



Stockholms
universitet

A Study of Generative Adversarial Networks with Applications to Paperboard Surfaces

Karin Pagels

Masteruppsats 2023:6
Matematisk statistik
Juni 2023

www.math.su.se

Matematisk statistik
Matematiska institutionen
Stockholms universitet
106 91 Stockholm

A Study of Generative Adversarial Networks with Applications to Paperboard Surfaces

Karin Pagels*

June 2023

Abstract

Generative adversarial networks, or GANs, are a type of unsupervised learning method that are known for being able to generate high-quality data. They can create a completely new image, a fake image, that looks just like a real one. Typically, to consider a generated image as a good one, one just lets people look at it. In some cases, this will not be enough. Specifically, one needs to make sure that the generated image has the same statistical properties as real images. In this thesis, images of paperboard surfaces are generated and evaluated considering their statistical properties. Evaluation of the generated images is performed in terms of multidimensional scaling (MDS) using Wasserstein distance and histograms to examine the distributions between images and between groups of images, as well as autocorrelation to consider vertical and horizontal correlations in images. Results with statistical properties similar to the real images were generated. The thesis was conducted together with the food processing and packaging company Tetra Pak.

*Postal address: Mathematical Statistics, Stockholm University, SE-106 91, Sweden.
E-mail: karinpagels@gmail.com. Supervisor: Chun-Biu Li.

Acknowledgement

I would like to express my deepest appreciation to my supervisor Chun-Biu Li, who generously provided expertise and knowledge. I am thankful for his valuable feedback and patience.

I give my sincere gratitude to my supervisors at Tetra Pak; Magnus Arnér and Erik Bergvall. I am grateful for their guidance and patience, as well as their great feedback. Special thanks to Magnus Arnér for all of the interesting discussions during the course of the thesis.

I am thankful to Tetra Pak for giving me the opportunity to carry out this thesis and for providing excellent data.

Abstract

Generative adversarial networks, or GANs, are a type of unsupervised learning method that are known for being able to generate high-quality data. They can create a completely new image, a fake image, that looks just like a real one. Typically, to consider a generated image as a good one, one just lets people look at it. In some cases, this will not be enough. Specifically, one needs to make sure that the generated image has the same statistical properties as real images. In this thesis, images of paperboard surfaces are generated and evaluated considering their statistical properties. Evaluation of the generated images is performed in terms of multidimensional scaling (MDS) using Wasserstein distance and histograms to examine the distributions between images and between groups of images, as well as autocorrelation to consider vertical and horizontal correlations in images. Results with statistical properties similar to the real images were generated. The thesis was conducted together with the food processing and packaging company Tetra Pak.

Contents

1	Introduction	2
2	Theoretical Background	4
2.1	Neural Networks	4
2.2	Convolutional Neural Networks	7
2.3	Deep Generative Models	9
2.4	Generative Adversarial Networks	10
3	Case study: Generating Images of Paperboard Surfaces	12
3.1	Image Data	13
3.2	Adjusting the Data	14
3.3	Generative Adversarial Networks on Data	17
4	Results and Evaluation of Them	23
4.1	Results: Full Images	23
4.2	Results: Partial Images	24
4.3	Methods for Evaluation	26
4.4	Evaluation of Results	27
5	Discussion	33
6	Conclusion	35
6.1	Further Studies	35
	Bibliography	37
A	Figures	39

Chapter 1

Introduction

Deep generative models can be explained as a mixture of classical generative models and neural networks. The goal is to, by using machine learning, generate new data. The models can get very good and it can in some cases be completely impossible to distinguish a real sample from a generated, fake, sample.

Generative adversarial networks (GANs) were first mentioned in 2014 by Ian Goodfellow with colleagues in the proceedings of the international conference on Neural Information Processing Systems, or NIPS [1]. GANs are a type of deep generative model that was special in the sense that they used the power of discriminative models. This was new for deep generative models at the time [2].

The basis of GAN is that a generative model is good if we cannot tell which data is generated and which is real [2]. In some cases, not being able to visually distinguish generated data from real data is not enough. We want to explore the statistical properties of the data to make sure that the real and generated images have the same ones.

The thesis was executed in incorporation with the food processing and packaging company Tetra Pak. As an application of GAN, we considered a dataset containing paperboard surface topography, which can be viewed as images. The goal was to generate new images of paperboard surfaces that had similar statistical properties as the real images.

Many properties of packaging material depend on the surface topography of the paperboard. Paperboard surfaces are used as inputs to simulation methods, for example, finite element analysis. For the simulation to properly reflect real-world variation, a large data set is needed. Measuring the surface topography of paperboard is demanding, and collecting a large enough data

set is too costly to be practically possible. For this reason, new images are generated. Generated images with the same statistical properties as the real images can then be used as inputs to the simulation methods.

It was decided to use GAN to generate the paperboard surface images. The reason for this was that GAN has been shown to be able to generate high-quality images, and can be better than other deep generative models such as variational autoencoders [3]. In the case of generating paperboard surfaces, it is important that the generated images are of high quality. This is because they should not only look like the real images but also have similar statistical properties as the real ones.

Despite being able to generate images with high quality, GAN is also known to be difficult to train [3]. The reason for this is that GAN contains two neural networks, which introduces a saddle point problem.

The expected results of the thesis are to demonstrate that GAN can generate images that have similar statistical properties as the real images, but there could be difficulties in the training. The quality of the output images depends on different factors such as the choices of hyper-parameters, and the generated images could deviate from the real images in some senses. It is expected that GAN will be difficult to train, but with a possibility for high-quality results [3].

The thesis is structured so that theoretical concepts of neural networks and deep generative models are first explained in Chapter 2. After that, in Chapter 3, the case study of generating images of paperboard surfaces is described. Here, the data is described as well as the used methods for adjustments of the data. In Chapter 4, the results are presented and evaluated according to the methods described in the same chapter. In Chapter 5, the results of the case study are discussed, and in Chapter 6 conclusions are drawn together with possible improvements to the study.

All analysis is done in the programming language Python. Any pictures included in this thesis that are a sample of the entire data set were chosen by using a random number generator.

Chapter 2

Theoretical Background

In this chapter, the concept of deep generative models and generative adversarial networks is explained. Before getting into generative models, basic concepts of neural networks are first reviewed.

2.1 Neural Networks

A neural network is a machine learning model designed to mimic the human brain in some senses. Just like the brain, a neural network consists of nodes that are connected to each other. In a neural network, the nodes are placed in layers where each layer is connected to the next layer.

