

# Masterarbeit

Eric Schaefer

Using Autoencoder for Compression, Denoising, and  
Clustering of GISAXS Simulations Images for Au-Cluster  
Physics

Eric Schaefer

# Using Autoencoder for Compression, Denoising, and Clustering of GISAXS Simulations Images for Au-Cluster Physics

Master thesis submitted for examination in Master's degree  
in the study course *Master of Science Informatik*  
at the Department Computer Science  
at the Faculty of Engineering and Computer Science  
at University of Applied Science Hamburg

Supervisor: Prof. Dr.-Ing. Marina Tropmann-Frick  
Supervisor: Prof. Dr. Peer Stelldinger

Submitted on: December 19, 2024

**Eric Schaefer**

## **Thema der Arbeit**

Verwendung von Autoencoder zur Komprimierung, Rauschunterdrückung und Clusterung von GISAXS-Simulationsbildern für Au-Cluster Physics

## **Stichworte**

Deep Learning, Autoencoder, Bildkomprimierung, Bildrekonstruktion, Rauschunterdrückung, Clustering, GISAXS, Au-Cluster Physik

## **Kurzzusammenfassung**

Diese Arbeit untersucht die Anwendung von Autoencodern (AE) zur Komprimierung, Rauschunterdrückung und Clusterung von Simulationsbildern der Kleinwinkelröntgenstreuung unter streifendem Einfall (GISAXS) im Zusammenhang mit der Selbstorganisation von Gold-(Au)-Nanoclustern auf Siliziumoberflächen. Die Forschung befasst sich mit den Herausforderungen, die sich aus den enormen Datenmengen ergeben, die bei GISAXS-Experimenten erzeugt werden, was direkte analytische Ansätze aufgrund des inversen mathematischen Problems von GISAXS erschwert. Durch den Einsatz von Autoencodern zielt die Studie darauf ab, die Effizienz der Datenanalyse in diesem Bereich zu verbessern, wobei der Schwerpunkt auf drei Hauptbereichen liegt: Reduzierung des Speicherbedarfs durch Bildkomprimierung, Verbesserung der Bildqualität durch Entfernen von Rauschen und Rekonstruieren von Lücken in den Daten sowie Erleichterung der Klassifizierung durch Generierung aussagekräftiger Cluster im latenten Raum. Die Leistung der vorgeschlagenen Methoden wird anhand von Simulationsdatensätzen bewertet, und die Ergebnisse zeigen das Potenzial von Autoencodern, einen wesentlichen Beitrag zur Verarbeitung und Analyse von GISAXS-Daten in Bezug auf Komprimierung und Rauschunterdrückung, jedoch nicht in Bezug auf Clusterung, zu leisten und so Fortschritte in der Au-Cluster Physik zu unterstützen.

**Eric Schaefer**

**Title of Thesis**

---

# Using Autoencoder for Compression, Denoising, and Clustering of GISAXS Simulations Images for Au-Cluster Physics

## **Keywords**

Deep Learning, Autoencoder, Image-compression, image reconstruction, Denoising, Clustering, GISAXS, Au-Cluster Physik

## **Abstract**

This thesis explores the application of autoencoders (AE) for the compression, denoising, and clustering of Grazing Incidence Small-Angle X-ray Scattering (GISAXS) simulation images related to the self-organization of gold (Au) nanoclusters on silicon surfaces. The research addresses the challenges posed by the vast data produced in GISAXS experiments, which complicates direct analytical approaches due to the inverse mathematical problem of GISAXS. By employing autoencoders, the study aims to enhance the efficiency of data analysis in this domain, focusing on three main areas: reducing the storage needs through image compression, improving image quality by removing noise and reconstructing gaps in the data, and facilitating classification by generating meaningful clusters in the latent space. The performance of the proposed methods is evaluated on simulation datasets, and the results demonstrate the potential of autoencoders to contribute significantly to the processing and analysis of GISAXS data regarding compression and denoising, but not clustering, thereby supporting advancements in Au-Cluster Physics.

# Contents

<b>List of Figures</b>	<b>viii</b>
<b>List of Tables</b>	<b>xii</b>
<b>Abbreviations</b>	<b>xiv</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motivation . . . . .	1
1.2 Goal . . . . .	3
1.3 Structure . . . . .	4
<b>2 Previous Work</b>	<b>6</b>
2.1 Autoencoder (AE) . . . . .	6
2.1.1 Vanilla Autoencoder . . . . .	7
2.1.2 Stacked Autoencoder . . . . .	9
2.1.3 Hyperparameters in Autoencoder . . . . .	10
2.1.4 Regularized Autoencoder . . . . .	13
2.1.5 Robust autoencoder . . . . .	15
2.1.6 Generative Autoencoder . . . . .	17
2.1.7 Convolutional Autoencoder (ConvAE) . . . . .	18
2.1.8 Semi-Supervised Autoencoder . . . . .	19
2.2 Compression . . . . .	20
2.2.1 CompressAI . . . . .	20
2.2.2 Factorized Prior Compression . . . . .	22
2.2.3 Scale Hyperprior Compression . . . . .	23
2.2.4 Mean and Scale Hyperprior Compression . . . . .	25
2.2.5 Joint Autoregressive and Hierarchical Priors Compression . . . . .	26
2.2.6 Residual Joint Autoregressive and Hierarchical Priors Compression . . . . .	27
2.3 Denoising . . . . .	29

2.4	Clustering . . . . .	30
2.5	Au-Cluster Physics with GISAXS . . . . .	31
2.5.1	Sputter Deposition of Gold on Silicon . . . . .	32
2.5.2	Surface Processes . . . . .	33
2.5.3	Growth process . . . . .	34
2.5.4	GISAXS . . . . .	35
2.5.5	Analysing GISAXS scattering pattern . . . . .	38
2.5.6	Simulation of GISAXS scattering patterns . . . . .	40
2.6	AI in the Field of Scattering Images . . . . .	43
<b>3</b>	<b>Proposed Approach</b>	<b>46</b>
3.1	Compression . . . . .	47
3.2	Denoising . . . . .	50
3.3	Clustering . . . . .	52
<b>4</b>	<b>Experiments</b>	<b>54</b>
4.1	Dataset . . . . .	54
4.1.1	Data Augmentation . . . . .	56
4.2	Training Procedures and Results . . . . .	60
4.2.1	Data Preparation . . . . .	60
4.2.2	Training Procedures . . . . .	62
4.2.3	Evaluation . . . . .	63
4.3	Discussion . . . . .	74
4.3.1	Compression . . . . .	76
4.3.2	Denoising . . . . .	80
4.3.3	Clustering . . . . .	81
<b>5</b>	<b>Conclusion and Future Work</b>	<b>84</b>
5.1	Limitations and Future Directions . . . . .	84
	<b>Bibliography</b>	<b>86</b>
<b>A</b>	<b>Appendix</b>	<b>97</b>
A.1	Source Code . . . . .	97
A.1.1	k-sparse Linear Layer . . . . .	97
A.1.2	Decreasing sparsity level (k) . . . . .	97
A.1.3	Warm Up . . . . .	98

A.1.4	Gaussian white Additive noise . . . . .	99
A.1.5	Bernoulli-masked noise . . . . .	99
A.1.6	Poisson noise . . . . .	99
A.1.7	PyTorch Dastate . . . . .	99
A.1.8	PyTorch Lightning’s DataModule . . . . .	101
A.1.9	PyTorch Lightning class for the compression models . . . . .	102
A.1.10	PyTorch Lightning Trainer . . . . .	105
A.1.11	Generate Callback . . . . .	106
<b>Glossary</b>		<b>108</b>
<b>Declaration of Authorship</b>		<b>109</b>

# List of Figures

2.1	The general structure of an Autoencoder (AE), mapping an input $\mathbf{x}$ to an output (called reconstruction) $\mathbf{r}$ through an internal representation or code $\mathbf{h}$ . The AE has two components: the encoder $f$ (mapping $\mathbf{x}$ to $\mathbf{h}$ ) and the decoder $g$ (mapping $\mathbf{h}$ to $\mathbf{r}$ ). [21, p. 500] . . . . .	7
2.2	Taxonomy of AE architectures categorized by network structure. Source: Berahmand et al. [8, p. 28] (Copyright © 2024, Springer International Publishing). . . . .	7
2.3	Illustration of the structure of an AE, where $X$ represents the input data of the input layer, $Z$ represents the data in the hidden layer, and $X'$ represents the reconstructed output data in the output layer. Source: Berahmand et al. [8, p. 12] (Copyright © 2024, Springer International Publishing). . . . .	8
2.4	Common activation Functions in Neuronal Network (NN). Source: Berahmand et al. [8, p. 12] (Copyright © 2024, Springer International Publishing). . . . .	11
2.5	Overview of the k-sparse AEs including the algorithm for the training. Source: Makhzani and Frey [40, p. 2] (arXiv preprint arXiv:1312.5663, 2014). . . . .	14
2.6	<b>Fully-Factorized Model</b> This model learns a fixed entropy model shared between the encoder and decoder systems. It is designed based on the assumption that all latents are Independent and Identically Distributed (IID) and ideally set to be a multinomial. In practice, it must be relaxed to ensure differentiability. The compression model is primarily an AE composed of convolutional layers and nonlinear activations, where the main complication is due to ensuring that the entropy model is differentiable and that useful gradients flow through the quantized latent representation. Source: Minnen et al. [46, p. 20] (arXiv preprint arXiv:1809.02736, 2018). . . . .	23



- 2.7 Network architecture of the hyperprior model. The left side shows an image autoencoder architecture and the right side corresponds to the autoencoder implementing the hyperprior. The factorized-prior model uses the identical architecture for the analysis and synthesis transforms  $g_a$  and  $g_s$ .  $Q$  represents quantization, and  $AE$ ,  $AD$  represent arithmetic encoder and arithmetic decoder, respectively. Convolution parameters are denoted as number of filters  $\times$  kernel support height  $\times$  kernel support width / down- or upsampling stride, where  $\uparrow$  indicates upsampling and  $\downarrow$  downsampling.  $N$  and  $M$  were chosen dependent on  $\lambda$ , with  $N = 128$  and  $M = 192$  for the 5 lower values, and  $N = 192$  and  $M = 320$  for the 3 higher values. Source: Ballé et al. [5, p. 6] (arXiv preprint arXiv:1802.01436, 2018). . . . 24
- 2.8 **Scale-only Hyperprior** This model uses a conditional Gaussian Scale Mixture (GSM) as the entropy model. The Gaussian Scale Mixture (GSM) is conditioned on a learned hyperprior, a (hyper-)latent representation formed by transforming the latent space using the Hyper-Encoder. The Hyper-Decoder can then decode the hyperprior to create the scale parameters for the GSM. The main advantage of this model is that the entropy model is image-dependent and can be adapted for each code. The downside is that the compressed hyperprior must be transmitted with the compressed latent space, which increases the total file size. Source: Minnen et al. [46, p. 20] (arXiv preprint arXiv:1809.02736, 2018). . . . . 25
- 2.9 **Mean & Scale Hyperprior** This model variant is a simple extension of the scale-only hyperprior model shown in Fig. 2.8 in which the GSM is replaced with a Gaussian Mixture Model (GMM). The Hyper-Decoder is therefore responsible for transforming the hyperprior into mean and scale parameters of the Gaussians. Source: Minnen et al. [46, p. 21] (arXiv preprint arXiv:1809.02736, 2018). . . . . 26
- 2.10 The joint autoregressive & hierarchical model jointly optimizes an autoregressive component that predicts latent spaces from their causal context (Context Model) along with a hyperprior and the underlying autoencoder. Real-valued latent representations are quantized ( $Q$ ) to create latent space ( $\hat{y}$ ) and hyper-latent space ( $\hat{z}$ ), which are compressed into a bitstream using an arithmetic encoder ( $AE$ ) and decompressed by an arithmetic decoder ( $AD$ ). The highlighted region corresponds to the components the receiver executes to recover an image from a compressed bitstream. Source: Minnen et al. [46, p. 3] (arXiv preprint arXiv:1809.02736, 2018). . 27

2.11	Operational diagrams of learned compression models (a)(b)(c) and proposed Gaussian Mixture Likelihoods (d). Source: Cheng et al. [14, p. 3] (Copyright © 2020, IEEE)	28
2.12	Network architecture for the residual joint autoregressive and hierarchical priors model. Source: Cheng et al. [14, p. 5] (Copyright © 2020, IEEE)	28
2.13	Different attention modules for the residual joint autoregressive and hierarchical priors model with attention modules. Source: Cheng et al. [14, p. 6] (Copyright © 2020, IEEE)	29
2.14	The denoising autoencoder architecture. An example $\mathbf{x}$ is stochastically corrupted (via $q_{\mathcal{D}}$ ) to $\tilde{\mathbf{x}}$ . The AE then maps it to $\mathbf{y}$ (via encoder $f_{\theta}$ ) and attempts to reconstruct $\mathbf{x}$ via decoder $g_{\theta'}$ , producing reconstruction $\mathbf{z}$ . Reconstruction error is measured by loss $L_H(\mathbf{x}, \mathbf{z})$ . Source: Vincent et al. [64, p. 9] (Copyright © 2010, ACM, Inc.)	30
2.15	An illustration of the concept of noise learning based DAE (nLDAE). Source: Lee et al. [34, p. 2] (arXiv preprint arXiv:2101.07937, 2021).	30
2.16	2D visualization of the latent space using t-SNE. Source: Pascal [49] (Copyright © 2023, Medium).	31
2.17	Overview of an Gold (Au)-Cluster-Physics experiment including generation of Small Angle X-ray Scattering Under Grazing Incident Angles (GISAXS) scattering image. Source: Schwartzkopf et al. [56] (Copyright © 2013, Royal Society of Chemistry).	32
2.18	Four growth phases of gold clusters according to the hemispherical model with the side view of the occurring surface processes. Source: Schwartzkopf et al. [56] (Copyright © 2013, Royal Society of Chemistry).	35
2.19	GISAXS geometry. Source Meyer [45] (Copyright © 2018, University of Hamburg).	37
2.20	Experimental GISAXS scattering pattern for the material system gold on silicon with marked characteristic features. Based on [55, p. 86]. Source: Almamedov, Eldar [1, p. 17] (Copyright © 2022, Hamburg University of Applied Sciences).	39
2.21	Morphological parameters according to the hemispheric model. Source: Schwartzkopf et al. [56] (Copyright © 2013, Royal Society of Chemistry).	40
2.22	Simulation of form factor and structure factor in $IsGISAXS$ to generate a GISAXS scatter pattern.	41
2.23	Influence of $\sigma/R$ on $IsGISAXS$ simulation for $R = 4.0\text{nm}$ , $D = 10.0\text{nm}$ , and $\omega/D = 0.15$	42

2.24	Influence of $\omega/D$ on $I_s$ GISAXS simulation for $R = 4.0\text{nm}$ , $\sigma/R = 0.15$ , and $D = 10.0\text{nm}$ . . . . .	42
2.25	Examples for the radius distribution of Ag (Silver). Source: Santoro et al. [54] (Copyright ©AIP Publishing LLC, 2014) . . . . .	45
3.1	Overview of our proposed model. The red rectangles belong to the compression step, the yellow rectangles to the denoising step, and the green rectangles to the clustering. We get the reconstructed image and the cluster as the model's output. . . . .	47
3.2	Potential architecture for the denoising step. . . . .	51
3.3	Proposed approach for clustering step. . . . .	53
4.1	8 Examples for simulated GISAXS scattering patterns. The title for each scattering patterns contains $R$ , $\sigma/R$ , $D$ , $\omega/D$ . . . . .	55
4.2	8 Examples for simulated GISAXS scattering patterns including the detector gaps. The title for each scattering patterns contains $R$ , $\sigma/R$ , $D$ , $\omega/D$ . . . . .	57
4.3	8 Examples for simulated GISAXS scattering patterns including the detector gaps and noise. The title for each scattering patterns contains $R$ , $\sigma/R$ , $D$ , $\omega/D$ . . . . .	59
4.4	The Multi-Scale Structural Similarity (MS-SSIM) loss for the Factorized Prior Compression AE (FP) models 14 and 16. . . . .	67
4.5	The MS-SSIM loss for the denoising model 4. . . . .	71
4.6	Training and validation loss of the $k$ -sparse model. . . . .	74
4.7	8 examples of simulated GISAXS scattering patterns. . . . .	75
4.8	Reconstructed results of the clean example images with the Mean and Scale Hyperprior Compression AE (MSH) model. . . . .	77
4.9	Reconstructed results of the example images with detector gaps to clean images trained with Residual Joint Autoregressive and Hierarchical Priors Compression AE with Attention Blocks (AT) model number 5. Mean Squared Error (MSE) loss was used. . . . .	78
4.10	Reconstructed results of the example noisy images with detector gaps to clean images trained with AT model number 9. MS-SSIM loss was used. . . . .	79
4.11	Reconstructed and denoising results of the example noisy images with detector gaps to clean images trained with AT model number 1. . . . .	81

# List of Tables

2.1	Re-implemented models from the state-of-the-art on learned image compression currently available in CompressAI. Training, fine-tuning, inference, and evaluation of the models listed in this table are fully supported. The rate-distortion performances reported in the original papers have been successfully reproduced from scratch [6]. . . . .	21
2.2	Lambda values used for training the networks are different bit-rates (quality setting from 1 to 8), for the MSE and MS-SSIM metrics [6] . . . . .	21
2.3	Loss functions used for training the networks, with $\mathcal{D}$ the distortion and $\mathcal{R}$ the estimated bit-rate. . . . .	21
2.4	Each row corresponds to a layer of the generalized model. The convolution layers are specified with the 'Conv' prefix followed by the kernel size, number of channels, and downsampling stride (for example, the first layer of the encoder uses $5 \times 5$ kernels with 192 channels and a stride of two). The 'Deconv' prefix corresponds to upsampled convolutions (that is, in TensorFlow, <code>tf.conv2d_transpose</code> ), while the 'Masked' prefix corresponds to the masked convolution as in [63]. Generalized Divisive Normalization Layer (GDN) stands for generalized divisive normalization, and Inverse Generalized Divisive Normalization Layer (IGDN) is inverse GDN [4]. Source: Minnen et al. [46, p. 4] (arXiv preprint arXiv:1809.02736, 2018). . . . .	27
2.5	Simulation parameters for form and structure factors. . . . .	43
2.6	Experimental simulation parameters for X-ray beam and detector. . . . .	43
4.1	Contains information about the simulation parameters and the different radii for each subset. . . . .	55
4.2	Results of the Factorized Prior [5] Compression AE run with the CompressAI [6, 7] training script. The $\lambda$ as a weight factor for the loss is set to 0.01. The primary Loss function is MSE. . . . .	65

4.3	Results of the Factorized Prior [5] Compression AE run with the Lightning AI [18, 19] front-end. The $\lambda$ as a weight factor for the loss is set to 0.01. The primary Loss function is MSE. . . . .	65
4.4	Results of the Factorized Prior [5] Compression AE run with the CompressAI [6, 7] training script. The $\lambda$ as a weight factor for the loss is set to 15. The primary Loss function is MS-SSIM. . . . .	66
4.5	Results of the Factorized Prior [5] Compression AE run with the Lightning AI [18, 19] front-end. The $\lambda$ as a weight factor for the loss is set to 15. The primary Loss function is MS-SSIM. . . . .	66
4.6	Metric test with MS-SSIM. With metric 6 it should change to the model using 64 and 107 as the number of channels. . . . .	67
4.7	MS-SSIM trained using quality metric 1. . . . .	68
4.8	MSE trained using quality metric 1. . . . .	68
4.9	MS-SSIM for different input-output pairs using AT(64). . . . .	69
4.10	MSE for different input-output pairs using AT(64). . . . .	69
4.11	Training denoising of masked input to clean target with MS-SSIM loss and quality metric 1. . . . .	70
4.12	Adding denoising to the compression model. The version for the model without noise refers to the compression tests. . . . .	71
4.13	Testing denoising with fully connected AE with depth 1. . . . .	72
4.14	Testing denoising with fully connected AE with increasing depth level. . .	72
4.15	MS-SSIM for different input-output pairs using AT(64). All versions can be found under the compression models. . . . .	73
4.16	Shows the different losses of the Label and Sparse Regularized Autoencoder (LSRAE) model. . . . .	73
4.17	Training results and setup of the $k$ -sparse model. . . . .	74
4.18	Show the size of the latent spaces and the Bits per pixel (Bpp) for the image and the compression. . . . .	79
4.19	Shows all the different simulated GISAXS scattering patterns distributed by the cluster created by the $k$ -sparse AE. The first column contains the index of the neuron that was still active. . . . .	82
4.20	Shows the classification results by the LSRAE model for the different classification parameters of the experiments. . . . .	83

# Abbreviations

**AdAtom** adsorption of an atom.

**AE** Autoencoder.

**AI** Artificial Intelligence.

**AN** Residual Joint Autoregressive and Hierarchical Priors Compression AE with Anchor Blocks.

**Ar** Argon.

**AT** Residual Joint Autoregressive and Hierarchical Priors Compression AE with Attention Blocks.

**Au** Gold.

**AWGN** Additive White Gaussian noise.

**BA** Born approximation.

**BCE** Binary Cross-Entropy.

**BMN** Bernoulli-Masked noise.

**Bpp** Bits per pixel.

**CNN** Convolutional Neuronal Network.

**ConvAE** Convolutional Autoencode.

**ConvSAE** Convolutional Sparse Autoencoder.

**DAE** Denoising Autoencoder.

**DCT** Discrete Cosine Transform.

**DESY** Deutsches Elektronen-Synchrotron.

**DiffusionAE** Diffusion Autoencoder.

**DL** Deep Learning.

**DWBA** Distorted Wave Born Approximation.

**DWT** Discrete Wavelet Transform.

**FC** fully connected.

**FP** Factorized Prior Compression AE.

**GDN** Generalized Divisive Normalization Layer.

**GISAXS** Small Angle X-ray Scattering Under Grazing Incident Angles.

**GMM** Gaussian Mixture Model.

**GSM** Gaussian Scale Mixture.

**HAW** Hamburg University of Applied Sciences.

**IGDN** Inverse Generalized Divisive Normalization Layer.

**IID** Independent and Identically Distributed.

**JAHP** Joint Autoregressive and Hierarchical Priors Compression AE.

**KL** Kullback-Leibler.

**LSRAE** Label and Sparse Regularized Autoencoder.

**M-DAE** Marginalized Denoising Autoencoder.

**ML** Machine Learning.

**MS-SSIM** Multi-Scale Structural Similarity.

**MSE** Mean Squared Error.

**MSH** Mean and Scale Hyperprior Compression AE.

**nIDAE** noise learning based DAE.

**NN** Neuronal Network.

**PCA** Principal Component Analysis.

**ReLU** Rectified Linear Unit (activation function).

**ResNet** Residual Neural Network.

**SAE** Sparse Autoencoder.

**SAXS** Small Angle X-ray Scattering.

**SGD** Stochastic Gradient Descent.

**SH** Scale Hyperprior Compression AE.

**VAE** Variational Autoencoder.

**WAXS** Wide Angle X-ray Scattering.



# 1 Introduction

## 1.1 Motivation

The self-organization of physical growth processes, a process in which one or more structures grow over time, has become a research focus. Au atoms, for example, tend to self-organize by forming three-dimensional clusters on surfaces [55, p. 23]. This means that each gold atom strives to minimize its energy and settles at the point of the greatest possible potential, which leads to the self-organized process of forming bonds with other gold atoms. From these bonds emerge nanostructured surfaces. These nanostructured surfaces can lead to new and improved material properties, which can be used to make progress in special technical applications. The gold nanostructure growth process has become a focal point of research and is important for the industry. The optoelectronic, electrical, and catalytic properties of gold are the reasons for its wide range of applications [56]. These range from catalysis [22] to biosensors [43] and solar cells [9]. To achieve progress in the efficient and controlled production of gold-based nanotechnology with application-specific, tailor-made properties, it is essential to thoroughly understand the influence of growth kinetics on the surface morphology of the gold cluster [56].

To understand this influence, the self-organization of the gold cluster during sputter deposition is of great interest. Sputter deposition is a coating technique for the controlled fabrication of active gold nanostructures on surfaces using an adjustable deposition rate of gold atoms [56]. The role of the deposition rate for the fabrication of customized gold nanostructures was discussed in Ref. [57]. The investigation of the surface structure is often carried out by Small Angle X-ray Scattering Under Grazing Incident Angles (GISAXS), which today – due to technical developments in the field of synchrotron radiation – allows time-resolved and non-destructive *in situ* measurements in milliseconds with a resolution in the nanometer range [57]. The electron density distribution of a surface structure can be measured by irradiating a surface structure, and averaged depth information about morphological parameters can be obtained. The interaction

of the X-ray beam with the electrons on the surface structure leads to reflection and scattering of the X-ray beam. Thereby, the scattered wave's amplitude is detected by a two-dimensional detector, leading to a scattering pattern in reciprocal space. However, the wave's phase is lost, creating an inverse mathematical problem. As a result, the resulting GISAXS scattering image of the detector cannot be used to draw one-to-one conclusions about the surface structure. This complicates the analysis of the scattering patterns in the GISAXS scattering image, which is necessary to obtain information about the morphology of the surface structure [25]. Alterations in the GISAXS scattering pattern correspond to quantitative changes in surface morphology following sputtering [56]. Parameters are determined to capture changes in the surface morphology of gold nanostructures. These parameters provide an overview of the shape and size of the morphology.

Today, GISAXS experiments produce a huge amount of data [23]. This makes a simple calculation of the surface structure from the GISAXS scattering pattern analytically impossible, not least due to the inverse mathematical problem, the time-consumption, and the inefficiency for large amounts of data [25]. Currently, the process of analyzing GISAXS scattering data includes the following steps:

1. running a GISAXS experiment
2. performing GISAXS simulation with software packages such as *IsGISAXS* [33]
3. comparing the simulated scattering pattern with the experimental scattering pattern [25].

If no agreement between the scattering patterns can be found, the simulation parameters are varied until a match is found. In addition to simulations, other methods are used to investigate surfaces in real space. This procedure is undoubtedly laborious and time-consuming. Therefore, investigations have been conducted to automate and accelerate this process by Artificial Intelligence (AI).

Two AI architectures are of great interest for the classification of GISAXS scattering images. Convolutional Neuronal Networks (CNNs) are usually great for classification. But when used for the classification of GISAXS scattering images, CNNs run into the limitation that the experimental GISAXS scattering images contain noise while the simulation does not. To close the simulation-to-real gap between the simulation and the experimental GISAXS scattering images, two options exist: adding noise to the simulations

(data augmentation) or trying to remove the noise from the experimental data (denoising). Denoising can be performed with the help of Autoencoder (AE) [21, p. 499-523]. AEs are self-supervised NNs. They are divided into an encoder and a decoder network and are used to learn the identity function of the input. Several architectural variations exist: preventing AEs from learning the identity function or improving the ability to capture important information and create a richer learning representation. One of these variations is Denoising Autoencoder (DAE) [32, 64] [21, p. 504f, 507-512]. The DAE removes noise in an image during the decoding process by learning the characteristics of the image without noise and storing them in its latent space. Another variation is Sparse Autoencoder (SAE)[48, 40], which tries to limit the number of hidden units active simultaneously. In addition, AEs [7, 5] can compress and decompress images using the latent space as compressed information, the encoder to compress the images, and the decoder for decompression.

Although the usage of CNNs for the classification of GISAXS scattering images has and still is used for experiments [1], not much research has been done on the usage of AEs in the field of GISAXS. This is why this work will focus on the use of AEs to remove noise, reconstruct the images (reconstruction of the detector gaps), and compress the GISAXS scattering images.

### 1.2 Goal

This work aims to see whether AE can generate clusters in the latent space that help to classify GISAXS scattering images. For example, is it possible that we can train AE in such a way that all GISAXS scattering images with the same radius have the same active neuron in the latent space? To answer this question, we will train an AE on a dataset of cropped GISAXS simulation images of the size  $256 \times 256$ . These cropped images are slightly larger than the cropped images used for the classification, which uses only image detail of the size of  $260 \times 220$  pixels.

If we are interested in using fully connected (FC) layers, we either need a lot of memory or generate filters and compress the image using convolutional layers. Since the Deutsches Elektronen-Synchrotron (DESY) started to discuss the need to save hardware memory, we need to compress the images to employ FC layers. Given that AE can compress images, we will evaluate their compression performance. We will also check whether

we can use these architectures as a basis for our AE architecture and how far we can compress GISAXS scattering images.

We will also test whether we can remove randomly added noise for the simulation images and reconstruct the information lost by adding detector gaps required to protect the detector from too many X-rays.

For this work, we will only focus on simulation images because we possess both a noise-free image without detector gaps and a noise-free version with detector gaps. For the experimental images only noisy images with detector gaps exist. The latter would require a longer and more complex evaluation and reliance on the DESY researchers to evaluate whether the decompressed and noise-free output without detector gaps is a correct reconstruction by the AE. In the case of simulation images, this comparison can partially be done via evaluation metrics since the original image exists and by comparing the output image to the original version before noise and detector gaps were added.

With this in mind, we came up with the following three research questions:

1. Can we combine compression AE architectures with our architecture, and how far can we compress the images without losing information when decompressing?
2. Is it possible to employ de-noise and image reconstruction to remove noise and the detector gaps from the simulation images?
3. What clusters does an AE generate if we allow only one active neuron in the latent space while having a latent space size of, e.g., the number of different images in the dataset?

Should this lead to satisfactory results, we will start experimenting with experimental GISAXS scattering images.

### 1.3 Structure

The Master thesis is structured as follows.

In chapter 2 we present a review of related works. It gives an overview of the different architectures and tasks that an AE can perform, and of the research done with AE on image compression, denoising, reconstruction, and clustering. It also contains an

introduction to Au-Cluster Physics, the evaluation with GISAXS, and how AI has been used in the field of scattering images.

Chapter 3 explains the method proposed for our model, which tries to compress, denoise, reconstruct, and cluster GISAXS simulation images.

Chapter 4 contains information on the dataset and explains its preparation for the experiments. It also describes the experiments, as well as evaluates and discusses the results of the experiments.

Chapter 5 concludes the thesis and contains the findings. In addition, it gives an idea of things that we could improve.

## 2 Previous Work

This chapter explains the theoretical foundations relevant to this work, which are intended to help answer the research questions posed in the Introduction in Section 1.2. The chapter can be divided into six parts. The first part gives an overview of what an AE is and the different AE-architectures of interest for this work. The second through fourth parts outline various autoencoder architectures and approaches for image compression, denoising, reconstruction, and latent space clustering. The fifth part describes the process from the GISAXS experiment to the morphological identity parameters in the GISAXS scattering images. The physical principles are discussed, focusing on the material system of gold as a deposited material on the silicon substrate. The last part addresses related work where deep learning was applied to scattering images.

### 2.1 Autoencoder (AE)

An AE is a NN trained to attempt to learn an identity function. Internally, it has a hidden layer that describes the **code** or **latent space/vector** used to represent the input. The architecture of AEs is divided into two parts: an encoder  $\mathbf{h} = f(\mathbf{x})$  and a decoder that produces a reconstruction  $\mathbf{r} = g(\mathbf{h})$  (see Fig. 2.1). Although AEs try to learn the identity function ( $g(f(\mathbf{x})) = \mathbf{x}$ ) they are unable to do so, and this would also not be useful. This is partially due to restrictions that allow only the approximation of an exact copy of the input or the reliance on training data that will allow AEs only to copy inputs that resemble training data. This allows AEs to prioritize the aspects of the input that it can copy, resulting in the learning of useful features of the training data [21, p. 499].

