Project report

Course: Project Course in Mathematical and Statistical Modelling
Project: Healing X-ray Scattering Images with the Deep Image Prior

Adnan Fazlinovic	adnanf@student.chalmers.se
Noa Onoszko	onoszko@student.chalmers.se
Ingrid Ingemarsson	ingridi@student.chalmers.se

Contact at Tetra Pak: Eskil Andreasson

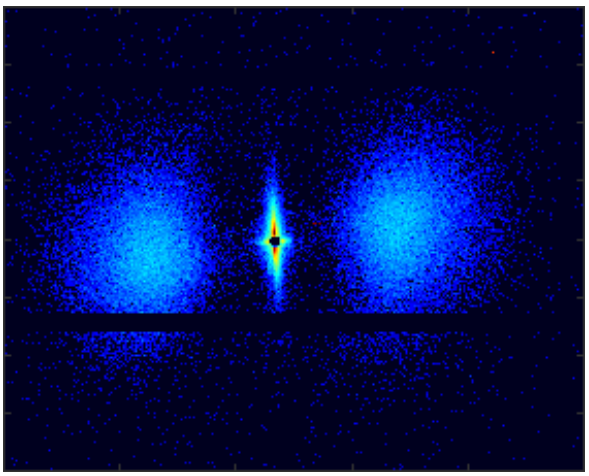


Figure 1: An X-ray scattering image with occlusions.

Abstract

Tetra Pak® uses X-ray Scattering images to study the structure and material orientation of semi-crystalline polymers used in polymer tops and opening devices. Physical limitations in the measuring equipment give rise to scattering-free regions in those obtained images. The aim of this project is to inpaint the regions in order to better visualise the scattering patterns in their entirety. The method used for this is the so called Deep Image Prior, which is a Neural Networks approach. The Deep Image Prior requires an occluded image and a mask that marks the missing pixels. This mask is created as a pre-processing step using intensity thresholds and other input provided by the user. A hyperparameter grid search is performed to acquire the best possible inpainting of the given occluded image, with promising results. However, the method is quite time consuming, which makes it appropriate only for inpainting a small number of images.

Contents

1 Introduction **3**

2 Theory **3**
2.1 X-ray Scattering 3
2.2 The Inpainting Problem 4
2.3 Neural Network Model 5

3 Method **8**
3.1 Mask Creation 8
3.2 Hyperparameter Tuning 9

4 Results **9**

5 Discussion **10**

6 Acknowledgments **13**

References **13**

A Source Code **13**

1 Introduction

Tetra Pak® has started using the experimental techniques Small and Wide Angle X-ray Scattering (SAXS and WAXS) at several different length scales in both laboratory and synchrotron X-ray sources. Scattering images from these experiments are used to study the structure and material orientation of semi-crystalline polymers used in polymer tops and opening devices. Physical limitations in the measuring equipment, like the use of beam stops, give rise to scattering-free sections in the resulting images which would not be present in the ideal case. The goal of this project is to inpaint the X-ray scattering images produced by Tetra Pak.

In 2017, Liu et al. [1] proposed a method for inpainting of X-ray scattering images. Their approach was mainly rule-based, using domain knowledge and symmetries in the images. There is literature on machine learning-based image inpainting but to our knowledge, the performance of this class of inpainting methods has not been evaluated on X-ray scattering images.

Machine learning models generally train on large amounts of data, and in return, have the ability to generalize. However, the data available for this project consists of only a handful of images which is far from sufficient to build a model capable of generalizing. In a paper from 2017 [2], the authors propose a method for inpainting and other image restoration tasks that they call the Deep Image Prior. It bridges the gap between learning-based methods, like convolutional neural networks, and learning-free methods, like the one proposed by Liu et al. The Deep Image Prior is a convolutional neural network model that takes only the occluded image as input and outputs an inpainted version of that image. Since it does not require large amounts of data, it is a suitable solution for our problem.

The reconstructed images will be used for presentation purposes later on. Further analyses cannot be made on the results because of the lack of knowledge regarding the ground truth of the X-ray scattering images. Since we inpaint the regions with missing data without prior knowledge of how the scattering behaves in these regions, we can't measure any type of difference between the output and the input.