In a neural network, the first layer is always an input layer. This is where raw data enters the model. The last layer is the output layer, where the final product exits the network. The layers in between are called hidden layers. These can be of various types, and have the purpose of transforming the data to learn patterns from input data. In Figure 2.1, a simple neural network can be seen. It has two hidden layers containing four and three nodes, respectively. The input layer has three nodes while the output layer has two. The number of nodes in a layer is fixed. [4]

The layers in a neural network can be seen as functions with weights and biases. These are not fixed and for the neural network to learn, these will be adjusted. This is how a neural network is trained to predict or classify what is desired. To get a neural network to perform desired tasks, it needs to be trained. The goal is to get the weights and biases to take values so that a loss function is minimized.

The usage of neural networks in machine learning is called deep learning. Deep learning can be either supervised or unsupervised. In supervised learn-

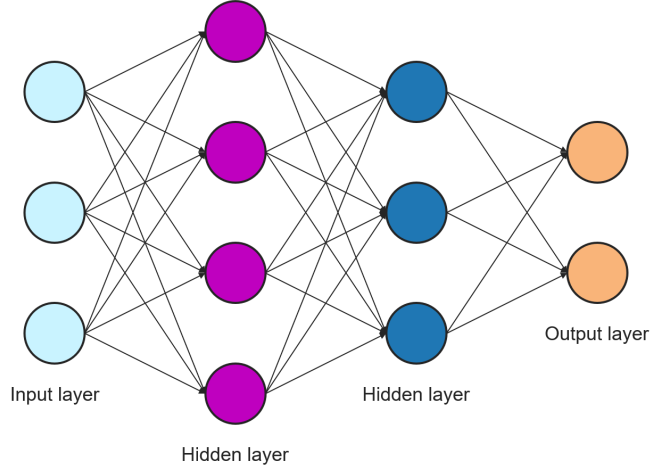


Figure 2.1: A simple neural network with two hidden layers.

ing, we want to solve a task where we have labeled data to use for the training of the neural network while there are no such labels in unsupervised learning. Neural networks can solve various tasks, where two fundamental tasks are regression and classification. In this thesis, the focus was on classification tasks.

In a neural network, activation functions are used. These functions are contained in the nodes of the neural network. The activation function defines the output of the node given its input. For classification tasks, a standard choice of activation functions is to use the **ReLU** (rectified linear unit) activation function for the hidden layers and the **sigmoid** activation function for the output layer. The ReLU activation function can be seen as the default activation function in hidden layers. It only outputs positive values and is nearly linear, so it preserves many properties that can make linear models easy to optimize [4]. The sigmoid activation function is an appropriate choice for the output layer because it outputs a value between zero and one. This value corresponds to the probability of belonging to the class it was characterized into.

The ReLU activation function is defined as

$$f(x) = \max(0, x) = \begin{cases} 0, & \text{if } x < 0, \\ x, & \text{otherwise,} \end{cases} \quad (2.1)$$

while the sigmoid activation function is defined as

$$f(x) = \frac{1}{1 + e^{-x}}. \quad (2.2)$$

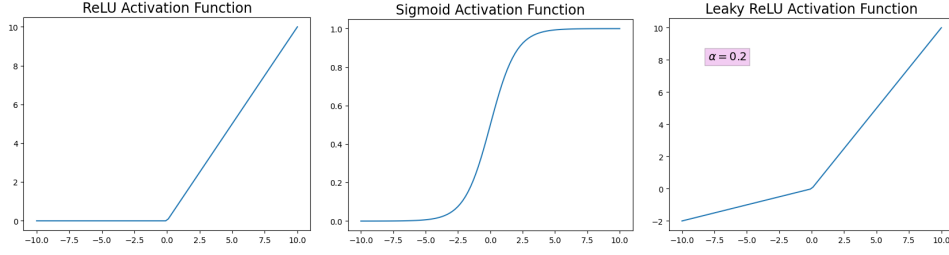


Figure 2.2: ReLU, sigmoid and leaky ReLU activation functions.

A generalization of ReLU activation function is called **leaky ReLU** activation function and is defined as

$$f(x) = \max(\alpha x, x) = \begin{cases} \alpha x, & \text{if } x < 0, \\ x, & \text{otherwise,} \end{cases} \quad (2.3)$$

where α is a value in the interval $[0, 1]$ [2]. Note that if $\alpha = 0$, leaky ReLU and ReLU are the same activation functions. We can say that ReLU is a special case of leaky ReLU. In Figure 2.2, graphs showing the ReLU activation function, the sigmoid activation function as well as the leaky ReLU activation function can be seen.

When training, the goal is to minimize a function [4]. This function is often called the loss function, but could also be called the cost function or the error function. This function is minimized using a specific algorithm, often referred to as an optimization algorithm.

When minimizing or maximizing a simple function, we typically compute the derivative of that function and find the point(s) in the parameter space where the derivative equals zero. The algorithm **gradient descent** uses this, by moving, in small steps, in the direction suggested by the derivative [4]. If the derivative is negative, the function is moved slightly in a positive direction while if the derivative is positive, the function is moved in a negative direction. After a certain number of iterations, the function will be at a local minimum. Note that the minimum found might not be the global minimum. The converging point could also be a saddle point. Typically, it is enough to find a converging point that is not too far from the true global minimum.

In machine learning, the gradient of the entire data set is typically not calculated because of the computational cost. This is where **stochastic gradient descent**, or SGD, comes in. Here, a stochastic approximation of the gradient is used instead of the actual gradient of the entire data set. For each step, a randomly selected **mini-batch** is used to compute the gradient. A

mini-batch is just a subset of the entire data set with a size selected as a hyper-parameter. One **epoch** is completed when all data have been used for a step in the algorithm. Each data point is typically used once per epoch.

An important hyper-parameter to specify when using SGD is the **learning rate**. The learning rate is the step size that the algorithm takes. In practice, the initial learning rate is what is specified and this is gradually decreased over the course of the algorithm. SGD can be slow and because of this, momentum is sometimes used. The momentum algorithm uses past gradients by taking the exponentially decaying moving average of them and continuing to move in their direction [4].

The learning rate can be a difficult hyper-parameter to set. Using an adaptive learning rate can help to steer clear of this problem. With an adaptive learning rate, different learning rate for each parameter of the model is used, and these are adapted throughout the training. **Adam**, with its name derived from adaptive moment estimation, is an adaptive learning rate optimization algorithm that uses momentum [4]. Adam uses both first-order moments and second-order moments of the gradient where two new hyper-parameters are introduced, β_1 and β_2 . The hyper-parameters β_1 and β_2 can be seen as exponential forgetting factors, where β_1 is the forgetting factor for past gradients and β_2 is the forgetting factor for past second moments of gradients. Adam is regarded as being quite robust to the choice of hyper-parameters and is therefore often used in deep learning. A detailed description of the Adam optimization method is out of the scope of this thesis and the interested reader is referred to the publication on Adam by D. P. Kingma and J. Lei Ba [5].