Today, AEs have generalized the idea of an encoder and a decoder beyond deterministic functions to stochastic mappings  $p_{\text{encoder}}(\mathbf{x}|\mathbf{h})$  and  $p_{\text{decoder}}(\mathbf{h}|\mathbf{r})$  [21, p. 499].

Fig. 2.2 summarizes the different AE architectures and categories.

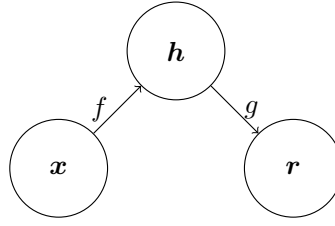


Figure 2.1: The general structure of an AE, mapping an input  $\mathbf{x}$  to an output (called reconstruction)  $\mathbf{r}$  through an internal representation or code  $\mathbf{h}$ . The AE has two components: the encoder  $f$  (mapping  $\mathbf{x}$  to  $\mathbf{h}$ ) and the decoder  $g$  (mapping  $\mathbf{h}$  to  $\mathbf{r}$ ). [21, p. 500]

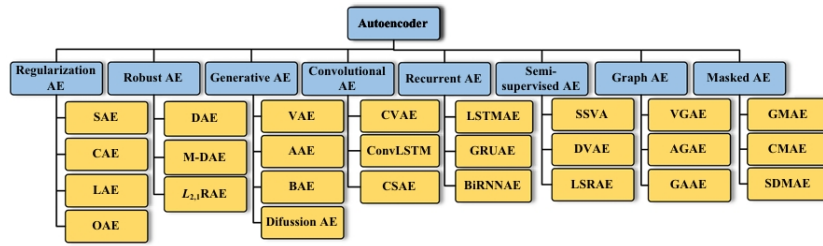


Figure 2.2: Taxonomy of AE architectures categorized by network structure.

Source: Berahmand et al. [8, p. 28] (Copyright © 2024, Springer International Publishing).

### 2.1.1 Vanilla Autoencoder

Rumelhart et al. [53] was the first to introduce the concept of AE as a way to learn and reconstruct input data and help obtain an "informative" data representation. This is done by encoding the input data in a compressed and semantically meaningful form and decoding it to a faithful reconstruction of the original input data. "Vanilla" [8] describes the simplest form of AE. This form of AE does not have additional complexities or architectural variations. A vanilla AE typically consists of an input layer, one or more hidden layers, and an output layer [74]. In addition, the decoder uses the same reverse layers as the encoder. As such, the architecture is symmetrical. The visualized structure of a vanilla AE can be seen in Fig. 2.3.

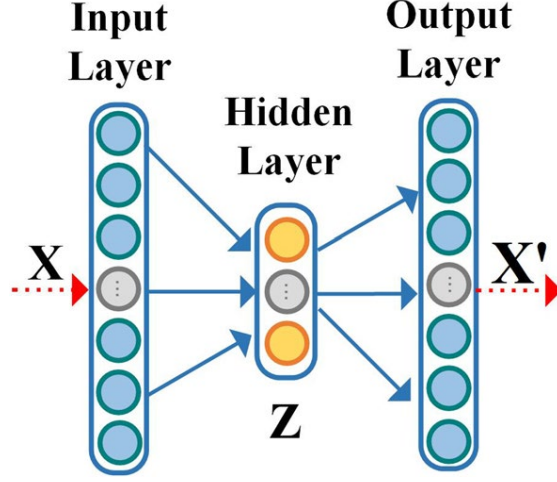


Figure 2.3: Illustration of the structure of an AE, where  $X$  represents the input data of the input layer,  $Z$  represents the data in the hidden layer, and  $X'$  represents the reconstructed output data in the output layer.

Source: Berahmand et al. [8, p. 12] (Copyright © 2024, Springer International Publishing).

Following the example in Fig. 2.3, AE maps the input vector  $X$  to the code vector  $Z$  during the encoder step using the function  $f_\theta$ . During the decoder step, the input vector  $Z$  is mapped back to the output vector  $X'$ , to reconstruct the input vector  $X$  using the function  $g_\theta$ . During training, AEs adjust the network weights ( $W$ ) through fine-tuning. This is done by minimizing the reconstruction error  $L$  between  $X$  and the reconstructed data  $X'$ . This reconstruction error  $L$  is the loss function for optimizing the network parameter during training. Thus, the objective function of AE can be written as follows [8]:

$$\min_{\theta} J_{AE}(\theta) = \min_{\theta} \sum_{i=1}^n l(x_i, x'_i) = \min_{\theta} \sum_{i=1}^n l(x_i, g_\theta(f_\theta(x_i))) \quad (2.1)$$

where  $x_i$  represents the  $i$ -th dimension of the training sample,  $x'_i$  the  $i$ -th dimension of the output data, and  $n$  is the total amount of training data. " $l$ " refers to the reconstruction error between the input and output, defined as [8]:

$$L(X, X') = \sum_{i=1}^n \|X_i - X'_i\|^2 \quad (2.2)$$



The encoder and decoder mapping functions are

$$\begin{aligned} Z &= f_\theta(X) = s(WX + b) \\ X' &= g_\theta(Z) = s(W'Z + b') \end{aligned} \tag{2.3}$$

where " $s$ " is a non-linear activation function like sigmoid or Rectified Linear Unit (activation function) (ReLU).  $W$  and  $W'$  are weight matrices and  $b$  and  $b'$  are bias vectors. During training, the weights and biases of AEs are adjusted to minimize the reconstruction error  $L$  using an optimization algorithm such as stochastic gradient descent. Once trained, the encoder function  $f_\theta$  can create low-dimensional representations ( $Z$ ) of new input data ( $X$ ), while the decoder function  $g_\theta$  can reconstruct the original data from the low-dimensional representation ( $X'$ ).

### 2.1.2 Stacked Autoencoder

Traditionally the vanilla AE only employs a single-layer encoder, making it difficult to extract deep features. If we want to improve feature extraction, an effective strategy is to deepen the structure of the NN. This can be achieved by employing a layer-wise learning approach in which multiple vanilla AEs can be stacked together to form a SAE. This allows for the extraction of complex data features. The training process of SAE involves learning a condensed data representation for each AE. The final output is then obtained by combining the outputs of these AEs.

Typically, training a SAE follows a layer-wise approach [27, 26]. After the first layer is trained, it serves as the input and output for the training of the second layer, and so on. During the reconstruction loss evaluation, the second layer's loss is assessed relative to the first layer and not the input layer.

The encoding process can be mathematically represented as follows [8]:

$$a^k = f(W_e^k a^{k-1} + b_e^k), \quad k = 1 : n \tag{2.4}$$

where  $k$  represents the  $k$ -th AE,  $a_k$  the encoding outcome of the  $k$ -th AE, and when  $k = 1$ ,  $a_0 = x$  denotes the input data.

The decoding process can be mathematically represented as follows [8]:

$$c^k = f\left(W^{n-(k-1)} c^{k-1} + b^{n-(k-1)}\right), \quad k = 1 : n \tag{2.5}$$

where  $c_0 = a_n$ , when  $k = 1$ , and  $c_n = \hat{x}$ , when  $k = n$ .  $\hat{x}$  represents the reconstructed data of the input variable  $x$  [27].

### 2.1.3 Hyperparameters in Autoencoder

AEs come with various hyperparameters. They must be defined before the training begins, and their values can significantly influence the model's performance. While some hyperparameters are set before the training and remain constant throughout, others can be dynamically tuned during training to optimize the model's performance. Selecting and adjusting hyperparameters often involves experimentation and validation to achieve the best results for a particular task. The following outlines the most common hyperparameters in AEs, according to Berahmand et al. [8]:

- **Number and Type of Hidden Layers:** The number of hidden layers within the AE and their type is configured before training. They define the depth of the network and its capacity to capture intricate data patterns. Adding more hidden layers can enhance the symbolic power, but also elevates the risk of overfitting, and introduces challenges for optimization.
- **Number of Neurons in Each Layer:** The number of neurons in each layer is normally set before training but can be modified during training. It is also possible that some neurons are forcibly set to zero during training. The number is responsible for the network's data representation capacity. Similarly as with the number of hidden layers, a higher count of neurons can amplify the network, but also elevate the risk of overfitting and complicate the optimization process.
- **Size of the Latent Space:** This parameter is set before training. It defines the size of the "bottleneck" layer and permits fine-tuning of the balance between the model complexity and its performance.
- **Activation Function:** The activation functions determine the non-linearity of the network and its ability to learn intricate data patterns. Common activation functions employed in the bottleneck layers include sigmoid, tanh, ReLU, and SELU. Further details, including their equations, outputs, and output curves, are given in Fig. 2.4.

**Table 3** Activation functions

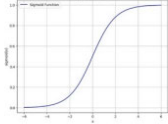
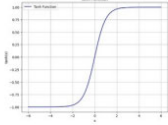
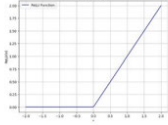
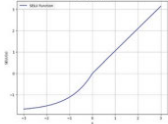
Activation Function	Equation	Output	Output Curve
Sigmoid	$f(x) = \frac{1}{1+e^{-x}}$	$[0,1]$	
Tanh	$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	$[-1,1]$	
ReLU	$f(x) = \max(0, x)$	$[0, \infty]$	
SELU	$f(x) = \lambda \begin{cases} x & \text{if } x > 0 \\ \alpha e^x - \alpha & \text{if } x \leq 0 \end{cases}$	$[-2, \infty]$	

Figure 2.4: Common activation Functions in NN.

Source: Berahmand et al. [8, p. 12] (Copyright © 2024, Springer International Publishing).

- **Objective Function (Loss Function):** The objective function, also known as the loss function, is a critical part of AE. It minimizes the distinction between input and output data during network training. The objective function depends on the type of data and the specific application and is generally determined before training. Common objective functions used in AEs include:

- **Mean Squared Error (MSE):** This is the predominant objective function in AEs, measuring the average squared difference between the input and output data. MSE is defined by the formula:

$$L_{AE}(X, X') = \min \left( \|X - X'\|_F^2 \right) \quad (2.6)$$

- **Binary Cross-Entropy (BCE):** Binary Cross-Entropy (BCE) is employed when the input data is binary (0 or 1). This function measures the difference between predicted and actual output in binary cross-entropy loss. This cross-

entropy is defined as:

$$L_{AE}(X, X') = - \sum_{i=1}^n (x_i \log(x'_i) + (1 - x_i) \log(1 - x'_i)) \quad (2.7)$$

When choosing the loss function, it is necessary to consider what the AE is supposed to achieve. MSE suits regression tasks, offering robustness against outliers but is sensitive to data scaling. It is also useful for comparing images. BCE is for binary classification but can be numerically unstable near 0 or 1 probabilities. The choice depends on the problem and task requirements. MSE is the predominant loss function.

- **Optimization Algorithm:** AEs utilize optimization algorithms to minimize the loss function during training. The optimization algorithm is responsible for adjusting the weights and biases for effective training. The choice of the algorithm is made before the training but may involve hyperparameter tuning during training. It is possible to choose multiple algorithms if different loss functions are used. The most common optimization algorithms are Stochastic Gradient Descent (SGD), Adam, and Adagrad.

- **Stochastic Gradient Descent (SGD):** A widely used algorithm that updates the network parameters after processing small batches of data. It is computationally efficient but may converge slowly for complex models and datasets. Careful tuning of initial learning rates is often needed.
- **Adam:** Combines features of SGD with adaptive learning rates and momentum to accelerate convergence. This reduces the risk of getting stuck in local minima. It requires tuning of hyperparameters like  $\beta_1$  and  $\beta_2$  and is suitable for non-stationary and noisy objectives.
- **Adagrad:** An adaptive algorithm to adjust learning rates based on the frequency of update of the parameters. Effective for sparse data, it can lead to quick convergence, but may also converge prematurely and face challenges with nonconvex optimization.

The algorithm choice depends on the data set's size, the model's complexity, the loss function, and the computational resources.

- **Learning Rate:** The learning rate dictates the step size during the optimization. It influences the updates of the weights and biases of the hidden layers, as well as the convergence speed of the loss function. High values may cause overshooting, but low values can lead to local minima trapping and a longer training time. Although the learning rate is pre-set at the beginning of the training, it can be adjusted with the help of a scheduler.
- **Number of Epochs:** An Epoch is one training iteration, which means that all batches of the training dataset have been used for training. The more epochs used for training, the better the model accuracy can be. However, if the number of epochs is set too high, the risk of overfitting increases. Early stopping can be used to curb overfitting which checks the validation loss after each epoch and stops if the loss increases over a given number of epochs.
- **Batch Size:** The batch size determines the size of the subset of the data set used in any step during the training. The size affects the gradient noise and the optimization efficiency. A small size yields a noisier gradient, but faster, memory-efficient optimization, while a larger size has stable gradients, but slower, memory-intensive optimization.

### 2.1.4 Regularized Autoencoder

Regularized AE is a NN architecture that extracts a compressed representation of input data while enforcing regularization constraints. These constraints encourage the formation of a low-dimensional discriminative feature space [8].

#### Sparse Autoencoder (SAE)

Sparse Autoencoders (SAEs) employ the idea of **Sparse Coding** and try to reduce the number of hidden units active simultaneously. To do this, SAEs try to reduce the weight of most hidden units to or close to zero so that they are considered inactive. There are two ways to achieve this. One way is to select the  $k$  highest activations of the latent space and set the weights for the rest to zero. This approach is called **k-sparse AE** [40].

The other approach adds a sparsity penalty or regularization loss to the loss function. This sparsity penalty depends on the amount of active hidden units using the weight of

these units. During the backpropagation step, these weights are continuously decreased until most are either zero or close to zero for each of the different inputs of the training set. These AEs are called **SAE** [48].

**$k$ -Sparse Autoencoder** The  $k$  in  $k$ -sparse AEs denotes the number of hidden units left active in the latent space, while all other hidden units are set to zero. Due to this,  $k$  is also called the sparsity level of AEs. The functioning of  $k$ -sparse AE can be seen in Fig. 2.5.

$k$ -Sparse Autoencoders:
<b>Training:</b> 1) Perform the feedforward phase and compute $\mathbf{z} = W^\top \mathbf{x} + \mathbf{b}$ 2) Find the $k$ largest activations of $\mathbf{z}$ and set the rest to zero. $\mathbf{z}_{(\Gamma)^c} = 0 \quad \text{where} \quad \Gamma = \text{supp}_k(\mathbf{z})$ 3) Compute the output and the error using the sparsified $\mathbf{z}$ . $\hat{\mathbf{x}} = W\mathbf{z} + \mathbf{b}'$ $E = \ \mathbf{x} - \hat{\mathbf{x}}\ _2^2$ 3) Backpropagate the error through the $k$ largest activations defined by $\Gamma$ and iterate. <b>Sparse Encoding:</b> Compute the features $\mathbf{h} = W^\top \mathbf{x} + \mathbf{b}$ . Find its $\alpha k$ largest activations and set the rest to zero. $\mathbf{h}_{(\Gamma)^c} = 0 \quad \text{where} \quad \Gamma = \text{supp}_{\alpha k}(\mathbf{h})$

Figure 2.5: Overview of the  $k$ -sparse AEs including the algorithm for the training.  
Source: Makhzani and Frey [40, p. 2] (arXiv preprint arXiv:1312.5663, 2014).

Two possible ways exist to achieve the sparsity level of the chosen  $k$ . For the first way,  $k$  maintains the same value throughout the training process. The second way starts with a low sparsity level (the number of active latent space units is close to or equal to the size of the latent space). Then the sparsity level increases to  $k$  (the number of active latent space units decreases to  $k$ ).

**Sparsity Penalty (sparsity regularization loss)** The use of an SAE with a sparsity penalty changes the loss function  $L$  from

$$L(\mathbf{x}, g(f(\mathbf{x}))) \tag{2.8}$$

which is the loss function for the AE to [21]

$$L(\mathbf{x}, g(f(\mathbf{x}))) + \lambda \Omega(\mathbf{h}) \quad (2.9)$$

where  $\lambda > 0$  measures the amount of sparsity the SAE wants to enforce,  $\Omega(\mathbf{h})$  is the penalty function and  $\mathbf{h}$  the code layer plus potentially one or more additional hidden layer of the encoder or decoder [48] [21, p. 502-504]. The most common sparse penalty functions are the Kullback-Leibler (KL) divergence or the L1 norm.

*L1 Regularization* If the L1 norm, also called L1 regularization, is used then the penalty function  $\Omega(\mathbf{h})$  is

$$L1 = \lambda * \sum |w_i| \quad (2.10)$$

where  $\lambda$  is called the regularization parameter, and  $w_i$  the  $i$ -th activation weights. This leads to the following loss function:

$$L(\mathbf{x}, g(f(\mathbf{x}))) + \lambda * \sum |w_i| \quad (2.11)$$

*KL-Divergence* The KL-divergence is defined as

$$KL(\rho || \hat{\rho}) = \rho * \log\left(\frac{\rho}{\hat{\rho}}\right) + (1 - \rho) * \log\left(\frac{1 - \rho}{1 - \hat{\rho}}\right) \quad (2.12)$$

where  $\hat{\rho}$  is the current value and  $\rho$  the "desired" sparsity. With this, the loss function looks as follows:

$$L(\mathbf{x}, g(f(\mathbf{x}))) + \beta * KL(\rho || \hat{\rho}) \quad (2.13)$$

### 2.1.5 Robust autoencoder

Robust AEs are utilized to enhance the robustness of AEs when dealing with noisy or corrupted input data. They are valuable when the input has noises, outliers, or imperfections. These issues are commonplace in real-world datasets such as healthcare, finances, and sensor networks. In these cases, robust AEs effectively remove imperfections in the data while retaining valuable information from the data.

### Denoising Autoencoder (DAE)

Denoising Autoencoders (DAEs) [64, 32] are designed to reconstruct clean data from noisy input. This is done by introducing noise during training. The primary objective is to minimize the dissimilarity between the clean data and the reconstructed output. DAEs achieve this by intentionally corrupting the input data with various forms of noise and minimizing the differences between the clean and original input data and the reconstruction and denoising of the corrupted input data. This allows DAEs to discern valuable features within the input data and disregard noise and irrelevant information. Due to this aspect of DAEs, they are additionally used, even if there is no noise in the data set, to help with feature extraction. The DAE loss function is expressed as follows:

$$L(\mathbf{x}, g(f(\hat{\mathbf{x}}))), \quad (2.14)$$

where  $\hat{\mathbf{x}}$  is a corrupted version of  $\mathbf{x}$ . As such, DAEs learn a structural representation of the input data.

### Marginalized Denoising Autoencoder (M-DAE)

The Marginalized Denoising Autoencoder (M-DAE) [12] is a specialized version of DAEs. It is designed to handle data sets with missing or incomplete features. Same as a DAE, the M-DAE reconstructs clean input data from noisy versions. This is achieved by restoring clean data from corrupted counterparts. The corrupted counterparts are obtained by intentionally subjecting the input data  $X$  to random corruption. This is done by setting each feature to 0 based on the probability  $p$  and creating these corrupted versions called  $\hat{X}_i$ . The primary goal of M-DAEs is to minimize the loss function [8]:

$$L_{\text{M-DAE}}(X, X') = \min \left( \frac{1}{m} \sum_{i=1}^m \|X - \hat{X}'_i W\|_F^2 \right) \quad (2.15)$$

where  $W$  signifies the learned transformation matrix, and  $m$  the total number of input examples.

The M-DAE seeks the best solution for  $W$ , which can be expressed mathematically as:

$$W = E[Q^{-1}]E[P] \quad (2.16)$$



Here  $E[Q^{-1}]$  is based on the inverse of the expected  $Q$  and  $E[P]$  on the expected  $P$ . The expectations are calculated following specific formulas that involve the covariance matrix of the uncorrupted data  $X$ .

### 2.1.6 Generative Autoencoder

Generative AEs do not focus on dimensionality reduction as traditional AEs do; instead, they try to learn the underlying probability distribution of the data. This allows AEs to generate new data samples based on the training data, which helps in tasks like image and text generation.

#### Variational Autoencoder (VAE)

A Variational Autoencoder (VAE) [2] learns to represent data in a lower-dimensional latent space and generates new data samples that resemble the input. Because of this VAEs are generative models that can capture the underlying distribution of input data. The VAE does not encode the input data in a fixed latent space representation  $Z$  instead it maps the input data to a posterior distribution  $q(Z|X)$ . During decoding,  $Z$  is sampled from the distribution  $q(Z|X)$  and then passed through the decoder. The regularization loss in VAE encourages  $q(Z|X)$  to match a specific distribution, often a standard Gaussian. The VAE loss function is defined as [8]:

$$L_{\text{VAE}} = -E(q(Z|X)) [\log p(X|Z)] + \text{KL}(q(Z|X)||p(Z)) \quad (2.17)$$

The first part of the function measures the difference between the original input data and the reconstructed data. The second term is a regularization component and quantifies the KL divergence between  $q(Z|X)$  and  $p(Z)$ , which is typically a standard Gaussian distribution. With this loss function, VAE can balance accurate data reconstruction and create a structured latent space for generative purposes.

#### Diffusion Autoencoder (DiffusionAE)

Diffusion Autoencoders (DiffusionAEs) [51] are specifically designed for generative modeling tasks. They are based on diffusion models and designed to capture a complex data distribution. Data is subjected to a progressive denoising process for DiffusionAE. This

allows the model to grasp complex data patterns effectively. DiffusionAE employs a unique loss function known as the Diffusion Probability Loss, which guides the training by modeling how the data evolves. The loss function looks as follows [8]:

$$L(X, X') = -\log P(X|X') \quad (2.18)$$

where  $P(X|X')$  is the conditional probability of observing the original data  $X$  given the reconstructed data  $X'$ . The primary objective of DiffusionAE is to generate  $X'$  that closely resembles the original data  $X$ .

### 2.1.7 Convolutional Autoencoder (ConvAE)

Convolutional Autoencode (ConvAE) [58] switches the fully connected layers in both the encoder and the decoder used by a traditional AEs to convolutional layers. The encoder uses convolutional layers to create a compact representation of the input images and the decoder to reconstruct the image. The decoder uses deconvolution layers instead of convolution layers, also called transposed convolution [73]. ConvAEs are especially effective for image data, as convolutional layers excel at capturing spatial dependencies. These are patterns and relationships between pixels or locations within individual images or data frames. They are used in tasks like image denoising, inpainting, segmentation, and super-resolution.

### Convolutional Sparse Autoencoder (ConvSAE)

Convolutional Sparse Autoencoder (ConvSAE) [39] combines ConvAE principles with techniques that induce sparsity. This includes max-polling and feature channel competition. ConvSAE includes a sparsifying module that creates sparse feature maps. The module retains the highest value and corresponding position within each local subregion and performs unpooling primarily through max pooling. The loss function used in ConvSAE is based on the Forbenius norm and quantifies the difference between the original input and the reconstructed output. The function is defined as follows [8]:

$$L_{CSAE}(X, X') = \min \sum_{l=1}^L \left( \left\| X^{(l)} - X'^{(l)} \right\|_F^2 \right) \quad (2.19)$$

$$X'^{(l)} = \sum_{i=1}^d (\text{rot}(W_i, 180) * Z_i') + c_i \quad (2.20)$$

$$Z^{(l)} = G_{p,s} \left( Z_i^{(l)} \right) = G_{p,s} \left( f(W_i \cdot X^{(l)} + b_i) \right) \quad (2.21)$$

where  $l$  is the number of layers,  $Z^{(l)}$  represents the original input at layer  $l$ ,  $X'^{(l)}$  the reconstructed output at layer  $l$ .  $d$  is the number of feature channels,  $Z_i^{(l)}$  is the  $i$ -th sparsified feature map, and  $G_{p,s}(X)$  represents the sparsifying operator, involving max-pooling and unpooling operations to create sparse feature maps.

### 2.1.8 Semi-Supervised Autoencoder

Semi-supervised AEs utilize labeled and unlabeled data to enhance feature learning. This is especially done in scenarios where only a limited amount of labeled data exists. The primary objective of semi-supervised AEs is to facilitate extracting crucial latent features using the available labeled data. These features can then be used for tasks such as clustering and classification [71]. Semi-supervised AEs prove to be highly valuable if labeled data are scarce because it enables the exploitation of abundant unlabeled data. This is a common feature in real-world applications.

#### Label and Sparse Regularized Autoencoder (LSRAE)

LSRAE [11, 8] combines label and sparse regularizations and creates a semi-supervised learning method. LSRAE leverages the strength of unsupervised and supervised learning. Sparse regularization enhances extracting localized and informative features by selectively activating a subset of neurons. This helps uncover underlying data concepts, which improves generalization. Label regularization improves the categorization accuracy by enforcing the extraction of features aligned with category rules. The loss function of LSRAE is defined as follows [8]:

$$L_{LSRAE}(X, X') = \min \left( \|X - X'\|_F^2 + KL(p \parallel q) + \sum_{i=1}^d \sum_{j=1}^l (W_{ij})^2 + \sum_{i=1}^n \|L - T\| \right) \quad (2.22)$$

$\|X - X'\|_F^2$  ensures the precise data reconstruction,  $KL(p \parallel q)$  promotes sparsity within the hidden layers, resulting in efficient feature extraction.  $\sum_{i=1}^d \sum_{j=1}^l (W_{ij})^2$  acts as a safeguard against overfitting by penalizing excessive weights and  $\sum_{i=1}^n \|L - T\|$  improves classification accuracy by quantifying the label error. In this function,  $L$  denotes the label returned by the LSRAE and  $T$  the desired label.

## 2.2 Compression

JPEG [67], JPEG2000 [59], HEVC [62], AV1 [13] and VVC [10], conventional lossy image compression methods, have iteratively improved on a similar coding scheme. They partition images into blocks of pixels, use the transform domain to decorrelate spatial frequencies with linear transforms (e.g., DCT or DWT), perform some predictions based on neighboring values, quantize the transform coefficients, and finally encode the quantized values and the prediction side-information into bit-stream with an efficient entropy coder (e.g., CABAC [41]) [6]. On the other hand, NN encodings based on NN rely on learned analysis and synthesis of non-linear transforms. The encoder maps the pixel values to a latent representation via an analysis transform. The latent space is then quantized and (losslessly) encoded in entropy. The decoder consists of an approximate inverse or synthesis transform and converts the latent representation to pixel values [6].

Learning complex non-linear transforms based on CNN enables NN-based codecs to match or outperform conventional approaches. These codecs aim to minimize the estimated length of the bitstream while keeping the reconstructed image’s distortion low compared to the original content. The distortion can be measured with metrics like Mean Squared Error (MSE) or Multi-Scale Structural Similarity (MS-SSIM) [70].

### 2.2.1 CompressAI

CompressAI [6] is a platform that aims to implement the most common operations needed to build deep NN architectures for data compression and provides evaluation tools to compare learned methods with traditional codecs. It is implemented using PyTorch and can be used for end-to-end compression. This implementation contains several domain-specific layers, operations, and modules, such as entropy models, quantization operations, and color transforms. Various architectures of state-of-the-art learning image compression have been implemented in PyTorch (see Table 2.1).

Model	Description
bmshj2018-factorized	Factorized prior [5]
bmshj2018-hyperprior	Hyperprior [5]
mbt2018-mean	Hyperprior with a Gaussian mixture model [46]
mbt2018	Joint autoregressive and hyperprior [46]
cheng2020-anchor	Extension from [46], residual blocks and sub-pixel deconvolution [14]
cheng2020-attn	Extension from cheng2020-anchor with attention blocks [14]

Table 2.1: Re-implemented models from the state-of-the-art on learned image compression currently available in CompressAI. Training, fine-tuning, inference, and evaluation of the models listed in this table are fully supported. The rate-distortion performances reported in the original papers have been successfully reproduced from scratch [6].

PyTorch also comes with a set of quality settings that set weights for the loss functions MSE and MS-SSIM (see Table 4.6). These weights change the reconstruction loss’s importance compared to the bitstream’s estimated length. The quality setting "1" means that the estimated length of the bitstream is lower, but the reconstruction might be worse. In contrast, the quality setting "8" prioritizes reconstruction over the bitstream’s estimated length, leading to improved reconstruction at the possible cost of a longer bitstream. Table 2.3 shows the weighted versions of the loss functions.

Metric	1	2	3	4	5	6	7	8
MSE	0.0018	0.0035	0.0067	0.0130	0.0250	0.0483	0.0932	0.1800
MS-SSIM	2.40	4.58	8.73	16.64	31.73	60.50	115.37	220.00

Table 2.2: Lambda values used for training the networks are different bit-rates (quality setting from 1 to 8), for the MSE and MS-SSIM metrics [6]

Metric	Loss function
MSE	$\mathcal{L} = \lambda \times 255^2 \times D_{\text{MSE}} + \mathcal{R}$
MS-SSIM	$\mathcal{L} = \lambda \times (1 - D_{\text{MS-SSIM}}) + \mathcal{R}$

Table 2.3: Loss functions used for training the networks, with  $\mathcal{D}$  the distortion and  $\mathcal{R}$  the estimated bit-rate.

### 2.2.2 Factorized Prior Compression

The compression using the factorized prior architecture [5] is the basis of all the following architectures. It uses identical architectures for the analysis and synthesis transforms called  $g_a$  and  $g_s$ . The analysis transform  $g_a$  functions as an encoder in an AE architecture, while the synthesis transform  $g_s$  functions as the decoder. The first part of the architecture depicted in Fig. 2.7 shows the architecture for the Factorized Prior compression. The arithmetic encoder encodes the latent space into a string, and the arithmetic decoder decodes the string back to the latent space.

The analysis transform consists of four convolutional layers that use a  $5 \times 5$  Gaussian kernel and half the image size. Each convolutional layer uses Generalized Divisive Normalization Layer (GDN) [4] as an activation function. Ballé et al. [5] recommend 128 or 192 filters for the first three convolutional layers and 192 or 320 for the last convolutional layer. This assumes the input is described using 3 values per pixel (RGB image). The synthesis transform changes the convolutional layers to transposed convolutional layers and uses the inverse of GDN called Inverse Generalized Divisive Normalization Layer (IGDN). All layers except the last have a recommendation of 128 or 192 filters. The layer that returns the reconstructed image has the same number of filters as the values per pixel of the original image. The factorized prior model uses the entropy bottleneck [5] as an arithmetic encoder and decoder. All convolutional layers downsample the image by half. All transposed convolutional layers upsample the image to double size.

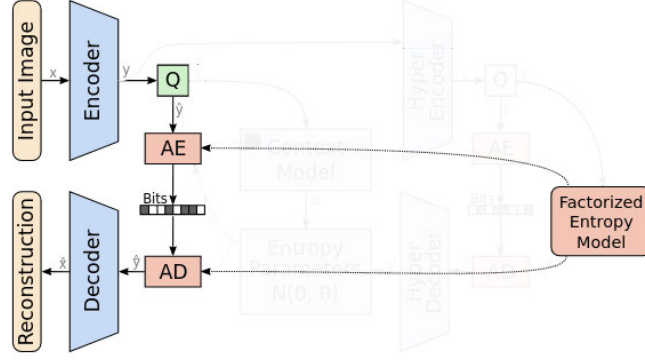


Figure 2.6: **Fully-Factorized Model** This model learns a fixed entropy model shared between the encoder and decoder systems. It is designed based on the assumption that all latents are Independent and Identically Distributed (IID) and ideally set to be a multinomial. In practice, it must be relaxed to ensure differentiability. The compression model is primarily an AE composed of convolutional layers and nonlinear activations, where the main complication is due to ensuring that the entropy model is differentiable and that useful gradients flow through the quantized latent representation. Source: Minnen et al. [46, p. 20] (arXiv preprint arXiv:1809.02736, 2018).