2 Theory

2.1 X-ray Scattering

The processed images in this project originate from X-ray scattering experiments where a monochromatic focused X-ray beam is aimed at a sample whose electrons interact with the beam. The intensity of the scattered X-rays is recorded by a detector, and the captured diffraction pattern corresponds to the Fourier transform of the electron density distribution within the unit cell of a crystal in the sample. Both Small Angle X-ray Scattering (SAXS) and Wide Angle X-ray Scattering (WAXS) images are considered, which give information on different length scales about the structure of the sample [3].

The data from the experiments are effectively photon counts in each pixel collected by

a two-dimensional area detector. This detector is fabricated by joining together many pixel-array module building blocks into a larger patchwork of such modules covering the appropriate region. The nature of this construction gives rise to "defects" in the scattering images at the places of borders between modules, which are masked away in the images. Other untrustworthy segments of the images are the pixels corresponding to the beamstop and the beamstop holder, which are also masked away [1]. An example image can be seen in Figure 2.

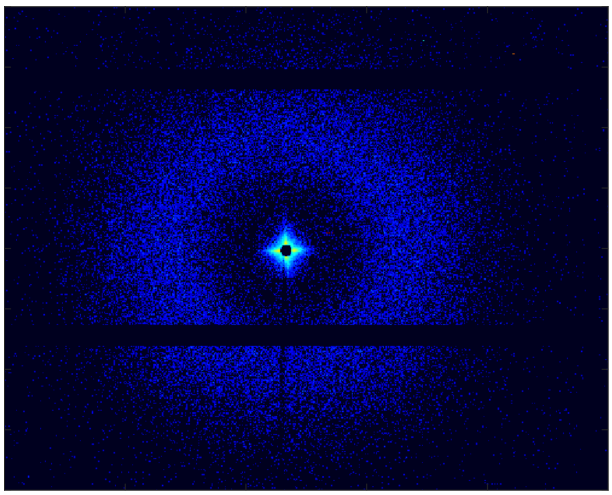


Figure 2: An example input image. There are two horizontal lines, a dot in the middle from the beamstop and a less visible vertical line that originates from the beamstop holder.

2.2 The Inpainting Problem

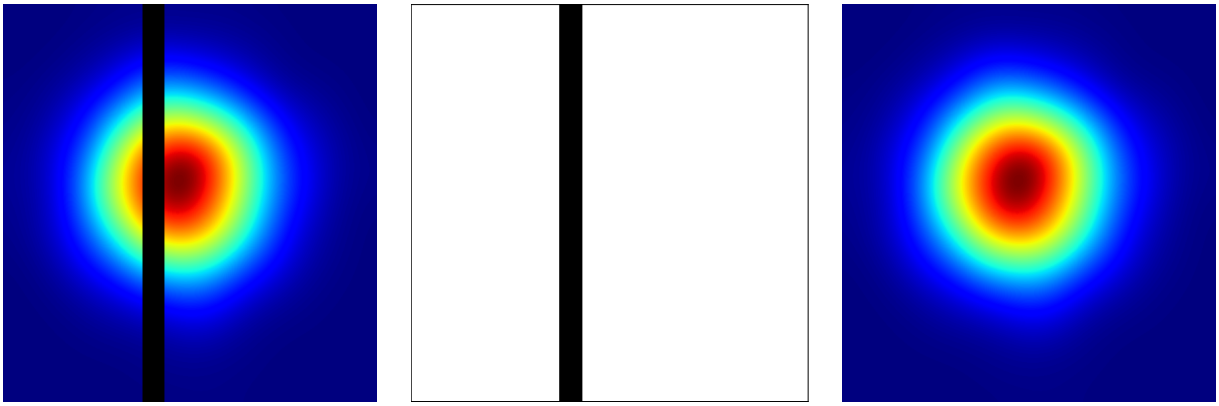
The image inpainting problem can be tackled by a variety of approaches ranging from manual or purely rule-based methods to machine learning algorithms. This project is based on a small sample of real scattering images, and thus requires an appropriate approach that does not assume a large training data set.