2.2 Convolutional Neural Networks

Convolutional neural networks, or CNNs, are a type of neural network that typically takes images as input data. Unlike a fully connected neural network, a CNN uses convolutions in at least one layer [4]. Convolution is defined as

$$s(t) = (x * w)(t) = \int x(a)w(t - a)da. \quad (2.4)$$

The definition gives the convolution between x and w . Here, x can be seen as an input and w as a weighting function while s is the output. In a CNN, the convolution can be found in the so-called convolutional layer. In Figure 2.3, a simple example of how the convolution in a layer works can be seen. The matrix to the left represents the input to the layer, and the matrix to

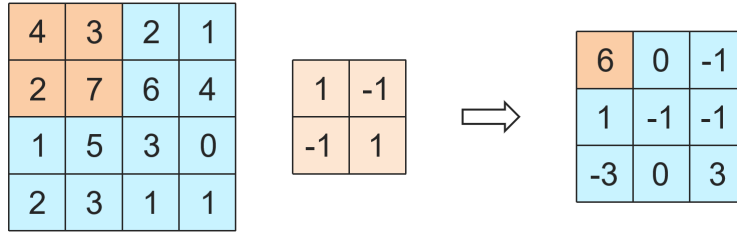


Figure 2.3: The occurrences in a convolutional layer in a CNN. The middle matrix, or kernel, is placed on top of the input (left) to create the output (right) by using convolution.

the right is the output of the layer. The kernel, the middle matrix consisting of 1 and -1, is applied to the input. It can be imagined that the kernel is placed on top of the input matrix and the convolution between the two is computed. The values in the same positions are multiplied, and everything inside the kernel is added together. What is computed is the output of the layer. Comparing this to Equation 2.4, the kernel corresponds to w , the input to x , and the output to s .

In some cases, we want a type of opposite convolution. For this, deconvolution is used. Deconvolution is not the inverse of convolution, but more of a transpose of convolution. Here, the output will be larger than the input. It can be said that deconvolution broadcasts the input elements with the kernel, instead of reducing them as regular convolution does. [2]

An example of what happens in a deconvolutional layer can be seen in Figure 2.4. The orange value in the input is multiplied by each value in the kernel. The resulting 4 by 4 matrix is placed in the top left corner of the output matrix. Though, these are not the final values for those positions in the output. Each value in the input is multiplied by the kernel and the result is placed in the corresponding position in the output. This means that values will overlap in all positions, except on the corner values. The overlapping values will be added together. As an example, in the second value from the top to the left, there are two overlapping values: -6 and 1 . When summing these we get the final value of -5 .

With convolution or deconvolution, something called **stride** can be added. This means that when applying the kernel to the input, for both convolution and deconvolution, rows and columns are slipped. As an example, with a stride of two, every other row and column is skipped. This will give an even smaller output in the convolution case and a larger output in the deconvolution case. Another tool that can be added to the convolution or

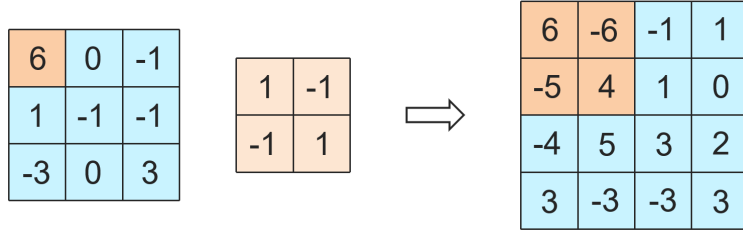


Figure 2.4: The occurrences in a deconvolutional layer. The middle matrix, or kernel, is placed on top of the input (left) to create the output (right) by using deconvolution.

deconvolution is **zero-padding**. Zero-padding means that, before applying the kernel, zeros are added around the input as a frame. If zero-padding with value one is added, then the added frame has a width of one. With zero-padding, the size of the output can be adjusted depending on the value of zero-padding. [2]

2.3 Deep Generative Models

The purpose of a deep generative model is just as the name implies, to generate. Specifically, its goal is to learn a representation of an intractable probability distribution where generated samples can be drawn from [3]. What will be generated can be of several types, but it is common to generate images. Commonly, a set of images are used for training the generative model. After the training, the model will then, if trained correctly, be able to generate images that are indistinguishable from those in the training set of images.

The way a deep generative model learns to generate is by learning a probability distribution. From this distribution, generated samples can be drawn. These generated samples are different from those in the training set of data but look, in case of successful training, indistinguishable from them. Deep generative models assume independent and identically distributed samples [3].

We define the probability distribution of the input data as $p_{\mathcal{X}}(x)$. The goal of the generator is to map points to a distribution that is as similar to $p_{\mathcal{X}}(x)$ as possible [3]. The generator is defined as

$$g_{\theta} : \mathbb{R}^q \rightarrow \mathbb{R}^n \quad (2.5)$$

where θ are the parameters of the generator, q is the dimension of what we

call a latent space and n is the dimension of the output space. Note that the input data is defined on a space with dimension n . The points are mapped from a known distribution, which we call $p_{\mathcal{Z}}(z)$. We denote a point from $p_{\mathcal{X}}(x)$ by x and a point from $p_{\mathcal{Z}}(z)$ by z . The mapping will give us the distribution $p_{g_{\theta}}(z)$ and we aim for this to be as close to $p_{\mathcal{X}}(x)$ as possible. We want that for every point x in $p_{\mathcal{X}}(x)$ that there should exist a point z in $p_{\mathcal{Z}}(z)$ such that $g_{\theta}(z) \approx x$.

The points of the distribution $p_{\mathcal{Z}}(z)$ exist in the latent space. This space has to be defined and the distribution $p_{\mathcal{Z}}(z)$ is chosen before training. A typical example of how it can be set is as a univariate Gaussian.

2.4 Generative Adversarial Networks

A generative adversarial network, often referred to as a GAN, is a type of deep generative model. There are several other types of deep generative models such as variational autoencoders and normalizing flows. GANs are chosen for this thesis because they can generate with high- quality and can, if trained correctly, be better than other types of deep generative models [3].

A GAN consists of two neural networks. The first neural network is the generator, which is the one that will generate the results and is the function presented in Equation 2.5. The second one is the discriminator, which works as a neural network for classification. It will take the real data and the generated data, and try to classify which is which.

Figure 2.5 presents a schematic picture of what happens in a GAN. A vector is drawn from the latent space and inserted into the generator. In the generator, a generated image is produced. Both generated and real images go into the discriminator, to classify whether they are fake or real.

The learning in GAN is often described as a tug-of-war between the generator and the discriminator. If the generator performs well, the discriminator struggles to classify the real and the generated data correctly. If the discriminator performs well, the generator is not generating very convincing data.

In GAN, the weights of θ are trained by minimizing a loss function [3]. The loss function measures the distance between $p_{g_{\theta}}(z)$ and $p_{\mathcal{X}}(x)$. This loss function needs to be chosen before training. This is an important choice and can make a great difference in the training. When training a GAN, no likelihoods are used which is common in other types of deep generative models.