### 2.2.3 Scale Hyperprior Compression

The scale hyperprior architecture [5] was introduced in the same paper as the factorized prior architecture and uses it as a basis. In addition, it adds a hyperprior encoder and decoder. This model consists of two latent spaces. One after the analysis transform and the other after the hyperprior analysis transform. Fig. 2.7 shows the model architecture. The extension is used to capture the spatial dependencies of the compressed image.

The hyperprior analysis transform consists of three convolutional layers and uses ReLU as an activation function. The first convolutional layer uses a  $3 \times 3$  Gaussian kernel and the rest uses a  $5 \times 5$  kernel. For the hyperprior synthesis transform, it is the other way around with the first to transposed convolutional layers using a  $5 \times 5$  Gaussian kernel and the last a  $3 \times 3$  kernel. Both layers with a  $3 \times 3$  kernel do not employ downsampling or upsampling, the other layers use a stride of 2. The hyperprior analysis transforms downsamples and the hyperprior synthesis transform upsamples. The latent space of the hyperprior uses the entropy bottleneck as an arithmetic encoder and decoder. The other latent space uses a Gaussian conditional layer [5]. Before the hyperprior analysis

transform we make sure that the values from the factorized prior latent space have all positive values by using the absolute value.

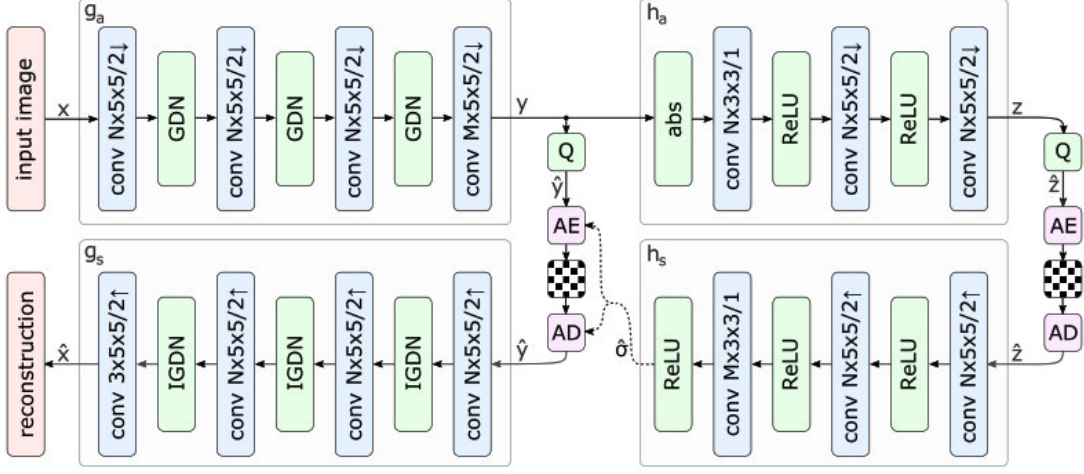


Figure 2.7: Network architecture of the hyperprior model. The left side shows an image autoencoder architecture and the right side corresponds to the autoencoder implementing the hyperprior. The factorized-prior model uses the identical architecture for the analysis and synthesis transforms  $g_a$  and  $g_s$ .  $Q$  represents quantization, and  $AE$ ,  $AD$  represent arithmetic encoder and arithmetic decoder, respectively. Convolution parameters are denoted as number of filters  $\times$  kernel support height  $\times$  kernel support width / down- or upsampling stride, where  $\uparrow$  indicates upsampling and  $\downarrow$  downsampling.  $N$  and  $M$  were chosen dependent on  $\lambda$ , with  $N = 128$  and  $M = 192$  for the 5 lower values, and  $N = 192$  and  $M = 320$  for the 3 higher values.

Source: Ballé et al. [5, p. 6] (arXiv preprint arXiv:1802.01436, 2018).



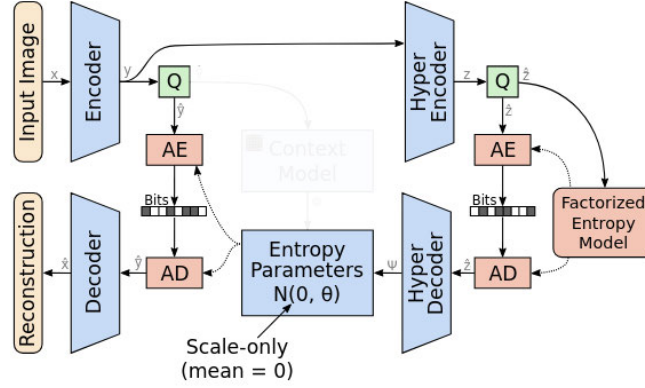


Figure 2.8: **Scale-only Hyperprior** This model uses a conditional Gaussian Scale Mixture (GSM) as the entropy model. The GSM is conditioned on a learned hyperprior, a (hyper-)latent representation formed by transforming the latent space using the Hyper-Encoder. The Hyper-Decoder can then decode the hyperprior to create the scale parameters for the GSM. The main advantage of this model is that the entropy model is image-dependent and can be adapted for each code. The downside is that the compressed hyperprior must be transmitted with the compressed latent space, which increases the total file size.

Source: Minnen et al. [46, p. 20] (arXiv preprint arXiv:1809.02736, 2018).

#### 2.2.4 Mean and Scale Hyperprior Compression

The mean and scale hyperprior model [46] is a variation of the scale hyperprior model. Instead of using ReLU for the hyper encoder and decoder, the mean and scale hyperprior model uses Leaky ReLU. It also changes the Gaussian conditional layer, using the mean as part of the entropy parameters.

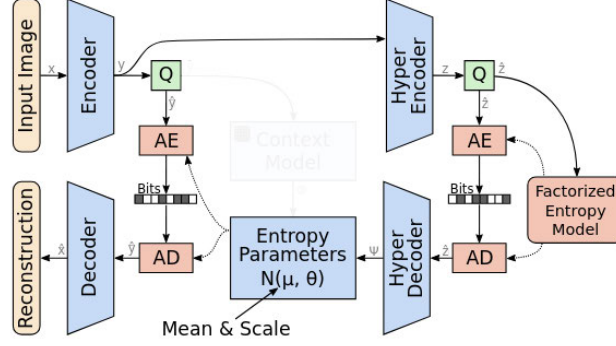


Figure 2.9: **Mean & Scale Hyperprior** This model variant is a simple extension of the scale-only hyperprior model shown in Fig. 2.8 in which the GSM is replaced with a Gaussian Mixture Model (GMM). The Hyper-Decoder is therefore responsible for transforming the hyperprior into mean and scale parameters of the Gaussians.

Source: Minnen et al. [46, p. 21] (arXiv preprint arXiv:1809.02736, 2018).

### 2.2.5 Joint Autoregressive and Hierarchical Priors Compression

The joint autoregressive and hierarchical priors model [46] is a variation of the mean and scale hyperprior model. It adds a context model consisting of a masked convolutional layer [63]. The context model takes the quantized output of the encoder as the input and passes its output to the Gaussian conditional layer. There, the output is concatenated with the output of the hyper encoder. The architecture is shown in Fig. 2.10 and the specific layers for each part of the architecture are shown in Table 2.4.

## 2 Previous Work

Encoder	Decoder	Hyper Encoder	Hyper Decoder	Context Prediction	Entropy Parameters
Conv: 5×5 c192 s2	Deconv: 5×5 c192 s2	Conv: 3×3 c192 s1	Deconv: 5×5 c192 s2	Masked: 5×5 c384 s1	Conv: 1×1 c640 s1
GDN	IGDN	Leaky ReLU	Leaky ReLU		Leaky ReLU
Conv: 5×5 c192 s2	Deconv: 5×5 c192 s2	Conv: 5×5 c192 s2	Deconv: 5×5 c288 s2		Conv: 1×1 c512 s1
GDN	IGDN	Leaky ReLU	Leaky ReLU		Conv: 1×1 c384 s1
Conv: 5×5 c192 s2	Deconv: 5×5 c192 s2	Conv: 5×5 c192 s2	Deconv: 3×3 c384 s1		
GDN	IGDN	Leaky ReLU			
Conv: 5×5 c192 s2	Deconv: 5×5 c3 s2				

Table 2.4: Each row corresponds to a layer of the generalized model. The convolution layers are specified with the 'Conv' prefix followed by the kernel size, number of channels, and downsampling stride (for example, the first layer of the encoder uses  $5 \times 5$  kernels with 192 channels and a stride of two). The 'Deconv' prefix corresponds to upsampled convolutions (that is, in TensorFlow, `tf.conv2d_transpose`), while the 'Masked' prefix corresponds to the masked convolution as in [63]. GDN stands for generalized divisive normalization, and IGDN is inverse GDN [4].

Source: Minnen et al. [46, p. 4] (arXiv preprint arXiv:1809.02736, 2018).

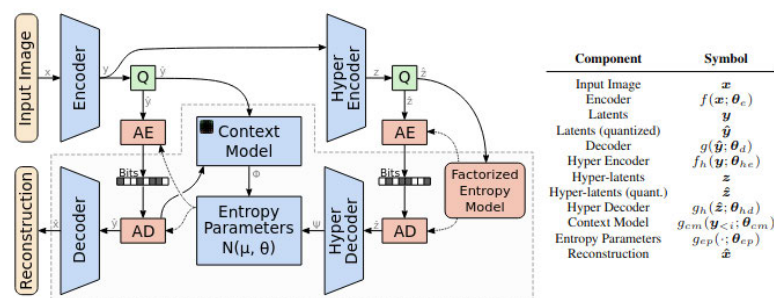


Figure 2.10: The joint autoregressive & hierarchical model jointly optimizes an autoregressive component that predicts latent spaces from their causal context (Context Model) along with a hyperprior and the underlying autoencoder. Real-valued latent representations are quantized ( $Q$ ) to create latent space ( $\hat{y}$ ) and hyper-latent space ( $\hat{z}$ ), which are compressed into a bitstream using an arithmetic encoder ( $AE$ ) and decompressed by an arithmetic decoder ( $AD$ ). The highlighted region corresponds to the components the receiver executes to recover an image from a compressed bitstream.

Source: Minnen et al. [46, p. 3] (arXiv preprint arXiv:1809.02736, 2018).

### 2.2.6 Residual Joint Autoregressive and Hierarchical Priors Compression

The residual joint autoregressive and hierarchical priors model [14] is a variation of the joint autoregressive and hierarchical priors model. The main difference between these two

models is the use of residual blocks [24] in the encoder  $g_a$  and the decoder  $g_s$ . Residual blocks were introduced as part of the ResNet architecture, allowing convolutional classification networks to perform better despite using fewer layers and making optimizing these networks easier than networks that relied on increased depth for better accuracy. In the case of compression AEs, the residual blocks increase the large receptive field and improve the rate-distortion performance. As seen in Fig. 2.11, the residual joint autoregressive and hierarchical priors model changes the Gaussian conditional layer. It uses a discretized Gaussian Mixture Model (GMM).

Furthermore, Cheng et al. [14] introduces a version of the residual joint autoregressive and hierarchical prior model that includes attention modules. The architecture for the attention modules can be seen in Fig. 2.13 and the overall architecture including the attention modules in Fig. 2.12. Attention modules are used because they improve performance for image restoration [75] and compression [35].

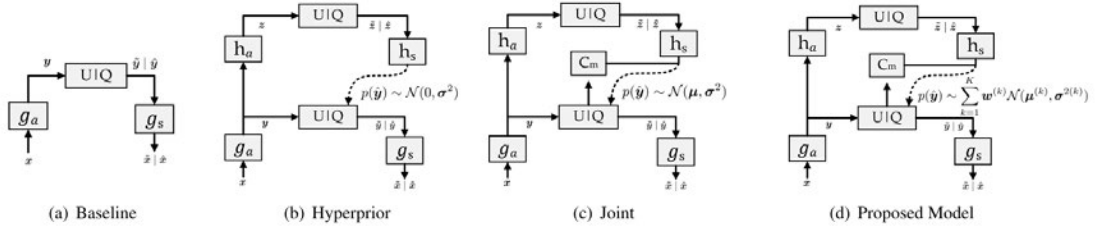


Figure 2.11: Operational diagrams of learned compression models (a)(b)(c) and proposed Gaussian Mixture Likelihoods (d).

Source: Cheng et al. [14, p. 3] (Copyright © 2020, IEEE)

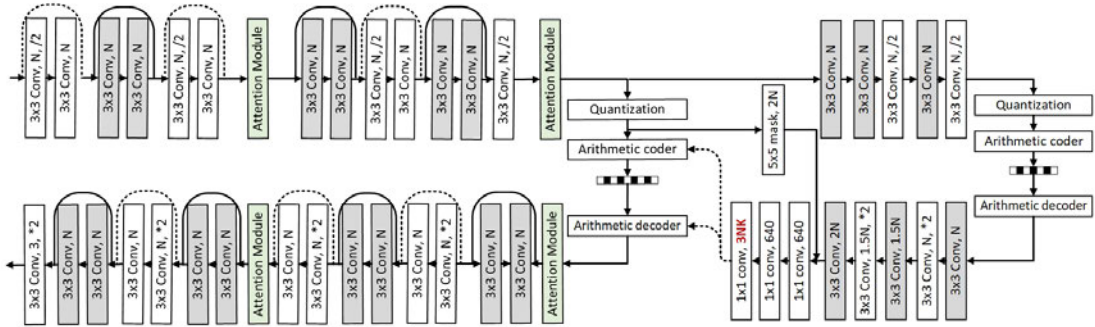


Figure 2.12: Network architecture for the residual joint autoregressive and hierarchical priors model.

Source: Cheng et al. [14, p. 5] (Copyright © 2020, IEEE)

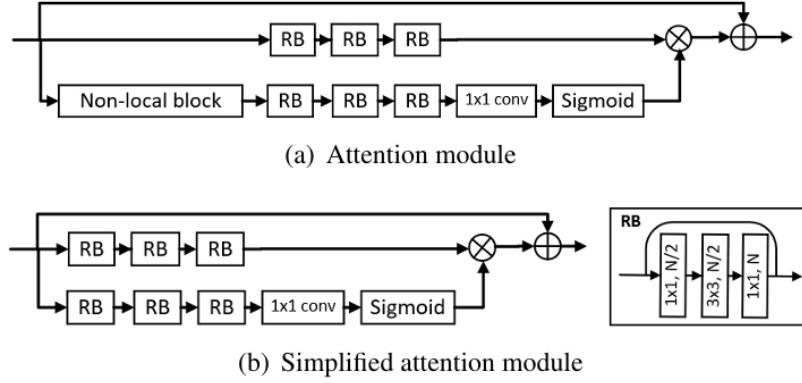


Figure 2.13: Different attention modules for the residual joint autoregressive and hierarchical priors model with attention modules.

Source: Cheng et al. [14, p. 6] (Copyright © 2020, IEEE)

## 2.3 Denoising

Denoising describes the process of cleaning partially corrupted inputs [64] and is mainly performed with the help of AE. A basic AE architecture for DAE can be seen in Fig. 2.14. Since denoising has been shown to improve the extraction of features, as mentioned in Section 2.1.5, no architecture is explicitly used for denoising. Vincent et al. [64] used in their introduction to denoising stacked under-complete AEs. Under-complete means the fully connected layers between the input and output layers have fewer neurons than the input. This means that they used AEs with more than one hidden layer and the number of neurons in the hidden layers would decrease. The latent space would then have the lowest number of neurons and, for the decoder, the number would increase again. Dobilas [15] on the other hand, recommends an over-complete AE, meaning that the hidden layers would have more neurons than the input and output layers. These two approaches rely on fully connected layers for denoising, but some architectures employ convolutional layers for denoising [3]. Some architectures employ noise learning and de-noise images by subtracting the regenerated noise from the noisy input, such as nlDAE [34]. This approach proves more effective than DAE when the noise is simpler to regenerate than the original data. The concept can be seen in Fig. 2.15.

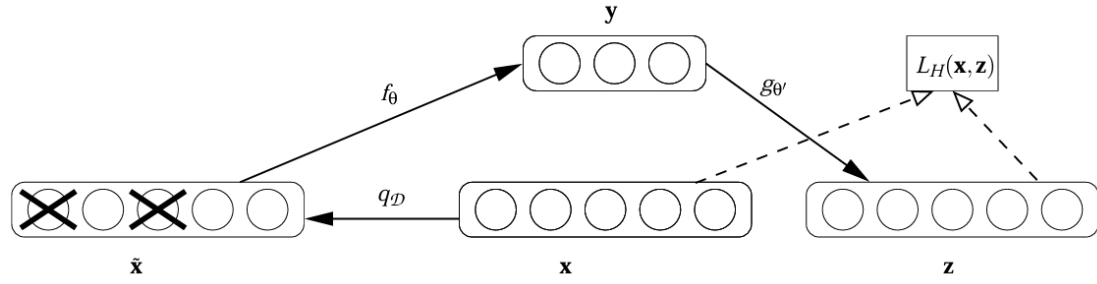


Figure 2.14: The denoising autoencoder architecture. An example  $\mathbf{x}$  is stochastically corrupted (via  $q_{\mathcal{D}}$ ) to  $\tilde{\mathbf{x}}$ . The AE then maps it to  $\mathbf{y}$  (via encoder  $f_{\theta}$ ) and attempts to reconstruct  $\mathbf{x}$  via decoder  $g_{\theta'}$ , producing reconstruction  $\mathbf{z}$ . Reconstruction error is measured by loss  $L_H(\mathbf{x}, \mathbf{z})$ .

Source: Vincent et al. [64, p. 9] (Copyright © 2010, ACM, Inc.)

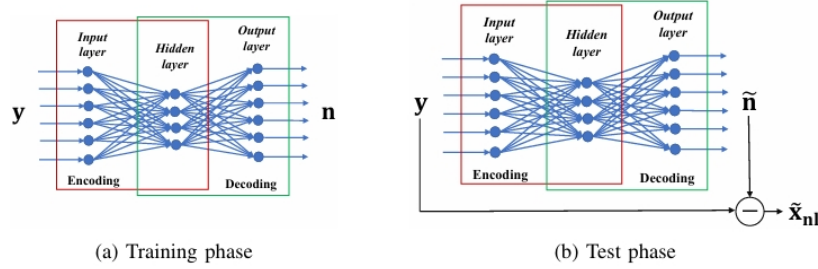


Figure 2.15: An illustration of the concept of nlDAE.

Source: Lee et al. [34, p. 2] (arXiv preprint arXiv:2101.07937, 2021).

## 2.4 Clustering

The ability of AE to extract features and create a compressed representation of inputs makes the latent space of AEs an interesting input for clustering or classification tasks. Because of this, AEs are used as part of AI pipelines or for pre-classification processes. The AE tutorial for the PyTorch lightning front end [50] uses the latent space to find the  $K$  closest images and for clustering with the training logging tool 'tensorboard.' Pascal [49] shows how the latent space can be used as input to Principal Component Analysis (PCA) and as part of a classification pipeline. Fig. 2.16 shows the result of the 2D visualization of the latent space of AEs trained on the MNIST dataset.

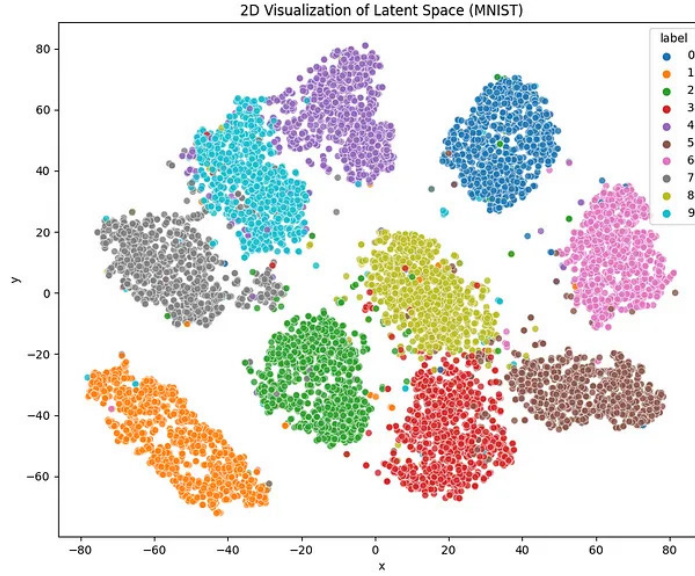


Figure 2.16: 2D visualization of the latent space using t-SNE.

Source: Pascal [49] (Copyright © 2023, Medium).

And while all approaches of using the latent space of AEs as input for clustering or classification tasks, the  $k$ -sparse AE and LSRAE suggest that AE itself has the potential to be used as a clustering algorithm.

## 2.5 Au-Cluster Physics with GISAXS

Since running an Au-Cluster Physics experiment, taking the scattering images with the help of GISAXS and evaluating the results, is complex. Fig. 2.17 gives an overview of the experiment and the creation of the GISAXS scattering images.

At the start of the experiment, gold atoms are deposited on a silicon substrate through sputter deposition creating various surfaces and growth processes on the silicon substrate (sample). By irradiating the sample surface with a grazing X-ray beam at an angle of incidence  $\alpha_i$ , scattering occurs on the sample surface. A detector then detects the scattered photons of the X-ray beam and images are deposited in reciprocal space. This allows the position of a photon to be expressed by the wave vector  $\mathbf{q}$  in the GISAXS

scattering pattern.  $\mathbf{q}$  in turn results from the vertical angle of reflection  $\alpha_f$  and the horizontal angle of  $2\theta_f$ . In addition, the positions of other characteristic features can be determined in the GISAXS scattering pattern: The direct beam transmitted through the sample at  $\alpha_f = -\alpha_i$  and the reflected specular beam at  $\alpha_f = \alpha_i$ . Direct and specular beams are shielded by a lead beam stop to protect the detector recording the GISAXS scattering pattern from oversaturation. The Yoneda peak is located at the critical angle of the sample material  $\alpha_f = \alpha_c$ . It is merely a material-specific intensity maximum [72].

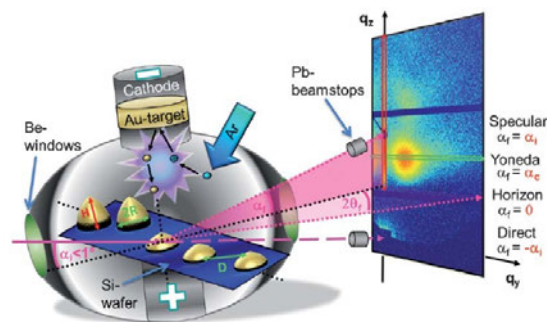


Figure 2.17: Overview of an Au-Cluster-Physics experiment including generation of GISAXS scattering image.

Source: Schwartzkopf et al. [56] (Copyright © 2013, Royal Society of Chemistry).

### 2.5.1 Sputter Deposition of Gold on Silicon

Sputter deposition is one of the most promising coating processes in the industry for controlled metallization of surfaces with custom layer properties [42]. The process works as follows [55, p. 24]: Argon (Ar) flows into a vacuum sputter chamber under constant working gas pressure. A glow discharge of the gas in the sputter chamber is caused by applying an electrical potential. Plasma forms as a result, consisting of argon ions and free electrons. When these particles collide, they produce positively charged argon ions, known as argon cations. They are accelerated towards the cathode by an external electromagnetic field. There, they collide with a solid surface (target), in the case of Au-Cluster physics, gold. As a result, gold atoms are knocked out of the solid surface by the high kinetic energy of the accelerated argon cations and transferred into the gaseous phase. This process is called sputtering. The sputtered, gaseous gold atoms move toward the silicon substrate and can settle as thin layers, also called *deposition*.



The gold atoms on the silicon substrate still have a high surface mobility and an average kinetic energy of 1 to 2eV with up to 100eV at the maximum. This influences layer growth and, by extension, the formation of a certain microstructure [42]. The kinetic energy and therefore the surface mobility can be influenced by the deposition rate and the working gas pressure, which are among the important process parameters [55, p. 26]. More information can be found in Ref. [16, 42, 57]

### 2.5.2 Surface Processes

The gold atoms deposited on the substrate are subject to various surface processes that compete with each other. The result is a complex non-linear self-organization behavior [17]. These surface processes have a significant influence on the growth kinetics. Rate equations with a weighted Arrhenius term are used to describe surface processes [17]. These give a probability  $P$  for the occurrence of a surface process, allowing one to determine the dominating atomic movements on the substrate in the early phases of growth:

$$P \propto A \cdot \exp\left(-\frac{E_A}{k_B T}\right) \quad (2.23)$$

where  $A$  represents the weighting factor depending on the surface process.  $E_A$  describes the activation energy required to trigger the respective surface process.  $k_B$  is the Boltzmann constant, and  $T$  is the temperature.

Unless otherwise stated, the following observations of essential surface processes come from Ref. [17] and Ref. [55, pp. 28-30]:

As seen from Section 2.5.1, the fundamental surface process is deposition. It describes adsorption of an atom (AdAtom) on the substrate from the gas phase. Deposition occurs when the binding energy between the atom and the substrate exceeds the atom's kinetic energy. Otherwise, re-evaporation occurs, and the atom returns to the gas phase because of its kinetic energy. If the AdAtom remains on the substrate, *diffusion* occurs and it moves along the substrate. If it collides with another AdAtom, another surface process occurs, *nucleation*. Triggered by the collision, the AdAtoms form a chemical bond and form a dimer. Should the binding energy significantly exceed the thermal energy, we talk about stable dimers. These stable dimers allow AdAtoms to attach laterally to them, thus initiating the two-dimensional horizontal growth of the clusters. Otherwise, *dissociation* occurs, whereby the clusters fall apart again. With increasing cluster size, diffusion is less likely to occur, and *aggregation* dominates. This results in the capture

of diffused AdAtoms in the immediate vicinity. This in turn leads to the growth of clusters or layers. As the size of the clusters increases, so does the probability that an atom is directly adsorbed from the gas phase. The atom settles on the surface of the cluster and diffuses. This allows three-dimensional, vertical growth to be observed. Three-dimensional growth can also occur should AdAtoms at the cluster's edges jump into the next layer, i.e. diffuse vertically. For this to happen, AdAtoms at the edges of the clusters must overcome the energy barrier to reach the cluster's surface.

### 2.5.3 Growth process

The competing surface processes trigger growth processes, leading to layer growth. A distinction can be made between three growth models. The growth model that ultimately leads to layer growth depends on the ratio of surface processes occurring [55, p. 27]. In the first model, the *Frank-van-der-Merwe* model [20], growth occurs layer-by-layer, with the next layer only forming when the previous layer is closed. In the next growth model, the *Volmer-Weber* model [66], layer growth is expressed by forming clusters that do not form closed layers. Here, the layers form if the cluster size increases. The last model, the *Stranski-Krastanov* model [61], combines both. At first, a closed layer forms on the substrate, and the clusters form on it. Concerning Au-Cluster physics, the Volmer-Weber growth model is highly significant, as gold on silicon is subject to this growth model [56].

Fig. 2.18 shows the growth process of gold clusters on silicon according to a hemispherical model (local hexagonal arrangement of monodisperse hemispheres [56]) and a side view of the surface processes that occur. This can be divided into four phases [55]. The average distance between the edges of the gold cluster is used to determine the respective phase. This can be viewed as the ratio between the average diameter of the gold clusters  $2R$  and the average distance between the gold clusters  $D$ . According to Schwartzkopf [55], the phases can be summarized as follows:

In the nucleation phase (I), the mobile AdAtoms on the substrate are looking for other AdAtoms to minimize their energy. The collisions between AdAtoms result in a bond that forms a gold cluster. The average distance between the clusters is massive, while the average diameter of the gold clusters is quite small. Therefore, the ratio is minuscule ( $D \gg 2R$ ).

In the diffusion phase (II), smaller gold clusters already exist, which diffuse. They fuse through contact with other gold clusters to form larger gold clusters, directly adsorb gold atoms from the gas phase, or serve as aggregation centers. This increases the average diameter of the gold clusters while the average distance between the gold clusters remains so that  $D > 2R$  applies.

In the adsorption phase (III), smaller gold clusters merge to form large gold clusters. Adsorption directly from the gas phase increases, while diffusion decreases with increasing cluster size. With increasing coverage marking the end of this phase, the percolation threshold is reached. This describes the transition from separated clusters to a continuous layer. When the surface is approximately 90% covered with gold clusters,  $2R = D$  applies [55, p. 138]. The percolation threshold is the starting signal for the start of the last phase (IV).

In the last phase (IV), grain growth occurs with  $D < 2R$ . Due to the high coverage and the resulting restrictions on growth, a dominant grain, 1 in Fig. 2.18, grows by shifting its grain boundaries without regard to other grains, 2 and 3 in Fig. 2.18. At the same time, the dominant grain can be consumed by another more dominant grain. Furthermore, vertical growth occurs through the direct adsorption of gold atoms from the gas phase.

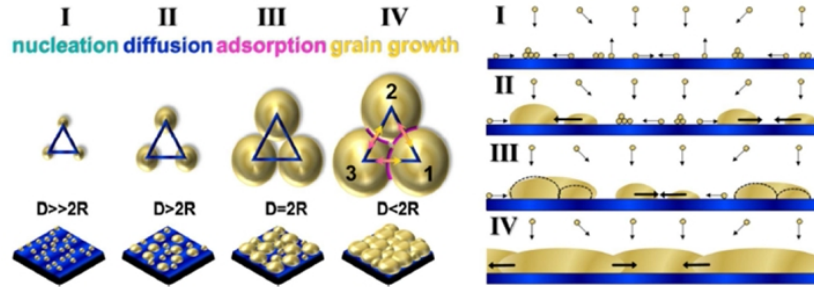


Figure 2.18: Four growth phases of gold clusters according to the hemispherical model with the side view of the occurring surface processes.

Source: Schwartzkopf et al. [56] (Copyright © 2013, Royal Society of Chemistry).

### 2.5.4 GISAXS

The method of Small Angle X-ray Scattering Under Grazing Incident Angles (GISAXS) examines the surface structures. GISAXS enables the measurement of the electron den-

sity distribution of a sample surface by irradiating it with X-rays. This enables better statistics on the average depth information of the morphological parameters of the surface structure to be obtained, allowing more precise statements about the morphology [47]. Due to its fine resolution in the nanometer range, it is particularly suitable for investigating nanostructures [47]. GISAXS can detect structures on and under the surface using short X-ray bursts. This can be done without changing the environmental conditions through interaction with the surface, causing artifacts, or even destroying the surface [52]. When the X-ray beam penetrates the surface, the depth information decreases sharply with increasing depth. As a high-energy X-ray source, a synchrotron is used, and in recent decades its intensity has increased exponentially. This allows nanostructures to be investigated in a time-resolved manner and *in situ* during sputter deposition to gain in-depth insight into surface processes and self-organization.

