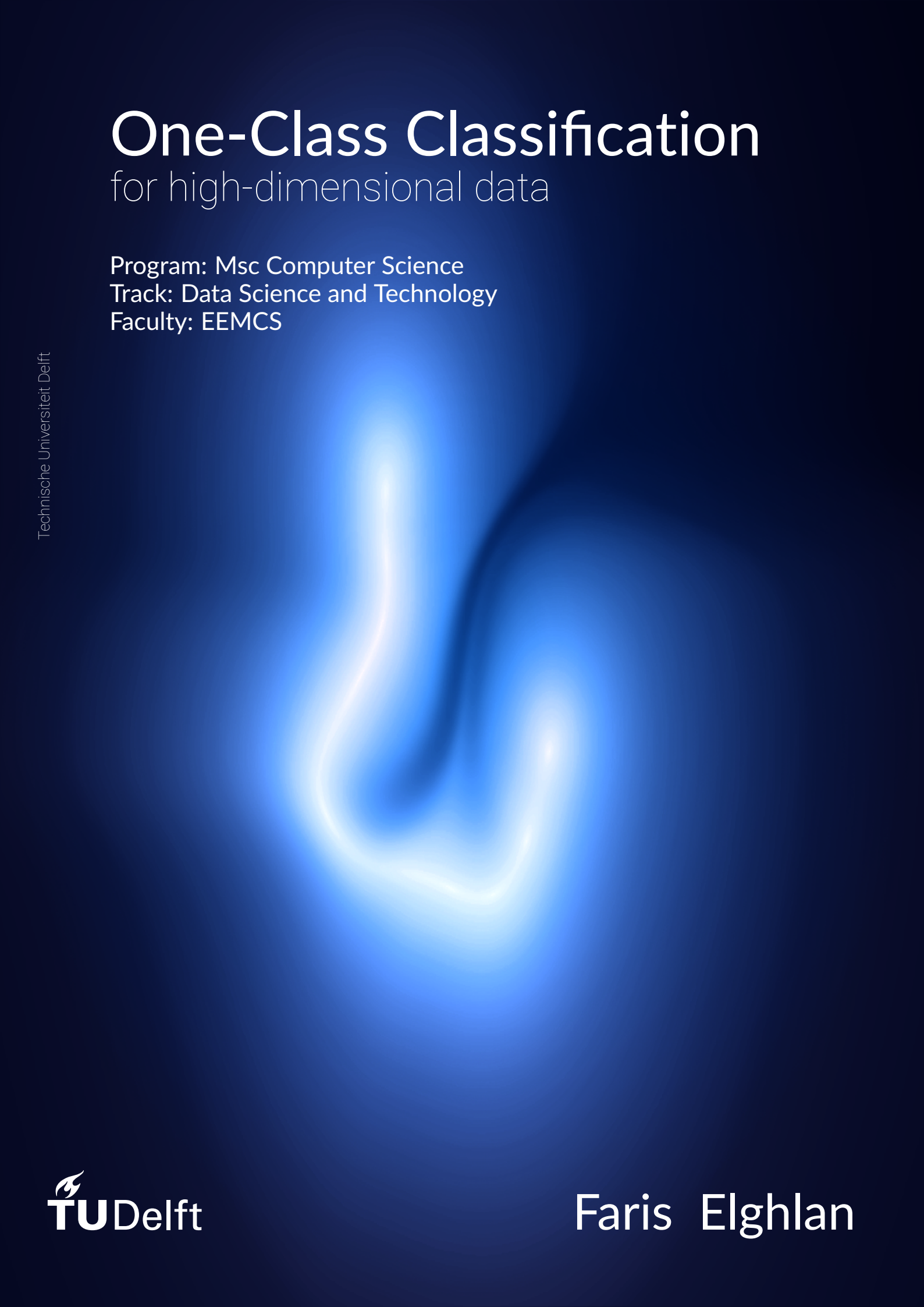
One-Class Classification
for high-dimensional data

Program: Msc Computer Science
Track: Data Science and Technology
Faculty: EEMCS

Technische Universiteit Delft



TU Delft

Faris Elghlan

One-Class Classification

for high-dimensional data

by

Faris Elghlan

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Tuesday August 27, 2019 at 14:00

Student number:	4341538	
Project duration:	November 19, 2018 – August 27, 2019	
Thesis committee:	Dr. D.M.J. Tax,	TU Delft, supervisor
	Prof. M.J.T. Reinders,	TU Delft
	Dr. M.M. de Weerdt,	TU Delft

The cover image represents the reconstruction error of a Wasserstein autoencoder that is trained on a manually created 2D datasets. A darker color corresponds to a larger reconstruction error. Adjustments in luminosity and black levels were manually performed to improve the visual appeal.

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Abstract

This M.Sc. thesis report investigates the application of one-class classification techniques to complex high-dimensional data. The aim of a one-class classifier is to separate target data from non-target data, but only a dataset containing target data is available for training. The issue with high-dimensional data is that it is difficult to perform density estimation due to the 'curse of dimensionality'. Most conventional method for one-class classification rely on density estimation.

This thesis focusses on the use of autoencoders and generative adversarial networks (GANs) for one-class classification problems involving image data. Autoencoders can learn encoding and decoding functions for samples from the target dataset. These encoding and decoding functions are, however, expected to not perform well for non-target samples, as they have never been seen during the training phase. This makes it possible to separate target and non-target data. For GANs, the discriminator is used to distinguish between target and non-target data.

Autoencoders and GANs are evaluated extensively in this report. Their behavior, desired parameters and strengths and weaknesses are evaluated by performing experiments. The main findings are that GANs do not perform well for one-class classification tasks, because of mode collapse and insufficient sampling of the non-target data. Even for extremely simple datasets these issues were observed. Autoencoders are shown to perform much better and behave according to the theoretical expectations.

Faris Elghlan
Delft, August 2019

Contents

Nomenclature

Mathematical objects

a	Scalar
$\mathbf{a}$	Vector
A	Set
$\mathbf{A}$	Matrix
$\mathcal{A}$	Space, typically $\mathbb{R}^n$

Mathematical symbols

$\mathbf{x}$	A sample from the train or test set
X	Train set, consisting of a finite number of samples drawn from the target distribution
T_{target}	Independent test set, consisting of samples drawn from the target distribution
$T_{outlier}$	Independent test set, consisting of samples *not* drawn from the target distribution
$\mathcal{X}$	Feature space
$\mathrm{P}_{\mathcal{X}}(\mathbf{x} \mid \omega_\ell)$	Distribution of class ω_ℓ in feature space $\mathcal{X}$
ω_T	Label of the target class
ω_O	Label of the outlier class
δ	Decision threshold
$\mathbf{I}$	Identity matrix
$\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$	Normal distribution with mean $\boldsymbol{\mu}$ and covariance matrix $\boldsymbol{\Sigma}$

Mathematical operators

$\lVert\cdot\rVert_2$	The ℓ_2-norm
$\sim$	Distributed as
$D_{KL}(\mathrm{A} \parallel \mathrm{B})$	The KL divergence between distributions A and B
$\mathrm{Diag}(\mathbf{x})$	Produces a diagonal matrix with the entries of vector $\mathbf{x}$ on its diagonal:

$$\begin{matrix} x_1 & 0 & \dots & 0 \\ 0 & x_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & x_n \end{matrix}$$

Generative Adversarial Network (GAN)

$\mathbf{z}$	Input of the generator
$\text{gen} : \mathcal{Z} \to \mathcal{X}$	Generator function
$\text{dis} : \mathcal{X} \to \mathbb{R}^1$	Discriminator function

Autoencoders

$\mathbf{z}$	Encoded sample: $\mathbf{z} = \text{enc}(\mathbf{x})$
$\text{enc} : \mathcal{X} \to \mathcal{Z}$	Encoder function
$\text{dec} : \mathcal{Z} \to \mathcal{X}$	Decoder function
$\text{rec\_err} : \mathcal{X} \to \mathbb{R}$	The reconstruction error for a given sample $\mathbf{x}$, defined as: $\text{rec\_err}(\mathbf{x}) = \lVert \mathbf{x} - \text{dec}(\text{enc}(\mathbf{x})) \rVert_2$
$\mathrm{P}_{\mathcal{Z}}(\mathbf{z} \mid \omega_\ell)$	Distribution of class ω_ℓ in the encoded domain: $\mathrm{P}_{\mathcal{Z}}(\mathbf{z} \mid \omega_\ell) = \mathrm{P}_{\mathcal{Z}}(\text{enc}(\mathbf{x}) \mid \omega_\ell)$ for $\mathbf{x} \sim \mathrm{P}_{\mathcal{X}}(\mathbf{x} \mid \omega_\ell)$
$\mathbf{o}_{enc}^{(i)}$	The output of layer i in the encoder network
$\mathbf{o}_{dec}^{(i)}$	The output of layer i in the decoder network
$\mathbf{n}_{enc}^{(i)}$	The number of neurons in layer i of the encoder network
$\mathbf{n}_{dec}^{(i)}$	The number of neurons in layer i of the encoder network

1 Introduction

In this M.Sc. thesis report, the subject of *one-class classification* involving *high-dimensional data* is discussed. One-class classification is defined as the act of separating target data from all dissimilar, or unlikely, data. As an example, one can think of the target data being apples. An ideal one-class classifier would accept all possible apple samples, while rejecting all other samples, such as a banana. In the literature, one-class classification is sometimes referred to as outlier, anomaly, or novelty detection.

In a one-class classification setting, usually only data belonging to the target class is available, or the non-target class is not properly sampled. Some practical examples of one-class classification problems are: detecting machine failures when only data of correctly functioning machines is available, detecting diseases given only data of healthy patients, detecting the presence of a human face in a picture, or detecting fraudulent bank transactions. In such settings, it is often expensive, time consuming, or even impossible to get a representative dataset of the non-target class. The non-target data could, for example, contain unknown objects (a patient with an new type of disease), or be too broad (all possible images that do not contain a human face).

The dimensionality of data is equal to the number of features, or attributed, that are measured for each sample. For images, the content of each pixel is typically encoded using RGB values, thus there are 3 features per pixel. Images can consist of thousands, or millions of pixels, thus resulting in very high-dimensional data. In this thesis, mainly images will be used. The discussed methods can also be applied to non-image data.

Most conventional methods for one-class classification rely on density estimation. To estimate the density of a target data distribution, sufficiently many samples are required in the space that is occupied by the data. The volume of this space increases exponentially with the number of dimensions, so the required number of samples for an accurate density estimation also increases exponentially with the number of dimensions. This is problematic when working with high-dimensional data and is often referred to as the 'Curse of Dimensionality'.

Another issue occurs when the target data lies in a lower-dimensional subspace. We suspect that this frequently occurs for image data. A very simple example is when a pixel near the border of an image is always black. When the target class lies in a subspace, the volume of the space that is occupied by the target data is equal to zero. According to the second axiom of probability, the probability that any event will occur should be equal to 1, so the integral over the probability density for this volume should equal 1. This causes the probability density to be infinite for the subspace in which the target data lies and zero everywhere else. In practice, there will always be some noise in the target dataset, so the probability density will not actually be infinite. In practice, the probability density will vary smoothly within the subspace, but rapidly approaches zero when moving perpendicularly from it. This causes a difference in scale and makes the probability density function difficult to learn. This issue could be remedied by projecting the target data into a lower dimensional space by using a feature selection technique, but this adds the task of finding a good feature selection algorithm for the problem.

In this thesis, methods that make use of artificial neural networks (ANNs) are considered. During the last decade, research has shown that ANNs are successful at learning concepts from data, even when the data is high-dimensional. Especially in the field of computer vision, ANNs that contain convolutional filters have been applied with great success to problems involving high-dimensional data [25, 30, 33]. This makes it reasonable to assume that ANNs can also be successfully applied to one-class classification problems.

1.1. One-Class Classification Problem Formalization

Let us consider a target dataset $X = \{\mathbf{x}^{(i)} \in \mathbb{R}^n\}$ and an independent test set $T_{target} = \{\mathbf{x}^{(i)} \in \mathbb{R}^n\}$ that are both sampled from the distribution $\mathrm{P}_{\mathcal{X}}(\mathbf{x} \mid \omega_T)$, and a test set consisting of outliers samples $T_{outlier} = \{\mathbf{x}^{(i)} \in \mathbb{R}^n\}_{i=1}^{L}$ that is sampled from the distribution $\mathrm{P}_{\mathcal{X}}(\mathbf{x} \mid \omega_O)$. Here ω_T and ω_O denote the target and outlier class labels respectively, and $\mathcal{X}$ denotes the feature space.

A one-class classifier should determine if a sample $\mathbf{x} \in \mathbb{R}^n$ belongs to the target distribution: $\mathrm{P}_{\mathcal{X}}(\mathbf{x} \mid \omega_T) > \delta > 0$. Thus, the classifier should determine if the probability density of sample $\mathbf{x}$ for the target distribution is larger than the decision threshold δ. In practice, a one-class classifier will have to learn to do this by using a finite training dataset X consisting of target samples. To make this problem feasible, we assume that the target class and outlier class do not overlap in Assumption 1.1.

Assumption 1.1. If a sample $\mathbf{x}$ belongs to the target class, then it cannot be sampled from the outlier distribution:

$$\mathrm{P}_{\mathcal{X}}(\mathbf{x} \mid \omega_T) > \delta > 0 \implies \mathrm{P}_{\mathcal{X}}(\mathbf{x} \mid \omega_O) = 0$$

1.2. Autoencoders and GANs

Autoencoders (AE) and Generative Adversarial Networks (GAN) will be investigated in this report and applied to one-class classification problems. Both of these techniques are unsupervised, meaning that they do not require labelled training data.

Autoencoders learn to encoding (compression) and decoding (decompression) target data. The intuition is that the compression and decompression will only work well for the target data, such that it is possible to distinguish non-target samples.

GANs learn to generate new samples that are similar to the target data and simultaneously learn to discriminate between 'real' samples from the target class and 'fake' generated samples. The discriminator is analyzed to determine if it can be used to separate target samples from non-target samples.

1.3. Research Question

Research question: *Can one-class classification problems involving high-dimensional image data be solved using artificial neural networks and which properties of such neural networks are the most important?*

Sub-Questions:

1. How do conventional one-class classification methods perform on high-dimensional data?
2. Can modern unsupervised neural network techniques, such as generative adversarial networks (GANs) and autoencoders (AE), be applied effectively to (high-dimensional) one-class classification problems?
   (a) What is the desired complexity of the network?
      - Is the depth or the width of the network more important?
      - Does the performance improve when using a more complex network?
      - Which parts of the network are most critical when varying the network complexity? For example, the encoder or the decoder of an AE.
   (b) Does the value of the encoded vector $\mathbf{z}$ of an AE have any interpretable meaning and how does the dimensionality of $\mathbf{z}$ influence the classification performance?
   (c) Are there any parameters that are difficult to optimize?
   (d) What are the strengths/weaknesses and what types of errors are to be expected?
   (e) Are these methods resilient against outliers, or noise in the training data?
   (f) How does the performance degrade when reducing the amount of training data?

1.4. Outline

The remainder of this thesis is structured as follows:

Chapter 2 Relevant topics from the literature are discussed, in order to provide sufficient background knowledge for the rest of the report.

Chapter 3 The one-class classification methods that are evaluated in the remained of this thesis are discussed and theoretically justified.

Chapter 4 The method of benchmarking and the experimental setup are explained.

Chapter 5 Two conventional methods for one-class classification are tested on high-dimensional datasets to justify the need for an alternative approach.

Chapter 6 GANs are analyzed to determine if they could be applied to one-class classification problems.

Chapter 7 Autoencoders are analyzed to determine if they could be applied to one-class classification problems.

Autoencoders were found to outperform GANs for one-class classification purposes, so only autoencoders are further investigated in the following chapters

Chapter 8 Several parameters and network variants are tested to determine which properties are the most important. The difficulty of optimizing each parameter is analyzed as well.

Chapter 9 A case study is performed, where digits 3 and 4 from the MNIST dataset are used as target data. This allows for a more detailed analysis, although the results are not shown to hold in general.

Chapter 10 Potential problems and limitations of the presented method are experimentally analyzed. The experiments mostly involve decreasing the quality, or amount of training data.

Chapter 11 Non-exhaustively tested methods that could be interesting for further research are discussed.

Chapter 12 The thesis is concluded and the most interesting findings are discussed. The limitation of the performed research and potential further research are also mentioned.

2
Background

Relevant topics from the literature are discussed, in order to provide sufficient background knowledge for the rest of the report.

2.1. Conventional One-Class classification methods

One-class classifiers can be distinguished into two categories [39]. The first category consists of density estimators, such as Gaussian mixture models and kernel density estimation [11]. These statistical methods attempt to estimate the probability density of the target class. Samples from low probability density regions are classified as outliers. These methods will generally not perform well for high-dimensional problems, since the number of training samples required to achieve accurate density estimates increases exponentially with the number of dimensions, due to the 'Curse of Dimensionality'.

The second category of conventional one-class classification techniques consists of distance-based methods. Here a model is fit to the target data and a classification is made based on the distance to this model. Clustering methods, such as k-means, could be used to model the target data. Other popular methods include nearest neighbor [15], support vector domain description (SVDD) [38] and autoencoders [10].

2.1.1. Nearest Neighbor

The Nearest Neighbor Distance (NN-d) method can be used as density estimator [6]. To evaluate if a sample lies in a high or low probability density area, the distance between the sample and the closest sample from the training dataset has to be calculated. The intuition is that a sample lies in a high probability density area when the nearest neighbor is close, while it lies in a low probability density area when the nearest neighbor lies far away. This is expected to hold, because when a sample lies in a higher probability density area it is more likely that a training sample lies close to it.

A one-class classifier can easily be constructed by thresholding the distance to the nearest neighbor. Thus, only samples that are expected to lie in high probability density areas will be accepted as target data.

2.1.2. Mixture of Gaussians

The mixture of Gaussians method assumes that the distribution of the data can be approximated by combining several Gaussian distributions. Thus, a linear combination of several simple Gaussian distributions are used to model the more complex distribution of the data. An example Gaussian mixture model and its gaussian components are shown in Figure 2.1.

A Gaussian mixture model can be optimized by using the Expectation-Maximization (EM) algorithm [24]. This algorithm iteratively maximizes the likelihood of each gaussian component. By doing this until convergence, a good estimate of the optimal parameters of the model can be found. Estimation is required, since there is often no closed-form solution to maximizing the likelihood of a Gaussian mixture model.

After fitting a Gaussian mixture model to the target dataset, outliers can be separated from target data by computing the probability density of a test sample for the Gaussian mixture model and thresholding it. The high probability samples will be classified as target data, while the low probability samples are classified as outliers.

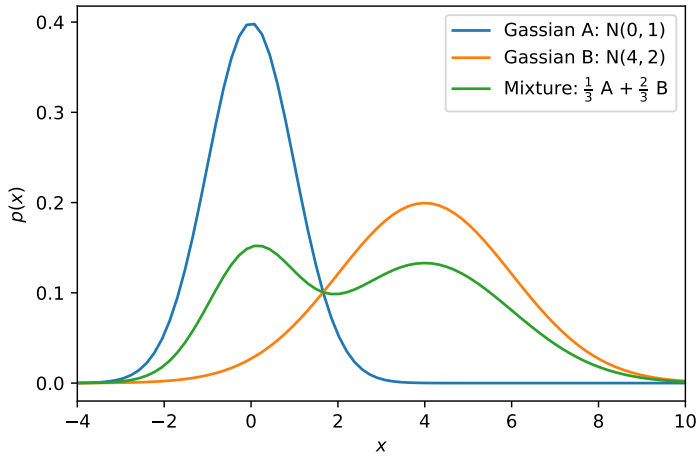


Figure 2.1: The probability density function of a Gaussian mixture model and its two Gaussian components A and B.

2.1.3. Kernel Density Estimation

In this report, Kernel density estimation KDE [27] is used for visualizing an approximation of the PDF of generated data. KDE estimates the PDF of a dataset by creating a kernel function for each sample in the dataset. This kernel function could, for example, be a uniform, triangular, or normal distribution. The PDF is estimated by adding together all of the kernel functions and normalizing, such that the area becomes equal to 1.

An important parameter for KDE is the bandwidth of the kernel function. When setting it too wide, the results are smoothed too much, while setting it too narrow will cause a noisy estimate. The ideal bandwidth depends on how many samples are available. In general, the more samples are available, the smaller the bandwidth can be set without getting noisy results. Several kernel density estimates that use varying bandwidths are shown in Figure 2.2. The KDE that uses a bandwidth equal to 1.0 is smoothed too much, while the KDE that uses a bandwidth equal to 0.1 is noisy. The KDE with bandwidth equal to 0.4 produces a PDF that is approximately equal to the true PDF of $\mathcal{N}(0,1)$

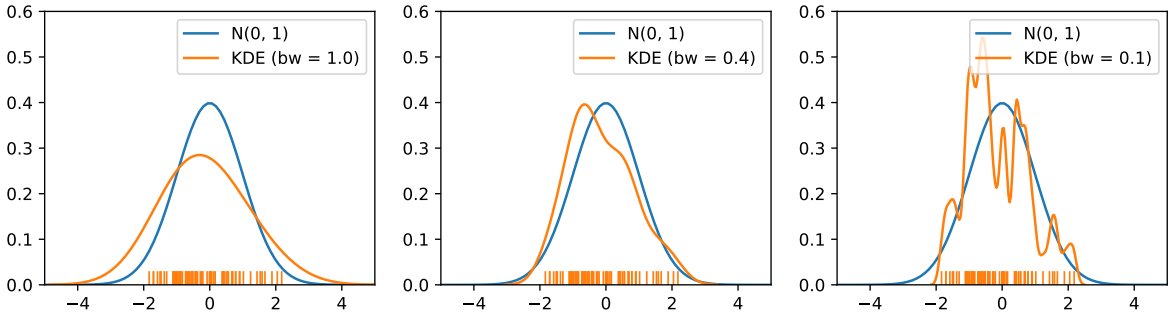


Figure 2.2: Three kernel density estimates, that use varying bandwidths, for a dataset consisting of 100 samples generated from $\mathcal{N}(0,1)$.

2.2. Generative Adversarial Networks

A Generative Adversarial Network (GAN) contains two parts: a generator and a discriminator [9]. The discriminator attempts to learn to discriminate between (real) samples from the target dataset and (fake) samples that were created by the generator. The generator attempts to 'fool' the discriminator, by generating more realistic samples. Eventually, when the network converges, the discriminator should not be able to discriminate between generated and real samples anymore. The generator should thus generate samples that are indistinguishable from the target class.

More formally, define gen : $\mathcal{Z} \to \mathcal{X}$ to be the generator function and dis : $\mathcal{X} \to \mathbb{R}$ to be the discriminator function. The generator and discriminator are both implemented as an artificial neural network, although

any mathematical function could be used. The generator and discriminator networks have trainable parameters $\boldsymbol{\theta}_{\text{gen}}$ and $\boldsymbol{\theta}_{\text{dis}}$, respectively. The generator and discriminator play the minimax game defined in Equation 2.1 in order to learn to generate better samples.

$$\min_{\boldsymbol{\theta}_{\text{gen}}} \max_{\boldsymbol{\theta}_{\text{dis}}} \mathbb{E}_{\mathbf{x}\sim \mathrm{P}_{\mathcal{X}}(\mathbf{x}|\omega_T)} [\log(\text{dis}(\mathbf{x}))] + \mathbb{E}_{\mathbf{z}\sim \mathrm{P}_{\mathcal{Z}}(z)} [\log(1 - \text{dis}(\text{gen}(\mathbf{z})))] \tag{2.1}$$

Here, $\mathrm{P}_{\mathcal{Z}}(z)$ is a prior distribution from which noise is generated that is used as input for the generator. Usually $\mathrm{P}_{\mathcal{Z}}(z) = \mathcal{N}(\mathbf{z}|\mathbf{0},\mathbf{I})$ is used.

The generator does not influence the first term in Equation 2.1, thus it can only minimize the term $\log(1 - \text{dis}(\text{gen}(\mathbf{z})))$. This is equivalent to maximizing $\text{dis}(\text{gen}(\mathbf{z}))$, thus the generator attempts to produce samples that are classified as 'real' by the discriminator.

The discriminator attempts to maximize the expression, so it has the opposite objective compared to the generator. The discriminator thus attempts to classify generated samples as 'fake'. Additionally, the discriminator maximizes the first term $\log \text{dis}(\mathbf{x})$ from Equation 2.1, which causes samples from the target dataset to be classified as 'real'.

A graphical representation of a typical GAN is shown in Figure 2.3. In the figure, a sample $\mathbf{z}$ is given as input to the generator, which generates a sample $\mathbf{x}'$. The discriminator attempts to distinguish between the generated samples $\mathbf{x}'$ and the 'real' samples from the target class $vecx$. The generator and discriminator are both depicted as networks consisting of multiple layers.

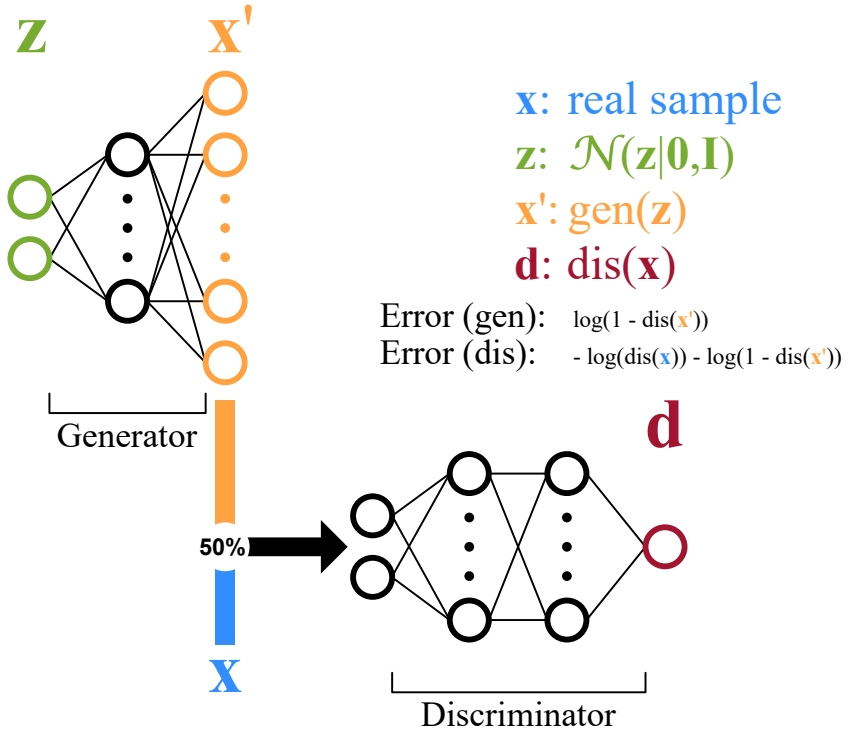


Figure 2.3: Graphical illustration of a typical GAN network.

2.3. Information Theory

The discussion of (variational) autoencoders requires prior knowledge about Information Entropy and the KL Divergence, which are explained in this section.

2.3.1. Information Entropy

Information Entropy is a measure of the average amount of information (or randomness) that is produced by sampling a value from a probability distribution [41]. This is equivalent to the average number of bits required to encode a sample from the distribution, given that an optimal encoding is used [37]. The Information Entropy is defined by Equation 2.2, for the discrete distribution $\mathrm{P}_{\mathcal{X}}(x)$.

$$\mathrm{H}(\mathrm{P}_{\mathcal{X}}) = -\sum_{x\in\mathcal{X}} \mathrm{P}(x) \log(\mathrm{P}(x)) \tag{2.2}$$

The term $\log(\mathrm{P}(x))$ represents the amount of information contained in a single outcome x. For an optimal encoding of x, $\log_2(\mathrm{P}(x))$ bits will be required. The smaller the probability that event x occurs, the more information it provides and the more bits are necessary to encode it. The term $p(x)$ represent the likelihood of sample x to occur and is used to average the amount of information contained in a sample from $\mathrm{P}_{\mathcal{X}}(x)$.

The Information Entropy is maximized when all outcomes are equiprobable. This is the case for a uniform distribution.

2.3.2. KL Divergence

The Kullback Leibler Divergence (KL Divergence) is a measure of the difference between two distributions and is closely related to the Information Entropy [41]. The KL divergence between the discrete distributions $\mathrm{P}_{\mathcal{X}}(x)$ and $\mathrm{Q}_{\mathcal{X}}(x)$ is defined in Equation 2.3.

$$\begin{aligned} D_{KL}(\mathrm{P}_{\mathcal{X}}(x) \parallel \mathrm{Q}_{\mathcal{X}}(x)) &= -\sum_{x \in \mathcal{X}} \mathrm{P}(x) \log(\mathrm{Q}(x)) - H(\mathrm{P}_{\mathcal{X}}(x)) \\ &= \sum_{x \in \mathcal{X}} \mathrm{P}(x) \log\left(\frac{\mathrm{P}(x)}{\mathrm{Q}(x)}\right) \end{aligned} \tag{2.3}$$

The KL divergence measures the amount of additional information that is required when encoding events from the distribution $\mathrm{P}_{\mathcal{X}}(x)$, using the optimal encoding of distribution $\mathrm{Q}_{\mathcal{X}}(x)$ instead of using the optimal encoding of distribution $\mathrm{P}_{\mathcal{X}}(x)$.

2.4. Autoencoders

Autoencoders are one of the methods that can be used to fit a model to a target dataset and have been applied to one-class classification problems before [13, 21, 22]. An autoencoder contains two parts, that can be implemented as artificial neural networks. The first part is a function that serves as the encoder $\mathrm{enc} : \mathcal{X} \to \mathcal{Z}$ and the second is the decoder $\mathrm{dec} : \mathcal{Z} \to \mathcal{X}$. Here $\mathcal{X}$ is the feature space and $\mathcal{Z}$ is the encoded space, which typically has a much lower number of dimensions than $\mathcal{X}$. The goal of an autoencoder is to learn the identity function $\mathrm{dec}(\mathrm{enc}(x)) = x$ for the target data. When the dimensionality of $\mathcal{Z}$ is smaller than that of $\mathcal{X}$, the encoder is a compression function, while the decoder is a decompression function.

A graphical representation of a typical autoencoder is shown in Figure 2.4. In the figure, a sample $\mathbf{x}$ is given as input and encoded as $\mathbf{z}$ by the encoder. The decoder maps the sample $\mathbf{z}$ to a reconstructed sample $\mathbf{x}'$. The encoder and decoder are both depicted as networks consisting of multiple layers. The (reconstruction) error of the network is defined in Equation 2.4 and measures the difference between the input and output of the autoencoder.

$$\mathrm{rec\_err}(\mathbf{x}) = \|\mathbf{x} - \mathrm{dec}(\mathrm{enc}(\mathbf{x}))\|_2 = \left\|\mathbf{x} - \mathbf{x}'\right\|_2 \tag{2.4}$$

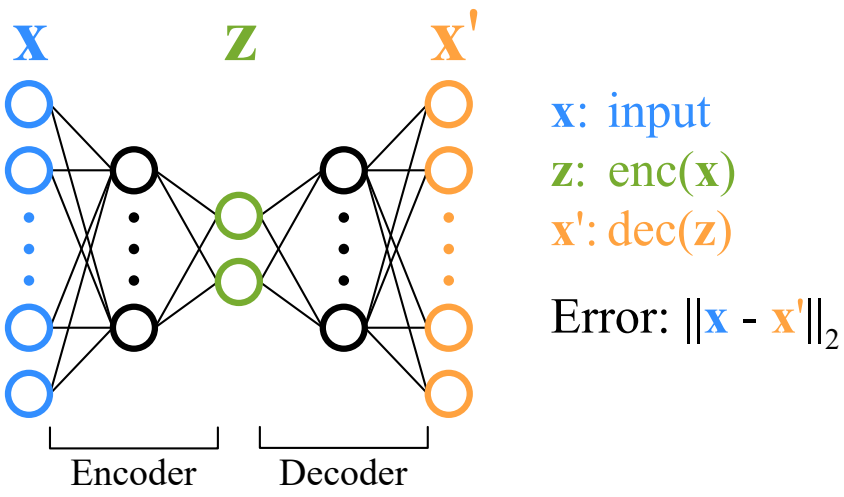


Figure 2.4: Graphical illustration of a typical autoencoder network.

2.4.1. Variational Autoencoders

A variational autoencoder network (VAE) is similar to a regular autoencoder, but the theoretical foundation differs significantly. An illustration of a variational autoencoder network is given in Figure 2.5. The difference compared to Figure 2.4 is that the encoded sample $\mathbf{z}$ no longer has a fixed value, but is sampled from a distribution $\mathbf{z} \sim \mathcal{N}(\boldsymbol{\mu}, \text{Diag}(\boldsymbol{\sigma}^2))$, where μ and σ are produced by the encoder. The remainder of this section discusses the most relevant theoretical details of variations autoencoders, but may be skipped by readers that are mostly interested in practical applications and results.