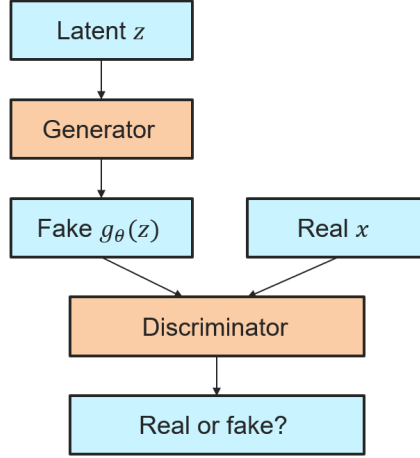


Figure 2.5: The process of a GAN. A vector z is drawn from the latent space. It goes into the generator to generate a sample $g_\theta(z)$. The sample goes into the discriminator to classify whether it is generated (fake) or real. Real samples x also go into the discriminator for classification.

There are several ways to define the discriminator [3]. A quite standard way to define the discriminator is based on binary classification and the discriminator is defined as

$$d_\phi : \mathbb{R}^n \rightarrow [0, 1]. \quad (2.6)$$

This is what is used in this thesis. For real data, $d_\phi(x) \approx 1$, and for generated data, $d_\phi(x) \approx 0$. Typically, a binary cross-entropy loss function is used, which is defined as

$$J(\theta, \phi) = \mathbb{E}_{\mathcal{X}}[\log(d_\phi(x))] + \mathbb{E}_{\mathcal{Z}}[\log(1 - d_\phi(g_\theta(z)))] \quad (2.7)$$

where $\mathbb{E}[\cdot]$ is the expected value. The goal for the discriminator is to minimize $-J(\theta, \phi)$ while the generator has the goal to minimize $J(\theta, \phi)$ [3]. Recall that the goal when training a neural network is to minimize a loss function with an optimization algorithm. In GAN there are two neural networks and two loss functions that are minimized.

The goals of the two neural networks are conflicting and training them gives a saddle point problem. This is a reason why GANs can be quite difficult to train. If either the discriminator or the generator is performing too well, the other one will not be able to improve. In practice, this can mean that all generated images will look the same as each other early in the training curve and will not improve over the epochs.

Chapter 3

Case study: Generating Images of Paperboard Surfaces

For the case study, images of paperboard surfaces were generated. An example of what the real paperboard surface images in the training set look like can be seen in Figure 3.1. The goal was to generate images that both visually looked like these but also possessed similar statistical properties. The reason for generating paperboard surface images was that they can be used as inputs to simulation methods. Quite a lot of data is needed for the simulation methods, and it is too costly to measure the surface topography of paperboard for that large of a data set.

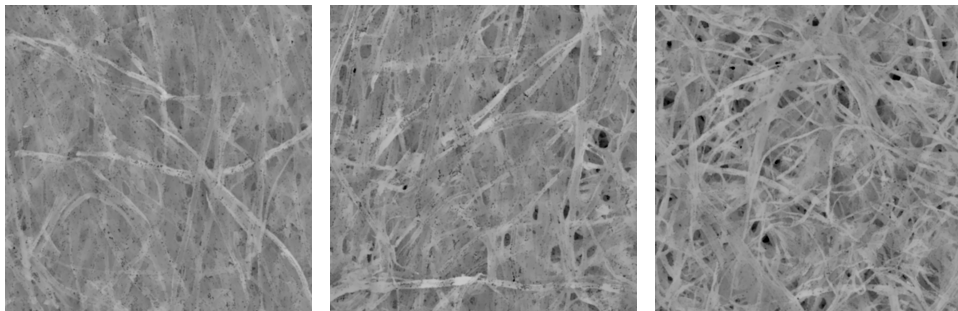


Figure 3.1: Example data images of the paperboard surfaces. Note that these images have been adjusted, see Section 4.2 for details.

3.1 Image Data

The data used in this thesis is images of paperboard surfaces, where we can see the fiber structures in the images. In total, the data set consisted of 110 image data frames, though due to outliers 108 were used. A data frame consisted of three columns where the first two described the location of the pixel and the last one the pixel's intensity. The data frame can be seen as a matrix due to locations being relatively evenly distributed. The matrix can then be considered as an image with a single scale for the color (for example a black-and-white image only has one numerical scale for the shades in it).

The images in the data set were actually height values, that have been measured on top of the paperboard surface. They were measured in a square with a side of 1.56 mm. Every row had 1000 values measured evenly spaced on the row, though some values were missing. There were 1000 rows, meaning that in total 1 000 000 height values were measured, and this was only for one image.

The images are anisotropic, meaning they are not the same in the horizontal and vertical directions. The fibers lay a bit differently depending on the direction. We call the directions machine direction, or MD, and cross direction, or CD. All images in the thesis have a vertical MD and a horizontal CD.

Two images of the data set contained a spot where the pixels had much larger height values than the rest of the image. These were considered to be outliers because it was concluded that the spots most likely were a product of a measurement error. These images were discarded from the data set.

The original images had a size of 1000 by 1000 pixels, which is a quite large image size for deep learning. It was decided to split each image into 250-by-250-pixel images instead. There were two reasons for this; firstly, that there would be more data to train on. Each 1000-by-1000-pixel image was divided up into $4^2 = 16$ new images. This meant that 108 images were divided up resulting in a larger data set containing 1728 images ($108 \cdot 4^2 = 1728$).

Secondly, enough information about the fiber structure was contained by only viewing 250 pixels ahead. To see this we consider the autocorrelation of the images in each direction. We view each row and each column as separate time series and calculate their autocorrelation. The CD autocorrelation is the mean value of all autocorrelations calculated from the row-wise time series, and the MD autocorrelation is the mean value of all autocorrelations calculated from the column-wise time series. An example of how these kinds of autocorrelation can look can be seen in Figure 3.2. It can be seen that both the CD autocorrelation and the MD autocorrelation decrease quite