The inpainting problem can be stated in mathematical terms as the task of (re)constructing the image x from the occluded image x_0 with missing pixels corresponding to some binary mask $m \in \{0,1\}^{W \times H}$ where W, H are the width and height of the images and x, $x_0 \in \mathbb{R}^{3 \times W \times H}$. An example is shown in Figure 3.

The inpainting task can be expressed as an energy minimization problem,

$$x^* = \underset{x}{\operatorname{argmin}} \ \|(x - x_0) \odot m\|^2 + R(x), \tag{1}$$

where $R(x)$ is a regularizer, and $\odot$ means element-wise matrix multiplication. Multiplying the mask matrix m element-wise with an image x creates a masked image. Therefore, the first term in (1) is the distance between the masked image and the masked occluded image in the space $\mathbb{R}^{3 \times W \times H}$. An example of this can be seen in Figure 3: (c) is the image x, (b) is the mask m and (a) is the masked image $x \odot m$. The Deep Image Prior replaces



(a) x_0 (b) m (c) x

Figure 3: Illustration of the three different concerned image types; the occluded image x_0 with missing pixels, the mask m and the constructed image x.

the problem (1) by

$$\theta^* = \underset{\theta}{\operatorname{argmin}} \ \|(f_\theta(z) - x_0) \odot m\|^2, \tag{2}$$

where $x^* = f_{\theta^*}(z)$ is the final inpainted image and $f_\theta(z)$ is a neural network [2].

2.3 Neural Network Model

There is not one Deep Image Prior model but rather a few different convolutional neural network models tailored for different image restoration tasks. The network used for inpainting is an autoencoder, which is a network that starts by downsampling and then upsamples toward the later layers. For us, this means that the resulting feature maps of the convolutional layers first get smaller and smaller in the first layers, and then increase in size toward the end of the network. The downsampling is done naturally by the convolutional layers. Upsampling is achieved with nearest neighbor upsampling and by introducing padding to the input to the convolutional layers.

Figure 4 illustrates how a convolutional layer works. A matrix called kernel slides over the input image, taking the sum of the element-wise product of the kernel and the input submatrix at each position, resulting in a smaller output matrix. The kernel elements are weights that are trained. There can be several kernels per layer, resulting in one output each. These output matrices are called feature maps. The output of each convolutional layer is normalized with batch normalization, which is technique to improve learning by centering and scaling the data. Normalization is performed by applying the function

$$\text{BatchNorm2d}(z) = \frac{z - \text{E}[z]}{\sqrt{\text{Var}[z] + \epsilon}}\gamma + \beta \tag{3}$$

to a feature map z. After normalization, we apply the activation function

$$\text{LeakyReLU}(z) = \begin{cases} z, & z \geq 0, \\ \alpha z, & z < 0, \end{cases} \tag{4}$$

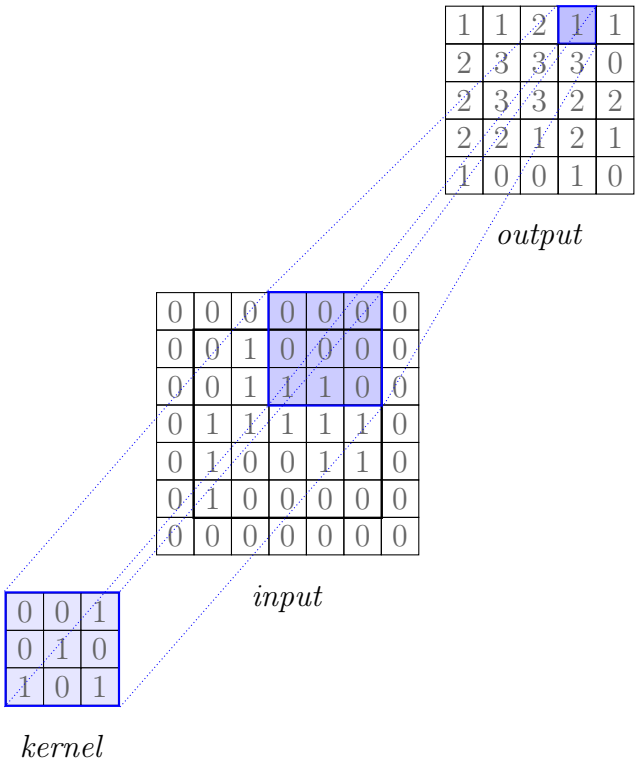


Figure 4: In a convolutional layer, a kernel slides over an input matrix which creates a new matrix. Here, the convolution downsamples the input image since there is no padding.