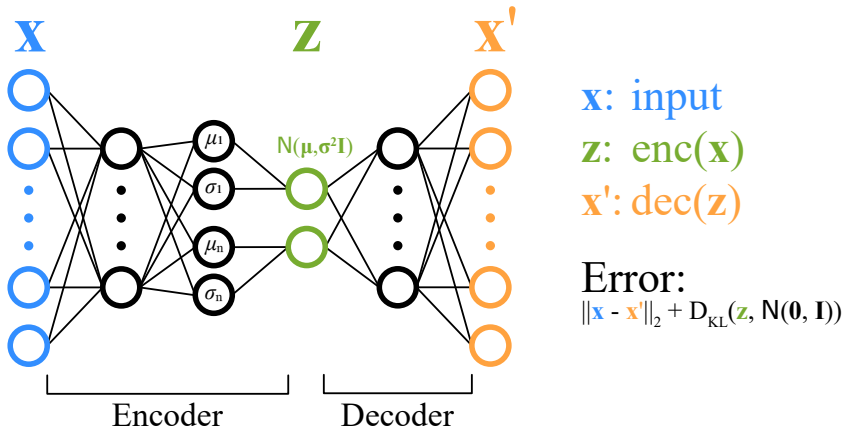


Figure 2.5: Graphical illustration of a typical variational autoencoder network. Here $\mathbf{z}$ is sampled from a normal distribution with mean and standard deviations as computed by the encoder network.

Variational autoencoders solve the following problem [14]: given some dataset X consisting of i.i.d. samples generated from some target distribution, how can more samples be generated from this distribution?

The method is based on the assumption that samples in X are distributed according to some distribution $p_{\theta^*}(\mathbf{x} \mid \mathbf{z})$, which is dependent on a latent variable $\mathbf{z} \sim p_{\theta^*}(\mathbf{z})$ and some hidden parameters θ^*. Here $\mathbf{z}$ corresponds to the values that are produced by the encoder. Since the optimal parameters θ^* and the model are unknown, they must be learned from dataset X. An artificial neural network is used as model and the parameters are optimized through maximum likelihood estimation (MLE). MLE optimizes parameters θ, such that the likelihood of the samples in dataset X is maximized, as is depicted in Equation 2.5.

$$\theta^* = \underset{\theta \in \Theta}{\arg\max} \sum_{x \in X} p_\theta(x) \tag{2.5}$$

When samples in X are likely to occur, it is assumed that samples similar to those in X are also likely to occur. Thus, if dataset X contains enough samples and provides a good representation of the underlying distribution, then the trained model should generate samples that are similar to the target distribution.

Equation 2.5 is, however, not trivial to optimize. For the full derivation of how to optimize this, we refer the reader to the original paper [14], or a simplified explanation in [8]. The resulting objective function that must be optimized to maximize the MLE is given in Equation 2.6, where θ and ϕ are the parameters of the model.

$$\begin{aligned} &\underset{\theta,\phi}{\arg\min}\ D_{KL}(q_\phi(\mathbf{z} \mid \mathbf{x}) \parallel p_\theta(\mathbf{z})) \\ &\underset{\theta,\phi}{\arg\max}\ \mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{x} \mid \mathbf{z})] \end{aligned} \tag{2.6}$$

The equation contains two parts. The first part minimizes the KL-divergence between the 'encoder' distribution $q_\phi(\mathbf{z} \mid \mathbf{x})$ and the assumed true distribution $p_\theta(\mathbf{z})$. The true distribution $p_\theta(\mathbf{z})$ is usually set equal to the normal distribution $\mathcal{N}(\mathbf{0}, \mathbf{I})$. The second part corresponds to the MLE, given that the variable $\mathbf{z}$ is distributed according to the 'encoder' distribution $q_\phi(\mathbf{z} \mid \mathbf{x})$. Both parts can be combined by negating the second part and adding them together, resulting in a single minimization objective.

In Equation 2.6, the term $q_\phi(\mathbf{z} \mid \mathbf{x})$ corresponds to the encoder of the variational autoencoder and the term $p_\theta(\mathbf{x} \mid \mathbf{z})$ corresponds to the decoder. The expectation in Equation 2.6 can be approximated by using stochastic gradient descent and maximizing $\log p_\theta(\mathbf{x} \mid \mathbf{z})$ is done by minimizing the reconstruction error of

each sample $\mathbf{x}$. The idea is that a small reconstruction error means the decoder is able to produce something similar to $\mathbf{x}$, thus the likelihood of producing $\mathbf{x}$ should be high as well.

It is possible to weight the two parts in Equation 2.6 differently, which is explored further in Experiment 8.3. The weight of the KL-divergence term could be increased to make the network learn an 'encoder' distribution that is more similar to the true distribution. The other possibility is to decrease the weight of the KL-divergence term, resulting in the 'encoder' distribution being less restricted, but the network can focus more on improving the MLE term, resulting in a better reconstruction performance. Thus, a tradeoff can be made between the quality of the 'encoder' distribution and the reconstruction error.

To let the encoder generate $\mathbf{z}$ according to a distribution, it does not produce a single value, but instead produces a mean $\boldsymbol{\mu}$ and standard deviation $\boldsymbol{\sigma}$ and samples a value from $\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\sigma}^2 I)$. The final loss function that should be minimized for a single sample $\mathbf{x}^{(i)}$ is shown in Equation 2.7. Here J is the number of dimensions of $\mathbf{z}$ and $\mu^{(i)}$ and $\sigma^{(i)}$ are the mean and standard deviation that are produced by encoding $\mathbf{x}^{(i)}$.

$$\mathcal{L}(\theta, \phi; \mathbf{x}^{(i)}) = \frac{1}{2} \sum_{j=1}^{J} (1 + \log((\sigma_j^{(i)})^2) - (\mu_j^{(i)})^2 - (\sigma_j^{(i)})^2) + \text{rec\_err}(\mathbf{x}^{(i)}) \tag{2.7}$$

2.4.2. Wasserstein Autoencoders

Wasserstein Autoencoders (WAE) [40] attempt to minimize the Wasserstein distance between the distribution of encoded target data $P_{\mathcal{Z}}(\mathbf{z} \mid \omega_T)$ and a prior distribution $Q_{\mathcal{Z}}(\mathbf{z} \mid \omega_T)$. The Wasserstein distance is a measure of the difference between two distributions. WAEs thus attempt to achieve a similar result as VAEs, but use a different error metric.

The reason that WAEs might perform better than VAEs, is that VAEs force the encoding of each individual sample to match $Q_{\mathcal{Z}}(\mathbf{z} \mid \omega_T)$. When all samples are encoded identically, it will not be possible to decode each sample correctly, so VAEs must find a tradeoff between this objective and the objective to minimize the reconstruction error. The distribution of the encoding of different target samples must, however, partially overlap and this causes issues when training the decoder. WAEs on the other hand, do not force the encoding of each sample to match some distribution. Instead, WAEs only force the distribution of all encoded target samples to match $Q_{\mathcal{Z}}(\mathbf{z} \mid \omega_T)$. Thus, a WAE is able to uniquely encode each target sample, without any overlap. This is further discussed in [40], especially Figure 1 in this paper provides an intuitive explanation of this problem.

In this report, the WAE-GAN method from [40] is used. This method makes use of a discriminator to minimize the Wasserstein distance. An illustration of such a WAE is given in Figure 2.6. The discriminator is trained similarly as the discriminator of a GAN. The discriminator is given a sample that is either created by encoding target data, or sampled the prior distribution and has to discriminate between the two. The term $-\lambda \cdot \log(\text{dis}(\mathbf{z}))$ is added to the error of the autoencoder, such that it tries to 'fool' the discriminator. This results in the distribution of encoded target data becoming similar to the prior distribution $Q_{\mathcal{Z}}(\mathbf{z} \mid \omega_T)$. λ is a parameter that can be increased to make the distribution of encoded target data match the prior distribution more closely, at the cost of the reconstruction error becoming worse.

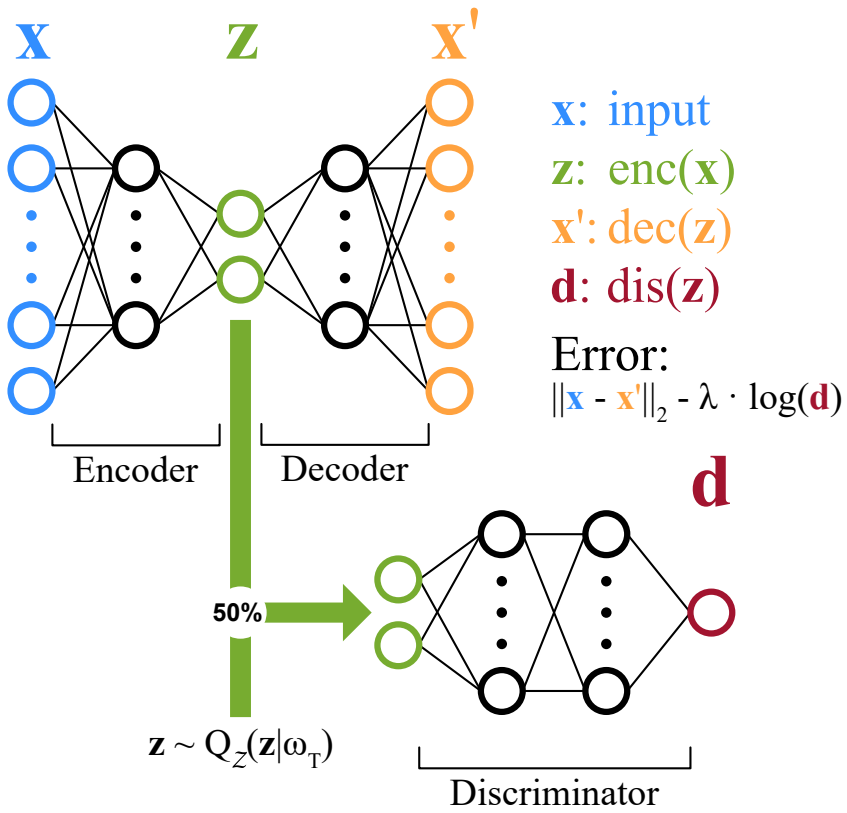


Figure 2.6: Graphical illustration of a typical Wasserstein autoencoder network. A discriminator is used to force the distribution of encoded target data to match the prior distribution $Q_{\mathcal{Z}}(\mathbf{z} \mid \omega_T)$.

3

Methods & Application to One-Class Classification

The one-class classification methods that are evaluated in the remained of this thesis are discussed and theoretically justified.

3.1. Image Data in Subspaces

We expect that for most high-dimensional one-class classification problems, the target data lies in subspaces. This means that when the data lies in the space $\mathbb{R}^n$, the target class occupies a space that can be described using only d degrees of freedom, where $d < n$. This causes the volume of the target class to be zero.

One example where this has been proven to be the case, is for images of a convex Lambertian object under different isotropic lighting conditions. Under these conditions, it has been shown that the reflectance function can be represented in approximately a 9D subspace [2]. Thus, the object can be represented under many different lighting conditions using a low number of dimensions.

The success in the field of subspace clustering [26] also supports the claim that high-dimensional target data often lies in subspaces. Subspace clustering methods attempt to find groups of data samples (clusters) that lie closely together in some subspace. This technique is also often applied to image data, as is discussed in [1].

One method to make density estimation feasible for problems where the data lies in a subspace, is to first apply a feature selection technique. Feature selection reduces the number of features by selecting only the most useful ones. This is, however, no trivial task and discarding the wrong features could cause a performance loss. Principle component analysis (PCA) [12] is a method that is often used for feature selection. PCA can be used to select the directions in which the variance of the training data is the highest. For one-class classification, this might, however, not select only the most useful features. For images of objects, there could be high variation in the background, while the background might not be important at all. Clusters of data that lie in different subspaces, or data that lies in *non*linear subspaces, might also make feature selection difficult.

Neural networks can learn a nonlinear feature selection procedure by reducing the size of the output of each succeeding layer in the network, while at the same time optimizing some objective. The applicability of such neural networks to one-class classification problems is further investigated in this thesis.

3.2. Applicability of Autoencoders to One-Class Classification

The intuition is that, for outliers, either the reconstruction error is large, or the encoding distribution is different than the distribution over encoded target data. Define $P_{\mathcal{Z}}(\mathbf{z} \mid \omega_T) = P_{\mathcal{Z}}(\text{enc}(\mathbf{x}) \mid \omega_T)$ for $\mathbf{x} \sim P_{\mathcal{X}}(\mathbf{x} \mid \omega_T)$ to be the distribution over encoded target data. Further define rec_err$(\cdot)$ as given in Equation 2.4 to be the reconstruction error function, that measures how well a sample $\mathbf{x}$ is encoded and decoded. Making use of Theorem 3.1 and Theorem 3.2 a one-class classifier can be constructed.

Theorem 3.1. *If a sample* $\mathbf{x} \sim P_{\mathcal{X}}(\mathbf{x} \mid \omega_T)$ *is given, then both of the following properties are expected to hold.*

Property 1. $\text{enc}(\mathbf{x}) \sim P_{\mathcal{Z}}(\mathbf{z} \mid \omega_T)$

Property 2. rec_err(**x**) *is small*

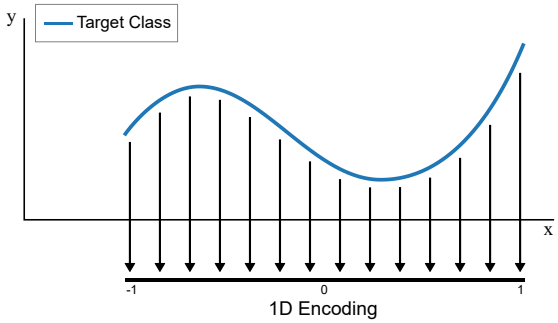
Proof. Given a sample $\mathbf{x} \sim P_{\mathcal{X}}(\mathbf{x} \mid \omega_T)$. (1) By definition it holds that $\text{enc}(\mathbf{x}) \sim P_{\mathcal{Z}}(\text{enc}(\mathbf{x}) \mid \omega_T)$. (2) The autoencoder is trained to minimize the reconstruction error for samples from the target dataset X. Assuming that X is a representative sampling from the target class and there is no severe overfitting, it is expected that rec_err(**t**) is small for all $\mathbf{t} \sim P_{\mathcal{X}}(\mathbf{x} \mid \omega_T)$, thus for **x** the reconstruction error is expected to be small. □

Theorem 3.2. *If a sample* $\mathbf{x} \in P_{\mathcal{X}}(\mathbf{x} \mid \omega_O)$ *is given, then at least one of the properties defined in Theorem 3.1 will not hold.*

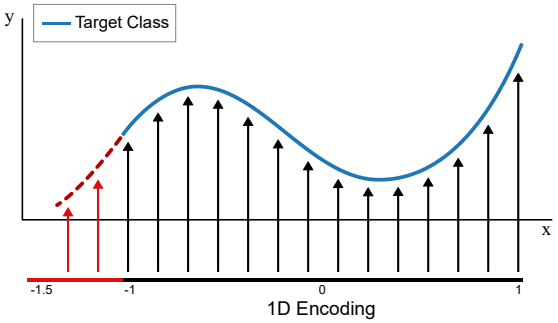
Proof. Assume Property 1 holds for sample $\mathbf{x} \sim P_{\mathcal{X}}(\mathbf{x} \mid \omega_O)$. This means that enc(**x**) is similar to the encoding of the target data. This implies that dec(enc(**x**)) must be similar to the target data. The input sample **x** is defined to be an outlier, so it must be dissimilar to the target data according to Assumption 1.1. Thus, the input and reconstructed output must be dissimilar and the reconstruction error must be large. This proves that if Property 1 holds then Property 2 does not hold, so it is impossible for both properties to be satisfied when $\mathbf{x} \in P_{\mathcal{X}}(\mathbf{x} \mid \omega_O)$.

□

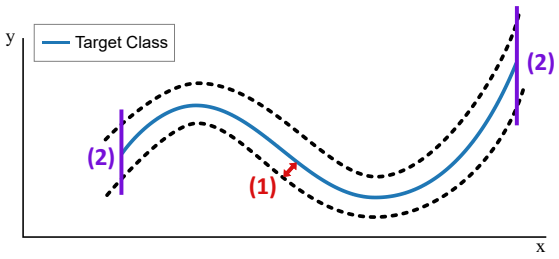
If, for a given sample, both properties hold, it belongs to the target data and otherwise it is an outlier. It is, however, hard to determine if an encoded sample enc(**x**) is likely to be sampled from $\mathcal{Z}$ for an arbitrary autoencoder. For this reason, variational autoencoders or Wasserstein autoencoders are preferred, as both of them approximate $P_{\mathcal{Z}}(\mathbf{z} \mid \omega_T) = \mathcal{N}(\mathbf{0}, \mathbf{I})$. This makes it possible to compute the likelihood of an encoded sample **z**, thus Property 1 can be verified. Property 2 can be verified by computing the reconstruction error of the training data and determining an appropriate threshold, such that most of the training data is classified correctly.



(a) Each sample from the target class (blue line) could be uniquely encoded in 1D by removing its y value (other encodings are also possible).



(b) By decoding the encoded target data, the original data can be reconstructed. Decoding smaller or larger values could, however, result in non-target data to be reconstructed as well, as is illustarted in red (left).



(c) By thresholding the reconstruction error (1), the data outside of the black dotted lines can be rejected. To avoid reconstructing non-target data, as is shown in Figure 3.1b, encoded samples that deviate from the target class encoding should be rejected as well. This is illustrated by the purple bounds (2).

Figure 3.1: Illustration of how an autoencoder could be used as one-class classifier for target data that lies in a 1D subspace in 2D space.

An illustration of the application of an autoencoder to a 2D problem is given in Figure 3.1. In Figure 3.1a, the 2D target class and a 1D encoding are shown. It is important that each sample from the target class is

encoded uniquely, such that the decoder is able to correctly decode each encoded sample. In Figure 3.1b, the decoder is shown. In Figure 3.1c, the bounds are shown that are necessary to construct a one-class classifier. The dotted line corresponds to thresholding the reconstruction error (this satisfies Property 2 from Theorem 3.1), while the solid purple line corresponds to bounding the range of encodings that is accepted (this satisfies Property 1 from Theorem 3.1).

3.3. Applicability of Generative Adversarial Networks to One-Class Classification

There are several ways in which GANs could be used for one-class classification, which are listed below

1. The discriminator network of a GAN attempts to separate real (target) samples from fake (generated) samples, thus it might be possible to use the discriminator as a one-class classifier. Since the discriminator is only trained to separate generated data from the target data, there is no guarantee that arbitrary non-target data is also separated correctly from the target data.

2. The generator of a GAN and the decoder of an autoencoder fulfill a similar function. They both take an input vector $\mathbf{z}$ as input and generate, or reconstruct, a target data sample. This makes it possible to 'append' a GAN to the end of an autoencoder, such that the decoder and generator are equal. This way, the autoencoder ensures that all target data samples in the training dataset can be reconstructed correctly, while the GAN ensures that the decoder/generator cannot produce outlier samples. Thus, the objective of the autoencoder is to satisfy Property 2 from Theorem 3.1 for all samples $\mathbf{x} \sim P_{\mathcal{X}}(\mathbf{x} \mid \omega_T)$ and the objective of the GAN is to satisfy Property 1 for all $\mathbf{x} \in \mathbb{R}^n$. Since Property 1 is satisfied for all samples, only Property 2 will have to be checked to determine if a sample belongs to the target class. This method is further discussed in Experiment 11.1 and Experiment 11.3.

3. GANs can be used to generate more target data. With additional target data it might be possible to improve the performance of various one-class classification techniques. This use-case of a GAN is not investigated in this thesis, but is listed for the sake of completeness.

4

Method of Benchmarking & Experimental Setup

4.1. Datasets

Multiple datasets were used to evaluate the performance and behavior of the implemented one-class classifiers.

4.1.1. Artificially Generated Data (LINE-2D)

A simple 2D dataset was artificially generated to get an initial idea about the behavior of the implemented classifiers on data distributed in subspaces. This dataset was used to verify if the classifiers behave as expected for trivial problems. The performance measures on this dataset are not representative of the quality of the classifier, as real-world problems tend to be much more complex.

The 2D dataset is shown in Figure 4.1. The data is generated on a line segment that is defined by the mathematical formula $y = (x^3 - 6x^2 + x^4 + 12)/8$ and is limited to the domain $x \in [-2, 6]$. This dataset was chosen to lie on a manifold, because we suspect that for higher dimensional data, such as images, the target class lies in subspaces, as was argued in Section 3.1. For brevity, this dataset is referred to as *LINE-2D*.

To compare the performance of different one-class classification methods for this dataset, a non-target class is also defined. The non-target class consists of samples that are uniformly distributed in the area defined by $x \in [-4, 8]$, $y \in [-6, 6]$.

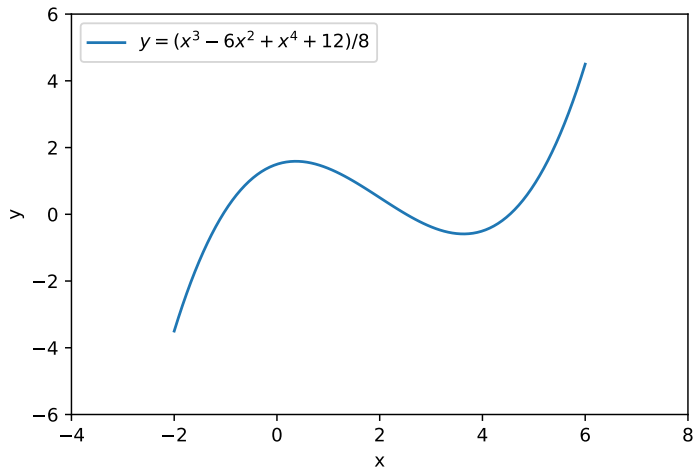


Figure 4.1: The line segment on which samples from the LINE-2D dataset are uniformly distributed.

4.1.2. MNIST

MNIST [18] is a dataset that consists of handwritten digits. This is one of the most popular datasets in the field of machine learning and computer vision to test simple algorithms. The MNIST dataset has been conveniently split up into an independent set of 60,000 train and 10,000 test grayscale images. The images are 28×28 pixels large, so each sample can be represented by a $28 \cdot 28 = 784$ dimensional feature vector. While this is still quite low-dimensional, especially compared to high-definition photos, it is a good starting point. A random subset of the images from the MNIST dataset is shown in Figure 4.2. This dataset might be comparable to the complexity of simple real-world problems.

To use this dataset as a one-class classification benchmarking problem, a subset of the digits is used as target data. For example, the digits 3 and 4 could be used as target data, while all other digits are used as outliers.

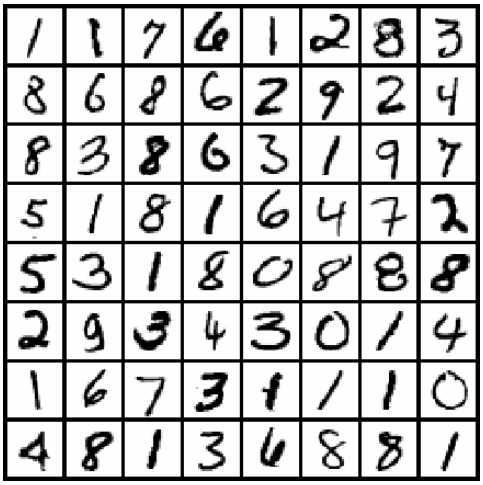


Figure 4.2: 64 randomly chosen samples from the MNIST dataset. Note that the colors are inverted to make the image more print-friendly. This will be done throughout the report.

4.1.3. Cifar-10

The Cifar-10 dataset [16] consists of small color pictures of 10 types of objects. The images have a resolution of 32×32 pixels and thus can each be represented by a $32 \cdot 32 \cdot 3 = 3072$ dimensional feature vector. The dataset consists of 5,000 train and 1,000 test images per class. The 10 types of classes are: (0) plane, (1) car, (2) bird, (3) cat, (4) deer, (5) dog, (6) frog, (7) horse, (8) ship and (9) truck. A subset of the images is shown in Figure 4.3. This dataset is much more complex than the MNIST dataset and it is expected to be comparable in complexity to real-world problems. For one-class classification, this dataset is expected to be especially complex, since a large part of each image is part of the background.

To use this dataset as a one-class classification benchmarking problem, one of the 10 classes will be considered as the target class and the other classes will be used as outliers.

4.2. Performance Measurement

Measuring the performance of a one-class classifier is not a trivial task. One typically wants to maximize the number of samples from the target dataset that are correctly classified as inliers, while minimizing the total volume of the feature space that is classified as inlier, such that as many outliers as possible will be correctly classified. While for low-dimensional problems it might be possible to use random sampling to approximate the volume that is classified as target data, for higher dimensional problems this becomes unfeasible, because the volume becomes very small compared to the total volume.

One solution is to measure the performance by using outliers that are relatively similar to the target data. For example, for the MNIST dataset, if digits 3 and 4 are used as target data, all other digits can be used as outliers. This results in a somewhat arbitrary performance measurement, but it can be used to compare different classifiers.

To compare different classifiers, the ROC (Receiver Operator Characteristic) curve is used. The ROC curve plots the true positive rate (TPR) against the false positive rate (FPR), when varying the decision threshold.

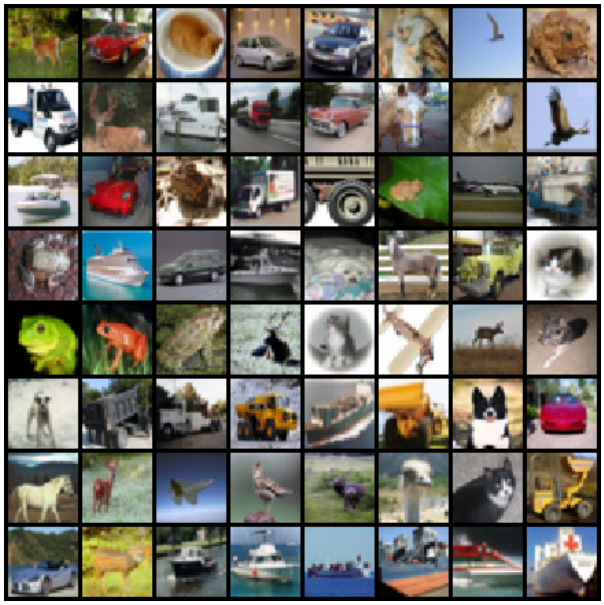


Figure 4.3: 64 randomly chosen samples from the cifar-10 dataset.

The TPR corresponds to the amount of target data that is correctly marked as inlier, while the FPR corresponds to the amount of outliers that are incorrectly marked as inliers. An ideal classifier would have a TPR equal to 1 and a FPR equal to 0. An example ROC curve is shown in Figure 4.4.

Comparing ROC curves by eye is, however, not accurate and can become especially difficult when comparing multiple results. For these cases, the AUC (Area Under the Curve) will be used. The AUC is a value between 0 and 1 which corresponds to the area under the ROC curve. A value closer to 1 corresponds to a better classification performance and a completely random classifier would have an AUC of 0.5. The AUC is an accurate method to evaluate the performance of a classifier and has many desirable properties, which are explained in more detail in [3]. One of the nice properties of the AUC is that it is resilient to class imbalance and outliers.

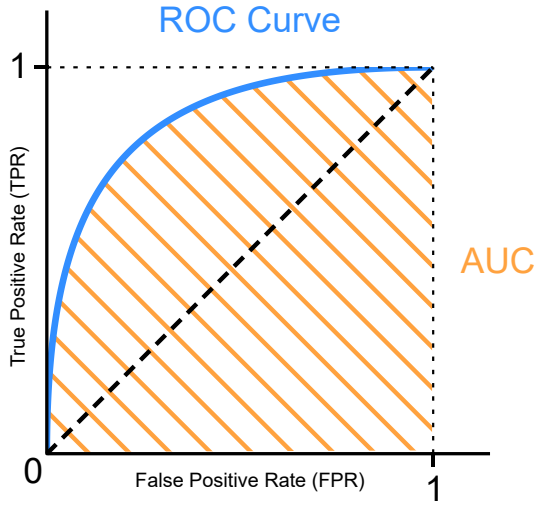


Figure 4.4: Example plot of an ROC curve. The dashed area corresponds to the AUC and the dashed line is the line TPR = FPR, which corresponds to the expected performance of a purely random classifier.

4.3. KL Divergence as Similarity Measure between Distributions

Whenever the similarity of one distribution to another must be determined in this report, the KL divergence is used. The theory behind the KL divergence was discussed in Section 2.3.2. In some experiments, the

distribution of the encoded target data is analyzed. Here, the exact distribution cannot be determined, since only a finite number of samples from the target dataset are available. Instead, kernel density estimation will be used to approximate the probability density function (PDF). Kernel density estimation was explained in Section 2.1.3.

To determine the KL divergence between an approximated PDF $\mathrm{p}_A(x)$ and a known PDF $\mathrm{p}_B(x)$, numerical integration is used to compute

$$D_{KL}(\mathrm{A} \parallel \mathrm{B}) = \int_{-\infty}^{\infty} \mathrm{p}_A(x) \log \frac{\mathrm{p}_A(x)}{\mathrm{p}_B(x)} dx$$

This is done by first choosing a range $[l, u]$ that contains all of the values where $\mathrm{p}_A(x)$ is non-zero. This range is usually not very large, since the models that were used tend to generate values that lie relatively close to 0. Most of the trained models did not generate any values outside of the range $[-5, 5]$. Next, a sufficiently small step size δ is used, such as $\delta = 0.001$ and the above integral is turned into the following summation

$$D_{KL}(\mathrm{A} \parallel \mathrm{B}) \approx \delta \cdot \sum_{i=l,\mathrm{step}=\delta}^{u} \mathrm{p}_A(i) \log \frac{\mathrm{p}_A(i)}{\mathrm{p}_B(i)}$$

As long as δ is chosen small enough, this should give an accurate approximation of the KL divergence. The kernel density estimator is the main source of error. Especially the bandwidth parameter is difficult to determine, as it is highly dependent on the amount of available data to correctly model the low density areas. In practice, the optimal bandwidth parameter will have to be determined separately for each dataset.

4.4. Fixed parameters

Some parameters have not been varied during any of the experiments and are specified here. All other parameters will be specified separately for each experiment.

- Most networks were trained using the Adam optimizer, with a learning rate of 1e−3 and a weight decay of 1e−5. The convolutional networks that are used in Chapter 11 were trained with the Adam optimizer, using a learning rate of 2e−4 and $\beta = 0.5$.
- For the LINE-2D dataset, the batch size is set to 256. The LINE-2D dataset has not been normalized.
- For the MNIST dataset, the batch size is set to 128. The MNIST data is normalized, such that the pixel values have unit variance and the mean is equal to zero.
- For the Wasserstein autoencoder (WAE), the discriminator network was chosen to consist of 3 layers. The input has the same number of dimension as $\mathbf{z}$ and the following three layers have the sizes: 16, 8, 1. ReLUs were used as activation function and a Sigmoid is applied at the end of the network. On a first inspection, varying the discriminator network for a WAE did not seem to have much impact, although no extensive analysis has been performed.

4.5. Structure of the Experiments

In Chapters 5 to 10, several experiments are performed to answer the research questions. To present the experiments in a structured manner, each experiment is preceded by a description, the purpose, and the conclusion of the experiment. This information also serves as a short summary of the experiment and is shown in a highlighted box to make it clearly visible. An example is shown in Experiment 4.5. The number in the black box in the top-right corner, that is preceded by the text 'RQ', corresponds to the relevant research question(s) for the given experiment.

Experiment 4.5 RQ. 1

Description ...

Purpose ...

Conclusion ...

5

Applying Conventional One-Class Classification Methods to High-Dimensional Data

Two of the conventional methods that were discussed in Section 2.1 are analyzed. The focus of this chapter is to answer research question 1: *"How do conventional one-class classification methods perform on high-dimensional data?"* The expectation is that conventional methods do not perform well for high-dimensional data, as was already mentioned in Section 2.1 and further discussed in Section 3.1.

5.1. Nearest Neighbor Distance

Experiment 5.1 RQ. 1

Description The performance of a nearest neighbor method for one-class classification is measured.