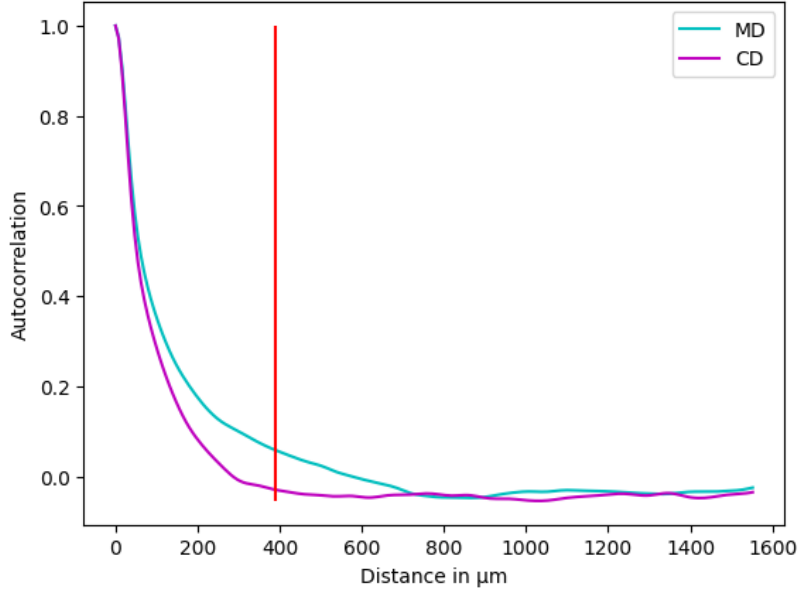


Figure 3.2: Autocorrelation of a data image. Each row and column in the image matrix is viewed as a time series. The CD curve is the mean value of all row-wise autocorrelations, while the MD curve is the mean value of all column-wise autocorrelation. It can be seen that there is not much correlation after the red line at 390 μm corresponding to 250 pixels.

fast. A red line is drawn in Figure 3.2 at 390 μm , which corresponds to 250 pixels. It can be seen that after the line, there is not much correlation for either CD or MD. Here, 200 lags are used. This means that the lag length is 5 pixels or 7.8 μm .

3.2 Adjusting the Data

Before the GANs could be implemented, the data had to be adjusted in a few senses. The images in the data set contained missing values which had to be filled in. The images also had to be untilted because some images were slightly tilted or possibly bent, which is not beneficial for the simulation methods. It was decided to also apply a Gaussian kernel to the image by using convolution. This removed irrelevant unevenness in the images that were not of interest for the simulations.

Firstly, the missing values in the data set had to be managed. The method for filling in missing values was put forward by E. Bergvall at Tetra Pak. It involved using a Gaussian kernel as well as convolutions to find values for

the missing points. We call the data image matrix I , and the value that is in row m and column n we call i_{mn} .

We have a Gaussian kernel G , which can be described as a two-dimensional Gaussian function. The size of the Gaussian kernel was chosen to be 24 by 24 pixels with a standard deviation of 7 pixels. A zero-one matrix C is defined, which specifies whether the entry c_{mn} is missing or not. The entry c_{mn} in C is defined by

$$c_{mn} = \begin{cases} 1, & \text{if } i_{mn} \text{ is not missing,} \\ 0, & \text{if } i_{mn} \text{ is missing.} \end{cases} \quad (3.1)$$

The matrix J is calculated as

$$J = \frac{(I \cdot C) * G}{C * G} \quad (3.2)$$

where \cdot is element-wise multiplication and \div element-wise division. The operator $*$ is convolution as defined in Equation 2.4. This matrix can be seen as a slightly more blurred version of I , but in contrast to I it does not contain any missing values. The matrix F_{fill} is obtained by calculating its values f_{mn} as

$$f_{mn} = \begin{cases} i_{mn}, & \text{if } c_{mn} = 1, \\ j_{mn}, & \text{if } c_{mn} = 0. \end{cases} \quad (3.3)$$

Here, j_{mn} are the values in the matrix J . This means that if the value in the image matrix I is not missing, then the matrix F_{fill} will take the same value as I . Only the values that are missing in I are changed, and the rest stay the same. The missing values are changed to values in J , the slightly blurred version of I . Because J look so similar to I , the filling in of the values matches very well with the image.

In Figure 3.3, the Gaussian kernel used can be seen as well as the matrix J and the matrix F_{fill} . It can be seen that J and F_{fill} look very similar, but that J is blurrier than F_{fill} .

When the height values for the images were measured, the images did in some cases lay slightly tilted. This means that one side of the image may appear closer than the other side of the image. The paperboard might also have been slightly bent when being measured. These qualities are undesirable and were settled by using a method for untilting.

The position of each measured height value is included in the data set. The positions are aimed to be spaced out evenly, but might not be in every case. We have a matrix X defined as

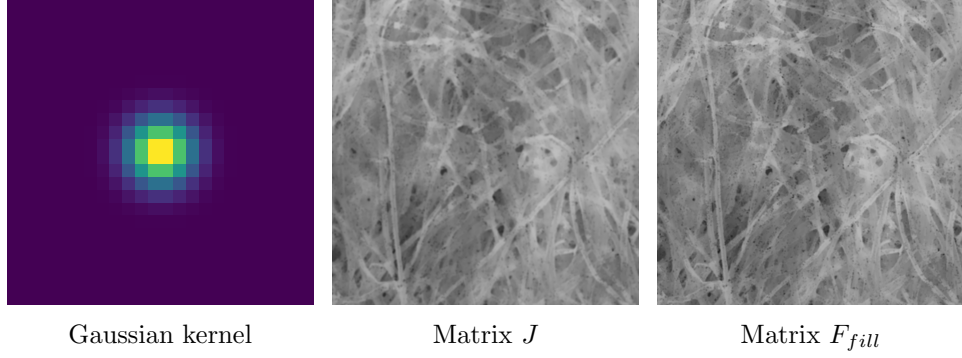


Figure 3.3: The Gaussian kernel, the matrix J which is a blurred version of the initial data matrix I and the matrix F_{fill} where missing values have been filled in.

$$X = \begin{bmatrix} 1 & cd_1 & md_1 \\ 1 & cd_2 & md_1 \\ 1 & cd_3 & md_1 \\ \vdots & \vdots & \vdots \\ 1 & cd_{n-1} & md_n \\ 1 & cd_n & md_n \end{bmatrix} \quad (3.4)$$

where cd_k is the position in CD and md_k the position in MD. The matrix F_{fill} is reshaped to a column vector

$$F_{reshape} = \begin{bmatrix} f_{11} \\ f_{12} \\ f_{13} \\ \vdots \\ f_{nn} \end{bmatrix}. \quad (3.5)$$

The vector θ , which is a vector with three values where each one corresponds to a column in X , is calculated as

$$\theta = (X^T X)^{-1} (X^T F_{reshape}) \quad (3.6)$$

where X^T is the transpose of X , and X^{-1} the inverse of X . We get the untilted image matrix by subtracting $X\theta$ from $F_{reshape}$ as

$$F_{untilt} = F_{reshape} - X\theta, \quad (3.7)$$

though F_{untilt} needs to be reshaped from a column vector to a matrix. The next step was to apply a Gaussian kernel to the images. The reason for this was to remove additional irrelevant unevenness in the image. The chosen width of the kernel was 64, or about 100 μm . The choice was based on and

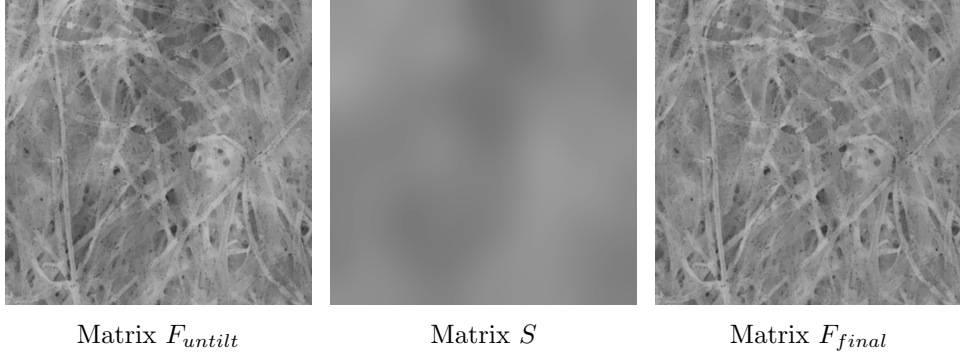


Figure 3.4: The matrices F_{untilt} , S and F_{final} . F_{final} is obtained by subtracting S from F_{untilt} .

tested so that it would not remove too much of the background.