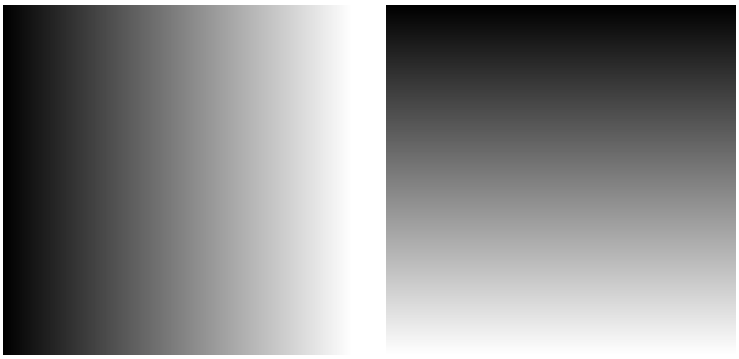
where α is a small, positive number. Finally, the Sigmoid activation function (5)

$$\text{Sigmoid}(z) = \frac{1}{1 + e^{-z}} \tag{5}$$

is applied to the output of the last layer.

The input $z \in \mathbb{R}^{2 \times W \times H}$ to the network is interestingly not the image to be inpainted. Instead, z is initialized by using two channels of different `numpy.meshgrid`:s, see Figure 5. Initializing every pixel z randomly according to a normal distribution is also possible. The meshgrid initialization however has the advantage that we start with some smoothness, as opposed to using a purely random input.
The model is trained in the following steps:

1. Initialize network weights and initial image z, e.g. noise or meshgrid.
2. Optional: Add normal noise to the initial image and/or weights.
3. Forward propagate through the network. The output is an image.
4. Mask the output.
5. Calculate MSE loss using the masked output and the target, $x_0 \odot m$.



(a) (b)

Figure 5: The two channels in the input z. The pixel values range from 0 to 1, where 1 is white and 0 is black.

6. Backpropagate and update the weights with Adam.

7. Repeat N times from step 2.

Training is another word for solving minimization problem (2). We are using the optimizer Adam, which is an extension to stochastic gradient descent that converges faster than the latter in many cases.

There are many details in the model. The most important design choices are summarized in Table 1.

Hyperparameter	value
number of layers	5
channels per layer	128
kernel size	3x3
bias in conv layers	yes
padding	reflection
activation	LeakyRELU
output layer activation	sigmoid
normalization	batch normalization
downsampling	convolution
upsampling	nearest neighbor
learning rate	tuned
input noise	tuned
weight noise	tuned

Table 1: Design choices for the model and for the training.

3 Method

3.1 Mask Creation

When creating the mask for the inpainting problem, the user itself must define the area of the missing pixels. Since these can vary from image to image, there's no straightforward implementation to solve this without introducing errors. The scatter plots themselves vary a lot since these are WAXS- and SAXS-measurements of different materials, which have different positioning and orientation. Trying to separate the missing pixels by brightness contrast (e.g. constructing a histogram of the gray-scale pixel values) can select low-intense pixels in the edges of the scatter plots, which are not to be masked. We only want to mask the missing pixels, which consists of a beamstop, beamstop holder and the horizontal stripes that correspond to the borders between modules.

When loading the scatter plots which will be masked, cropping is necessary since the images are figures with corresponding x- and y-labels and additional information describing the scatter plot. The cropping is done automatically to only utilize the scatter plot with the pixel values, where the resulting image is of dimensions 226×282 (pixels). These are then converted to gray-scale since we are interested in the intensity of each pixel. We normalize the values so that they range between 0 and 255, to get a clearer separation between the areas we need to mask and the areas that should be kept as they are.

An appropriate way to create the mask is to manually select the pixel areas by visual inspection. In this way, the user has better control over what area is to be selected. We define three separate masks, one for the beamstop, one for the beamstop holder and one for the horizontal stripes. Each mask is created as a matrix with the dimensions 226×282, the same as the scatter plots, with each entry being the value 1. The pixels that will be masked will be given the value 0 in its corresponding entry.

