if x_k \geq 0 \\ 0, & if x_k < 0 \end{cases} \tag{2.20}$$

The derivative is:

$$relu'(x_k) = \begin{cases} 1, & if x_k > 0 \\ 0, & if x_k < 0 \\ undefined, & if x_k = 0 \end{cases} . \tag{2.21}$$

It is important to note that the derivative of ReLU is undefined for the case $x = 0$ in terms of analysis. As the derivative is necessary for back-propagation, it is mostly set to 0 in the field of NNs (e.g., in TensorFlow). The plotted graph of ReLU and its derivative is shown in green in Figure 2.3 together with the previously presented activation function, and in black with other ReLU based activation functions in black in Figure 2.4. The set value $f'(0) = 0$ is used for derivation, as mentioned.
ReLU offers better performance in deep learning than sigmoid and tanh due to its simplicity. It does not compute exponentials or divisions in either direction. ReLU furthermore offers more generalization and produces sparse hidden units as it forces the values between zero and maximum [67]. It solves the biggest problems of sigmoid and tanh, but still has its shortcomings. It is prone to overfitting and sometimes fragile during training, due to the dead gradient problem. As negative values are strictly set to zero, gradients can become trapped in a zero update loop, and therefore neurons give zero activation [67].

Softplus

Another function to mention here is Softplus, presented by Dugas *et al.* [73].

Definition 2.19. ***Softplus***

$$softplus(x_k) = \log(1 + e^{x_k}) \tag{2.22}$$

The function is a smoothed version of ReLU and is a primitive of the sigmoid function. It has a non-zero gradient, which stabilizes the performance of deep NNs [67] and decreases the possibility of neuronal death [70]. It is not often used due to its complexity, and is mostly applied in statistical applications [67]. The graph and derivative can be seen in Figure 2.3 in orange.

Parametrized ReLU and Leaky ReLU

As setting negative values simply to zero sometimes results in a zero update loop, a few other ReLU variants were introduced. For leaky Rectified Linear Unit (leaky ReLU) L. Maas *et al.* [74] replaced the negative side of the function with the extremely small linear function $0.01x$. This results in a binary linear function. The small value is now mostly replaced by a fixed parameter, α.

Definition 2.20. ***Leaky ReLU***

$$lrelu(x_k) = \begin{cases} x_k, & if x_k \geq 0 \\ \alpha x_k, & if x_k < 0. \end{cases} \tag{2.23}$$

Common values are $\alpha = 0.2$ (keras standard value) or $\alpha = 0.3$ (TensorFlow standard value).

In 2015, He *et al.* [75] replaced the simple scalar α with a vector $a = a_0, ..., a_n$, learned while training. This results in a parametrized linear function known as Parametric Rectified Linear Unit (PReLU).

Definition 2.21. ***Parametrized ReLU***

$$prelu(x_k) = \begin{cases} x_k, & if x_k \geq 0 \\ a_k x_k, & if x_k < 0 \end{cases} \tag{2.24}$$

Both versions seem to deliver promisingly better results in so far tested cases than ReLU. Furthermore, a randomized ReLU function as well as an S-shaped ReLU were proposed, but they lack deeper research [67].

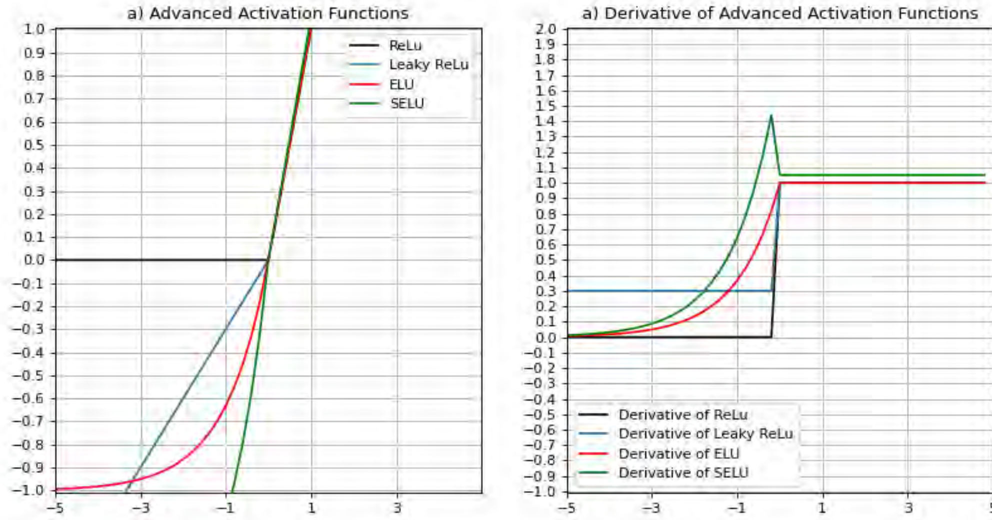

Figure 2.4: Graphic a) shows the plot of ReLu and its different proposed variations, namely leaky ReLU, ELU and SELU. For leaky ReLU $\alpha = 0.3$ is chosen, the standard used in numpy_ml and TensorFlow. The scale was set differently for the other figures to better highlight the differences, as the positive values stayed the same for all of them. Graphic b) again shows their corresponding derivatives. Interesting to mention is the spike at around zero for SELU, with values greater than one. The parametrized versions of ReLu and ELU are omitted in the graphic due to practical problems with drawing them. As their parameters are learned as hyperparameters during training, their graphs change as well.

2.5.4 Exponential Linear Units

Whereas the previously presented methods replaced the negative part of ReLU with a linear function, Exponential Linear Unit (ELU) and its variants replaced the negative side with exponential functions.

Exponential Linear Unit (ELU)

The basic ELU function was presented in 2015 by Clevert *et al.* [76].

Definition 2.22. ***Exponential Linear Unit***

$$elu(x_k) = \begin{cases} x_k, & if x_k > 0 \\ \alpha(\exp(x_k) - 1), & if x_k \leq 0 \end{cases} \tag{2.25}$$

Whereas α is a hyperparameter to control saturation for negative net inputs. ELUs derivative

is given by:

$$elu'(x_k) = \begin{cases} 1, & if x_k > 0 \\ elu(x_k) + \alpha, & if x_k \leq 0 \end{cases}. \tag{2.26}$$

"In contrast to ReLUs, ELUs have negative values which pushes the mean of the activations closer to zero. Mean activations that are closer to zero enable faster learning as they bring the gradient closer to the natural gradient [...]" [76, p. 5]. The ELU function provides more robust representations, faster learning, and better generalization compared to ReLU variants, but it suffers from known problems with other activation functions as values are not centered at zero [67].

Parametric Exponential Linear Unit (PELU)

To solve the shortcomings of ELU Trottier *et al.* [77] introduced PELU. It differs by two additional parameters, β and γ and is given by:

Definition 2.23. ***Parametric ELU***

$$pelu(x_k) = \begin{cases} \gamma x = \frac{\alpha}{\beta} x_k, & if x_k > 0 \\ \alpha(\exp(\frac{x_k}{\beta}) - 1), & if x_k \leq 0 \end{cases} \tag{2.27}$$

The equation $\gamma = \frac{\alpha}{\beta}$ is derived by the constraint of always being differentiable at $x_k = 0$. As PELU provides fewer bias shifts and vanishing gradients, it seems to be a good option for convolutional NNs.

Scaled Exponential Linear Unit (SELU)

Another novel variant of ELU is SELU presented by Klambauer *et al.* [78] in 2017. Its goal is to receive a self-normalizing NN, by normalized activations that cannot be derived from any other presented activation function. Activations of a NN are normalized if the mean and variance across samples are within predefined intervals, which results in them being transitive.

Definition 2.24. ***Scaled ELU***

$$selu(x_k) = \lambda \begin{cases} \frac{\alpha}{\beta} x_k, & if x_k > 0 \\ \alpha(\exp(x) - \alpha), & if x_k \leq 0 \end{cases} \tag{2.28}$$

With $\lambda > 1$ being a slope control parameter. This ensures a slope larger than one for positive net inputs. The parameters α and λ are constants; as a result, the authors propose to initialize weights using Lecun-Normal initialization [78] $w_{ij} \sim \mathcal{N}(0, \frac{1}{fan_in})$.

2.6 Loss Functions

In section 2.5 an important building block of the forward processing in an AE network was presented. To enable the model to learn, it has to be changed over time in such a way that it adapts to the data. This learning process is archived by changing the weights of the neurons during back-propagation. In order to update the weights in this backward calculating process accordingly, it is mandatory to first evaluate the correctness of the current results. For this purpose, loss functions are used, mostly in the form of performance metrics.
Loss functions compute a derivation of the label value from the predicted result. The concepts are based on the ones of distance functions and metrics shown in section 2.1. In contrast to metrics, the notation here is changed to a and p for the actual value respectively predicted value. Loss functions consist of three parts: a point distance d as presented, a normalization $\mathbb{N}$ to enable comparison between multiple series of various dimensions, and an aggregation operator $\mathbb{G}$ to retrieve a single value [79].

2.6.1 Error Measures

The properties of a performance metric are mostly determined by the distance function used. Most common are performance metrics based on subtracting point distances, often familiar from the Minkowski metric and Lp norm. These are simply referred to as error measures [79], whereas the error is denoted as $\mathbb{D}$. Important error types are [79]:

- $\mathbb{D}1 = A_k - Pk$, the (magnitude of) error: The simplest way of determining a point distance is by just subtracting the predicted from the actual. It is efficient, uses the same units as the data, is easy to interpret, and, in many cases, is proportional to the business objective. Its greatest advantage is simultaneously its greatest disadvantage. The positive and negative values may diminish during the aggregation phase, which could lead to a falsely accurate result, but showing the magnitude and hence the direction of the error can be used to distinguish between overestimation and underestimation.
- $\mathbb{D}2 = |A_k - P_k|$, the absolute error: This calculation only uses positive values and therefore avoids falsely accurate results. As a result of this, the bias can no longer be derived. Apart from that, the error still sticks to the data units and is easily interpretable.
- $\mathbb{D}3 = (Ak - Pk)^2$, the squared error: This error also avoids negative values but no longer weights all values the same. This leads to strong errors being penalized more, while small errors are penalized less. It has good mathematical properties, like being continuously differentiable.
- $\mathbb{D}4 = \ln(P_k/A_k) = \ln(P_k) - \ln(A_k)$, the logarithmic quotient error
- $\mathbb{D}5 = |ln(P_k/A_k)|$, the absolute logarithmic quotient error

The logarithmic errors are only included for completeness. A detailed description is given in the cited literature of Botchkarev [79].

For image AEs the compare value is the input, precisely single-series vectors of continuous values including zeros. This would lead to problems in losses with normalization $\mathbb{N} = a_i^{-c}$ like mean percentage error [79]. There is also no need for the loss function to be dimensionless.

Hence, only scale-dependent error measures are presented in this thesis. They are sufficient for the task and defined by [79]:

Definition 2.25. ***Scale-Dependent Error Measure***
A performance metric L is called a scale-dependent error measure if and only if the following properties apply:

(i) L is an error measure, e.g. $\mathbb{D}$ *is a point-wise distance function based on subtraction*

(ii) L is scale-dependent, e.g. it is unitary normalized by $\mathbb{N} = 1$

The final phase of aggregating over a data set is accomplished by common aggregation functions like the arithmetic mean (referred simply to as mean in the following), the median, the geometric mean, the sum, or the harmonic mean [79]. The following sections present some of the most commonly used scale-dependent error measures. These use sum and mean aggregation and are based on the magnitude of the absolute and squared errors. For a complete overview of aggregation, normalization, and point distance techniques, the reader is referred to [79].

2.6.2 Sum Aggregated Scale-Dependent Error Measures

Sum aggregation is the most basic aggregation operator possible, without any averaging. As values are only summed together, there are no new properties added by this aggregation operator. There seems to be no commonly used loss based on the magnitude of error or sum aggregation, only the absolute and squared errors. These are denoted as L1 loss and L2 loss, respectively [80].

Definition 2.26. ***L1 Loss***

$$L_1(a,p) = \sum_{k=1}^{n} |a_k - p_k| = \sum_{k=1}^{n} e_k \tag{2.29}$$

Definition 2.27. ***L2 Loss***

$$L_2(a,p) = \sum_{k=1}^{n} (a_k - p_k)^2 = \sum_{k=1}^{n} e_k \tag{2.30}$$

2.6.3 Mean Aggregated Scale-Dependent Error Measures

The (arithmetic) mean aggregation is the most popular aggregation method used. It is simple to calculate, and the result value corresponds to the expected value of the error. The result, however, is strongly affected by outliers and skewed data, resulting in an asymmetrical distribution of data and extreme values [79].

The most basic loss function in this class is the Mean Bias Error (MBE), the mean aggregation of the magnitude of error e_k. In contrast to the previous section, there is therefore a magnitude-based loss in this class, but it is still rarely used in machine learning due to the canceling problem of the magnitude error [81].

Definition 2.28. ***Mean Bias Error (MBE)***

$$mbe(a, p) = \frac{\sum_{k=1}^{n}(a_k - p_k)}{n} = \frac{\sum_{k=1}^{n} e_k}{n} \tag{2.31}$$

More common is the MSE with the absolute error e_i which of course means aggregated.

Definition 2.29. ***Mean Absolute Error (MAE)***

$$mae(a, p) = \frac{\sum_{k=1}^{n}|a_k - p_k|}{n} = \frac{\sum_{k=1}^{n} e_k}{n} \tag{2.32}$$

Similar, the MSE is defined based on the squared error.

Definition 2.30. ***Mean Squared Error (MSE)***

$$mse(a, p) = \frac{\sum_{k=1}^{n}(a_k - p_k)^2}{n} = \frac{\sum_{k=1}^{n} e_k}{n} \tag{2.33}$$

2.7 Optimization

The optimization algorithm is responsible for the network's actual learning. The algorithm addresses the problem of minimizing or maximizing a function, called the objective function or criterion $J(\theta, X, A)$. In the case of deep learning, it is common practice to only use minimization criteria. The objective function is then referred to as a loss function or error function [52], as presented in the previous section 2.6. The predicted values are calculated from the given input dataset X and the parameters θ.

In most cases, this calculation is gradient-based. It is important to notice that the gradient calculation itself is not done by the optimization algorithm. The optimization algorithm only defines how to change the weights and is an important part of the bigger back-propagation algorithm. This algorithm calculates the loss and gradient, then adjusts the weights of each layer in accordance with the chosen optimization algorithm while propagating the error through the whole network. For more information on the back-propagation algorithm, the reader is referred to chapter 6.5 in [52, p. 204].

2.7.1 Gradient-Based Optimization

The basic gradient descent technique for reducing an objective function $J(\theta, x, a)$ was already introduced in 1847 by Cauchy. It reduces $J(\theta, x, a)$ by changing θ in small steps of the opposite sign, according to the derivative. For more details, see Chapter 4.3 in [52, p. 82].

Batch Gradient Descent

Batch gradient descent computes the gradient descent for the entire dataset X. This results in accurate calculation, but it is slow, intractable for big datasets depending on memory size, and does not allow for on-the-fly updates [82].

Definition 2.31. ***Batch Gradient Descent [82]***
Let $\theta^t \in \mathbb{R}^d$ be a model's parameter set at step t, $J(\theta, X, A)$ an objective function, and η the learning rate determining how big the update steps should be. The calculation of batch gradient descent for the entire dataset is then given by:

$$\theta^{t+1} = \theta^t - \eta \cdot \nabla_{\theta^t} J(\theta^t, X, a). \tag{2.34}$$

According to Ruder, "Batch gradient descent is guaranteed to converge to the global minimum for convex error surfaces and to a local minimum for non-convex surfaces" [82, p. 2].

Stochastic Gradient Descent

Stochastic gradient descent performs one update at a time, which reduces redundant computations for large datasets. It is much faster than batch gradient descent and makes online learning possible, but it causes the objective function to fluctuate [82].

Definition 2.32. ***Stochastic Gradient Descent [82]***
Let again $\theta^t \in \mathbb{R}$ be a model's parameters at step t, $J(\theta, X, A)$ an objective function, and η the learning rate. Furthermore, let x^i be a single training example from the dataset X with its corresponding label a^i. Stochastic gradient descent then computes the update:

$$\theta^{t+1} = \theta^t - \eta \cdot \nabla_{\theta^t} J(\theta^t; x^i; a^i). \tag{2.35}$$

The fluctuation enables exploration of better local minima but also complicates convergence to the global minimum as it continues to overshoot. When decreasing the learning rate slowly over time, stochastic gradient descent, however, converges to a local or global minima and therefore shows similar convergence behavior to batch gradient descent [82].

Mini-Batch Gradient Descent

The mini-batch gradient descent algorithm is a compromise between both previously presented algorithms by using subsets, or mini-batches, of the training examples. This reduces the variance and therefore fluctuation, while optimized matrix operations make it computationally efficient [82].

Definition 2.33. ***Mini-Batch Gradient Descent [82]***
Let $\theta^t \in \mathbb{R}$ be a model's parameters at step t, $J(\theta, X, A)$ an objective function, and η the learning rate. Furthermore, let $x^{(i;i+n)}$ be a batch of n training examples with its n labels $a^{(i;i+n)}$. Mini-batch gradient descent computes the update rule:

$$\theta^{t+1} = \theta^t - \eta \cdot \nabla_{\theta^t} J(\theta^t; x^{(i;i+n)}; a^{(i;i+n)}) \tag{2.36}$$

2.7.2 Momentum-Based Optimization

The classical gradient algorithms are known to possibly converge very slowly. To improve the speed, multiple methods have been proposed, like the conjugate gradient method or the momentum gradient method. Due to calculation difficulties, only the momentum method has been used until now [82].

Momentum

The basic momentum method includes a fraction γ of the previous update vector $\Delta_{\theta^{t-1}}$ to the calculated update Δ_{θ_t}. This accelerates the gradient descent in the right direction, which reduces oscillations [82].

Definition 2.34. ***Gradient Descent with Momentum [83]***
Let $\theta^t \in \mathbb{R}$ be a model's parameters at step t, $J(\theta, X, A)$ an objective function, and η the learning rate. Furthermore, let γ be the momentum term, usually set to 0.9 *and Δ_θ be the update vector. Gradient descent's update rule is then given by:*

$$\Delta_{\theta^t} = \eta \cdot \nabla_{\theta^t, X, A} J(\theta^t) + \gamma \Delta_{\theta^{t-1}} \tag{2.37}$$

$$\theta^{t+1} = \theta^t - \Delta_{\theta^t}. \tag{2.38}$$

The momentum term increases gradients in the same direction and decreases gradients in changing directions [82]. This averages short-axis oscillation while contributing to the long-axis and hence improves convergence speed [83]. The shown definition uses batch gradient descent just for simplicity. Momentum can be combined with either of the presented gradient descent algorithms and is mostly combined with mini-batch gradient descent if used in real networks.

Nesterov Accelerated Gradient

Nesterov accelerated gradient is an improvement of the basic momentum method, which implements a lookahead. By calculating $\theta^t - \gamma\Delta_{\theta^{t-1}}$, an approximation of the future position of the parameters is retrieved [82].

Definition 2.35. ***Nesterov Accelerated Gradient [84]***
Let $\theta^t \in \mathbb{R}$ be a model's parameters at step t, $J(\theta, x, a)$ an objective function, and η the learning rate. Furthermore, let γ be the momentum term and Δ_θ be the update vector. The Nesterov-accelerated gradient update rule is given by:

$$\Delta_{\theta^t} = \eta \cdot \nabla_{\theta^t} J(\theta^t - \gamma\Delta_{\theta^{t-1}}, X, A) + \gamma \Delta_{\theta^{t-1}} \tag{2.39}$$

$$\theta^{t+1} = \theta^t - \Delta_{\theta^t}. \tag{2.40}$$

Instead of first computing the gradient and taking an accordingly big step in the direction, like basic momentum, Nesterov's accelerated gradient first takes a big step, then calculates the gradient and makes a correction. This leads to better responsiveness and, hence, increased performance [82].

2.7.3 Adaptive Optimization

In 2011, Duchi *et al.* [85] published the paper "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization". Their presented algorithm, Adaptive Gradient Algorithm (Adagrad) soon became the basis for state-of-the-art optimization algorithms until now.

Adaptive Gradient Algorithm (Adagrad)

The Adagrad algorithm assigns frequently occurring features low learning rates and vice versa, which means it adapts the learning rate. In contrast to the previously presented procedural schemes, it dynamically uses knowledge from previous observed data [85].

Definition 2.36. ***Adagrad [85]***
Let $\theta_i^t \in \mathbb{R}$ be a model's parameter i at step t, let $g_{t,i} = \nabla_{\theta_t} J(\theta_i^t, X, A)$ be the gradient of the objective function with respect to the parameter θ_i a time step t and η the learning rate. Let $G_t \in R^{d \times d}$ be a diagonal matrix with the entries i, i being the sum of the squares of the gradients and ϵ a smoothing term to avoid division by zero. Adagrad then computes the update rule:

$$\theta_i^{t+1} = \theta_i^t - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i}. \tag{2.41}$$

Due to G_t being a diagonal axis, the update rule can further be vectorized for an efficient implementation [82].

Adadelta and RMSprop

The Adadelta Method [86] is an improvement of the Adagrad algorithm, addressing the two main drawbacks. According to Zeiler, these are "1) the continual decay of learning rates throughout training and 2) the need for a manually selected global learning rate" [86, p. 3]. It reaches this goal by restricting the window of the accumulated past gradients to a fixed size w. This sum is recursively defined due to efficiency reasons as a decaying average of all past squared gradients. [82].

Definition 2.37. ***Adadelta [86]***
Let $E[g^2]_t$ be the running average of squared gradients at time step t with another fraction γ (also usually set to 0.9) and the gradient g_t. Let furthermore $E[\Delta\theta^2]_t$ be another exponentially decaying average of squared parameter updates. Let ϵ be a smoothing term and RMS be the root mean squared error criterion. The Adadelta update rule is given by:

$$\Delta_{\theta^t} = \frac{\sqrt{E[\Delta\theta^2]_t + \epsilon}}{\sqrt{E[g^2]_t + \epsilon}} =^{approx.} \frac{RMS[\Delta\theta]_{t-1}}{RMS[g]_t} g_t \tag{2.42}$$

$$\theta^{t+1} = \theta^t - \Delta_{\theta^t} \tag{2.43}$$

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2 \tag{2.44}$$

$$E[\Delta\theta^2]_t = \gamma E[\Delta\theta^2]_{t-1} + (1-\gamma)\Delta\theta_t^2. \tag{2.45}$$

The Adadelta algorithm requires no setting of the learning rate and is insensitive to hyperparameters. It separates dynamic learning rates per dimension, while its computational overhead compared to gradient descent stays low. The results are robust to large gradients and noise. The method can handle a variety of architectures and is applicable in local and distributed environments [86].

A quite similar adaptive learning rate method to Adadelta is RMSprop, which is an unpublished algorithm that has been proposed by Hinton [87] in his Coursera class. The algorithms have been developed independently, addressing the same shortcomings as Adagrad. In short, RMSprop only uses the decaying average of the squared gradients [82].

Adaptive Moment Estimation (Adam)

The Adam algorithm basically combines the adaptive algorithms from before (Adadelt, RMSprop) with the Momentum algorithm [88].

Definition 2.38. ***Adam [88]***
Let m_t be an exponentially decaying average of past gradients similar to Momentum, and v_t be an exponentially decaying average of past squared gradients like in Adadelta. Then m' is the bias corrected first moment estimator, and v' is the bias-corrected second moment estimator. The Adam update rule is:

$$\theta^{t+1} = \theta^t - \frac{\eta}{\sqrt{v'} + \epsilon} m'_t \tag{2.46}$$

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \tag{2.47}$$

$$m'_t = \frac{m_t}{1 - \beta_1^t} \tag{2.48}$$

$$v' = \frac{v_t}{1 - \beta_2^t}. \tag{2.49}$$

According to Kingma & Ba [88] acadam performs better than other previous adaptive learning methods, which were shown empirically on a few different architectures. They also proposed a variant of Adam based on the infinity norm called AdaMax which hasn't been tested widely. Another proposed variant of the Adam algorithm is Nesterov-accelerated Adaptive Moment Estimation (Nadam) presented by Dozat [89]. Nadam changes the momentum term in Adam to the known Nesterov Momentum, thus combining both advantages.