As mentioned in Section 2.5 and shown in Fig. 2.17, a wave vector  $\mathbf{q}$  can be used to represent X-ray scattering. It consists of the difference between the scattered wave vector  $\mathbf{k}_f$  and the incident wave vector  $\mathbf{k}_i$ :

$$\mathbf{q} = \mathbf{k}_f - \mathbf{k}_i. \quad (2.24)$$

To determine the wave vector, a corresponding geometry must be defined specifically for the GISAXS experiment. This can be seen in Fig. 2.19. For the GISAXS geometry, a three-dimensional coordination system is chosen, where the sample surface is in the  $x$ - $y$  plane and the  $z$ -axis is perpendicular to it [55, p. 87]. Additionally, the  $x$ -axis runs in the direction of the X-ray beam, allowing the incident wave vector  $k_i$  to be dependent only on the  $x$ - $z$  plane. Thus, the incident wave vector with the wavelength  $\lambda$  of the X-ray beam and the angle of incidence  $\alpha_i$  is the following:

$$\mathbf{k}_i = \frac{2\pi}{\lambda} \begin{pmatrix} \cos(\alpha_i) \\ 0 \\ -\sin(\alpha_i) \end{pmatrix} \quad (2.25)$$

To allow the X-ray beam to fall on the substrate in a grazing manner, thus causing as many scattering events as possible, the constant incidence angle is chosen to be very small for GISAXS experiments ( $\alpha_i < 1^\circ$ ). The incident X-ray beam is deflected due to the electron density fluctuations on the sample surface, resulting in scattering events. The reflected beam spreads out towards the detector as diffuse scattering. The diffuse

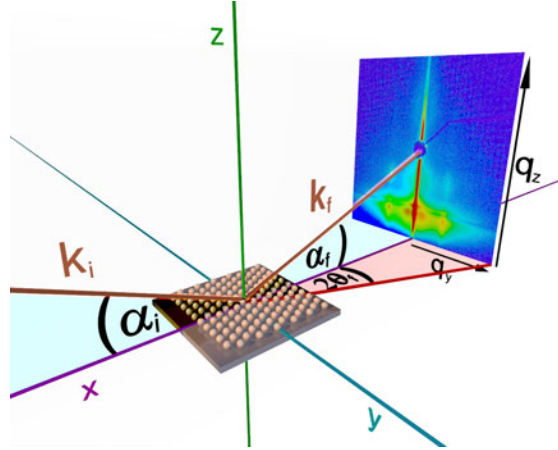


Figure 2.19: GISAXS geometry.

Source Meyer [45] (Copyright © 2018, University of Hamburg).

scattering is represented by the scattered wave vector  $\mathbf{k}_f$  and is detected by a two-dimensional detector position at a distance, called the detector-sample distance  $D_{SD}$ . The diffuse scattering is deflected by the reflection angles  $\alpha_f$  or  $2\theta_f$  vertically along the  $q_z$ -axis or horizontally along the  $q_y$ -axis of the detector. The scattered wave vector  $\mathbf{k}_f$  is thus:

$$\mathbf{k}_f = \frac{2\pi}{\lambda} \begin{pmatrix} \cos(2\theta_f) \cos(\alpha_f) \\ \sin(2\theta_f) \cos(\alpha_f) \\ \sin(\alpha_f) \end{pmatrix} \quad (2.26)$$

Finally, the difference between the scattered and incident wave vectors can be determined. But only the components  $q_y$  and  $q_z$  are important [55, p. 61]:

$$\mathbf{q} = \frac{2\pi}{\lambda} \begin{pmatrix} \cos(2\theta_f) \cdot \cos(\alpha_f) - \cos(\alpha_i) \\ \sin(2\theta_f) \cdot \cos(\alpha_f) \\ \sin(\alpha_f) + \sin(\alpha_i) \end{pmatrix} \quad (2.27)$$

The resulting wave vector maps the diffuse scattering in reciprocal space onto the detector. Because of this, each coordinate of the detector represents a projection of the wave vector. The sensor measures the intensity distributions of the scattered X-ray beam and uses them to define the GISAXS scattering pattern. From this, the following relationship can be derived [33]:

$$I(q_y, q_z) \propto |F(q_y, q_z)|^2 \cdot S(q_y, q_z) \quad (2.28)$$

Especially later for simulation, it is important to know that the scattering intensity consists of two parts, a structure factor  $S(q_y, q_z)$  and a form factor  $F(q_y, q_z)$ . The spatial arrangement of the particles on the surface is described by the structure factor and the geometry, i.e., the shape of an individual cluster by the form factor [56]. The form factor is also affected by the multiple reflections and scattering that occur in GISAXS near the critical angle of a material. This complicates the GISAXS scattering pattern [33]. As a result, diffuse scattering can only be inadequately described by the first-order Born approximation (BA). However, this captures only the scattering on the hemispherical particle and neglects the scattering on the substrate. To take into account higher-order scattering processes, BA was expanded to include the concept of Distorted Wave Born Approximation (DWBA) [65]. According to this, the form factor of the GISAXS scattering pattern must consider four scattering processes on the hemispherical particle to be able to describe the scattering pattern. The first describes the undisturbed scattering event on the hemispherical particle. The remaining three scattering processes describe a disturbed scattering event on the substrate before and after the scattering on the hemispherical particle. The last describes the events of both before and after the scattering. Additional information can be found in Ref. [55, 65].

When all scattering processes are accounted for using the form factor, the scattered intensity – and, consequently, the key characteristics of the GISAXS scattering pattern – can be described with minimal error. In the next section, more information can be found on analyzing these patterns developed in the GISAXS scattering pattern.

### 2.5.5 Analysing GISAXS scattering pattern

The analysis of the GISAXS scattering images is used in particular to determine the morphological parameters of the cluster. However, to determine these parameters from the experimental GISAXS scattering images, a closer look at the characteristic features of the GISAXS scattering image is required. Fig. 2.20 shows an experimental GISAXS scattering image. The intensity scale on the right side of Fig. 2.20 shows the logarithmic distribution of the intensities on the detector. For each pixel, the intensity depicts the number of photons detected by the sensor. In addition, characteristic features have

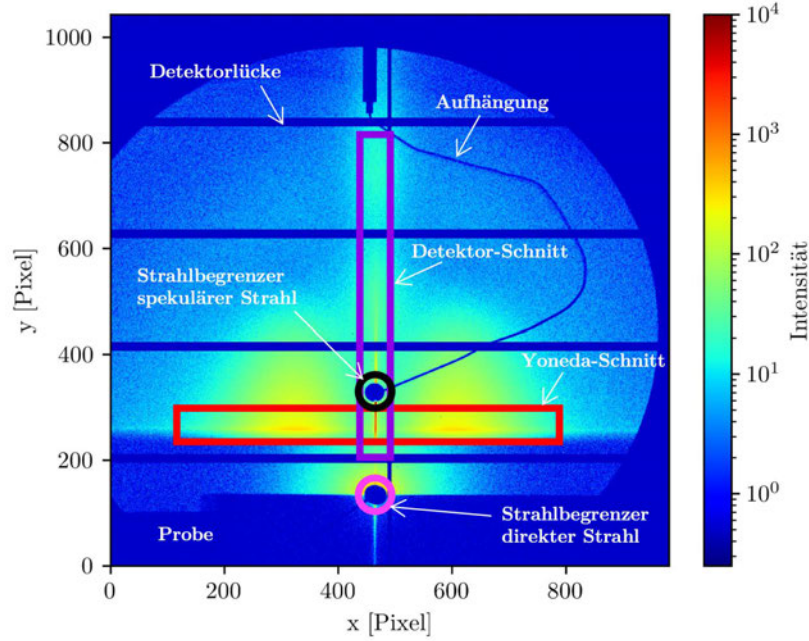


Figure 2.20: Experimental GISAXS scattering pattern for the material system gold on silicon with marked characteristic features. Based on [55, p. 86].

Source: Almamedov, Eldar [1, p. 17] (Copyright © 2022, Hamburg University of Applied Sciences).

been marked in color for a more detailed discussion. The following observations on the distinctive features of the GISAXS scattering image come from [55, pp. 86-89].

Some of these characteristic features are due to the experimental setup of the GISAXS experiment. They are important because of the AE's ability to reconstruct missing information if a clean version of the image exists. First, the GISAXS scattering image contains horizontal and vertical detector gaps (Detektorlücke), caused by the modular design of the detector. They may vary from manufacturer to manufacturer. The detector gaps do not contain any X-ray-sensitive pixels and thus do not count the number of photons that could be hitting these pixels. In addition, two point-shaped areas on the GISAXS scattering image also contain almost no photons caused by the beam stops (Strahlbegrenzer). They protect the detector from saturation and even damage. Their job is to shield the direct X-ray beam transmitted through the sample at  $\alpha_f = -\alpha_i$  (pink circle) and the specular X-ray beam reflected by the sample at  $\alpha_f = \alpha_i$  (black circle). Below the direct beam (pink circle), the sample (Probe) casts a shadow containing no photons.

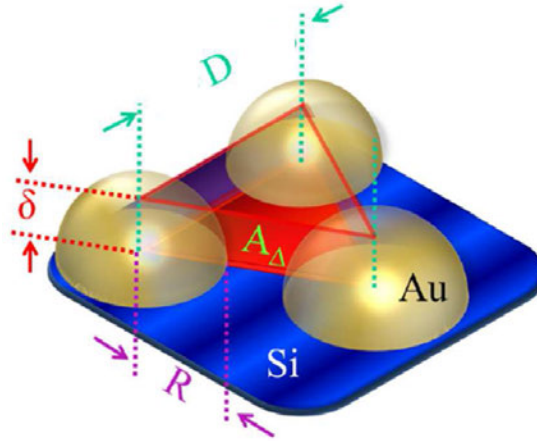


Figure 2.21: Morphological parameters according to the hemispheric model.

Source: Schwartzkopf et al. [56] (Copyright © 2013, Royal Society of Chemistry).

The two-dimensional GISAXS scattering image also contains a lot of embedded structural information, such as the average distance of the forming gold clusters  $D$ , the effective layer thickness  $\delta$ , the volume of the cluster  $V_c$ , and the average radius of the gold cluster  $R$ . Their relation can be seen in Fig. 2.21 and for more information on how to calculate them and where to find them, refer to Ref. [1, 55].

Only the mean distance  $D$  between the gold clusters and the mean radius  $R$  of the gold clusters are used for the classification of the GISAXS scattering images [1].

### 2.5.6 Simulation of GISAXS scattering patterns

Alongside the mean distance  $D$  between the gold clusters and the mean radius  $R$  of the gold clusters, their respective distributions also play an important role in the classification. These two distribution factors play an important role in the classification and the creation of GISAXS simulation images. The simulation of GISAXS scattering patterns is carried out using software such as *IsGISAXS* [33]. It is used to interpret experimental GISAXS scattering images and to verify their plausibility. Based on a model, *IsGISAXS* can calculate two-dimensional GISAXS scattering images, provided that the model contains all relevant information about the geometry and arrangement of the nanoparticles, i.e., the existing surface morphology [30]. Equation 2.28 calculates the scattering intensity formed by the product of the form factor and the structure factor. Fig. 2.22 shows



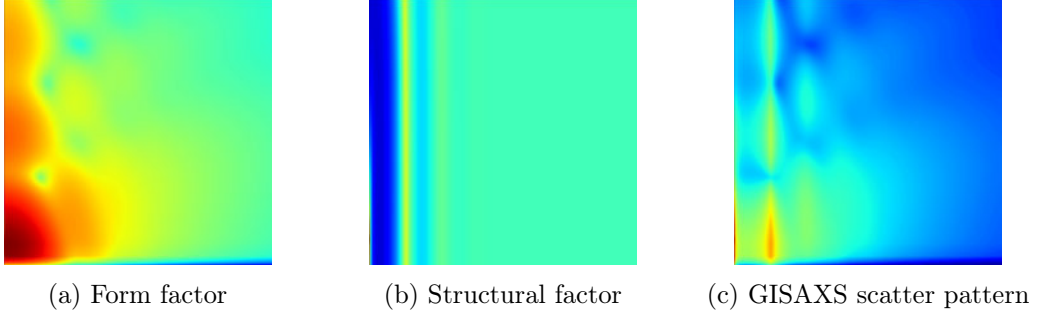


Figure 2.22: Simulation of form factor and structure factor in *IsGISAXS* to generate a GISAXS scatter pattern.

an example of the form factor, structure factor, and the resulting GISAXS scattering pattern of an *IsGISAXS* simulation. The form factor is determined using DWBA, which describes the resulting reflection and scattering on the sample surface. The grazing incidence of the X-ray beam is the cause of this. Since the hemispherical model represents the shape of the nanoparticles as hemispheres, we simulate the form factor in *IsGISAXS* for a given average nanoparticle radius  $R$  and a Gaussian distribution of the radius variation  $\sigma/R$ .  $\sigma/R$  represents the width of the probability density function of the Gaussian distribution [33]. Increasing this parameter results in a broadening of the form factor maxima, leading to a smearing of the simulated GISAXS scattering patterns [55, p. 94]. This makes the simulated and experimental GISAXS scattering patterns similar. The smearing of the *IsGISAXS* scattering pattern increases with an increase of  $\sigma/R$ . This can be seen in Fig. 2.23. Finally, the variation in the radius of a nanoparticle at position  $x$  on the surface can be described by the density function of the Gaussian distribution:

$$p_F(x) = \frac{1}{\sigma\sqrt{2\pi}} \cdot \exp\left(-\frac{(x-R)^2}{2\sigma^2}\right) \quad (2.29)$$

The same is true for the structure factor describing the spatial arrangement of the nanoparticles on the sample surface. The structure factor is simulated in *IsGISAXS* based on a reference nanoparticle, determining the distance to the next nanoparticle. For this purpose, the probability of the next nanoparticle at position  $x$  is described by the density function of the Gaussian distribution depending on a reference nanoparticle [33]:

$$p_S(x) = \frac{1}{\omega\sqrt{2\pi}} \cdot \exp\left(-\frac{(x-D)^2}{2\omega^2}\right) \quad (2.30)$$

Thus, the structure factor is simulated as a function of the average distance  $D$  and the Gaussian distributed variation of the distance  $\omega/D$ . The  $\omega/D$  describes the width of the Gaussian distribution. Increasing this parameter leads to a broadening of the side maxima [55, p. 95], see Fig. 2.24.

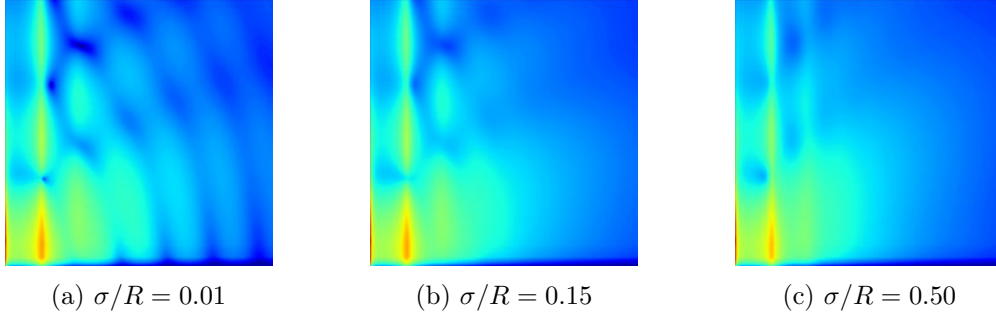


Figure 2.23: Influence of  $\sigma/R$  on *IsGISAXS* simulation for  $R = 4.0\text{nm}$ ,  $D = 10.0\text{nm}$ , and  $\omega/D = 0.15$

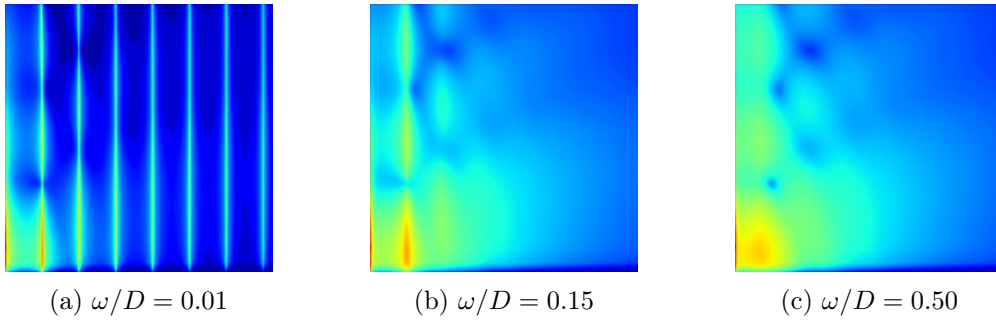


Figure 2.24: Influence of  $\omega/D$  on *IsGISAXS* simulation for  $R = 4.0\text{nm}$ ,  $\sigma/R = 0.15$ , and  $D = 10.0\text{nm}$

*IsGISAXS* is not only capable of defining the geometry and arrangement of the particles with the associated distribution functions (Eq. 2.28 - 2.30). It can also be used to define experimental parameters. This facilitates describing the reciprocal space forming when the X-ray beam interacts with the sample surface. More about that in Ref. [33].

### Simulation dataset

The DESY created for the work of Almamedov, Eldar [1] a dataset of GISAXS scattering patterns. The scattering patterns were simulated for the material gold on silicon using

*Is*GISAXS. The dataset contains 3550 entries for both the form and structure factor, and when combined they create a GISAXS scattering pattern (see Fig. 2.22). This allows us to create a dataset of GISAXS scattering patterns of the size of  $3550^2$  since each form factor can be combined with each structure factor. In addition to the experimental parameters, each entry contains unique simulation parameter combinations  $R$ ,  $\sigma/R$ ,  $D$ , and  $\omega/D$ . These parameters classify the simulated and experimental GISAXS scattering patterns. Table 2.5 presents the value ranges and step sizes for the simulation parameters related to the form and structure factor. Table 2.6 shows the experimental parameters.

	$R$	$\sigma/R$	$D$	$\omega/D$
Value range	0.5-7.5 nm	0.01-0.5	1-15 nm	0.01-0.5
Step size	0.1 nm	0.01	0.2 nm	0.01

Table 2.5: Simulation parameters for form and structure factors.

$\lambda$	$\alpha_i$	$\alpha_{f,\min}$	$\alpha_{f,\max}$	$2\theta_{f,\min}$	$2\theta_{f,\max}$	$n_1$	$n_2$
0.096 nm	0.4°	0°	4°	0°	4°	499	500

Table 2.6: Experimental simulation parameters for X-ray beam and detector.

## 2.6 AI in the Field of Scattering Images

Applying Machine Learning (ML) to scatter images is, as in many other fields employing ML, a relatively new field of research. However, it has become increasingly important in recent years. This occurred partly due to advances in Deep Learning (DL), which enabled the recognition of complex structures, and also due to improvements in efficiency and hardware. This allows for the real-time application of ML.

One of the first works was done in 2014 by Hadi Kiapour et al. [23]. They employed ML to detect and classify important attributes from scattering images generated with Wide Angle X-ray Scattering (WAXS) and Small Angle X-ray Scattering (SAXS). Previously, experts determined these attributes manually. In 2016, Wang et al. [68] built upon the work of Hadi Kiapour et al. [23]. They applied deep learning techniques to extract features from scattering images automatically. The study found that using a CNN for automatic feature extraction improved the performance of ML methods compared to manually extracted features by 10%. This work was the first to use simulated scattering images for training. The simulated images were augmented with features of experimental

scattering images such as Poisson shot noise, detector gaps, and shadows of the experimental setup. The trained model was tested on experimental and simulated scattering images. In 2017 Wang et al. [69] continued their work by trying Residual Neural Network (ResNet) [24].

2019 marks the first time DL has been employed in the field of GISAXS scattering images [38]. Here, a CNN-based classification was used to determine seven combinations of 3D nanoparticle lattice alignments from simulated and experimental scattering images. The simulated images were again augmented. After training, the model achieved 52 % precision in classifying 33 experimental scattering images.

Ikemoto et al. [29] classified GISAXS scattering images with a CNN in 2020. Here, the focus was on determining the shape of the nanoparticles in the scattering pattern.

Stielow et al. [60] used ResNet to perform a regression determining the particle size and orientation from WAXS scattering images. Again, simulated scattering images were augmented. The ResNet achieved a mean prediction error below 1% when applied to test data. The model was also able to accurately and quickly determine the shape of nanoparticles from experimental data.

In 2021 Herck et al. [25] used DenseNet [28] to determine the rotation distribution of hexagonal nanoparticle arrangements from GISAXS scattering images. The simulated GISAXS scattering images were extensively augmented: using intensity scaling, adding Poisson shot noise, masking with shadows from the experimental setup and the detector gaps, and normalizing. The work showed that DL is suitable for analyzing GISAXS scattering images. It also showed that using transfer learning is possible. Here, a model is trained with simulated data and then is tasked with applying the learned information to the experimental data.

In 2022 the DESY and the Hamburg University of Applied Sciences (HAW) started using DL to analyze the cluster growth of GISAXS scattering images [1]. As mentioned in Section 2.5.5, the clusters are described by their mean distance and mean radius. This proves tricky since Santoro et al. [54] manually determined that the radius distribution is closely related to the log-normal distribution, and creating simulation scattering images based on the log-normal distribution is time-sensitive; see Fig. 2.25. Their solution for this problem was to simulate scattering images using the Gaussian distribution and use a weighted sum of multiple simulation scattering images to approximate the log-normal

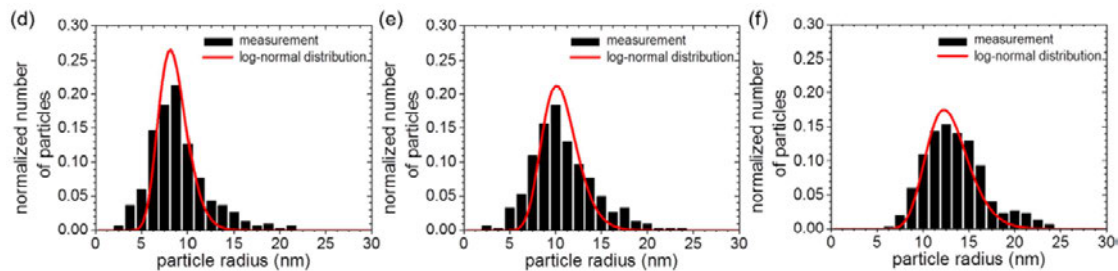


Figure 2.25: Examples for the radius distribution of Ag (Silver).

Source: Santoro et al. [54] (Copyright ©AIP Publishing LLC, 2014)

distribution. This work aims to help this approach by using AE to help with the pre-classification.

As for whether data augmentation or image healing is the better approach to close the sim-to-real gap, there is still an ongoing debate. The data augmentation process involves adding detector gaps, noise, and the shadow of the GISAXS setup to the simulation. Conversely, image healing focuses on removing these elements from the experimental data. Examples of image healing can be found in Ref. [37, 36], whereas Meister et al. [44] discourages it. They argue that image healing demands strong assumptions about the features in experimental scattering images and requires reference data from experiments. Furthermore, manual procedures are required for healing, which are error-prone and experience-dependent.

### 3 Proposed Approach

Our goal in this work is to test whether AEs can compress, heal, and form meaningful clusters with the help of AE’s latent space. In this case, healing means removing the influence of the experiment setup and noise. AEs (as discussed in Section 2.1) can effectively perform a wide range of functions, depending on the architectures and loss functions chosen. It is also possible to train an AE capable of combining multiple functions by stacking architectures together (stacked AE) or combining different loss functions (i.e., LSRAE). In this work, we will combine ConvAE, DAE, and SAE to compress, denoise, and cluster the simulated GISAXS scattering images. This is done by stacking the three different AEs on top of each other, resulting in a three-step training. In the first step, we will compress the images to save memory on both the hard drive and the system memory. In the next step, we will try to remove the detector gaps and the noise with which we augmented the simulated GISAXS scattering images. In the last step, we will use the compressed latent space after denoising as an input for our sparse model to compress the latent space further and cluster it.

Due to recent advances in DL, we combined multiple AE architectures to help assess the structure of the gold cluster in the experiments. Our model comprises three steps; the first compresses the GISAXS scattering pattern, the second uses the latent space of the first step to help denoising the image, and the last step again uses the compressed latent space after denoising to help cluster the patterns using its latent space. The compression and denoising steps will help reconstruct the detector gaps, and there might be a chance that the denoising step is directly integrated into the compression step depending on whether the added noise helps with reconstruction. The third step, clustering, on the other hand, can consist of two steps instead. The first step would use LSRAE to help with initial clustering and reduce the latent space size near the amount of cluster we are interested in investigating. The second step would generate clusters using a  $k$ -sparse AE architecture with  $k = 1$ . This allows us to train clusters in either a one-to-one or many-to-one relationship. The restriction of  $k$ -sparse AE is only active during training. When

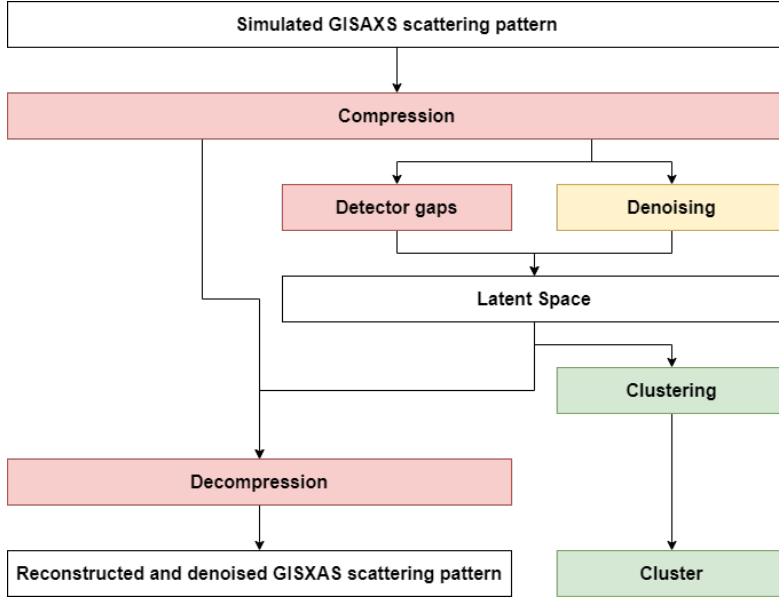


Figure 3.1: Overview of our proposed model. The red rectangles belong to the compression step, the yellow rectangles to the denoising step, and the green rectangles to the clustering. We get the reconstructed image and the cluster as the model’s output.

we evaluate real-world and not simulated experiments, we might require many-to-many relationships to evaluate the experiments because the evaluation is set up in a way where the return value is a probability vector where each entry refers to a specific simulated pattern. An overview of the proposed model is illustrated in Fig. 3.1.

In the following sections of this chapter, we describe the behavior of each step, its inputs, and its corresponding outputs, as well as its specific loss function. Section 3.1 will focus on compressing the simulated GISAXS scattering pattern, Section 3.2 on denoising, and Section 3.3 on clustering.

### 3.1 Compression

Since the amount of data generated during a GISAXS experiment and the high demands on RAM in the case of using a fully connected AE, it is important to at first compress the GISAXS scattering pattern. Because of this, we investigated AE architectures that employ convolutional layers to create a compressed representation of the input in the latent space. During this investigation, we found the PyTorch CompressAI tool [6, 7] and

decided to use some of the models to test their compression and reconstruction concerning the GISAXS scattering patterns. These models and information on CompressAI can be found in Section 2.2. The ability to compress the input is the reason why compression is the first step in our proposed model.

Regardless of which model we will use, modifications to the model are required. The models expect the images to be RGB images with a value range from 0 to 255 for each pixel and each channel. In the case of GISAXS scattering pattern, we have a distribution of detected X-ray beams as input, where the only information about the value range is that the lowest value for simulations is 0, and -2 for experiments. The range for the experiments can be changed to 0 since -2 indicates defect detectors or detectors that were for one or another reason unable to detect reflections, while the surrounding detectors were, and -1 indicates the detector gaps. Although the input value range, in this case, has no impact on the model, we are required to make changes to the first and last layers of the models to reflect the fact that the GISAXS scattering patterns only have one value per pixel compared to the RGB images with three.

With this, the pixel values of the GISAXS scattering pattern are represented as  $X(Channels \times Width \times Height)$  and given to several convolutional layers in the encoder case and transposed convolutional layers in the decoder case. While the transposed convolutional layer can be seen as the gradient of the convolutional layer concerning its input and as a true inverse of convolution, the output of a convolution layer can be computed as:

$$Y_{i,j,c_{out}} = \sum_{c_{in}=1}^{C_{in}} \sum_{m=1}^K \sum_{n=1}^K X_{i+m-1,j+n-1,c_{in}} \cdot W_{m,n,c_{in},c_{out}} + b_{c_{out}} \quad (3.1)$$

where  $Y_{i,j,c_{out}}$  is the value of the output feature map at position  $(i, j)$  for channel  $c_{out}$ ,  $X_{i+m-1,j+n-1,c_{in}}$  the input value at position  $(i + m, j + n)$  for channel  $c_{in}$ ,  $W_{m,n,c_{in},c_{out}}$  the weight at position  $(m, n)$  in the kernel for input channel  $c_{in}$  and output channel  $c_{out}$ , and  $b_{c_{out}}$  the bias term for the output channel  $c_{out}$ . The output dimensions for the convolutional and transposed convolutional layers can be seen in Eq. 3.2 and Eq. 3.3 respectively. Here  $H$  is the height and  $W$  the width of the input and output features, respectively,  $K$  the kernel size,  $S$  the stride, and  $P$  the padding. The equations are



calculated using the floor function, which is rounded to the nearest integer.

$$\begin{aligned} H_{\text{out}} &= \left\lfloor \frac{H_{\text{in}} - K + 2P}{S} \right\rfloor + 1 \\ W_{\text{out}} &= \left\lfloor \frac{W_{\text{in}} - K + 2P}{S} \right\rfloor + 1 \end{aligned} \quad (3.2)$$

$$\begin{aligned} H_{\text{out}} &= (H_{\text{in}} - 1) \times S - 2P + K \\ W_{\text{out}} &= (W_{\text{in}} - 1) \times S - 2P + K \end{aligned} \quad (3.3)$$

Different element-wise activation functions might be applied to the output of neurons, the analysis transform, also  $g_a$  or encoder, uses GDN, the synthesis transform, also  $g_s$  or decoder, uses the inverse of the GDN, while for the remaining layer with an activation function, Leaky ReLU is used. The equations for these activation functions can be seen in Eq. 3.4, 3.5, and 3.6, where  $\alpha$ ,  $\beta$ , and  $\gamma$  are parameters.

$$y_i = \frac{x_i}{\sqrt{\beta_i + \sum_j (\gamma_{ji} * x_j^2)}} \quad (3.4)$$

$$y_i = x_i * \sqrt{\beta_i + \sum_j (\gamma_{ji} * x_j^2)} \quad (3.5)$$

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{if } x \leq 0 \end{cases} \quad (3.6)$$

CompressAI trains its models with two optimizers; one is responsible only for the entropy bottlenecks, used to quantize and transform the latent space to a string and back; the other is responsible for the rest of the model and is used for reconstruction and compression. The first optimization based on the auxiliary loss does not depend on the image data. It is responsible for determining the range within which most of the mass of a given distribution is contained, as well as its median (i.e., 50% probability). The 'net' optimizer uses the main loss function for which two versions exist. One uses MSE to calculate the distortion and the other uses MS-SSIM. Table 2.3 shows how the loss is calculated for both versions, while Eq. 2.6 shows the loss function for the MSE loss and 3.8 for the MS-SSIM loss. The MS-SSIM loss function combines the luminance comparison function  $l(\mathbf{x}, \mathbf{y})$ , the contrast comparison function  $c(\mathbf{x}, \mathbf{y})$ , and the structure comparison function  $s(\mathbf{x}, \mathbf{y})$  shown in Eq. 3.7, where  $\alpha > 0$ ,  $\beta > 0$ ,  $\gamma > 0$  denote the relative importance of

each of the metrics.

$$\begin{aligned} l(\mathbf{x}, \mathbf{y}) &= \frac{2\mu_x\mu_y + C_1}{\mu_x^2 + \mu_y^2 + C_1}, \\ c(\mathbf{x}, \mathbf{y}) &= \frac{2\sigma_x\sigma_y + C_2}{\sigma_x^2 + \sigma_y^2 + C_2}, \\ s(\mathbf{x}, \mathbf{y}) &= \frac{\sigma_{xy} + C_3}{\sigma_x\sigma_y + C_3}, \end{aligned} \tag{3.7}$$

$$\text{MS-SSIM}(\mathbf{x}, \mathbf{y}) = [l_M(\mathbf{x}, \mathbf{y})]^{\alpha_M} \cdot \prod_{j=1}^M [c_j(\mathbf{x}, \mathbf{y})]^{\beta_j} [s_j(\mathbf{x}, \mathbf{y})]^{\gamma_j} \tag{3.8}$$

The success of the compression is calculated with the Bpp loss, which can be expressed as:

$$\text{BPP\_Loss} = \sum_{l \in L} \left( \frac{\sum \log(l)}{-\log(2) \times \text{num\_pixels}} \right) \tag{3.9}$$

where num\_pixels is the total number of pixels and  $L$  is the set of all likelihood values. The likelihood is a string containing the compressed image.