Purpose Nearest neighbor algorithms are simple and perform well when the training data consists of a dense sampling of the target class. It is, however, expected that the amount of data required to densely sample the target class grows exponentially with the number of dimensions, as was explained in Section 2.1. Nearest neighbor algorithms are thus expected to not perform well for high-dimensional problems. The purpose of this experiment is to determine if the aforementioned reasoning is correct and if the chosen datasets are sufficiently complex to justify the need for a more complex approach.

Conclusion The performance on the MNIST dataset is quite high, thus the MNIST dataset does not justify the need for a more complex one-class classification technique. The Cifar-10 dataset was found to be much more difficult, resulting in a much worse performance for the nearest neighbor method. The Cifar-10 dataset could thus be used as a difficult dataset to test if the considered methods perform well for a complex problem, while the MNIST dataset can be used as a simpler datasets for comparison purposes and to find points where the methods could be improved.

First, the performance on the MNIST dataset is measured. A pair of two digits is chosen as target data and all other digits are considered to be outliers. This is done for all 45 combinations of digit pairs and for each pair the performance is measured.

For each test sample, the algorithm calculates the Euclidean distance to the closest sample from the training dataset. This distance can then be thresholded to result in a one-class classifier. The performance of this algorithm for the MNIST dataset is shown in Figure 5.1. The performance of this nearest neighbor algorithm is quite high. The performance for most digit pairs can still be improved a lot, so the MNIST dataset could be used to benchmark the performance of various methods. The difficulty of this problem is, however, not high enough to justify that complex one-class classification method, such as autoencoders or GANs, are required.

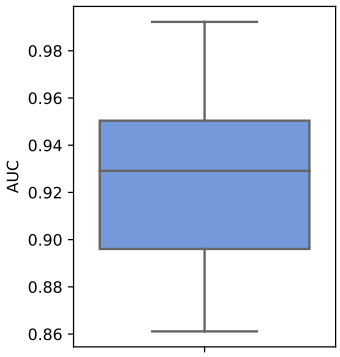


Figure 5.1: The performance of a nearest neighbor algorithm for the MNIST dataset. In each run, two digits were used as the target class and all other digits were used as outliers.

A similar experiment was done using the Cifar-10 dataset. For this dataset, one class is chosen as target class and samples from the other classes are used as outliers. The performance for this problem is shown in Table 5.1 and is clearly much worse than for the MNIST dataset.

Target Class	AUC
Airplane	0.71
Automobile	0.46
Bird	0.69
Cat	0.52
Deer	0.77
Dog	0.51
Frog	0.73
Horse	0.53
Ship	0.69
Truck	0.44

Table 5.1: The performance of nearest neighbor for each target class from the Cifar-10 dataset.

5.2. Mixture of Gaussians

Experiment 5.2 RQ. 1

Description The performance of a gaussian mixture model is measured for the MNIST dataset.

Purpose A gaussian mixture model is expected to not perform well for high-dimensional data, because it is a probability density estimation technique. This hypothesis is tested in this experiment.

Conclusion A gaussian mixture model has a much worse performance than the nearest neighbor method that was tested in Experiment 5.1. The performance on the MNIST dataset is, however, not as bad as was expected. Only for some specific digits a gaussian mixture model performs well, such as digit 1 and 6. The performance on the Cifar-10 dataset is bad.

A gaussian mixture model is trained using pairs of digits from the MNIST dataset as target class. The number of gaussian components is varied between 1 and 64 to ensure that the results do not depend on an arbitrary choice of this parameter. The results are shown in Figure 5.2. The performance does not seem to depend much on the number of gaussian components that are used. Compared to the nearest neighbor classifier tested in Experiment 5.1, the performance of a gaussian mixture model is much worse.

The performance in Figure 5.2 varies a lot, which might indicate that certain digit combinations are much simpler to learn than others. To verify this, the performance per digit is shown in Figure 5.3. Note that pairs of two digits were chosen as target data, so in this figure $digit = 2$ actually means all digit pairs that include

digit 2. There is a clear difference in performance between the digits. Any target class that includes digit 1 is apparently simple to learn, resulting in a high performance. Digit 2 and 8 seem to be the most difficult.

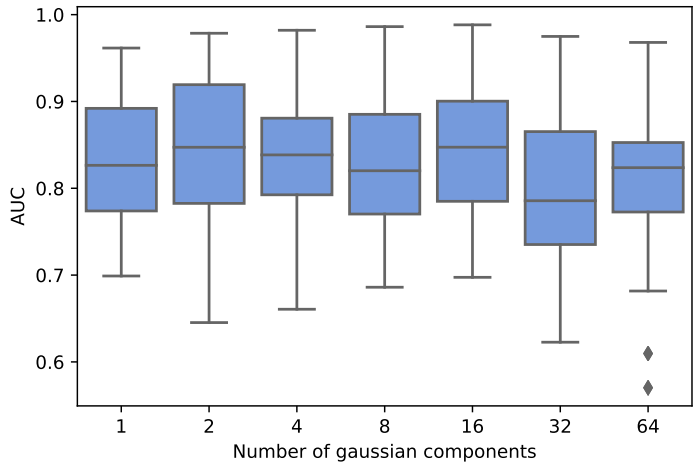


Figure 5.2: The performance of a mixture of gaussians for the MNIST dataset, for a varying number of gaussian components. In each run, two digits were considered to be the target class and all other digits were used as outliers.

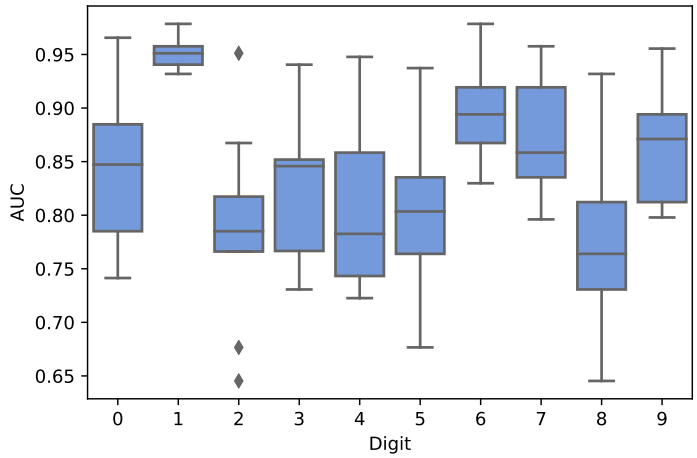


Figure 5.3: The performance per digit of a mixture of gaussians for the MNIST dataset. Here the number of gaussian components is set to 2 for all models. The models were trained on pairs of two digits. The x-axis denotes one of these digits. For example, for $x = 2$ a boxplot is shown of the AUC for each problem where one of the two target class digits is equal to 2.

A similar experiment was done using the Cifar-10 dataset. For this dataset, one class is chosen as target class and samples from the other classes are used as outliers. The performance for this problem, using a varying number of gaussian components is shown in Figure 5.4. The performance of a gaussian mixture model for the Cifar-10 dataset is not good, as the AUC is close to 0.5 for most of the classes.

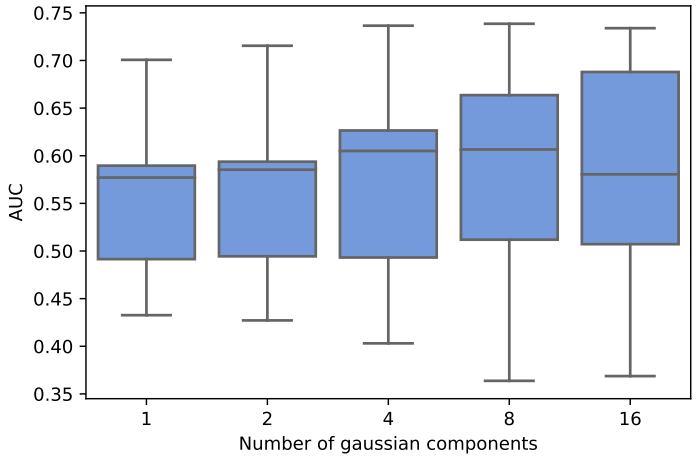


Figure 5.4: The performance of a mixture of gaussians for the Cifar-10 dataset, for a varying number of gaussian components. In each run, one class from the Cifar-10 dataset is considered to be the target class and all other classes were used as outliers.

6

Feasibility of using Generative Adversarial Networks for One-Class Classification

GANs are analyzed to determine if they could be applied to one-class classification problems. This chapter (together with Chapter 7) attempts to answer research question 2: *"Can modern unsupervised neural network techniques, such as generative adversarial networks (GANs) and autoencoders (AE), be applied effectively to (high-dimensional) one-class classification problems?"*

6.1. Applying GANs to the Artificially Generated LINE-2D dataset

Experiment 6.1

RQ. 2

Description GANs of varying complexity are trained on the LINE-2D dataset and the one-class classification performance (AUC) of the discriminator is reported. The depth of the generator and discriminator networks are each varied between 2 to 4 layers and the number of neurons in each layer is varied between 4 and 32.

Purpose The LINE-2D dataset is very simple, so any successful one-class classification technique should achieve a good performance. This experiment shows that if the discriminator of a GAN has a good performance and thus when they could potentially be used for one-class classification purposes. Multiple network variants are tested to provide a more objective answer to this question, since it ensures that we did not choose a specific variant that happens to perform good or bad. It also provides insight into the consistency of GANs. When applying a GAN to a real-world one-class classification problem it would be desirable if most networks have similar performance, such that little time has to be spend on optimizing the hyperparameters.

Conclusion Using the discriminator of a GAN as one-class classifier does not result in a good/consistent performance, even though the LINE-2D dataset that is tested with is simple. Autoencoders show much better performance and more stable results in later experiments, so GANs are not investigated in much detail in the remainder of this report.

Multiple network variants are tested in this experiment. Each hidden layer of the generator and discriminator is followed by a ReLU activation function and the last layer of the discriminator is followed by a Sigmoid function. The number of neurons $n^{(i)}$ in layer i is varied as follows. For the generator $n^{(i)} \in \{8,4,2\}$ and for the discriminator $n^{(i)} \in \{32,16,8,4\}$. Each layer is chosen such that is has at most as many neurons as the previous layer, so the networks cannot become wider: $n^{(i)} \geq n^{(j)}$ for $i < j$. The number of neurons in the last layer is equal to 2 for the generator and 1 for the discriminator, such that the output is of the correct dimensionality. The used networks are similar to the one shown in Appendix A.1.

In total, 19 variants for the generator and 34 variants for the discriminator are tested. The size of the input of the generator is also varied between $z\_size \in \{1,2,4\}$ and each combination has been trained 5 times. This

results in a total of $19 \cdot 34 \cdot 3 \cdot 5 = 9{,}690$ trained networks. Each network was trained for 2,000 epochs, to ensure convergence.

Figure 6.1 shows the performance for each discriminator when used to separate the target and non-target data. The target and non-target class for this dataset were defined in Section 4.1.1. Many discriminators have an AUC around 0.5, which is equivalent to the score of a purely random discriminator and many discriminators perform even worse than that. Thus, the performance of GANs for one+class classification does not seem good, even though a simple dataset was used.

In Figure 6.2, the performance is compared for various complexities of the generator network. On the x-axis, the number of neurons in each layer is shown and the y-axis shows the performance (AUC). For example, the value "8-4" on the x-axis corresponds to a generator that contains 8 neurons in the first layer, 4 neurons in the second layer and it does not have a third layer. The results show that using more neurons per layer is beneficial, while using three instead of two layers does not result in an improvement in performance. Thus, it seems that the width of the network is much more important than the depth. In the further analysis, only the networks where the generator has 8 neurons in each layer are kept, since they perform significantly better than the other networks.

The performance of the remaining networks for the discriminator variants is shown in Figure 6.3. Note that the results are split into two parts, because the plot is too wide to fit on the page. Once again, the performance mainly depends on the number of neurons per layer.

After selecting the best performing generator and discriminator networks, the performance of the remaining networks is shown in Figure 6.4. In this figure the performance is compared for various sizes of $\mathbf{z}$. The performance of the discriminator seems slightly better when using a higher number of dimensions for the random noise $\mathbf{z}$ that is used as input for the generator. The performance for none of the variants is, however, consistent, as high variance is observed.

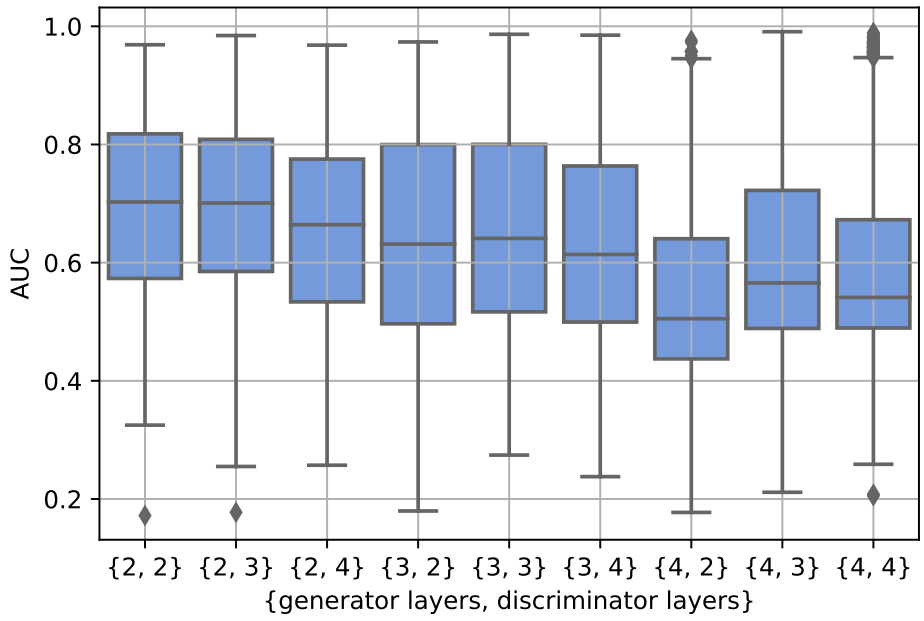


Figure 6.1: The AUC for various numbers of layers for the generator and discriminator.

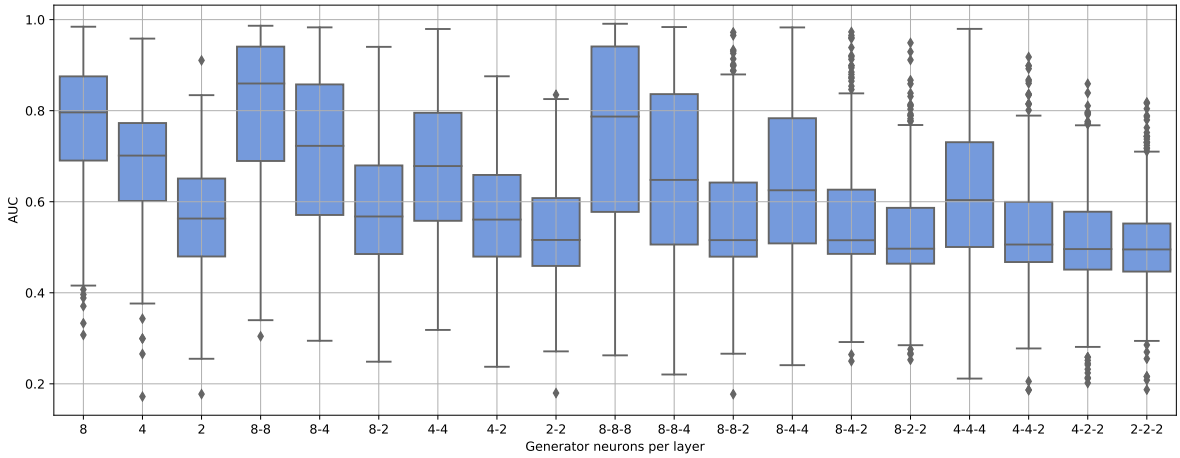


Figure 6.2: The AUC for all generator networks that were tested.

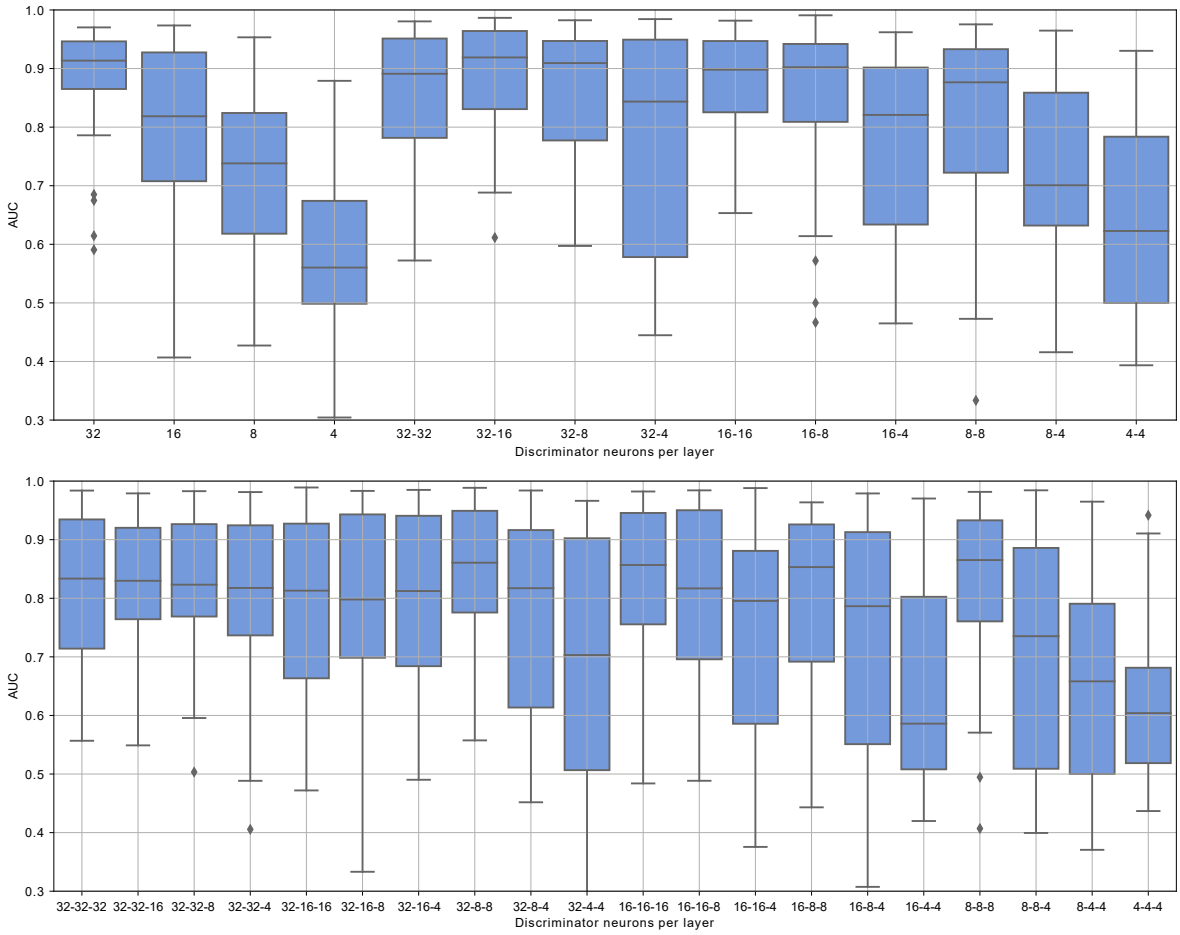


Figure 6.3: The AUC for all discriminator networks that were tested. All networks where the generator has fewer than 8 neurons in one of its layers have been removed, as they were shown to not perform well in Figure 6.2.

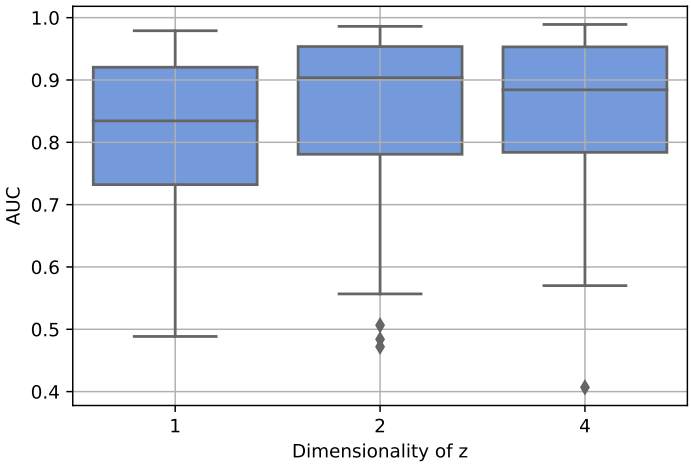


Figure 6.4: The AUC for various numbers of dimensions for z, which is the input of the generator.

6.2. Analyzing the Generator and Discriminator

Experiment 6.2 RQ. 2

Description The behavior of the generator and discriminator of a GAN during the training phase are analyzed.

Purpose The purpose of this experiment is to determine how the discriminator of a GAN learns, if there are any potential weaknesses and if the discriminator could be used for one-class classification purposes.

Conclusion The issue of mode collapse was shown to occur for GANs, even for the simple LINE-2D dataset. This means that the generator does not converge to a state where it is able to generate samples from the entire target class, but instead can only generate samples from a limited part of the target class. Mode collapse was also shown to not always occur. Simply restarting the training phase could result in a network that does not have any mode collapse. Mode collapse can thus occur at random and is difficult to avoid.

The discriminator was also shown to only learn a correct classification for the non-target class near areas where the generator generates samples during the training phase. This is to be expected, as the discriminator cannot know how to classify a sample in an area where it has never seen any target or generated sample.

The discriminator of a GAN can thus not reliably be used as a one-class classifier for non-target data. For target data, the classification does seem reliable, so it might be possible to use an ensemble of GANs to create a more reliable one-class classifier. An ensemble of GANs could, for example, classify any sample as an outlier in case at least one GAN classifies it as outlier. An ensemble of GANs is not evaluated in this report, as autoencoders were found to perform much better and have fewer potential issues.

The network architecture that is described in Appendix A.1 is used during this experiment. This is one of the best performing networks that was found in Experiment 6.1.

In Figure 6.5 (on page 30), the output of the generator and decoder are shown after varying numbers of training epochs. In the figures on the left, generated samples are shown in orange and the target class is shown in blue. These samples were generated by using 64 evenly spaced values from the range $\mathbf{z} \in [-3, 3]$ as input for the generator. In the figures on the right, the classification of the discriminator is shown. Areas classified as 0 (light red) are classified as 'fake', while areas classified as 1 (dark red) are classified as 'real' data by the discriminator. When using the discriminator of a GAN as a one-class classifier, ideally only the area near the target class would be classified as 1 (dark red) and all other areas in the figure would be classified as 0 (light red). The discriminator is seen to only learn a correct classification in areas where samples are generated

during the training phase. In places where the generator does not generate any samples, the discriminator could produce any result. Figure 6.5d (on page 30) also shows that the generator did not learn to generate samples from the entire target class. This is often referred to as *mode collapse* and is a well known issue of GANs [34].

Mode collapse does not always occur and simply restarting the training process could cause a different outcome. This is shown in Figure 6.6, where the training process was restarted and the resulting generator does not have any mode collapse. The discriminator also performs much better.

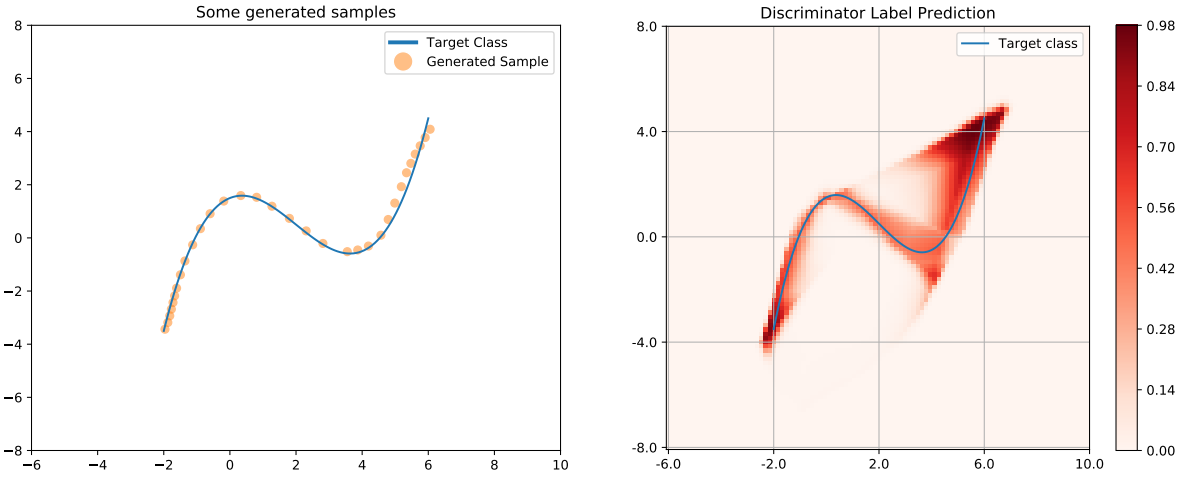
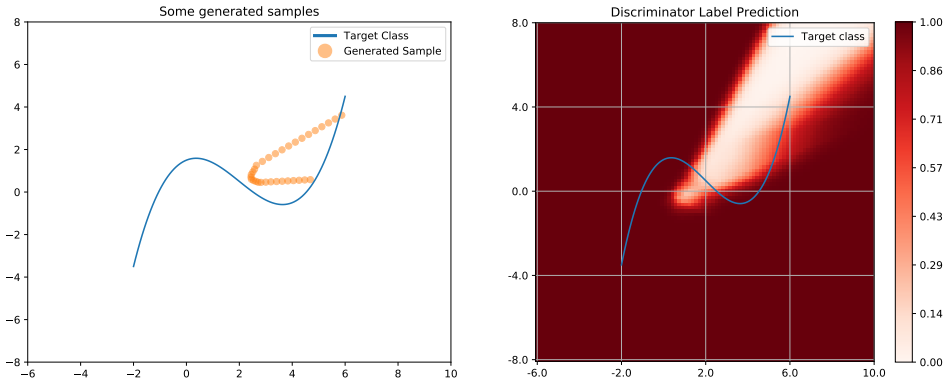
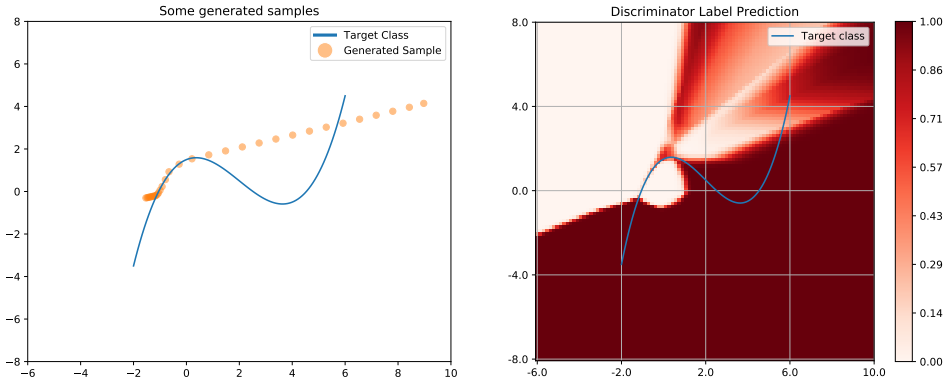


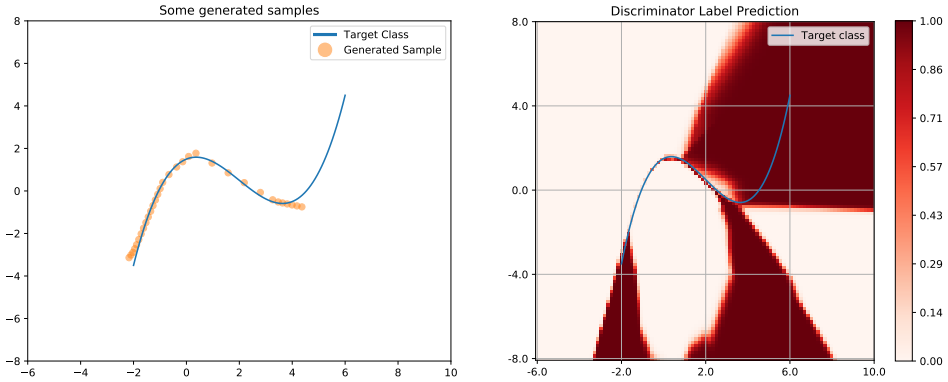
Figure 6.6: Generated samples of the generator and classification of the discriminator after training for 2,000 epochs. The same network was used as in Figure 6.5, but it converged to a different result.



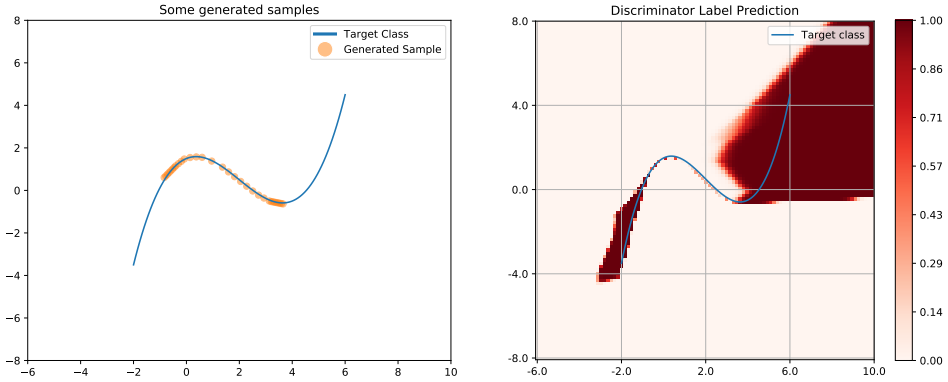
(a) Results after 50 training epochs



(b) Results after 150 training epochs



(c) Results after 350 training epochs



(d) Results after 2000 training epochs

Figure 6.5: Generated samples and learned discriminator classification after varying numbers of training epochs. The generated samples were created by using 64 linearly spaced samples between -3 and 3 as input for the generator.

7

Feasibility of using Autoencoders for One-Class Classification

Autoencoders are analyzed to determine if they could be applied to one-class classification problems. This chapter (together with Chapter 6) attempts to answer research question 2: *"Can modern unsupervised neural network techniques, such as generative adversarial networks (GANs) and autoencoders (AE), be applied effectively to (high-dimensional) one-class classification problems?"*

7.1. Applying Autoencoder to the Artificially Generated LINE-2D Dataset

Experiment 7.1

RQs. 2 & 2a

Description Autoencoders of varying complexity are trained on the LINE-2D dataset and their performance (AUC) is reported. The depth of the encoder and decoder networks are each varied between 2 to 4 layers and the number of neurons in each layer is chosen between 2 and 8.

Purpose This experiment is similar to Experiment 6.1, where network variants were tested for a GAN. The LINE-2D dataset is very simple, so any successful one-class classification technique should achieve a good performance. This experiment shows if autoencoders indeed do have a good performance and thus if they could potentially be used for one-class classification purposes. Multiple network variants are tested to provide a more objective answer to this question, since it ensures that we did not choose a specific variant that happens to perform good or bad. It also provides insight into the consistency of autoencoders. When applying an autoencoder to a real-world one-class classification problem it would be desirable if most networks have similar performance, such that little time will have to be spend to optimize it.

Conclusion The performance of an autoencoder is high and consistent for the LINE-2D problem, provided that the network has sufficient complexity. Only when the second to last layer of the decoder contains too few neurons, the performance was shown to degrade a lot. Other than this, slight variations in complexity of the network did not result in significantly deviating results. This result suggests that optimizing the network should be easy, since no minor adjustments in complexity will have to be tested.