The Gaussian kernel was applied to the matrix F_{untilt} with convolution as defined in Equation 2.4 and the matrix S was obtained. S contained only the very broad changes in the image. The final image F_{final} was calculated by removing the broad parts of the image as

$$F_{final} = F_{untilt} - S. \quad (3.8)$$

In Figure 3.4, F_{untilt} , S and F_{final} can be seen. It can be seen that F_{final} contains less unevenness broadly considered.

Both the untilting and the Gaussian filtering were suggested by M. Arnér at Tetra Pak, and a Matlab code for the computations was available.

3.3 Generative Adversarial Networks on Data

GANs were applied to two nearly identical data sets that only differed in image size. First, it was applied to the images as they were. 1728 images sized 250 by 250, as described in Section 3.1, were used to train the model. We say that the GAN is trained on the full images when referring to this application.

250 by 250 is quite a large image size, and issues with this training were expected, such as training time and finding the right hyper-parameters. Because of this, a GAN was also applied to smaller images. To not increase the training time, the training image set size was kept the same as for the full image GAN. Smaller images were extracted from the full images by cutting out a corner from each image. The same corner for each image was extracted, and the size of the data set remained the same. The size of the

smaller images was chosen to be 64 by 64 and we call them the partial images.

The type of GAN used in this thesis was DCGAN, which stands for deep convolutional generative adversarial network. DCGAN uses convolutions, as the name implies, and is more straightforward in the training than some other types of GANs. The code and parameter choices for this thesis were inspired by the DCGAN tutorial for PyTorch [6]. We call the parameters used in this tutorial the default parameters. These have been tested and shown to work for 64 by 64 images.

The most important default hyper-parameters can be seen in Table 3.1. The first hyper-parameter in the table is the size of the vector that is drawn from the latent space, which is chosen to have a normal distribution with a mean of 0 and a variance of 1. Secondly, the learning rates for the optimization algorithm are specified. The optimization algorithm used in this thesis is Adam because of its robustness. Lastly, the two hyper-parameters β_1 and β_2 that are the forgetting factors for the first and second moment of the gradient in Adam are specified.

For both image sizes, binary cross-entropy loss was used. For the convolutional and deconvolutional layers, the sizes of the kernels were chosen to be 4 by 4. This was for all of the layers and for both image sizes. A reason for using 4 by 4 kernels for deconvolution is to avoid checkerboard artifacts [7]. Checkerboard artifacts mean that the image does not look smooth and pixels can be seen in the image clearer than without the artifacts. With a stride of two, the overlap of the 4 by 4 kernel will be the same over the entire image meaning the overlap will be even. There will be no emphasis on certain points in the generated images. If the kernel length is divisible by the stride, then checkerboard artifacts should not be an issue. 4 by 4 kernels are also what is used in the original DCGAN paper [8]. It is quite convenient to use a convolutional or deconvolutional layer that downscales or upscales the input by a factor of exactly two. It is also convenient to design to architectures of the generator and the discriminator so that they mirror each other.

Size of latent vector/nz	Learning rate/lrD & lrG	β_1	β_2
100	0.0002	0.5	0.999

Table 3.1: The default parameters that were originally used for training. The latent vector is the vector drawn from the latent distribution. The learning rate for the discriminator is lrD while the learning rate for the generator is lrG.

For the partial images, four different parameter settings were tested and reported, each of them run two times giving two models. The chosen parameter settings can be seen in Table 3.2. The choices of these were based on what had improved the full image training as well as general suggestions from various sources. It was suggested in an article by T. Mittal [9] to use different learning rates for the discriminator and the generator, which was shown to improve the training for the full-sized images, and because of this also tested for the partial images.

The last layers in both the discriminator and the generator, as seen in Table 3.3 and Table 3.4, are sigmoid layers. Though they are the same function, they work quite differently depending on their placement. In the discriminator, the sigmoid layer outputs a probability, that of an image being real or generated. In the generator, the sigmoid layer outputs a value of each pixel in the image. Using sigmoid for a probability is reasonable because it takes values between 0 and 1. Though there are other options such as softmax activation function which also takes values between 0 and 1. The reason for using sigmoid as the output function for the generator was because the training images were scaled to take values between 0 and 1, and thus the output should also take values between 0 and 1.

In Table 3.3 and Table 3.4, it can be seen that stride and zero-padding are used in most layers. The architecture that is used in the DCGAN tutorial [6] is for 64 by 64 images. It has five convolutional, or deconvolutional, layers where all layers have a stride of two and zero-padding of one except in the last layer. Here there is no zero-padding and a stride of one, meaning no skips. If the image size would be increased by a factor of 2^k , then adding k convolutional and deconvolutional layers with a stride of two and zero-padding of one would be the corresponding architecture for this size of an image. For the 250 by 250, we can proceed from the architecture for a 256 by 256 image. This would mean that we add two convolutional and

	O	B	Z	D
β_1	0.5	0.3	0.5	0.5
nz	100	100	500	100
lrD	0.0002	0.0002	0.0002	0.00005

Table 3.2: Parameter settings for the partial images. The table describes the choices for each parameter setting. O, B, Z, and D are the four parameter settings. Each of the four parameter settings were run two times, giving two models for each parameter setting that can be viewed as replicates of each other. Two models are called O1 and O2 if they have parameter setting O and so on. O1 and O2 have identical parameter settings: O.

deconvolutional layers with a stride of two and zero-padding of one, because $64 \cdot 2^2 = 256$. For a 250 by 250, this would mean a too fast decrease of the inputs to the generator and the discriminator. Removing the zero-padding and using a stride of one in the second to last layer would make the architecture work for an input of size 250 by 250. There are other changes that could be done, such as changing the kernel sizes in the convolutional and deconvolutional layers, but this change is quite minimal and therefore chosen. The two architectures should be as similar as possible to be able to compare as well as possible.