2.8 Regularization

Regularization plays a huge role in machine learning, especially deep learning. In the past, regularization was mostly defined as any modification made to a learning algorithm to reduce its test error but not its training error [52]. More recent works like [90] however mention that many considered regularization techniques also reduce the training error and therefore give the more general definition: "Regularization is any supplementary technique that aims at making the model generalize better, i.e., produce better results on the test set" [90, p. 1].

These methods cover a huge collection of techniques, including changes to the activation function or optimizer and the whole network itself. The following sections only present a brief overview of important topics that are not included in standard building blocks and maybe later used regularization methods. For deeper information, the reader is referred to chapter 7 in [52, p. 228] and [90].

2.8.1 Data Augmentation

"The best way to make a machine learning model generalize better is to train it on more data" [52, p. 240]. A way to increase the limited training datasets in practice is to generate fake data and add it to the training set. This process is known as data augmentation or dataset augmentation [52].

Data augmentation is especially effective in object recognition tasks, as images can be transferred easily. Operations such as translation, rotation, scaling, or noise are trivial image operations. Due to images being high-dimensional data with a huge feature space, these extra training samples improve generalization quite effectively [52].

The possibilities of data regularization allow one to theoretically generate infinitely augmented datasets, as long as one doesn't disturb the dataset. A lot of more general transformations rely on disturbing with a stochastic parameter [90]. In the case of this thesis, not only augmented but also pure synthetic data is used and mixed with real-life data for reliable results.

2.8.2 The Regularization Term

The regularization term involves adding a penalty term to the objective function, which limits the capacity of the models. In deep learning, regularization can theoretically be applied to the weights or the activations of the models, but application to the weights is usual. The abstract regularization looks something like [52]

$$J\ (\theta, X, A) = J(\theta, X, A) + \alpha\Omega(\theta), \tag{2.50}$$

with a regularization penalty $\Omega(\theta)$, which relative contribution is weighted by the hyperparameter α. The most common norm penalties $\Omega(\theta)$ used are the L1 or L2 weight decay.

Definition 2.39. ***The L1 Regularization Term***
L1 regularization adds a penalty term to the objective function that encourages the model to use fewer features by driving some of the weights to zero. L1 regularization results in a more sparse representation in comparison with L2 regularization. The regularization term used is given by [52]

$$\Omega(\theta) = ||w||_1. \tag{2.51}$$

Definition 2.40. ***The L2 Regularization Term***
L2 is the most common regularization term used, which drives the weights closer to the origin. The term used is given by [52]

$$\Omega(\theta) = \frac{1}{2}||w||_2^2. \tag{2.52}$$

For a more detailed overview and exact analysis of the behavior of these regularization terms, the reader is referred to [52] chapter 7.

2.8.3 Dropout

Dropout was presented in 2014 by Srivastava *et al.* [91] as a simple way to prevent overfitting. "The term 'Dropout' refers to dropping out units (hidden and visible) in a neural network" [91, p. 1930]. This means each unit is deactivated by a fixed probability p at random, and all its incoming and outgoing edges are temporarily removed. This dropout of neurons is only performed during training, while all neurons and connections stay active at test time. The dropout technique therefore creates a subsample at each iteration, which can be seen as training a collection of 2^n thin subnetworks. All are using weight sharing, and all are getting trained very rarely.
Dropout is a broadly used technique in deep learning nowadays, but it has mostly been used in classification tasks. For a mathematical description of the Dropout neural network model, the reader is referred to the original literature [91, p. 1933].

2.9 Retrieval Performance Measures

After having presented all the theoretical background needed for designing the desired CAE based retrieval system, one finally needs a way to measure the performance of different configurations or systems. For this purpose, search engine metrics are used to evaluate the ability of the found retrieval systems to retrieve and rank relevant material in response to a query [92].

The fundamental retrieval performance measures are precision and recall. Whereas precision measures the accuracy of an information retrieval system, recall measures the coverage of the relevant documents. Most other, more sophisticated performance measures are either precision-based, recall-based, or a combination of both [92].

2.9.1 Recall

As mentioned, recall is a performance measure for the coverage of the relevant results of a retrieval system. Precisely, its value is the fraction of all relevant results retrieved [92]. As for the case of this thesis, recall itself as a fundamental measure is not of much interest. The thesis is focused on designing a CAE based method for similarity comparison suitable for volumes but does not implement a complete retrieval system retrieving only relevant results from a database. Instead, for evaluation, small datasets are used that are ranked completely,

and a predefined number of results are returned in ranked order. Thus, no decision is made for a cutoff of relevant results. Recall might, however, be used as part of the combined measures presented in subsequent subsections.

Definition 2.41. ***Recall [92]***
Let Q be a query, R be the set of relevant results in the whole searched document collection, and A be the retrieved document set. Let further $|A|$ and $|R|$ be the number of documents in A and R, respectively, and $|R \cap A|$ the number of documents both in R and A. Then the recall of the retrieval system is defined by

$$R = \frac{|R \cap A|}{|R|}. \tag{2.53}$$

2.9.2 Precision

Precision is a retrieval performance measure for the accuracy of an information retrieval system. Instead of the fraction of all relevant results retrieved by recall, precision is the fraction of retrieved responses that are relevant [92].

Definition 2.42. ***Precision [92]***
Let Q be a query, R be the set of relevant results in the whole searched collection, and A be the retrieved document set. Let further $|A|$ and $|R|$ be the number of documents in A and R, respectively, and $|R \cap A|$ the number of documents both in R and A. Then the precision of the retrieval system is defined by

$$P = \frac{|R \cap A|}{|A|}. \tag{2.54}$$

2.9.3 Average Precision

As the name average precision implies, this measure is the mean of the precision scores. As the score after each relevant document retrieved is taken, it actually combines recall and precision for ranked retrieval results. The measure is highly sensitive to the ranking of retrieval results, as higher-ranked responses contribute more to the average than lower-ranked responses [92].

Definition 2.43. ***Average Precision [92]***
Let again R be the set of relevant results of size $|R|$ in the searched collection, and $r \in R$ be the ranks of the retrieved relevant document. Let further $P(r)$ be the precision of the top r retrieved responses. Then the average precision of the retrieval system is defined by

$$AP = \frac{\sum_{r \in R} P(r)}{|R|}. \tag{2.55}$$

2.9.4 Mean Average Precision

The Mean Average Precision (MAP) as the name suggests, is the arithmetic mean of the average precision. The mean is calculated for the average precision values over a set of n

queries. MAP was the standard retrieval performance measure for over 25 years and is still the go-to approach, except for systems with incomplete relevance information [92]. For the sake of the thesis, MAP is more than sufficient and is thus the last performance measure presented.

Definition 2.44. ***Mean Average Precision [92]***
Let N be the evaluation set of $|N|$ queries. Let further $AP(n), n \in N$ be the average precision of a single query response n. Then the MAP of the retrieval system is defined by

$$MAP = \frac{1}{|N|} \sum_{n \in N} AP(n). \tag{2.56}$$

2.10 Wavelets

The first usages of wavelets date back until 1909, when Haar [93] proposed what is known as the Haar wavelet today. However, most of the basic research concerning wavelets was done much later by Ingrid Daubechies in the 1990s [94]. Since then, they have been widely used in various fields, including computer vision, image compression, and edge detection. Nowadays, a whole deck of wavelets exists, including Haar, Daubechies, Coiflet, Symmlet, and more [95]. In the case of this thesis, only the Haar wavelet is presented in detail, which has been proven to be an effective edge detector due to its ability to capture abrupt changes in a signal.

The wavelet transform is a mathematical technique used for signal processing and image analysis that is related to the Fourier transform and filter banks [96]. Concerning the discrete space, "The fast Fourier transform (FFT) and discrete Wavelet transform (DWT) are both linear operations that generate a data structure containing $\log_2 n$ segments of various lengths" [95, p. 53]. Their inverse matrix is the transposed matrix for both, which makes them interpretable as rotations in function space. While the FFT transforms signals into the sine and cosine spaces, the new basis domain of the wavelet transform is more complex, proposing an infinite set of possible basis functions. These more complicated basis functions are the so-called wavelets, or analyzing wavelets. While both functions are localized in frequency, wavelets are also localized in time or space. This additional localization in space is what makes wavelets useful when dealing with 2D or 3D images instead of one dimensional (1D) signals [95].

2.10.1 Analyzing Wavelet and Scaling Function

In algorithmic usage, usually a multi-scale analysis is performed to capture information at various resolutions. In the case of this thesis, however, only the analyzing wavelet and scaling function with the highest resolution in its filter representation are used as replacements for the first convolutional layer. Thus, a full definition of wavelets and the wavelet transform, including the wavelet decomposition by subsequent scaling, is omitted here. Interested readers on the whole wavelet topic are referred to [96].

For the case of the thesis, the definition is restricted to the most significant properties of the analyzing wavelet, $\psi(t)$ as follows: The analyzing wavelet can be seen as a high-pass filter concerning frequency analysis.

Definition 2.45. ***Characteristics of Wavelets [96]***
A function $\psi(t)$ is called a mother or analyzing wavelet if it fulfills the following two properties:

(i) $\psi(t)$ *is localized in time*

(ii) $\int_{-\infty}^{\infty} \psi(t)dt = 0.$

Furthermore, each wavelet has a corresponding scaling function, $\varphi(t)$. The scaling function is a low pass filter, which gives the complement (or missing) frequencies after filtering with $\psi(t)$. This makes the transform invertible [95].

2.10.2 Haar Wavelet

The Haar wavelet system basically decomposes a signal into the difference and the arithmetic mean of its neighbors. The definition for the 1D Haar wavelet and scaling function for signals in time is given in 2.46. Their corresponding plots are shown in Figure 2.5.

Definition 2.46. ***Haar wavelet [96]***

$$\varphi(t) = \begin{cases} 1, & \text{if } 0 \leq t < 1 \\ 0, & \text{otherwise} \end{cases} \quad \text{and} \quad \psi(t) = \begin{cases} 1, & \text{if } 0 \leq t < 1/2 \\ -1, & \text{if } 1/2 \leq t < 1 \\ 0, & \text{otherwise} \end{cases} \tag{2.57}$$

One usually further wants the wavelets to be normalized in regards to energy preservation, which means $||\psi||_2 = 1$. For a Haar wavelet operating in $R^n, n \in \mathbb{N}$ with scale s, this can be achieved by a simple prefactor [96].

Definition 2.47. ***Normalization of the Haar wavelet***
Let $n \in \mathbb{N}$ be the dimension of the operating space, and let further s be the scaling of the wavelet. Then the Haar wavelet can be L2 normalized by

$$a_{n,s} = 2^{-ns/2} \tag{2.58}$$

$$a_{n,1} = 2^{-n/2}. \tag{2.59}$$

2.10.3 Multidimensional Haar Wavelet

The normalization factor is already presented in correlation to the used dimension. As already mentioned, the 1D Haar wavelet calculates the mean and the difference of two neighbors, which is sufficient for reconstruction. In the 2D case, this expands to the mean of a $2x2$ square and three differences, and for 3D to the mean of a 2^3 volume and seven differences, respectively.

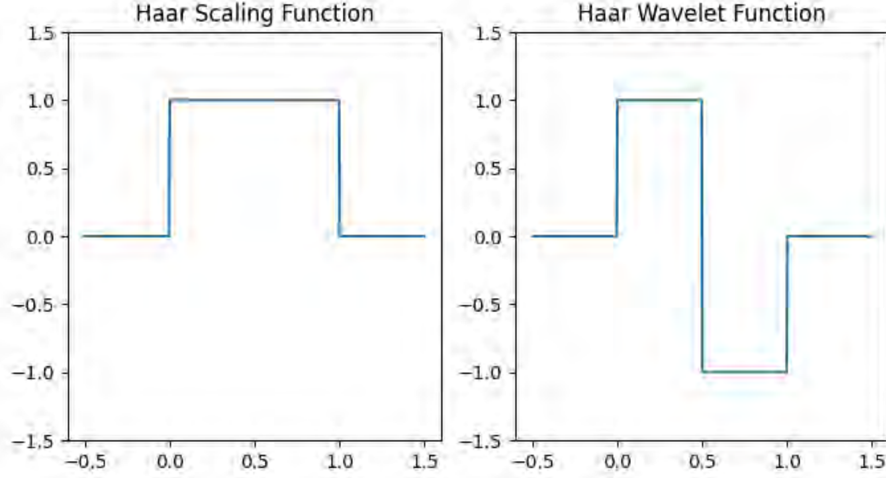

Figure 2.5: The graphic shows the plot of the Haar scaling function $\varphi(t)$ (left) and the Haar wavelet $\psi(t)$ (right). The presented plots are the basic versions with a scaling factor of one and normalization omitted. One can clearly see how the Haar wavelet fulfills both the localization in time as well as the zero integral properties of a wavelet.

In general, the operations are separable, and the multi-dimensional ones can be obtained by subsequent applications in the different dimensions. Thus, for the wavelet and scaling functions, a multi-dimensional version can be obtained by subsequent outer products[96].

Definition 2.48. ***Outer Product***
Given two vectors x and y of size $m \times 1$, then the outer product is defined as

$$\mathbf{x} \otimes \mathbf{y} = \begin{bmatrix} x_1y_1 & x_1y_2 & \dots & x_1y_n \\ x_2y_1 & x_2y_2 & \dots & x_2y_n \\ \vdots & \vdots & \ddots & \vdots \\ x_my_1 & x_my_2 & \dots & x_my_n \end{bmatrix}. \tag{2.60}$$

With the definition of the outer product and a rewriting of the wavelet and scaling functions to 1D filters in the form of $L_\varphi = [1, 1]$ and $H_\psi = [1, -1]$, the 2D Haar wavelets can now be obtained by [96]

$$\text{Approximation}: A = L_\varphi \otimes L_\varphi \tag{2.61}$$
$$\text{Vertical}: V = L_\varphi \otimes H_\psi \tag{2.62}$$
$$\text{Horizontal}: H = H_\psi \otimes L_\varphi \tag{2.63}$$
$$\text{Diagonal}: D = H_\psi \otimes H_\psi. \tag{2.64}$$

The resulting 2D Haar wavelets of the outer product calculation in filter matrices notation then are:

$$A = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \quad H = \begin{bmatrix} 1 & -1 \\ 1 & -1 \end{bmatrix}, \quad V = \begin{bmatrix} 1 & 1 \\ -1 & -1 \end{bmatrix}, \quad D = \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}. \tag{2.65}$$

In the same form, the 3D haar wavelets can be obtained by the outer product calculation:

$$\mathrm{W1} = L_\varphi \otimes L_\varphi \otimes L_\varphi \tag{2.66}$$
$$\mathrm{W2} = L_\varphi \otimes L_\varphi \otimes H_\psi \tag{2.67}$$
$$\mathrm{W3} = L_\varphi \otimes H_\psi \otimes L_\varphi \tag{2.68}$$
$$\mathrm{W4} = L_\varphi \otimes H_\psi \otimes H_\psi \tag{2.69}$$
$$\mathrm{W5} = H_\psi \otimes L_\varphi \otimes L_\varphi \tag{2.70}$$
$$\mathrm{W6} = H_\psi \otimes L_\varphi \otimes H_\psi \tag{2.71}$$
$$\mathrm{W7} = H_\psi \otimes H_\psi \otimes L_\varphi \tag{2.72}$$
$$\mathrm{W8} = H_\psi \otimes H_\psi \otimes H_\psi. \tag{2.73}$$

3 Method

The primary objective of this thesis is to devise an effective CAE architecture that can encode 3D voxel data such that, together with a suitable similarity measure, a retrieval algorithm can be obtained. To accomplish this, a four-step approach, as delineated in this chapter, will be followed.

The first step in this thesis is to define the term ”similarity” accurately and propose a suitable similarity measure that operates on 1D arrays. In the middle part of an CAE, the feed-forward AE is composed of fully connected layers. Before processing through these dense layers, the feature maps obtained from the convolutional layers are flattened since feedforward layers can only process 1D data. The latent dimension of an AE is also a fully connected layer, producing a 1D array as output or encoding. Therefore, the desired similarity measure operates on 1D vectors, regardless of the input dimension. A detailed definition of the similarity measure can be found in section 3.1.

In the second step, various network architectures for encoding 2D image datasets are proposed, and their performance is evaluated briefly afterwards. As datasets, a MNIST-like fashion product database (Fashion-MNIST) and a 2D shapes dataset known as dSprites are used. The architectures are optimized using the basic reconstruction error until a satisfactory solution is found. Various layer and hyperparameter configurations are tested on these two datasets and evaluated for their quality of outcome.
The middle part of an CAE architecture is nearly identical for 2D and 3D input data, as it works on flattened input. Therefore, the results for the middle feed-forward AE can be directly transferred to the later 3D architectures. While the number of neurons may differ due to the larger data size of 3D (the number of voxels versus the number of pixels), the relative compression rates should remain applicable. Furthermore, 2D and 3D convolution as well as 2D and 3D pooling share many similarities in terms of their calculation and parameters. Thus, the well-working parameter settings for 2D datasets should be mostly transferable to 3D shapes, especially if they share spatial similarities.
To enable accurate similarity comparisons between shapes, it is crucial that the encoding capture the necessary shape information. Therefore, the architectures are designed with retrieval suitability in mind. For the architectures with the best reconstruction, further quality evaluation of the encoding based on its performance with the proposed similarity measure on retrieval is done. This approach ensures that the encoding effectively includes the desired shape information for accurate similarity comparisons. Thus, this part also serves to evaluate the effectiveness of the proposed similarity measure. The groundwork laid in this section will be essential for the subsequent sections that focus on 3D data.
As the part is quite huge, it is split into two sections. In section 3.2 the theoretical design and TensorFlow implementations of the models are described, whereas section 3.3 presents the incremental tests carried out as well as their corresponding layer configurations.

Third, the previous results are used to design a working 3D network model in section 3.4. For

the scope of this thesis, it is difficult to accumulate enough data from different 3D CT scans to fulfill the task on real-life 3D data only. The model is therefore trained on increasingly difficult synthetically generated 3D shape datasets. On the way, the AE is incrementally tested and improved based on the results of various proposed performance tests. All tests are designed with regard to retrieval performance. In order to still provide a practical proof of concept at the end, a half-synthetic dataset is created and used. For this purpose, CT data is augmented by various transformations, to blend in with the purely synthetic shapes to create multiple classes of objects of comparable size and space.

Fourth and last, the previously found model is expanded to exploit the known edge detecting Haar wavelet filters in subsection 3.4.4. The newly created model is tested against the previous one in various domains.

3.1 Similarity Measure

There are numerous distance functions, similarity metrics, and measures available, each suitable for different purposes. In the background section, several distance functions suitable for vector comparison were discussed. When designing a similarity measure for a specific case, a lot of questions arise about the desired behavior of the measure.

AEs encode input data into a smaller descriptor describing the content of the input data. As they are trained on reconstruction, these descriptions not only contain the content inside but also encode rigid transformations such as translation, rotation, or scaling that need to be restored in the decoder. The same counts for non-rigid transformations. Although non-rigid transformations are also important, for the current thesis's desired use case, which is industrial CT scans, they can mostly be ignored.

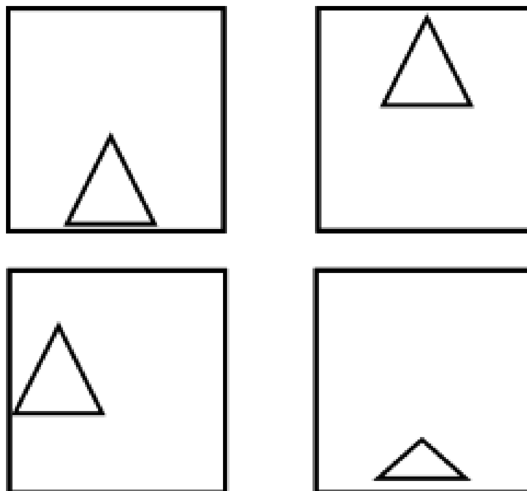

Figure 3.1: The graphic shows four different images, each with a triangle inside. The first triangle in the upper left image sits down in the middle; the second in the upper right is up in the middle; and the third is down left in the middle left. The fourth down right sits again down in the middle but is scaled differently. Which images are the most similar?

To better understand the problems that arise when designing a similarity measure for industrial CT scans, let's consider Figure 3.1. A human observer would assume that triangles one, two, and three are all the same but translated across the image, while triangle four is another yet somewhat similar triangle. However, suppose an AE encodes the images based on the left corner x, left corner y, length of the triangle, and height of the triangle in total pixels. In that case, triangle four would be the most similar to triangle one, regardless of using Manhattan or Euclidean distance. This is because the translation of all others is always nominally bigger than the difference in triangle height.

Unfortunately, the question of how to generally weight outliers in comparison to multiple smaller differences is a tough one to answer. The question has been studied extensively in regards to the L_k norm, with only reasonable success. [97] showed that the traditional approach of using the Euclidean norm fails in higher dimensions because the data becomes more and more sparse, causing traditional algorithmic techniques to fail. This issue has become known as the curse of dimensionality. Later in 2020, [98] showed that fractional norms and quasi norms do not offer any improvement either.

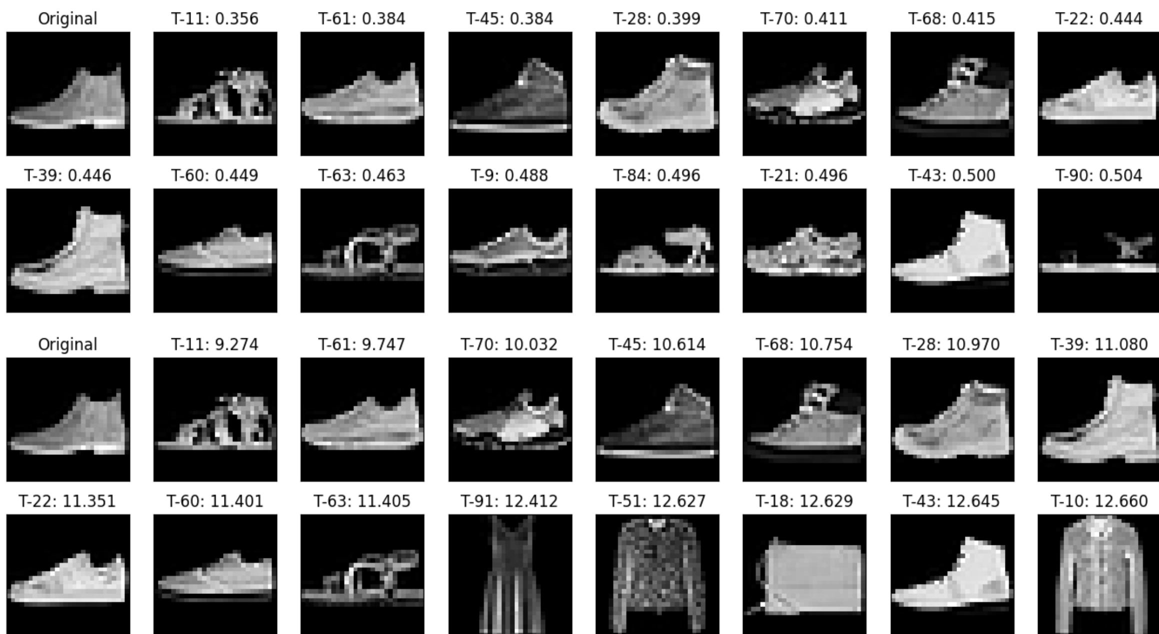

Figure 3.2: The graphic shows the different ordering received when directly applying the Euclidean (rows 1 and 2) or the Manhattan distance (rows 3 and 4) to the latent encodings received. To really see the differences, a very tiny subset of just 100 samples was used. Whereas the Euclidean distance still only returns shoes, the ordering if only looked at returned shoes is sometimes a bit worse than with the Manhattan distance.