Multiple autoencoder networks are used during this experiment. Each layer except for the last layer of the encoder and the last layer of the decoder is followed by a Tanh activation function and the number of layers of the encoder and decoder are each varied between $2-4$. The number of neurons $n^{(i)}$ in layer i is varied as follows:

2 layers: $n^{(1)} \in \{8,4,2\}$

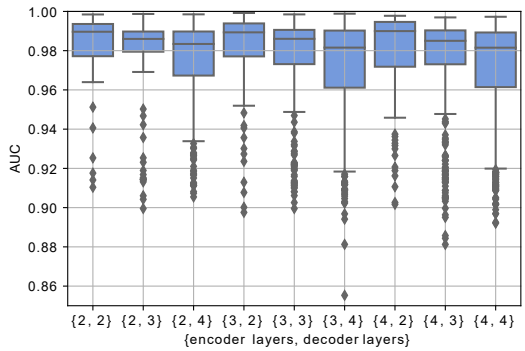
3 layers: $n^{(1)} \in \{8,4,2\}, n^{(2)} \in \{4,2\}$ where $n^{(1)} \geq n^{(2)}$

4 layers: $n^{(1)} \in \{8,4,2\}, n^{(2)} \in \{4,2\}, n^{(3)} \in \{4,2\}$ where $n^{(1)} \geq n^{(2)} \geq n^{(3)}$

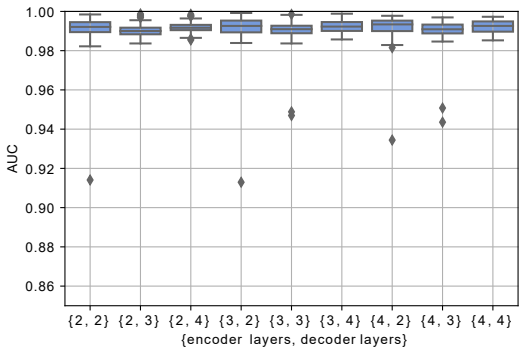
Note that the number of neurons in the last layer is not specified, since it is defined by the dimensionality of $\mathbf{z}$ for the encoder and the dimensionality of the input for the decoder. The input is 2-dimensional and $\mathbf{z}$ is chosen to be 1-dimensional, so $n_{enc}^{(\text{last})} = 1$ and $n_{dec}^{(\text{last})} = 2$. This results in 15 variations for the encoder and decoder, thus 225 combinations in total. Each of these networks are trained using a dataset consisting of 2,560 target samples for 500 epochs, using a batch size of 256 and the Adam optimizer with a learning rate of 1e−3 and a weight decay of 1e−5. Each run is repeated 10 times to get more statistically significant results, thus 2,250 networks are trained in total.

To determine the classification performance of the networks, samples from both the target and outlier class are generated, as was described in Section 4.1. From the target class 25,600 independent test samples are generated and from the outlier class 40,000 evenly spaced samples are generated. For each sample the reconstruction error is calculated and the AUC of a classifier based solely on this reconstruction error is used as performance metric.

First, the depth of the network is analyzed. In Figure 7.1a, the AUC is reported for the number of layer of both the encoder and decoder. There seems to be a high variance and a lot of outliers that negatively impact the average performance. On closer inspection, the networks where the second to last layer of the decoder contains 2 neurons perform significantly worse than all other networks. This is shown in Figure 7.2, where the AUC for each type of decoder network is shown. When removing all networks where the second to last layer of the decoder has only 2 neurons, the results are much more consistent, as is shown in Figure 7.1b.



(a) The AUC for various numbers of layers for the encoder and decoder.



(b) The AUC for various numbers of layers for the encoder and decoder. All variants where the second to last layer of the decoder has 2 neurons were removed, as these networks were shown to perform significantly worse.

Figure 7.1

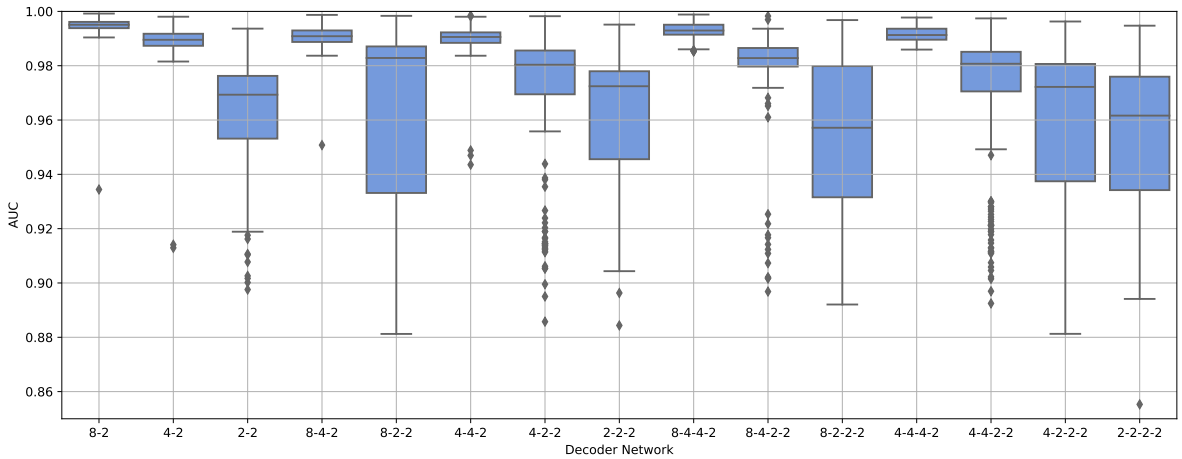


Figure 7.2: The AUC for all decoder variants that were tested.

To provide insight into what AUC performance value could be considered good, a plot showing the reconstruction error of the target class versus the AUC is shown in Figure 7.3. Here all networks where the second to last layer of the decoder has 2 neurons have been removed, resulting in 900 remaining networks. It is not possible to provide an objective measure, but considering that the LINE-2D dataset contains values ranging from $x \in [-2, 6]$, $y \in [-3.5, 5.5]$, a reconstruction error below 0.2 seems quite good, which nearly all of the networks have achieved.

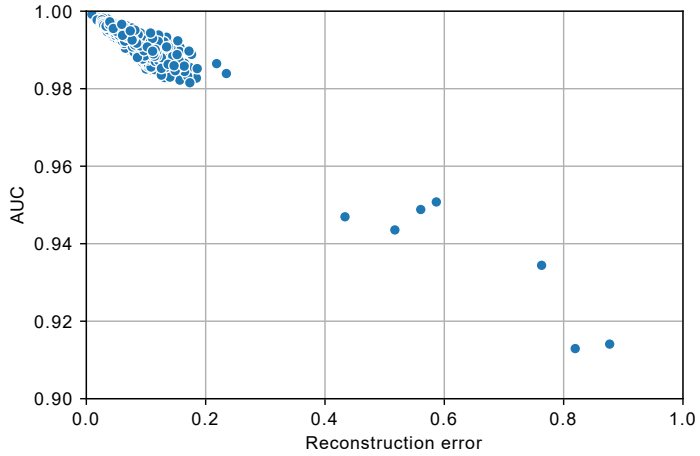


Figure 7.3: The AUC as a function of the average reconstruction error for samples from the target class. The plot shows one dot for each of the trained networks, excluding the networks where the second to last layer of the decoder has 2 neurons, as these networks were shown to perform significantly worse.

7.2. Initial Look at the Reconstruction Error

Experiment 7.2 RQ. 2

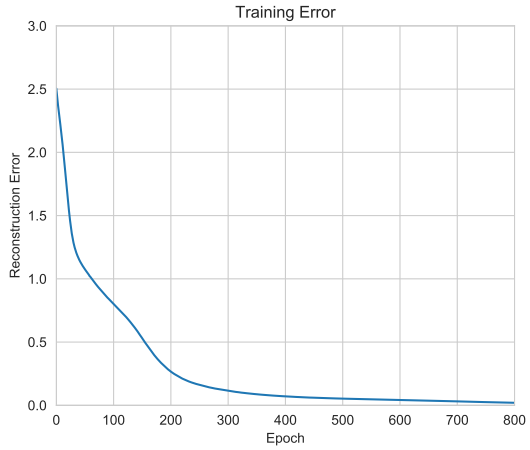
Description The reconstruction error of an autoencoder is analyzed for the target and non-target class to get an initial idea of the behavior of autoencoders.

Purpose As was described in Section 3.2, the reconstruction error is critical when using an autoencoder as one-class classifier. The reconstruction error for target data should be as low as possible and the reconstruction error for non-target data should be much higher, such that the two classes are easily separable.

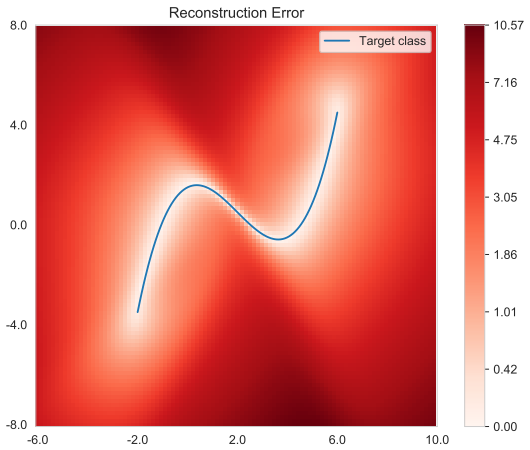
Conclusion For the Line2D dataset the reconstruction error could be thresholded to get an accurate one-class classifier.

This experiment uses the Line2D dataset. The network from Experiment 7.1 with 2 layers for the encoder and decoder and $n^{(1)}_{enc} = 8, n^{(1)}_{dec} = 8$ was used. When training an autoencoder on the Line2D dataset, the reconstruction error of the target class eventually becomes very small, as can be seen in Figure 7.4a.

For the Line2D dataset, all samples that lie on the line segment, or close to it, are considered target data. All other samples are considered to be non-target data. In Figure 7.4b the reconstruction errors are shown for all samples within the domain $x : [-6, 10]$, $y : [-8, 8]$. In the figure it is visible that the reconstruction error becomes larger when moving further away from the target class. This result shows that a one-class classifier that solely considers the reconstruction error of this autoencoder can be very accurate if the threshold is set correctly.



(a) Average reconstruction error for the first 800 epochs, averaged over all training data.



(b) The reconstruction error of the area surrounding the target class. A darker color indicates a larger reconstruction error and the blue line represents the target class.

Figure 7.4: Plots of the reconstruction error.

7.3. Initial Look at the Decoder

Experiment 7.3 RQ. 2

Description The codomain of the decoder is analyzed for an autoencoder that is trained on the Line2D dataset.

Purpose In Experiment 7.2, the reconstruction error for the outlier class was large, but no explanation was given for why it is large. In this experiment, the decoder is analyzed to attempt to answer this question.

Conclusion In this experiment, the codomain of the decoder was found to be similar to the space that is occupied by the target class. This means that it is impossible for this decoder to generate an output for nearly all of the possible outliers. This explains why in Experiment 7.2, the reconstruction error for outliers was observed to be large. When using a VAE or WAE, this result can be improved by rejecting encodings that are unlikely to belong to the target class.

The decoder function maps a 1D encoding to a 2D reconstructed sample. Since the domain of the decoder is one dimensional, the codomain must be a one dimensional subspace of the 2D sample space. Any sample that lies in this 1D subspace can be reconstructed without error, given a correct encoding. Any sample that lies far away from this 1D subspace must have a large reconstruction error, since the decoder is unable to generate a sample that lies close to it. This means that an autoencoder that has a low reconstruction error for the target class, must have a codomain that (roughly) overlaps with the target class.

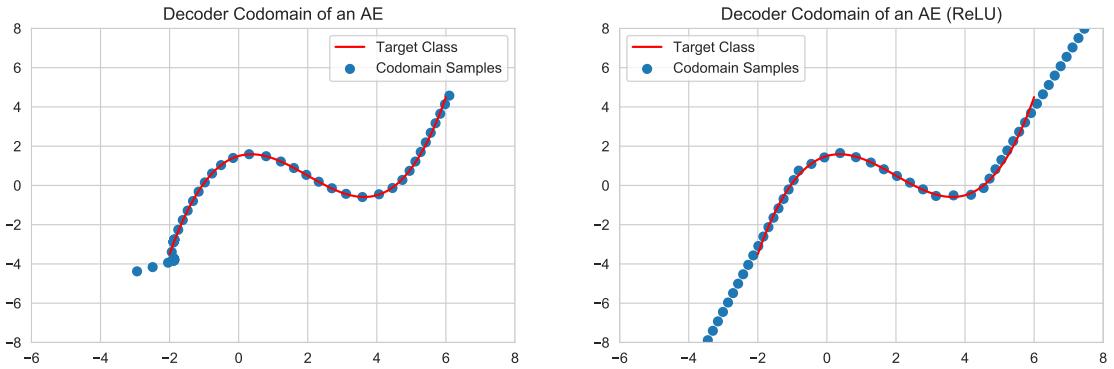
In Figure 7.5a, the entire codomain of the decoder is visualized. This figure was generated by varying the input of the decoder and plotting the output. To make the figure visually clearer, the plotted samples were chosen such that they are evenly spaced. As expected, the codomain of the decoder overlaps with the target class and lies on a 1D subspace of the 2D sample space. Interestingly, the 1D subspace ends abruptly in the bottom left and top right corners of the image and thus does not cover much of the outlier space. The codomain is bounded like this because of the chosen activation function. The Tanh activations ensure that the output of each layer is scaled between -1 and 1, thus when increasing the input of the decoder, at some point the output of the first layer will consist of a vector of values that are each either very close to -1 or 1 and increasing the input of the decoder even further will have a negligible effect on the output. When using values outside of the range $[-3.5, 3.5]$ as input for the decoder, no noticeable difference was observed in the output.

To verify that, indeed, the choice of activation function causes the codomain of the decoder to be limited,

the codomain of a similar network that uses ReLU activations is shown in Figure 7.5b. When decoding extremely large values, this network seems to behaves like a linear function. To illustrate this, some computed values are shown in Equation 7.1. Multiplying the input by 10 also causes the output to become 10 times larger, thus the function seems to behave linearly for extremely large inputs. This is to be expected when considering that $\text{ReLU}(x) = \max(0, x)$. If for a certain part of the domain the input of none of the ReLU activation functions changes between a negative and positive value, then the network will behave like a linear function for this part of the domain. When using extremely large values as input for the decoder, this appears to be the case.

$$\begin{aligned} \text{dec}(-10000) &= (2798,\ 8621) \\ \text{dec}(-100000) &= (27959,\ 86254) \\ \text{dec}(-1000000) &= (279568,\ 862589) \end{aligned} \tag{7.1}$$

$$\begin{aligned} \text{dec}(10000) &= (-2384,\ -8056) \\ \text{dec}(100000) &= (-23844,\ -80582) \\ \text{dec}(1000000) &= (-238437,\ -805851) \end{aligned}$$



(a) A network that uses Tanh activations. Here the entire codomain is shown, which is possible because the Tanh function constrains its output to $[-1, 1]$.

(b) A network that uses ReLU activations. The ReLU function does not constrain its output, so the codomain is unconstrained as well.

Figure 7.5: Samples from the codomain of the decoder function from an autoencoder. The samples were chosen such that they are evenly spaced.

There are, however, no guarantees that the codomain does not cover additional space that does not overlap with the target class. An example of this can be seen in the bottom left corner of Figure 7.5a. Outliers in this area could potentially be reconstructed well, although in Figure 7.4b it is visible that samples in this area do actually have a relatively large reconstruction error. This indicates that the encoder did not encode these outlier samples optimally. The encoder has never seen any outlier samples while training, thus it is likely to not encode them optimally.

Instead of relying on 'luck' that the encoder does not encode outliers well, it is also possible to reject any samples whose encoding deviates significantly from the target encoding distribution $P_{\mathcal{Z}}(\mathbf{z} \mid \omega_T)$, as was discussed in Section 3.2. When using a VAE or WAE this is simple, because the encoded distribution should be distributed as $\mathcal{N}(\mathbf{0}, \mathbf{I})$. To include 99.7% of the target data, all values deviating at most three times the standard deviation from the mean should be included. This corresponds to the interval $[-3, 3]$. When using a WAE and limiting the input of the decoder to values in the interval $[-3, 3]$, the resulting codomain looks as is shown in Figure 7.6. The codomain now only contains samples that belong to the target class, so any outlier sample is guaranteed to have a high reconstruction error.

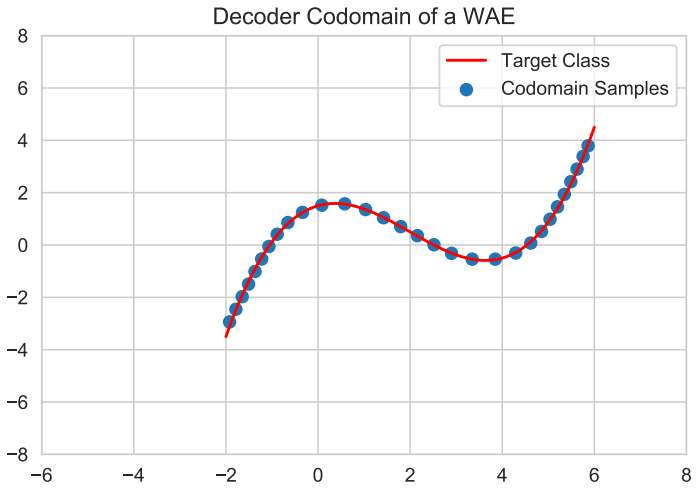


Figure 7.6: Samples from the codomain of the decoder function from a WAE. The input of the decoder is varied between $[-3,3]$.

7.4. Applying Autoencoder to the MNIST Dataset

Experiment 7.4 RQs. 2a & 2c

Description Autoencoders of varying complexity are trained on the MNIST dataset and their performance (AUC) is reported. The depth of the encoder and decoder network are each varied between 2 to 5 layers and the number of neurons in each layer is chosen between 16 and 128.

Purpose In Experiment 7.1, many autoencoder networks were trained on the LINE-2D dataset and the results show that the performance is high and consistent. The purpose of the current experiment is to determine if autoencoders behave similarly for the higher-dimensional MNIST dataset. Once again, a large number of networks is trained to provide an objective measure for the performance and stability.

Conclusion The result of this experiment are similar to what was found in Experiment 7.1. The performance is consistent for all networks and depends mostly on the complexity of the decoder. Especially the number of neurons in the second-to-last layer of the decoder is important. Although there is not a network that performs much better than all others, the best one that was found has as encoder: "128-32-32-32-6" and as decoder: "32-32-128-784", so this network will be used for the remainder of the experiments. The architecture of this network is detailed in Appendix A.2. Potentially an even better network could be found when using more neurons, but optimizing the performance specifically for the MNIST dataset is not the goal of this thesis, so this is not further evaluated.

The dimensionality of $\mathbf{z}$ is set to 6 during this experiment. This value was determined by trial and error. In Experiment 8.2 different sizes for $\mathbf{z}$ are evaluated. The trained networks are similar to the ones that were used in Experiment 7.1, except that more layer and neurons are used. The number of neurons $n^{(i)}$ in layer i is varied as follows:

2 layers: $n^{(1)} \in \{128, 64, 32, 16\}$

3 layers: $n^{(1)} \in \{128, 64\}, n^{(2)} \in \{32, 16\}$

4 layers: $n^{(1)} \in \{128, 64\}, n^{(2)} \in \{64, 32\}, n^{(3)} \in \{32, 16\}$

5 layers: $n^{(1)} \in \{128, 64\}, n^{(2)} \in \{64, 32\}, n^{(3)} \in \{32, 16\}, n^{(4)} \in \{32, 16\}$ where $n^{(3)} \geq n^{(4)}$

Note that for the decoder, the weights of the layers are listed in reverse order (such that the output of each layer gradually becomes larger). Also, the last layer is not listed and contains 6 neurons for the encoder (such that the output has the same dimensionality as $\mathbf{z}$) and 784 neurons for the decoder. Each of the networks is trained on 5 different target classes, each consisting of two digits: $\omega_T \in \{\{3,4\},\{1,5\},\{2,8\},\{6,9\},\{0,7\}\}$. Here

$\omega_T = \{3,4\}$, for example, means that the target class contains all digit 3 and digit 4 samples from the MNIST dataset. Each network is trained for 300 epoch.

The resulting performance values for the various network depths are shown in Figure 7.7. In the figure it is visible that networks where the decoder contains 2 layers perform significantly worse than the other networks. Also the network consisting of 5 layer for both the encoder and decoder performs worse. The remaining networks all have approximately similar performance.

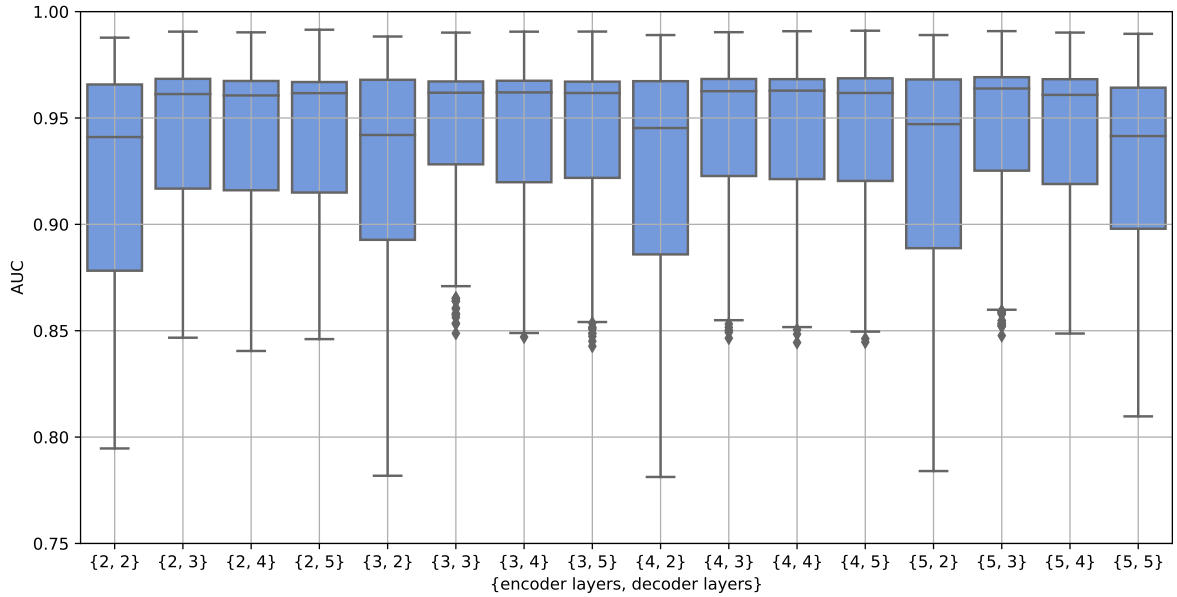


Figure 7.7: Performance for various encoder an decoder network depth.

In Figure 7.8, the performance for the encoder network variants is shown. The networks where the decoder has 2 layers and the networks where both the encoder and decoder have 5 layers were removed, as these were shown to perform worse in Figure 7.7. Varying the encoder network does not seem to have much influence on the performance. Only the encoder network "16-6" (the fourth one from the left) performs slightly worse for some target sets.

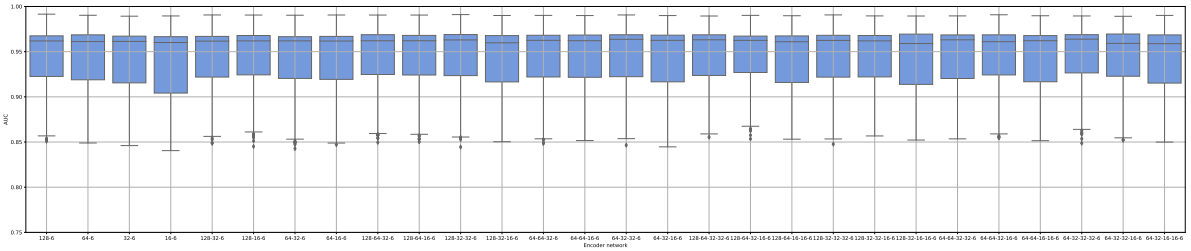


Figure 7.8: Performance for various encoder networks. The worst network depth that were found in Figure 7.7 are removed.

Finally, in Figure 7.9, the performance for the decoder network variants is shown. The networks where the decoder contains two layers, where both the encoder and decoder networks have 5 layers and where the encoder network equals "16-6" have been removed. The performance does not vary much, although it is visible that the decoder variants where the second-to-last layer contains 128 neurons perform slightly better than the variants containing 64 neurons in this layer. This is shown more clearly in Figure 7.10.

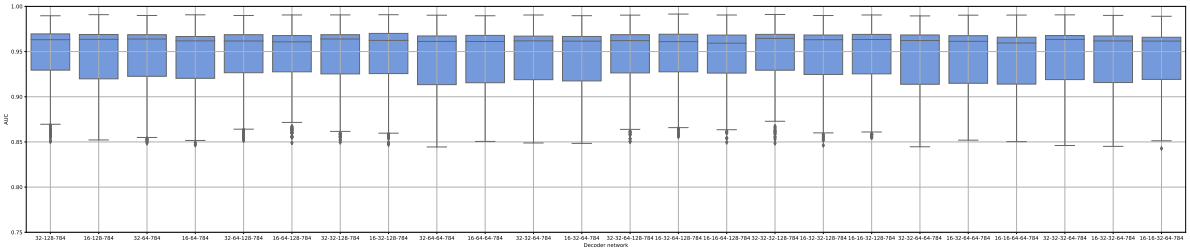


Figure 7.9: Performance for various decoder networks. The worst networks that were found in Figure 7.7 and Figure 7.8 are removed.

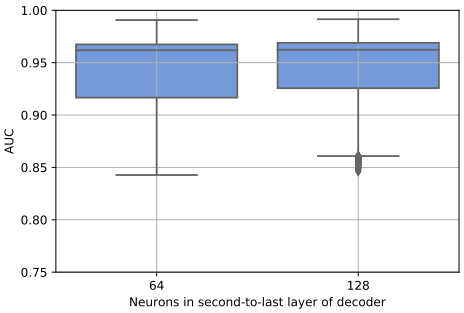


Figure 7.10: Performance depending on the number of neurons in the second-to-last layer of the decoder. The worst network length that were found in Figure 7.7 and Figure 7.8 are removed.

8

Optimizing Parameters

Several parameters and network variants are tested to determine which properties are the most important. The difficulty of optimizing each parameter is analyzed as well. This chapter focusses on research question 2c: *"Are there any parameters that are difficult to optimize?"*

8.1. Activation Function: ReLU vs Tanh

Experiment 8.1 — RQ. 2b

Description Two types of autoencoders are trained, one uses ReLU (rectified linear unit) as activation function and the other uses Tanh (hyperbolic tangent) activations. The resulting encodings are compared to determine which type of activation function performs better.

Purpose A large difference in the distribution of target data encodings was observed for ReLU and Tanh activation functions in Experiment 7.3. The differences are explained and the activation function with the most desirable properties is determined.

Conclusion Tanh activation functions cause the outputs of the intermediate layers of the network to roughly have a bell-shaped distribution. This makes Tanh activation function perform better than ReLU activations when a bell-shaped encoding distribution, such as a gaussian, is desirable. Additionally, a method to distribute the encoding of target samples according to the probability density function $p_{\mathbf{z}}(x) = \frac{1}{2}(1 - \tanh^2(x))$ was discussed. This can be done by adding a Tanh activation at the start of the decoder and, optionally, a small amount of random noise can be added after this activation function to make the distribution match more closely. This technique does not result in a better one-class classification performance when using data from the MNIST dataset, but perhaps could be beneficial when using different datasets. IT also might be interesting for different tasks where the distribution of the encoded target data is important, such as generating new samples by decoding random noise. This does, however, add an additional hyperparameter: the amount of noise to add.

Additional Note When using an additional Tanh activation function directly after the bottleneck layer $\mathbf{z}$, it is also possible to analyze the distribution of the output of this Tanh activation function, instead of analyzing the distribution of $\mathbf{z}$ itself. The expected distribution after applying the Tanh activation is the uniform distribution $\mathcal{U}(-1,1)$. This might be less expensive to compute.

While experimenting with Tanh and ReLU activation functions, it became apparent that the distribution of the encoded target data is much more bell-shaped (and more similar to $\mathcal{N}(0,I)$) when using Tanh activations than when using ReLU activations. Especially when using a regular (non-variational/Wasserstein) autoencoders, this effect is clear, as is shown in the kernel density plots in Figure 8.1. Four autoencoders where trained with ReLU activations (top) and four autoencoders were trained with Tanh activations (bottom) on the Line2D dataset. Both of these types of networks were trained four times, to allow for more significant

results. The networks that use ReLU activations clearly deviate a lot more from the distribution $\mathcal{N}(0,1)$, as much larger values are produced, and looks less bell-shaped.

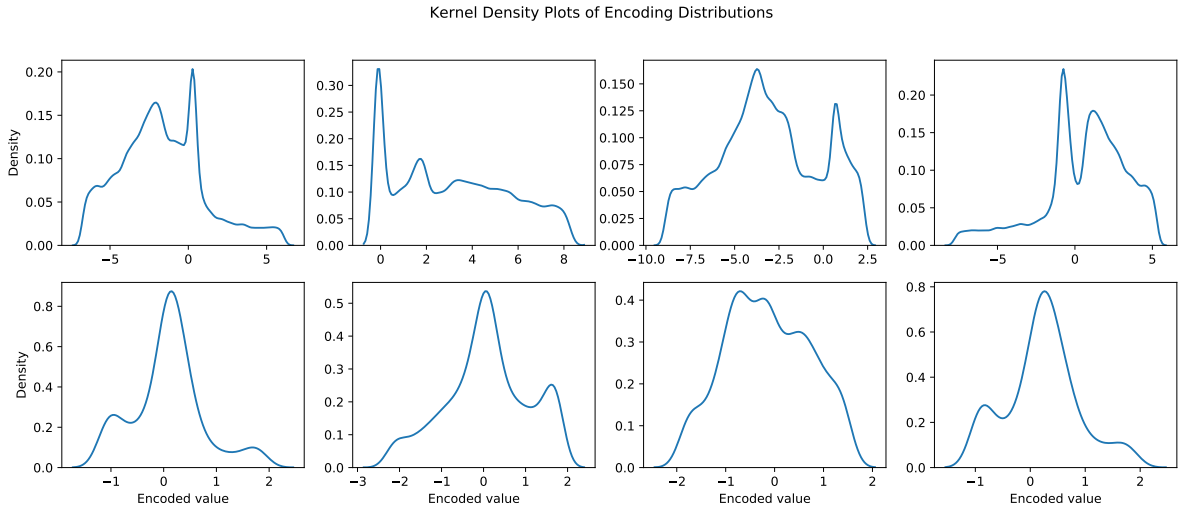


Figure 8.1: Kernel density plots of the distribution of encoded target data from the Line2D dataset. Each figure corresponds to a separately trained network. The four autoencoders on the top row use ReLU activations, while the four autoencoders on the bottom row use Tanh activations.

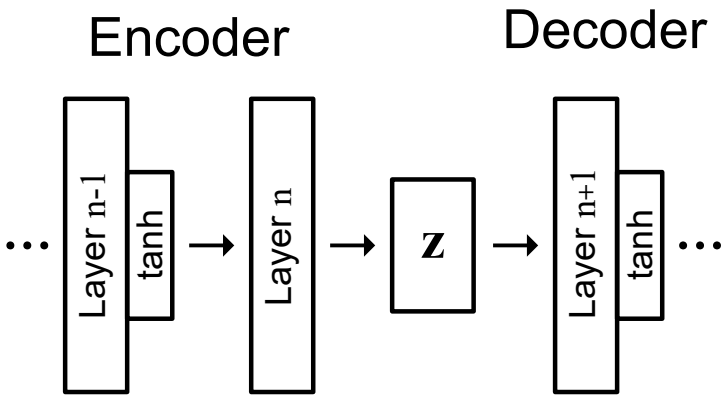


Figure 8.2: Last two layer of the encoder and first layer of the decoder, including their indexes and activation functions (tanh).