The outputs from the generator had values as an RGB-colored image, meaning it had the dimension $3 \times n \times n$, where n is the length of the image. It can be seen as three pretty much identical matrices. The mean value of these was taken, and a one-channel colored image was created.

#	Layer	Stride	Pad	C part	C full
1	2D deconvolutional layer	1	0	512	8000
2	Batch normalization layer				
3	ReLU layer				
4	2D deconvolutional layer	1	0	-	4000
5	Batch normalization layer				
6	ReLU layer				
7	2D deconvolutional layer	2	1	-	2000
8	Batch normalization layer				
9	ReLU layer				
4/10	2D deconvolutional layer	2	1	256	1000
5/11	Batch normalization layer				
6/12	ReLU layer				
7/13	2D deconvolutional layer	2	1	128	500
8/14	Batch normalization layer				
9/15	ReLU layer				
10/16	2D deconvolutional layer	2	1	64	250
11/17	Batch normalization layer				
12/18	ReLU layer				
13/19	2D deconvolutional layer	2	1	1	1
14/20	Sigmoid layer				

Table 3.3: Layers in the generator for the partial and full images. The first and last rows (in black) are common for both the partial and the full images, while the red rows are solely for the full images and not the partial images. C stands for channels and is the number of output channels for the partial images (C part) as well as the full images (C full). Pad stands for zero-padding.

#	Layer	Stride	Pad	C part	C full
1	2D convolutional layer	2	1	64	250
2	LeakyReLU layer				
3	2D convolutional layer	2	1	128	500
4	Batch normalization layer				
5	LeakyReLU layer				
6	2D convolutional layer	2	1	256	1000
7	Batch normalization layer				
8	LeakyReLU layer				
9	2D convolutional layer	2	1	512	2000
10	Batch normalization layer				
11	LeakyReLU layer				
12	2D convolutional layer	2	1	-	4000
13	Batch normalization layer				
14	LeakyReLU layer				
15	2D convolutional layer	1	0	-	8000
16	Batch normalization layer				
17	LeakyReLU layer				
12/18	2D convolutional layer	1	0	1	1
13/19	Sigmoid layer				

Table 3.4: Layers in the discriminator for the partial and full images. The first and last rows (in black) are common for both the partial and the full images, while the red rows are solely for the full images and not the partial images. C stands for channels and is the number of output channels for the partial images (C part) as well as the full images (C full). Pad stands for zero-padding.

Chapter 4

Results and Evaluation of Them

In this chapter, the results of the case study are presented and evaluated according to the methods described in Chapter 3.

4.1 Results: Full Images

What was found when training on the full-size images was that either the discriminator loss or the generator loss converged to zero quickly. After only one epoch, the loss was pretty much zero for the discriminator or the generator.

For most parameter settings, the loss of the discriminator was the one that converged to zero. It was first when experimenting with using different learning rates for the discriminator and the generator, that the loss of the generator went to zero. In Table 4.1, two parameter settings that are quite similar are shown. It was found that Training 1, with a smaller learning rate for the discriminator and a smaller size of the latent vector, led to the generator loss converging to zero quickly. Here, the learning rate for the discriminator was 0.00001, and the latent vector size was 10000. When the learning rate for the discriminator and the latent vector size were just slightly larger in Training 2, 0.00005 and 12000, the loss of the discriminator

Parameter	Training 1	Training 2
Learning rate discriminator	0.00001	0.00005
Size of latent vector	10000	12000

Table 4.1: Hyper-parameters that gave different convergence in the discriminator versus the generator.

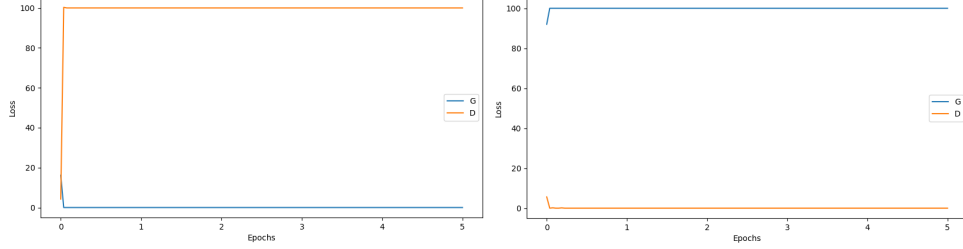


Figure 4.1: Training process for the full-sized images. It can be seen that when we have the parameters for the first training (left), we have a generator that goes to zero, and when we have the parameters for the second training (right) the discriminator goes to zero, both within the first epoch. The blue curve (G) is the loss of the generator while the orange curve (D) is the loss of the discriminator.

went to zero quickly. In Figure 4.1, two examples of training curves can be seen.

The behavior of the training curves could be a consequence of the game between the generator and the discriminator. When the images are larger, the power will be in one player’s hands for a longer time and give that player an immediate advantage. Both Training 1 and Training 2 were trained in replicates and the results of the replicates were quite similar to those shown in Figure 4.1. These can be seen in Figure A.1 (Appendix A).

4.2 Results: Partial Images

For the partial images, the default parameters worked well so there was no struggle with finding parameter settings that worked. In Figure 4.2, we see the results from the eight runs. Without considering the results too closely, they all seem to resemble the real images quite well. Recall that the label of a set of generated images, O1 or B2 for example, refers to the parameter settings shown in Table 3.2.

One thing that can be seen though, is that B2, Z1, and D1, which are runs from the parameter settings in Table 3.2, seem to be slightly lighter in color. This means that they most likely have a greater mean value than the other runs. This actually not an issue because these mean values can be adjusted.

In Figure 4.3, two training curves can be seen, and in Figure A.2 two additional curves can be seen. Jumps in the losses can be seen in all training