A preliminary experiment conducted on one of the trained models for Fashion-MNIST confirmed the common belief that the Manhattan distance is more effective in producing well-ordered retrieval results locally. However, the Euclidean distance was able to better sort the images globally, especially to first return same class images even with substantial variations like objects that had large deformations. One example of the test is shown in Figure 3.2. Al-

though the Euclidean distance might be more robust against transformations, it becomes less and less effective in sorting highly similar objects accurately as the dimensionality increases. As the dimensionality tends to increase in more complex 3D scenarios, Manhattan more and more outperforms Euclidean. It is hard to say for now how much bigger the 3D encodings will be. As a starting point, the Euclidean distance is chosen, as in the tests performed, it was still superior. As additional adjustments are applied that improve the results unrelated to the distance function used, the advantages of Manhattan shrink further.

Since there is no information on how exactly an AE will encode the given data, a useful step is to normalize all values into the same space. As these are real-valued floats without a given maximum or minimum, a mapping function like tanh has to be used for this purpose. However, if one directly normalizes the input values with such a function, significant differences are reduced as the normalization is not linear. For the Euclidean distance, this reduces the weighting of the outliers and could lead to a weighting in between the two former distances and thus a better result. For Manhattan, the question arises as to whether one should prefer to normalize the absolute difference values of the encodings subtraction. To answer the question, a few more tests were conducted on the Fashion-MNIST dataset that briefly evaluated the two different approaches for both distance functions. Surprisingly, both performed worse when the difference was normalized, while both performances increased by normalizing the input values. The improved results obtained with the Euclidean distance together with the tanh normalization directly performed on the input values are shown in Figure 3.3.

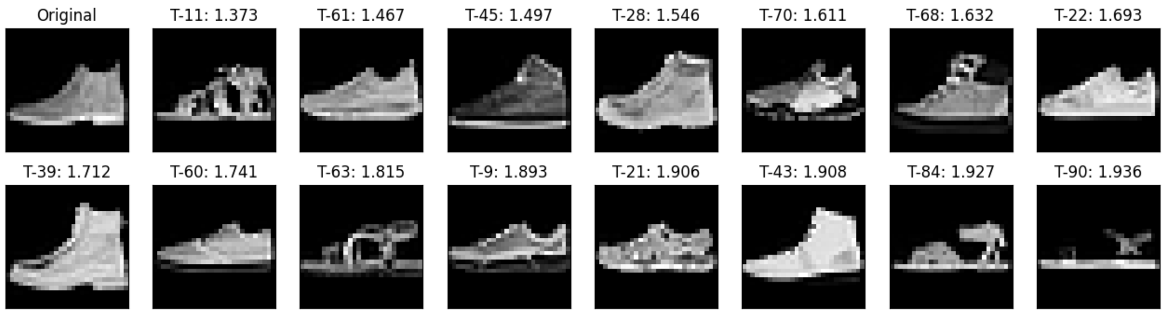

Figure 3.3: The graphic shows the order received when directly applying the tanh normalization to the input values before using the Euclidean distance.

All together, the Euclidean distance is used on the tanh squished values as it seems to be the most appropriate solution based on the empirical knowledge and pretests conducted. Switching to the Manhattan distance is left open for choice if the proposed similarity measure does not perform well in the higher-dimensional 3D encodings. The similarity measure is given by definition 3.1.

Definition 3.1. ***The tanh squashed Euclidean distance***

$$d_{teuc}(x, y) = \sum_{k=1}^{n} (tanh(x_k) - tanh(y_k))^2. \tag{3.1}$$

3.2 2D Model Design

In this section, the designed network and training architectures for the first 2D evaluations, as well as the code used for loading, training, and testing, are presented. The network and training architectures are designed to achieve the thesis' goal of efficient and accurate similarity retrieval. The aim is to develop a robust and efficient network for similarity retrieval on 2D images, which can then be mostly transferred to the 3D case. The section should provide a comprehensive understanding of the pretest architectures by presenting the design considerations along with the TensorFlow implementations and supporting code. To ensure that the code examples in the thesis remain relevant and concise, only select portions are presented.

The implementation is carried out using Python 3.10 inside PyCharm Professional 2021.3.2. The TensorFlow framework (GPU version 2.10.1) is utilized along with its integrated Keras Application Programming Interface (API) (tensorflow.keras) for implementing the CAEs. In addition, TensorFlow datasets 4.8.2 is used for testing the models on more complex datasets. Packages such as pickle, pathlib, scipy (version 1.7.1), numpy (version 1.22.3), and matplotlib (version 3.4.3) assist with data manipulation, loading and storing, computation, and visualization. To maintain organization, all packages are installed using a Conda virtual environment and the provided TensorFlow installation option.

3.2.1 Network Architecture

One of the most fundamental questions in designing an AE is determining its depth. According to Glorot & Bengio [99], the optimal depth for a feed-forward AE is typically five, with the exception of the sigmoid activation function. However, the choice of depth in CAEs can depend on various factors and is often based on empirical knowledge. For instance, it is commonly recommended to use two or three convolutional layers for feature extraction, followed by one or two dense layers for further compression. The exact amount varies depending on the complexity of the data being processed. Convolutions are performed with the same padding in this thesis, according to the empirical and TensorFlow standards. For exactly choosing the hyper-parameters of the fully connected and convolutional layers, various pretests are performed in 2D.

The second consideration addresses the use of pooling layers. "Once features are extracted, their exact location becomes less important as long as their approximate position relative to others is preserved." [100, p. 10]. Using some kind of pooling or down-sampling seems crucial for efficiency, and throughout the years, pooling has indeed provided a good solution for the task. According to Springenberg *et al.* [56] the vast majority of CNNs use max-pooling, hence self presenting an architecture where pooling is completely replaced by convolutions. Unfortunately, that only counts for supervised tasks.
Concerning CAEs in feature learning, Wang *et al.* [45] decided for average pooling, stating it preserves shape information well. Zhao *et al.* [63] presented an improved kind of max-pooling that also preserves the raster location and not only the maximum itself, skipping the information to the decoder. For CAEs in the feature learning position, information stays important.

Figure 3.4: The presented graphic illustrates one half of a CAE, specifically the encoder, which features an architecture similar to one used in later 2D tests on the Fashion MNIST dataset. The gray rectangle boxes represent the processed data at different layers. The input image of 28x28 pixels and depth 1 is located on the left-most side.
The blue boxes with red arrows in between represent the convolutions performed (3x3 kernels) or average pooling operations (2x2 kernels). The depth of the feature maps increases after the convolution steps as multiple feature maps are produced, while the pooling steps decrease the height and width of each feature map.
The black arrows on the right side indicate the flatten and fully connected layers. The decoder half would look similar, but mirrored to the right. The pooling operations become unpooling operations, the convolutions become transposed convolutions, and the flattening becomes a reshape.

For shape retrieval in an unsupervised field, it is clearly mandatory to preserve shape information while compressing. TensorFlow currently provides a max_pool_with_argmax layer that preserves the position of the maximum, but only for 2D data, and the appropriate unpooling operation for even using the provided argument is still lacking. TensorFlow, on the other hand, provides average pooling, including an 3D implementation, as well as an upscaling layer suitable for average-unpooling, which is used for the sake of this thesis. Strided convolution and deconvolution are considered alternatives in the case of average pooling not performing well enough; otherwise, they are left open for future research.

Addressing the decoder, the transposed convolution layers are used such that filter sizes are symmetrical, not the processing through the layers. This means, for example, that the transposed convolution layer for a convolution layer with filter size eight would also have filter size eight. Thus, an additional transposed convolution layer with filter size one is needed to regain a one-channel image at the end. This additional last transposed convolution layer can be seen as the symmetric corresponding layer to the input layer. This style of decoder layer design is commonly used and, for example, recommended in TensorFlow Tutorials [101].

In the algorithmic part of the model, only the activation function is left open to choice. Most state-of-the-art deep networks use the sigmoid function at the output layer, squashing the values in the positive range between zero and one. Such a configuration is useful for classifica-

tion, but to allow the last deconvolution to reconstruct the input data, activation is omitted at the output layer of AEs. In regards to the similarity measure of the latent vector, which uses a sigmoid function before comparison, it could be interesting to directly apply it as an activation function at the latent layer. As usually the hidden layer also omits an activation function, the applicability of directly integrating the sigmoid function in the hidden layer is evaluated in the 2D pretests.
Addressing the hidden layers, ReLU and its multiple variants seem to be good choices. SELU is omitted, as it comes with a lot of adjustments to make to get a normalized network and therefore is too complex for 3D data later. Computation time and complexity are crucial factors to keep in mind when dealing with voxel data. Thus, linear is preferred over exponential and non-parametric over parametric. So ReLU is used at the first starting point, then leaky ReLU is evaluated followed by PReLU and ELU. The more complex activation functions are only chosen if the improvement in results is significant enough to outweigh the longer computation times.

3.2.2 Network Implementation

The AE models are implemented in their own classes, which inherit from a base model class specific to the dataset. To improve code readability, self-defined aggregated layers are used in conjunction with pre-existing layers provided by Keras. Additionally, certain parameters are implemented as enums or given default values from utility classes, as is common in programming.

The Base Class

```
1  class AEBase(keras.models.Model):
2    DATASET_NAME_SHAPE = (X, Y, Channels)
3    # additional specific constants
4
5    def __init__(self): # , additional arguments
6      # additional specific attributes
7      self.encoder = keras.Sequential()
8      self.encoder.add(keras.layers.Input(shape=self.
           DATASET_NAME_SHAPE))
9      self.decoder = keras.Sequential()
10
11   def call(self, input_data):
12     encoded = self.encoder(input_data)
13     decoded = self.decoder(encoded)
14     return decoded
```

Python Code 3.1: The Autoencoder base class

The base class for the dataset-specific AE models encapsulates the underlying encoder and decoder structures, as well as standard parameters such as the input shape. This allows for a

more streamlined implementation of the models and ensures consistency across the dataset-specific classes.

When instantiated, the class initializes the encoder and decoder as separate sequential models and assigns the constant input shape to the encoder. This allows for easy access to the encoder for extracting the latent representation after the model is trained.

To enable training of the AE models using the standard Keras training interface, the call() method is overridden. It is worth pointing out that the method of writing is correct. One should never override Python's __call__() operator in Keras, as it is already implemented to keep track of internal state information. As it then forwards to the provided call() method, the desired behavior of the operator gets available despite not overriding it itself. The call() method takes a tensor as input and passes it through the encoder to get the latent representation. It then passes the latent representation through the decoder to obtain the reconstructed output. Finally, the reconstructed output is returned as the output of the call() method.

Self-Defined Layers

In Keras, there is a wide range of built-in layer types with various arguments and options that can be used to customize the network architecture. Before addressing the self-defined layers, a short overview of the included layers is given, as the newly introduced layers are based on them. In the case of 2D data, commonly used layers are Input, Flatten, Reshape, Activation, LeakyReLU, Conv2D, Dense, AveragePooling2D, Conv2DTranspose, and UpSampling2D.

The input layer specifies the shape of the input data, while the flatten layer flattens multi-dimensional data into an 1D array suitable for subsequent dense layers. The Reshape layer converts an 1D array back into a multidimensional format for further (transposed) convolution processing. These layers were only included for completeness. They are not used in the self-defined layers.

In most cases, the activation function can be directly specified as a parameter to the Conv2D, Conv2DTranspose, or Dense layers. In some cases, like for the leaky ReLU activation function, which requires an additional parameter *alpha*, the activation is implemented as a separate layer in TensorFlow. These activation functions, implemented as their own layers, cannot be passed as parameters. To restore the behavior of passing activation functions as a parameter to the layers, own layers with additional parameters, otherwise deriving the behavior from the Keras ones, are implemented. For implementing this behavior, the explicit activation layer is used.

Further, Conv2d and Conv2DTranspose are always followed by an AveragePooling2D or UpSampling2D layer after activation, respectively. To further streamline the model implementations, the pooling or upsampling operations are directly included in the chain of these new layers. In the case of the dense layer, only the functionality of passing layer activations as parameters is implemented.

To create custom layers in Keras, one can derive a new layer class from the Keras layer base class and override the constructor and call() methods. In this case, the initialization method initializes the three desired layers that will be chained together. For the Conv2D and AveragePooling2D layers, the arguments are simply passed on to the integrated layers. For the

desired activation function, a separate function is called, which either returns an activation layer with the specified activation function or a LeakyReLU layer if it is chosen. The three layers are then sequentially chained by calling them in the overridden __call__ method.

```
class ConvActPool(layers.Layer):
  def __init__(... expanded arguments):
    super().__init__()
    self.conv = layers.Conv2D(... pass forward arguments)
    self.activation = get_activation_layer(activation,
       leaky_alpha)
    self.pooling = layers.AveragePooling2D(... pass forward
       arguments)

  def call(self, input_tensor):
    x = self.conv(input_tensor)
    x = self.activation(x)
    return self.pooling(x)
```

Python Code 3.2: An implementation of a chained Convolution-Activation-Pooling layer

Three custom layers have been implemented based on this approach. The ConvActPool layer is used to replicate the Conv2DisActivation. AveragePooling2D chain from the example. The UpsConvTAct layer is used for the decoder chain of UpSampling2D, Conv2DTranspose and Activation. Finally, the DenseActivation layer is used for the usual dense activation chain. All three subclasses are implemented similarly to the example code provided. Thus, the code provides a flexible and easily customizable way to define a new layer type based on a layer chain in Keras.

An Example AE Network Model

```
class AE(AEBase):
  def __init__(self, alpha=STANDARD_LEAKY_RELU_ALPHA):
    super().__init__(alpha)
    self.create_encoder()
    self.create_decoder()
    self.compile(optimizer=OPTIMIZER_FM, loss=STANDARD_LOSS,
       metrics=STANDARD_METRICS)

  def create_encoder(self):
    # encoder definition here

  def create_decoder(self):
    # decoder definition here
```

Python Code 3.3: An example AE implementation

The code snippet presents an example AE with the basic structure of the constructor that

is common to all AE implementations. It includes an additional alpha argument for the LeakyReLU activation function that is passed on to the superclass initialization. The encoder and decoder methods are defined to create the corresponding layers and add them to the respective Keras sequential models.

An example encoder using the presented building blocks is shown in the next code snippet, consisting of two ConvActPool layers, a Flatten layer, two Dense layers with and without LeakyReLU activation, and a final Dense layer.

```
1 def create_encoder(self):
2   self.encoder.add(ConvActPool(8, kernel_size=3, leaky_alpha=
      self.alpha))
3   self.encoder.add(ConvActPool(16, kernel_size=3, leaky_alpha=
      self.alpha))
4   self.encoder.add(layers.Flatten())
5   self.encoder.add(DenseLeakyReLU(512, leaky_alpha=self.alpha))
6   self.encoder.add(layers.Dense(64))
```

Python Code 3.4: An example implementation of an encoder

The last code example shows the decoder definition consisting of an Input layer, two Dense-LeakyReLU layers, a Reshape layer, two UpsConvTAct layers, and a final Conv2D layer. The activation parameter of the self-defined layers is set to LeakyReLU as the standard, so it does not need to be explicitly defined.

```
1 def create_decoder(self):
2   self.decoder.add(layers.Input(shape=64)),
3   self.decoder.add(DenseLeakyReLU(512, leaky_alpha=self.alpha))
4   self.decoder.add(DenseLeakyReLU(784, leaky_alpha=self.alpha))
5   self.decoder.add(layers.Reshape(target_shape=(7, 7, 16)))
6   self.decoder.add(UpsConvTAct(16, kernel_size=3, leaky_alpha=
      self.alpha))
7   self.decoder.add(UpsConvTAct(8, kernel_size=3, leaky_alpha=
      self.alpha))
8   self.decoder.add(layers.Conv2D(1, kernel_size=3, strides=1,
      padding='same'))
```

Python Code 3.5: An example implementation of a decoder

3.2.3 Training Architecture

The first decision in training the proposed network is to choose a suitable loss function for calculating the reconstruction error. Since the input and output data are 2D images, the direction of the error is not relevant, but it is important that diminishing values do not occur. As a result, a magnitude-based error is not suitable. Two appropriate options are the absolute error and the squared error. While the absolute error is a possibility, there is no particular

advantage to using it since the result does not need to be interpretable as it is primarily used for automatically adjusting the weights.
However, if large adjustments are made to unsuitable weights, this may lead to differences in the outcome for other bins of the reconstruction vector. Then changing all network weights to be similarly penalized might lead to bad progression. Therefore, a squared error-based loss function is preferred, as it primarily adjusts the biggest problems in the network in each iteration. It may require a few more training iterations in some cases, but it should increase the chances of reaching a (local) optimum at all. In addition, the squared error does not require the computation of the square root and is therefore more efficient.
Mean aggregation may further amplify strong outliers, while sum aggregation preserves the current attributes. It is difficult to determine how much stronger outliers should be penalized compared to small errors, and this likely depends on the specific data properties. Since TensorFlow provides implementations of MAE and MSE but not L2, MSE is chosen for the initial training. If the MSE does not deliver satisfactory results, a self-implemented L2 loss function may be used. In addition to automatic training, both MSE and MAE are used for evaluation purposes. In summary, the chosen loss function is the MSE because it primarily addresses the biggest problems in the network, is more efficient than the absolute error, and is implemented in TensorFlow in contrast to the L2 loss.

Regarding optimization, the choice of the algorithm can have a significant impact on the training performance and convergence speed. Adam is a popular choice as it delivers state-of-the-art performance and has been extensively evaluated. Recent studies have shown that other algorithms, such as AdaMax and Nadam, can achieve competitive results in some cases [82, 102]. Although they have not been evaluated that much, it would be reasonable to consider Nadam for the given case. Nesterov Momentum is known to perform better than basic Momentum; therefore, the empirical advantages found so far are strongly supported by a theoretic advantage of Nadam. As currently the Nadam and AdaMax TensorFlow implementations are experimental, this is open for future research, and for now Adam is used.

As for regularization, it is important to prevent overfitting and ensure that the network generalizes well to unseen data. The used datasets provide a large number of diverse training examples. While data augmentation is therefore not needed, additional regularization techniques might still be useful. A few of these techniques, like using the regularization term or ropout are tested briefly to see whether they are able to improve test results in the pretests. Normalizing the input data is used to ensure that the network is not biased towards specific features or values. As is common practice for images, the data is normalized to values in an interval ranging from one to zero. This additionally helps calculations during back-propagation.

3.2.4 Training Implementation

The implementation of the training process consists of several components. At the center of it all, a database-specific training script is used to manage preparing the data, performing the training, and validating the received results. The script first loads the desired dataset, then either loads a saved AE or builds a new one. It trains or continues to train and saves the trained models, and at the end, it quickly evaluates the current AE architecture's performance based on the reconstruction error or by visualizing the reconstruction. The script defines the

processing flow, by mostly calling functions from util scripts or the CAE manager class.

The load util script delivers functions to get the save and load paths, for loading the data, preprocessing it and splitting it into training and validation sets. Once the data is prepared, the CAE manager class is used as a high-level wrapper for loading, saving, and fitting the models. The manager class provides a user-friendly interface for performing common tasks such as training the model, saving the trained model, and loading a saved model. The final validation of the results validation util script together with a plotter util if graphical output is desired.

The loss function and optimizer, as discussed in detail in subsection 3.2.3 are only parameters to the compile() method of the Keras model API. The method is called at the end of the base class, as shown in the code example, and the values are set through a configuration file, so one could change them for all architectures on a single line.

Overall, the training implementation is designed to be modular and flexible, allowing for easy customization and adaptation to different datasets and use cases.

CAEManager

```
1  def __init__(self, base_path_template, load_epoch=
       DEFAULT_LOAD_EPOCH, autoencoder=None):
2
3  def load(self):
4
5  def save(self):
6
7  def fit_until_epoch(self, train_data, epoch, validation_data=
       None):
8
9  def train_and_save(self, train_data, epochs, val_data=None,
       interim_callback=False):
10
11 def validate(self, validation_data, keep_history=True,
       keep_params=True, batch_size=None):
12
13 def get_encoding(self, test_data):
14
15 def get_decoding(self, encoded):
16
17 def get_from_history(self, key):
18
19 def get_trained_epochs(self):
```

Python Code 3.6: Interface of the CAEManager Class

The CAEManager class is responsible for managing instances of the AE models and provides an interface of methods to interact with the model. The manager class extends the func-

tionality of the Keras API by keeping track of, saving, and restoring the history and params dictionaries of the model after each call to the fit() or evaluate() methods. The class can be initialized with a new AE instance or by specifying a save and load folder built from a base path template and load epoch, from which a saved model is loaded.

Most of the methods in the CAEManager class are extended wrapper methods of the Keras model methods. For example, the load() and save() methods extend the Keras inbuilt save and load methods by also saving the current history and params dictionaries of the models using the pickle package. The fit_until_epoch() and validate() methods are also just wrappers around the corresponding Keras methods, but they keep the total history and params dictionaries after calling fit() multiple times. This ensures that the dictionaries do not get overwritten and only consist of the values from the last fit call.

The train_and_save() method is a high-level wrapper of the fit_until_epoch() method. It takes an array of epochs as input and calls the fit wrapper, save method, and any specified callback for each epoch value. The getters in the CAEManager class are simply short-cuts to frequently used values or methods inside the managed AE instance.

Overall, the CAEManager class provides a user-friendly interface for managing and training AE models while keeping track of important meta-data.

Load Util

The code for loading the data, forming the input pipeline, and retrieving the model save and load folder is contained in a load script within the util module. To retrieve the model save and load folder, the get_base_path_templates() method returns either a template path open to include the epochs folder or the exact folder with the epoch included. The method involves a simple chaining of folders for the AE names, versions, etcetera that is not shown here.

For the first dataset used, Fashion-MNIST, the input pipeline is straightforward, as it is included in TensorFlow under the keras datasets package. The dataset comes in a supervised representation given by numpy arrays. The pipeline loads the data, converts the gray-scale images into the scale $[0, 1]$, and returns the result together with the image shape.

```python
1 def load_prepared_fashion_mnist():
2   (x_train, y_train), (x_test, y_test) = fashion_mnist.load_data
      ()
3   x_train = x_train.astype('float32') / 255.
4   x_test = x_test.astype('float32') / 255.
5   return (x_train, y_train), (x_test, y_test),
      FASHION_MNIST_SHAPE
```

Python Code 3.7: Loading and preprocessing of Fashion-MNIST

The second dataset used, dsprites, is loaded from the external TensorFlow Datasets package and comes in an unsupervised format as a TensorFlow Dataset object. The images are constructed from a latent representation, whose values are given with each image. The input pipeline extracts only the images for all splits, as the other values are not needed for training an AEs. For the train split, it duplicates the images to a tuple of (image, image) as required

for the MSE measure. The not shown function as_tuple applies a given function and returns a tuple of the result. The images are converted to grayscale in the extract_image function by using tf.cast(). The resulting dataset is then prepared for subsequent training using predefined calls to the TensorFlow functional Dataset API. The values used for shuffle and batch_size are chosen accordingly to work with the used Graphics Processing Unit (GPU) for testing. The provided code is prepared for the thesis to best show the pipline preprocessing done; in the actual code, the calls are divided into multiple functions, and the validation set is prepared in parallel.

```
1 # Prepare data for efficient training
2 ds_train, ds_val = tfds.load('dsprites', split=split,
    shuffle_files=True, as_supervised=False)
3 ds_train = ds_train.map(partial(as_tuple, extract_image),
    num_parallel_calls=tf.data.AUTOTUNE)
4 ds_train = ds_train.shuffle(200000)
5 ds_train = ds_train.cache()
6 ds_train = ds_train.batch(batch_size=500)
7 ds_train = ds_train.prefetch(tf.data.AUTOTUNE)
```

Python Code 3.8: dSprites input pipeline for the train set

3.2.5 2D Retrieval Implementation

```
1 def calculate_nearest(encodings, subject_index,
    calculate_indices=True):
2   subject_encoding = encodings[subject_index]
3   retrieval_list = []
4   for i in range(len(encodings)):
5     other_encoding = encodings[i]
6     if calculate_indices:
7       retrieval_list.append((i, sigma_manhattan(subject_encoding
          , other_encoding)))
8     else:
9       retrieval_list.append((other_encoding, sigma_manhattan(
          subject_encoding, other_encoding)))
10  retrieval_list.sort(key=lambda a: a[1])
11  return retrieval_list
```

Python Code 3.9: The calculation of the similarity list

At last, retrieval is implemented as a script inside the util package. The main interface in the script is save_retrieval, which takes as input the image encodings, subject index, data to plot, save path, data type, and subsample size. The function uses a while loop to gradually decrease the subsample size, and saves the resulting plot to the specified save path. By saving a multiple of subsamples of decreasing size, differences in the quality of retrieval can be easier detected. For the saving and retrieval of the plotter util script provides an appropriate

function.