The reason why autoencoders with Tanh activations produce a much nicer bell-shaped encoding distribution is hard to justify theoretically, but at least some intuition can be given. To minimize the reconstruction error, the autoencoder should preserve as much information about the input as possible. Each layer of the network is, however, limited by the machine precision, so the output of each layer can only contain a finite amount of information. Maximizing the amount of information that is produced by a layer is equivalent to maximizing the Information Entropy of the output distribution of the layer. The Information Entropy is maximized when the output distribution is uniform over all possible values. At the end of each hidden layer there is a Tanh activation function, which is non-linear and changes the distribution of the output. The last two layers of the encoder and first layer of the decoder are shown in Figure 8.2. Define the vector $\mathbf{o}_i$ to be the output of layer i, which can be computed in terms of the output of the previous layer and the weight and bias of the current layer

$$\mathbf{o}_i = \tanh(\mathbf{W}_i \mathbf{o}_{i-1} + \mathbf{b}_i)$$

Using the layer names as specified in Figure 8.2, this can be written for the first layer of the decoder as

$$\mathbf{o}_{n+1} = \tanh(\mathbf{W}_{n+1}\mathbf{z} + \mathbf{b}_{n+1})$$

As was argued before, the output $\mathbf{o}_{n+1}$ should be uniformly distributed to retain the maximum amount of information about $\mathbf{z}$. The Tanh function limits the range of the output to $[-1,1]$, so $\mathbf{o}_{n+1} \sim \mathcal{U}(-1,1)$ should

hold. The PDF of $\mathbf{o}_{n+1}(x)$ is thus equal to

$$\mathrm{p}_{\mathbf{o}_{n+1}}(x) = \mathcal{U}(-1,1) = \begin{cases} 0.5 & \text{if } x \in [-1,1] \\ 0 & \text{otherwise} \end{cases}$$

Using the change of variable technique for probability density functions, the required PDF of $\mathbf{W}_{n+1}\mathbf{z}+\mathbf{b}_{n+1}$ can be computed. For brevity, we define $\mathbf{y} = \mathbf{W}_{n+1}\mathbf{z}+\mathbf{b}_{n+1}$ and compute the desired distribution for each dimension $y^{(i)}$ of $\mathbf{y}$ separately

$$\begin{aligned} \mathbf{o}_{n+1}^{(i)} &= \tanh(y^{(i)}) \\ \mathbf{y}^{(i)} &= \operatorname{arctanh}(\mathbf{o}_{n+1}^{(i)}) \\ \mathrm{p}_{\mathbf{y}^{(i)}}(x) &= \left|\frac{d}{dx}\tanh(x)\right| \mathrm{p}_{\mathbf{o}_{n+1}}(\tanh(x)) \\ \mathrm{p}_{\mathbf{y}^{(i)}}(x) &= (1-\tanh^2(x))\mathrm{p}_{\mathbf{o}_{n+1}}(\tanh(x)) \end{aligned}$$

The value of $\tanh(x)$ is always in the interval $[-1,1]$, so $\mathrm{p}_{\mathbf{o}_{n+1}}(\tanh(x)) = 0.5$ for all $x \in \mathcal{R}$. This simplifies the equation to

$$\mathrm{p}_{\mathbf{y}^{(i)}}(x) = \frac{1}{2}(1-\tanh^2(x))$$

The resulting PDF is shown in Figure 8.3. The PDF of the distribution $\mathcal{N}(0,1)$ is also shown as reference. The distribution is bell-shaped, symmetric around zero and most of the probability density is close to zero. The probability that a value is sampled from the interval $[-a,a]$ is simple to compute:

$$\begin{aligned} &\int_{-a}^{a} \frac{1}{2}(1-tanh^2(x)) \\ &= \frac{1}{2}[\tanh(x)]_{-a}^{a} \\ &= \frac{1}{2}(\tanh(a)-\tanh(-a)) \\ &= \frac{1}{2}(\tanh(a)+\tanh(a)) \\ &= \tanh(a) \end{aligned}$$

The interval $[-3,3]$, for example, contains $\tanh(3) \approx 0.995$ of the samples.

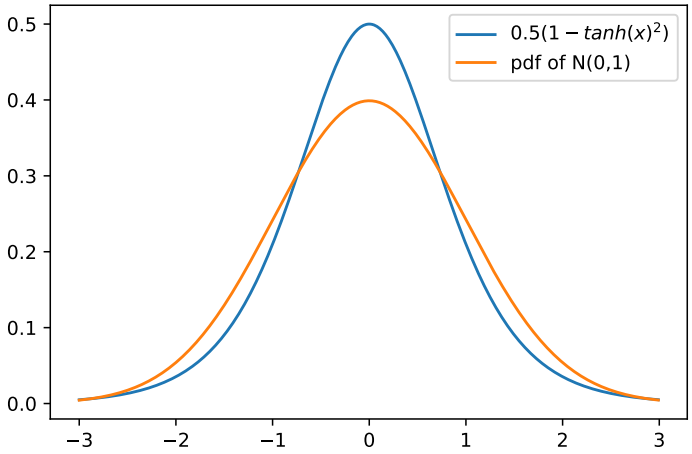


Figure 8.3: The function $0.5(1-\tanh(x)^2)$. This is the PDF of the function $\tanh(x)$ where $x \sim \mathcal{U}(-1,1)$. To provide a frame of reference, the PDF of $\mathcal{N}(0,1)$ is also shown.

Assuming that the output of $\mathbf{W}_{n+1}\mathbf{z}+\mathbf{b}_{n+1}$ is indeed distributed as described above, it is still not trivial to show that, in general, the distribution of $\mathbf{z}$ should be bell-shaped. Only for this specific experiment where $\mathbf{z}$ is a 1 dimensional vector, a further derivation is provided. This makes each output of layer $n+1$ equal to

$$o_{n+1}^{(i)} = \tanh(w_{n+1}^{(i,1)} z + b_{n+1})$$

Thus, the scalar z is scaled by the weight $w_{n+1}^{(i,1)}$ and translated by b_{n+1} to result in the output $y^{(i)}$. In case the distribution of $y^{(i)}$ is bell-shaped, the distribution of z must thus also be bell-shaped. Since $y^{(i)}$ was shown to be distributed according to $\mathrm{p}_{\mathbf{y}^{(i)}}(x) = 0.5(1-\tanh(x)^2)$, z must also be bell-shaped.

This result can, however, not be shown for higher dimensional $\mathbf{z}$ and also relies on the assumption that the distribution of o_{n+1} must encode as much information as possible. This assumption is not desirable for at least two reasons. First, the output o_{n+1} is not the output of the bottleneck layer, so the encoding of o_{n+1} does not necessarily have to be as efficient as possible and thus maximize the information entropy. Second, the encoding might not even have to be as efficient as possible for $\mathbf{z}$ in case the variable $\mathbf{z}$ can store more information than is necessary. The machine precision might be too high, thus allowing to store much more information than is necessary, or the dimensionality of $\mathbf{z}$ might be chosen to be too large.

These issues can be resolved by making two small changes to the network. First, an additional Tanh activation function should be added at the start of the decoder network, directly after $\mathbf{z}$, as is shown in Figure 8.4. Since $\mathbf{z}$ is the bottleneck layer, it is assumed that the amount of stored information should be maximized. Thus, $\tanh(\mathbf{z}) \sim \mathcal{U}(-1,1)$ should hold and, similarly to was shown before, this implies that the PDF of $\mathbf{z}$ must be equal to

$$\mathrm{p}_{\mathbf{z}}(x) = \frac{1}{2}(1-\tanh^2(x))$$

Second, a small amount of random noise can be added to $\tanh(\mathbf{z})$ to limit the amount of information that can be stored. This will make the learned target encoding distribution match the expected distribution $\mathrm{p}_{\mathbf{z}}(x) = 0.5(1-\tanh^2(x))$ more closely.

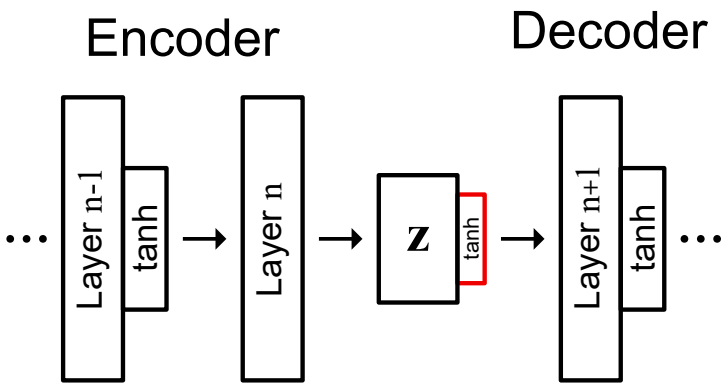


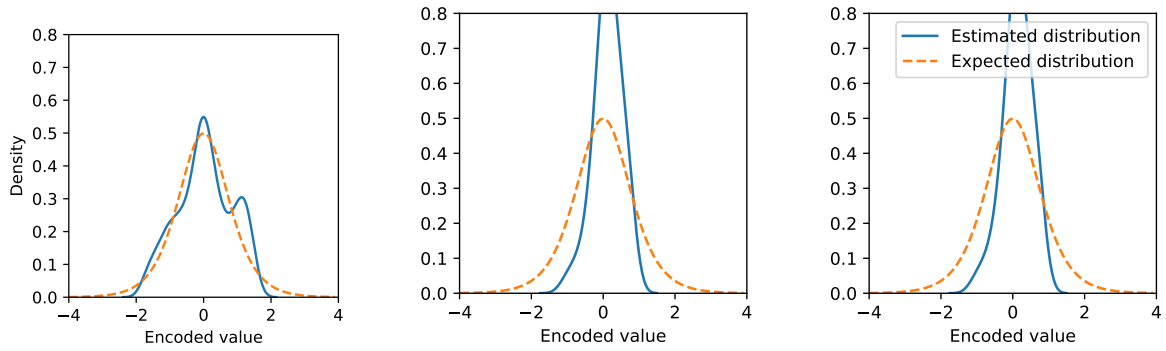
Figure 8.4: Last two layer of the encoder and first layer of the decoder, including their indexes and activation functions (tanh). An additional activation function (tanh) has been added after the block that represents the encoded vector $\mathbf{z}$. This added activation function is highlighted in red.

These theoretical ideas have been experimentally verified and the results are shown in Table 8.1. Varying amounts of noise sampled from $\mathcal{N}(0,\sigma)$ were added to the output of the encoder and the KL divergence and reconstruction error are reported. The reported KL divergence is a measure of the difference between the distribution of the encoded target data and the expected distribution $\mathrm{p}_{\mathbf{z}}(x) = 0.5(1-\tanh^2(x))$. From this experiment it is clear that increasing the amount of noise can decrease the KL divergence and thus make the target encoding distribution more similar to the expected distribution. For the current setup, noise with $\sigma = 0.05$ seems to produce the best results, although it is expected that the optimal amount of noise is problem dependent and also depends on the dimensionality of $\mathbf{z}$. When the dimensionality of $\mathbf{z}$ is increased, more information can be stored, so it might also be necessary to increase the amount of noise.

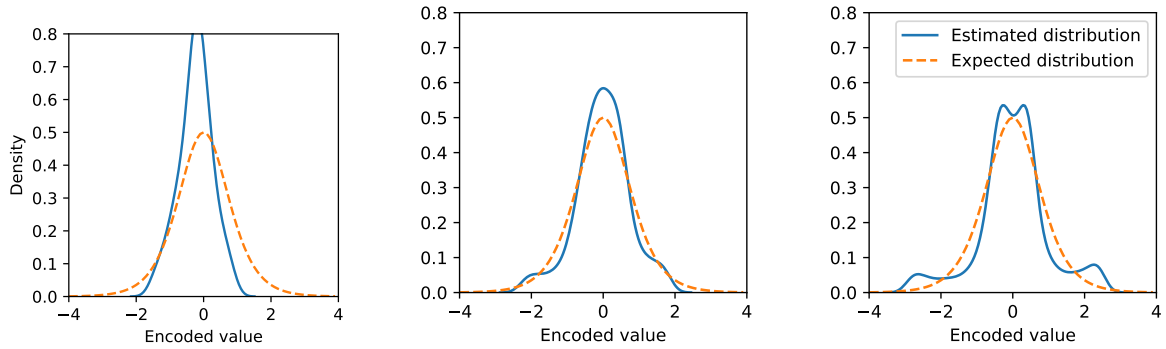
In Table 8.1 the first row shows the results for a network without an additional Tanh function after $\mathbf{z}$. This network has a lower reconstruction error than the others, but interestingly also has a lower KL divergence than some of the networks that do contain an additional Tanh. On a closer inspection, these networks with an additional Tanh have a distribution that is much more peaked close to 0, with little to no values outside of the range $[-1,1]$. Only when adding a larger amount of noise, the network has to make use of a larger portion of the encoding space to ensure that all information is stored with sufficient accuracy. The distributions of encoded target data for networks trained with varying amounts of noise are shown in Figure 8.5.

Noise σ	KL Divergence	Reconstruction Error
No additional Tanh	0.19 ± 0.097	0.017 ± 0.013
0.0	0.44 ± 0.075	0.030 ± 0.015
0.001	0.43 ± 0.069	0.028 ± 0.015
0.01	0.22 ± 0.029	0.030 ± 0.015
0.05	0.048 ± 0.0043	0.053 ± 0.0085
0.1	0.089 ± 0.0076	0.099 ± 0.014

Table 8.1: The KL divergence between the distribution of encoded target data and the expected distribution $0.5(1-\tanh^2(x))$ and the reconstruction error for varying amounts of noise. The noise is sampled from $\mathcal{N}(0,\sigma)$ and added to the output of the Tanh activation that is placed directly after $\mathbf{z}$. The experiment has been repeated 10 times and the resulting averages and standard deviations are reported in the table. The KL divergence has been calculated by using numerical integration, as was explained in Section 4.3.



(a) No additional Tanh (b) Noise = 0.0 (c) Noise = 0.001



(d) Noise = 0.01 (e) Noise = 0.05 (f) Noise = 0.1

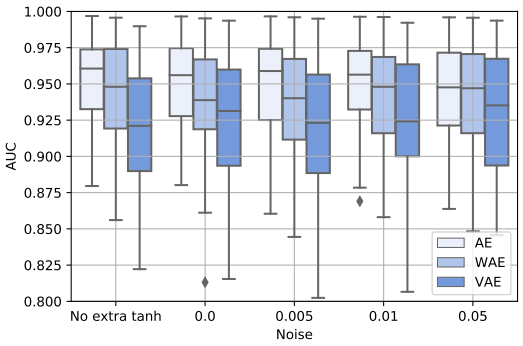
Figure 8.5: Several estimated distributions of encoded target data for various amounts of noise.

When using ReLU activations, the reasoning used before does not hold. ReLU activations map the input according to the function: $\text{ReLU}(x) = \max(0, x)$, so their codomain is equal to $[0, \infty]$. Maximizing the Information Entropy on this domain would result in the expected output distribution $\mathcal{U}(0, \infty)$. This means that an optimal autoencoder that uses only ReLU activations could have an arbitrary encoding distribution. The same trick that was discussed before could, however, be applied to a network that uses ReLU activations as well. The only necessity is to add a Tanh activation function directly after $\mathbf{z}$, as is shown in Figure 8.4, and optionally a small amount of noise can be added to the output of this Tanh activation function in the training phase. The same experiment as was done to create Table 8.1 was also performed with a network that uses ReLU activations. The results are shown in Table 8.2. When comparing the two tables, the results seem roughly similar. Thus, with the added Tanh activation, a network that uses ReLU activations can learn an encoding distribution that approximately matches $p_{\mathbf{z}}(x) = 0.5(1-\tanh^2(x))$ as well.

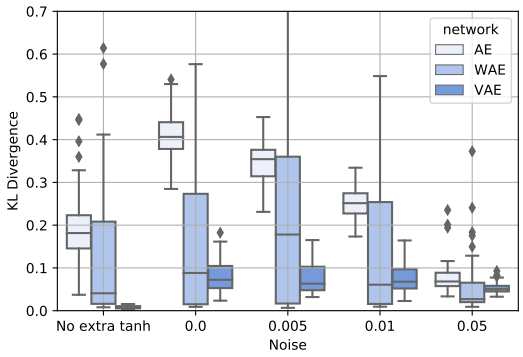
Noise σ	KL divergence	Reconstruction Error
No additional Tanh	3.48 ± 2.53	0.022 ± 0.030
0.0	0.35 ± 0.12	0.013 ± 0.019
0.001	0.31 ± 0.11	0.062 ± 0.16
0.01	0.25 ± 0.091	0.0097 ± 0.015
0.05	0.073 ± 0.028	0.077 ± 0.15
0.1	0.029 ± 0.023	0.11 ± 0.10

Table 8.2: The same experiment as in Table 8.1, but ReLU activations were applied after every hidden layer. Only directly after $\mathbf{z}$ there is still is a Tanh activation function.

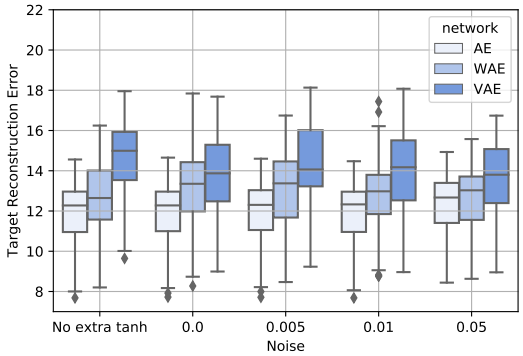
The same behavior is observed while testing with the MNIST dataset. An AE, VAE and WAE are trained with and without the additional Tanh activation function and with varying amounts of noise. The WAE was altered slightly, such that the discriminator uses $p_{\mathbf{z}}(x) = 0.5(1 - \tanh^2(x))$ as expected distribution instead of $\mathcal{N}(0, \mathbf{I})$. For VAEs this is more difficult to do, so the VAE still uses the distribution $\mathcal{N}(0, \mathbf{I})$ as expected distribution. In Figure 8.6a, it is visible that the performance stays similar when adding an additional Tanh activation function. Figure 8.6b shows that for an AE and WAE the distribution of the encoding only improves when adding sufficient noise. For a VAE, adding an additional Tanh activation function does not improve the distribution at all. The reconstruction error, which is shown in Figure 8.6c, shows a similar trend as the AUC. It is also interesting to note that both the AE and WAE perform better at reconstructing than the VAE. This is caused by the KL divergence error term of the VAE being weighted too heavily, which is further investigated in Experiment 8.3.



(a) The AUC for various network types and varying amounts of noise.



(b) The KL divergence between the distribution of encoded target data and the expected distribution $0.5(1 - \tanh^2(x))$ for various network types and varying amounts of noise. For the VAE, the KL divergence was computed with regards to the distribution $\mathcal{N}(0, 1)$.



(c) The reconstruction error for various network types and varying amounts of noise.

Figure 8.6: Results for the MNIST dataset

8.2. Number of Hidden Features

Experiment 8.2 RQs. 2b & 2c

Description The number of hidden features, or equivalently: the number of dimensions of $\mathcal{Z}$, is varied for the MNIST dataset to find the optimum.

Purpose The number of hidden features determines the amount of information that can be encoded. When increasing the number of hidden features, it should thus become easier to learn the identity function $\text{dec}(\text{enc}(\mathbf{x})) = x$ and it is expected that the reconstruction error will decrease. When too many hidden features are used, it might not be necessary for the network to learn efficient compression and decompression functions, possibly causing the non-target data to be reconstructed with low errors as well. On the other hand, using too few hidden features could cause the reconstruction of target data to become inaccurate.

Conclusion For the MNIST dataset it seems most efficient to use 8 or slightly more neurons. When varying the number of neurons between 8 and 32, only a slight variation in performance is observed. This means that it should be simple to optimize the number of hidden features, since using slightly more features than necessary does not seem to impact the performance much.

This experiment uses the best network that was found in Experiment 7.4, that is detailed further in Appendices A.2, A.3 and A.4. When disregarding the bottleneck layer, the layer containing the fewest neurons has 32 neurons, thus any size between 1 and 32 for the bottleneck layer can be tested. Using larger sizes is not possible with this network, since then the bottleneck layer would not actually be the bottleneck anymore.

Networks with bottleneck layer sizes ranging from 1 to 32 have been trained and the resulting performance are shown in Figure 8.7. Pairs of two digits from the MNIST dataset were chosen as target class and each network has been trained on all 45 possible digit pairs. For the AE and WAE networks, the ideal number of neurons seems to be around 8 or slightly higher. When using much more than 8 neurons, the performance seems to degrade very slowly.

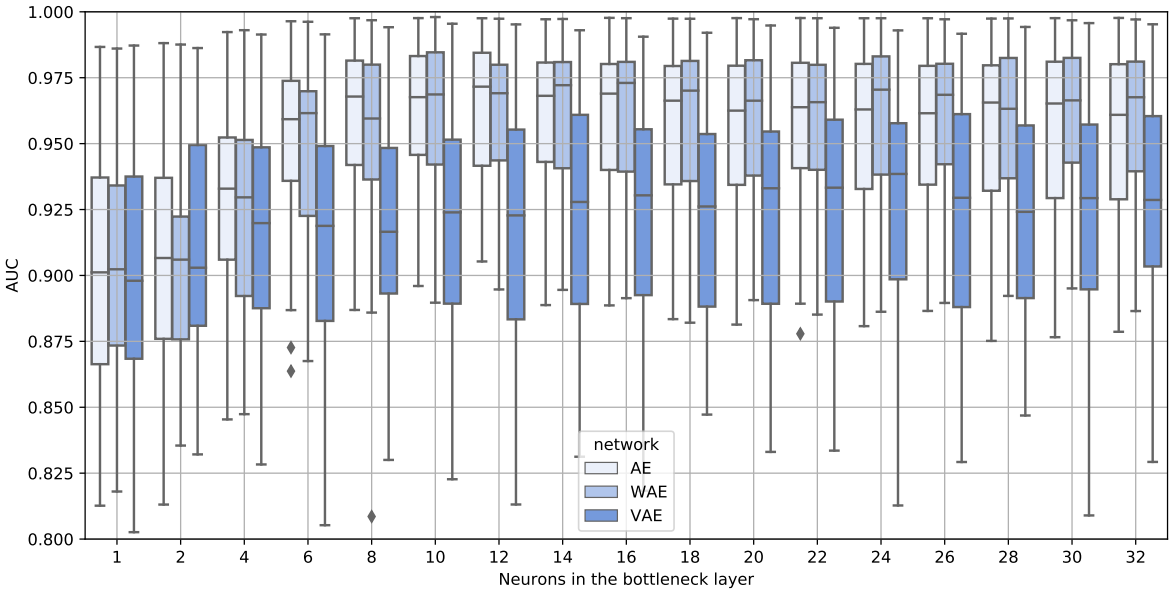


Figure 8.7: The performance of AEs, WAEs and VAEs, using varying sizes for the bottleneck layer $\mathbf{z}$.

The reconstruction error of the target class for AE and WAE networks does seem to keep decreasing when adding more neurons, albeit slowly, as is shown in Figure 8.8. Since the performance decreases slowly when increasing the number of neurons beyond 8, this must mean that the reconstruction error of the non-target class decreases as well when using more neurons.

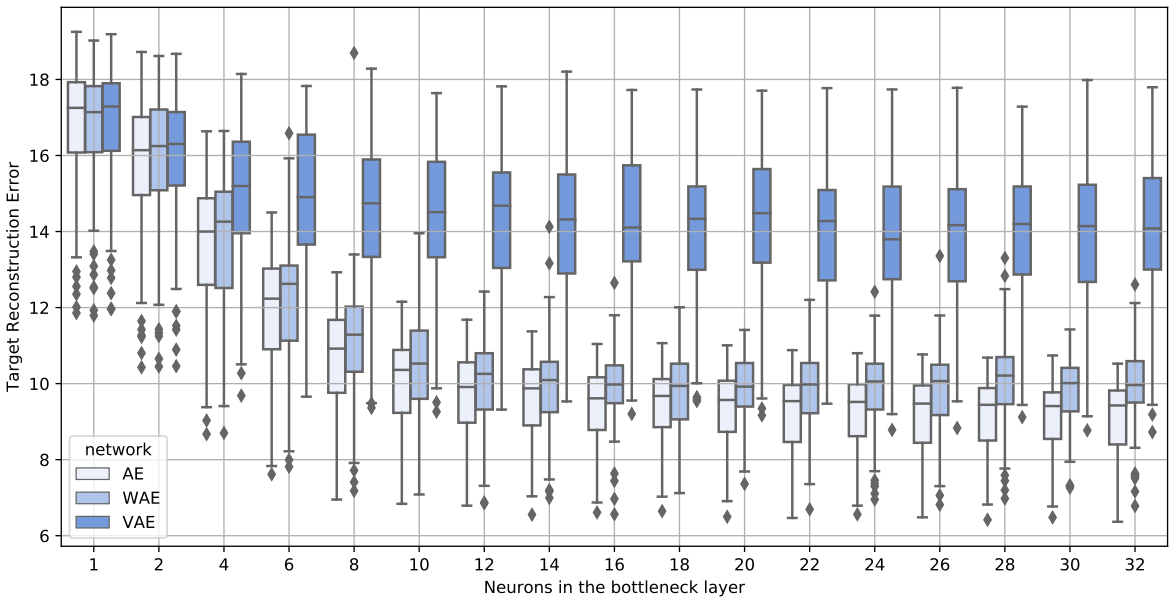


Figure 8.8: The target reconstruction error of AEs, WAEs and VAEs for varying numbers of sizes for the bottleneck layer $\mathbf{z}$.

In both Figure 8.7 and Figure 8.8, the VAE does not improve significantly when more than 4 neurons are used. This occurs because the KL divergence term in the error function of the VAE starts dominating the objective and avoids that the network makes use of all hidden features. This observation is analyzed more in depth in Experiment 8.3.

8.3. Weighting the KL divergence term (VAE)

Experiment 8.3 RQ. 2b

Description A VAE has two objectives: to minimize the reconstruction error and to minimize the KL divergence. In this experiment, the weight of the KL divergence objective is varied and the resulting behavior is observed.

Purpose Correctly weighting the objectives of a VAE might be important for the performance.

Conclusion Increasing the weight of the KL divergence objective causes the encoding of each sample to become more similar to $\mathcal{N}(\mathbf{0},\mathbf{I})$. Thus, less information about the input sample $\mathbf{x}$ can be encoded and the reconstruction error is expected to increase. When using a VAE for one-class classification, a higher weight for the KL divergence term resulted in a lower classification performance on the MNIST dataset. This result might not hold for all datasets, but at least for the MNIST dataset, a regular AE or a VAE with a small weight for the KL divergence objective (around 0.1-0.5) should be used.

Additionally, it was shown that VAEs do not make full use of all hidden features that are available when the weight of the KL divergence objective is large, or when the number of hidden features is large. When 12 hidden features are available to encode an input sample $\mathbf{x}$ and the weight of the KL divergence objective is set to 1, it was shown that only 3 or 4 out of 12 features are actually used to encode information. The number of features that is used was also observed to not be consistent. Simply restarting the training process can result in a trained VAE that makes use of a different number of hidden features. This could make it difficult to optimize VAEs and makes their performance inconsistent. For this reason, WAEs are preferred over VAEs for the remained of this report.

The weight for the KL divergence objective of a VAE is chosen from $\{0.0, 0.1, 0.5, 1.0, 2.0, 10.0\}$, while the reconstruction error objective has a constant weight of 1.0. For each of these weights, a VAE is trained and evaluated using pairs of two digits from the MNIST dataset as target class. The resulting one-class classifica-

tion performance is shown in Figure 8.9. Increasing the weight of the KL divergence term is shown to have a negative effect on the performance. Setting the weight equal to 0.0 actually results in one of the best performing models. When the weight of the KL divergence objective is set to 0.0 it does not influence the error, so only the reconstruction error objective is optimized. This makes the VAE equivalent to a regular AE. Thus, for one-class classification purposes, a VAE does not perform better than a regular AE on the MNIST dataset.

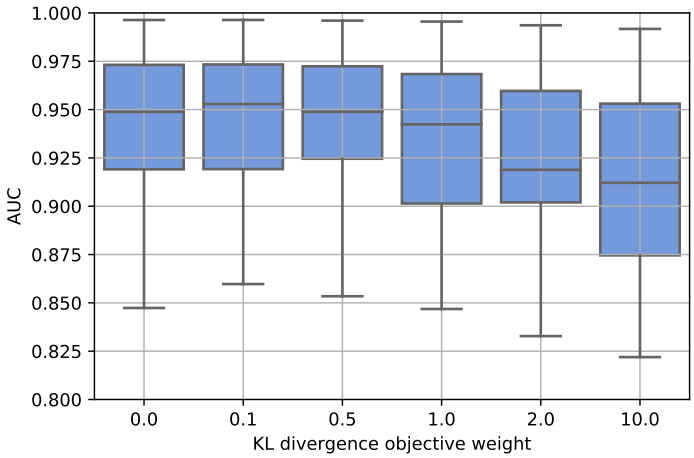


Figure 8.9: The AUC of a VAE, using pairs of digits from the MNIST dataset as target class. Different weights for the KL divergence objective are shown on the x-axis. The VAE uses 6 hidden features for the encoding.

Another noteworthy property of VAEs was observed during the execution of this experiment. When the weight of the KL divergence objective is increased, some of the features in $\mathbf{z}$ are always sampled from a normal distribution with a mean close to 0 and standard deviation close to 1, for all input samples $\mathbf{x}$. This means that these features are not used to encode information about the original input sample $\mathbf{x}$. Thus, a VAE does not always 'make use' of all features in $\mathbf{z}$ to encode $\mathbf{x}$. This phenomenon is also observed when increasing the number of features in $\mathbf{z}$, while keeping the weight of the KL divergence objective constant. This also explains the behavior observed in Experiment 8.2, where the performance of a VAE does not increase as much as the performance of an AE and WAE when increasing the number of hidden features.

To illustrate this, a VAE with 12 hidden features and the KL divergence objective weighted at 1.0 is trained on the MNIST dataset. Digits 3 and 4 were used as target class. When encoding data, a VAE produces a mean μ and standard deviation σ for each hidden feature and samples it from $\mathcal{N}(\mu,\sigma^2)$. All data from the target class is encoded and the average absolute value of μ and σ are reported in Table 8.3 (on page 48). Rows marked in red correspond to hidden features that are always encoded identically and thus do not provide any information regarding the input $\mathbf{x}$. This VAE apparently only makes use of hidden features z_0, z_2, z_3 and z_7 to encode information. This result is, however, not consistent. After restarting the training process, the new trained VAE made use of only 3 features.

To prove that the 'unused' features that are marked in red in Table 8.3 (on page 48) have little influence on the output, the weights in the first layer of the decoder were analyzed. The average absolute weight by which the 'unused' hidden features are multiplied lies between 0.0028 and 0.0043. For the other hidden features, this value lies between 0.25 and 0.38. Thus, the 'unused' hidden features have approximately 100 times less influence on the output of the decoder compared to the 'used' hidden features.

It is also interesting to note that the learned standard deviation σ in Table 8.3 (on page 48) does not vary much for different input samples $\mathbf{x}$, as the standard deviation of σ lies between 0.01 and 0.02. It does not seem necessary to compute σ dependent on $\mathbf{x}$, thus a single trainable parameter could be used instead.

Feature	Average value of $\lvert\mu\rvert$	Average value of σ
z_0	0.80 ± 0.62	0.11 ± 0.02
z_1	0.01 ± 0.01	0.99 ± 0.01
z_2	0.81 ± 0.64	0.04 ± 0.01
z_3	0.81 ± 0.62	0.06 ± 0.01
z_4	0.01 ± 0.01	1.00 ± 0.01
z_5	0.01 ± 0.01	1.00 ± 0.01
z_6	0.02 ± 0.02	0.98 ± 0.01
z_7	0.94 ± 0.58	0.03 ± 0.01
z_8	0.01 ± 0.01	0.98 ± 0.01
z_9	0.01 ± 0.01	0.99 ± 0.01
z_{10}	0.01 ± 0.01	0.99 ± 0.00
z_{11}	0.01 ± 0.01	0.99 ± 0.01

Table 8.3: When a sample is encoded, a VAE samples each hidden feature from the distribution $\mathcal{N}(\mu,\sigma^2)$, where μ and σ are produced by the encoder. This table reports the average size of μ and σ for each hidden feature, averaged over all data in the training dataset.

9

Case Study: MNIST Digits 3 & 4

A case study is performed, where digits 3 and 4 from the MNIST dataset are used as target data. This allows for a more detailed analysis, although the results are not shown to hold in general. This chapter focusses on research question 2b: *"Does the value of the encoded vector* $\mathbf{z}$ *of an AE have any interpretable meaning and how does the dimensionality of* $\mathbf{z}$ *influence the classification performance"*

9.1. Reconstruction Error per Digit

Experiment 9.1 — RQ. 2d

Description The reconstruction errors for each of the target and non-target digits is compared.

Purpose The target digits 3 and 4 should have a low reconstruction errors, while other digits should have significantly higher errors. This experiment analyses the errors that are made to expose potential weaknesses.

Conclusion An autoencoder that is trained on digits 3 and 4 is able to reconstruct digits 3 and 4 better than the non-target digits on average. Some non-target digit samples are, however, also reconstructed well. Especially non-target digits 1 and 9 are often reconstructed well, causing these non-target samples to not be separable from the target class. Digit 9 is often reconstructed well, because its shape is similar to the shape of target digits 3 and 4. This type of error is to be expected, since outliers that are more similar to the target class are harder to separate.

For digit 1, a different type of error is observed. Digit 1 does not look similar to digit 3 or 4, but by squeezing digit 3 or 4, an image with many similar pixels can be constructed. The reason why such squeezed images can be reconstructed is further analyzed in Experiment 9.2.

Additional Note This experiment has also been performed with a WAE, but all results are similar, so only the results of an AE are reported.

This experiment uses an autoencoder with 8 neurons in the bottleneck layer, an additional Tanh activation function and noise sampled from $\mathcal{N}(0, 0.05)$, as was discussed in Experiment 8.1.

The resulting reconstruction errors per digit are shown in Figure 9.1. The target class is reconstructed with the lowest error on average, but many digit 1 and 9 samples are reconstructed with similar errors. When using this autoencoder as one-class classifier, outlier digits 1 and 9 can thus often not be separated well from the target class.