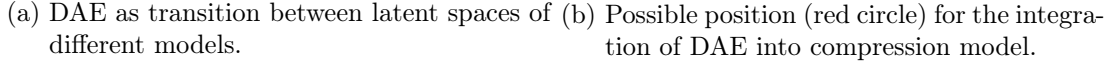
Adam [31] is the optimizer for both the 'net' and 'aux' training, but only the 'net' optimizer uses a scheduler for the learning rate. ReduceLROnPlateau is the scheduler, and it reduces the learning rate when a metric stops improving.

With this plan in mind and the research done in Chapter 2, we are now able to concertize the first of our research questions:

**"Which model, introduced in Section 2.2 and implemented with CompressAI, is the most effective at compressing and reconstructing GISAXS scattering patterns?"**

## 3.2 Denoising

Since denoising images is a potentially helpful way to increase the reconstruction by the AE and the goal of our proposed architecture, we propose two possible ways of integrating the denoising step into our architecture. The first proposed integration of DAE into our architecture is by using DAE as a transition between two different AEs, as seen in Fig. 3.2a. Here we take an AE trained with noised simulated GISAXS scattering patterns containing detector gaps and an AE trained with clean GISAXS scattering patterns. We



one-dimensional array is converted back to the three-dimensional array  $Y$  after the last fully connected layer. The output of the fully connected layers can be computed as:

$$\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b} \quad (3.10)$$

where  $\mathbf{x} \in \mathbb{R}^n$  is the input vector,  $\mathbf{W} \in \mathbb{R}^{m \times n}$  a weight matrix,  $\mathbf{y} \in \mathbb{R}^m$  the output vector, and  $\mathbf{b} \in \mathbb{R}^m$  the bias vector. An activation function, such as Leaky ReLU, can be added after each fully connected layer. MSE functions as the loss function for the denoising step in the case of the transition approach and where the denoising step is integrated into the compression step, the loss function will be the same as for the compression step.

The optimizer is again Adam [31] and the learning rate scheduler is ReduceLROnPlateau.

### 3.3 Clustering

Clustering is the last step in our proposed model and is responsible for the third and final output, while the compression step is responsible for the compressed representation and reconstruction. The step is marked green in Fig. 3.1 and we propose splitting it into two parts. The first part is responsible for reducing the latent space to a size of a power of two close to the number of possible clusters. The second part would then use  $k$ -sparse with  $k = 1$  to create the cluster. We propose the split because it is recommended to start with  $k$  set to the number of neurons in the latent space and decrease it during the first part of the training for a higher reconstruction. By dividing the clustering process into two stages, we can train the latent space reduction and the clustering separately, ensuring that they do not interfere. This also allows us to use two different loss functions to optimize the models and removes the possibility that the model would have two distinct layers with two separate goals running parallel at the same depth in the model.

For the first step of the clustering, we propose using a LSRAE model to create a sparse latent space of a size of a power of two near the number of possible clusters. We would additionally use a classification layer using the latent space as input to classify the images. This enables the creation of a latent space that is both sparse and slightly clustered, serving as the input for our proposed second step. Fig. 3.3a shows the framework of a LSRAE model and Fig. 3.3b shows our proposed approach. Here, red marks the latent space of the compression model, light green the LSRAE reconstruction part, yellow the

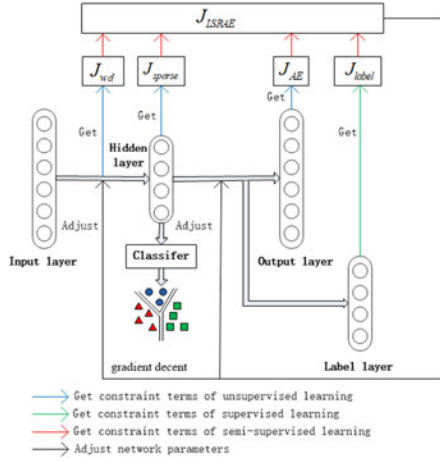
LSRAE classification part, and dark green the clustering with  $k$ -sparse. The visualization of the model layers is based on the assumption that the latent space has the shape:  $64 \times 4 \times 4$ .

Fully connected layers are used throughout the clustering step, as seen in Fig. 3.3b. The input and output of our DAE are represented as  $X(Channels \times Width \times Height)$ , where  $X$  depends on the latent space after the hyper encoder in the compression model. ReLU is the encoder activation function. The equation used by ReLU is shown in Fig. 2.4. The complete loss function for the LSRAE is shown in Eq. 2.22 and the  $k$ -sparse part uses MSE as the loss function. The implementation of the  $k$ -sparse layer can be found in A.1.1. Cross-entropy is the loss function for the classification and is calculated as:

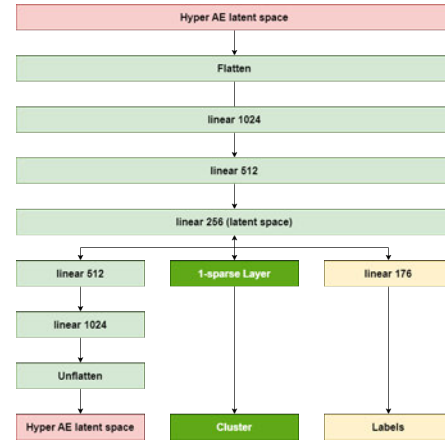
$$L = - \sum_{i=1}^N y_i \log(p_i) \quad (3.11)$$

where  $N$  is the number of classes,  $y_i$  is the true label, and  $p_i$  is the predicted label.

For the LSRAE, we use a variation of the Adam optimizer [31], called AdamW, which includes the weight decay that is part of our loss function. By using AdamW, we can move the weight decay from the loss function to the optimizer. For the  $k$ -sparse AE, we again use Adam, and the learning rate scheduler for both is ReduceLROnPlateau.



(a) The framework of LSRAE for classification.  
Source: Chai et al. [11, p. 207] (Copyright © 2019, Elsevier B.V.)



(b) Overview of the model for clustering.

Figure 3.3: Proposed approach for clustering step.

## 4 Experiments

This chapter focuses on the experiments we performed to check whether our proposed approach works or not. In Section 4.1, we explain the dataset we use and how the images are annotated. In Section 4.2, we focus on the experiments divided into two steps.

### 4.1 Dataset

Since there are no clean versions of experimental GISAXS scattering patterns, and thus evaluating the success of the denoising and removal of setup interference, such as the detector gaps and the experimental surface, can not be performed automatically, we have decided to focus on only using simulated GISAXS scattering patterns. To acquire these, we utilize the simulation dataset from DESY, as detailed in Section 2.5.6. Information about simulation parameters and experimental setup can be found in Tables 2.5 and 2.6. This would allow us, at least in theory, to create a dataset with  $3550^2$  simulated GISAXS scattering patterns. In reality, only a small part of these scattering patterns are usable since we can simulate scattering patterns with a higher average diameter than the average distance between clusters. But these scattering patterns cannot exist in the real world. Even after excluding these scattering patterns, the size of the dataset would still be too large for a reasonable training time.

Furthermore, verifying the clusters generated from such a large dataset would be excessively time-consuming and complex. As such, we came up with conditions that the dataset should be able to fulfill that would allow us to train models faster and make it easier to evaluate the cluster. These conditions are as follows:

- Contains both clean and masked versions of the simulated.
- Contain all possible radii  $R$  for a given distance since we intended to focus on the radius for the clustering segment.

## 4 Experiments

- Distances  $D$  should be common.
- $\omega/D$  should be chosen to act as a boundary for all possible distances.
- A test dataset should be created from the dataset so that it is not part of the training set but is somehow related to it.
- The size of the scattering patterns should be  $256 \times 256$ .
- $2R < D$ .

The resulting dataset can be seen in Table 4.1 and examples of the simulated GISAXS scattering patterns in Fig. 4.1.

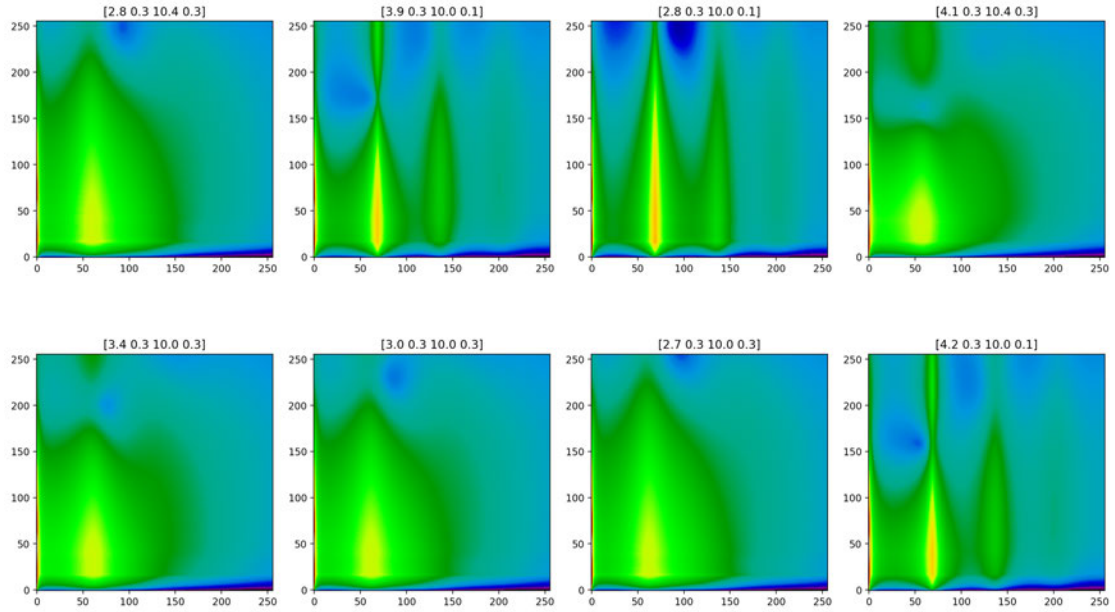


Figure 4.1: 8 Examples for simulated GISAXS scattering patterns. The title for each scattering patterns contains  $R$ ,  $\sigma/R$ ,  $D$ ,  $\omega/D$ .

$\sigma/R$	0.3					
$\omega/D$	0.1			0.3		
$D$	10.0nm	10.2nm	10.4nm	10.0nm	10.2nm	10.4nm
Range of $R$	2.5nn - 5.0nn	2.6nn - 5.1nn	2.6nn - 5.2nn	2.5nn - 5.0nn	2.6nn - 5.0nn	2.6nn - 5.2nn
scattering patterns	26	26	27	26	26	27
dataset type	train	test	train	train	test	train

Table 4.1: Contains information about the simulation parameters and the different radii for each subset.

The dataset was divided into 106 training images and 52 test images. We choose the simulated datasets with  $D = 10.2$  for the test dataset. We did this to utilize scattering patterns with smaller and larger distances for training while using patterns at intermediate distances to evaluate the reconstruction performance of our compression and denoising methods. Since we cannot automatically test experimental GISAXS scattering patterns, we selected simulated scattering patterns similar to those we used in training but not the same. Using the distance in the middle of our options ensures that images with a smaller and larger distance are known, instead of only images with a smaller or larger distance.

A second, larger dataset exists that contains all simulated GISAXS scattering patterns. Here, the distance starts with the lowest value and increases to the highest, and for every increase by 1 nm, two distances are chosen. These distances end with .0 and .6 since an increase of 0.5 nm (see Table 2.5) is impossible. We do not know whether this dataset will be used during this work, since the number of simulated GISAXS scattering patterns is around 60000. Training a model with this dataset requires at least a week unless a GPU cluster is used to accelerate the process.

### 4.1.1 Data Augmentation

We augmented the GISAXS scattering patterns by adding random noise and detector gaps to test our model.

#### Detector Gaps

During the preparation of the dataset by the research team at the DESY, we created a second version of every GISAXS image in our dataset. This helps us save time and makes choosing whether the input or output should include the detector gaps easier. Fig. 4.2 shows simulated GISAXS scattering patterns with detector gaps.



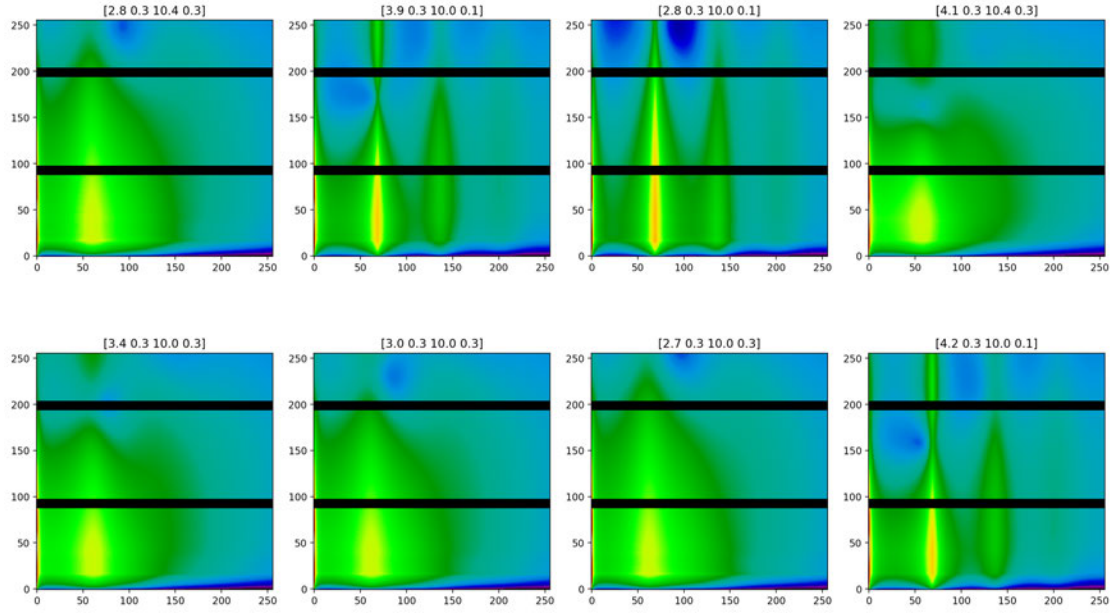


Figure 4.2: 8 Examples for simulated GISAXS scattering patterns including the detector gaps. The title for each scattering patterns contains  $R$ ,  $\sigma/R$ ,  $D$ ,  $\omega/D$ .

### Noise

We decided to create noise for our experiment using three different types of noise: Additive White Gaussian noise (AWGN), Bernoulli-Masked noise (BMN), and Poisson noise.

AWGN is a fundamental noise model in information theory used to simulate the effects of numerous random processes in natural systems. The noise image is generated with equation 4.1 and implemented as described in A.1.4. Compute the sum of the image  $X$  and the noise  $Z$ , where  $Z$  is independent and identically distributed, drawn from a normal distribution with a mean of zero and a variance of  $N$ . The resulting noise image is called  $Y$ . In this experiment, we used 0.01 as the variance. Examples can be found in Fig. 4.3a.

$$\begin{aligned} Z &\sim \mathcal{N}(0, N) \\ Y &= X + Z \end{aligned} \tag{4.1}$$

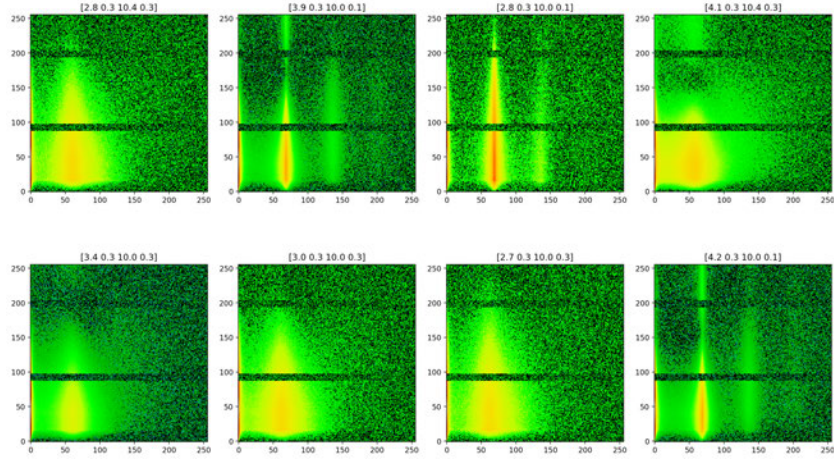
BMN employs the Bernoulli distribution to randomly multiply each pixel in the image by either 0 or 1. The matrix  $A$  contains the probabilities to draw the binary random numbers. The equation can be seen in 4.2 and the implementation in A.1.5. We used 0.7 as the probability for each pixel. Examples can be found in Fig. 4.3b.

$$Y = X \times \text{Bernoulli}(A) \quad (4.2)$$

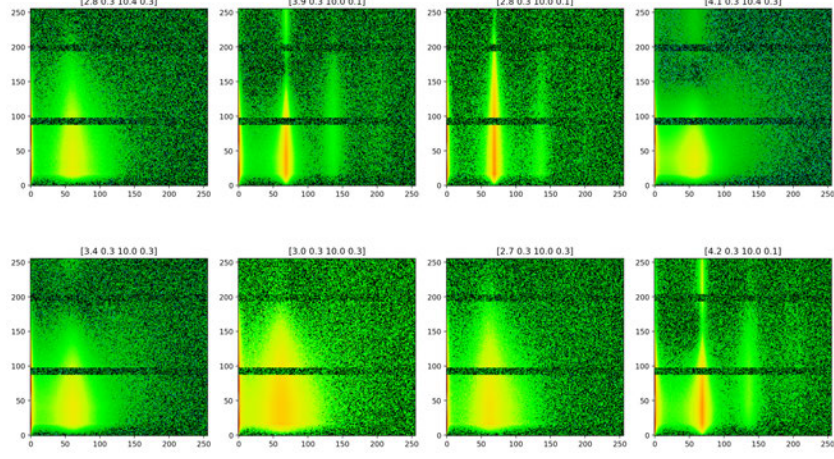
Poisson noise adds a normalized Poisson distribution based on the rates in matrix  $A$  to the image. Equation 4.3 shows the equation for the Poisson noise and A.1.6 the implementation. The rates are the same for every pixel and set to 1. Examples can be found in Fig. 4.3c.

$$\begin{aligned} P &= \text{Poisson}(A) \\ Y &= X + \|P\| \end{aligned} \quad (4.3)$$

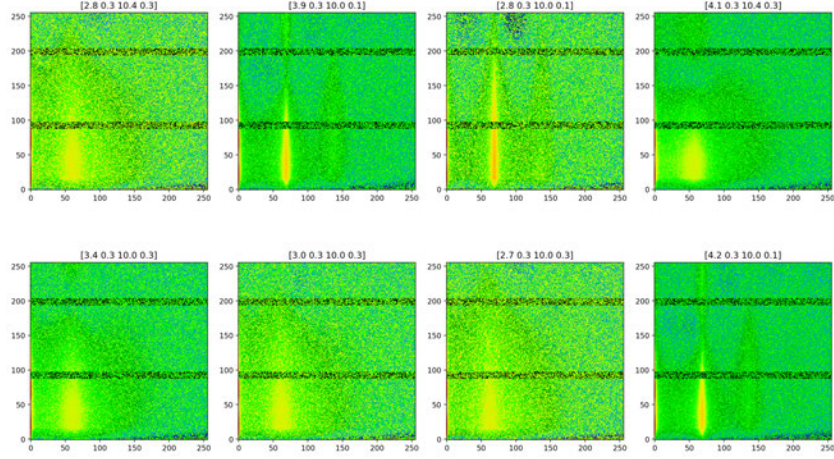
## 4 Experiments



(a) Additive White Gaussian noise (AWGN)



(b) Bernoulli-Masked noise (BMN)



(c) Poisson noise

Figure 4.3: 8 Examples for simulated GISAXS scattering patterns including the detector gaps and noise. The title for each scattering patterns contains  $R$ ,  $\sigma/R$ ,  $D$ ,  $\omega/D$ .

## 4.2 Training Procedures and Results

In this section, we provide details of the training procedures and the results of the training procedures. We train different AE network structures (a variety of compression, DAE, and sparsity models) for separate tasks and combine various approaches in Chapter 3 to find the best combination for the different assignments (compression, denoising, and clustering).

### 4.2.1 Data Preparation

Before evaluating whether our approach works, we need to prepare the datasets to run our different AE models. For all our models, we will rely on the combination of PyTorch’s `DATASET()` and `DATALOADER()` classes, NumPy’s binary file (`npz`), including memory-mapping the file, and PyTorch Lightning’s `DATAMODULE()` to load and process the dataset. Our implementation of this can be seen in A.1.7 and A.1.8. Memory mapping becomes extremely important in the case of large files, as it opens the way for memory-mapped arrays. These are stored on disk and can be accessed and sliced like any other NumPy `ndarray`. This allows us to access small fragments of the large file without reading the entire file into memory. PyTorch’s `DATASET()` class provides an interface for implementing custom dataset classes, assisting with data loading, determining dataset size, and extracting and transforming individual entries. The `DATALOADER()` helps us load the dataset and manages the batch sizes and the number of processor units responsible for loading the data. PyTorch Lightning’s `DATAMODULE()` handles data loader management and enables the LightningAI framework [18, 19] to log the dataset alongside the training process.

### Compression

To let our different compression networks process the specific input and target images, the PyTorch `DATASET()` allows us to simultaneously access the clean and masked version of the simulated GISAXS scattering patterns depending on the one we require. Additionally, it handles adding the different noise types from Section 4.1.1 while keeping the clean version. The labels for the GISAXS scattering patterns are not required for the

compression and are not handled by the DATASET() class. The CompressAI team recommends that the batch size for the training step should be 16 and 64 for the validation and testing step.

### Denoising

Depending on the approach we chose for the denoising process, we either use the same data preparation as the compression process or create a new dataset containing the latent space of the datasets used during the compression process. This new dataset was designed to save time during training and testing by eliminating the need to pass GISAXS scattering patterns through the compression model encoder to obtain the latent space. The rest remains the same.

### Clustering

Since the clustering process contains two steps, we prepare the data for both steps differently. For the LSRAE model, we save a new dataset containing the  $\hat{z}$  latent space after denoising. We create the new dataset to save time by not letting the input run through the compression and denoising model to construct the  $\hat{z}$  latent space each time. We employ PyTorch's ONE\_HOT() function to get a one-hot encoded label vector from the indices of inputs used for training. We use the index because it is easier to create the vector from them than from the labels, and every class is only present once in the dataset. We need the one-hot encoded vector for the cross entropy loss in the LSRAE model.

For the  $k$ -sparse model, we extract the latent space from the LSRAE model and use it as input. Additionally, we have access to the original labels of the input, in case we want to use them to check which classes are clustered together.

For the clustering, we no longer separate the dataset in training and test sets, since we need all dataset entries and are not interested in handling unknown data. Even if we later use experimental GISAXS scattering patterns and no longer simulated ones, the experimental ones can still be represented using multiple simulated ones.

### 4.2.2 Training Procedures

Since our evaluation consists of many experiments and some training results are needed to explain the decisions throughout the experiments, we will explain the different experiments we have done before evaluating them. During the process, we relied on PyTorch Lightning as a front-end for the training with PyTorch. The advantages of PyTorch Lightning are not only in the form of logging and an easier structure for the training loop, but also functions as version control. Furthermore, it allows us to save the hyperparameters used during training. PyTorch Lightning restructures the typical PyTorch training loop by extracting the training, validation, test steps, optimizer configuration, and other potential steps. This makes it easier to distinguish between the different parts of the training. PyTorch Lightning can automatically run the optimization process without requiring explicit programming when using a single optimizer. Most experiment results will include the version of the model as a reference. The naming scheme for the different compression models follows the one used by CompressAI. If not explicitly stated, the model can be found using the model name, the version number, and the task it is part of.

### Compression

Our first experiment compares the training results from the PyTorch Lightning front-end against those obtained with the CompressAI training loop. This is needed since the handling of multiple optimizers and a learning rate scheduler are required to be implemented in PyTorch Lightning and are no longer handled automatically. We needed a reference to verify that our implementation of the training loop is correct. The implementation of the PyTorch Lightning training loop can be seen in A.1.9 and A.1.10. In addition, we used the first experiment to test how many convolutional filters we need for good results. We used the FP model for this experiment.

In the second experiment, we tested the different quality settings (see Table 2.2 for the settings) for the MS-SSIM loss.

After choosing the  $\lambda$  value of the best-performing quality setting, we tested the different compression models: FP, Scale Hyperprior Compression AE (SH), MSH, Joint Autoregressive and Hierarchical Priors Compression AE (JAHP), Residual Joint Autoregressive and Hierarchical Priors Compression AE with Anchor Blocks (AN), and AT.

The next experiments first focus on reconstructing the detector gaps and then denoising the simulated GISAXS scattering patterns. We decided to check the reconstruction of the compression model in case of noise and also check which noise types we can denoise.

All experiments except for the investigation of denoising and quality setting were performed with the MSE and MS-SSIM loss.

### Denoising

For the denoising task, we tested the different positions in which DAE can be integrated into the compression model. These positions can be found in Fig. 3.2b. We then did tests by denoising the different input-output combinations to check whether the transitional approach between the two models should be tested. Depending on the results of this test, we performed the experiment using the transitional approach.

Additional tests were conducted to check how the over-complete, under-complete, and complete architectures AE would perform for denoising and how increasing the depth of the model by adding more layers would affect the reconstruction.

### Clustering

We performed mainly experiments with the  $k$ -sparse AE for the clustering task. Here, we tested the influence of the decreasing sparsity level and a warm-up phase. In the warm-up phase, the learning rate is not updated, and the length of this phase is the same as the length it takes to decrease the sparsity level to 1.

#### 4.2.3 Evaluation

We conducted several experiments to evaluate the performance of our proposed algorithm. As illustrated in Fig. 3.1, we proposed an AE model combining compression, denoising, and clustering. We divided the model into three tasks and performed the experiments described in Section 4.2.2.

The evaluation is divided according to the three tasks, but, as mentioned, a denoising experiment is performed in the compression experiment, and compression experiments are performed in the denoising evaluation. This was done because the experiments are

part of the performance of the task or needed as part of the decision-making for the experiments.

We will highlight the best-performing models in the tables in **bold**.

### Compression

As explained in Section 4.2.2, we combined testing the PyTorch Lightning training setup with testing different amounts of convolutional filters. We decided to test the following number of filters:

- (1, 1)** Simply compressing the GISAXS scattering pattern without allowing for more filters to be used.
- (1, 6)** The last convolutional layer uses the same number of filters as the dataset has different classes of distances.
- (1, 27)** The last convolutional layer uses the same number of filters as the dataset has different classes of radii.
- (6,27)** The number of distance classes is used for the number of filters throughout the compression model, but the last increases to the number of radii classes.
- (43, 64)** We use the smaller set of filters from the CompressAI implementation and divide them by 3. We only have one input channel and not three.
- (64, 107)** We use the larger set of filters from the CompressAI implementation and divide them by 3. We only have one input channel and not three.
- (128, 192)** Is the smaller set of filters from the CompressAI implementation.

Tables 4.2 and 4.3, as well as Tables 4.4 and 4.5 show that we get the same training result when we use our PyTorch Lightning implementation compared to CompressAI implementation for all losses except auxiliary loss. In the case of the auxiliary loss, our implementation performs slightly better. This is true regardless of whether we use the MSE or MS-SSIM loss. The tables also show that, while using a single filter results in excellent compression, it performs poorly in terms of reconstruction. The most interesting result of these experiments is that when we use MS-SSIM loss, only three models can produce a reconstruction. For all the other models, the reconstructed image was gray.



As a result, we chose to use only the filter sets (43, 64) and (64, 107). These were our modifications to the original CompressAI filter sets.

Model	Loss	MSE	Bpp	Aux Loss
FP(1, 1)	157675.563	242.485	0.020	36.305
FP(1, 6)	157726.453	242.563	0.124	217.812
FP(1, 27)	157701.281	242.523	0.560	980.283
FP(6, 27)	3604.289	5.542	0.559	979.936
FP(43, 64)	1360.904	2.091	1.349	2338.443
FP(64, 107)	357.034	0.546	2.237	3904.545
FP(128, 192)	308.501	0.468	4.021	7009.549

Table 4.2: Results of the Factorized Prior [5] Compression AE run with the CompressAI [6, 7] training script. The  $\lambda$  as a weight factor for the loss is set to 0.01. The primary Loss function is MSE.

Model	Loss	MSE	Bpp	Aux Loss	Version
FP(1, 1)	157675.563	242.485	0.020	36.304	0
FP(1, 6)	157726.453	242.563	0.124	217.812	1
FP(1, 27)	157701.297	242.523	0.560	980.286	2
FP(6, 27)	2868.320	4.410	0.559	979.825	3
<b>FP(43, 64)</b>	<b>1192.837</b>	<b>1.832</b>	<b>1.348</b>	<b>2337.879</b>	4
<b>FP(64, 107)</b>	<b>356.673</b>	<b>0.545</b>	<b>2.237</b>	<b>3904.538</b>	5
FP(128, 192)	305.524	0.464	4.021	7009.542	6

Table 4.3: Results of the Factorized Prior [5] Compression AE run with the Lightning AI [18, 19] front-end. The  $\lambda$  as a weight factor for the loss is set to 0.01. The primary Loss function is MSE.