The interesting kernel implementation of finding the retrieval order itself is inside the calculate_nearest() function. The function calculates the distance between the subject encoding and all other encodings in the list using the proposed distance function. The distance function itself is implemented using Scipy's distance.cityblock() and expit() functions. By storing the results in a tuple list, they can be sorted by the distances, while later the indices or other encoding can be retrieved.

3.3 2D Layer Configuration Tests

In order to establish a starting point for designing the subsequent 3D architecture, implementations and evaluations of specific network configurations were made to test the quality of the previously made theoretic decisions and to retrieve good settings for open choices. These 2D pretests are conducted on multiple architectures with a variety of different configurations. As already mentioned, the two widely used datasets are: a MNIST-like fashion product database (Fashion-MNIST) and a dataset of 2D shapes known as dSprites. The architecture evaluation results and their implications for further architecture design are presented, and the retrieval performance of the best architectures is also briefly discussed.

The Fashion-MNIST dataset is used to rapidly try different architectures and layer configurations and evaluate their ability to efficiently encode and reconstruct the data. The best-performing solutions are then subjected to a brief performance test using the proposed similarity measure for retrieval.
The dSprites dataset afterwards is used to further test the capability and transferability of the found architecture concepts. The encoding and decoding of these simple, rigidly translated shapes act as a simplified version of the later 3D data encoding. The test in particular further develops the convolutional parts of the architectures.

3.3.1 Fashion-MNIST

The first dataset used in this thesis is the Fashion-MNIST dataset, introduced by Xiao *et al.* [103] in 2017. This dataset consists of 70.000 gray-scale 2D images, each with a shape of 28×28 pixels. The images depict fashion products from 10 different categories, and the dataset is intended to serve as a more challenging alternative to the basic MNIST dataset. As a result, it is directly included in TensorFlow's tensorflow.keras.datasets package. The dataset was chosen due to its simplicity and direct integration with TensorFlow, allowing for rapid testing of initial concepts. Examples of the input data can be seen in Figure 3.5.

Dense Layer Configuration

To start with, a simple architecture is proposed. It consists of two convolutional layers with 8 and 16 filters, respectively, each with a kernel size of 3. Basic ReLU activation is used, and no regularization techniques are applied. Two Dense layers are included in the middle of the architecture, and different configurations of neuron sizes are tested to determine the most

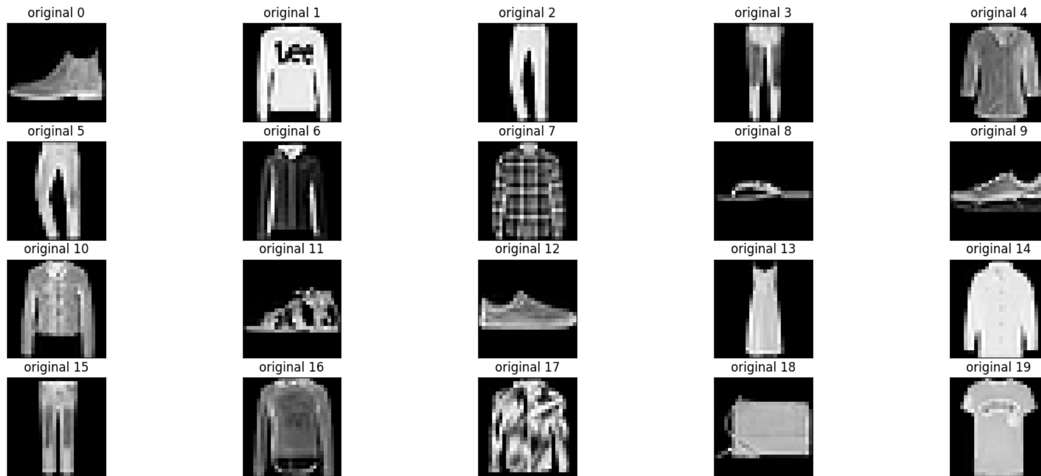

Figure 3.5: The graphic shows 20 example images from the Fashion-MNIST dataset. The examples are chosen randomly from the test dataset.

suitable one. The results of the tests are presented in Table 3.1, which shows the different configurations of Dense layers and their corresponding MAE and MSE values for the train and validation sets after 30 trained epochs. The configuration with the best reconstruction error is highlighted in green. An important thing to notice for reproducibility is that these tests were performed at an early stage and had sigmoid activation at the latent layer.

Dense Layers	Train MSE	Train MAE	Validation MSE	Validation MAE
256, 32	0.0076	0.0438	0.0079	0.0452
256, 64	0.0065	0.0409	0.0066	0.0412
512, 64	0.0062	0.0395	0.0064	0.0400
400, 64	0.0064	0.0403	0.0070	0.0430

Table 3.1: The table shows the results of different neuron size configurations of the Dense layers in fm_1. The best results where obtained by a combination of 512 and 64 neurons as marked with green.

The best found configuration colored in green is denoted as fm_1. For later retrieval testing, also the first worst row is denoted as fm_0.

Regularization

At the next step, different regularization techniques were tested, including L1 and L2 kernel regularization with a standard regularization value of 0.01 and dropout. The L1 regularization layers were placed between each layer at the encoder and after the dense layers only. L2 regularization and dropout were only tested after the dense layers of the encoder. As Dropout spans a sub-network, the training epochs were increased to 100 in the affected tests. The tests were also performed with sigmoid activation at the latent layer and not rerun due to the extremely bad results.

Regularization	Train MSE	Train MAE	Val MSE	Val MAE
L1 Encoder	0.6251	0.2321	0.6270	0.2313
L1 Dense	0.6169	0.2320	0.6179	0.2283
L2 Dense	0.0215	0.0757	0.0220	0.0781
Dropout (0.2)	0.0056	0.0369	0.0054	0.0353
Dropout (0.3)	0.0061	0.0386	0.0056	0.0357
Dropout (0.5)	0.0071	0.0421	0.0061	0.0381

Table 3.2: The table shows the results of different regularization techniques evaluated on the current fm_1 model. All regularizations and positions of regularization performed extremely poorly.

As Table 3.2 clearly shows, none of the tested regularization techniques improved the results at all. Regularization was therefore chosen not to be further used or researched in the scope of this thesis.

Activation Function

As part of the last tests of configurations for fm_1, various activation functions were evaluated. The first leaky ReLU was tested as an alternative activation function with different alphas on the previous found best configuration, as due to its linear function at the negative side, it is the least computation intensive activation of the more complex ReLU versions.
The results after again 30 epochs of training are shown in Table 3.3. The tests were also performed with a sigmoid activation in the middle, to gain comparable values to ReLU, as removing that sigmoid activation improves the results already on its own.

Alpha	Train MSE	Train MAE	Validation MSE	Validation MAE
0.3	0.0051	0.0371	0.0054	0.0380
0.2	0.0052	0.0374	0.0055	0.0383
0.4	0.0052	0.0382	0.0055	0.0398
0.35	0.0051	0.0376	0.0053	0.0383

Table 3.3: The table shows how ReLU's alpha parameter affects the outcome of the fm_1 model. The best results were obtained with a value of 0.35, but the differences are so tight that these could be random numeric variances. The value is still used from now on, as it definitely does not perform worse.

It seems that leaky ReLU significantly increases the results at all tested alpha values, with a maximum of improvement at around 0.35. The model was thus retrained without the sigmoid activation at the latent layer and the best-found alpha value. The updated results and configuration are used as a base for further activation function comparisons. The values are shown together with the values for ELU and PReLU in Table 3.4. Additionally, the training time per epoch is given at this time to also take computation times into account.

Activation	Time	Train MSE	Train MAE	Validation MSE	Validation MAE
leaky ReLU	12s/epoch	0.0048	0.0362	0.0051	0.0368
PReLU	17s/epoch	0.0046	0.0343	0.0049	0.0354
ELU	12s/epoch	0.0050	0.0359	0.0052	0.0359
SELU	12s/epoch	0.0056	0.0406	0.0057	0.0410

Table 3.4: The table shows the effect of more sophisticated RelU variants and their computation time per epoch. While PReLU was the most accurate, it came with a significant increase in computation time.

The two exponential-based ReLU versions could not improve the reconstruction results any further. It is, however, worth mentioning that the difference for ELU is marginal; it actually delivered better results than leaky ReLU did for any other alpha values tested. Concerning computation times, no practice difference per training epoch could be measured for the used data.
PReLU on the other hand, was actually able to further improve the model a bit. The improvement, however, comes along with a significant 41.67 % longer computation timer per training epoch. leaky ReLU with an alpha value of 0.35 is thus kept as the activation function for now, but PReLU might be considered again later if further improvements are necessary.

Figure 3.6 shows a few example inputs, their latent encoding, and reconstruction received from using the fm_1 model as presented, after further training to 100 epochs.

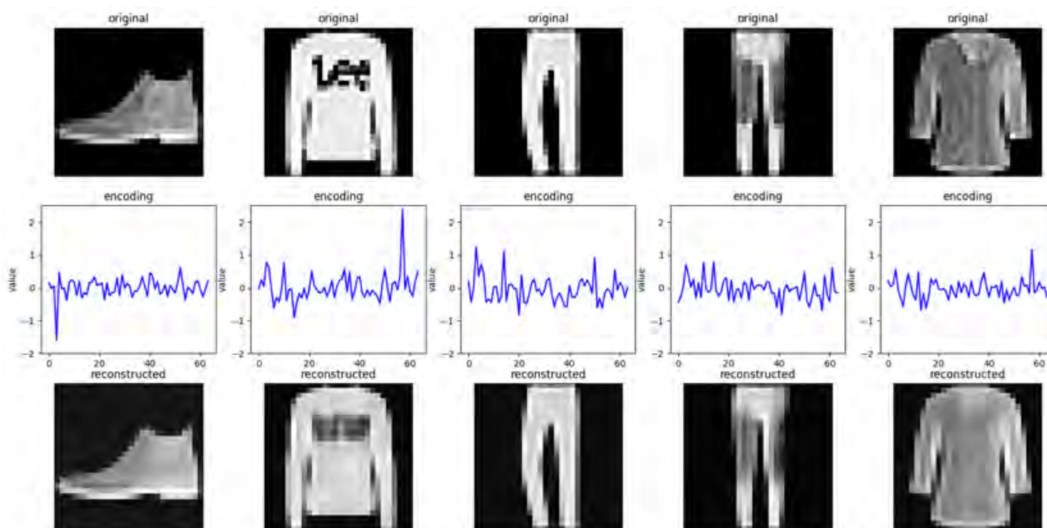

Figure 3.6: The graphic shows the processing of five example inputs by fm_1. In the first row, the original input data is displayed. The second row shows a plot of the received 64-value latent encoding. The reconstruction built from the encoding with the decoder is shown in row three.

FM_2

After testing the different effects of Dense layer configuration, activation, and regularization on the first architecture, another configuration of convolutional layers is tested. The number of convolutional layers, the kernel, pooling sizes, and paddings of them stay the same. The number of filters is increased to 16 and 32, respectively, for a theoretically more powerful model. As more feature maps result in more data at the hidden layers, suitable sizes for the neuron parameter of the Dense layers are evaluated again. This should give a more general relative compression size to the input size. Furthermore, an architecture with three Dense layers is proposed. As more input data is put into the first Dense layer, stronger compression is required, and thus more Dense layers might be appropriate. The architecture is denoted as fm_2. The results are shown in Table 3.5.

Dense Layers	Train MSE	Train MAE	Val MSE	Val MAE
512, 64	0.0046	0.0354	0.0049	0.0358
800, 64	0.0041	0.0339	0.0047	0.0354
1028, 64	0.0041	0.0340	0.0047	0.0358
1200, 64	0.0041	0.0342	0.0047	0.0351
1200, 600, 64	0.0046	0.0359	0.0051	0.0372

Table 3.5: The table again shows various neuron sizes at the Dense layers, but now for the new architecture, fm_2. The ideal configuration varies due to different input sizes from previous and is now found at 800 and 64 neurons, respectively.

The results obtained from the new architecture confirm the intuition that increasing the number of feature maps can improve the model's performance. The best configurations were found to have a neuron size in the range of 800 to 1200 at the first Dense layer. The smallest configuration (row one) and the three-layer configuration (last row) yielded slightly worse results than the other configurations. The three-layer configuration may have suffered from slower training, but the potential improvements did not seem to be worth the additional training time. It is worth noting that the training time per epoch has already increased hugely overall, even for row two configuration, by 208% to 37 s/epoch.

Rows two to four of the table showed extremely similar results, making it difficult to determine which configuration is slightly better. Depending on whether MSE or MAE and train or validation values are considered, the best configuration differs. Therefore, row two with a first neuron size of 800 was chosen to be denoted as fm_2 due to its lower computational complexity. In general, a maximum compression of about 50% in the first layer seems appropriate.

After further training to 100 epochs, the values presented in Table 3.6 were obtained. Figure 3.7 shows the retrieved latent encoding and reconstruction from fm_2 using the same inputs as for the fm_1 visualization. Overall, fm_2 delivered slightly better results than fm_1.

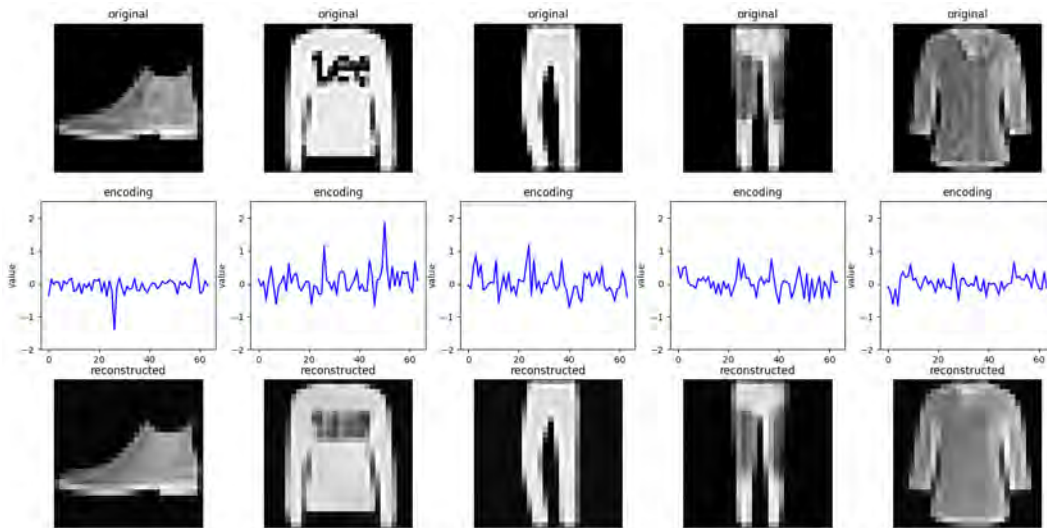

Figure 3.7: The graphic shows the processing of five example inputs by fm_2. In the first row, the original input data is displayed. The second row shows a plot of the received 64-value latent encoding. The reconstruction built from the encoding with the decoder is shown in row three.

Comparison of FM_1 and FM_2

Architectures	Time	Train MSE	Train MAE	Val MSE	Val MAE
fm_1	12s/epoch	0.0043	0.0313	0.0048	0.0329
fm_2	37s/epoch	0.0035	0.0318	0.0034	0.0307

Table 3.6: The table shows the final values of reconstruction accuracy and computation time obtained for fm_1 and fm_2. The models were both trained for 100 epochs at the time of comparison.

The found models fm_1 and fm_2 are now briefly tested and compared with respect to their results provided together with the proposed similarity measure. This comparison is intended to briefly confirm two intuitive hypotheses:
First, the proposed architecture and similarity measure are generally suitable to deliver a retrieval algorithm. This implies that shape information is indeed preserved during encoding and that rigid or non-rigid transformations are handled well by the AEs.
Second, the reconstruction error used for training approximately also implicates good encoding results for retrieval. For this hypothesis, the model fm_0 is also taken into comparison.

For retrieval, the test split of the Fashion-MNIST dataset containing 10,000 test images is used. Example results for two calls with different queries on architectures fm_1 and fm_2 are shown in Figure 3.8 and Figure 3.9.

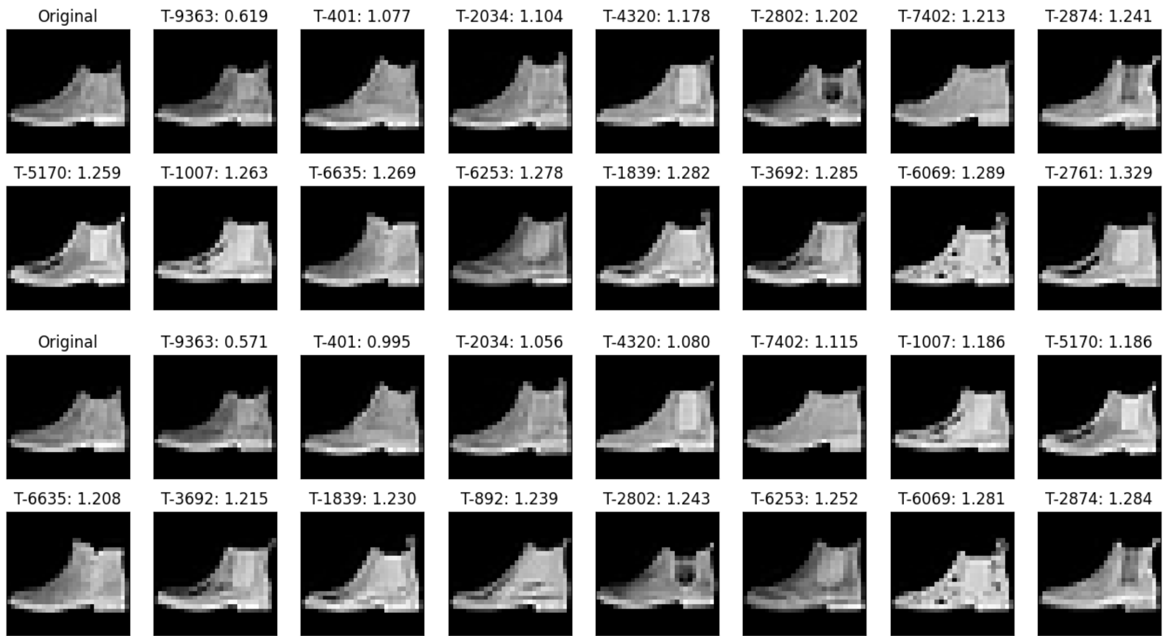

Figure 3.8: The graphic shows the retrieval results of a call with a shoe image as a query on fm_1 (row one and two) and fm_2 respectively (row three and four). The input query image is shown in the left upper corner. Then the 15 most similar images are shown in descending order, together with the calculated value of the proposed distance function.

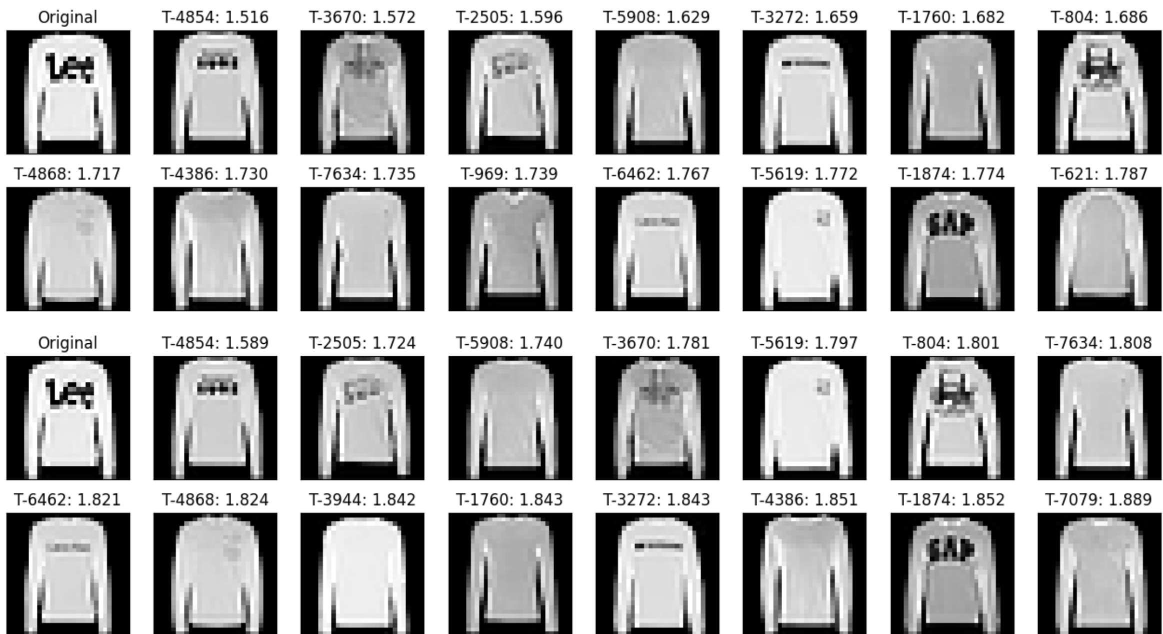

Figure 3.9: The graphic shows the retrieval results of a call with a pullover image as a query on fm_1 (row one and two) and fm_2 respectively (row three and four). The input query image is shown in the left upper corner. Then the 15 most similar images are shown in descending order, together with the calculated value of the proposed distance function.

The retrieved images in both fm_1 and fm_2 show that the proposed architecture and similarity measure are indeed suitable for delivering a retrieval algorithm. In both cases, similar items are grouped together in the retrieved images, demonstrating that shape information is preserved during encoding and that the AEs can handle rigid or non-rigid transformations well.

In order to better visualize the differences between the retrieval results of the three architectures, for the second hypothesis, multiple subsets are built. Each subset only consists of the first x test images. This reduces the amount of very similar shapes, so correct ordering can be easily checked by human vision. Figure 3.10 shows the results for the three architectures fm_0, fm_1 and fm_2 on a quite extreme subset of 50 images only.

Figure 3.10: The graphic shows the retrieval results of the previous shoe image query on a 50-image subset. The first two rows are the results of fm_0, row three and four are the results of fm_1 and row five and six are the results of fm_2.

The results overall confirm hypothesis 2. However, the results of fm_1 are slightly better than the results of fm_2 with a lower reconstruction error. Approximately the same implication holds as suggested. However, a too powerful model further reduces the reconstruction error

but reduces the abstraction generality of the encoding. Thus, the retrieval results obtained get worse if the improvement comes solely from more feature maps, but the correlation holds for different configurations with the same settings for the convolution layers. Although the results are still of very high quality, the problem should not be that important on a bigger dataset or when using data augmentation to force the learning of a more general encoding, unless the model is chosen to be too powerful.

3.3.2 dSprites

Before transferring the previously found architecture concepts to 3D data, a few more tests are made with a more comparable 2D dataset. "dSprites is a dataset of 2D shapes procedurally generated from six ground truth independent latent factors. These factors are color, shape, scale, rotation, and the x and y positions of a sprite" [104]. A few random example images from the dataset are shown in Figure 3.11.

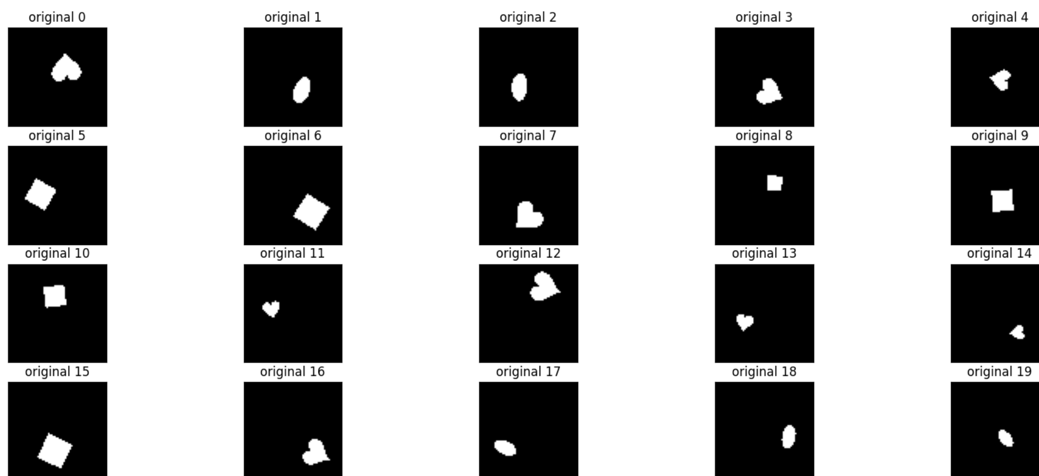

Figure 3.11: The graphic shows 20 randomly chosen example images from the dSprites dataset.