The first mask represents the masking of the beamstop. The beamstop is manually masked in the program, letting the user select a rectangle, by defining the upper left corner and the lower right corner, surrounding the beamstop, which then filters out the pixels with high-intensity values, keeping only the values 0 (corresponding to the masked area). The beamstop holder is also left for the user to select. It's quite well-presented in the scatter plots and straight forward to detect visually. However, because of its thin contours, it's hard to detect with standardized contour detection algorithms provided by e.g. openCV. These are handled by letting the user select a polygon that includes this area of missing data, which is then masked. There are also several horizontal strips of missing pixel values presented in the scatter plots. These are fairly easy to detect since they're rather wide and stretch over the whole scatter plot. To detect these, the mean value of the pixels are calculated over the x-axis of the scatter plot, resulting in a vector $\bar{\mathbf{X}} \in \mathbb{R}^H$, where H is the number of pixels in the y-axis. Since the stripes are aligned horizontally and quite wide, we will obtain elements in the mean-vector with the value 0. The indices of these elements represent the vertical position of the masking, and they stretch over the whole image.

When the three separate masks have been created, each of the same dimensions (number of pixels vertically × number of pixels horizontally), each entry is binary encoded, where a 1 represents pixels that shall not be masked and 0 represents pixels that shall be masked, we combine these three by performing element-wise multiplication over all three matrices. This yields a mask matrix that corresponds to all three masks, a 0 indicates a masked pixel despite the entry value in the other masks, and if the value for a specific entry is 1 in all three masks, then that pixel will not be masked. The resulting masks, one for each scatter plot, are then used as inputs for the inpainting algorithm used in the Deep Image Prior assessment.

3.2 Hyperparameter Tuning

To maximize the performance of the model, a small hyperparameter tuning was carried out. We identified three hyperparameters as particularly important, denoted in the code by `lr`, `param_noises` and `reg_noise_stds`. `lr` is the learning rate, which is a number between 0 and 1 that decides how much the weights will be updated at each step during training. The default value in the original implementation was 0.01, so we decided to try values larger and smaller than this. However, in applications, the learning rate is rarely larger than 0.1 so that is the largest value that we included in the search. `param_noises` is a boolean variable that decides if normally distributed noise should be added to the weights in each iteration. This is used to prevent overfitting. `reg_noise_stds` is the standard deviation of the additive noise that is added to the input image at every iteration. The latter two variables are previously referred to as weight noise and input noise in Table 1.

The optimization was carried out as a grid search, that is, every combination of each of the three hyperparameters was used once. Since the model does not generalize once trained, this was performed on multiple images.

Hyperparameter	Values tested
`lr`	{0.0001, 0.001, 0.01, 0.05, 0.1}
`param_noises`	{False, True}
`reg_noise_stds`	{0.3, 0.03, 0.003, 0}

Table 2: Hyperparameters used in the grid search.

The tuning was done by training one model for each of the 40 hyperparameter combinations for 6000 iterations, on all available data. Every 1500:th output image was saved.

4 Results

The hyperparameter tuning was carried out on a GeForce RTX 3090 GPU and took 10 hours, i.e. just below 2 minutes per image and hyperparameter combination. Out of thirteen images, we managed to successfully create image masks for eight of them. The five that were not correctly masked differ from the others either by having different colors

(not black and dark blue) or by having a scattering pattern that is more zoomed in or zoomed out. Hyperparameter tuning for the Deep Image Prior model was performed on the eight successfully masked images which gave us 40 inpainted images corresponding to the different parameters per masked image. The best images were identified by manually looking at all the output images. Since several input images were similar, four were selected to display here. The best resulting output image for each of those four input images are shown in Figure 6 and an example of bad masking and bad inpainting is shown in Figure 7. Figure 8 shows the evolution of the inpainting of the image in Figure 6(d) during training. More results can be found at our Github repository[1].