Model	Loss	MS-SSIM	Distortion	Bpp	Aux Loss
FP(1, 1)	3.171	0.790	0.210	0.015	30.916
FP(1, 6)	15.040	0.0	1.0	0.040	100.645
FP(1, 27)	15.021	0.0	1.0	0.021	0.025
FP(6, 27)	15.022	0.0	1.0	0.022	0.022
FP(43, 64)	1.954	0.957	0.043	1.313	2310.637
FP(64, 107)	3.194	0.934	0.067	2.187	3879.112
FP(128,192)	15.014	0.0	1.0	0.014	217.056

Table 4.4: Results of the Factorized Prior [5] Compression AE run with the CompressAI [6, 7] training script. The  $\lambda$  as a weight factor for the loss is set to 15. The primary Loss function is MS-SSIM.

Model	Loss	MS-SSIM	Distortion	Bpp	Aux Loss	Version
FP(1, 1)	3.171	0.790	0.210	0.015	30.916	7
FP(1, 6)	15.040	0.0	1.0	0.040	61.220	8
FP(1, 27)	15.021	0.0	1.0	0.021	0.023	9
FP(6, 27)	15.021	0.0	1.0	0.021	0.020	10
<b>FP(43, 64)</b>	<b>1.874</b>	<b>0.962</b>	<b>0.038</b>	<b>1.308</b>	<b>2306.742</b>	11
<b>FP(64, 107)</b>	<b>2.437</b>	<b>0.983</b>	<b>0.017</b>	<b>2.179</b>	<b>3860.838</b>	12
FP(128,192)	15.014	0.0	1.0	0.014	0.344	13

Table 4.5: Results of the Factorized Prior [5] Compression AE run with the Lightning AI [18, 19] front-end. The  $\lambda$  as a weight factor for the loss is set to 15. The primary Loss function is MS-SSIM.

Since we found that models trained using MS-SSIM loss as the reconstruction loss function are not all able to reconstruct images, we chose to test the different  $\lambda$  values for the quality settings only using MS-SSIM loss and not MSE loss. The result of this experiment, seen in Table 4.6, shows that metrics 1 and 3 perform the best when we use the FP model. Although metric 3 does a slightly better job reconstructing the GISAXS scattering patterns, it needs almost twice the number of bits per pixel. We also decided to continue with the quality metric 1 due to MS-SSIM converging faster, which would allow us to employ an early stoppage in case we want to use the second dataset with 60000 simulated GISAXS scattering patterns. This can be seen in Fig. 4.4.

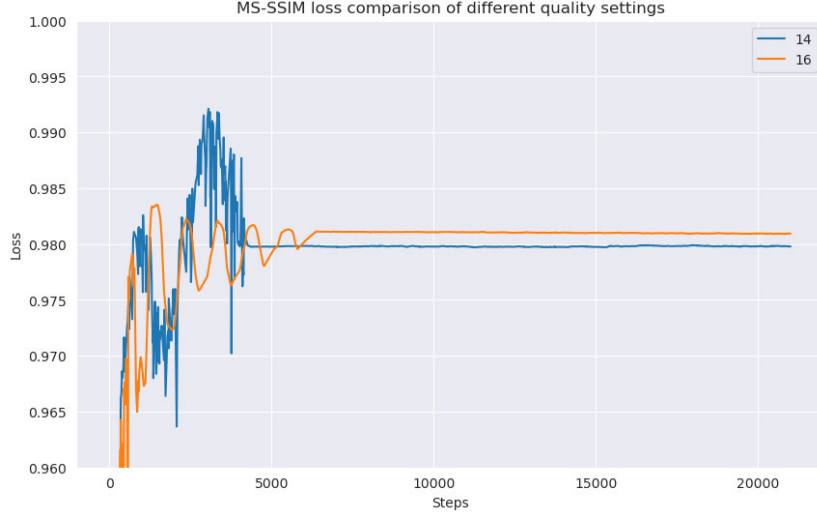


Figure 4.4: The MS-SSIM loss for the FP models 14 and 16.

Metric	1	2	3	4	5	6	7	8
$\lambda$	<b>2.40</b>	4.58	<b>8.73</b>	16.64	31.73	60.50	115.37	220.00
Loss	<b>0.405</b>	1.102	<b>1.295</b>	2.075	2.734	2.401	4.895	8.225
MS-SSIM	<b>0.980</b>	0.970	<b>0.981</b>	0.955	0.955	0.979	0.977	0.973
Distortion	<b>0.020</b>	0.030	<b>0.019</b>	0.045	0.045	0.021	0.023	0.027
Bpp	<b>0.357</b>	0.965	<b>1.128</b>	1.320	1.321	1.150	2.209	2.211
Aux Loss	<b>316.975</b>	1939.027	<b>2139.264</b>	2316.424	2316.652	2161.291	3882.946	3884.008
Filter	<b>43, 64</b>	43, 64	<b>43, 64</b>	43, 64	43, 64	43, 64	64, 107	64, 107
Version	14	15	16	17	18	19	20	21

Table 4.6: Metric test with MS-SSIM. With metric 6 it should change to the model using 64 and 107 as the number of channels.

In the next experiment, we tested the different compression models mentioned in Section 2.2. Here, the Residual Joint Autoregressive and Hierarchical Priors Compression AE with Anchor Blocks (AN) model performed the best for both the MSE and MS-SSIM loss. This can be seen in Tables 4.7 and 4.8. In Table 4.7, we also see that the Mean and Scale Hyperprior Compression AE (MSH) model shows better results. This is related to the fact that to obtain a working model for the Factorized Prior Compression AE (FP) model with ReLU as activation function, Scale Hyperprior Compression AE (SH), and MSH model we needed to increase the number of filter for the convolutional layers from 43 and 64 to 64 and 107. Otherwise, these models would have an MS-SSIM value of 0.0. We included these models in the results to show that these models would be

able to compress and reconstruct simulated GISAXS scattering patterns. We decided to continue the experiments with the AT model as the best-performing model because during training, the reconstruction results recorded revealed that the MS-SSIM metric indicated a potentially worse performance. We determine that the AT model had fewer artifacts than the reconstructed images of the MSH model. You can find example images in Fig. 4.10 (AT) and 4.8 (MSH).

Model	Loss	MS-SSIM	Distortion	Bpp	Aux Loss
FP(43, 64) + GND	0.405	0.980	0.020	0.357	316.975
FP(64, 107) + ReLU	0.613	0.858	0.142	0.274	0.050
SH(64, 107)	0.080	0.987	0.013	0.048	1116.982
<b>MSH(64, 107)</b>	<b>0.072</b>	<b>0.996</b>	<b>0.004</b>	<b>0.062</b>	<b>1960.890</b>
JAHP(64, 64)	0.141	0.972	0.028	0.074	2187.948
An(64)	0.250	0.932	0.068	0.086	2290.003
<b>At(64)</b>	<b>0.114</b>	<b>0.982</b>	<b>0.018</b>	<b>0.072</b>	<b>2162.043</b>

Table 4.7: MS-SSIM trained using quality metric 1.

Model	Loss	MSE	MS-SSIM	Distortion	Bpp	Aux Loss
FP(43, 64) + GND	100.484	0.847	0.448	0.552	1.345	2335.265
FP(64, 107) + ReLU	2.972	6.168e-03	0.956	0.044	2.250	3796.063
SH(64, 107)	52.676	0.447	0.284	0.716	0.313	2336.813
MSH(64, 107)	64.491	0.550	0.539	0.461	0.117	2337.351
JAHP(64, 64)	47.231	0.402	0.426	0.574	0.126	2335.211
An(64)	2.174	0.017	0.862	0.138	0.151	2318.844
<b>At(64)</b>	<b>1.764</b>	<b>0.014</b>	<b>0.834</b>	<b>0.166</b>	<b>0.162</b>	<b>2317.046</b>

Table 4.8: MSE trained using quality metric 1.

Tables 4.9 and 4.10 show the results when we tested how effective the AT models were in reconstructing different input-output pairs. The best-performing model for the MS-SSIM loss is when we want to compress simulated GISAXS scattering patterns that include the detector gaps. This result is somewhat misleading, as the improved performance is directly due to the presence of detector gaps. They introduce a new structure to the scattering patterns, which is straightforward to learn and reconstruct. That reconstructing the missing information that exists due to the detector gaps by using the scattering

patterns with decor gaps as input and the clean patterns as target perform the worst of the three is not that surprising since we only use convolutional layers in our model and only add fully connected layers during the denoising task. Despite this, we will continue to use masked input and clean target output from here on. However, we will cease experiments with the MSE loss function because it proves unreliable. This can be seen in Table 4.9. Here, we see similar results for the three cases. However, suppose that we check the MS-SSIM loss that we additionally calculated for the MSE loss evaluation. In that case, the values for MS-SSIM vary considerably, as seen in Fig 4.10.

in	out	Loss	MS-SSIM	Distortion	Bpp	Aux Loss	Version
gap	gap						
✓	✓	0.103	0.988	0.012	0.073	2170.004	2
✗	✗	0.114	0.982	0.018	0.072	2162.043	0
✓	✗	0.146	0.972	0.028	0.080	2248.583	4

Table 4.9: MS-SSIM for different input-output pairs using AT(64).

in	out	Loss	MSE	MS-SSIM	Distortion	Bpp	Aux Loss	Version
gap	gap							
✓	✓	1.912	0.015	0.768	0.232	0.162	2311.497	3
✗	✗	1.764	0.014	0.834	0.166	0.162	2317.046	1
✓	✗	1.661	0.013	0.788	0.212	0.159	2314.307	5

Table 4.10: MSE for different input-output pairs using AT(64).

In our final experiment with the compression models, we evaluated the effectiveness of different noise functions and assessed how well the model denoised images during the compression and decompression phases. Table 4.11 shows that we get the best results when we add Gaussian noise (AWGN). If we add Bernoulli noise (BMN) the reconstruction accuracy drops slightly, and adding Poisson noise destroys the reconstruction. We still decided to use both Gaussian and Bernoulli noise, since Bernoulli noise appears closer to the real noise we get from experimental GISAXS scattering patterns.

Noise	Loss	MS-SSIM	Distortion	Bpp	Aux Loss	Version
no	0.146	0.972	0.028	0.080	2248.583	4
<b>Gaussian</b>	<b>0.085</b>	<b>0.989</b>	<b>0.011</b>	<b>0.058</b>	<b>1877.479</b>	6
Bernoulli	0.154	0.970	0.030	0.082	2250.811	7
Poisson	2.419	0.0	1.0	0.019	62.798	8
<b>Gaussian &amp; Bernoulli</b>	<b>0.111</b>	<b>0.984</b>	<b>0.016</b>	<b>0.072</b>	<b>2133.349</b>	9

Table 4.11: Training denoising of masked input to clean target with MS-SSIM loss and quality metric 1.

### Denoising

We will start evaluating the denoising task by exploring various methods for integrating the DAE model into our compression framework. The integration options are shown in Fig. 3.2b. For denoising the latent space  $z$  located directly after the second compression encoder, we included a leaky ReLU activation function in the decoder for the best-performing model. We got better results than without the addition of the DAE model. This can be seen in Table 4.12. We got even better results by adding the DAE model before the first compression decoder and using the latent space  $\hat{z}$ . Here, we finally reached a reconstruction of 0.99 with 1.0 being the best possible value. Combining these approaches resulted in a slightly worse performance than using only the  $z$  latent space version.

The results for the first latent space after the first compression encoder and before the second decoder were not good. When using the latent space  $y$  after the first compression encoder, the reconstruction with the MS-SSIM loss is 0.0 for every variation we tested. Table 4.12 shows no other values for this test than the MS-SSIM loss and the distortion because we stopped the training directly after it started with a 0.0 MS-SSIM loss. In previous instances where this happened the MS-SSIM loss did not recover as seen in the other experiments with a AE model. Additionally, for the first latent space, the training duration took extremely long if we wanted to train for 3000 epochs as we have done for the other experiments. Using the latent space  $\hat{y}$  located before the second compression decoder, we stopped the test after 135 epochs and 1 hour and 39 minutes. At that point, the important metrics of the reconstruction loss and the compression reached nearly stable values and only the auxiliary loss was still decreasing. The results for a single model using the  $\hat{z}$  latent space and using two models, one for the  $\hat{y}$  and the other one

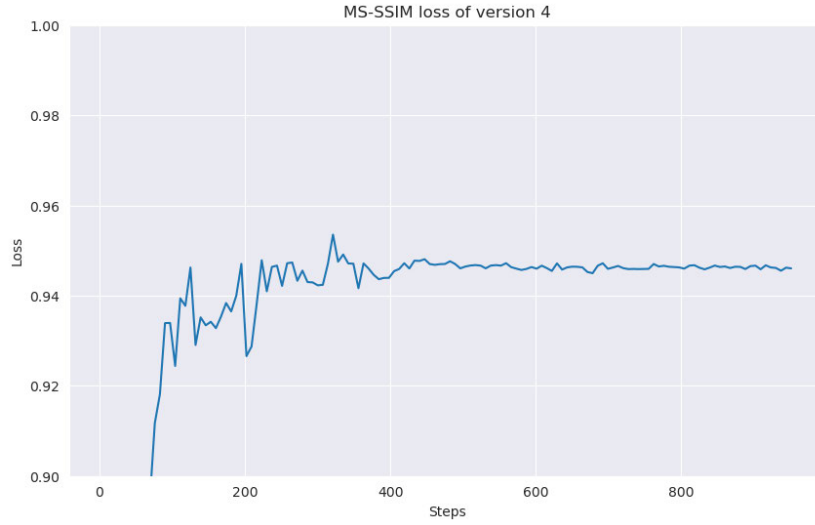


Figure 4.5: The MS-SSIM loss for the denoising model 4.

for  $\hat{z}$  performed worse than the compression model with which we compared it. This is another reason why we stopped training. The training process is shown in Fig. 4.5.

Input	Loss	MS-SSIM	Distortion	Bpp	Aux Loss	Version
no	0.111	0.984	0.016	0.072	2133.349	9
$z$	0.097	0.988	0.012	0.068	2066.906	0
$\hat{z}$	<b>0.080</b>	<b>0.990</b>	<b>0.010</b>	<b>0.056</b>	<b>1811.330</b>	1
$z$ & $\hat{z}$	0.099	0.987	0.013	0.068	2055.411	2
$y$	-	0.0	1.0	-	-	3
$\hat{y}$	0.209	0.946	0.054	0.080	-	4
$\hat{y}$ & $\hat{z}$	0.223	0.940	0.060	0.078	-	5

Table 4.12: Adding denoising to the compression model. The version for the model without noise refers to the compression tests.

We then tested the influence of over-complete, complete, and under-complete fully connected layers on the model’s performance. The results showed that the over-complete DAE model had the same reconstruction performance as the model without it but had a worse compression score. The under-complete version has performed between the over-complete and the previously tested complete version. The results are shown in Table 4.13.

Additionally, we test whether increasing the depth (more hidden layers in the AE) influences performance. Expanding the depth from one to two levels reduced the performance. This is shown in Table 4.14. Given the deteriorating results, we decided to discontinue this experiment.

Input	Loss	MS-SSIM	Distortion	Bpp	Aux Loss	Version
over-complete	0.115	0.984	0.016	0.077	2198.785	6
under-complete	0.096	0.987	0.013	0.066	2050.161	7
<b>complete</b>	<b>0.080</b>	<b>0.990</b>	<b>0.010</b>	<b>0.056</b>	<b>1811.330</b>	1

Table 4.13: Testing denoising with fully connected AE with depth 1.

Input	Loss	MS-SSIM	Distortion	Bpp	Aux Loss	Version
<b>1</b>	<b>0.080</b>	<b>0.990</b>	<b>0.010</b>	<b>0.056</b>	<b>1811.330</b>	1
2	0.096	0.988	0.012	0.065	2012.214	8

Table 4.14: Testing denoising with fully connected AE with increasing depth level.

To determine whether the transitional approach should be considered, we tested the performance of the two models that we needed for this approach. We then compared the results with our current best-performing model. In the transitional approach, we used the encoder of a model trained with noisy and masked simulated GISAXS scattering patterns and the decoder of a model trained with noisy detector gapless simulated GISAXS scattering patterns. Although both models outperformed those used as the basis for our integration approach, the results of the model intended for use as the decoder still fell short of our best-performing model. Since the decoder model performs worse than our best-performing model and creating the transition between the models is not as straightforward as we would like, we decided to discontinue the test for this approach. The problem with the transitional approach is that our current compression model uses two latent spaces  $y$  and  $z$ . These are combined before the second conversion to a string. This would require us somehow to gain access to the  $y$  latent space of the model we use as the decoder since we have already seen that changing the  $y$  latent space might break the reconstruction. The results are shown in Table 4.15.



in	out	Loss	MS-SSIM	Distortion	Bpp	Aux Loss	Version
gap	gap						
✓	✓	0.085	0.990	0.010	0.061	1955.136	10
✗	✗	0.102	0.988	0.012	0.073	2156.510	11
✓	✗	0.111	0.984	0.016	0.072	2133.349	9

Table 4.15: MS-SSIM for different input-output pairs using AT(64). All versions can be found under the compression models.

### Clustering

Table 4.16 shows that our LSRAE model has an average MSE loss of 0.0597 which means that the model can relatively accurately reconstruct the latent space of our compression model. This means that the model can create an accurate compressed representation of the input, which makes it suitable for our purpose. Additionally, the table contains the results of the other loss functions that are part of the LSRAE. Although these results are not necessarily important for the clustering task evaluation, they might hold valuable information for future experiments. This is particularly relevant to the classification process, as this work focuses on researching pre-classification methods.

Loss	MSE	Cross Entropy	Sparse Loss	Epochs
0.1844	0.0597	1.2472	1.0286	2700

Table 4.16: Shows the different losses of the LSRAE model.

Fig. 4.6 and Table 4.17 show that when we trained a  $k$ -sparse model with  $k = 1$ , a downscaling  $k$  for half of the training duration from the original latent space size of 158 without warm-up, we no longer can use this model to create a compressed representation of the input and return to it. During training, we could see that the reconstruction of a compressed representation worked for part of the training; at some point, the number of still active neurons per input made it harder to create an accurate reconstruction from the compressed representation. From this, we can deduce that our dataset contains inputs that can be grouped and we do not get a one-to-one relationship. If we got a one-to-one relationship, each neuron in the latent space would represent one input since the number of neurons is the same as the number of input classes we had. This would result in a perfect reconstruction. However, since this is not the case, we can assume that we were

able to create multiple clusters and get a many-to-one relationship. Different simulated GISAXS scattering patterns were grouped to have the same active neuron in the latent space, although they were simulated using dissimilar parameters.

warm-up	scaling $k$	scaling epochs	val loss	train loss	epochs
✗	✓	500	407.9619	410.2629	1000

Table 4.17: Training results and setup of the  $k$ -sparse model.

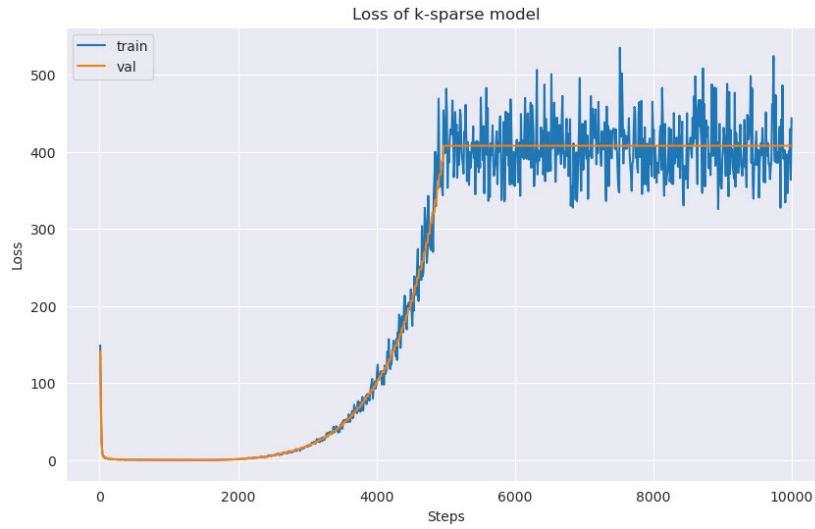
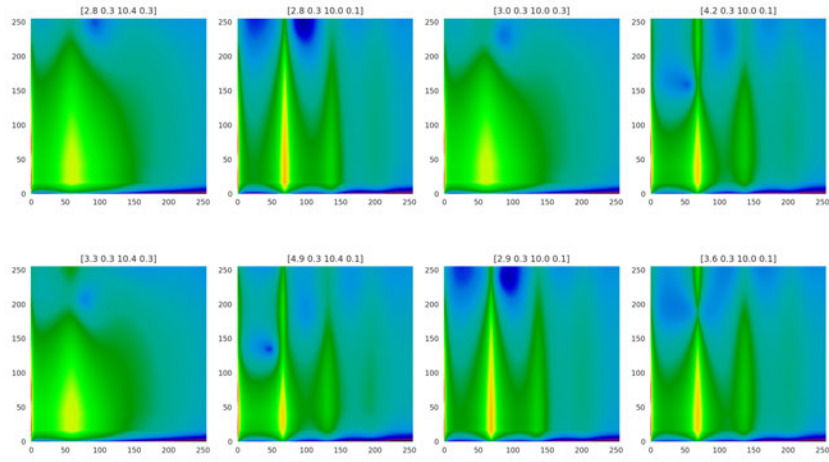


Figure 4.6: Training and validation loss of the  $k$ -sparse model.

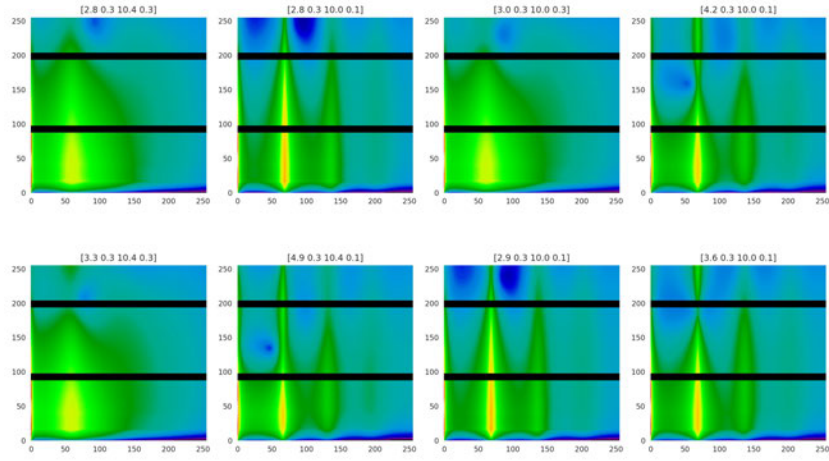
### 4.3 Discussion

After we described the training procedures and listed the results of our experiments in Sections 4.2.2 and 4.2.3, we will discuss the results in this section and find the best-performing model. Throughout the discussion, we will use the simulated GISAXS scattering patterns that can be seen in Fig. 4.7a (clean images), 4.7b (with sector gaps), and 4.7c as input for the discussion.

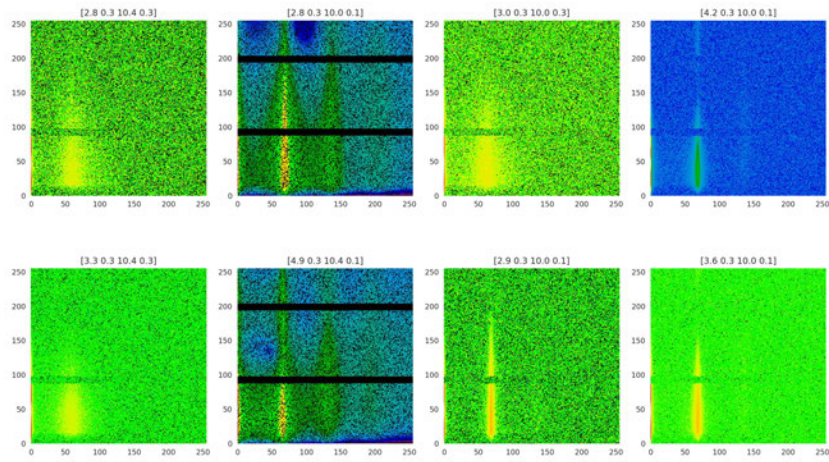
## 4 Experiments



(a) Without noise or detector gaps.



(b) With detector gaps but no noise.



(c) With detector gaps and random Gaussian and Bernoulli noise.

Figure 4.7: 8 examples of simulated GISAXS scattering patterns.

### 4.3.1 Compression

For orientation, we again list the first research question formulated in Section 1.2 and modified in Section 3.1: **"Which model, introduced in Section 2.2 and implemented with CompressAI, is the most effective at compressing and reconstructing GISAXS scattering patterns?"**

To answer the reconstruction part of the question, we will show the results of the reconstruction of some of our trained models and compare them to the input GISAXS scattering patterns shown in Fig. 4.7a, 4.7b, and 4.7c. We will consider the Mean and Scale Hyperprior Compression AE model that performed the best in Table 4.7 but was not followed further since we needed to change the experiment parameters to prove it worked. In addition, we will investigate why the MSE loss does not perform well for training despite the small loss it archives. To answer the whole research question we will show the reconstruction results of the Residual Joint Autoregressive and Hierarchical Priors Compression AE with Attention Blocks model we trained on noisy simulated GISAXS scattering patterns with detector gaps as input and clean ones as target. The validation metrics for the models can be found in Tables 4.7, 4.10, and 4.9.

Fig. 4.8 shows the result of the reconstruction with MSH(64, 107). The left side of the reconstructed image reveals numerous artifacts generated during the reconstruction process. This part of the image is not included if we consider the size of the GISAXS scattering size normally used for classification. We modified the size for simplicity reasons. The main structure was reconstructed fairly successfully, but the background again shows a lot of artifacts. However, because these artifacts consistently appear in all reconstructed images, we can infer that they will likely occur in other images. Therefore, they should not affect the classification, particularly since the structure of the scattering pattern remains intact.

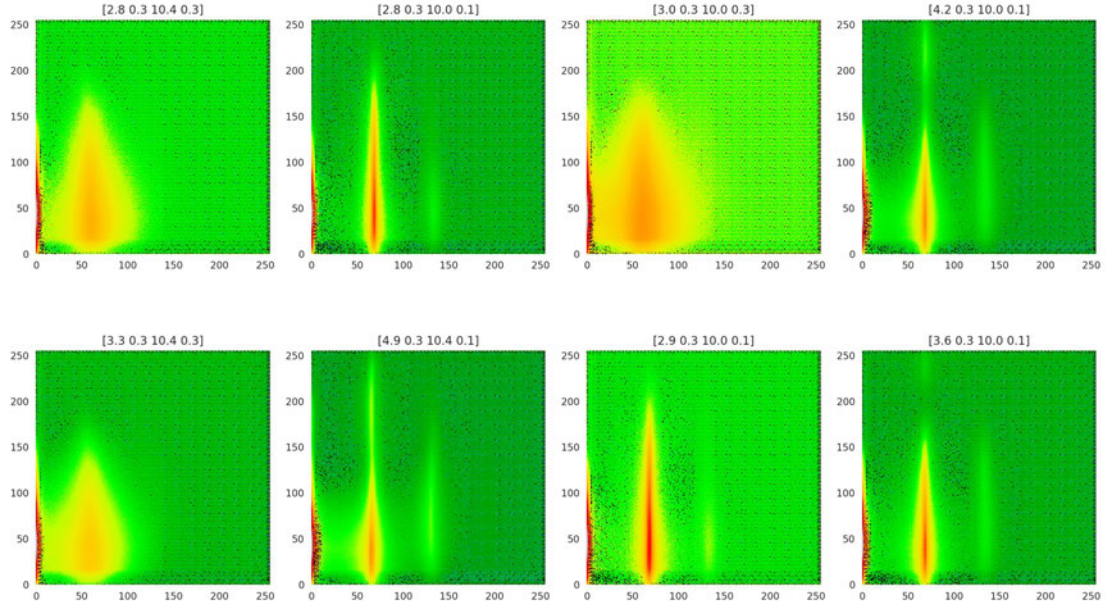


Figure 4.8: Reconstructed results of the clean example images with the MSH model.

The Residual Joint Autoregressive and Hierarchical Priors Compression AE with Attention Blocks model version number 5 boosts an impressive MSE of 0.013 but, as we can see in Fig. 4.9, we are unable to recognize any structure and the reconstructed images are riddled with artifacts. This is why we no longer used MSE loss for the compression and denoising.



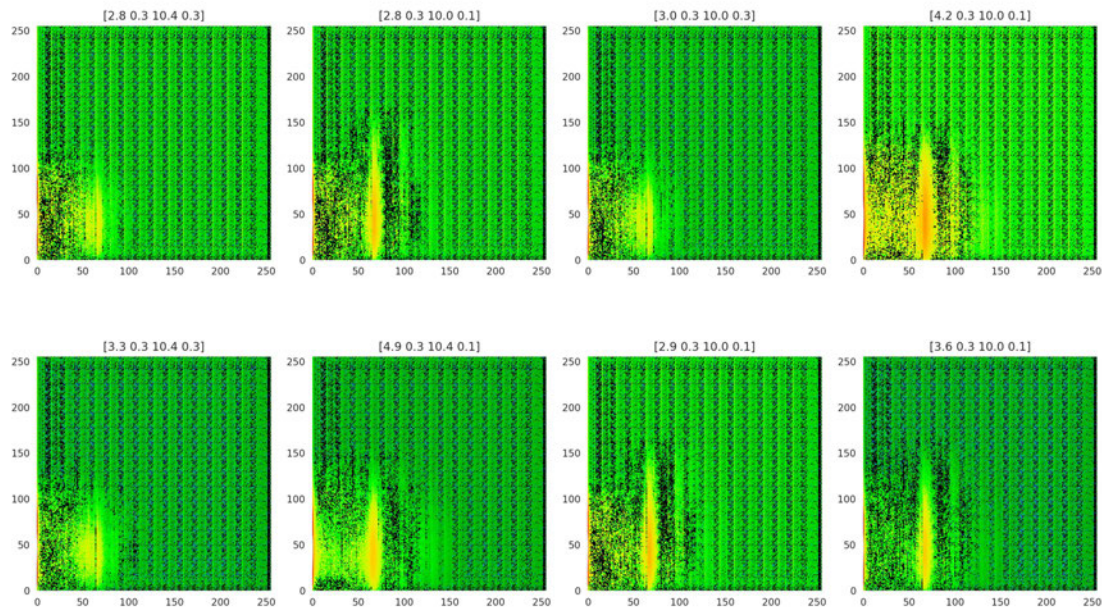


Figure 4.9: Reconstructed results of the example images with detector gaps to clean images trained with AT model number 5. MSE loss was used.

Fig. 4.10 shows the result of the Residual Joint Autoregressive and Hierarchical Priors Compression AE with Attention Blocks model version number 9 that we trained using noisy simulated GISAXS scattering patterns with detector gaps as input and clean patterns as target. As we can see, the detector gaps were successfully reconstructed and are no longer present, as is the noise. We can identify some artifacts, especially at the image borders, but the overall structure of the patterns was successfully reconstructed. In Table 4.18, we can see the success of the compression of the model for the example scattering patterns. It shows that we save 400 % per pixel in terms of memory, which is quite impressive. Sadly, we cannot show the strings to which the GISAXS scattering patterns were compressed, as we did not find an encoding that allowed us to represent the compression as a string. For this to happen, the auxiliary loss needs to be lowered. This is still possible, but it would require a longer training duration.