To investigate why digits 1 and 9 are also reconstructed with low errors, several original and reconstructed samples are shown in Figure 9.2. For digit 9, all reconstructed images look like a 3 or a 4. The reason why digit 9 is reconstructed well is because its shape is often similar to digits 3 and 4, causing the reconstruction to have a lot of pixels in common with the original.

The reconstructions of digit 1 that are shown in Figure 9.2c do not look similar to digits 3 or 4. Some reconstructions look like squeezed versions of digit 3. The shape of the reconstructions does not look similar to the original digit 1, but the images do have many pixels in common, resulting in a low reconstruction error.

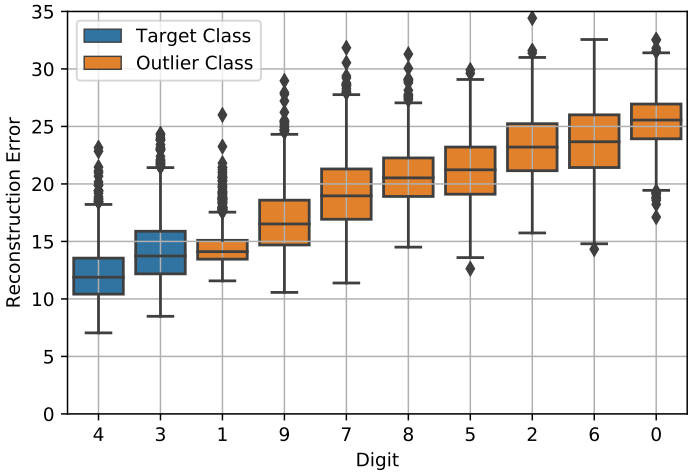
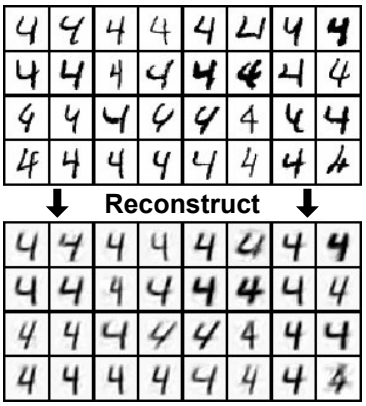
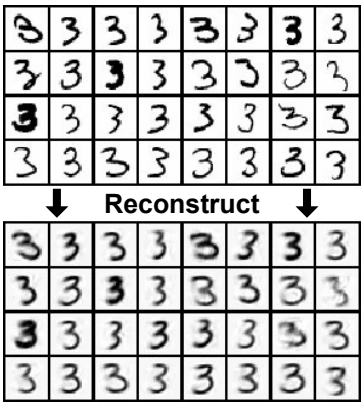


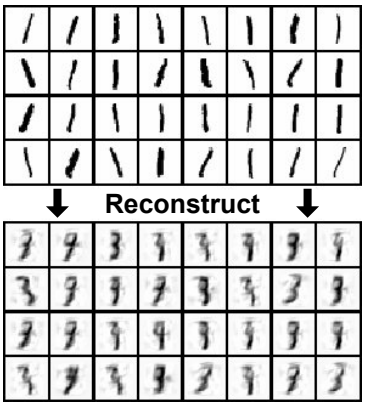
Figure 9.1: Reconstruction errors of an AE that is trained on digits 3 and 4.



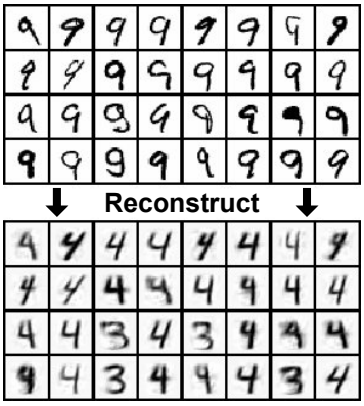
(a) Digit 4



(b) Digit 3



(c) Digit 1



(d) Digit 9

Figure 9.2: Original image (top) and their reconstructions (bottom), for digits 4, 3, 1 and 9. Digits were reconstructed by encoding and decoding the original image with an autoencoder that was trained on digits 3 and 4.

9.2. Comparing Target and Outlier Encodings

Experiment 9.2

RQs. 2b & 2d

Description The encoding distribution of the target data and misclassified non-target data are compared for WAEs.

Purpose By comparing the encoding of target and misclassified non-target data, it might be possible to find patterns that could be used to improve the performance.

Conclusion The encoding of non-target digit 9 is most of the time similar to the encoding of digit 4 samples. The encoding does not seem easily separable from the target data encodings. The encodings of non-target digit 1 lie in the 'transition' area between digits 3 and 4 and also do not seem easy to separate from the encodings of target data.

This experiment uses a WAE with 8 neurons in the bottleneck layer, an additional Tanh activation function and noise sampled from $\mathcal{N}(0, 0.05)$, as was discussed in Experiment 8.1.

After visualizing the encoded target data, it became apparent that one half of the encoding space is occupied by samples of digit 3 and the other half by digit 4. This is shown in Figure 9.3, where feature 0 (x-axis) and 3 (y-axis) of the encoded target data are plot. Feature 3 can be used to separate the encodings from digits 3 and 4. This is quite remarkable, a WAE is apparently able to learn an encoding where a single feature can be used to determine the class label. After running the experiment several more times, this did not occur in general, but in all observed cases the encodings of digits 3 and 4 were approximately linearly separable in the encoded space.

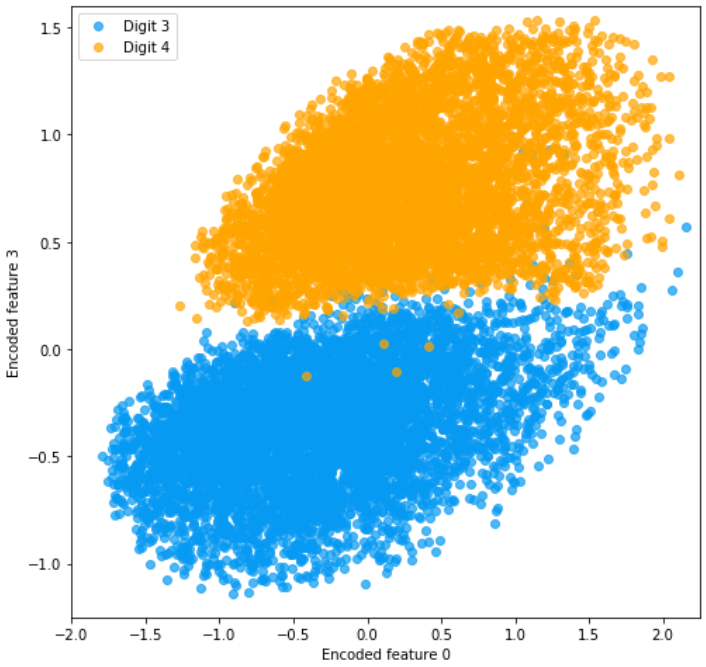
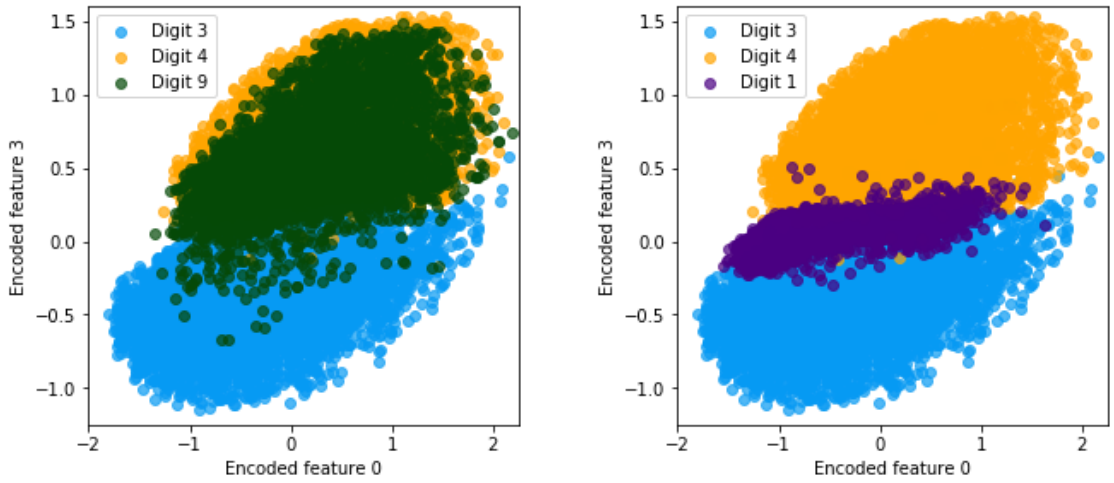


Figure 9.3: Encoded target samples from the training set. A WAE with 8 neurons in the bottleneck layer was trained on digits 3 and 4. Two of these 8 encoded features are shown in the plot.

In Figure 9.4, the encodings of non-target digits 1 and 9 are also shown. These two non-target digits were shown in Experiment 9.1 to have low reconstruction errors, causing them to be difficult to separate from the target data.

The encodings of non-target digit 9 in Figure 9.4a show that digit 9 is most of the time encoded as a 4. The encodings are difficult to separate from the encodings of target data, so it is most likely not possible to improve the one-class classification performance for digit 9 outlier samples by using its encoding.

The encodings of non-target digit 1 are shown in Figure 9.4b. The encodings of digit 1 mostly lie in-between the encodings of digit 3 and 4. This is the area where the encodings 'transition' between producing digit 3 and 4, which is further explored in Experiment 9.4. The encodings of digit 1 might be separable from the encodings of digits 3 and 4, although this is not an easy task. Using the theory discussed in Section 3.2, such that samples from low probability density areas of the expected distribution are excluded, does however not work. The probability density function of the expected distribution is $p_{\mathbf{z}}(x) = 0.5(1-\tanh^2(x))$ (see Experiment 8.1 for the derivation). This distribution has a higher probability density near 0 and the digit 1 outlier samples in Figure 9.4b actually lie closer to 0 in the direction of feature 3 than the target samples, thus these outliers can lie in a high probability density region.



(a) Shows encodings of non-target digit 9

(b) Shows encodings of non-target digit 1

Figure 9.4: Encoded target and non-target samples.

9.3. Improving performance by predicting the reconstruction error

Experiment 9.3

RQs. 2 & 2b

Description The observation made in Figure 9.1 (on page 50) that digit 4 has a lower reconstruction error than digit 3 and the observation in Figure 9.3 (on page 51) that shows that the encodings of digit 3 and 4 lie far apart in the encoded space sparked an idea that is analyzed in this experiment. Perhaps it is possible to predict the reconstruction error of a test sample based on its encoding.

Purpose Some target samples are easier to reconstruct than others, leading to differences in reconstruction errors. It might be possible to improve the performance by measuring the difference between the reconstruction error of training samples that have a similar encoding and the reconstruction error of the test sample. For example, in Figure 9.4a, it was shown that digit 9 is most of the time encoded similarly to digit 4. The reconstruction error of digit 9 is a lot higher than the reconstruction error of digit 4, as is shown in Figure 9.1, so it should be possible to distinguish digit 9 as an outlier when comparing to the expected reconstruction error of samples with a similar encoding. Comparing the reconstruction error of digit 9 samples to the reconstruction error of all samples from the target class would cause considerably more misclassifications, since the reconstruction error of target digit 3 is much higher than that of digit 4.

Conclusion It is possible to increase the performance by predicting the reconstruction error for samples, based solely on their encoding. This was done as follows. When determining if a test sample $\mathbf{x}$ belongs to the target class, it is first encoded. Next, the 25 nearest neighboring target encodings in the training data are found and their reconstruction error is averaged. This becomes the predicted recon-

struction error. The difference between the actual and predicted reconstruction error is thresholded to result in a one-class classifier.

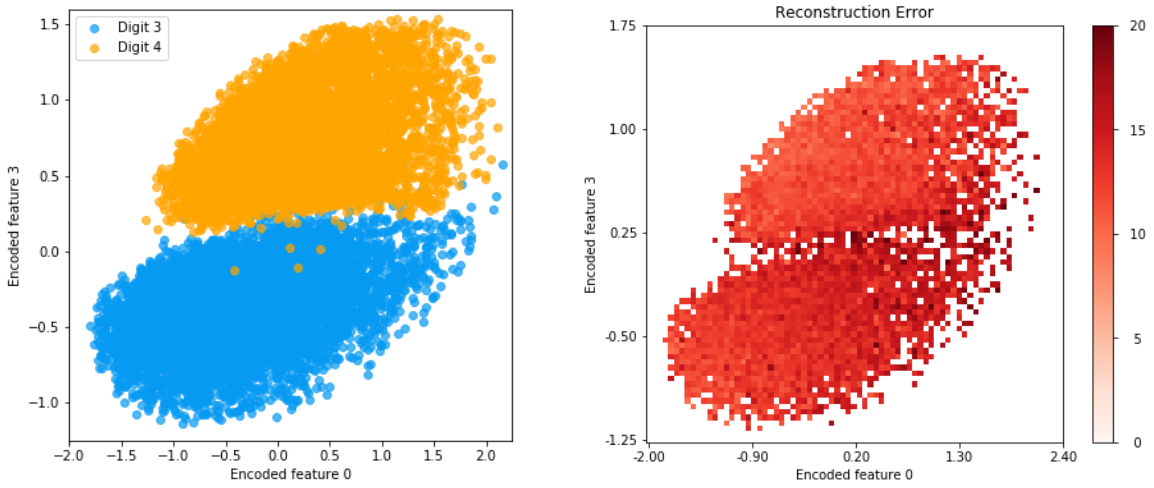
This method can improve the performance, because the encoding generalizes properties of the original sample and certain properties can make samples more difficult to reconstruct. By predicting the reconstruction error based on the encoding, it is possible to rectify the difference in reconstruction difficulty for different types of samples from the target dataset, resulting in a better classifier.

It is probably possible to get better results by predicting the reconstruction error with a more elaborate method that simply finding the 25 nearest neighbors. This is left as a topic future research.

Additional Note A further extension of this method is discussed as in Experiment 11.2. There, an autoencoder is trained on an unlabeled dataset that contains both target and non-target samples. This autoencoder is used to predict the expected reconstruction error for the test samples and the results show that this information can dramatically increase the performance.

The class labels of encoded target samples are shown in Figure 9.5a and the reconstruction errors are shown in Figure 9.5b. The colors in Figure 9.5b represent the reconstruction error and were computed by splitting the shown area into a grid consisting of 10,000 squares and for each square, the color is determined by averaging the reconstruction errors of the samples that lie within the square. All squares that do not contain any training samples are white.

In Figure 9.5b, it is visible that samples in the bottom half of the plot have slightly higher reconstruction errors. These are samples of digit 3. When looking exclusively at the encodings of digit 3, the samples closer towards the bottom left also seem to have lower reconstruction errors, thus for samples from the same digit it is also possible to observe differences in the reconstruction quality, based on their encoding.



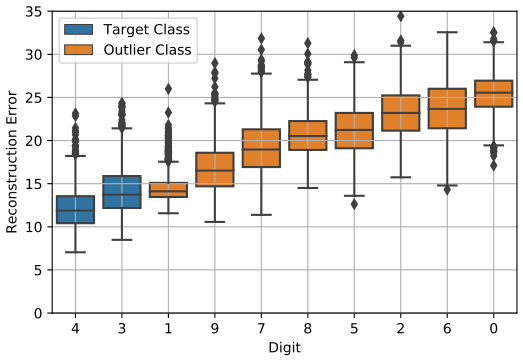
(a) Class labels of encoded target digits.

(b) Reconstruction errors of encoded target digits.

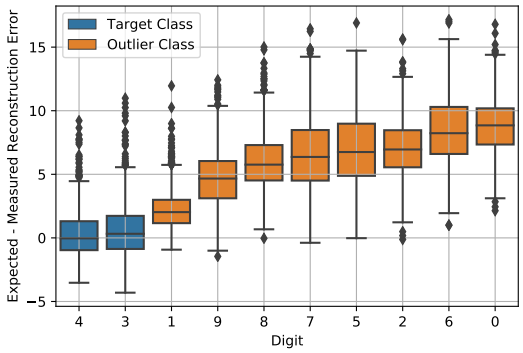
Figure 9.5: Encoded target samples from the training dataset.

A simple test was done to determine if the classification performance can be improved by predicting the reconstruction error based on the encoding of a sample. The expected reconstruction error is estimated by averaging the reconstruction error of the 25 closest neighbors from the training dataset in the encoded space. The error per sample is then computed by subtracting the expected reconstruction error from the actual reconstruction error. For samples from the target class this value is expected to be close to zero and for non-target samples the error is expected to be higher. This results in errors per digit as shown in Figure 9.6b. As a comparison, the reconstruction errors are shown in Figure 9.6a.

Both of the target digits in Figure 9.6b indeed have an error around 0 and the errors of digit 9 samples are now much easier to separate from the target class. The performance of the resulting classifier has also improved. Whereas an AUC of approximately 0.91 is obtained when using the reconstruction errors that are reported in Figure 9.6a, by using the errors in Figure 9.6b the AUC becomes 0.94.



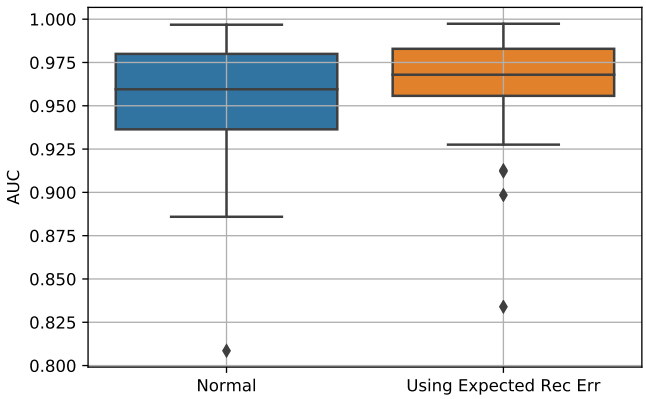
(a) The reconstruction error per digit.



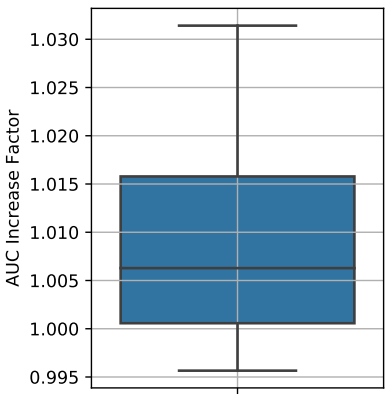
(b) The reconstruction error minus the expected reconstruction error. The expected reconstruction error is computed by averaging the reconstruction error for the 25 closest training samples in the encoded space.

Figure 9.6: Errors of a WAE that is trained on digits 3 and 4.

To test if making use of the predicted reconstruction error can improve the performance in general, the same experiment has been performed for multiple target classes for the MNIST dataset. The results are shown in Figure 9.7. In Figure 9.7a, the AUC is shown to improve slightly when making use of the expected reconstruction error. For each problem the relative increase in performance is shown in Figure 9.7b. For the majority of the problems, the performance improves when making use of the predicted reconstruction error.



(a) The AUC when not making use of the expected reconstruction error (left) and when making use of the expected reconstruction error (right).



(b) The relative increase in performance per problem, when making use of the expected reconstruction error.

Figure 9.7: Comparison between the performance (AUC) achieved when making use of only the reconstruction error and when additionally making use of the expected reconstruction error. The expected reconstruction error is determined by encoding a test sample and averaging the reconstruction error of training samples that have similar encodings. For each of the 45 possible combinations of 2 digits, a model has been trained using the 2 digits as target class and the remaining digit as non-target class.

9.4. Interpreting the Meaning of the Encoded Features

Experiment 9.4 RQ. 2b

Description The meaning of the encoded features is analyzed by varying each feature individually and visualizing the output.

Purpose It is expected that concepts found in the original dataset are generalized by an autoencoder and efficiently encoded in a lower dimensional space. This experiment analyses if any interpretable

meaning can be associated with the features, such as rotation or skew.

Conclusion Each encoded feature was found to have an interpretable meaning and transitions in the output are smooth when varying the encoded features. Some of the observed visual effects when varying a single feature are: change in skew, stroke width, if digit 3 or 4 is generated. Most features were also found to influence a single type of transformation. This shows that autoencoders are able to generalize concepts in the training dataset well and that the encoded features can even have human-interpretable results.

As was discovered in Experiment 9.2, the trained model uses a single feature to encode if the original sample is a 3 or a 4. In Figure 9.8, this feature is varied between -0.5 and 0.5, while keeping all other features constant at 0. Besides influencing if a 3 or 4 is generated, the feature also causes slight additional variations. For example, the 3 generated when the feature is set to -0.5 has a more circular bottom half than when the feature is set to -0.125.

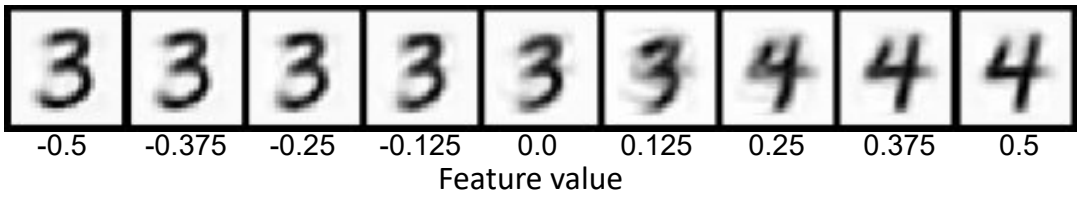


Figure 9.8: Varying encoded feature 3, while keeping the other features constant at 0.

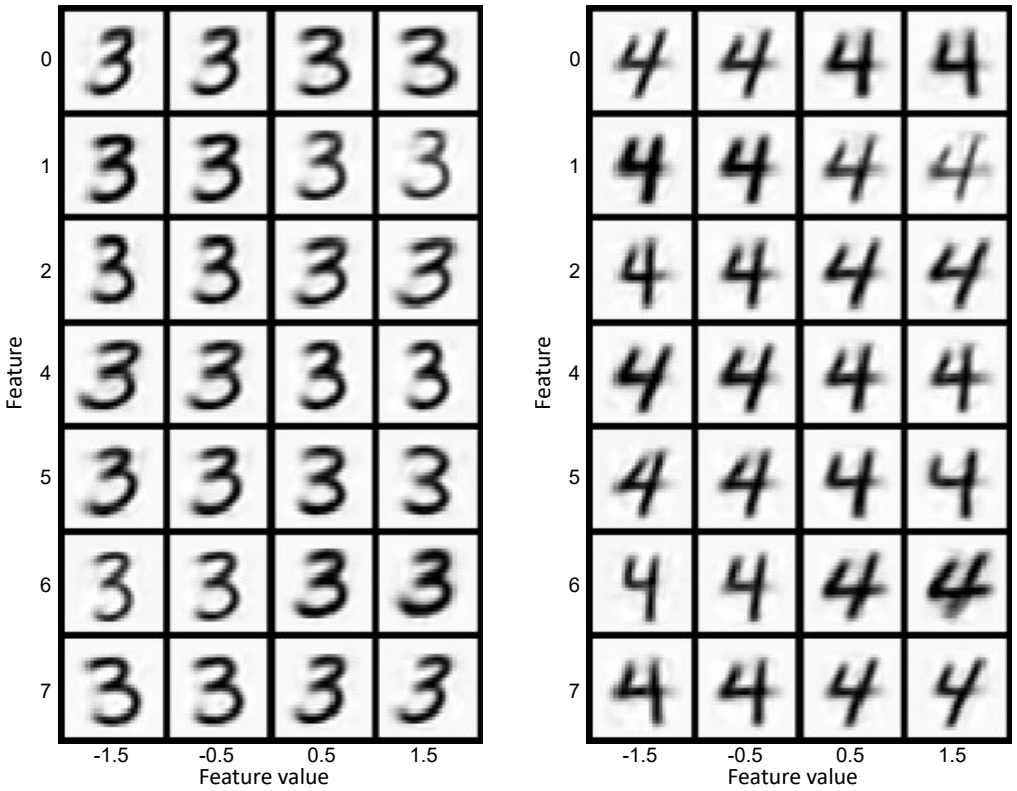
All remaining features are varied in Figure 9.9. One feature is varied at a time, while the remaining features are set to 0. Only feature 3, which influences whether the generated digit is a 3 or a 4, is set to -0.5 for the generated samples in Figure 9.9a and set to 0.5 for the samples in Figure 9.9b. This makes it possible to analyze the effect each feature has separately for digits 3 and 4.

The differences might be hard to determine from these static images, so the perceived property that is influenced by each feature is described in Table 9.1. These results were determined visually from an animation where the features are slowly varied. Each feature seems to have roughly the same type of effect on both digits. Note that only the most prominent observations are described, more changes can be observed, but these are smaller and hard to describe.

Feature	Effect on output
0	Skew
1	Line thickness
2	Width
3	If digit 3 or 4 is generated
4	Width
5	Size of the top half
6	Line thickness
7	Skew

Table 9.1: Perceived property that is influenced by each encoded feature.

Combining the 8 features listed in Table 9.1, various types of digits can be generated. Sampling each feature from the distribution $\mathcal{N}(0,1)$ results in samples such as the ones that are shown in Figure 9.10.



(a) Feature 3 is fixed at -0.5, such that digit 3 is generated.

(b) Feature 3 is fixed at 0.5, such that digit 4 is generated.

Figure 9.9: Effect on the output when varying a single encoded feature, while keeping the other features constant.

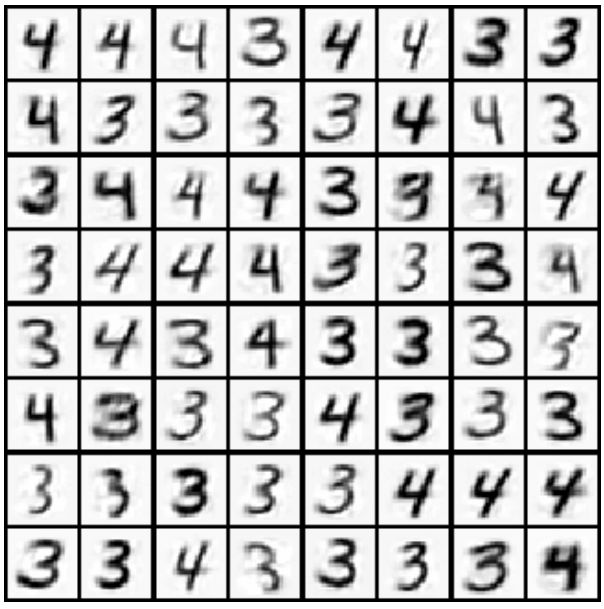


Figure 9.10: Resulting samples when decoding noise sampled from $\mathcal{N}(0,1)$ with a WAE that is trained on digits 3 and 4.

10

Analyzing Potential Problems of Autoencoders

Potential problems and limitations of the presented method are experimentally analyzed. The experiments mostly involve decreasing the quality, or amount of training data. This chapter focusses on research questions: 2d *"What are the strengths/weaknesses and what types of errors are to be expected?"* and 2e *"Are these methods resilient against outliers, or noise in the training data?"*

10.1. Missing Training Data

Experiment 10.1

RQ. 2f

Description Parts of the training data from the LINE-2D dataset are removed and the decision boundary is analyzed.

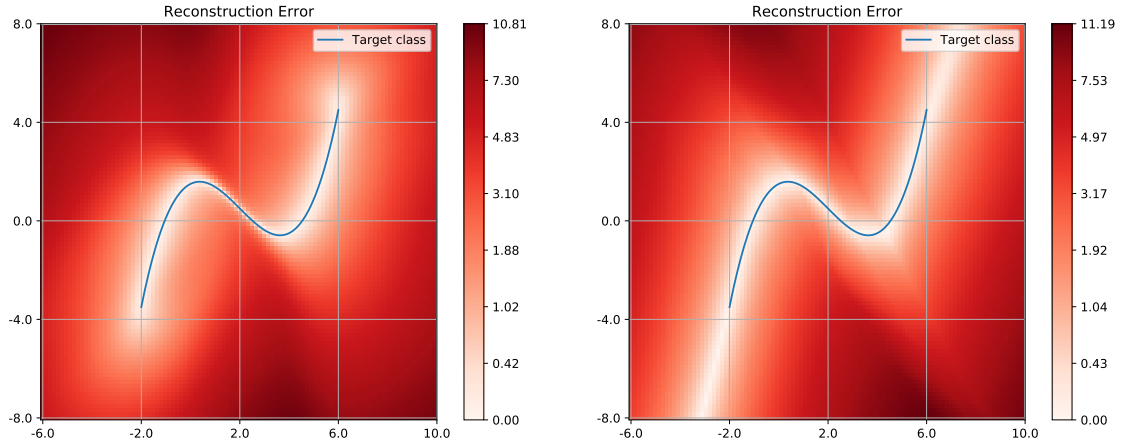
Purpose In practice, the target dataset might not be sampled properly and part of the target distribution might not be sampled at all. It is important to know how the classifier behaves in these scenarios.

Conclusion There is a major difference between autoencoders that use Tanh activations and ones that use ReLU activations. When much data is missing, the Tanh network will not learn to reconstruct samples from the area where no training data is available. This makes Tanh networks unable to extrapolate and interpolation is only performed when there is a small gap of missing data. Networks that use ReLU networks can linearly extrapolate far away from the training data and also interpolate linearly. The desired behavior is problem-dependent. When the target class consists of separate groups of data, the Tanh network where no interpolation is performed might be preferred, while the ReLU network might be preferred when a sparse sampling of the target data is available and a lot of interpolation is desired. Similarly, extrapolation could also either be desired or unwanted behavior. In the given example, the ReLU network does show that it extrapolates its decision boundary far from the training data, so this might cause unexpected behavior, where samples that lie far from the training data are also classified as target data.

Additional Note The experiment has only been performed on an artificial low-dimensional dataset, because for higher dimensional datasets it is difficult to remove samples such that extrapolation or interpolation is needed and interpreting the results would be difficult as well.

Two scenarios for missing data can be distinguished. The pattern found in the training data has to either be (1) extrapolated, or (2) interpolated. Extrapolation is needed when samples near the 'edge' of the target distribution are missing, while interpolation is needed when samples in the 'middle' of the target distribution are missing.

A case where extrapolation occurs has already been seen before in Experiment 7.3. There it was shown that a network that uses Tanh activations does not extrapolate, while a network that uses ReLU activations does linearly extrapolate. This also holds for the reconstruction errors, as is shown in Figure 10.1. The reconstruction errors near the target class are shown for a network that uses Tanh activations (left) and a network that uses ReLU activations (right). When basing a decision boundary on the reconstruction errors of the ReLU network in Figure 10.1b, samples that are linearly extrapolated at either end of line segment that constitutes the target class, will also be classified as target data, since these samples have a small reconstruction error. For the network that uses Tanh activations in Figure 10.1a, the decision boundary would not extrapolated.



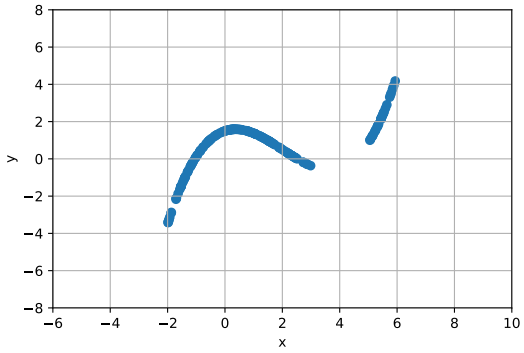
(a) Network with Tanh activations.

(b) Network with ReLU activationss.

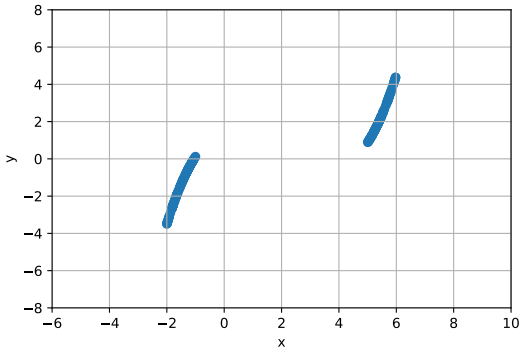
Figure 10.1: Reconstruction errors near the target class of the LINE-2D dataset for 2 networks with as only difference that one uses Tanh activations, while the other uses ReLU activations.

To analyze the behavior when interpolation is needed, a segment is removed from the center of the LINE-2D dataset. Two scenarios are tested. Samples from the training set of both scenarios are shown in Figure 10.2. In the first scenario, samples where $x \in [3,5]$ are removed and in the second scenario sample where $x \in [-1,5]$ are removed.