5 Discussion

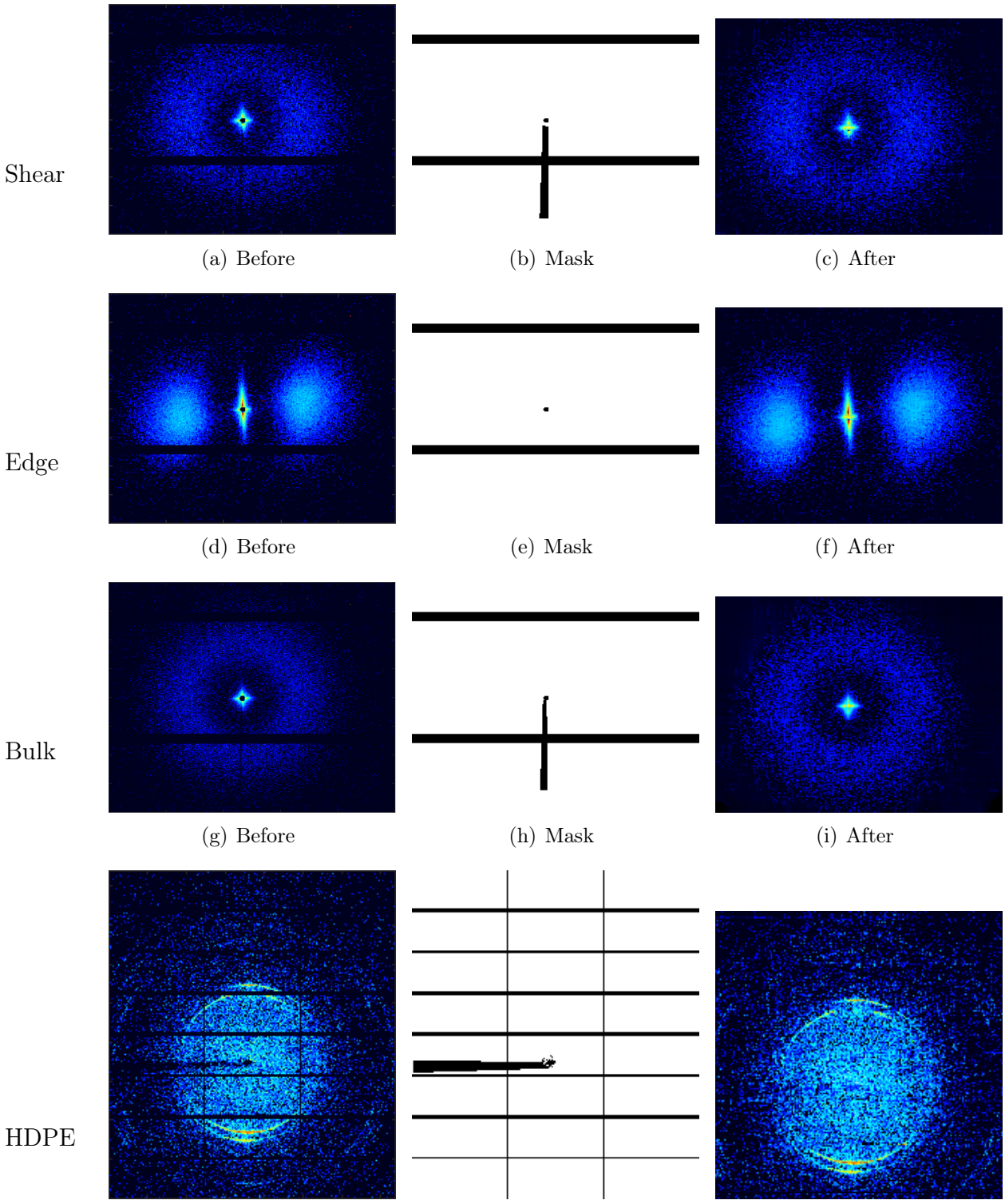
The reason for wanting to do the inpainting in the first place is the lack of perfect non-occluded images of the X-ray scattering due to the reasons stated in section 2.1. This has been a challenge during our project since this means we have no ground truth image to reconstruct and compare against. The question of quality of our result has thus been answered by use of manual visual inspection, which of course is not as exact as using some error metric. Setting this issue aside, let us inspect the result shown in Figure 6. In general, it seems as if the inpainting has worked quite well in these cases, effectively removing the dark lines and capturing the overall structure. However, some flaws will be discussed further.