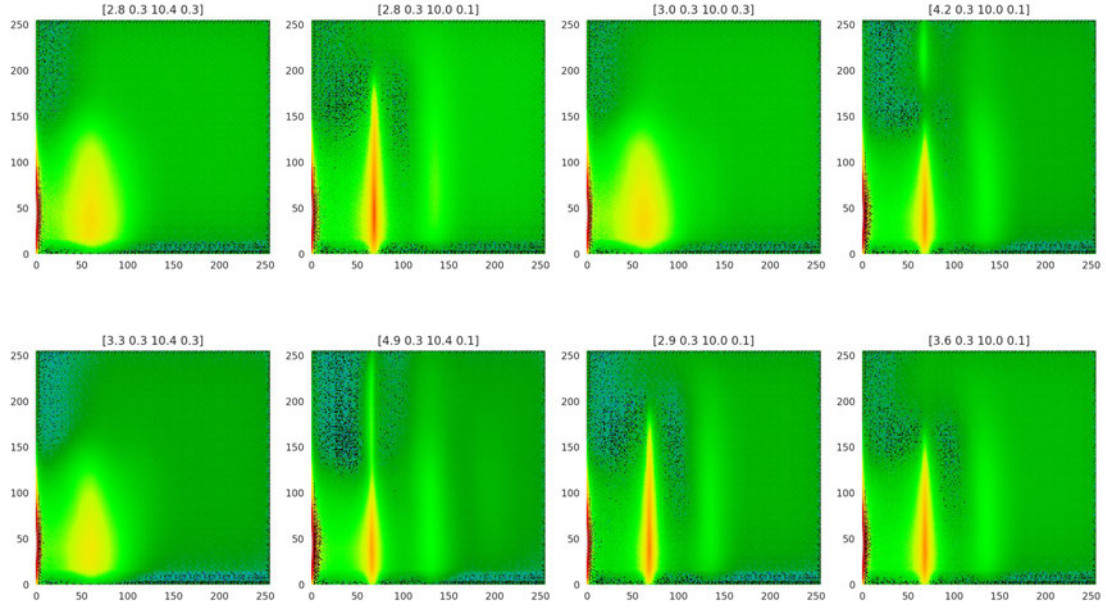


Figure 4.10: Reconstructed results of the example noisy images with detector gaps to clean images trained with AT model number 9. MS-SSIM loss was used.

image	image size in Byte	Bpp of image	$z$ size in Byte	$y$ size in Byte	$z + y$ in Byte	Bpp of $z + y$
[2.8, 0.3, 10.4, 0.3]	262144	32.0	41	609	650	0.0793
[2.8, 0.3, 10.0, 0.1]	262144	32.0	45	609	654	0.0798
[3.0, 0.3, 10.0, 0.3]	262144	32.0	41	609	650	0.0793
[4.2, 0.3, 10.0, 0.1]	262144	32.0	61	609	670	0.0818
[3.3, 0.3, 10.4, 0.3]	262144	32.0	45	609	654	0.0798
[4.9, 0.3, 10.4, 0.1]	262144	32.0	113	613	726	0.0886
[2.9, 0.3, 10.0, 0.1]	262144	32.0	49	609	658	0.0803
[3.6, 0.3, 10.0, 0.1]	262144	32.0	53	609	662	0.0808

Table 4.18: Show the size of the latent spaces and the Bpp for the image and the compression.

Although we used only a small dataset for training and validation and did not test experimental GISAXS scattering patterns, we can say that the Residual Joint Autoregressive and Hierarchical Priors Compression AE with Attention Blocks model performs

effectively in restoring and compressing simulated GISAXS scattering patterns. The reconstructed images show some artifacts, but these are mostly at the image borders or would not even be part of the pattern used for classification.

### 4.3.2 Denoising

For reference, here again the second research question formulated in Section 1.2:

**"Is it possible to employ denoising and image reconstruction to remove noise and the detector gaps from the simulation images?"**

To answer this question, we will compare the reconstructed GISAXS scattering patterns of the best-performing model with the clean example patterns in Fig. 4.7a. The best-performing model is the Residual Joint Autoregressive and Hierarchical Priors Compression AE with Attention Blocks model used in the previous step for compression. We incorporated a two-layer DAE module that operates on the second latent space  $z$  after it has been reconverted from a string back into the latent space. The results of the reconstruction can be seen in Fig. 4.11, and they prove even better than the results of the plain compression model (see Fig. 4.10). The better result is not surprising, since Table 4.12 showed that the MS-SSIM increased by 6 %.

In conclusion, we can say that including the DAE module in the compression model did not primarily help with the denoising and removing of the sector gaps, since the plain model was already quite successful with this, but instead increased the overall performance of the model. Fig. 4.11 shows the better reconstruction, and Table 4.12 shows that the performance of the model is not only better in reconstruction (a 6 % increase in MS-SSIM) but also in increasing the compression from 0.072 to 0.056 Bpp. As such, we can say that at least for our small dataset, the research question is fulfilled.



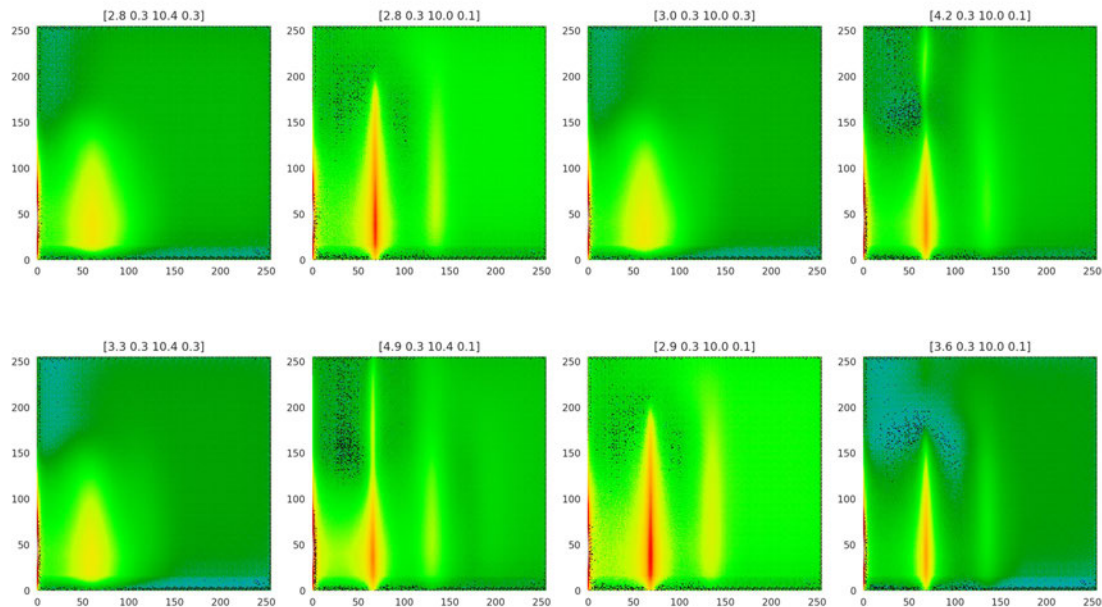


Figure 4.11: Reconstructed and denoising results of the example noisy images with detector gaps to clean images trained with AT model number 1.

### 4.3.3 Clustering

The third research question as mentioned in Section 1.2 is: **"What clusters does an AE generate if we allow only one active neuron in the latent space while having a latent space size of, e.g., the number of different images in the dataset?"**

This was originally the first question since this work was started to investigate further the creation of identities for the GISAXS scattering patterns that began in the Almamedov, Eldar [1] master thesis. During the preparation and research for this work, the focus shifted from creating identities to denoising and compression, but the clustering question remained. Regardless of the results of this question, the performance of the LSRAE used to create a further compressed latent space representation of GISAXS scattering patterns is interesting. This is because the current approach for the classification also uses a one-hot encoded representation of the different classes since the classification is interested in creating a probability vector containing the probabilities that a simulated

## 4 Experiments

59	154	39	136	82	98	13	156	124
[4.8, 0.3, 10.2, 0.3] [4.9, 0.3, 10.4, 0.1]	[2.6, 0.3, 10.2, 0.3] [2.9, 0.3, 10.2, 0.1] [3.0, 0.3, 10.2, 0.3] [4.9, 0.3, 10.0, 0.1]	[2.7, 0.3, 10.4, 0.1] [3.6, 0.3, 10.2, 0.1] [4.4, 0.3, 10.4, 0.3] [4.6, 0.3, 10.0, 0.1] [4.9, 0.3, 10.0, 0.3]	[2.6, 0.3, 10.0, 0.3] [2.6, 0.3, 10.2, 0.1] [2.7, 0.3, 10.2, 0.1] [2.7, 0.3, 10.2, 0.3] [2.9, 0.3, 10.4, 0.3] [3.2, 0.3, 10.2, 0.3] [3.6, 0.3, 10.4, 0.3] [3.7, 0.3, 10.2, 0.1] [3.8, 0.3, 10.4, 0.1] [4.1, 0.3, 10.0, 0.3] [4.1, 0.3, 10.2, 0.1] [4.2, 0.3, 10.0, 0.1] [4.2, 0.3, 10.4, 0.3] [4.9, 0.3, 10.4, 0.3] [5.1, 0.3, 10.4, 0.3] [5.2, 0.3, 10.4, 0.1] [5.2, 0.3, 10.4, 0.3]	[2.6, 0.3, 10.0, 0.1] [2.8, 0.3, 10.2, 0.3] [2.8, 0.3, 10.4, 0.3] [2.9, 0.3, 10.0, 0.1] [2.9, 0.3, 10.0, 0.3] [3.1, 0.3, 10.2, 0.1] [3.1, 0.3, 10.4, 0.1] [3.4, 0.3, 10.0, 0.1] [3.4, 0.3, 10.4, 0.3] [3.7, 0.3, 10.0, 0.1] [3.9, 0.3, 10.0, 0.1] [4.0, 0.3, 10.0, 0.1] [4.4, 0.3, 10.0, 0.3] [4.4, 0.3, 10.2, 0.3] [4.5, 0.3, 10.0, 0.3] [4.6, 0.3, 10.2, 0.3] [4.8, 0.3, 10.0, 0.3] [5.0, 0.3, 10.2, 0.1] [5.0, 0.3, 10.4, 0.1] [5.1, 0.3, 10.4, 0.1]	[2.6, 0.3, 10.4, 0.1] [2.8, 0.3, 10.0, 0.1] [2.8, 0.3, 10.2, 0.1] [2.8, 0.3, 10.4, 0.1] [2.9, 0.3, 10.2, 0.3] [3.2, 0.3, 10.2, 0.1] [3.2, 0.3, 10.4, 0.3] [3.3, 0.3, 10.0, 0.1] [3.5, 0.3, 10.0, 0.3] [3.6, 0.3, 10.0, 0.3] [3.7, 0.3, 10.4, 0.3] [3.9, 0.3, 10.4, 0.1] [4.0, 0.3, 10.0, 0.3] [4.0, 0.3, 10.2, 0.1] [4.0, 0.3, 10.4, 0.3] [4.1, 0.3, 10.2, 0.3] [4.2, 0.3, 10.0, 0.3] [4.3, 0.3, 10.0, 0.3] [4.3, 0.3, 10.4, 0.3] [4.5, 0.3, 10.2, 0.3] [4.7, 0.3, 10.0, 0.3] [5.0, 0.3, 10.0, 0.1] [5.0, 0.3, 10.2, 0.3] [5.1, 0.3, 10.2, 0.1]	[2.5, 0.3, 10.0, 0.1] [2.7, 0.3, 10.4, 0.3] [3.0, 0.3, 10.0, 0.1] [3.0, 0.3, 10.4, 0.1] [3.1, 0.3, 10.0, 0.1] [3.1, 0.3, 10.4, 0.3] [3.2, 0.3, 10.0, 0.1] [3.3, 0.3, 10.2, 0.3] [3.3, 0.3, 10.4, 0.1] [3.5, 0.3, 10.0, 0.1] [3.5, 0.3, 10.2, 0.1] [3.8, 0.3, 10.0, 0.1] [3.8, 0.3, 10.2, 0.1] [3.9, 0.3, 10.2, 0.1] [4.0, 0.3, 10.2, 0.3] [4.1, 0.3, 10.4, 0.1] [4.2, 0.3, 10.2, 0.1] [4.3, 0.3, 10.2, 0.1] [4.5, 0.3, 10.4, 0.1] [4.6, 0.3, 10.0, 0.3] [4.7, 0.3, 10.4, 0.1] [4.7, 0.3, 10.4, 0.3] [4.8, 0.3, 10.2, 0.1] [4.8, 0.3, 10.4, 0.3] [4.9, 0.3, 10.2, 0.3]	[2.6, 0.3, 10.4, 0.3] [2.7, 0.3, 10.0, 0.1] [3.0, 0.3, 10.4, 0.3] [3.1, 0.3, 10.0, 0.3] [3.3, 0.3, 10.4, 0.3] [3.4, 0.3, 10.2, 0.3] [3.4, 0.3, 10.4, 0.1] [3.5, 0.3, 10.2, 0.3] [3.5, 0.3, 10.4, 0.3] [3.6, 0.3, 10.0, 0.1] [3.6, 0.3, 10.4, 0.1] [3.7, 0.3, 10.4, 0.1] [3.8, 0.3, 10.0, 0.3] [3.8, 0.3, 10.2, 0.3] [3.9, 0.3, 10.2, 0.3] [3.9, 0.3, 10.4, 0.3] [4.0, 0.3, 10.4, 0.1] [4.1, 0.3, 10.4, 0.3] [4.2, 0.3, 10.2, 0.3] [4.2, 0.3, 10.4, 0.1] [4.3, 0.3, 10.0, 0.1] [4.3, 0.3, 10.4, 0.1] [4.4, 0.3, 10.0, 0.1] [4.4, 0.3, 10.4, 0.1] [4.5, 0.3, 10.2, 0.1] [4.5, 0.3, 10.4, 0.3] [4.9, 0.3, 10.2, 0.1] [5.0, 0.3, 10.4, 0.3]	[2.5, 0.3, 10.0, 0.3] [2.7, 0.3, 10.0, 0.3] [2.8, 0.3, 10.0, 0.3] [2.9, 0.3, 10.4, 0.1] [3.0, 0.3, 10.0, 0.3] [3.0, 0.3, 10.2, 0.1] [3.1, 0.3, 10.2, 0.3] [3.2, 0.3, 10.0, 0.3] [3.2, 0.3, 10.4, 0.1] [3.3, 0.3, 10.0, 0.3] [3.4, 0.3, 10.4, 0.1] [3.5, 0.3, 10.2, 0.1] [3.6, 0.3, 10.4, 0.1] [3.7, 0.3, 10.2, 0.1] [3.8, 0.3, 10.4, 0.1] [3.9, 0.3, 10.2, 0.3] [3.9, 0.3, 10.4, 0.3] [4.0, 0.3, 10.2, 0.3] [4.1, 0.3, 10.4, 0.1] [4.2, 0.3, 10.2, 0.3] [4.3, 0.3, 10.0, 0.3] [4.3, 0.3, 10.4, 0.1] [4.4, 0.3, 10.0, 0.1] [4.4, 0.3, 10.4, 0.1] [4.5, 0.3, 10.2, 0.1] [4.6, 0.3, 10.2, 0.1] [4.6, 0.3, 10.4, 0.3] [4.7, 0.3, 10.0, 0.1] [4.7, 0.3, 10.2, 0.1] [4.7, 0.3, 10.2, 0.3] [4.8, 0.3, 10.0, 0.1] [4.8, 0.3, 10.4, 0.1] [5.0, 0.3, 10.0, 0.3] [5.1, 0.3, 10.2, 0.3]

Table 4.19: Shows all the different simulated GISAXS scattering patterns distributed by the cluster created by the  $k$ -sparse AE. The first column contains the index of the neuron that was still active.

pattern can represent a part of the experimental GISAXS scattering patterns. Both the classifier and the LSRAE also use cross entropy as the loss function which opens up the option that we try to integrate the classifier directly into our proposed compression and denoising model.

Regarding clustering, the poor performance of the  $k$ -sparse model in reconstructing the original latent space that we used as input suggests that this approach does create groups of similar simulated GISAXS scattering patterns. The only question remaining is what these clusters look like. We can see the cluster in Table 4.19 and while we sometimes find similar parameters for the simulation in the table, it is still mostly random clustering. As such, it does not help us with the classification.

With this, we have to conclude that, at least with this approach, we do not get clusters that we can use further, and we have to answer the third research question with no. However, the experiment still resulted in an interesting realization. We can use the latent space from the compression model to train an LSRAE that uses the same approach as the classifier that the team at DESY is currently researching.

An initial test was performed where the LSRAE architecture follows the design used by the DESY for the classification. The results are shown in Table 4.20 and show at

least partially good results. A Softmax layer was added as the classification's activation function for the test, and BCE was used as the loss function.

Parameter	number of classes	Loss	MSE Loss	Label Loss	Version
$R + \sigma$	28	0.0365	0.0282	0.083	2
$D$	3	4.086	4.0224	0.636	3
$\omega$	2	0.9594	0.8812	0.7825	4
$D + \omega$	6	0.1585	0.1226	0.3583	5
all	158	0.0336	0.032	0.016	6

Table 4.20: Shows the classification results by the LSRAE model for the different classification parameters of the experiments.

## 5 Conclusion and Future Work

In this work, we tried to build a model that could compress, denoise, reconstruct, and cluster simulated GISAXS scattering patterns. It is part of a larger research project as a cooperation between the HAW and the DESY where we try to use ML to classify experiments of Au-Cluster Physics. To achieve our objectives, we propose dividing the model into three potential segments, using the latent space of each preceding segment as the input for the next. The first segment would be compression, the second the denoising, and the third the clustering. We evaluated numerous compression models using the CompressAi implementation but made the implementation compatible with the PyTorch Lightning front-end. The results demonstrate that Residual Joint Autoregressive and Hierarchical Priors Compression AE with Attention Blocks models provide superior compression and effective reconstruction of compressed images. We saw a marked performance improvement by incorporating a DAE module into the compression model, reaching an MS-SSIM of 99 % using only 0.056 bits per pixel. This would result in an average compression of around 570 % per pixel. The clustering did not work as we intended. Although we grouped different simulated GISAXS scattering patterns, the group members did not show the intended similarities. Despite the failure of the clustering segment, we were still able to use the LSRAE architecture to create a model that would use the further compressed representation of the compressed and denoised latent space to classify the GISAXS scattering patterns according to the need of the classifier. Time constraints prevented us from testing our approach on a larger dataset.

### 5.1 Limitations and Future Directions

A major limitation of this work lies in the hardware and time required to train the compression and denoising model efficiently. We recommend using a GPU cluster to run the experiments for even more epochs than we already did. This would also solve the problem of training the bigger model, as it would either no longer run for multiple weeks

or when it did, would train for more epochs and reach higher performance. Maybe it will even be possible to analyze the strings to which the latent space of the compression model is converted. Despite this option, we recommend continuing to use the smaller dataset for fine-tuning, then increasing the size of the dataset lightly and repeating the whole process until we reach the size of the larger dataset.

For the fine-tuning of the compression and denoising model, we still have the higher quality settings that we only tested for the Factorized Prior Compression AE model, but chose the lowest for the test we performed afterward.

Although the clustering failed, the idea can be kept in reserve, especially, since a part of the segment did work. We could test the LSRAE with the fully connected layers that the DESY uses for their classifier and use the same setup for a new test. Should the results be promising, we might be able to completely integrate the pre-classifying into the classifying model and then run the  $k$ -sparse model again and see what the clusters would look like.

It would also be interesting to check whether the strings resulting from the compression can be used for clustering or whether certain patterns appear for all inputs with the same parameter. For that, we either need to train longer or find a string encoding that consists of enough different characters to encode the byte arrays we get as compression output. Maybe a Chinese string encoding exists with enough characters.

# Bibliography

- [1] Almamedov, Eldar. Entwicklung eines Deep Learning Algorithmus zur Bestimmung von morphologischen Identitätsparametern aus GISAXS-Streubildern. Master's thesis, Deutsches Elektron Synchrotron & University of Applied sciences Hamburg (HAW), Hamburg, 2022.
- [2] Jinwon An and Sungzoon Cho. Variational Autoencoder based Anomaly Detection using Reconstruction Probability. 2015, <https://api.semanticscholar.org/CorpusID:36663713> (Accessed: 2024-08-04).
- [3] Komal Bajaj, Dushyant Kumar Singh, and Mohd. Aquib Ansari. Autoencoders Based Deep Learner for Image Denoising. *Procedia Computer Science*, 171:1535–1541, 2020. ISSN 18770509. doi: 10.1016/j.procs.2020.04.164, <https://linkinghub.elsevier.com/retrieve/pii/S1877050920311431> (Accessed: 2024-08-13).
- [4] Johannes Ballé, Valero Laparra, and Eero P. Simoncelli. Density Modeling of Images using a Generalized Normalization Transformation, February 2016, <http://arxiv.org/abs/1511.06281> (Accessed: 2024-06-04).
- [5] Johannes Ballé, David Minnen, Saurabh Singh, Sung Jin Hwang, and Nick Johnston. Variational image compression with a scale hyperprior, May 2018, <http://arxiv.org/abs/1802.01436> (Accessed: 2024-06-10).
- [6] Jean Bégaint, Fabien Racapé, Simon Feltman, and Akshay Pushparaja. CompressAI: A PyTorch library and evaluation platform for end-to-end compression research, November 2020, <http://arxiv.org/abs/2011.03029> (Accessed: 2024-06-10).
- [7] Jean Bégaint, Fabien Racapé, Simon Feltman, and Akshay Pushparaja. InterDigitalInc/CompressAI. InterDigital, June 2024, <https://github.com/InterDigitalInc/CompressAI> (Accessed: 2024-06-04).

- [8] Kamal Berahmand, Fatemeh Daneshfar, Elaheh Sadat Salehi, Yuefeng Li, and Yue Xu. Autoencoders and their applications in machine learning: A survey. *Artif Intell Rev*, 57(2):28, February 2024. ISSN 1573-7462. doi: 10.1007/s10462-023-10662-6, <https://doi.org/10.1007/s10462-023-10662-6> (Accessed: 2024-06-10).
- [9] L. Britnell, R. M. Ribeiro, A. Eckmann, R. Jalil, B. D. Belle, A. Mishchenko, Y.-J. Kim, R. V. Gorbachev, T. Georgiou, S. V. Morozov, A. N. Grigorenko, A. K. Geim, C. Casiraghi, A. H. Castro Neto, and K. S. Novoselov. Strong Light-Matter Interactions in Heterostructures of Atomically Thin Films. *Science*, 340(6138):1311–1314, June 2013. doi: 10.1126/science.1235547, <https://www.science.org/doi/abs/10.1126/science.1235547> (Accessed: 2024-06-12).
- [10] Benjamin Bross, Ye-Kui Wang, Yan Ye, Shan Liu, Jianle Chen, Gary J. Sullivan, and Jens-Rainer Ohm. Overview of the Versatile Video Coding (VVC) Standard and its Applications. *IEEE Transactions on Circuits and Systems for Video Technology*, 31(10):3736–3764, October 2021. ISSN 1558-2205. doi: 10.1109/TCSVT.2021.3101953, <https://ieeexplore.ieee.org/document/9503377> (Accessed: 2024-08-08).
- [11] Zhilei Chai, Wei Song, Huiling Wang, and Fei Liu. A semi-supervised auto-encoder using label and sparse regularizations for classification. *Applied Soft Computing*, 77: 205–217, April 2019. ISSN 15684946. doi: 10.1016/j.asoc.2019.01.021, <https://linkinghub.elsevier.com/retrieve/pii/S1568494619300250> (Accessed: 2024-07-31).
- [12] Minmin Chen, Kilian Weinberger, Fei Sha, and Yoshua Bengio. Marginalized Denoising Auto-encoders for Nonlinear Representations. In *Proceedings of the 31st International Conference on Machine Learning*, pages 1476–1484. PMLR, June 2014, <https://proceedings.mlr.press/v32/cheng14.html> (Accessed: 2024-08-03).
- [13] Yue Chen, Debargha Murherjee, Jingning Han, Adrian Grange, Yaowu Xu, Zoe Liu, Sarah Parker, Cheng Chen, Hui Su, Urvang Joshi, Ching-Han Chiang, Yunqing Wang, Paul Wilkins, Jim Bankoski, Luc Trudeau, Nathan Egge, Jean-Marc Valin, Thomas Davies, Steinar Midtskogen, Andrey Norkin, and Peter de Rivaz. An Overview of Core Coding Tools in the AV1 Video Codec. In *2018 Picture Coding Symposium (PCS)*, pages 41–45, June 2018. doi: 10.1109/PCS.2018.8456249, <https://ieeexplore.ieee.org/document/8456249> (Accessed: 2024-08-08).

- [14] Zhengxue Cheng, Heming Sun, Masaru Takeuchi, and Jiro Katto. Learned Image Compression With Discretized Gaussian Mixture Likelihoods and Attention Modules. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 7936–7945, June 2020. doi: 10.1109/CVPR42600.2020.00796, <https://ieeexplore.ieee.org/abstract/document/9156817> (Accessed: 2024-08-08).
- [15] Saul Dobilas. Denoising Autoencoders (DAE) — How To Use Neural Networks to Clean Up Your Data, April 2022, <https://towardsdatascience.com/denoising-autoencoders-dae-how-to-use-neural-networks-to-clean-up-your-data-cd9c19bc6915> (Accessed: 2023-07-11).
- [16] Ralph Döhrmann, Stephan Botta, Adeline Buffet, Gonzalo Santoro, Kai Schlage, Matthias Schwartzkopf, Sebastian Bommel, Johannes F. H. Risch, Roman Mannweiler, Simon Brunner, Ezzeldin Metwalli, Peter Müller-Buschbaum, and Stephan V. Roth. A new highly automated sputter equipment for in situ investigation of deposition processes with synchrotron radiation. *Rev Sci Instrum*, 84(4): 043901, April 2013. ISSN 1089-7623. doi: 10.1063/1.4798544.
- [17] Mario Einax, Wolfgang Dieterich, and Philipp Maass. Colloquium: Cluster growth on surfaces - densities, size distributions and morphologies. *Rev. Mod. Phys.*, 85(3): 921–939, July 2013. ISSN 0034-6861, 1539-0756. doi: 10.1103/RevModPhys.85.921, <http://arxiv.org/abs/1402.7095> (Accessed: 2024-08-14).
- [18] William Falcon and The PyTorch Lightning team. Lightning AI | Turn ideas into AI, Lightning fast, <https://lightning.ai/> (Accessed: 2024-06-10).
- [19] William Falcon and The PyTorch Lightning team. PyTorch Lightning, March 2019, <https://github.com/Lightning-AI/lightning> (Accessed: 2024-06-10).
- [20] F. C. Frank, J. H. van der Merwe, and Nevill Francis Mott. One-dimensional dislocations. I. Static theory. *Proceedings of the Royal Society of London. Series A. Mathematical and Physical Sciences*, 198(1053):205–216, January 1997. doi: 10.1098/rspa.1949.0095, <https://royalsocietypublishing.org/doi/10.1098/rspa.1949.0095> (Accessed: 2024-08-14).
- [21] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. Adaptive Computation and Machine Learning. The MIT Press, Cambridge, Massachusetts, 2016. ISBN 978-0-262-03561-3.



- [22] Isabel Xiaoye Green, Wenjie Tang, Matthew Neurock, and John T. Yates. Spectroscopic Observation of Dual Catalytic Sites During Oxidation of CO on a Au/TiO<sub>2</sub> Catalyst. *Science*, 333(6043):736–739, August 2011. doi: 10.1126/science.1207272, <https://www.science.org/doi/abs/10.1126/science.1207272> (Accessed: 2024-06-12).
- [23] M. Hadi Kiapour, Kevin Yager, Alexander C. Berg, and Tamara L. Berg. Materials discovery: Fine-grained classification of X-ray scattering images. In *IEEE Winter Conference on Applications of Computer Vision*, pages 933–940, March 2014. doi: 10.1109/WACV.2014.6836004, <https://ieeexplore.ieee.org/abstract/document/6836004> (Accessed: 2024-06-24).
- [24] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. *arXiv:1512.03385 [cs]*, December 2015, <http://arxiv.org/abs/1512.03385> (Accessed: 2021-02-25).
- [25] Walter Van Herck, Jonathan Fisher, and Marina Ganeva. Deep learning for x-ray or neutron scattering under grazing-incidence: Extraction of distributions. *Mater. Res. Express*, 8(4):045015, April 2021. ISSN 2053-1591. doi: 10.1088/2053-1591/abd590, <https://dx.doi.org/10.1088/2053-1591/abd590> (Accessed: 2023-07-31).
- [26] Geoffrey E. Hinton, Simon Osindero, and Yee-Whye Teh. A Fast Learning Algorithm for Deep Belief Nets. *Neural Computation*, 18(7):1527–1554, July 2006. ISSN 0899-7667. doi: 10.1162/neco.2006.18.7.1527, <https://doi.org/10.1162/neco.2006.18.7.1527> (Accessed: 2024-08-01).
- [27] Duy-Tang Hoang and Hee-Jun Kang. A survey on Deep Learning based bearing fault diagnosis. *Neurocomputing*, 335:327–335, March 2019. ISSN 09252312. doi: 10.1016/j.neucom.2018.06.078, <https://linkinghub.elsevier.com/retrieve/pii/S0925231218312657> (Accessed: 2024-08-01).
- [28] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. Densely Connected Convolutional Networks, January 2018, <http://arxiv.org/abs/1608.06993> (Accessed: 2024-08-17).
- [29] Hiroyuki Ikemoto, Kazushi Yamamoto, Hideaki Touyama, Daisuke Yamashita, Masataka Nakamura, and Hiroshi Okuda. Classification of grazing-incidence small-angle X-ray scattering patterns by convolutional neural network. *J Synchrotron Rad*, 27(4):1069–1073, July 2020. ISSN 1600-5775. doi: 10.1107/S1600577520005767,

- <http://scripts.iucr.org/cgi-bin/paper?S1600577520005767> (Accessed: 2024-08-16).
- [30] Gunar Kaune, Matthias A. Ruderer, Ezzeldin Metwalli, Weinan Wang, Sebastien Couet, Kai Schlage, Ralf Röhlberger, Stephan V. Roth, and Peter Müller-Buschbaum. In situ GISAXS study of gold film growth on conducting polymer films. *ACS Appl Mater Interfaces*, 1(2):353–360, February 2009. ISSN 1944-8244. doi: 10.1021/am8000727.
- [31] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization, January 2017.
- [32] M. A. Kramer. Autoassociative neural networks. *Computers & Chemical Engineering*, 16(4):313–328, April 1992. ISSN 0098-1354. doi: 10.1016/0098-1354(92)80051-A, <https://www.sciencedirect.com/science/article/pii/009813549280051A> (Accessed: 2024-02-05).
- [33] Rémi Lazzari. *IsGISAXS* : A program for grazing-incidence small-angle X-ray scattering analysis of supported islands. *J Appl Crystallogr*, 35(4):406–421, August 2002. ISSN 0021-8898. doi: 10.1107/S0021889802006088, <https://scripts.iucr.org/cgi-bin/paper?S0021889802006088> (Accessed: 2023-07-31).
- [34] Woong-Hee Lee, Mustafa Ozger, Ursula Challita, and Ki Won Sung. Noise Learning Based Denoising Autoencoder. *IEEE Commun. Lett.*, 25(9):2983–2987, September 2021. ISSN 1089-7798, 1558-2558, 2373-7891. doi: 10.1109/LCOMM.2021.3091800, <http://arxiv.org/abs/2101.07937> (Accessed: 2024-08-13).
- [35] Haojie Liu, Tong Chen, Peiyao Guo, Qiu Shen, Xun Cao, Yao Wang, and Zhan Ma. Non-local Attention Optimized Deep Image Compression. *IEEE Trans. on Image Process.*, 30:3179–3191, 2021. ISSN 1057-7149, 1941-0042. doi: 10.1109/TIP.2021.3058615, <http://arxiv.org/abs/1904.09757> (Accessed: 2024-08-13).
- [36] Jiliang Liu and Kevin G. Yager. Unwarping GISAXS data. *IUCrJ*, 5(6):737–752, November 2018. ISSN 2052-2525. doi: 10.1107/S2052252518012058, <https://scripts.iucr.org/cgi-bin/paper?S2052252518012058> (Accessed: 2024-08-16).