The dataset was chosen for several reasons, including its use as inspiration for the 3D datasets used in later tests. This addresses both the procedural generation from a latent vector as well as the use of augmented basic shapes. As the retrieval algorithm is designed to be used in the context of industrial CT scans, basic shapes are a good first approximation.

Aside from that, the dataset consists of binary images with hard edges, and the shapes themselves are edgeless and filled inside with a single value. This makes it very difficult to automatically encode images concerning feature detection and rigid transformation invariance. The dataset acts as an early edge case test for CAE retrieval. It should be very well suited for testing one more configuration parameter, which could be crucial in later 3D volumes: The kernel size of the convolutional layers.

The kernel size is an important hyper-parameter in CNNs, as it determines the size of the local receptive field that the network uses to detect features in the input data [105]. In the case of images with hard edges and no surface features, a smaller kernel size may be more

appropriate to capture the sharp edges of the shapes. This is particularly relevant in 3D volumes from CT scans, where objects are often filled with a single material, resulting in a lack of surface features. A larger kernel size may instead smooth out the edges and capture more high-level features. This leads to most modern networks having larger sizes in the lower levels [105], which will also be adapted in the following models.

By testing different kernel sizes on the dSprites dataset, important insights into the optimal configuration for later experiments on 3D volumes are gained. Evaluating kernel sizes by testing or visualization is still a common approach, although more sophisticated ways like automated retrieval of ideal sizes have been presented in recent years [105, 106].

As the kernel size affects the generalization of the feature maps gained, more transformation-robust features might have worse reconstructions than more specific features. Using the reconstruction error like in the previous section is not applicable for finding the best results. As the dSprites dataset is not split into a train and test set, exact information on relevant results of a chosen subset and query could only be retrieved by counting through all test examples. The only applicable performance measure, as it is, is precision. The size of the retrieved results (A) is set to the 15 best matches. As precision alone would, however, not take into account the ordering of these 15 items, a modified average precision is also applied in case precision returns the same values. The set of relevant items R is simply set to the set of relevant items A in the response. As overall precision is the same as the precondition and thus the amount of correctly retrieved shapes in A, the modified average precision gives a higher score for better sorted responses.

As a beginning, a configuration similar to fm_1 with 8 and 16 feature maps, respectively, is trained. As the input images now have the shape 64×64, adjusting the dense layer sizes is again required. Based on the previous configuration, an estimated size of 2400 should work well. For training, a subset of the first 200,000 images is used, and for evaluation, the 1,000 following images are used. The model has been trained for 20 epochs on this training subset. Using kernel size three for all layers is used as a configuration, and the results are used as a base comparison point. For the other configuration, the following considerations were made:

The first layer should detect edges. As the images include no noise and were generated with perfect edges, a kernel size of three should be sufficient for detection. For the later disturbed 3D data, the size might be increased to four or five.
The second layer should then directly retrieve the global features from the pooled 32×32 feature maps. The retrieved features are already more global by using a same-size kernel due to the reduced spatial dimension of the data after pooling. Assuming the original objects are at most a third of the size of the original input image, a maximum filter size of eleven is needed to capture the whole object. As a lower bound, the next bigger filter size of four is taken. Although it is mostly common to use uneven filter sizes, a valid padded size might be beneficial when retrieving the final features in the last convolution layer. The sizes tested are therefore between four and eleven.

An additional convolutional layer for first detecting medium-level features like corners could be required. This further reduces the spatial dimension of the feature maps and thus helps computation efficiency with bigger data, which could be beneficial if not even necessary in the upcoming 3D models. A short test of configurations with a third convolutional layer of 32 filters is performed, too. The models have a medium kernel size of four to five in the second

layer and a filter kernel size of four to eight in the third layer. As the input to the third layer is reduced down to 16×16, filter kernel size eight already captures half of the whole image and should be more than sufficient. The input to the first Dense layer is half as big, so the neuron size was reduced to 1200.

The resulting precision and modified average precision values for all tested configurations are presented in Table 3.7.

Kernels	Heart 1000	Heart 100	Ellipse 250
3, 3	1/3, 0.61	1/3, 0.29	4/15, 0.25
3, 4	1/3, 0.55	2/5, 0.34	1/5, 0.24
3, 5	1/3, 0.64	2/5, 0.42	1/5, 0.23
3, 7	1/3, 0.55	4/15, 0.37	2/15, 0.33
3, 8	1/3, 0.62	2/5, 0.39	2/15, 0.25
3, 9	1/3, 0.64	2/5, 0.36	4/15, 0.25
3, 11	1/3, 0.64	2/5, 0.35	2/15, 0.26
3, 4, 4	2/5, 0.61	2/5, 0.35	1/5, 0.30
3, 5, 8	2/5, 0.63	2/5, 0.33	1/5, 0.47

Table 3.7: The table shows the precision and modified average precision of the different convolution kernel configurations tried on the dSprites dataset. The left side gives the kernel sizes of the convolutional layers. Inside the table, the first number is the obtained precision score, and the second is the modified average precision.

The results are not the best overall, but in general, they got better by increasing the second layer kernel as well as by adding a third kernel. The precision and modified average precision values given, however, only slightly represent this finding. Unfortunately, the model still has a lot of problems dealing with similar shapes and strong rotation or scaling. When retrieving for a heart, a lot of ellipses with similar edges are obtained, and vice versa. Visualized examples for the heart 1000 query are presented inFigure 3.12. Furthermore, Figure 3.13 shows a not-in-the-table-evaluated example of an ellipse 1000 query. The query actually returns more hearts than the heart query itself, giving a strong hint for yet-to-be-solved problems with shapes with similar substructures.

Overall, the dSprites tests clearly showed some yet-to-be-solved shortcomings of the proposed models. Especially when dealing with rotation and scaling, which seemed to perform badly until now if no transformation-invariant surface features could be detected, a better-suited setting of the convolutional kernel sizes only slightly improved the ability to handle transformations.

Most real-world objects, even from industrial CT scans, will have more surface features than these perfectly generated one value shapes. However, a lot of smaller objects, like nails or other small construction parts, do indeed only consist of a single material. More improvement possibilities for this are discussed in the upcoming section of the 3D model, as the generator-based dataset is more suitable to provide the desired manipulations to the data for further improvements.

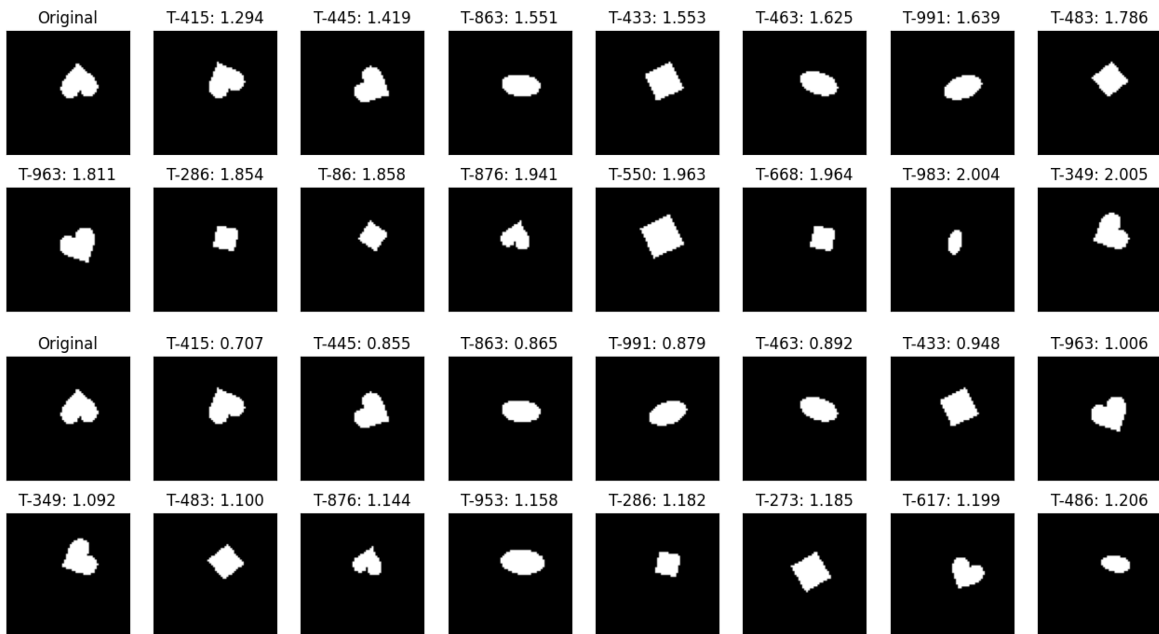

Figure 3.12: The graphic shows the retrieval results for the first heart query on a subset of 1000 samples. The first three rows correspond to the first 3, 3, and 8 configurations, and rows three and four correspond to the last 3, 5, and 8 configurations. Although only one correct heart more is returned in these first 15 responses, these hearts, as well as the shape similar ellipses, are sorted better..

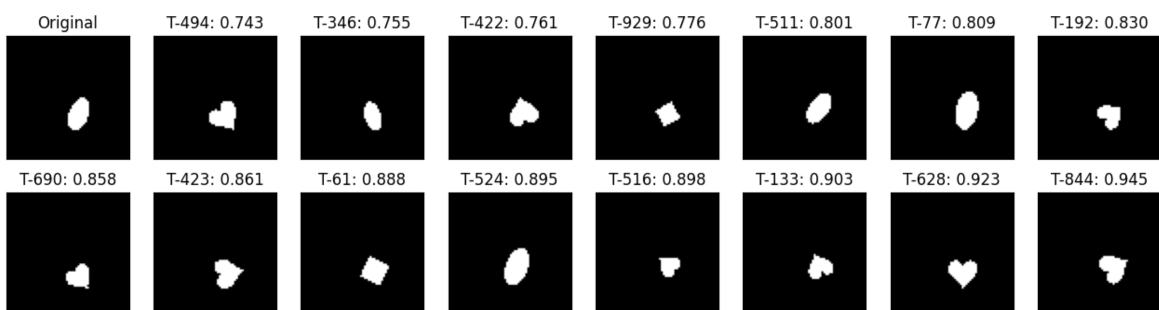

Figure 3.13: The graphic shows the retrieval results for the ellipsoid query, but on a subset of 1000 samples on the last model with configurations 3, 5, and 8. The query returns more hearts than the previous heart query, though it still returns a comparable amount of ellipses.

3.4 3D Model Design

As all the basic questions addressing configuration and design are answered, this section presents the extension of the previous findings to 3D data. In the 3D-CAE architectures, the input data is a volumetric image. The previous convolutional layers are replaced by 3D convolutional layers, as well as the 2D pooling layers with 3D ones, which take into account the spatial information in the x, y, and z dimensions. Aside from that, the architecture stays mostly the same. Overall, two architectures are proposed based on the 2D findings and restricted by computational feasibility.

Regarding the implementation, a lot of previous code can be reused, like the CAEManager class or the retrieval index calculation on the encodings. All together, three big blocks have to be changed or recoded: First, new self-defined layers have to be implemented using the corresponding 3D layers of Keras to gain the same comfort for defining the models as before. Second, a completely new input pipeline is needed for the self-generated 3D datasets. Third, new train and evaluate scripts are needed, as well as a new 3D plotter for visualization and an implementation of the MAP calculation.

3.4.1 Network Implementation

The new building blocks, or self-defined layers as called before, for the 3D models are implemented just the same as the previous 2D ones as sublayers. Instead of Conv2D or AveragePooling2D the included Conv3D, Conv3DTranspose or AveragePooling3D layers are used. The DenseActivation layer can even be reused directly as is, as the data at that point is already flattened.

Based on these 3D chained layers, two architectures are proposed. Only the add code for the encoder and decoder is given here for a more clean presentation. Inside the project itself, the models are implemented the same way as the 2D models, as a subclass of the AEBase class.

```
encoder.add(ConvActPool3D(4, kernel_size=3))
encoder.add(ConvActPool3D(8, kernel_size=5))
encoder.add(ConvActPool3D(16, kernel_size=8))
encoder.add(layers.Flatten())
encoder.add(DenseActivation(5000))
encoder.add(layers.Dense(128))

decoder.add(layers.Input(shape=128))
decoder.add(DenseActivation(5000))
decoder.add(DenseActivation(8192))
decoder.add(layers.Reshape(target_shape=(8, 8, 8, 16)))
decoder.add(UpsConvTAct3D(16, kernel_size=8))
decoder.add(UpsConvTAct3D(8, kernel_size=5))
decoder.add(UpsConvTAct3D(4, kernel_size=3))
decoder.add(layers.Conv3DTranspose(1, kernel_size=3))
```

Python Code 3.10: Layers and configuration of architecture 1

```
encoder.add(ConvActPool3D(8, kernel_size=3))
encoder.add(ConvActPool3D(16, kernel_size=3))
encoder.add(ConvActPool3D(32, kernel_size=3))
encoder.add(ConvActPool3D(64, kernel_size=4))
encoder.add(layers.Flatten())
encoder.add(DenseActivation(2500))
encoder.add(layers.Dense(128))

decoder.add(layers.Input(shape=128))
decoder.add(DenseActivation(2500))
decoder.add(DenseActivation(4096))
decoder.add(layers.Reshape(target_shape=(4, 4, 4, 64)))
decoder.add(UpsConvTAct3D(64, kernel_size=4))
decoder.add(UpsConvTAct3D(32, kernel_size=3))
decoder.add(UpsConvTAct3D(16, kernel_size=3))
decoder.add(UpsConvTAct3D(8, kernel_size=3))
decoder.add(layers.Conv3DTranspose(1, kernel_size=3))
```

Python Code 3.11: Layers and configuration of architecture 2

3.4.2 Input Pipeline

The datasets used in the 3D cases are generated from a latent generator and stored in the TFRecord format, compressed with GNU zip (GZIP). Thus, the input pipeline has to first read and parse a GZIP compressed TFRecord, then build a dataset from it, and on the fly apply transformations if desired.

For the first task, a tf_records_util module is used. It proposes a function read_tfrecords_gzip, which takes a file path and a train bool to either read a train dataset or a test dataset with the additional relevant information needed for MAP calculation. The function code is omitted here, as it is basically the same as in the TensorFlow documentation for how to read and parse a TFRecord.

The dataset building pipeline itself is this time encapsulated inside a Shapes3DLoader class, which is useful for retrieving different combinations of applied augmentation and batching to the data by using different values when initializing the Shapes3DLoader object for batch_size, max_shift and noise parameters. Although named after the first dataset used later, it is generalized and reused for the Multi-Shapes3D dataset, too.
The actual interface to read and build the dataset is the read_n_prepare_dataset method, whose code is shown in the following.

The method first reads the given file by using the tf_records_util module and then applies basic shuffle and cache operations to the data. It then maps the random_augmentation method on the dataset, which performs the desired augmentation and is discussed in more detail afterwards. The next call adds the channel dimension and makes a double tuple, hence adding the target data for loss calculation. At last, the dataset is batched, pre-fetched, and returned.

```
def read_n_prepare_dataset(self, records_file):
  dataset = read_tfrecords_gzip(records_file)
  dataset = dataset.cache().shuffle(75)
  dataset = dataset.map(self.random_augmentation,
     num_parallel_calls=tf.data.AUTOTUNE)
  dataset = dataset.map(self.expand_channels_add_target,
     num_parallel_calls=tf.data.AUTOTUNE)
  dataset = dataset.batch(self.batch_size)
  dataset = dataset.prefetch(tf.data.AUTOTUNE)
  return dataset
```

Python Code 3.12: Definition of the read_n_prepare_dataset method

The mentioned random_augmentation method itself consists of a simple chain of if statements based on the initialization values for the different augmentations. For each true path, a random set is chosen, and the random augmentation is applied to the data by calling functions from a transformation util module. The transformation module proposes a function for translation as well as a function for speckle, gauss, and salt and pepper noise. It also has a wrapper function for applying the desired noise. The code for all four augmentations implemented is given in the following code snippets.

```
padding = ((abs(x_shift), abs(x_shift)), (abs(y_shift), abs(
   y_shift)), (abs(z_shift), abs(z_shift)))
padded_tensor = tf.pad(tensor, padding, mode='constant')

# Determine the slice indices for the translated tensor
return padded_tensor[x_start:x_end, y_start:y_end, z_start:z_end
   ]
```

Python Code 3.13: The implementation of the translation augmentation. Only the most significant code lines are given.

For translation, the volume is expanded by padding in all directions needed, and then a volume of original size is clipped out of the padded volume, such that the resulting volume corresponds to a volume with the desired translation applied.

```
noise = tf.random.normal(shape=tf.shape(tensor), mean=mean,
   stddev=stddev, dtype=tensor.dtype)
noise_tensor = tf.add(tensor, noise)
return tf.clip_by_value(noise_tensor, 0.0, 1.0)
```

Python Code 3.14: The relevant code lines of the implementation of gauss noise augmentation

Gauss noise is applied as usual by adding a normal distributed volume of the same size to the input tensor. The mean is fixed at 0.0 and the derivation at 0.03 for augmentation usage.

```python
noise = tf.random.uniform(shape=tf.shape(tensor), minval=0,
    maxval=1, dtype=tensor.dtype)
salt_mask = tf.cast(noise < prob, tf.float32)
pepper_mask = tf.cast(noise > (1 - prob), tf.float32)
#Some further preparations of the masks
noise_tensor = tf.multiply(tensor, salt_mask_transpose) +
    pepper_mask
return tf.clip_by_value(noise_tensor, 0.0, 1.0)
```

Python Code 3.15: The relevant code lines of the implementation of salt&pepper noise augmentation

Salt and Pepper noise is implemented on the basis of a single, uniform distribution for efficiency reasons. The not-shown further preparation of the masks deals with points that are in both masks. The probability is fixed set to 0.02 for augmentation.

```python
noise = tf.random.normal(shape=tf.shape(tensor), mean=0.0,
    stddev=std_dev, dtype=tf.float32)
noisy_tensor = tensor + tensor * noise
return tf.clip_by_value(noisy_tensor, clip_value_min=0.0,
    clip_value_max=1.0)
```

Python Code 3.16: The significant lines of implementation of the speckle noise augmentation

For the speckle noise, the standard derivation is again fixed at 0.03 in the case of augmentation.

For the later, also used rotation transformation, a different train dataset with rotations already applied is used. Rotations would require a rearranging of the data from voxel format to point cloud format, such that a rotation matrix can be applied. This would be computationally costly to do on the fly in the input pipeline, and as the latent generators are capable of generating rotated objects, this approach was chosen.

3.4.3 Training, Evaluation and Visualization

Training, evaluation, and visualization are, like before, performed in separate scripts. For training, these are quite simple, as the loading and preparing are handled by the previously presented Shapes3DLoader class. Like before, training itself is encapsulated in the CAEManager class; thus, the script only instantiates and then calls the provided function in the right order and with the desired arguments to train the desired model by the desired train method.

The evaluation script, however, comes in with more new code needed. Most of the evaluation happens in the evaluate_query function, which evaluates a single test case query completely, returning the average precision received. First, another wrapper method, get_encodings encapsulates the calculation of the latent encoding for a whole input test set, returning the input volumes, the encoded latent representations, and the corresponding relevant information. This method is called inside evaluate_query and the information used for sorting the relevant information according to the latent distances.

For the calculation of the average precision itself, a new util script for calculating the average precision of a single query is added. The calculation is performed with a basic for loop on a relevant boolean array, which uses the sorted relevant information.

```
1 index = 1
2 relevant_so_far = 0
3 score = 0.0
4 for entry in ds_val_sorted[1:]:
5   if entry:
6     relevant_so_far += 1
7     score += relevant_so_far / index
8   index += 1
9 return score / relevant_so_far
```

Python Code 3.17: The calculation of the average precision in the util module

For a more convenient evaluation, the whole calculation is wrapped by an evaluate_group function, which gets every test case TFRecord from a given folder, calls the single evaluation, and at the end calculates the Group Mean Average Precision (GMAP) received. For even further convenience, another function calls the evaluate_group function subsequently for each group, calculating the overall MAP. For simplicity, print statements to obtain the values were removed from the code here.

```
1 files = get_files_from_folder(folder)
2 scores = []
3 counter = counter_start
4 for file in files:
5   ds_val = tf_records_util.read_tfrecords_gzip(file, enc_type='
      test')
6   scores.append(evaluate_query(cae_m, ds_val))
7   counter += 1
8 gmap = numpy.array(scores).sum() / scores.__len__()
9 return gmap
```

Python Code 3.18: The convenience function evaluate_group

Concerning visualization, the vedo package is now used, and a new plotter3d util script is provided. The vedo package delivers an append method for volumes, which makes it possible to show multiple volumes in sorted order in a single plot. The data used later is again grayscale, but visualized in color by Vedo's standard color mapping. For the visualization of the detailed look into the latent encodings, the z2D plotter is expanded to provide a plotting option that uses a previously made snapshot of a 3D volume as the first image.

3.4.4 Haar Wavelet Integration

To integrate the Haar-Wavelet filters, another sublayer ConvolveFilters is implemented. For reusability, the layer is implemented to perform a fixed convolution, but for any given filter. It furthermore proposes the option to load the filter from a numpy file. The filtering is performed by a convolutions handled by TensorFlows' conv3d function.

```
1 class ConvolveFilters(layers.Layer):
2   def __init__(self, filters, is_file=True):
3     super().__init__()
4         self.filters = self.load_prepare_filters(filters) if
            is_file else filters
5
6   def call(self, input_tensor):
7     filtered = tf.nn.conv3d(input_tensor, self.filters, strides
        =(1, 1, 1, 1, 1), padding='SAME')
8     return filtered
```

Python Code 3.19: The self-defined layer ConvolveFilters for an arbitrary fixed convolution, which is used to apply the wavelet filters to the input.

For using the new layer, one has to define the Haar-Wavelet filters either in a numpy file or as a TensorFlow constant in the code. The layer can then be used as a replacement for the first convolution layer of the encoder in the model definition. The implementation includes a util script, haar_wavelets that returns the normalized 2D or 3D Haar wavelets depending on the parameter n. The shown code snippet was altered. 2D specific code is omitted, and only the relevant 3D calculation is presented.

```
1 scaling = numpy.ones(2)
2 wavelet = numpy.array([1, -1])
3 normfactor = 2 ** (-n / 2)
4 mean = numpy.tensordot(scaling, scaling, 0)
5 vertical = numpy.tensordot(scaling, wavelet, 0)
6 horizontal = numpy.tensordot(wavelet, scaling, 0)
7 diagonal = numpy.tensordot(wavelet, wavelet, 0)
8
9 w1 = numpy.tensordot(scaling, mean, 0) * normfactor
10 w2 = numpy.tensordot(scaling, vertical, 0) * normfactor
11 w3 = numpy.tensordot(scaling, horizontal, 0) * normfactor
12 w4 = numpy.tensordot(scaling, diagonal, 0) * normfactor
13 w5 = numpy.tensordot(wavelet, mean, 0) * normfactor
14 w6 = numpy.tensordot(wavelet, vertical, 0) * normfactor
15 w7 = numpy.tensordot(wavelet, horizontal, 0) * normfactor
16 w8 = numpy.tensordot(wavelet, diagonal, 0) * normfactor
```

Python Code 3.20: The calculation of the Haar wavelet filters in numpy

4 Results

4.1 Test Cases and Evaluation

For evaluating the quality of the current state of a trained model, the gold standard in retrieval evaluation, MAP is used. As MAP is an accumulation of average precision values, the retained average precision values for each query are given as well. The MAP values are first calculated for each category of queries, denoted as GMAP. For a further overall comparison with a single value, MAP based on the categories MAP values instead of single average precision values is built, whereas shape similarity is double weighted. The model's training strategy is adjusted in order to improve results as desired.