The resulting reconstruction errors for the first scenario are shown in Figure 10.3. Both the autoencoder that use Tanh and ReLU activations manage to interpolate the results well. The resulting reconstruction errors for the second scenario are shown in Figure 10.4. In this scenario, much more data is missing. Both the network that uses Tanh and the one that uses ReLU activations are unable to learn to reconstruct all samples from the original target class well. There is, however, an interesting difference between the Tanh and ReLU networks. The Tanh network learns to reconstruct samples in the lower left and upper right corners where training samples are available, but has high reconstruction errors for most samples in-between these two group of samples. The ReLU network also learns to reconstruct samples in between the two groups that lie on an arbitrary line.

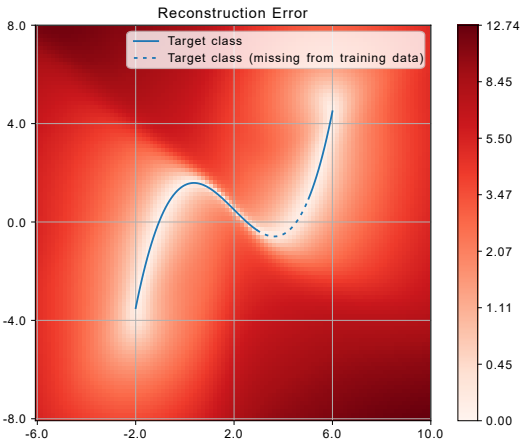


(a) First scenario where interpolation is needed: samples where $x \in [3,5]$ are removed from the LINE-2D dataset.

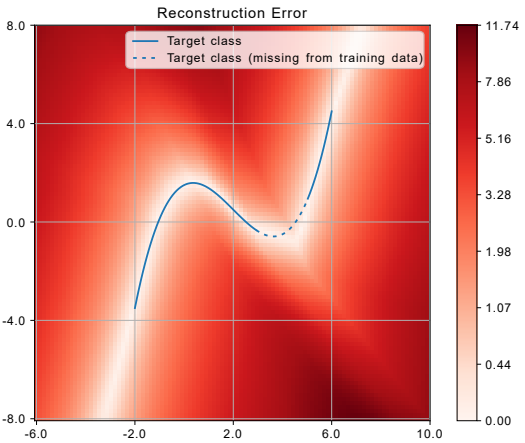


(b) Second scenario where interpolation is needed: samples where $x \in [-1,5]$ are removed from the LINE-2D dataset.

Figure 10.2: Two scenario where interpolation is required to learn the original target class.

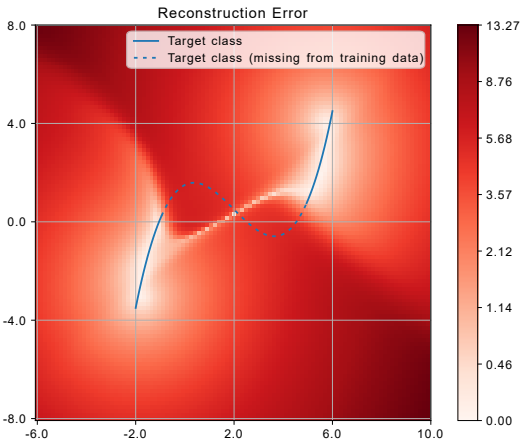


(a) Network with Tanh activations.

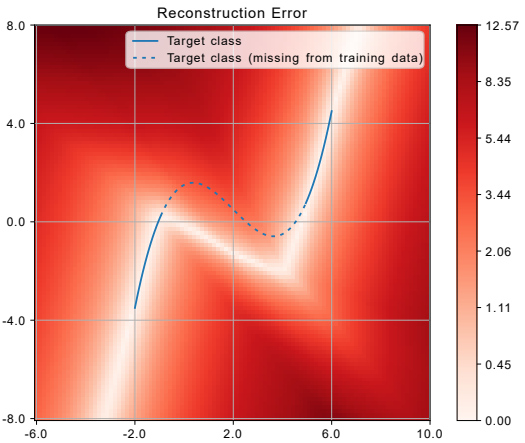


(b) Network with ReLU activations.

Figure 10.3: Reconstruction errors for the first scenario from Figure 10.2a.



(a) Network with Tanh activations.



(b) Network with ReLU activations.

Figure 10.4: Reconstruction errors for the second scenario from Figure 10.2b.

10.2. Limited Training Data

Experiment 10.2 RQ. 2f

Description The performance of an AE and WAE is measured when the number of training samples from the MNIST dataset is reduced.

Purpose In practice, it is often difficult, or expensive, to gather much data. It is thus interesting to know how algorithms perform when little data is available and how much the performance could increase when more data is added.

Conclusion The performance remains reasonable even when little data is available. Adding more data does increase the performance, as expected.

During this experiment, the bottleneck layer was chosen to contain 8 neurons. The digit pairs that were used as target class are: $\{\{3,4\},\{1,5\},\{2,8\},\{6,9\},\{0,7\}\}$ and the number of available training digits that were used are: $\{100,500,1000,2000,4000,\text{all}\}$, where "all" is approximately equal to 6000. Here, 100 available digits for target class $\{3,4\}$ means that 100 samples of digit 3 and 100 samples of digit 4 were included in the training set. Each experiment has been repeated 5 times for each combination of the aforementioned variables.

The results are shown in Figure 10.5. The performance clearly degrades when fewer samples are available, but the performance remains reasonable, even for as little as 100 training samples per digit. Only the WAE performs much worse when fewer samples are available. The only difference between a WAE and an AE is that the WAE has an additional error term to force the encoded target data to follow some distribution. The cause of the bad performance of a WAE is expected to be that, with few samples, the discriminator can overfit on the data too easily.

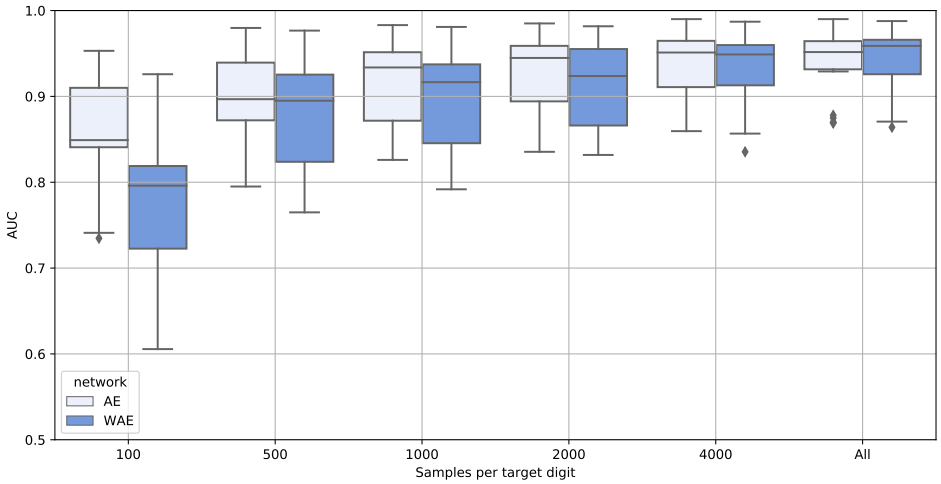


Figure 10.5: The number of training samples is limited to determine how the performance degrades.

10.3. Outliers in Training Data

Experiment 10.3 RQ. 2f

Description The performance of an AE and WAE is measured when outlier samples are included in the MNIST training data.

Purpose The MNIST dataset contains few incorrectly labeled samples. In practice, mistakes might be made more frequently when labelling the data. Thus, it is interesting to investigate how sensitive the performance is to outliers in the training dataset.

Conclusion Outliers in the training dataset decrease the performance a lot. It might be better to have much fewer training samples without any outliers, than to have more training samples with some outliers. This means that it might be better in practice to label fewer data with a higher accuracy.

Alternatively, it might be possible to label a lot of data including some mistakes and to attempt to filter out the outliers from the training data. Even if a significant amount of the training data is filtered out, the performance should remain high, according to the results from Experiment 10.4. Filtering out outliers would, for example, be possible by using any one-class classification technique in combination with cross-validation on the training dataset.

During this experiment, the bottleneck layer was chosen to contain 8 neurons. The digit pairs that were used as target class are: $\{\{3,4\},\{1,5\},\{2,8\},\{6,9\},\{0,7\}\}$. The number of outliers was chosen such that the relative amount of outliers in the training dataset is either 0.0, 0.01, 0.05, 0.1, 0.2, or 0.3. Each experiment has been repeated 5 times for each combination of the aforementioned variables.

The results are shown in Figure 10.6. Outliers in the training dataset clearly cause a large reduction in performance and should thus be avoided at all costs. When comparing to the results that were found in Experiment 10.4, where the number of training samples was limited, it becomes clear that, for this dataset, it is better to have much fewer training samples (for example 500 instead of all 6000 samples), than to have all training samples, but with a small amount of outliers (for example 5% outliers).

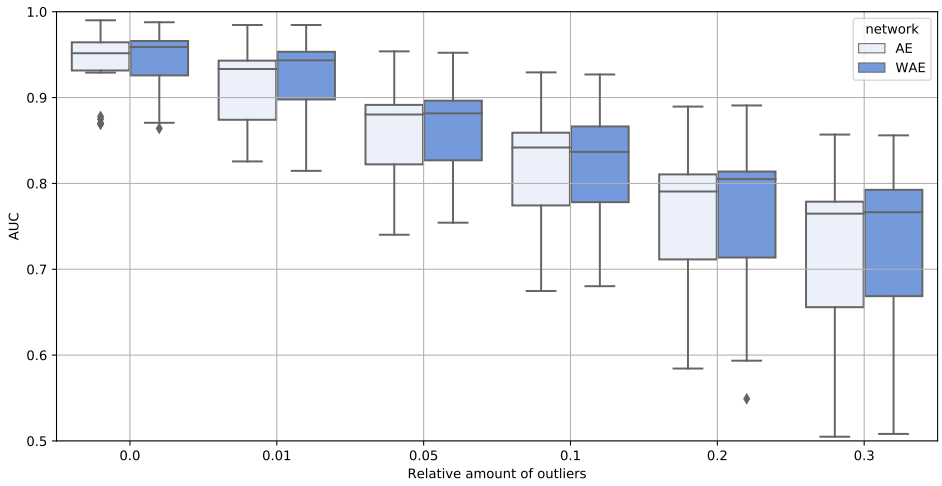


Figure 10.6: The number of outliers in the training data is varied to determine the impact on the performance.

10.4. Noisy Training Data

Experiment 10.4

RQ. 2f

Description The performance of an AE and WAE is measured when noise generated from $\mathcal{N}(0,\sigma^2)$ is added to the MNIST training data.

Purpose The MNIST dataset contains little noise, but in practice data might be noisy. Thus, it is interesting to know how autoencoders behave for noisy data.

Conclusion No significant changes in performance are observed when noise is added to the training data.

During this experiment, the bottleneck layer was chosen to contain 8 neurons. The digit pairs that were used as target class are: $\{\{3,4\},\{1,5\},\{2,8\},\{6,9\},\{0,7\}\}$. The amount of noise is chosen from: $\sigma \in \{0, 0.001, 0.01, 0.05, 0.1, 0.2\}$. Each experiment has been repeated 5 times for each combination of the aforementioned variables. To provide some intuition regarding the amount of noise that was added, some digit samples with $\sigma = 0.2$ are shown in Figure 10.7.

The results are shown in Figure 10.8. The performance does not change much when the amount of noise is increased. Autoencoders can thus be considered to be resilient to random noise when used for one-class classification purposes.

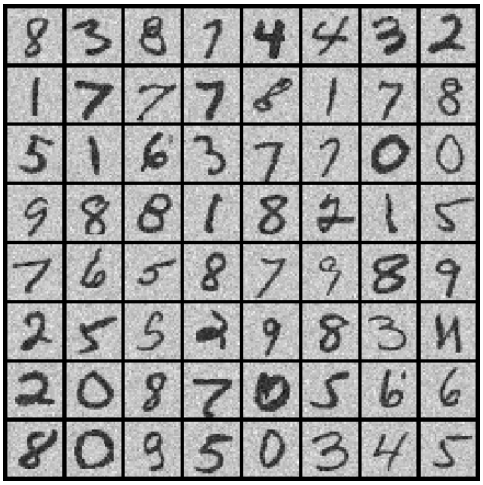


Figure 10.7: Samples from the MNIST dataset with added noise sampled from $\mathcal{N}(0, 0.2^2)$.

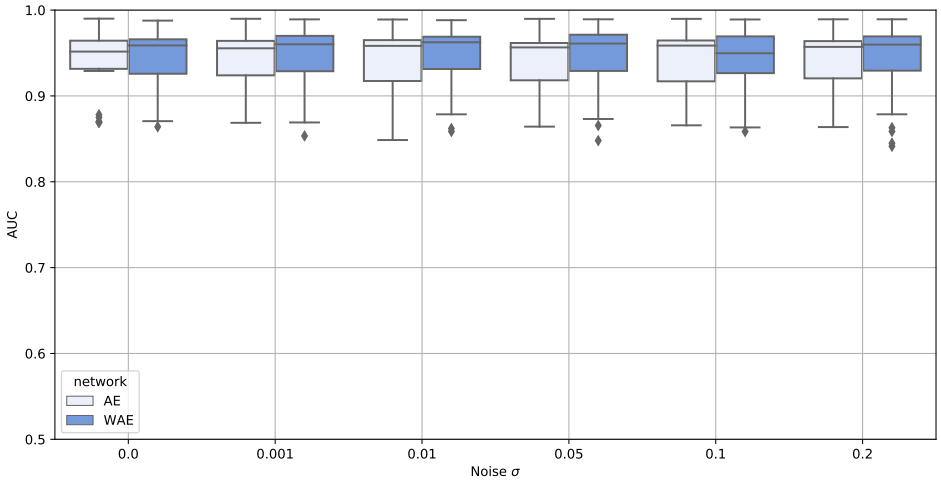


Figure 10.8: The amount of noise of the training data is varied to determine the impact on the performance.

11

Interesting Topics for Further Research

Non-exhaustively tested methods that could be interesting for further research are discussed.

11.1. Applying Convolutional Filters

Experiment 11.1 RQ. 2

Description A WAE with convolutional layers is tested, using as target class the digits 3 and 4 from the MNIST dataset.

Purpose Neural networks that use convolutional filters do not require significantly more trainable weights to be added when the size of the input is increased. This makes convolutional neural networks scale much better to problems of high dimensionality than networks that contain only fully-connected layers. Convolutional filters are thus interesting to investigate in this thesis. Due to the limited amount of time that is available to complete this thesis, convolutional networks are not investigating in as much detail as the methods discussed in the previous chapters.

Conclusion A network with convolutional layers was shown to have a similar (or maybe even slightly better) performance than an autoencoder without convolutions. Thus, it should be possible to use convolutional autoencoders to scale the methods discussed in the previous sections to much higher resolution image data.

It was also shown that the quality of the data that is generated by decoding random noise can be improved when adding an additional discriminator at the end of the autoencoder. Together with the decoder, this discriminator will form a GAN that tries to let the decoder only produce results that are similar to the target data. This might be useful to ensure that the decoder cannot generate non-target data, although during a first test the performance did not improve.

The structure of the Convolutional WAE (ConvWAE) network was chosen similar to that of the DCGAN (Deep Convolutional GAN) [29]. The decoder is structured similarly to the generator of a DCGAN and the encoder is similar to the discriminator. The network architecture is described in more detail in Appendix A.5. The number of hidden units is set equal to $H = 16$. This network should have sufficient complexity to learn to encode and decode the MNIST and Cifar-10 data, since a DCGAN was shown to perform well on these datasets for similar tasks in [29].

It is important to note that all digits were normalized between -1 and 1. This is necessary, because the last layer of the decoder is followed by a Tanh activation function, which cannot output values outside of the range $[-1, 1]$.

When training this network with 16 neurons in the bottleneck layer on digits 3 and 4 from the MNIST dataset, the resulting classifier performs well. The AUC was measured to be 0.952 and the reconstruction error per digit is shown in Figure 11.1. Especially regarding the reconstruction error, a ConvWAE performs much better than a WAE. The ConvWAE has a reconstruction error of approximately 5.2, while a WAE has a

reconstruction error of approximately 10.6 ± 0.33 when the same number of neurons in the bottleneck layer is used. There is, however, not much of a difference in the classification performance. A ConvWAE has an AUC of 0.952, while a WAE has an AUC of 0.948 ± 0.005. This means that the ConvWAE also learns to reconstruct the non-target digits much better. This could mean that the ConvWAE network learns a generic compression and decompression that works reasonably well for all digits, but works especially well for the target digits. To test if the network could reconstruct a much wider range of shapes than just the digits 3 and 4, several digits were generated by decoding random noise. The resulting generated digits are shown in Figure 11.2. Most generated images do represent a 3 or a 4, but some clearly show a different shape, which is not desirable.

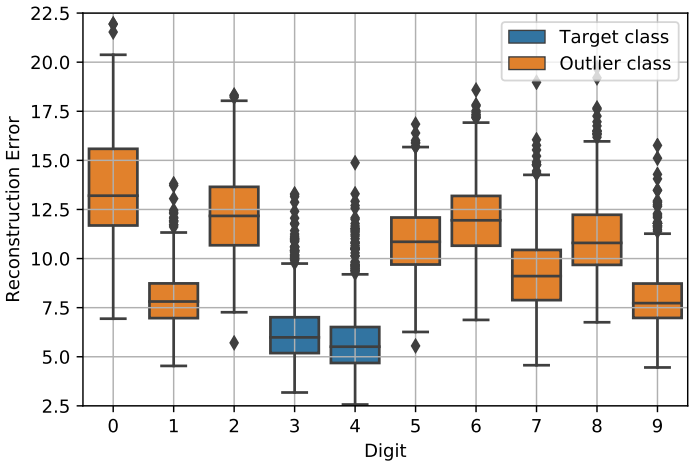


Figure 11.1: Reconstruction error of a ConvWAE that is trained on digits 3 and 4.

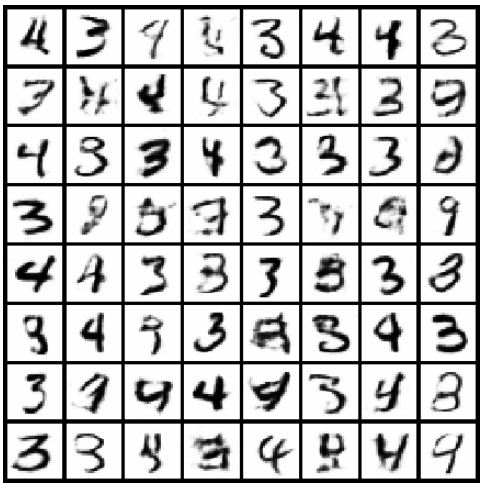


Figure 11.2: 64 generated digits by decoding random noise with a ConvWAE that is trained on digits 3 and 4.

An attempt was made to improve the performance by enforcing that the decoder can only generate digits that look like a 3 or 4. This is done by adding an additional discriminator, which is either given an image of a real digit, or a digit that is generated by decoding random noise. Similarly to a GAN, the discriminator should try to differentiate between real and generated images, while the decoder attempts to fool the discriminator by generating images that look as real as possible. This network is illustrated in Figure 11.3. The network architecture is described in detail in Appendix A.6 and is referred to as ConvWAE-GAN in the further discussion.

The idea of combining an autoencoder and a GAN has been invented prior to this work and is, for example, discussed in [17] and [31]. Most research that combines autoencoders and GANs seems to focus on creating a generative model.

A first attempt at training a ConvWAE-GAN resulted in an AUC of 0.88, thus this network performed worse

than the network without an added GAN, although not much effort was put into training and comparing different variants and tuning parameters. For future research, it is expected that an especially important parameter of this network is the weighting of the objectives of the decoder, as the decoder attempts to both minimize the reconstruction error and 'fool' the discriminator. When generating samples with the trained ConvWAE-GAN, the resulting digits do look much better, although this comparison is subjective. Some generated digits made with a ConvWAE-GAN are shown in Figure 11.4.

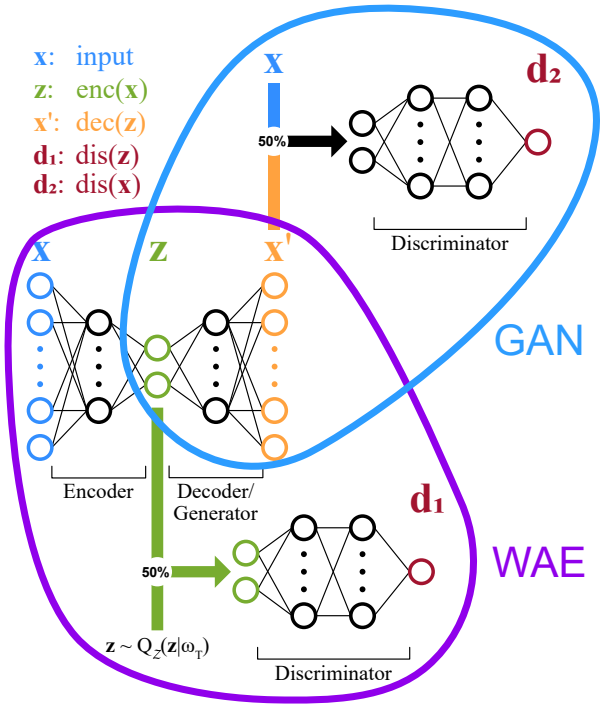


Figure 11.3: An illustration of a WAE combined with a GAN, resulting in a WAE-GAN (or a ConvWAE-GAN when making use of convolutional layers).

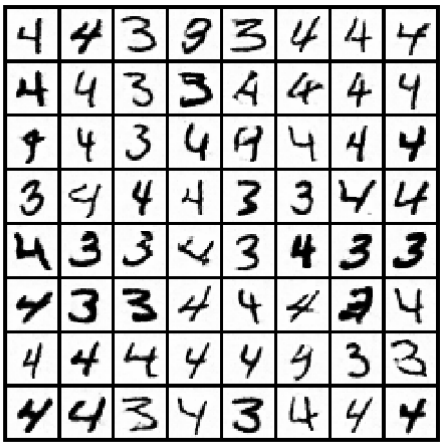


Figure 11.4: 64 generated digits by decoding random noise with a ConvWAE-GAN that is trained on digits 3 and 4.

11.2. Improving the Performance with Unlabeled Data

Experiment 11.2 RQ. 2

Description Experiment 9.3 showed that the performance can be improved by predicting the reconstruction error of a sample, based on its encoding. In this experiment, a slight variation of this method is used. Instead of predicting the reconstruction error by making use of the encoding of the sample, an additional autoencoder is used that is trained on an unlabeled dataset consisting of both target and non-target data. The reconstruction error of this autoencoder will be used as the expected reconstruction error for an autoencoder that is trained on only the target class. For target samples it is expected that both autoencoders have similar errors, while for non-target data, the autoencoder trained on only target data should have a much larger error, making it possible to separate target and non-target data.

Purpose In Experiment 9.3, an increase in performance was observed when predicting the expected reconstruction error. The method treated in this experiment could potentially also cause an increase in performance

Conclusion The presented method that makes use of unlabeled data achieves a much higher performance than previously discussed methods. A drawback is that it requires an unlabeled dataset that contains a representative sampling of the non-target class. A representative sampling of the non-target data is often not available for problems that require a one-class classifier, so the presented approach might not be directly applicable in practice. It might, however, be possible to achieve similar results if the expected reconstruction error can be approximated in a different way, but this is left as a topic for future research. A simple approach that does not require unlabeled data was discussed in Experiment 9.3, but there remains much room for improvement.

Two autoencoders are used in this experiment. The first autoencoder is only trained on the target class. The reconstruction error function of this autoencoder is referred to as $\text{rec\_err}_{\text{TARGET}}(\cdot)$. Another autoencoder is trained on an unlabeled dataset that contains both target and non-target data. The reconstruction error of this autoencoder is called $\text{rec\_err}_{\text{UNLABELED}}(\cdot)$.

The idea is that, for a target sample $\mathbf{x}_T$ and an outlier sample $\mathbf{x}_O$, Equations 11.1 and 11.2 hold.

$$\begin{aligned} \text{rec\_err}_{\text{TARGET}}(\mathbf{x}_T) &\approx \text{rec\_err}_{\text{UNLABELED}}(\mathbf{x}_T) \\ \text{rec\_err}_{\text{TARGET}}(\mathbf{x}_T) - \text{rec\_err}_{\text{UNLABELED}}(\mathbf{x}_T) &\approx 0 \end{aligned} \tag{11.1}$$

$$\begin{aligned} \text{rec\_err}_{\text{TARGET}}(\mathbf{x}_O) &> \text{rec\_err}_{\text{UNLABELED}}(\mathbf{x}_O) \\ \text{rec\_err}_{\text{TARGET}}(\mathbf{x}_O) - \text{rec\_err}_{\text{UNLABELED}}(\mathbf{x}_O) &> 0 \end{aligned} \tag{11.2}$$

Thus, a distinction can be made between target and outlier samples by thresholding the error in Equation 11.3

$$\text{Error}(\mathbf{x}) = \text{rec\_err}_{\text{TARGET}}(\mathbf{x}) - \text{rec\_err}_{\text{UNLABELED}}(\mathbf{x}) \tag{11.3}$$

The resulting performance on the MNIST dataset is compared in Figure 11.5 for the original method that only makes use of the error $\text{rec\_err}_{\text{TARGET}}(\cdot)$ and the newly presented error in Equation 11.3 that makes use of unlabeled data. Clearly, the method that makes use of unlabeled data outperforms the original method by a large margin.

The presented method that makes use of unlabeled data is expected to perform better because of similar reasons as were presented in Experiment 9.3. Some samples are expected to be more difficult to reconstruct, because their shape is more complex and thus harder to generalize. For example, digit 1 consists of a straight line, possibly rotated slightly, and thus is simple to learn to reconstruct, while digit 3 is a much more complex shape that can have many more variations. This is rectified by the term $\text{rec\_err}_{\text{UNLABELED}}(x)$ in the error function, as it reduces the error based on how difficult it is to reconstruct the digit.

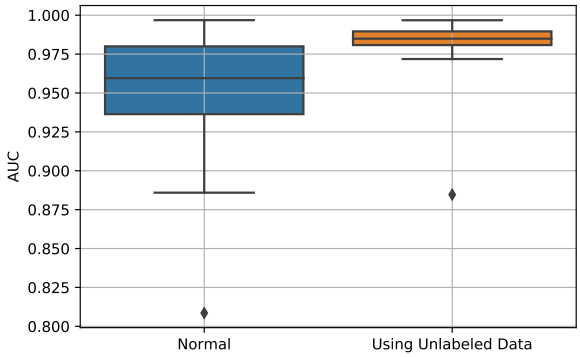


Figure 11.5: The AUC when not making use of unlabeled data to predict the reconstruction error (left) and when making use use of unlabeled data (right).

11.3. Cifar-10 Dataset

Experiment 11.3

RQ. 2

Description Autoencoders, WAE and their convolutional versions are applied to the Cifar-10 dataset and their performance is compared. A comparison to alternative techniques found in the literature is also made.

Purpose The Cifar-10 dataset is much more complex than the MNIST dataset and contains real-world images. This makes the measured performance on the Cifar-10 dataset more meaningful than the earlier results that were based on the MNIST dataset.

Conclusion The performance on the Cifar-10 dataset is not that good for both the discussed methods in this research and the methods found in recent literature. This suggests that the Cifar-10 dataset is a difficult dataset for one-class classification. A possible explanation for this, is that the images from the Cifar-10 dataset consist of a target object, surrounded with a background that is irrelevant for the target class. This background varies a lot between the images, so an autoencoder attempts to reconstruct it as well as possible and does not focus solely on the target object.

The autoencoders discussed in this report perform slightly worse than the best methods found in the literature. Only when making use of an unlabeled dataset that contains both target and non-target data (see Experiment 11.2), our method performs better, although this is not a fair comparison. This does, however, show that it is worthwhile to investigate if determining the reconstruction difficulty of samples can also be done without the use of an unlabeled dataset. This should also be applicable to other methods found in the literature that make use of autoencoders, such as RCAE and DCAE. RCAE and DCAE are the methods from the literature that perform the best amongst the compared methods in Table 11.3.

The measured AUC is reported in Table 11.1. Each network was trained for 100 epochs on the given target class and samples from the other classes were used as outliers. 64 neurons were used in the bottleneck layer of each network. The performance of the networks is nearly identical for all of the tested methods. The convolutional methods have roughly the same performance as the non-convolutional methods. Also, the networks that attempt to distribute the encoded target data according to some prior distribution (WAE, CWAE and networks with an extra Tanh activation) perform similar to their counterparts that do not learn a specific distribution for the encoded target data (AE and CAE). Thus, for the purpose of one-class classification, it does not seem beneficial to constrain the distribution of encoded target data.

Target Class	AE (extra Tanh)	AE (ReLU)	WAE	CAE (extra Tanh)	CAE	CWAE
Airplane	0.67	0.66	0.66	0.68	0.77	0.67
Automobile	0.45	0.40	0.45	0.39	0.39	0.40
Bird	0.66	0.66	0.65	0.68	0.68	0.63
Cat	0.51	0.52	0.51	0.53	0.53	0.48
Deer	0.73	0.74	0.73	0.75	0.75	0.75
Dog	0.50	0.50	0.49	0.54	0.55	0.53
Frog	0.70	0.68	0.71	0.71	0.69	0.64
Horse	0.50	0.50	0.50	0.50	0.51	0.50
Ship	0.67	0.68	0.66	0.70	0.72	0.69
Truck	0.41	0.40	0.40	0.38	0.37	0.40

Table 11.1: The performance (AUC) of several tested methods for each target class from the Cifar-10 dataset.

The technique from Experiment 11.2, where an additional autoencoder that is trained on unlabeled data is used to predict the reconstruction error for test samples, has also been applied to the Cifar-10 dataset and the results are shown in Table 11.2. Compared to the previous results in Table 11.1, a substantial increase in performance is observed. The observed increase in performance for the non-convolutional methods is larger, but the cause for this difference is unclear.

Target Class	AE (extra Tanh)	AE (ReLU)	WAE	CAE (extra Tanh)	CAE	CWAE
Airplane	0.79	0.81	0.72	0.67	0.67	0.65
Automobile	0.84	0.84	0.84	0.65	0.65	0.62
Bird	0.72	0.73	0.65	0.66	0.66	0.57
Cat	0.65	0.72	0.61	0.56	0.56	0.48
Deer	0.71	0.73	0.69	0.75	0.75	0.75
Dog	0.70	0.73	0.62	0.59	0.58	0.57
Frog	0.81	0.79	0.75	0.84	0.84	0.78
Horse	0.79	0.78	0.73	0.66	0.66	0.62
Ship	0.74	0.80	0.71	0.73	0.73	0.69
Truck	0.82	0.81	0.79	0.63	0.63	0.58

Table 11.2: The performance (AUC) of several tested methods for each target class from the Cifar-10 dataset. An unlabeled dataset that contains both target and non-target data is used to rectify the reconstruction errors, as was explained in Experiment 11.2.

The AE (with extra Tanh) results from Table 11.1 and Table 11.2 are compared to several methods found in the literature in Table 11.3. The used abbreviations are explained and referenced in Table 11.4. The results for the methods from the literature were copied from Table 3 in [5]. The methods presented in [5] were trained on the Cifar-10 dataset with 10% of the training data consisting of outliers sampled from the 9 outlier classes, so the same was done while training the AE, to allow for a fair comparison. Surprisingly, the performance did not decrease much because of these added outliers.

In Table 11.3, the performance of the AE that does not use unlabeled data is comparable to the other methods for the Airplane, Bird, Deer, Frog and Ship class, but for the remaining classes the AE performs much worse than the methods from the literature. It is strange that the methods from the literature do not have difficulty with the same classes. There are two possible reasons for this difference that seem plausible. The first is that "global contrast normalization" is used in [5] to preprocess the data. This preprocessing step is not clearly described, so it is unclear what was done exactly and thus not performed for our own methods. The second plausible reason for the observed difference is that in [5] the AUC might be defined as $\max(\text{AUC}, 1-\text{AUC})$. This is not specified in the paper, but it would cause the performance for the Automobile and Truck class of our own methods to become suspiciously similar to the ones reported in [5]. Note that the value of $1-\text{AUC}$ is equivalent to the AUC if the computed labels were negated, which is possible, although it would not really make sense.

With the additional use of an unlabeled dataset, the AE performs better than the methods found in the literature most of the time, as is shown in Table 11.3. Although this is not a fair comparison, it does show that additional research to predict the expected reconstruction error without the use of unlabeled data might

be worthwhile. This method can most likely also be used to improve the DCAE and RCAE methods listed in Table 11.3, as these methods also use autoencoders.