In Figure 6(f) we can spot a dark region where the upper line used to be. Generally, it appears to be harder for the Deep Image Prior to inpaint lines in transition regions between high scattering and low scattering than in scatter rich environments like the lower line in Figure 6(c), which seems reasonably inpainted.

Of course, it should also be emphasized that the images in Figure 6 are the best case scenarios from the hyperparameter grid search, and hence we have to discuss the algorithm somewhat in light of its computational complexity. From the examination of the result, it looks as though learning rates 0.001 and 0.0001 yield badly inpainted results in most cases, probably because the algorithm does not converge in our maximum number of iterations. Otherwise, the best parameter setup was not the same for all input images, supporting the need of performing a grid search. Since this is such a time-consuming task, it is not feasible to use this solution to inpaint a large number of images. However, for a few images, this might not be an issue.

Another topic related to computation time is the number of iterations to use. Figure 8 shows the evolution of one of the inpainted images. Here, we see a clear improvement of the inpainting in the first three images, and interestingly a very smoothed image in 8(d). Of these four images, the best one is 8(c), but the optimal image can of course be a few iterations before or after that one since we did not save every output image. One way to improve the inpainting could be to run the Deep Image Prior once again with more plots to choose from. Alternatively, one could implement early stopping. This would

[1] `https://github.com/adnanfazlinovic/MVE385`



(a) Before (b) Mask (c) After

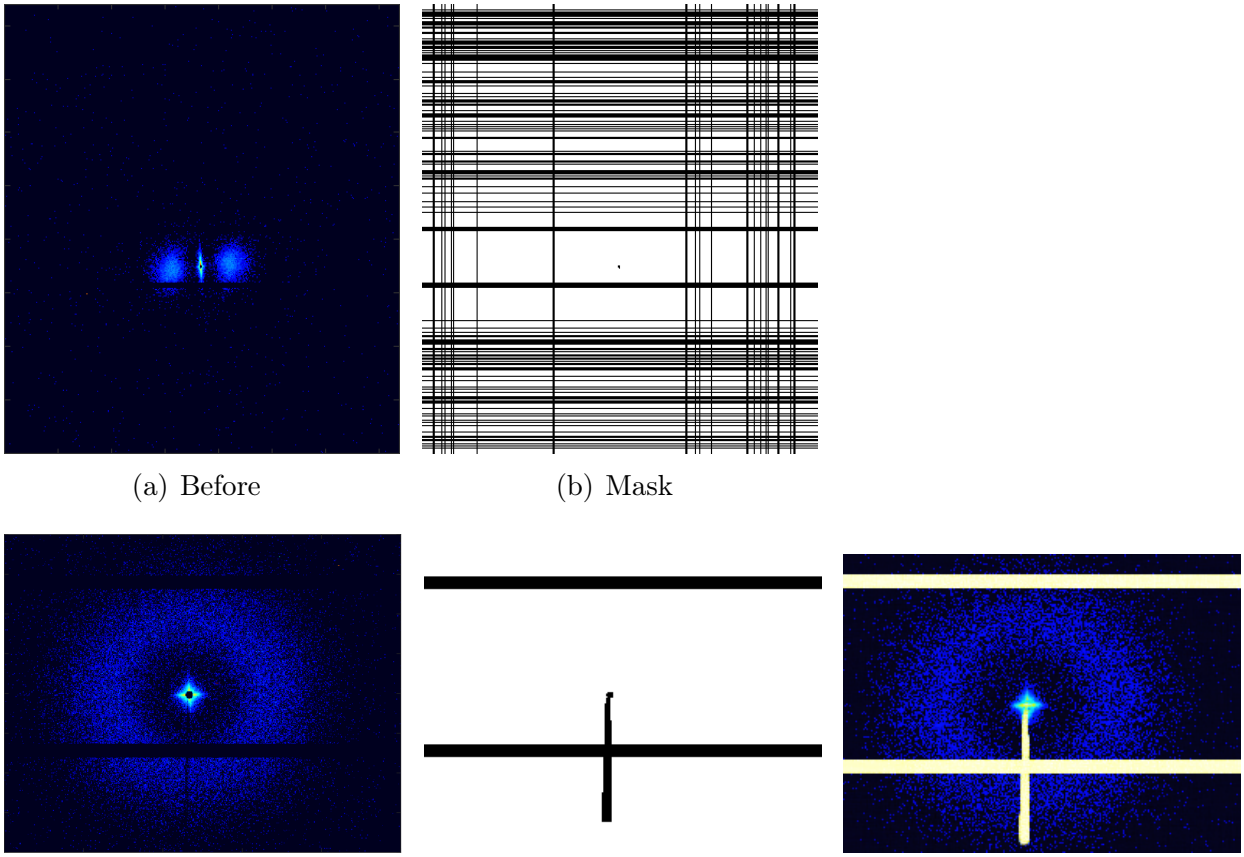
(d) Before (e) Mask (f) After

(g) Before (h) Mask (i) After

(j) Before (k) Mask (l) After

Figure 6: Original images (left), mask (middle) and best in-painted images from the hyperparameter grid search (right).

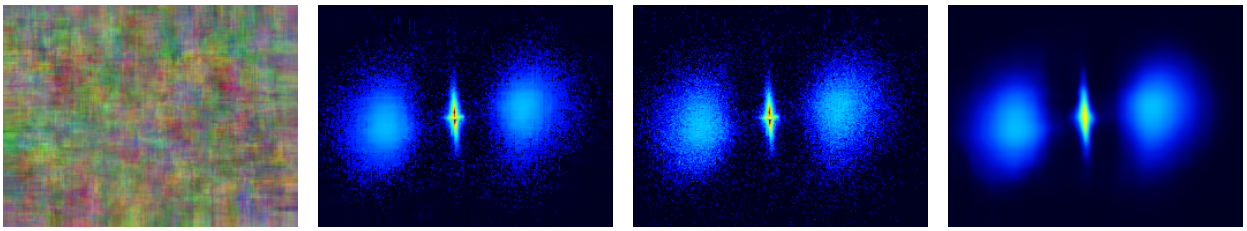
however require an evaluation metric, which could be tricky to find due to the absence of



(a) Before (b) Mask

(c) Before (d) Mask (e) After

Figure 7: Bad results from masking, top and inpainting, bottom.