The test datasets and corresponding queries used for calculating the MAP are described abstractly, such that the tests can be applied to both 3D datasets used as well as for evaluating the later Wavelet model.

4.1.1 Shape Similarity Tests

The first queries used for testing should evaluate the model's overall quality in detecting and retrieving similar shapes. For this, three test cases in total are used in ascending difficulty. Each of these test cases is run with three different input queries, resulting in nine total queries. Using multiple inputs for each test case is done to improve the quality of the scores.

Test Case 1: Single Object Isotropic Scaling
Test Dataset: All basic shapes of the dataset plus multiple isotropic scaled versions of one specific shape
Query: The basic non-scaled version of the chosen shape
Description: A very simple test on how well isotropically scaled shapes are handled as similar shapes all scaled versions are counted as relevant with respect to the average precision calculation and ideally should be retrieved first.

Test Case 2: Single Object Scaling
Test Dataset: All basic shapes of the dataset plus multiple isotropic and anisotropic scaled versions of one specific shape
Query: the basic version of the specific shape
Description: A medium test on how well handled shapes are with either isotropic or anisotropic scaling. For average precision calculations, all scaled versions are seen as relevant and thus should be retrieved first without a special order. More information on this treatment can be found in the corresponding discussion section in the later discussion chapter.

Test Case 3: Multiple Object Scaling
Test Dataset: All basic shapes of the dataset in multiple isotropic and anisotropic scaled versions
Query: the basic version of a chosen shape
Description: A more advanced test on how well handled shapes are with either isotropic or anisotropic scaling. By scaling all shapes in multiple ways, the transformed shapes can no longer be simply distinguished by size from the other shapes. For average precision calculation again, all scaled versions are seen as relevant and thus should be retrieved first without a special order.

4.1.2 Translation Robustness Tests

In the second test group, the robustness of the model against translations is evaluated. The translations overall are minor translations, as in practice objects for CT scans are usually inside a holder, and thus major translations are not relevant for the case. The tests themselves are structured quite similarly to the previous ones but are only performed with one input shape per test case, except for the first case, which is performed with a different shape once for each axis. The total transformations applied are reduced the more axes and objects are translated, to keep the test sets at a pleasant size. In total, five queries are evaluated.

Test Case 4: Single Object Single Axis Translation
Test Dataset: All basic and scaled shapes of the dataset plus multiple single axis translated versions of one specific shape
Query: The basic version of the chosen translated shape
Description: A very simple test on how well minor single-axis translations are handled as similar shapes. All translated versions are counted as relevant with respect to the average precision calculation and ideally should be retrieved first.

Test Case 5: Single Object Translation
Test Dataset: All basic and scaled shapes of the dataset plus multiple single and multi axis translated versions of one specific shape
Query: The basic version of the chosen translated shape
Description: A medium test on how well translations are handled as similar shapes, all translated versions are counted as relevant with respect to the average precision calculation and ideally should be retrieved first.

Test Case 6: Multiple Object Translation
Test Dataset: All basic and scaled shapes of the dataset in multiple single and multi axis translated versions
Query: The basic version of the chosen translated shape
Description: A more difficult, real-data-comparable test on how well translations are handled as similar shapes all translated versions of the input shape are counted as relevant with respect to the average precision calculation and ideally should be retrieved first.

4.1.3 Rotation Robustness Tests

Third, the more complex transformation of rotating the object is used, and the model's robustness is evaluated against it. The tests are structured quite similarly to the previous one and performed with one shape per axis for the first test and a single shape for the subsequent test cases. Like with translations, minor rotations are used only in regards to practical relevance. However, an additional test is added specifically for 90-degree rotations, which could occur in practice when placing the object into the holder. This results in a total of six test queries.

Test Case 7: Single Object Single Axis Rotation
Test Dataset: All basic and scaled shapes of the dataset plus multiple single axis rotated versions of one specific shape
Query: The basic version of the chosen rotated shape
Description: A very simple test on how well minor single-axis rotations are handled as similar shapes. All rotated versions are counted as relevant with respect to the average precision calculation and ideally should be retrieved first.

Test Case 8: Single Object 90 Degree Rotation
Test Dataset: All basic and scaled shapes of the dataset plus multiple 90 degree rotated versions of one specific shape
Query: The basic version of the chosen rotated shape
Description: A very simple test on how well 90-degree rotations are handled as similar shapes. All rotated versions are counted as relevant with respect to the average precision calculation and ideally should be retrieved first.

Test Case 9: Single Object Rotations
Test Dataset: All basic and scaled shapes of the dataset plus multiple single and multi axis rotated versions of one specific shape
Query: The basic version of the chosen rotated shape
Description: A medium test on how well rotations are handled as similar shapes. All rotated versions are counted as relevant with respect to the average precision calculation and ideally should be retrieved first.

Test Case 10: Multiple Object Rotation
Test Dataset: All basic and scaled shapes of the dataset in multiple single and multi axis rotated versions
Query: The basic version of the chosen rotated shape
Description: A more difficult, real-data-comparable test on how well rotations are handled as similar shapes all rotated versions of the input shape are counted as relevant with respect to the average precision calculation and ideally should be retrieved first.

4.1.4 Noise Robustness Tests

At last, the models robustness against various forms of noise is evaluated. The first noise type used is speckle noise, as it is common in CT scans and, furthermore, only affects non-empty

space. The second type of noise used is salt and pepper noise, which is generally common for digital image processing. It affects the empty space but only sets zero or maximum values, so it should still be manageable to handle. Last, random gauss noise is evaluated as a more difficult noise type. The noise test cases are only performed with one shape input each, but a different one for each noise type and for single or multiple noisy objects, which should be sufficient. In total, six test queries are evaluated.

Test Case 11: Single Noisy Volume Speckle
Test Dataset: All basic and scaled shapes of the dataset plus multiple speckle noisy versions of one specific shape
Query: The basic version of the chosen noisy shape
Description: A very simple test on how well speckle noise is handled. All noisy versions are counted as relevant with respect to the average precision calculation and ideally should be retrieved first.

Test Case 12: Multiple Noisy Volumes Speckle
Test Dataset: All basic and scaled shapes of the dataset in speckle noisy versions
Query: The basic version of the chosen noisy volume
Description: A more difficult, real-data-comparable test on how well speckle noise is handled. All noisy versions of the input shape are counted as relevant with respect to the average precision calculation and ideally should be retrieved first.

Test Case 13: Single Noisy Volume Salt & Pepper
Test Dataset: All basic and scaled shapes of the dataset plus multiple salt and pepper noisy versions of one specific shape
Query: The basic version of the chosen noisy shape
Description: A very simple test on how well salt and pepper noise is handled. All noisy versions are counted as relevant with respect to the average precision calculation and ideally should be retrieved first.

Test Case 14: Multiple Noisy Volumes Salt & Pepper
Test Dataset: All basic and scaled shapes of the dataset in salt & pepper noisy versions
Query: The basic version of the chosen noisy volume
Description: A more difficult, real-data comparable test on how well salt and pepper noise is handled All noisy versions of the input shape are counted as relevant with respect to the average precision calculation and ideally should be retrieved first.

Test Case 15: Single Noisy Volume Gaussian
Test Dataset: All basic and scaled shapes of the dataset plus multiple Gaussian noisy versions of one specific shape
Query: The basic version of the chosen noisy shape
Description: A very simple test on how well Gaussian noise is handled. All noisy versions are counted as relevant with respect to the average precision calculation and ideally should be retrieved first.

Test Case 16: Multiple Noisy Volumes Gaussian
Test Dataset: All basic and scaled shapes of the dataset in Gaussian noisy versions
Query: The basic version of the chosen noisy volume
Description: A more difficult, real-data comparable test on how well Gaussian noise is handled. All noisy versions of the input shape are counted as relevant with respect to the average precision calculation and ideally should be retrieved first.

4.2 Shapes3D

The first 3D dataset used in this thesis is a dataset of basic augmented 3D shapes, denoted as Shapes3D. The dataset generator is based on Python's raster_geometry package and is basically a high-level latent generator wrapper of it, with some additional functionality like noise and rotation added. The latent generator gives the opportunity to generate self-defined sub-datasets from it, which are used for incremental training and the generation of the proposed test datasets. The shapes included are cube and cuboid, sphere and ellipsoid, rhomboid, cylinder, and prism. A few random examples of these shapes can be seen in Figure 4.1.

Figure 4.1: The graphic shows 10 random example shapes from the Shapes3D dataset. The examples were restricted to isotropically scaled versions only.

4.2.1 Model Comparison: Performance Evaluation

For a start, the two proposed models are trained for 1000 epochs on the scaled versions of the shapes only. Afterwards, the better model is further trained with first translation as augmentation only and second with translation, rotation, and noise as augmentation. The retained average precision, GMAP and MAP scores for all four models are presented in Table 4.1. Visualized examples for some test case queries are shown in Figure 4.2, along with the corresponding first ten retrieval results of the best model in Figure 4.3.

Query	Model 1	Model 2	Model 2 "Translation"	Model 2 "Augmentation"
1: Cuboid	0.6488	0.6667	0.7750	0.8125
1: Rhomboid	1.0000	1.0000	1.0000	1.0000
1: Prism	1.0000	0.9500	1.0000	0.9500
2: Cuboid	0.8318	0.8241	0.9067	0.8065
2: Rhomboid	0.9644	0.9787	0.9805	0.9651
2: Prism	0.9952	0.9502	0.8984	0.9707
3: Cuboid	0.6791	0.6730	0.8294	0.6310
3: Rhomboid	0.3760	0.5336	0.6216	0.5568
3: Prism	0.9491	0.7593	0.4440	0.8913
GMAP Shape	0.8272	0.8151	0.8284	0.8427
4: Cuboid	0.9673	0.9705	0.9861	0.9615
4: Ellipsoid	0.8554	0.8871	0.9034	0.9182
4: Cylinder	0.8423	0.8772	0.9152	0.8831
5: Prism	0.9991	0.9970	0.9859	0.9981
6: Rhomboid	0.3677	0.5614	0.4531	0.5684
GMAP Translation	0.8063	0.8586	0.8487	0.8659
7: Cuboid	0.9533	0.9506	0.9780	0.9294
7: Ellipsoid	0.8491	0.8389	0.8799	0.8829
7: Cylinder	0.7986	0.8204	0.8808	0.8422
8: Cylinder	0.9700	0.9565	0.9529	0.9475
9: Prism	0.9998	0.9951	0.9765	0.9985
10: Rhomboid	0.7698	0.8281	0.8407	0.6618
GMAP Rotation	0.8901	0.8983	0.9181	0.8771
11: Cuboid	0.5250	0.9533	0.9708	0.9606
12: Cuboid	0.2909	0.7706	0.8420	0.7573
13: Rhomboid	0.9047	0.9621	0.9821	0.9451
14: Ellipsoid	0.7166	0.7954	0.8913	0.7561
15: Prism	0.5248	0.9609	0.9471	0.9689
16: Cylinder	0.4462	0.7070	0.9165	0.7294
GMAP Noise	0.5680	0.8582	0.9249	0.8529
MAP	0.7838	0.8490	0.8697	0.8562

Table 4.1: The table shows the received retrieval scores for the two proposed model architectures, with basic in columns one and two. For the better Model 2, the results after more sophisticated training with additional translation or full augmentation training are presented in columns three and four.

4.2.2 Model 2 "Translation": Retrieval Visualization

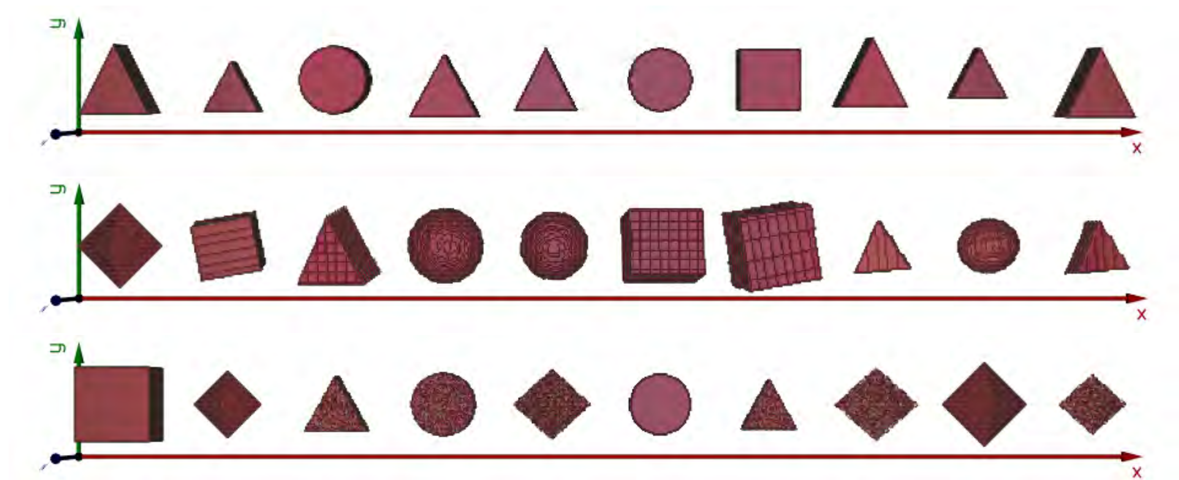

Figure 4.2: The graphic shows the first 10 shapes of three chosen test query examples for translation robustness in row one, rotation robustness in row two, and noise robustness in row three. An example for shape similarity queries is omitted, as scaled versions are already shown in the example images of the database.

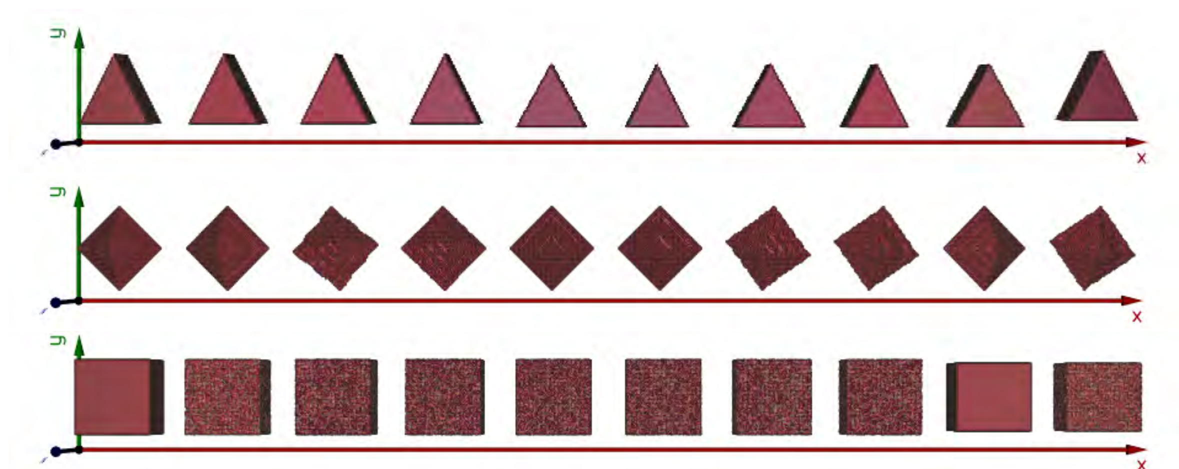

Figure 4.3: The graphic shows the first 10 shapes retrieved by Model 2 "Translation". The results correspond to the test case queries presented above.

4.3 Blended Multi-Shapes3D and Wavelet Integration

The second 3D dataset used is the Mult-Shapes3D dataset, which consists of three augmented real object CT-scans and three synthetic double shapes. The shapes are displayed in Figure 4.4 and have a volume size of $64 \times 64 \times 64$. The Mult-Shapes3D dataset enables the evaluation of the model 2 Translations' effectiveness in capturing intricate details of more complex 3D shapes. It gives a more real-life comparable benchmark for the performance of the so far best model and is thus also used for the comparison with the Haar Wavelet integrated Model 3.

Figure 4.4: The graphic shows the six shapes from the Multi-Shapes3D dataset. The first three shapes (piston, egg with kangaroo, and egg with figure) are down-sampled and clipped versions of real-object CT scans. The other three shapes are synthetic shapes, generated by merging two shapes from the Shapes3D dataset into one volume.

4.3.1 Model Comparison: Performance Evaluation

The two models are compared using the defined test cases as previously mapped on the Multi-Shapes3D database. Graphic Figure 4.5 shows the plotted loss curve for Model 2 "Translation" and Model 3 "Haar Wavelet" on Multi-Shapes3D.

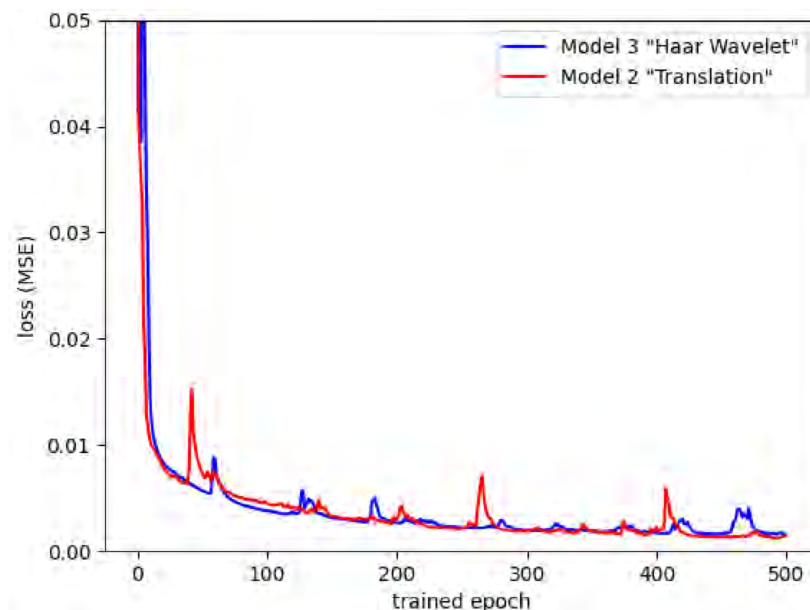

Figure 4.5: The loss curves of Model 2 and Model 3

The plot is clipped in the y direction for better visualization. The actual starting values after one epoch are 0.2743 for Model 2 and 0.0996 for Model 3. Both models reach a value of around 0.0400 after the second training epochs, which further training curve is approximately the same as the presented plot.

The retrieved average precision and MAP scores are shown in Table 4.2. The training times per epoch are four seconds for Model 2, five seconds for Model 3 with filter loading from file and again four seconds for Model 3 with filters hard-coded in the model declaration. Addressing memory usage, no significant differences are obtained.

Query	Model 2 "Translation"	Model 3 "Haar Wavelet"
1: Cuboid/Rhomboid	0.6667	0.7679
1: Piston	0.9167	1.0000
1: Prism/Cylinder	0.8304	0.7361
2: Cylinder/Ellipsoid	0.7511	0.7389
2: Egg 1	1.0000	1.0000
2: Prism/Cylinder	0.7389	0.7245
3: Egg 2	0.9805	0.9805
3: Cuboid/Rhomboid	0.7712	0.7462
3: Prism/Cylinder	0.4300	0.3934
GMAP Shape	0.7873	0.7875
4: Piston	0.9986	1.0000
4: Cuboid/Rhomboid	0.9468	0.9599
4: Egg 2	0.9943	0.9938
5: Prism/Cylinder	0.9842	0.9841
6: Cuboid/Rhomboid	0.7152	0.6158
GMAP Translation	0.9278	0.9107
7: Cuboid/Rhomboid	0.9229	0.9357
7: Cylinder/Ellipsoid	0.8210	0.7740
7: Egg 1	0.9909	0.9956
8: Egg 2	0.9964	0.9991
9: Prism/Cylinder	0.9740	0.9694
10: Cuboid/Rhomboid	0.6873	0.7247
GMAP Rotation	0.8988	0.8997
11: Cuboid/Rhomboid	0.9443	0.6607
12: Piston	0.2848	0.2104
13: Cylinder/Ellipsoid	0.8110	0.6071
14: Egg 1	0.9205	0.3404
15: Prism/Cylinder	0.8433	0.4326
16: Egg 2	0.2674	0.2241
GMAP Noise	0.6785	0.4126
MAP	0.8159	0.7596

Table 4.2: The table shows the received GMAP and overall MAP scores of the previous best Model 2 "Translation" and Model 3 "Haar Wavelet". Model 3 has the integration of wavelet filters as the first layer on Blended Multi-Shapes3D. The models were both trained for 500 epochs before evaluation.

4.3.2 Exploring Edge Cases: Poor Retrieval Results

This subsection provides visualizations of edge cases with comparable bad retrieval results. This aims to gain a deeper understanding of the capabilities of the proposed models. These visualizations provide insights into the retrieval performance of Model 2 and Model 3 on easy test cases, highlighting the instances where the models struggled to produce accurate results. Figures 4.6, 4.7, and 4.8 display the input shapes and retrieval results for Model 2 and Model 3 on the chosen bad results of easy test cases from the Multi-Shapes3D dataset.

Figure 4.6: The graphic shows the first 10 shapes of the three chosen single transformation test queries, which should be comparable easy but returned quite bad results for both models. The chosen queries are 1: Cuboid/Rhomboid presented in row one, 2: Prism/Cylinder in row two, and 7: Cylinder/Ellipsoid in row two.

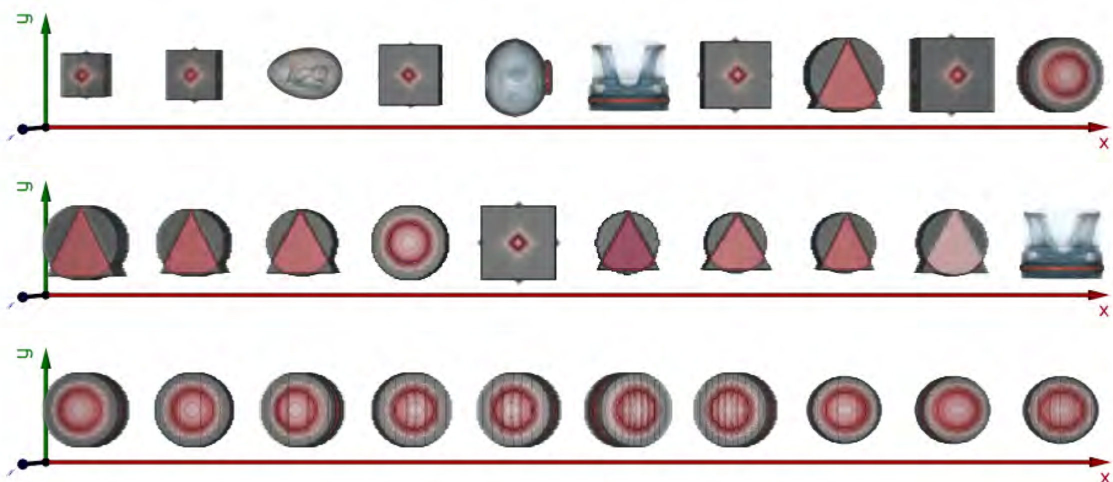

Figure 4.7: The graphic shows the first 10 shapes retrieved by Model 2 "Translation" for the chosen bad results on easy test case examples.