Target Class	OCSVM / SVDD	KDE	IF	DCAE	AnoGAN	SOFT-BOUND DEEP SVDD	ONE-CLASS DEEP SVDD	OC-NN	RCAE	AE (own)	AE (own) UNLABELED
Airplane	0.60 ± 0.01	0.61	0.64 ± 0.01	0.71 ± 0.02	0.67 ± 0.03	0.66 ± 0.01	0.67 ± 0.02	0.60 ± 0.02	0.72 ± 0.03	0.66	**0.74**
Automobile	0.63 ± 0.01	0.63	0.61 ± 0.01	0.63 ± 0.02	0.55 ± 0.03	0.58 ± 0.03	0.58 ± 0.03	0.62 ± 0.02	0.63 ± 0.02	0.39	**0.82**
Bird	0.63 ± 0.01	0.50	0.65 ± 0.01	**0.72** ± 0.01	0.53 ± 0.03	0.62 ± 0.01	0.61 ± 0.02	0.64 ± 0.01	**0.72** ± 0.01	0.64	0.66
Cat	0.60 ± 0.01	0.56	0.56 ± 0.03	0.61 ± 0.00	0.55 ± 0.02	0.58 ± 0.04	0.56 ± 0.01	0.54 ± 0.02	0.61 ± 0.01	0.51	**0.68**
Deer	0.69 ± 0.01	0.66	**0.73 ± 0.01**	0.71 ± 0.02	0.65 ± 0.03	0.63 ± 0.01	0.63 ± 0.01	0.67 ± 0.02	**0.73 ± 0.03**	**0.73**	0.70
Dog	0.66 ± 0.02	0.62	0.61 ± 0.03	0.63 ± 0.02	0.60 ± 0.03	0.59 ± 0.01	0.59 ± 0.01	0.56 ± 0.02	0.64 ± 0.03	0.49	**0.70**
Frog	0.72 ± 0.02	0.71	0.68 ± 0.02	0.65 ± 0.04	0.59 ± 0.01	0.64 ± 0.03	0.64 ± 0.02	0.63 ± 0.03	0.65 ± 0.04	0.68	**0.77**
Horse	0.63 ± 0.01	0.63	0.63 ± 0.02	0.61 ± 0.01	0.63 ± 0.01	0.60 ± 0.02	0.60 ± 0.03	0.60 ± 0.03	0.64 ± 0.00	0.48	**0.79**
Ship	0.60 ± 0.01	0.65	0.68 ± 0.01	0.74 ± 0.01	0.75 ± 0.04	0.70 ± 0.01	0.67 ± 0.02	0.65 ± 0.02	0.75 ± 0.01	0.66	**0.76**
Truck	0.76 ± 0.01	0.74	0.73 ± 0.01	0.71 ± 0.04	0.67 ± 0.03	0.73 ± 0.03	0.68 ± 0.03	0.60 ± 0.05	0.74 ± 0.02	0.38	**0.77**

Table 11.3: Comparing the performance (AUC) of an AE with a wide variety of methods from the literature. Performance measures for the methods from the literature are taken from [5].

Abbreviation	Description and reference
OCSVM / SVDD	One Class Support Vector Machine / Support Vector Data Description [36, 38]
IS	Isolation Forest [19]
KDE	Kernel Density Estimation [27]
DCAE	Deep Convolutional Autoencoder [23]
AnoGAN	Anomaly GAN [29, 35]
SOFT-BOUND DEEP SVDD	Soft-Bound Deep Support Vector Data Description [32]
ONE-CLASS DEEP SVDD	One-Class Deep Support Vector Data Description [32]
OC-NN	One Class Neural Network [5]
RCAE	Robust Convolutional Autoencoder [4]

Table 11.4: Explanation of abbreviations used in Table 11.3 and references to used techniques.

11.4. Improving the Performance using Ensemble Techniques

Experiment 11.4

RQ. 2

Description An ensemble technique is tested to combines multiple GANs.

Purpose Using an ensemble technique can improve the performance.

Conclusion An ensemble of GANs is shown to perform much better than a single GAN for one-class classification purposes. This technique does have a downside, as it requires many GANs to be trained, which could be expensive. The performance is shown to be extremely good on the Line2D dataset, so it is worth investigating this method further. It might also be possible to improve the performance of autoencoders by making use of ensemble techniques, but unfortunately no time was left to test this.

During the final proof-reading of this thesis, a potential improvement for GANs was found. An observation was made in Experiment 6.2 that the discriminator of a GAN seems to always learns a correct classification for the target class, but the classification for the non-target class is not always correct. The cause of this problem is that the discriminator is not supplied 'fake' samples from the entire non-target class. The discriminator will only learn about the non-target class in areas where the generator generates 'fake' samples during the training phase. When the generator converges to generate samples that are similar to the target class, no further learning about the non-target class is possible. When restarting the training phase, the newly generated 'fake' samples are expected to be generated in different areas, so the discriminator can learn to correctly classify different non-target areas.

This makes it possible to improve the performance by using an ensemble of GANs instead of a single GAN. All discriminators are expected to produce a high probability estimate for samples from the target class and for non-target data it is expected that some GANs will produce a low probability estimate.

This intuition is tested for the best GAN networks that were found in Experiment 6.1. The chosen networks are the ones with generator network '8', '8-8', or '8-8-8', with discriminator network '32', '32-32', '32-16', '32-8', '16-16', or '16-8' and with either 1, 2, or 4 hidden features. The trained networks from Experiment 6.1 are reused for this experiment. In that experiment, each type of network was trained 5 times, so a total of $3 \cdot 6 \cdot 3 \cdot 5 = 270$ GANs are combined.

To combine the probability estimates that are produced by these 270 discriminators, a combining rule is required. According to the aforementioned intuition, the product and minimum combining rules should both perform well, while the maximum combining rule should perform bad. This is expected because for non-target data the low probability estimates should not be compensated for by incorrect high probability estimates. More information regarding combining rules for one-class classifiers can be found in [39], especially section 2.5 where the mean and product rule are compared is of interest. The resulting performance for several types of combining rules is reported in Table 11.5.

The minimum and product combining rule indeed work well, while the maximum combining rule results in a terrible performance. The resulting performance of the ensemble method is sufficiently good, that it seems likely that GANs can be used successfully for one-class classification tasks, as opposed to the conclusion of Experiment 6.1 and Experiment 6.2, where it was concluded that the discriminator of GANs does not perform well enough to be used for one-class classification. A downside to this method is that an ensemble of many GANs is required to get a decent performance, which causes the training to become much more expensive.

Combining rule	AUC
Mean	0.970
Product	0.981
Minimum	0.993
Maximum	0.332

Table 11.5: The performance (AUC) of an ensemble of GANs for the Line2D dataset. The result is reported for several types of combining rules.

An initial attempt to apply this technique to the MNIST dataset did not yield successful results. A hundred GANs were trained on the target class consisting of digits 2 and 8. This target class was found to be the most difficult for autoencoders, resulting in an AUC of 0.89 ± 0.0051 for a WAE. The ensemble of GANs achieved an AUC of 0.58, using the minimum combining rule. GANs were, however, shown to be difficult to optimize, so it is likely that the results can be improved significantly, as not much effort was put into optimizing the number of neurons in the network.

The network that was tested with on the MNIST dataset contains 3 layers for the generator and 3 layers for the discriminator. The number of neurons in the layers was set to '64-128-784' for the generator and '128-64-1' for the discriminator. The dimensionality of $\mathbf{z}$ was set to 12. The networks were each trained for 100 epochs. For future research into this topic, we suggest to try to add more neurons to the discriminator and to train for more epochs.

12

Conclusion & Discussion

The topic of one-class classification for high dimensional data was discussed in this thesis. Conventional one-class classification methods were shown to perform well on the MNIST dataset in Chapter 5, but on the Cifar-10 dataset that has a comparable number of dimensions, the conventional methods did not perform well at all. This result suggests that problems involving high-dimensional data do not necessarily require complex neural network approaches to be solved. Only the combination of high complexity and a high number of dimensions justifies the need for more complex methods, such as neural networks.

To solve complex, high dimensional one-class classification problems, two neural network-based approaches were tested. Neural networks are often successful at finding patterns in a dataset, so they might be able to represent the complex high-dimensional data internally by using a much smaller number of features. This is usually done by gradually making the output of each consecutive layer in the network smaller. In this thesis, GANs and autoencoders were tested. The discriminator of a GAN was shown to not perform well as a one-class classifier, especially in areas of the feature space where few samples were generated during the training phase. An ensemble of multiple GANs was shown to produce much better results, although further research on this topic is required. Alternatively, a GAN could be used to generate new samples from the target distributed with reasonable success, although even for the simplest datasets that were used, mode collapse was observed. This means that data is not generated from the entire target distribution.

For the tested autoencoders, no mode collapse was observed. Autoencoders were also found to be much more stable and perform much better at the classification task, thus outperforming GANs at both classification and generating new data. Autoencoders were shown to perform flawlessly at an artificially generated 2D one-class classification problem. The observed behavior exactly matched the theoretical expectations. The target class is reconstructed with near-zero error, while samples far away from the target class are reconstructed with increasingly larger errors, which makes the target and non-target data easily separable. Additionally, autoencoders that match the distribution of encoded target data to some prior distribution (such as Wasserstein and Variational autoencoders) were shown to be able to limit the codomain of the decoder, such that the decoder is unable to generate non-target data.

Experiments performed using the MNIST dataset yielded similar results, although two types of errors were found. The first type of error is when the target and outlier data are roughly similar, which results in them being hard to separate. This was, for example, observed when digit 4 is part of the target class and digit 9 is part of the outlier class, as these digits look much alike. The second type of error is that digit 1 is often reconstructed correctly, even when it does not look like the target data at all. This is caused by the relatively simple shape of digit 1. By making any other type of digit very narrow, the result would have many pixels in common with a sample of digit 1. This causes the reconstruction error to become small, which makes digit 1 difficult to separate from the target data. Both of these types of errors could, to some extent, be avoided by predicting the reconstruction difficulty of the original sample. It was, for example, shown that it is possible to make digit 9 easier to separate from the target class by comparing its error with the error of digits that have a similar encoding. This can increase the performance when there is are large differences in reconstruction difficulty for different types of samples from the target class. This is discussed in Experiment 9.3 and Experiment 11.2, although further improvements are expected to be possible and left as topics for future research.

Autoencoders were shown to require little parameter optimization. All of the important parameters are easy to optimize, as most values produce similar results. The results were also consistent. Whilst GANs might

get stuck while optimizing and require a restart, autoencoders did not have this issue.

While varying the network complexity, it was found that mostly the width of the network is important. Increasing the depth of the network beyond 3 layers was never shown to result in a significant increase in performance. The number of neurons in the decoder network was found to cause the largest variance in performance. It is expected that the task of the encoder is less complex, as it only needs to ensure that samples have a unique encoding. For the encoder it is, for example, possible to remove a feature that can be derived from the other features, which is extremely simple (the feature is multiplied by 0). In this case, the decoder might have to learn a complex (non-linear) function to compute the value of the removed feature based on the other features. When optimizing an autoencoder, it is thus more efficient to focus on the decoder and varying the width of the network.

The distribution of encoded target data was shown to not be very important for simple dataset, such as MNIST. For the more complex Cifar-10 dataset, it was, however, shown that it might be important. For the Cifar-10 dataset, the autoencoder learns to encode and decode samples that do not belong to the target class. This is expected to occur, because the background of the images from the target class varies a lot, which causes the network to learn a generic encoder and decoder that perform well for non-target data as well.

Noise in the training data did not seem to cause autoencoders to perform worse. Decreasing the amount of available training data does decrease the performance, although even with very little data the performance remained reasonable. Outliers in the training dataset, on the other hand, do significantly impact the performance. For the MNIST dataset, it was shown that a better performance is achieved when only 1/12th of the training data is available, than when all data is available, but 5% of the data consists of outliers. In practice, it might thus be better to use a smaller training dataset that does not contain any outliers, than to use a large training dataset that does contain some outliers.

Some novel findings were also made during this research. Adding a Tanh activation function after the bottleneck layer of an autoencoder results in the distribution of the encoded target data to approximately be distributed according to $p_{\mathbf{z}}(x) = 0.5(1 - \tanh^2(x))$. While this was not shown to have a significant impact on the one-class classification performance, it could prove useful for other tasks, such as generative models. For generative models this could be useful, because to generate new samples the distribution of the encoding of target data needs to be known. If the encoded target data is distributed according to $p_{\mathbf{z}}(x) = 0.5(1 - \tanh^2(x))$, then a sample from this distribution could be produced by computing $\operatorname{arctanh}(\mathbf{z})$, given that $\mathbf{z} \sim \mathcal{U}(-1, 1)$. Decoding a sample from this distribution is expected to result in a generated sample that is similar to the target data.

Another novel finding is that an *additional* dataset containing data from both the target and outlier class could be used to significantly improve the one-class classification performance. This is assumed to only improve the performance when the unlabeled dataset contain outliers that are representative for the outliers that will be encountered during the testing phase, so the applications might be limited.

12.1. Limitations

The biggest limitation of using autoencoders for one-class classification that was found is that they are hard to apply to images where a small part consists of the target object and the rest of the image consists of the background. In this case, an autoencoder would learn to reconstruct both the background and the target object. This causes non-target images that have similar backgrounds to often be misclassified.

When there are non-target samples present in the dataset, the performance of autoencoders degrades significantly. This makes autoencoders inappropriate for one-class classification problems where it is hard, or expensive, to remove all non-target samples from the dataset.

Autoencoders also perform poorly when the non-target class images are similar to the target class images, but with small anomalies. For example, when in a 1000×1000 image a region of 5 pixels with a deviating color cause the image to not belong to the target class, the reconstruction error for this small area will be negligible compared to the reconstruction error that is expected for all other 999,995 pixels. Thus, an autoencoder will not be able to detect such small anomalies.

12.2. Future Research

Several topics that are partially discussed and could be interesting for further research are mentioned in Chapter 11.

With the use of convolutional filters, the reconstruction error for the target class can become very small, but some of the non-target samples have a small reconstruction error as well. It might be possible to attempt

to restrict the decoder, such that it is only able to generate target images. This could be done by adding an additional discriminator at the end of the autoencoder, as was discussed in Experiment 11.1, although an initial attempt did not result in a better one-class classifier. This might also make it possible to apply autoencoders as one-class classifiers to datasets where the background varies a lot, such as for the Cifar-10 dataset, which is discussed in Experiment 11.3.

Another approach that could be interesting to investigate is the use of a different error measure than the mean squared error in the pixel space of the original and reconstructed images. When a large part of the images can be considered as the background, measuring the reconstruction error in this area is not beneficial. One alternative method that could be used is discussed in Experiment 11.3. Here a discriminator is added at the end of the autoencoder and the error between the original and reconstructed image is minimized for the features that are produced in an intermediate layer of this discriminator. This idea was introduced in [17].

Of course, the discussed methods could also be tested on different datasets. It is especially interesting to test the methods on high-resolution images to see if the method still behaves correctly. The Celeba [20] or ImageNet [7] datasets could be used for this purpose. This will require a lot of compute time.

In this thesis, several findings were made that could also be interesting for generative models. For example, the method of adding an additional Tanh activation function after the bottleneck layer could be used for generative models (see Experiment 8.1).

The method to use an *additional* unlabeled dataset to improve the classification performance that is discussed in Experiment 11.2 could also be interesting for further research. This was shown to significantly improve the one-class classification performance. This approach could be interesting to a wider variety of problems where unlabeled data is available. It might also be possible to predict the reconstruction difficulty of test samples to get similar results, without making use of an additional unlabeled dataset. A simple approach was successfully demonstrated in Experiment 9.3, but it is expected that there remains much room for improvement.

Finally, in Experiment 11.4 it was shown that the use of ensemble techniques to combine many trained GANs can be used to significantly improve the one-class classification performance. This has only been demonstrated successfully for the artificially generated 2D dataset, as an initial attempt for the MNIST dataset did not result in a satisfactory performance. It might also be possible to improve the performance of autoencoders by using ensemble techniques.

Bibliography

[1] Mahdi Abavisani and Vishal M Patel. Deep multimodal subspace clustering networks. *IEEE Journal of Selected Topics in Signal Processing*, 12(6):1601–1614, 2018.

[2] Ronen Basri and David W. Jacobs. Lambertian reflectance and linear subspaces. In *ICCV*, 2001.

[3] Andrew P Bradley. The use of the area under the roc curve in the evaluation of machine learning algorithms. *Pattern recognition*, 30(7):1145–1159, 1997.

[4] Raghavendra Chalapathy, Aditya Krishna Menon, and Sanjay Chawla. Robust, deep and inductive anomaly detection. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 36–51. Springer, 2017.

[5] Raghavendra Chalapathy, Aditya Krishna Menon, and Sanjay Chawla. Anomaly detection using one-class neural networks. 2019.

[6] Thomas M Cover, Peter Hart, et al. Nearest neighbor pattern classification. *IEEE transactions on information theory*, 13(1):21–27, 1967.

[7] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[8] Carl Doersch. Tutorial on variational autoencoders. 2016.

[9] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[10] Simon Hawkins, Hongxing He, Graham Williams, and Rohan Baxter. Outlier detection using replicator neural networks. In *International Conference on Data Warehousing and Knowledge Discovery*, pages 170–180. Springer, 2002.

[11] Kathryn Hempstalk, Eibe Frank, and Ian H Witten. One-class classification by combining density and class probability estimation. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 505–519. Springer, 2008.

[12] Harold Hotelling. Analysis of a complex of statistical variables into principal components. *Journal of educational psychology*, 24(6):417, 1933.

[13] Nathalie Japkowicz, Catherine Myers, Mark Gluck, et al. A novelty detection approach to classification. In *IJCAI*, volume 1, pages 518–523, 1995.

[14] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. 2013.

[15] Edwin M Knorr, Raymond T Ng, and Vladimir Tucakov. Distance-based outliers: algorithms and applications. *The VLDB Journal—The International Journal on Very Large Data Bases*, 8(3-4):237–253, 2000.

[16] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. Technical report, Citeseer, 2009.

[17] Anders Boesen Lindbo Larsen, Søren Kaae Sønderby, Hugo Larochelle, and Ole Winther. Autoencoding beyond pixels using a learned similarity metric. 2015.

[18] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010. URL `http://yann.lecun.com/exdb/mnist/`.

[19] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. Isolation forest. In *2008 Eighth IEEE International Conference on Data Mining*, pages 413–422. IEEE, 2008.

[20] Ziwei Liu, Ping Luo, Xiaogang Wang, and Xiaoou Tang. Deep learning face attributes in the wild. In *Proceedings of International Conference on Computer Vision (ICCV)*, 2015.

[21] Larry Manevitz and Malik Yousef. One-class document classification via neural networks. *Neurocomputing*, 70(7-9):1466–1481, 2007.

[22] Erik Marchi, Fabio Vesperini, Stefano Squartini, and Björn Schuller. Deep recurrent neural network-based autoencoders for acoustic novelty detection. *Computational intelligence and neuroscience*, 2017.

[23] Jonathan Masci, Ueli Meier, Dan Cireşan, and Jürgen Schmidhuber. Stacked convolutional auto-encoders for hierarchical feature extraction. In *International Conference on Artificial Neural Networks*, pages 52–59. Springer, 2011.

[24] Todd K Moon. The expectation-maximization algorithm. *IEEE Signal processing magazine*, 13(6):47–60, 1996.

[25] Taesung Park, Ming-Yu Liu, Ting-Chun Wang, and Jun-Yan Zhu. Semantic image synthesis with spatially-adaptive normalization. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019.

[26] Lance Parsons, Ehtesham Haque, and Huan Liu. Subspace clustering for high dimensional data: a review. *Acm Sigkdd Explorations Newsletter*, 6(1):90–105, 2004.

[27] Emanuel Parzen. On estimation of a probability density function and mode. *The annals of mathematical statistics*, 33(3):1065–1076, 1962.

[28] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

[29] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. 2015.

[30] Ali Razavi, Aäron van den Oord, and Oriol Vinyals. Generating diverse high-fidelity images with VQ-VAE-2. *CoRR*, 2019.

[31] Mihaela Rosca, Balaji Lakshminarayanan, David Warde-Farley, and Shakir Mohamed. Variational approaches for auto-encoding generative adversarial networks. 2017.

[32] Lukas Ruff, Robert Vandermeulen, Nico Goernitz, Lucas Deecke, Shoaib Ahmed Siddiqui, Alexander Binder, Emmanuel Müller, and Marius Kloft. Deep one-class classification. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 4393–4402. PMLR, 2018.

[33] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[34] Tim Salimans, Ian J. Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen. Improved techniques for training gans. *CoRR*, 2016.

[35] Thomas Schlegl, Philipp Seeböck, Sebastian M Waldstein, Ursula Schmidt-Erfurth, and Georg Langs. Unsupervised anomaly detection with generative adversarial networks to guide marker discovery. In *International Conference on Information Processing in Medical Imaging*, pages 146–157. Springer, 2017.

[36] Bernhard Schölkopf, Alexander J Smola, Francis Bach, et al. *Learning with kernels: support vector machines, regularization, optimization, and beyond*. MIT press, 2002.

[37] Claude Elwood Shannon. A mathematical theory of communication. *Bell system technical journal*, 27 (3):379–423, 1948.

[38] David MJ Tax and Robert PW Duin. Support vector domain description. *Pattern recognition letters*, 20 (11-13):1191–1199, 1999.

[39] David MJ Tax and Robert PW Duin. Combining one-class classifiers. In *International Workshop on Multiple Classifier Systems*, pages 299–308. Springer, 2001.

[40] Ilya Tolstikhin, Olivier Bousquet, Sylvain Gelly, and Bernhard Schoelkopf. Wasserstein auto-encoders. 2017.

[41] Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola. *Dive into Deep Learning*. 2019. Section 3.4.3, Information Theory Basics. `http://www.d2l.ai`.

A

Network Architectures

This appendix contains several network architecture specifications. All networks were modelled and trained making use of PyTorch [28].

A.1. GAN

A GAN consists of a Generator and a Discriminator. The Discriminator is given either a real sample, or a fake, generated, sample that was created by the Generator and has as objective to discriminate between real and fake samples. The Generator instead has the objective to fool the Discriminator. The Generator and Discriminator are trained simultaneously, causing the two networks to learn from each other. Eventually, the Generator is expected to generate samples that are similar to the target data, such that the discriminator cannot discriminate between real and generated samples anymore.

The GAN network architecture that was used for learning the Line2D dataset is shown in Listing A.1. In Experiment 6.1, other network variants were also trained. These networks have varying numbers of neurons per layer and varying numbers of layers, but are otherwise similar to the shown network.

Listing A.1: GAN network architecture

```
GAN(
  (discriminator): Discriminator(
    (network): Sequential(
      (0): Linear(in_features=2, out_features=32, bias=True)
      (1): ReLU()
      (2): Linear(in_features=32, out_features=32, bias=True)
      (3): ReLU()
      (4): Linear(in_features=32, out_features=32, bias=True)
      (5): ReLU()
      (6): Linear(in_features=32, out_features=1, bias=True)
      (7): Sigmoid()
    )
  )
  (generator): Generator(
    (network): Sequential(
      (0): Linear(in_features=1, out_features=8, bias=True)
      (1): ReLU()
      (2): Linear(in_features=8, out_features=8, bias=True)
      (3): ReLU()
      (4): Linear(in_features=8, out_features=8, bias=True)
      (5): ReLU()
      (6): Linear(in_features=8, out_features=2, bias=True)
    )
  )
)
```

```
(optimizer): Adam (
  amsgrad: False
  betas: (0.9, 0.999)
  eps: 1e-08
  lr: 0.001
  weight_decay: 0
)
```

A.2. Autoencoder (AE)

An AE consists of an encoder and a decoder. Typically, the encoder encodes its input data into a lower dimensional space. The decoder attempts to map the encoded samples back to the original. These two networks are trained jointly. To encode samples efficiently, the networks will have to learn to generalize the original data by using a limited number of features.

The AE network architecture that was found to perform the best on the MNIST dataset in Experiment 7.4 is shown in Listing A.2. Here "z_size" denotes the number of dimensions of the encoding. In Experiment 8.2, 8 was found to be a good number for "z_size".

Listing A.2: Autoencoder network architecture

```
AE(
  (encoder): Sequential(
    (0): Linear(in_features=784, out_features=128, bias=True)
    (1): Tanh()
    (2): Linear(in_features=128, out_features=32, bias=True)
    (3): Tanh()
    (4): Linear(in_features=32, out_features=32, bias=True)
    (5): Tanh()
    (6): Linear(in_features=32, out_features=32, bias=True)
    (7): Tanh()
    (8): Linear(in_features=32, out_features=z_size, bias=True)
  )
  (decoder): Sequential(
    (0): Linear(in_features=z_size, out_features=32, bias=True)
    (1): Tanh()
    (2): Linear(in_features=32, out_features=32, bias=True)
    (3): Tanh()
    (4): Linear(in_features=32, out_features=128, bias=True)
    (5): Tanh()
    (6): Linear(in_features=128, out_features=784, bias=True)
  )
)

(optimizer): Adam (
  amsgrad: False
  betas: (0.9, 0.999)
  eps: 1e-08
  lr: 0.001
  weight_decay: 1e-05
)
```

A.3. Variational Autoencoder (VAE)

VAEs are similar to AEs, but encode each sample as a probability distribution. This makes the distribution of encoded target data similar to some chosen prior probability distribution. For VAEs the network architecture was chosen equivalent to the one shown in Listing A.2. The only difference is that "z_size" should be multiplied by 2 for the encoder, since both a mean μ and standard deviation σ should be produced.

A.4. Wasserstein Autoencoder (WAE)

WAEs are similar to AEs, but have an additional objective that enforces that the encoded target data follows some prior distribution. This is done by a discriminator that discriminates between encoded samples ('fake' samples) and samples generated from the prior distribution ('real' samples).

The WAE network architecture that was used is shown in Listing A.3. Here "z_size" denotes the number of dimensions of the encoding. In Experiment 8.2, 8 was found to be a good number for "z_size".

Listing A.3: Wasserstein autoencoder network architecture

```
WAE(
  (encoder): Sequential(
    (0): Linear(in_features=784, out_features=128, bias=True)
    (1): Tanh()
    (2): Linear(in_features=128, out_features=32, bias=True)
    (3): Tanh()
    (4): Linear(in_features=32, out_features=32, bias=True)
    (5): Tanh()
    (6): Linear(in_features=32, out_features=32, bias=True)
    (7): Tanh()
    (8): Linear(in_features=32, out_features=z_size, bias=True)
  )
  (decoder): Sequential(
    (0): Linear(in_features=z_size, out_features=32, bias=True)
    (1): Tanh()
    (2): Linear(in_features=32, out_features=32, bias=True)
    (3): Tanh()
    (4): Linear(in_features=32, out_features=128, bias=True)
    (5): Tanh()
    (6): Linear(in_features=128, out_features=784, bias=True)
  )
  (discriminator): Sequential(
    (0): Linear(in_features=z_size, out_features=16, bias=True)
    (1): ReLU(inplace)
    (2): Linear(in_features=16, out_features=8, bias=True)
    (3): ReLU(inplace)
    (4): Linear(in_features=8, out_features=1, bias=True)
    (5): Sigmoid()
  )
)

(optimizer): Adam (
  amsgrad: False
  betas: (0.9, 0.999)
  eps: 1e-08
  lr: 0.001
  weight_decay: 1e-05
)
```

A.5. Convolutional WAE (ConvWAE)

The ConvWAE consists of an Encoder, Decoder and a Discriminator. The Discriminator ensures that the encoding of the target class has the desired distribution. This is done by discriminating between encoded samples and samples drawn from the desired distribution.

The network architecture of the ConvWAE that was used is shown in Listing A.5. *CWAE_ENC* is the encoder, *CWAE_DEC* is the decoder and *CWAE_DIS* is the discriminator of the ConvWAE. The parameter H is the number of neurons in the bottleneck layer. Note that the given network is of the correct size for the MNIST dataset, which contains 28×28 pixel images with 1 channel (i.e. grayscale). The Cifar-10 dataset contains images of 32×32 pixels with 3 color channels, so the number of number of channels in the encoder at (0) and in

the decoder at (12) should be changed from 1 to 3 and the padding in the encoder at (0) and in the decoder at (9) should be changed from 3 to 1.

Listing A.4: ConvWAE network architecture

```
CWAE(
  (encoder): CWAE_ENC(
    (network): Sequential(
      (0): Conv2d(1, 64, kernel_size=(4, 4), stride=(2, 2), padding=(3, 3), bias=False)
      (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): LeakyReLU(negative_slope=0.2)
      (3): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
      (4): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (5): LeakyReLU(negative_slope=0.2)
      (6): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
      (7): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (8): LeakyReLU(negative_slope=0.2)
      (9): Conv2d(256, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
      (10): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (11): LeakyReLU(negative_slope=0.2)
      (12): Conv2d(512, H, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    )
  )
  (decoder): CWAE_DEC(
    (network): Sequential(
      (0): ConvTranspose2d(H, 512, kernel_size=(4, 4), stride=(1, 1), bias=False)
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU()
      (3): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1),
                           bias=False)
      (4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (5): ReLU()
      (6): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1),
                           bias=False)
      (7): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (8): ReLU()
      (9): ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2), padding=(3, 3),
                           bias=False)
      (10): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (11): ReLU()
      (12): ConvTranspose2d(64, 1, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (13): Tanh() # Note: the data is normalized between -1 and +1
    )
  )
  (discriminator): CWAE_DIS(
    (network): Sequential(
      (0): Linear(in_features=H, out_features=16, bias=True)
      (1): ReLU()
      (2): Linear(in_features=16, out_features=8, bias=True)
      (3): ReLU()
      (4): Linear(in_features=8, out_features=1, bias=True)
      (5): Sigmoid()
    )
  )
)
```

A.6. Convolutional WAE with additional Discriminator (ConvWAE-GAN)

Equivalent to the network shown in Appendix A.5, but with an additional discriminator at the end that discriminates between data generated by the decoder when $z \sim U(-1,1)$ and the real training images. An

additional Tanh was also added to the end of the encoder (13) and during training noise generated from $\mathcal{N}(0, 0.05^2)$ was added to the output of the encoder, to ensure that the encoder also outputs values in the range $[-1, 1]$ with an approximately uniform distribution when encoding the target data (see Experiment 8.1 for details on why this works).

The autoencoder and GAN parts are all trained at the same time. For each batch, first a gradient update was done for the autoencoder and after that for the GAN (although the order in which the gradients are updates is probably not important).

Listing A.5: ConvWAE-GAN network architecture

```
CWAE_GAN(
  (encoder): CWAE_ENC(
    (network): Sequential(
      (0): Conv2d(1, 64, kernel_size=(4, 4), stride=(2, 2), padding=(3, 3), bias=False)
      (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): LeakyReLU(negative_slope=0.2)
      (3): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
      (4): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (5): LeakyReLU(negative_slope=0.2)
      (6): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
      (7): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (8): LeakyReLU(negative_slope=0.2)
      (9): Conv2d(256, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
      (10): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (11): LeakyReLU(negative_slope=0.2)
      (12): Conv2d(512, H, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
      (13): Tanh()
    )
  )
  (decoder): CWAE_DEC(
    (network): Sequential(
      (0): ConvTranspose2d(H, 512, kernel_size=(4, 4), stride=(1, 1), bias=False)
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU()
      (3): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1),
                           bias=False)
      (4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (5): ReLU()
      (6): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1),
                           bias=False)
      (7): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (8): ReLU()
      (9): ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2), padding=(3, 3),
                            bias=False)
      (10): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (11): ReLU()
      (12): ConvTranspose2d(64, 1, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (13): Tanh() # Note: the data is normalized between -1 and +1
    )
  )
  (discriminator): CWAE_DIS(
    (network): Sequential(
      (0): Linear(in_features=H, out_features=16, bias=True)
      (1): ReLU()
      (2): Linear(in_features=16, out_features=8, bias=True)
      (3): ReLU()
      (4): Linear(in_features=8, out_features=1, bias=True)
      (5): Sigmoid()
    )
  )
```

```
  (discriminator_additional): CWAE_DIS2(
    (network): Sequential(
      (0): Conv2d(1, 64, kernel_size=(4, 4), stride=(2, 2), padding=(3, 3), bias=False)
      (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): LeakyReLU(negative_slope=0.2)
      (3): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
      (4): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (5): LeakyReLU(negative_slope=0.2)
      (6): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
      (7): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (8): LeakyReLU(negative_slope=0.2)
      (9): Conv2d(256, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
      (10): Sigmoid()
    )
  )
)
```