- [37] Jiliang Liu, Julien Lhermitte, Ye Tian, Zheng Zhang, Dantong Yu, and Kevin G. Yager. Healing X-ray scattering images. *IUCrJ*, 4(Pt 4):455–465, July 2017. ISSN 2052-2525. doi: 10.1107/S2052252517006212.
- [38] Shuai Liu, Charles N. Melton, Singanallur Venkatakrisnan, Ronald J. Pandolfi, Guillaume Freychet, Dinesh Kumar, Haoran Tang, Alexander Hexemer, and Daniela M. Ushizima. Convolutional neural networks for grazing incidence x-ray scattering patterns: Thin film structure identification. *MRS Communications*, 9(2):586–592, June 2019. ISSN 2159-6859, 2159-6867. doi: 10.1557/mrc.2019.26, <https://www.cambridge.org/core/journals/mrs-communications/article/convolutional-neural-networks-for-grazing-incidence-x-ray-scattering-patterns-thin-film-structure-identification/66D850A49D1BA34C4CA202D8BE974577> (Accessed: 2024-08-16).
- [39] Wei Luo, Jun Li, Jian Yang, Wei Xu, and Jian Zhang. Convolutional Sparse Autoencoders for Image Classification. *IEEE Transactions on Neural Networks and Learning Systems*, 29(7):3289–3294, July 2018. ISSN 2162-2388. doi: 10.1109/TNNLS.2017.2712793, <https://ieeexplore.ieee.org/document/7962256> (Accessed: 2024-08-04).
- [40] Alireza Makhzani and Brendan Frey. K-Sparse Autoencoders, March 2014, <http://arxiv.org/abs/1312.5663> (Accessed: 2024-02-05).
- [41] D. Marpe, H. Schwarz, and T. Wiegand. Context-based adaptive binary arithmetic coding in the H.264/AVC video compression standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7):620–636, July 2003. ISSN 1558-2205. doi: 10.1109/TCSVT.2003.815173, <https://ieeexplore.ieee.org/document/1218195> (Accessed: 2024-08-08).
- [42] P. J. Martin. Ion-based methods for optical thin film deposition. *J Mater Sci*, 21(1):1–25, January 1986. ISSN 1573-4803. doi: 10.1007/BF01144693, <https://doi.org/10.1007/BF01144693> (Accessed: 2024-08-14).
- [43] Ying Mei, Christopher Cannizzaro, Hyounghsin Park, Qiaobing Xu, Said Bogatyrev, Kevin Yi, Nathan Goldman, Robert Langer, and Daniel G. Anderson. Cell-Compatible, Multi-Component Protein Arrays with Subcellular Feature Resolution. *Small*, 4(10):1600–1604, October 2008. ISSN 1613-6810. doi: 10.1002/smll, <https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2679812/> (Accessed: 2024-06-12).

- [44] Nicole Meister, Ziqiao Guan, Jinzhen Wang, Ronald Lashley, Jiliang Liu, Julien Lhermitte, Kevin Yager, Hong Qin, Bo Sun, and Dantong Yu. Robust and scalable deep learning for X-ray synchrotron image analysis. In *2017 New York Scientific Data Summit (NYSDS)*, pages 1–6, New York, NY, USA, August 2017. IEEE. ISBN 978-1-5386-3161-4. doi: 10.1109/NYSDS.2017.8085045, <http://ieeexplore.ieee.org/document/8085045/> (Accessed: 2024-08-16).
- [45] A. Meyer. GISAXS, March 2018, <http://www.gisaxs.de/> (Accessed: 2023-07-31).
- [46] David Minnen, Johannes Ballé, and George Toderici. Joint Autoregressive and Hierarchical Priors for Learned Image Compression, September 2018, <http://arxiv.org/abs/1809.02736> (Accessed: 2024-06-04).
- [47] P. Müller-Buschbaum. A Basic Introduction to Grazing Incidence Small-Angle X-Ray Scattering. In Marian Gomez, Aurora Nogales, Mari Cruz Garcia-Gutierrez, and T.A. Ezquerra, editors, *Applications of Synchrotron Light to Scattering and Diffraction in Materials and Life Sciences*, pages 61–89. Springer, Berlin, Heidelberg, 2009. ISBN 978-3-540-95968-7. doi: 10.1007/978-3-540-95968-7\_3, [https://doi.org/10.1007/978-3-540-95968-7\\_3](https://doi.org/10.1007/978-3-540-95968-7_3) (Accessed: 2024-06-24).
- [48] Andrew Ng. Sparse autoencoder - CS294A Lecture notes, 2001.
- [49] Julien Pascal. Neural Networks: Unleashing the Power of Latent Space Compression, September 2023, <https://medium.com/@julien.pascal/neural-networks-unleashing-the-power-of-latent-space-compression-2c8630f6f6cc> (Accessed: 2024-08-13).
- [50] Phillip Lippe. Tutorial 8: Deep Autoencoders — PyTorch Lightning 2.1.4 documentation, October 2023, [https://lightning.ai/docs/pytorch/stable/notebooks/course\\_UvA-DL/08-deep-autoencoders.html](https://lightning.ai/docs/pytorch/stable/notebooks/course_UvA-DL/08-deep-autoencoders.html) (Accessed: 2024-02-07).
- [51] Konpat Preechakul, Nattanat Chatthee, Suttisak Wizadwongsa, and Supasorn Suwajanakorn. Diffusion Autoencoders: Toward a Meaningful and Decodable Representation, March 2022, <http://arxiv.org/abs/2111.15640> (Accessed: 2024-08-04).

- [52] Markus Rauscher, Rogerio Paniago, Hartmut Metzger, Zoltan Kovats, Jan Domke, Johann Peisl, Hans-Dieter Pfannes, Jörg Schulze, and Ignaz Eisele. Grazing incidence small angle x-ray scattering from free-standing nanostructures. *Journal of Applied Physics*, 86(12):6763–6769, December 1999. ISSN 0021-8979. doi: 10.1063/1.371724, <https://doi.org/10.1063/1.371724> (Accessed: 2024-08-15).
- [53] D.E. Rumelhart, G.E. Hinton, and R.J. Williams. Learning Internal Representations by Error Propagation. In *Readings in Cognitive Science*, pages 399–421. Elsevier, 1988. ISBN 978-1-4832-1446-7. doi: 10.1016/B978-1-4832-1446-7.50035-2, <https://linkinghub.elsevier.com/retrieve/pii/B9781483214467500352> (Accessed: 2024-07-31).
- [54] G. Santoro, S. Yu, M. Schwartzkopf, P. Zhang, Sarathlal Koyiloth Vayalil, J. F. H. Risch, M. A. Rübhausen, M. Hernández, C. Domingo, and S. V. Roth. Silver substrates for surface enhanced Raman scattering: Correlation between nanostructure and Raman scattering enhancement. *Applied Physics Letters*, 104(24):243107, June 2014. ISSN 0003-6951. doi: 10.1063/1.4884423, <https://doi.org/10.1063/1.4884423> (Accessed: 2024-03-07).
- [55] Matthias Schwartzkopf. *In-situ- $\mu$ GISAXS-Untersuchungen der Wachstumskinetik von Goldclustern*. PhD thesis, University of Hamburg, Hamburg, 2013.
- [56] Matthias Schwartzkopf, Adeline Buffet, Volker Körstgens, Ezzeldin Metwalli, Kai Schlage, Gunthard Benecke, Jan Perlich, Monika Rawolle, André Rothkirch, Berit Heidmann, Gerd Herzog, Peter Müller-Buschbaum, Ralf Röhlsberger, Rainer Gehrke, Norbert Striebeck, and Stephan V. Roth. From atoms to layers: In situ gold cluster growth kinetics during sputter deposition. *Nanoscale*, 5(11): 5053–5062, May 2013. ISSN 2040-3372. doi: 10.1039/C3NR34216F, <https://pubs.rsc.org/en/content/articlelanding/2013/nr/c3nr34216f> (Accessed: 2023-07-31).
- [57] Matthias Schwartzkopf, Alexander Hinz, Oleksandr Polonskyi, Thomas Strunskus, Franziska C. Löhner, Volker Körstgens, Peter Müller-Buschbaum, Franz Faupel, and Stephan V. Roth. Role of Sputter Deposition Rate in Tailoring Nanogranular Gold Structures on Polymer Surfaces. *ACS Appl. Mater. Interfaces*, 9(6): 5629–5637, February 2017. ISSN 1944-8244. doi: 10.1021/acsami.6b15172, <https://doi.org/10.1021/acsami.6b15172> (Accessed: 2024-06-12).

- [58] Mehmet Saygın Seyfioğlu, Ahmet Murat Özbayoğlu, and Sevgi Zubeyde Gürbüz. Deep convolutional autoencoder for radar-based classification of similar aided and unaided human activities. *IEEE Transactions on Aerospace and Electronic Systems*, 54(4):1709–1723, August 2018. ISSN 1557-9603. doi: 10.1109/TAES.2018.2799758, <https://ieeexplore.ieee.org/document/8283539> (Accessed: 2024-08-04).
- [59] A. Skodras, C. Christopoulos, and T. Ebrahimi. The JPEG 2000 still image compression standard. *IEEE Signal Processing Magazine*, 18(5):36–58, September 2001. ISSN 1558-0792. doi: 10.1109/79.952804, <https://ieeexplore.ieee.org/document/952804> (Accessed: 2024-08-08).
- [60] T. Stielow, R. Schmidt, C. Peltz, T. Fennel, and S. Scheel. Fast reconstruction of single-shot wide-angle diffraction images through deep learning. *Mach. Learn.: Sci. Technol.*, 1(4):045007, October 2020. ISSN 2632-2153. doi: 10.1088/2632-2153/abb213, <https://dx.doi.org/10.1088/2632-2153/abb213> (Accessed: 2024-08-16).
- [61] I. N. Stranski and L. Krastanow. Zur Theorie der orientierten Ausscheidung von Ionenkristallen aufeinander. *Monatshefte für Chemie*, 71(1):351–364, December 1937. ISSN 0026-9247, 1434-4475. doi: 10.1007/BF01798103, <http://link.springer.com/10.1007/BF01798103> (Accessed: 2024-08-14).
- [62] Gary J. Sullivan, Jens-Rainer Ohm, Woo-Jin Han, and Thomas Wiegand. Overview of the High Efficiency Video Coding (HEVC) Standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 22(12):1649–1668, December 2012. ISSN 1558-2205. doi: 10.1109/TCSVT.2012.2221191, <https://ieeexplore.ieee.org/document/6316136/?arnumber=6316136> (Accessed: 2024-08-08).
- [63] Aaron van den Oord, Nal Kalchbrenner, Oriol Vinyals, Lasse Espeholt, Alex Graves, and Koray Kavukcuoglu. Conditional Image Generation with PixelCNN Decoders, June 2016.
- [64] Pascal Vincent, Hugo Larochelle, Isabelle Lajoie, Yoshua Bengio, and Pierre-Antoine Manzagol. Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion. *J. Mach. Learn. Res.*, 11:3371–3408, December 2010. ISSN 1532-4435.
- [65] George H. Vineyard. Grazing-incidence diffraction and the distorted-wave approximation for the study of surfaces. *Phys. Rev. B*, 26(8):4146–4159, October 1982. doi:

- 10.1103/PhysRevB.26.4146, <https://link.aps.org/doi/10.1103/PhysRevB.26.4146> (Accessed: 2024-08-15).
- [66] M. Volmer and A Weber. Keimbildung in übersättigten Gebilden. *Zeitschrift für Physikalische Chemie*, 119U(1):277–301, January 1926. ISSN 2196-7156. doi: 10.1515/zpch-1926-11927, <https://www.degruyter.com/document/doi/10.1515/zpch-1926-11927/html> (Accessed: 2024-08-14).
- [67] Gregory K. Wallace. The JPEG still picture compression standard. *Commun. ACM*, 34(4):30–44, April 1991. ISSN 0001-0782, 1557-7317. doi: 10.1145/103085.103089, <https://dl.acm.org/doi/10.1145/103085.103089> (Accessed: 2024-08-08).
- [68] Boyu Wang, Ziqiao Guan, Shun Yao, Hong Qin, Minh Hoai Nguyen, Kevin Yager, and Dantong Yu. Deep learning for analysing synchrotron data streams. In *2016 New York Scientific Data Summit (NYSDS)*, pages 1–5, August 2016. doi: 10.1109/NYSDS.2016.7747813, <https://ieeexplore.ieee.org/abstract/document/7747813> (Accessed: 2024-08-16).
- [69] Boyu Wang, Kevin Yager, Dantong Yu, and Minh Hoai. X-Ray Scattering Image Classification Using Deep Learning. In *2017 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 697–704, March 2017. doi: 10.1109/WACV.2017.83, <https://ieeexplore.ieee.org/abstract/document/7926666> (Accessed: 2024-08-16).
- [70] Z. Wang, E.P. Simoncelli, and A.C. Bovik. Multiscale structural similarity for image quality assessment. In *The Thrity-Seventh Asilomar Conference on Signals, Systems & Computers, 2003*, volume 2, pages 1398–1402 Vol.2, November 2003. doi: 10.1109/ACSSC.2003.1292216, <https://ieeexplore.ieee.org/document/1292216> (Accessed: 2024-06-12).
- [71] Xiangli Yang, Zixing Song, Irwin King, and Zenglin Xu. A Survey on Deep Semi-Supervised Learning. *IEEE Transactions on Knowledge and Data Engineering*, 35(9):8934–8954, September 2023. ISSN 1558-2191. doi: 10.1109/TKDE.2022.3220219, <https://ieeexplore.ieee.org/document/9941371> (Accessed: 2024-08-04).
- [72] Y. Yoneda. Anomalous Surface Reflection of X Rays. *Phys. Rev.*, 131(5):2010–2013, September 1963. doi: 10.1103/PhysRev.131.2010, <https://link.aps.org/doi/10.1103/PhysRev.131.2010> (Accessed: 2024-08-15).

- [73] Matthew D. Zeiler, Dilip Krishnan, Graham W. Taylor, and Rob Fergus. Deconvolutional networks. In *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 2528–2535, June 2010. doi: 10.1109/CVPR.2010.5539957, <https://ieeexplore.ieee.org/document/5539957> (Accessed: 2024-08-24).
- [74] Yan Zhang, Erhu Zhang, and Wanjun Chen. Deep neural network for halftone image classification based on sparse auto-encoder. *Engineering Applications of Artificial Intelligence*, 50:245–255, April 2016. ISSN 09521976. doi: 10.1016/j.engappai.2016.01.032, <https://linkinghub.elsevier.com/retrieve/pii/S0952197616000361> (Accessed: 2024-07-31).
- [75] Yulun Zhang, Kunpeng Li, Kai Li, Bineng Zhong, and Yun Fu. Residual Non-local Attention Networks for Image Restoration, March 2019, <http://arxiv.org/abs/1903.10082> (Accessed: 2024-08-13).



# A Appendix

## A.1 Source Code

### A.1.1 k-sparse Linear Layer

```
1 class KSparseLinear(torch.nn.Linear):
2     def __init__(self, in_features: int, out_features: int, k:
3         ↪ int, bias: bool = True, device: str = None, dtype: str =
4         ↪ None) -> None:
5         super(KSparseLinear, self).__init__(in_features,
6         ↪ out_features, bias, device, dtype)
7         self.k = k
8
9     def forward(self, x: torch.Tensor) -> torch.Tensor:
10         z = super().forward(x)
11         _, indices = torch.topk(torch.abs(z), self.k)
12         mask = torch.zeros(z.size()).to(z.device)
13         mask = mask.scatter_(1, indices, 1)
14         return torch.mul(z, mask)
```

### A.1.2 Decreasing sparsity level (k)

```
1 class UpdateSparsityLevel(Callback):
2     def __init__(self, initial_sparsity: int, final_sparsity: int)
3         ↪ -> None:
4         super(UpdateSparsityLevel, self).__init__()
5         self.initial_sparsity = initial_sparsity
6         self.final_sparsity = final_sparsity
7         self.sparsity_levels = None
```

```

7
8  def setup(self, trainer: pl.Trainer, pl_module:
    ↳ pl.LightningModule, stage: str) -> None:
9      duration = trainer.max_epochs // 4
10     self.sparsity_levels = np.hstack((
11         np.linspace(self.initial_sparsity,
12             ↳ self.final_sparsity, duration,
13             ↳ dtype=np.int32),
14         np.repeat(self.final_sparsity, trainer.max_epochs
15             ↳ - duration + 1)),
16         dtype=np.int32)[:trainer.max_epochs]
17
18  def on_train_start(self, trainer: "pl.Trainer", pl_module:
    ↳ "pl.LightningModule") -> None:
19
20     l = pl_module.encoder.net[-1]
21     l.k = self.sparsity_levels[0]
22     pl_module.log('k-sparse', l.k, prog_bar=True,
23         ↳ batch_size=pl_module.batch_size)
24
25  def on_train_epoch_end(self, trainer: pl.Trainer, pl_module:
    ↳ pl.LightningModule) -> None:
26
27     l = pl_module.encoder.net[-1]
28     if trainer.current_epoch + 1 < trainer.max_epochs:
29         l.k = self.sparsity_levels[trainer.current_epoch + 1]
30     else:
31         l.k = self.sparsity_levels[-1]
32     pl_module.log('k-sparse', int(l.k), prog_bar=True,
33         ↳ batch_size=pl_module.batch_size)

```

### A.1.3 Warm Up

```

1  # Learning rate warm-up
2  def optimizer_step(self, epoch: int, batch_idx: int, optimizer:
    ↳ Union[Optimizer, LightningOptimizer], optimizer_closure:
    ↳ Optional[Callable[[], Any]] = None) -> None:
3      # update params
4      optimizer.step(closure=optimizer_closure)
5

```

```
6     # manually warm up lr without a scheduler
7     if self.trainer.current_epoch < self.trainer.max_epochs // 2:
8         lr_scale = 1.
9         for pg in optimizer.param_groups:
10             pg['lr'] = lr_scale * self.learning_rate
```

### A.1.4 Gaussian white Additive noise

```
1 def __add_gaussian_noise(img_in: torch.Tensor) -> torch.Tensor:
2     return (img_in + torch.normal(torch.mean(img_in), 0.1,
3     ↪ img_in.shape)).clip(img_in.min(), img_in.max())
```

### A.1.5 Bernoulli-masked noise

```
1 def __add_bernoulli_noise(img_in: torch.Tensor) -> torch.Tensor:
2     a = 0.7 * torch.ones(img_in.shape)
3     return img_in * torch.bernoulli(a)
```

### A.1.6 Poisson noise

```
1 def __add_poisson_noise(img_in: torch.Tensor) -> torch.Tensor:
2     a = 1 * torch.ones(img_in.shape)
3     p = torch.poisson(a)
4     p_norm = p / p.max()
5     return (img_in + p_norm).clip(img_in.min(), img_in.max())
```

### A.1.7 PyTorch Datasets

```
1 class CompressionDataset(Dataset):
2     """
3     Constructs a pytorch Dataset for a denoising autoencoder and a
4     ↪ later classification using the latent space of
5     the autoencoder. The format of the files for the dataset is
6     ↪ .npz
```

```
5
6     Args:
7         dataset: Path to the .npz file for the image dataset.
8         transform: Optional transformation applied to the images,
9         ↪ so that they can be used by the autoencoder.
10    """
11    def __init__(self, data_dir: Path, dataset: str, transform:
12    ↪ Optional[Callable] = None, add_noise: bool = False):
13        super(CompressionDataset, self).__init__()
14        self._x = np.load(data_dir / f'masked{dataset}',
15        ↪ mmap_mode='r')
16        self._y = np.load(data_dir / f'clean{dataset}',
17        ↪ mmap_mode='r')
18        self.transform = transform
19        self.add_noise = add_noise
20
21    def __len__(self) -> int:
22        """
23        Returns the number of images included in the input file.
24
25        Returns:
26            int: Size of the DataSet.
27        """
28        # A DataSet must know its size
29        return self._x.shape[0]
30
31    def __getitem__(self, item: int) -> Tuple[torch.Tensor,
32    ↪ torch.Tensor]:
33        """
34
35        Args:
36            item (int): Index
37
38        Returns:
39            Tuple[Any, Any]: (noisy image, clean image).
40        """
41        image_in = self._x[item]
42        image_in = Image.fromarray(image_in)
43        image_out = self._y[item]
```

```
39     image_out = Image.fromarray(image_out)
40
41     if self.transform is not None:
42         image_in = self.transform(image_in)
43         image_out = self.transform(image_out)
44
45     if self.add_noise:
46         if random.choice([True, False]):
47             image_in = self.__add_gaussian_noise(image_in)
48         else:
49             image_in = self.__add_bernoulli_noise(image_in)
50
51     return image_in, image_out
```

### A.1.8 PyTorch Lightning's DataModule

```
1 class CompressionDataModule(pl.LightningDataModule):
2     def __init__(self, data_dir: Path, batch_size: int,
3         ↪ num_workers: int, test_batch_size: Optional[int] = None,
4         ↪ train_transform: Optional[Callable] = None,
5         ↪ test_transform: Optional[Callable] = None, add_noise: bool
6         ↪ = False):
7         super(CompressionDataModule, self).__init__()
8         self.batch_size = batch_size
9         if test_batch_size is None:
10             self.test_batch_size = batch_size
11         else:
12             self.test_batch_size = test_batch_size
13         self.num_workers = num_workers
14         self.data_dir = data_dir
15         self.dims = (1, 256, 256)
16         self.train_transform = train_transform
17         self.test_transform = test_transform
18         self.add_noise = add_noise
19         self.save_hyperparameters()
20
21     def prepare_data(self):
22         pass
```

```

19
20     def setup(self, stage: Optional[str] = None):
21         #if stage == 'fit' or stage is None:
22             self.train_dataset = CompressionDataset(self.data_dir,
23                 ↳ '_train.npy', transform=self.train_transform,
24                 ↳ add_noise=self.add_noise)
25             self.val_dataset = CompressionDataset(self.data_dir,
26                 ↳ '_test.npy', transform=self.test_transform,
27                 ↳ add_noise=self.add_noise)
28
29         #if stage == 'test' or stage is None:
30             self.test_dataset = CompressionDataset(self.data_dir,
31                 ↳ '_test.npy', transform=self.test_transform,
32                 ↳ add_noise=self.add_noise)
33
34     def train_dataloader(self):
35         return DataLoader(self.train_dataset,
36             ↳ batch_size=self.batch_size, shuffle=True,
37             ↳ num_workers=self.num_workers)
38
39     def val_dataloader(self):
40         return DataLoader(self.val_dataset,
41             ↳ batch_size=self.test_batch_size, shuffle=False,
42             ↳ num_workers=self.num_workers)
43
44     def test_dataloader(self):
45         return DataLoader(self.test_dataset,
46             ↳ batch_size=self.test_batch_size, shuffle=False,
47             ↳ num_workers=self.num_workers)

```

### A.1.9 PyTorch Lightning class for the compression models

```

1 class LightningCompressionModel(pl.LightningModule):
2     def __init__(self):
3         super(LightningCompressionModel, self).__init__()
4         self.avg_loss = AverageMeter()
5         self.avg_mse_loss = AverageMeter()
6         self.avg_ms_ssim_loss = AverageMeter()

```

```
7         self.avg_bpp_loss = AverageMeter()
8         self.avg_aux_loss = AverageMeter()
9         self.avg_distortion = AverageMeter()
10
11         self.clip_max_norm = 1.0
12
13         # Example input array needed for visualizing the graph of
14         ↪ the network
15         self.example_input_array = torch.rand((16, 1, 256, 256))
16         self.automatic_optimization = False
17
18     def configure_optimizers(self) -> OptimizerLRScheduler:
19         """
20         Separate parameters for the main optimizer and the
21         ↪ auxiliary optimizer.
22
23         Return two optimizers
24         """
25         conf_1 = {
26             'net': {'type': 'Adam', 'lr': 1e-4},
27             'aux': {'type': 'Adam', 'lr': 1e-3},
28         }
29         optimizer_1 = net_aux_optimizer(self, conf_1)
30         scheduler =
31         ↪ optim.lr_scheduler.ReduceLROnPlateau(optimizer_1['net'],
32         ↪ 'min')
33
34         return [optimizer_1['net'], optimizer_1['aux']],
35         ↪ [scheduler]
36
37     def training_step(self, batch, batch_idx):
38         optimizer_1, aux_optimizer = self.optimizers()
39         optimizer_1.zero_grad()
40         aux_optimizer.zero_grad()
41
42         x_in, x_out = batch
43
44         out_hat = self.forward(x_in)
```

```
41     out_criterion = self.criterion(out_hat, x_out)
42     self.manual_backward(out_criterion['loss'])
43     if self.clip_max_norm > 0:
44         torch.nn.utils.clip_grad_norm_(self.parameters(),
45                                         ↪ self.clip_max_norm)
46     optimizer_1.step()
47
48     aux_loss = self.aux_loss()
49     self.manual_backward(aux_loss)
50     aux_optimizer.step()
51
52     out_criterion['aux_loss'] = aux_loss
53     self.log_dict(out_criterion, prog_bar=True, logger=True)
54
55     return out_criterion
56
57 def on_train_epoch_end(self) -> None:
58     sch = self.lr_schedulers()
59
60     # If the selected scheduler is a ReduceLROnPlateau
61     ↪ scheduler.
62     if isinstance(sch,
63                   ↪ torch.optim.lr_scheduler.ReduceLROnPlateau):
64         sch.step(self.trainer.callback_metrics['val_loss'])
65
66 def validation_step(self, batch, batch_idx):
67     x_in, x_out = batch
68
69     out_hat = self.forward(x_in)
70
71     out_criterion = self.criterion(out_hat, x_out)
72     out_criterion['aux_loss'] = self.aux_loss()
73
74     out_criterion = {re.sub(r'loss', 'val_loss', f'{k}'): v
75                     ↪ for k, v in out_criterion.items()}
76     out_criterion = {re.sub(r'distortion', 'val_distortion',
77                             ↪ f'{k}'): v for k, v in out_criterion.items()}
78
79     self.log_dict(out_criterion, prog_bar=True, logger=True)
```



```
75
76     return out_criterion
77
78     def test_step(self, batch, batch_idx):
79         x_in, x_out = batch
80
81         out_hat = self.forward(x_in)
82
83         out_criterion = self.criterion(out_hat, x_out)
84         out_criterion['aux_loss'] = self.aux_loss()
85
86         out_criterion = {re.sub(r'loss', 'test_loss', f'{k}'): v
87             ↪ for k, v in out_criterion.items()}
88         out_criterion = {re.sub(r'distortion', 'test_distortion',
89             ↪ f'{k}'): v for k, v in out_criterion.items()}
89
90         self.log_dict(out_criterion, prog_bar=True, logger=True)
91
92     return out_criterion
```

#### A.1.10 PyTorch Lightning Trainer

```
1 pl.seed_everything(42)
2
3 datamodule = CompressionDataModule(DATA_DIR, 16, 8, 64,
4     ↪ train_transforms, test_transforms, add_noise=True)
5 datamodule.setup('')
6
7 # Create a PyTorch Lightning trainer with the generation callback
8 trainer = pl.Trainer(
9     default_root_dir=os.path.join(MODELS_DIR,
10     ↪ 'cheng2020-attn-gray'),
11     accelerator='auto',
12     devices=1,
13     max_epochs=3000,
14     log_every_n_steps=7,
15     callbacks=[
16         ModelCheckpoint(save_weights_only=True),
```

```

15     LearningRateMonitor('epoch'),
16     RichModelSummary(-1),
17     RichProgressBar(),
18     GenerateCallback(get_train_images(16, datamodule),
19         ↪ every_n_epochs=50),
20 ]
21 )
22 trainer.logger._log_graph = False # If True, we plot the
23     ↪ computation graph in tensorboard
24 trainer.logger._default_hp_metric = None # Optional logging
25     ↪ argument that we don't need
26
27 model = Cheng2020AttentionGray(64, lambda=2.40, metric='ms-ssim',
28     ↪ input_gap=True, output_gap=False) # , 0.0018, 'mse')# , 2.40)
29 trainer.fit(model, datamodule)
30 # Test best model on validation and test set
31 val_result = trainer.test(model, datamodule.val_dataloader(),
32     ↪ verbose=False)
33 test_result = trainer.test(model, datamodule, verbose=False)
34 result = {'test': test_result, 'val': val_result}
35 result

```

### A.1.11 Generate Callback

```

1 class GenerateCallback(Callback):
2     def __init__(self, input_imgs: torch.Tensor, every_n_epochs:
3         ↪ int = 1):
4         """
5         During the training, we want to keep track of the learning
6         ↪ progress by seeing reconstructions made by our model.
7         For this, we implement a callback object in PyTorch
8         ↪ Lightning which will add reconstructions every N
9         ↪ epochs to
10         our tensorboard
11
12         Args:
13             input_imgs: Images used to see the reconstructed
14             ↪ progress.

```

```
10         every_n_epochs: Interval for callback.
11         """
12         super(GenerateCallback, self).__init__()
13         self.input_imgs = input_imgs # Images to reconstruct
14         ↪ during training
15         # Only save those images every N epochs (otherwise the
16         ↪ tensorboard gets quite large)
17         self.every_n_epochs = every_n_epochs
18
19     def on_train_epoch_end(self, trainer: pl.Trainer, pl_module:
20     ↪ pl.LightningModule) -> None:
21         if trainer.current_epoch % self.every_n_epochs == 0:
22             # Reconstruct images
23             input_imgs = self.input_imgs.to(pl_module.device)
24             with torch.no_grad():
25                 pl_module.eval()
26                 reconst_imgs = pl_module(input_imgs)
27                 pl_module.train()
28             # Plot and add to tensorboard
29             imgs = torch.stack([input_imgs,
30             ↪ reconst_imgs["x_hat"]], dim=1).flatten(0, 1)
31             grid = torchvision.utils.make_grid(imgs, nrow=2,
32             ↪ normalize=True, value_range=(-1, 1))
33             trainer.logger.experiment.add_image('Reconstructions',
34             ↪ grid, global_step=trainer.global_step)
```

# Glossary

***IsGISAXS*** A simulation software for GISAXS scattering images.

***in situ*** Allows the data collection or manipulation of a sample without exposure to an external environment.

### Erklärung zur selbständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

_____	_____	
Ort	Datum	