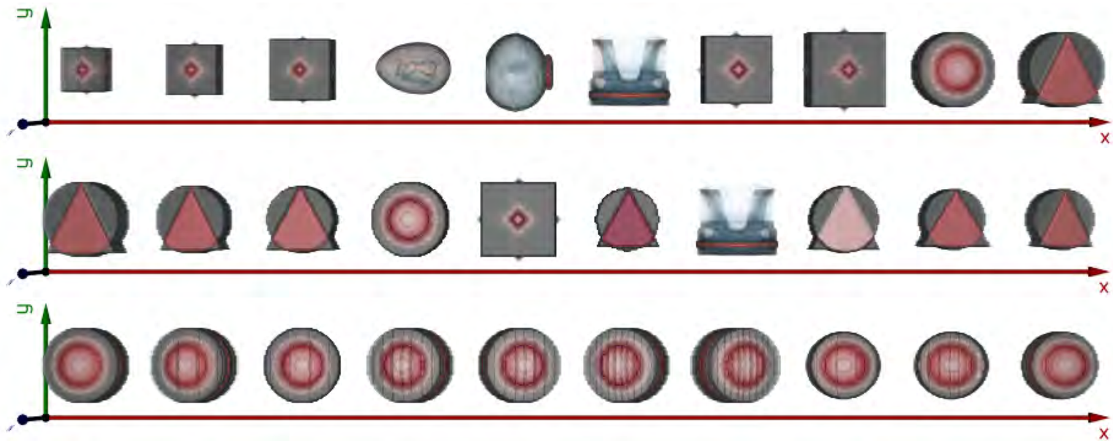

Figure 4.8: The graphic shows the first 10 shapes retrieved by Model 3 "Haar Wavelet" for the chosen bad results on easy test case examples.

4.3.3 Real Data: Retrieval Visualization

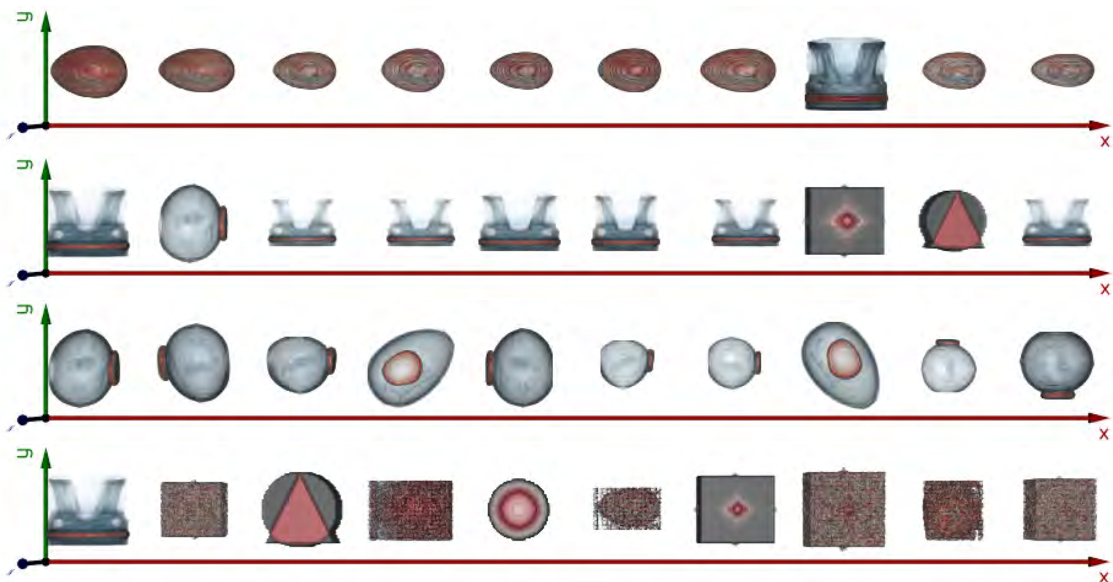

Figure 4.9: The graphic shows the first 10 shapes of four test query examples, each chosen at random from the real data test cases of each category. Test case 2 with Egg 1 as query is shown in row one; test case 4 with Piston as query is shown in row two; test case 8 with Egg 2 as query is shown in row three; and test case 12 with Piston as query is shown in row four.

This subsection presents the visualizations of test queries with real data-based input and the corresponding retrieval results for Model 2 and Model 3 on the Multi-Shapes3D dataset. These visualizations provide insights into the retrieval performance of Model 2 and Model 3 on real data test queries, giving an understanding of how the models handle real-world

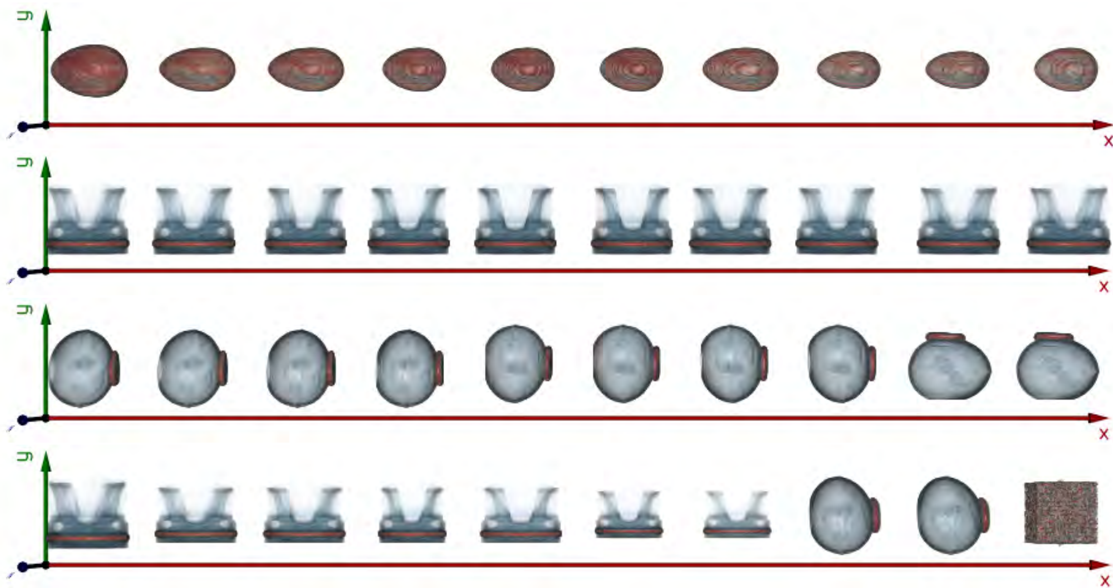

Figure 4.10: The graphic shows the first 10 shapes retrieved by Model 2 "Translation" for the randomly chosen real data test cases.

shapes.

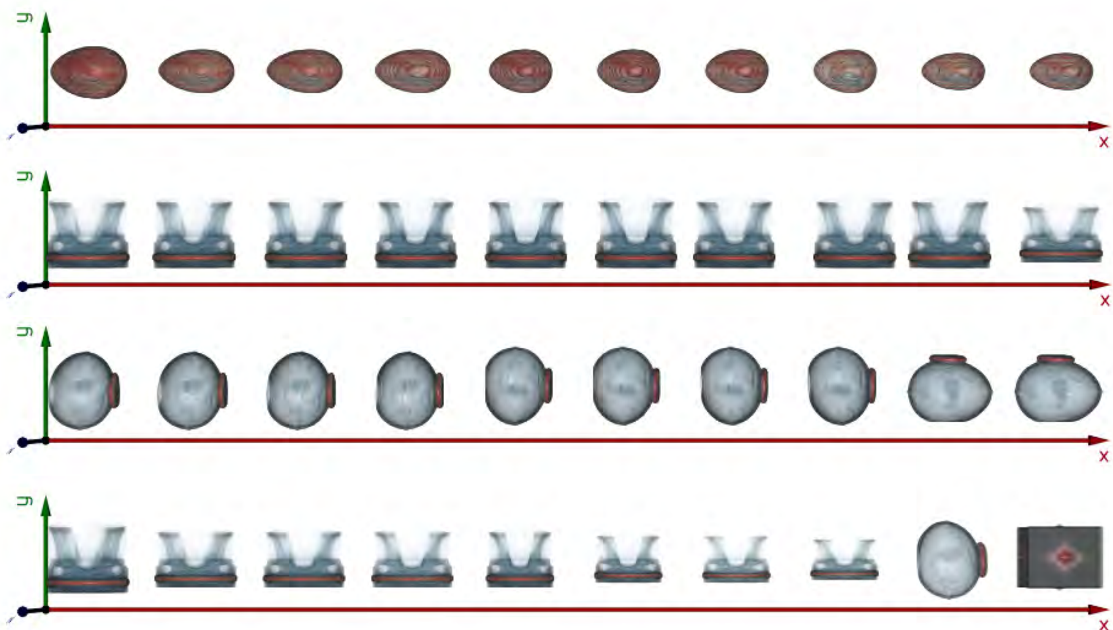

Figure 4.11: The graphic shows the first 10 shapes retrieved by Model 3 "Haar Wavelet" for the randomly chosen real data test cases.

4.3.4 Differences in Noise Robustness

The wavelet-expanded model performed comparable well except for the noise test cases. For this reason, a deeper look into the encoding under various types of noise is specifically examined here. The noise types considered in this evaluation are speckle noise, salt and pepper noise, and Gaussian noise. The changing encoding of both models is visualized and compared as the noise scale varies for each type of noise.

The resulting encodings for the different noise types are displayed in the corresponding figures. Figure 4.12 showcases the encodings for the cuboid/rhomboid shape augmented with speckle noise. Figure 4.13 presents the encodings for the cylinder/ellipsoid shape augmented with salt and pepper noise. Figure 4.14 illustrates the encodings for the egg1 shape augmented with Gaussian noise.

Insights into the noise robustness of the wavelet expanded model are provided by examining these encodings. The encodings under different noise conditions allow for the assessment of the model's ability to handle noise and the identification of any limitations or areas for improvement.

Speckle Noise

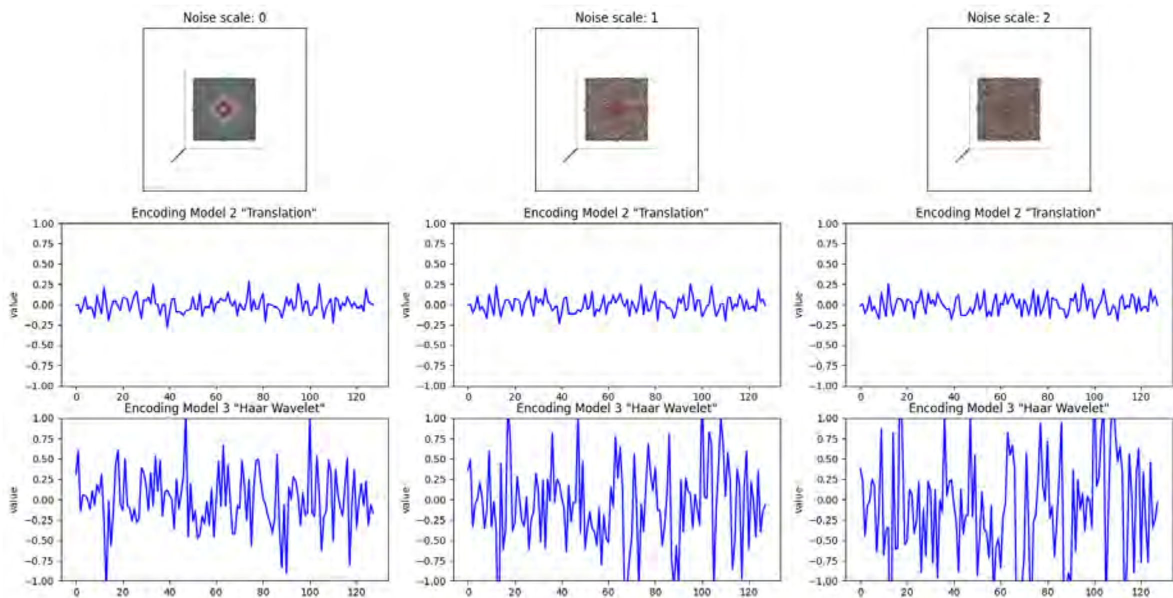

Figure 4.12: The graphic presents the received encoding for the cuboid/rhomboid shape augmented by speckle noise in ascending noise scale order. The noise is scaled to fit a (0,255) gray-scale image; noise scale 2 corresponds to a Gauss standard derivation of 2/255 for the multiplicative part.

Salt & Pepper Noise

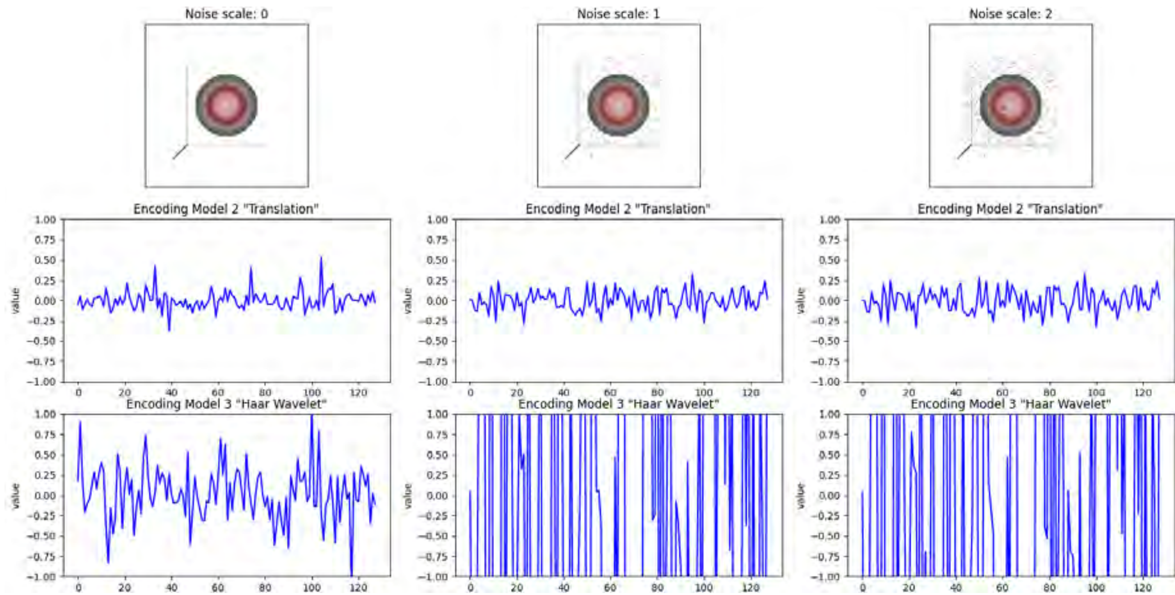

Figure 4.13: The graphic presents the received encoding for the cylinder/ellipsoid shape augmented by salt and pepper noise in ascending order. The noise is scaled to fit a (0,255) gray-scale image; noise scale 2 corresponds to a probability of 2/255 for a changed pixel.

Gauss Noise

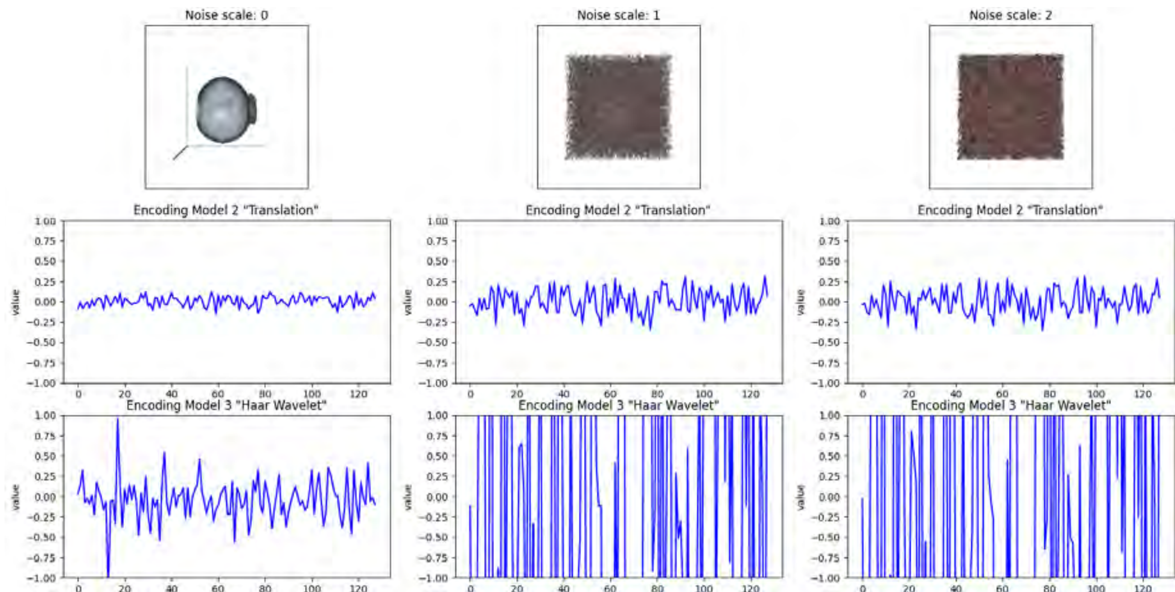

Figure 4.14: The graphic presents the received encoding for the egg1 shape augmented by gauss noise in ascending noise scale order. The noise is scaled to fit a (0,255) gray-scale image; noise scale 2 corresponds to a Gauss standard derivation of 2/255.

5 Discussion

5.1 Handling Isotropic versus Anisotropic Scaling

Concerning the second and third test cases, all scaled versions of the objects are considered relevant. However, in terms of human perception or mathematical reasoning, the algorithm should ideally prioritize returning the isotropically scaled objects before the anisotropically scaled ones. Humans tend to perceive proportionally scaled versions of objects as the same but scaled, while anisotropically scaled objects are perceived as similar but distinct objects. For instance, a scaled cube remains a cube, while a cuboid is a distinct shape resulting from scaling one side only. From an objective standpoint, objects with only one side length changed are often spatially more overlapping than those with all side lengths scaled.

In conclusion, while treating all scaled versions of objects as relevant in the second and third test cases is acceptable for evaluating the algorithm's ability to detect similar shapes, a more human-centric or mathematically logical approach would prioritize isotropic scaling before anisotropic scaling. This approach would result in a similarity measure that aligns more with human perception. However, implementing this approach would require more sophisticated performance measures than MAP to properly account for both types of scaling. As this thesis is intended to provide groundwork for research, this is left open for future researchers.

5.2 Model Size and Augmentation

The first results obtained for Shapes3D by comparing Model 1 with basic shape training and Model 2 with basic shape training are mostly as expected. Overall, the bigger model delivers better results. Except for the noise robustness tests, the results are, however, quite comparable. If noise robustness is not desired, one might even prefer the smaller model for complexity reasons. For the thesis, however, the second model is chosen to be further evaluated as noise is quite common in CT scans.

Let's take a look at the differences between Model 2 and Model 2 further trained with random translation as augmentation. Overall, the translation-trained version performs better than expected, but the group scores are a bit surprising at first. The model gets better in the shape, rotation, and noise similarity tests. At the translation tests, which it was explicitly trained on, it performs worse.
A similar unexpected behavior occurred with the fully augmentation-trained Model 2. It delivers better results concerning shape and translation similarity but worse results for rotation and noise similarity.

If you think about this, at first sight, contrary behavior becomes logic. Although augmentation is used a lot in classification tasks, using it in AEs cannot accomplish the same improve-

ments. In classification, a predefined label is used for training, and thus the model learns to correctly detect the augmented shapes of these classes. An AE on the other hand, uses the input as a target for learning and compares it with its own reconstruction. As a comparison with the MSE or MAE is pointwise, augmentation is part of the reconstruction error obtained for adjusting the weights. Thus, as soon as augmented shapes are used for training, the model is forced to encode at least some information for restoring the augmentation information in the decoder later.

Having this behavior of augmentation with AEs in mind, the obtained scores became obvious. Adding translation forces the model to encode spatial position information into the latent dimension. This position information helps distinguish different shapes from each other, even if they are rotated or noisy. Then adding noise and rotation information in the latent dimension improves shape and translation similarity, with the trade-off of reducing the former robustness.

Using one type of augmentation improves the robustness against all others and thus the overall performance of a model. Using too much augmentation can, however, decrease performance again. It seems plausible to use translation augmentation as is for the more advanced dataset and the given task. The real data objects are usually placed in a holder that is in a fixed position. Translations should thus not occur much, but the object can be rotated inside as well as the images obtained noisy. Using translation as augmentation thus makes the model most robust against real-world occurring augmentations in CT scans.

5.3 Using Haar Wavelet Filters

The rationale for using Haar wavelet filters was based on their reputation as high-quality edge detectors. The hope was that using them would reduce the number of trainable parameters by eliminating a convolutional layer, resulting in more efficient training and faster backpropagation.

However, the experiments conducted did not yield the desired results. Training times per epoch remained the same when using the filters hard-coded, and even increased when loading the filters from a file. It appears that having one less convolutional layer for backpropagation does not have a significant impact on performance compared to using a self-implemented convolution layer with fixed filters. Although the self-defined layer may be less optimized, the difference should be minor since it utilizes TensorFlow's convolution operation.
There were no significant differences observed in memory usage. When increasing the number of filters until a GPU memory error occurs, the error typically arises during the second convolution operation. As the second layer is trainable anyway, using fixed filters in the first convolution step has little effect on the architecture's memory usage.
In addition to training times and memory usage, evaluating loss over time is also important. Since one less layer of filters is trained, the number of epochs required to reach a (local) optimum may be significantly reduced, resulting in an overall reduced training time, although the computation time per epoch remains constant. Unfortunately, the results did not confirm this hypothesis either. The models achieved comparable loss values after only two epochs, with only a small advantage observed for the Haar-Wavelet filters during the first 100 epochs. The only significant difference was observed at the beginning and after a single training

epoch. However, it is important to note that Haar-Wavelet filters still outperformed random initialization at first.

Concerning similarity, Model 3 performed similarly to Model 2 for shape similarity and translation robustness. This is as expected, as the filters are known for their high-quality edge detection, making them suitable for subsequent high-level feature detection. Translation invariance is achieved through pooling and is independent of the filters used in the first layer.
The results regarding rotation robustness were unexpected but impressive. Despite rotated edges producing distinct feature maps, the model was able to handle them well. It was originally hypothesized that the learned edge detectors might be more robust to rotation, but it seems that rotation is handled by the deeper pooling and convolution layers in a similar way to translation. This is consistent with the common understanding that CNNs tends to learn edge-detecting filters in the first layer, which are then used for higher-level feature detection in subsequent layers. Overall, these findings suggest that the use of Haar wavelet filters in the first convolutional layer may not be as crucial for rotation robustness as originally thought.

However, the quality of the results dropped significantly when noise was introduced. The additional visualization with ascending noise scales clearly illustrates what is happening. Speckle noise, which only alters the shape but not the empty volume, is still handled well, but edges become slightly disorganized, and the model performs worse than Model 2. Nevertheless, the latent representation still looks similar.
On the other hand, a small amount of salt and pepper noise was enough to completely disrupt the algorithm. The Haar-Wavelet filters detected edges everywhere in the volume, making it impossible to recover them in subsequent layers. As a result, the latent representation was a nearly random mess with very little relevant information for similarity. This behavior was consistent with the handling of Gaussian noise. Edges were detected everywhere, resulting in a completely disorganized encoding. The latent dimension had even less similarity-relevant information, if any.

6 Conclusion and Future Work

Considering the concept of similarity in latent encodings, there is still room for further research on designing a specific distance function. While existing literature has covered the differences between various distance metrics, more research is needed on how they interact with different forms of normalization. This would help improve the accuracy and efficiency of similarity measures in latent space.

For the general model architecture, using two or three convolutional layers and two dense layers seems appropriate. There hasn't been much change in recent years, and neither has this thesis. Using slowly increasing filter kernel sizes is also in accordance with state-of-the-art architectures. The newly proposed activations, however, achieved better results than the state-of-the-art go-to approach of using ReLU. The brief comparison of the 2D dataset results might be worth expanding into in-depth research on the 3D models. In terms of regularization, the extremely bad results achieved here are open to further exploration. Quite a few papers used some form of regularization to receive sparse latent encodings. However, diving further into regularization in the field of CAEs and exploring sparse representation for similarity measures would have been out of the scope of this thesis.

The proposed test suite and evaluation measure for 3D shape retrieval have proven effective in assessing the quality of various model architectures on the Shapes3D and Multi-Shapes3D datasets. The final architectures have achieved well-working results, which seem to be overall promising to further develop into a real-life retrieval algorithm.

While various data augmentations were found to improve the models performances, they may not be ideal for similarity-based tasks. They enforce the encoding of spatial information such that the reconstruction error is minimized. As spatial data is not relevant for shape retrieval, future research could explore the use of denoising AEs as an alternative. There seem to be three reasonable ways to go: directly train the proposed models further in an all-denoising way; either train two separate AEs, one for denoising and one for shape matching; or even train a separate AE for each augmentation form to denoise before matching shapes.