(a) 0 iterations (b) 1500 iterations (c) 3000 iterations (d) 4500 iterations

Figure 8: Evolution of image: after 0, 1500, 3000 and 4500 iterations.

ground-truth images.

Some possible improvements could be carried out to improve the results. Automatic masking could help mask the images that the current masking technique failed on, and would make user input unnecessary. Also, the loading of the images into Python could be reviewed, to preserve image quality since this currently results in a considerable loss of quality. Lastly, it could be interesting to try overlaying the original images to ensure that the non-inpainted parts of the result are the same as in the original, although a disadvantage with this could be less smoothness in the transition.

6 Acknowledgments

First, we would like to thank Linnea Björn at Chalmers University of Technology for the data that was used, and whose research in X-ray scattering gave rise to this project. We would also like to thank our supervisor Eskil Andreasson at Tetra Pak for great supervision and a nice collaboration.

We would like to acknowledge that we have used the PyTorch implementation of the Deep Image Prior written by Dmitry Ulyanov, Andrea Vedaldi and Victor Lempitsky. Nothing in the model was changed except for the hyperparameters used in the tuning. Some surrounding code was changed and added for convenience.

Lastly, we would like to thank Joel Lidin for running the hyperparameter tuning on his GPU.

References

[1] Jiliang Liu, Julien Lhermitte, Ye Tian, Zheng Zhang, Dantong Yu, and Kevin G. Yager. Healing X-ray scattering images. *IUCrJ*, 4(4):455–465, Jul 2017.

[2] V. Lempitsky, A. Vedaldi, and D. Ulyanov. Deep image prior. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 9446–9454, 2018.

[3] Linnea Björn. Characterisation of Injection Moulded Polymer Materials using SAXS and WAXS. Master's thesis, Chalmers University of Technology / Department of Physics (Chalmers), 2018.

A Source Code

The source code for this project and for the original implementation of the Deep Image Prior can be found here[2] and here[3] respectively.

[2] `https://github.com/adnanfazlinovic/MVE385`
[3] `https://github.com/DmitryUlyanov/deep-image-prior`