Regarding the use of wavelet filters for faster computations, the results obtained were comparable in shape matching as well as computation time and memory usage, with a strong trade-off in terms of noise. Real improvements could only be achieved in the first two epochs, regarding the loss values. Still, the simple approach of just using the Haar-Wavelet filters as the first layer performed comparably well in most cases, which shows that further research into such an approach could be beneficial.

In terms of future work, there are several approaches that could be explored to further fine-tune the Haar-Wavelet integration for more efficient computation and memory usage. One potential option is to apply the wavelet filters to the dataset as a preprocessing step. This would allow the AE to directly encode and reconstruct the multiple wavelet feature maps. By this approach, the first wavelet layer as well as the last deconvolution layer could be

omitted, resulting in a smaller AE. To implement this approach, the wavelet filter maps could be calculated once for every training set, and a Haar wavelet preprocessed dataset could be created. For the test set, the input pipeline could take care of the first filtering step. This would eliminate one layer each in the encoder and decoder. The complexity of the input filtering and deconvolution would thus be reduced from $O(2mn)$ down to $O(n)$ where m is the number of training epochs and n the number of training examples. The bigger loss calculation should be negligible in comparison to the total calculation time. Aside from the theoretical big computation improvements, it should be noted that the issue of bad noise handling would still need to be addressed. One solution to this could be to explore a chain of denoising, wavelet filtering, and shape encoding techniques.

Another potential option for improving the use of Haar wavelet filters in CAEs is to consider initializing the first layer of a CAE with Haar-Wavelet filters rather than using random initialization. Model 3 yielded significantly better results for very little training time, suggesting that it could be a promising approach worth further investigation. However, it remains an open question whether this initialization strategy could cause the model to become stuck in a local optimum and fail to handle noise effectively, or if the model would adapt out of this niche with further training. Further research is needed to explore this possibility.

Overall, this thesis provides a thorough analysis of using CAEs for similarity-based shape retrieval. While the proposed models show promise for real-world applications, further optimization is needed to improve accuracy on transformed data as well as computation and memory efficiency. Although compression is a legitimate tool to use for the retrieval of 3D data, the compression used to get to a 64^3 volume removed quite a bit of information relevant, for example, to correctly sort similar shapes with different textures.

Bibliography

1. Turk, M. & Pentland, A. Eigenfaces for recognition. *Journal of Cognitive Neuroscience* **3,** 71–86 (1991).
2. Geiger, A., Lenz, P. & Urtasun, R. *Are we ready for autonomous driving? The KITTI vision benchmark suite* in *Conference on Computer Vision and Pattern Recognition* (2012), 3354–3361.
3. Litjens, G. *et al.* A survey on deep learning in medical image analysis. *Medical Image Analysis* **42,** 60–88 (2017).
4. Arulkumar, S., Ganesan, K. & Priyadharsini, R. A review of computer vision applications in surveillance systems. *Journal of Ambient Intelligence and Humanized Computing* **10,** 3407–3423 (2019).
5. He, K., Zhang, X., Ren, S. & Sun, J. *Deep residual learning for image recognition* in *Conference on Computer Vision and Pattern Recognition* (2016), 770–778.
6. Simonyan, K. & Zisserman, A. *Very deep convolutional networks for large-scale image recognition* in *International Conference on Learning Representations* (2015).
7. Long, J., Shelhamer, E. & Darrell, T. *Fully convolutional networks for semantic segmentation* in *Conference on Computer Vision and Pattern Recognition* (2015), 3431–3440.
8. Krizhevsky, A., Sutskever, I. & Hinton, G. E. *Imagenet classification with deep convolutional neural networks* in *Advances in Neural Information Processing Systems* (2012), 1097–1105.
9. He, K., Zhang, X., Ren, S. & Sun, J. *Identity mappings in deep residual networks* in *European Conference on Computer Vision* (2016), 630–645.
10. Ma, J., Jiang, X., Fan, A., Jiang, J. & Yan, J. Image Matching from Handcrafted to Deep Features: A Survey. *International Journal of Computer Vision* **129,** 23–79 (2021).
11. Duda, R. O., Hart, P. E. & Stork, D. G. *Pattern Classification* 2nd (John Wiley & Sons Inc. A Wiley-Interscience Publication, New York, USA, 2001).
12. Stahl, T., Koch, C. & Wersig, S. Künstliche Intelligenz in der Praxis: Autoencoder – Eine vielseitig einsetzbare Architektur. *AI Spektrum - Das Magazin für künstliche Intelligenz* (2019).
13. A. Goodrum, A. Image Information Retrieval: An Overview of Current Research. *Informing Science: The International Journal of an Emerging Transdiscipline* **3,** 063–066 (2000).
14. Humblet, C. & Dunbar, J. B. in *Annual reports in medicinal chemistry, Volume 28* (ed Hagmann, W. K.) 275–284 (Academic Press, 1993). ISBN: 9780120405282.

15. Ankerst, M., Kastenmüller, G., Kriegel, H.-P. & Seidl, T. in *Advances in spatial databases* (eds Goos, G. *et al.*) 207–226 (Springer, 1999). ISBN: 978-3-540-66247-1.
16. Saupe, D. & Vranić, D. V. *3D Model Retrieval with Spherical Harmonics and Moments* in *Pattern recognition* (eds Goos, G., Hartmanis, J., van Leeuwen, J., Radig, B. & Florczyk, S.) **2191** (Springer, 2001), 392–397. ISBN: 978-3-540-42596-0.
17. Vranic, D. V., Saupe, D. & Richter, J. *Tools for 3D-object retrieval: Karhunen-Loeve transform and spherical harmonics* in *2001 IEEE Fourth Workshop on Multimedia Signal Processing (Cat. No.01TH8564)* (2001), 293–298.
18. Osada, R., Funkhouser, T., Chakzelle, B. & Dobkin, D. *Shape Distributions* Princeton, USA, 2002.
19. Chen, D.-Y., Tian, X.-P., Shen, Y.-T. & Ouhyoung, M. On Visual Similarity Based 3D Model Retrieval. *Computer Graphics Forum* **22,** 223–232 (2003).
20. Assfalg, J., Bertini, M., Bimbo, A. D. & Pala, P. Content-Based Retrieval of 3-D Objects Using Spin Image Signatures. *IEEE Transactions on Multimedia* **9,** 589–599 (2007).
21. Jain, V. & Zhang, H. A spectral approach to shape-based retrieval of articulated 3D models. *Computer-Aided Design* **39,** 398–407 (2007).
22. Papadakis, P., Pratikakis, I., Theoharis, T., Passalis, G. & Perantonis, S. *3D Object Retrieval using an Efficient and Compact Hybrid Shape Descriptor* in *Eurographics 2008 Workshop on 3D Object Retrieval* (eds Stavros Perantonis, Nikolaos Sapidis, Michela Spagnuolo & Daniel Thalmann) (The Eurographics Association, 2008). ISBN: 978-3-905674-05-7.
23. Zaharescu, A., Boyer, E., Varanasi, K. & Horaud, R. *Surface feature detection and description with applications to mesh matching* in *2009 IEEE Conference on Computer Vision and Pattern Recognition* (IEEE, 2009), 373–380. ISBN: 978-1-4244-3992-8.
24. Agathos, A. *et al.* 3D articulated object retrieval using a graph-based representation. *Visual Computer* **26,** 1301–1319 (2010).
25. Lavoué, G. Combination of bag-of-words descriptors for robust partial shape retrieval. *The Visual Computer* **28,** 931–942 (2012).
26. Tabia, H. & Laga, H. Covariance-Based Descriptors for Efficient 3D Shape Matching, Retrieval, and Classification. *IEEE Transactions on Multimedia* **17,** 1591–1603 (2015).
27. Yang, C. & Yu, Q. Multiscale Fourier descriptor based on triangular features for shape retrieval. *Signal Processing: Image Communication* **71,** 110–119 (2019).
28. Iyer, N., Jayanti, S., Lou, K., Kalyanaraman, Y. & Ramani, K. Three-dimensional shape searching: state-of-the-art review and future trends. *Computer-Aided Design* **37,** 509–530 (2005).
29. Tangelder, J. W. H. & Veltkamp, R. C. A survey of content based 3D shape retrieval methods. *Multimedia Tools and Applications* **39,** 441 (2007).
30. Devareddi, R. B. & Srikrishna, A. *Review on Content-based Image Retrieval Models for Efficient Feature Extraction for Data Analysis* in *2022 International Conference on Electronics and Renewable Systems (ICEARS)* (2022), 969–980.

31. Le Cun, Y. *et al. Handwritten Digit Recognition with a Back-Propagation Network* in *Proceedings of the 2nd International Conference on Neural Information Processing Systems* (ed MIT Press) (1989), 396–404.

32. LeCun, Y., Cortes, C. & Burges, C. *MNIST handwritten digit database* `http://yann.lecun.com/exdb/mnist/` (2022).

33. Hinton, G. E. & Salakhutdinov, R. R. Reducing the Dimensionality of Data with Neural Networks. *Science (New York, N.Y.)* **313,** 504–507 (2006).

34. Krizhevsky, A. & Hinton, G. *Using Very Deep Autoencoders for Content-Based Image Retrieval* tech. rep. (2011).

35. Salakhutdinov, R. & Hinton, G. Semantic hashing. *International Journal of Approximate Reasoning* **50,** 969–978 (2009).

36. Xu, G. & Fang, W. *Shape retrieval using deep autoencoder learning representation* in *13th International Computer Conference on Wavelet Active Media Technology and Information Processing (ICCWAMTIP)* (IEEE, 2016), 227–230. ISBN: 978-1-5090-6126-6.

37. Cai, Z., Gao, W., Yu, Z., Huang, J. & Cai, Z. *Feature extraction with triplet convolutional neural network for content-based image retrieval* in *Proceedings of the 2017 12th IEEE Conference on Industrial Electronics and Applications (ICIEA)* (IEEE, 2017), 337–342. ISBN: 978-1-5090-6161-7.

38. Cheng, Z., Sun, H., Takeuchi, M. & Katto, J. *Performance Comparison of Convolutional AutoEncoders, Generative Adversarial Networks and Super-Resolution for Image Compression* arXiv.org. 1807.00270v1 (Tokyo, Japan, 2018).

39. Zhu, Z., Wang, X., Bai, S., Yao, C. & Bai, X. *Deep Learning Representation using Autoencoder for 3D Shape Retrieval* arXiv.org (PR, China, 2014).

40. Liu, Z.-M. *et al. 3D model retrieval based on deep Autoencoder neural networks* in *2017 International Conference on Signals and Systems* (2017), 290–296.

41. Zhou, W. & Jia, J. A learning framework for shape retrieval based on multilayer perceptrons. *Pattern Recognition Letters* **117,** 119–130 (2019).

42. Leng, B., Guo, S., Zhang, X. & Xiong, Z. 3D object retrieval with stacked local convolutional autoencoder. *Signal Processing* **112,** 119–128 (2015).

43. Xie, J., Fang, Y., Zhu, F. & Wong, E. *Deepshape: Deep learned shape descriptor for 3D shape matching and retrieval* in *2015 IEEE Conference on Computer Vision and Pattern Recognition* (2015), 1275–1283.

44. Bu, S., Han, P., Liu, Z., Han, J. & Lin, H. Local deep feature learning framework for 3D shape. *Computers & Graphics* **46,** 117–129 (2015).

45. Wang, Y., Xie, Z., Xu, K., Dou, Y. & Lei, Y. An efficient and effective convolutional auto-encoder extreme learning machine network for 3d feature learning. *Neurocomputing* **174,** 988–998 (2016).

46. Yu, E. M. & Sabuncu, M. R. *A Convolutional Autoencoder Approach To Learn Volumetric Shape Representations For Brain Structures* in *2019 IEEE 16th International Symposium on Biomedical Imaging* (2019), 1559–1562. ISBN: 1945-8452.

47. Kausar, T., Wang, M., Idrees, M. & Lu, Y. HWDCNN: Multi-class recognition in breast histopathology with Haar wavelet decomposed image based convolution neural network. *Biocybernetics and Biomedical Engineering* **39,** 967–982 (2019).
48. Gentleman, R., Ding, B., Dudoit, S. & Ibrahim, J. in *Bioinformatics and Computational Biology Solutions Using R and Bioconductor* (eds Gentleman, R., Carey, V. J., Huber, W., Irizarry, R. A. & Dudoit, S.) 189–208 (Springer, New York, NY, 2005).
49. Khachumov, M. V. Distances, metrics and cluster analysis. *Scientific and Technical Information Processing* **39,** 310–316 (2012).
50. Hoi, S., Liu, W., Lyu, M. R. & Ma, W.-Y. *Learning Distance Metrics with Contextual Constraints for Image Retrieval* in *Proceedings / 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, CVPR 2006, June 17 - 22, 2006, New York, NY* (eds Fitzgibbon, A., Taylor, C. J. & LeCun, Y.) (IEEE Computer Society, 2006), 2072–2078. ISBN: 0-7695-2597-0.
51. Rosenblatt, F. *The perceptron - A perceiving and recognizing automaton* BibTeX 85-460-1 (Ithaca, New York, 1957).
52. Goodfellow, I., Bengio, Y. & Courville, A. *Deep Learning* (The MIT Press, Cambridge, USA, 2016).
53. *Machine Learning mit Python und Keras, TensorFlow 2 und Scikit-learn: Das umfassende Praxis-Handbuch für Data Science, Deep Learning und Predictive Analytics* 3rd (eds Raschka, S. & Mirjalili, V.) (MITP, Frechen, Germany, 2021).
54. Werner, M. *Digitale Bildverarbeitung: Grundkurs mit neuronalen Netzen und MATLAB-Praktikum* (Springer Vieweg, Wiesbaden, 2021).
55. Dumoulin, V. & Visin, F. *A guide to convolution arithmetic for deep learning* arXiv.org. 1603.07285v2 (2018).
56. Springenberg, J. T., Dosovitskiy, A., Brox, T. & Riedmiller, M. *Striving for Simplicity: The All Convolutional Net* arXiv.org. 1412.6806v3 (Freiburg, 2014).
57. Zhou, Y.-T. & Chellappa, R. *Computation of optical flow using a neural network* in *IEEE 1988 International Conference on Neural Networks* **2** (1988), 71–78. `https://www.semanticscholar.org/paper/Computation-of-optical-flow-using-a-neural-network-Zhou-Chellappa/32bbc3ac0235055f5d8cf4fe8a8a3637e2017e3c`.
58. Sermanet, P., Chintala, S. & LeCun, Y. *Convolutional Neural Networks Applied to House Numbers Digit Classification* arXiv.org. 1204.3968v1 (2012).
59. Rumelhart, D. E., Hinton, G. E. & Williams, R. J. *Learning Internal Representations by Error Propagation* (Defense Technical Information Center, Fort Belvoir, VA, 1985).
60. Ballard, D. H. *Modular Learning in Neural Networks* in *Proceedings of the Sixth National Conference on Artificial Intelligence* (AAAI Press, 1987), 279–284. ISBN: 0934613427.
61. Le Cun, Y. & Fogelman-Soulié, F. Modèles connexionnistes de l'apprentissage. *Intellectica Revue de l Association pour la Recherche Cognitive* **2,** 114–143 (1987).
62. jun94. *[DL] 12. Unsampling: Unpooling and Transpose Convolution* `https://medium.com/jun94-devpblog/dl-12-unsampling-unpooling-and-transpose-convolution-831dc53687ce` (2023).

63. Zhao, J., Mathieu, M., Goroshin, R. & LeCun, Y. *Stacked What-Where Auto-encoders* arXiv.org. 1506.02351v8 (2015).
64. Ronneberger, O., Fischer, P. & Brox, T. U-Net: Convolutional Networks for Biomedical Image Segmentation. *Medical Image Computing and Computer-Assisted Intervention (MICCAI)* **9351,** 234–241 (2015).
65. Sharma, S., Sharma, S. & Athaiya, A. ACTIVATION FUNCTIONS IN NEURAL NETWORKS. *International Journal of Engineering Applied Sciences and Technology* **04,** 310–316 (2020).
66. Vu, L. & Nguyen, Q. U. *An Ensemble of Activation Functions in AutoEncoder Applied to IoT Anomaly Detection* in *2019 6th NAFOSTED Conference on Information and Computer Science (NICS)* (eds Bao, V. N. Q., Quang, P. M. & van Hoa, H.) (IEEE, 2019), 534–539. ISBN: 978-1-7281-5163-2.
67. Nwankpa, C., Ijomah, W., Gachagan, A. & Marshall, S. *Activation Functions: Comparison of trends in Practice and Research for Deep Learning* arXiv.org. 1811.03378v1 (2018).
68. Chakraborty & Arunava. Derivative of the Sigmoid function - Towards Data Science. *Towards Data Science* (2018).
69. Z, P. *Derivative of Tanh Function* `https://blogs.cuit.columbia.edu/zp2130/derivative_of_tanh_function/` (2022).
70. Wang, Y., Li, Y., Song, Y. & Rong, X. The Influence of the Activation Function in a Convolution Neural Network Model of Facial Expression Recognition. *Applied Sciences* **10,** 1897 (2020).
71. Fukushima, K. Cognitron: a self-organizing multilayered neural network. *Biological Cybernetics* **20,** 121–136 (1975).
72. Nair, V. & Hinton, G. E. Rectified Linear Units Improve Restricted Boltzmann Machines (2010).
73. Dugas, C., Bengio, Y., Bélisle, F., Nadeau, C. & Garcia, R. Incorporating Second-Order Functional Knowledge for Better Option Pricing. *Advances in Neural Information Processing Systems* **13** (2000).
74. L. Maas, A., Hannun, A. Y. & Ng, A. Y. *Rectifier nonlinearities improve neural network acoustic models* (2013).
75. He, K., Zhang, X., Ren, S. & Sun, J. *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification* arXiv.org. 1502.01852v1 (2015).
76. Clevert, D.-A., Unterthiner, T. & Hochreiter, S. *Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)* arXiv.org. 1511.07289v5 (2015).
77. Trottier, L., Giguère, P. & Chaib-draa, B. *Parametric Exponential Linear Unit for Deep Convolutional Neural Networks* arXiv.org. 1605.09332v4 (2016).
78. Klambauer, G., Unterthiner, T., Mayr, A. & Hochreiter, S. Self-Normalizing Neural Networks. *Advances in Neural Information Processing Systems 30 (NIPS* (2017).
79. Botchkarev, A. A New Typology Design of Performance Metrics to Measure Errors in Machine Learning Regression Algorithms. *Interdisciplinary Journal of Information, Knowledge, and Management* **14,** 045–076 (2019).

80. Janocha, K. & Czarnecki, W. M. *On Loss Functions for Deep Neural Networks in Classification* arXiv.org. 1702.05659v1 (2017).
81. Parmar, R. Common Loss functions in machine learning - Towards Data Science. *Towards Data Science* (2018).
82. Ruder, S. *An overview of gradient descent optimization algorithms* arXiv.org. 1609.04747v2 (2016).
83. Qian, N. On the momentum term in gradient descent learning algorithms. *Neural Networks* **12,** 145–151 (1999).
84. Nesterov, Y. A method for unconstrained convex minimization problem with the rate of convergence o(1/k squared) (1983).
85. Duchi, J., Hazan, E. & Singer, Y. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *Journal of Machine Learning Research* **12,** 2121–2159 (2011).
86. Zeiler, M. D. *ADADELTA: An Adaptive Learning Rate Method* arXiv.org. 1212.5701v1 (2012).
87. Hinton, G. *Neural Networks for Machine Learning: Lecture 6e* tech. rep. ().
88. Kingma, D. P. & Ba, J. *Adam: A Method for Stochastic Optimization* arXiv.org. 1412.6980v9 (San Diego, 2017).
89. Dozat, T. *Incorporating Nesterov Momentum into Adam* in *Proceedings of the 4th International Conference on Learning Representations* (2016), 1–4.
90. Kukačka, J., Golkov, V. & Cremers, D. *Regularization for Deep Learning: A Taxonomy* arXiv.org. 1710.10686v1 (2017).
91. Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I. & Salakhutdinov, R. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research* **15,** 1929–1958 (2014).
92. Liu, L. *Encyclopedia of database systems* (Springer, New York, NY, 2020).
93. Haar, A. Zur Theorie der orthogonalen Funktionensysteme. *Mathematische Annalen* **69,** 331–371 (1910).
94. Daubechies, I. The wavelet transform, time-frequency localization and signal analysis. *IEEE Transactions on Information Theory* **36,** 961–1005 (1990).
95. Graps, A. An introduction to wavelets. *IEEE Computational Science and Engineering* **2,** 50–61 (1995).
96. Bergh, J., Ekstedt, F. & Lindberg, M. *Wavelets mit Anwendungen in Signal- und Bildbearbeitung* (Springer, Berlin and Heidelberg, 2007).
97. Aggarwal, C. C., Hinneburg, A. & Keim, D. A. in *ICDT '01: Proceedings of the 8th International Conference on Database Theory* (ed van den Bussche, J.) 420–434 (Springer, 2001).
98. Mirkes, E. M., Allohibi, J. & Gorban, A. Fractional Norms and Quasinorms Do Not Help to Overcome the Curse of Dimensionality. *Entropy (Basel, Switzerland)* **22** (2020).

99. Glorot, X. & Bengio, Y. Understanding the difficulty of training deep feedforward neural networks. *Journal of Machine Learning Research - Proceedings Track* **9,** 249–256 (2010).

100. Khan, A., Sohail, A., Zahoora, U. & Qureshi, A. S. A survey of the recent architectures of deep convolutional neural networks. *Artificial Intelligence Review* **53,** 5455–5516 (2020).

101. *Convolutional Variational Autoencoder* `https://www.tensorflow.org/tutorials/generative/cvae` (2023).

102. ADEM, K. & KILICARSLAN, S. *Performance Analysis of Optimization Algorithms on Stacked Autoencoder* in *3rd International Symposium on Multidisciplinary Studies and Innovative Technologies* (IEEE, 2019), 1–4. ISBN: 978-1-7281-3789-6.

103. Xiao, H., Rasul, K. & Vollgraf, R. *Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms* arXiv.org. 1708.07747v2 (2017).

104. Loic Matthey, Irina Higgins, Demis Hassabis & Alexander Lerchner. *dSprites: Disentanglement testing Sprites dataset* 2017.

105. Han, S. *et al. Optimizing Filter Size in Convolutional Neural Networks for Facial Action Unit Recognition* arXiv.org. 1707.08630v2 (2017-07-26).

106. Zeiler, M. D. & Fergus, R. *Visualizing and Understanding Convolutional Networks* arXiv.org. 1311.2901v3 (2013-11-12).

Declaration of Academic Integrity / Eidesstattliche Erklärung

Ich erkläre, dass ich die vorliegende Arbeit selbständig, ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel verfasst habe und dass alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet sind. Mit der aktuell geltenden Fassung der Satzung der Universität Passau zur Sicherung guter wissenschaftlicher Praxis und für den Umgang mit wissenschaftlichem Fehlverhalten bin ich vertraut. Ich erkläre mich einverstanden mit einer Überprüfung der Arbeit unter Zuhilfenahme von Dienstleistungen Dritter (z.B. Anti-Plagiatssoftware) zur Gewährleistung der einwandfreien Kennzeichnung übernommener Ausführungen ohne Verletzung geistigen Eigentums an einem von anderen geschaffenen urheberrechtlich geschützten Werk oder von anderen stammenden wesentlichen wissenschaftlichen Erkenntnissen, Hypothesen, Lehren oder Forschungsansätzen.

Passau, 21. Mai 2023

____

Firstname Lastname

I hereby confirm that I have composed this scientific work independently without anybody else's assistance and utilising no sources or resources other than those specified. I certify that any content adopted literally or in substance has been properly identified. I have familiarised myself with the University of Passau's most recent Guidelines for Good Scientific Practice and Scientific Misconduct Ramifications. I declare my consent to the use of third-party services (e.g., anti-plagiarism software) for the examination of my work to verify the absence of impermissible representation of adopted content without adequate designation violating the intellectual property rights of others by claiming ownership of somebody else's work, scientific findings, hypotheses, teachings or research approaches.

Passau, 21. Mai 2023

____

Firstname Lastname