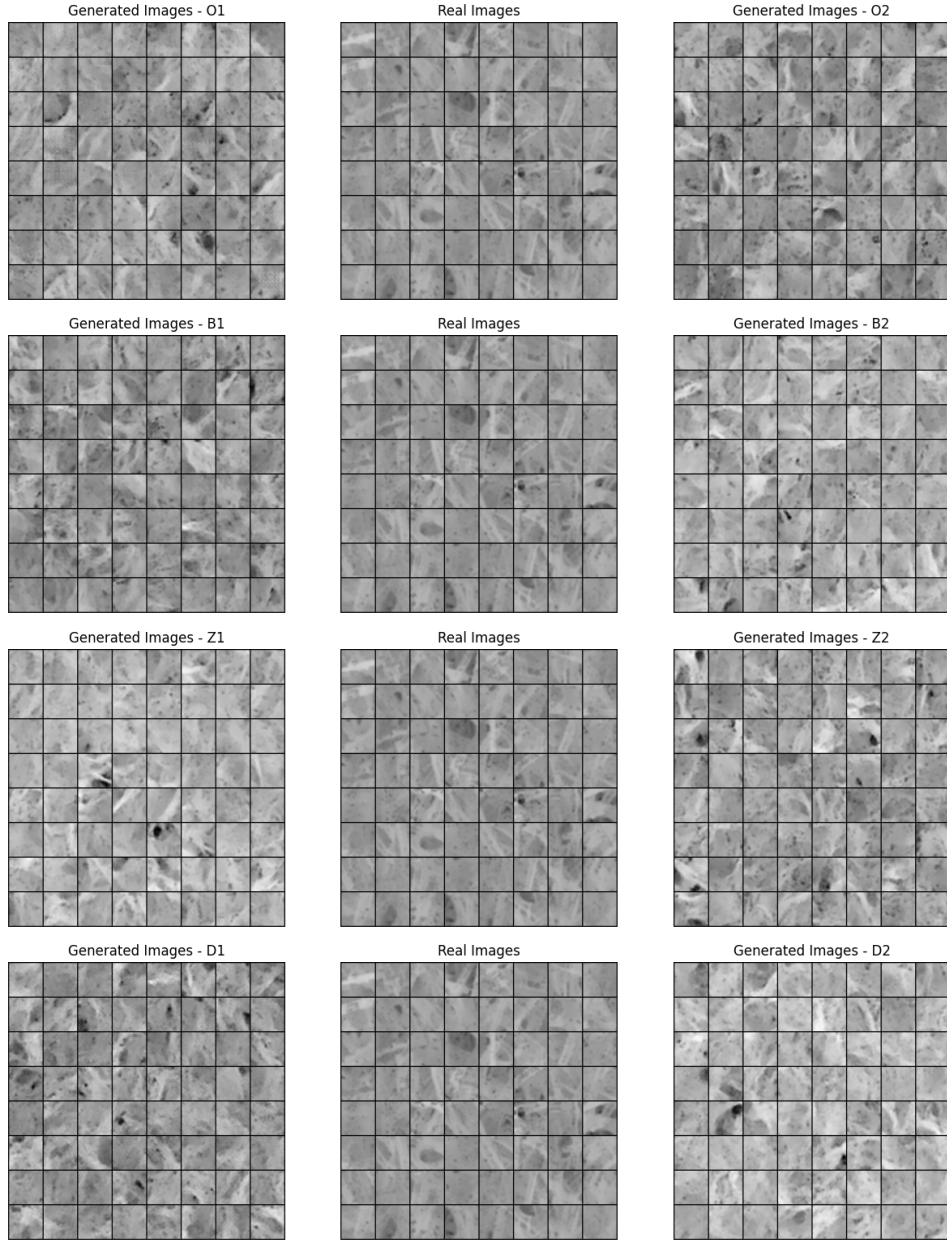


Figure 4.2: The generated images compared to the real images (center row). The labels of the generated images refer to the parameter settings in Table 3.2. Overall, the generated images look quite similar to the real images.

curves. This could be an indication that the training is not completely healthy. The losses should fluctuate to be able to train properly, but these training curves might have too great peaks.

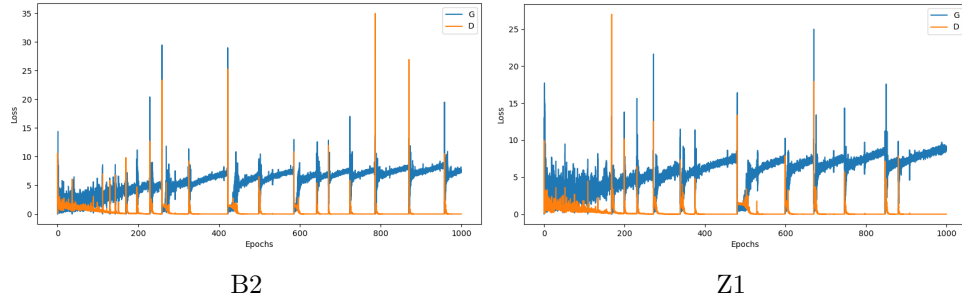


Figure 4.3: Two training curves where quite a lot of fluctuation can be seen. The blue curve (G) is the loss of the generator while the orange curve (D) is the loss of the discriminator.

4.3 Methods for Evaluation

To compare the results from the partial images to each other, the use of some statistical measurements was required. This was because the generated images needed to have the same statistical properties as the real images. Below, the used methods of evaluation are listed. Each set of images, generated or real, was always the same randomly selected 64 images.

- Visual inspection of the outputs
- Compare distributions between groups of images using histograms
- Autocorrelation to consider vertical and horizontal correlations in the images
- Multidimensional scaling (MDS) with Wasserstein distance to examine the distance between images

A simple, but effective, tool of investigation is to just look at the images and see how well you can distinguish the real images from the generated images. If we can clearly see that the generated images do not resemble the real images, there is no reason to further consider them as good results with the same statistical properties as the real images. The statistical methods of evaluation might even tell the same thing.

Another method of evaluation is the use of histograms. They show how the intensity in the pixels is distributed. One histogram per set of images is computed, taking all pixels, or values, in the images and plotting it in a histogram. Recall that an image contains $64 \cdot 64 = 4096$ pixels, and a set of images contains 64 images. This means that a histogram will contain

$64^3 = 262144$ pixels.

Autocorrelation is used to see how the correlation in the images behaves. The autocorrelation is computed as described in Chapter 3, computing the autocorrelation in both MD and CD and summing the results for each direction.

Multidimensional scaling, or MDS, is an unsupervised learning method for visually representing distances or similarities between objects. Objects that are closer together on the graph are more similar to each other. When using MDS, a choice of how many dimensions to construct a graph for is made. Typically two, or possibly three, dimensions are used because this is where it is the easiest to visualize. In this sense, MDS works as a dimensionality reduction technique. With MDS, a matrix containing pairwise distances between objects can be analyzed. This matrix is symmetric and has a zeros diagonal; because the distance between an object and itself is zero. [10]

To get a distance matrix, Wasserstein distance is used. Wasserstein distance is a distance measure between distributions of objects pairwise. For this thesis, the usage of Wasserstein distance was to find the distance between the distributions of the images. This is used to create a distance matrix for the MDS.

4.4 Evaluation of Results

In this section, the generated images from the set of partial images are evaluated using the methods from Section 4.3. Only these results are evaluated by statistical methods. This is because it is straightforward by looking at the training curves of the full images, that the results do not resemble those in the training set.

Figure 4.4 shows a sample image from the two models within the four parameter settings as seen in Table 3.2. Note that these images are quite small; a length of 64 pixels is only about 100 μm . For the parameter setting O, which is the default parameter setting, we see some checkerboard artifacts in the generated images, and more so in replicate O1. This effect is also seen in B1, Z2, and D1, though not as strongly as in O1 and O2. For this reason, the main models of consideration will be B2, Z1, and D2. Evaluation will also be done on O1 along with B2, Z1, and D2 to compare the performance of the methods.

Histograms of B2, Z1, D2, and O1 were computed together with the real

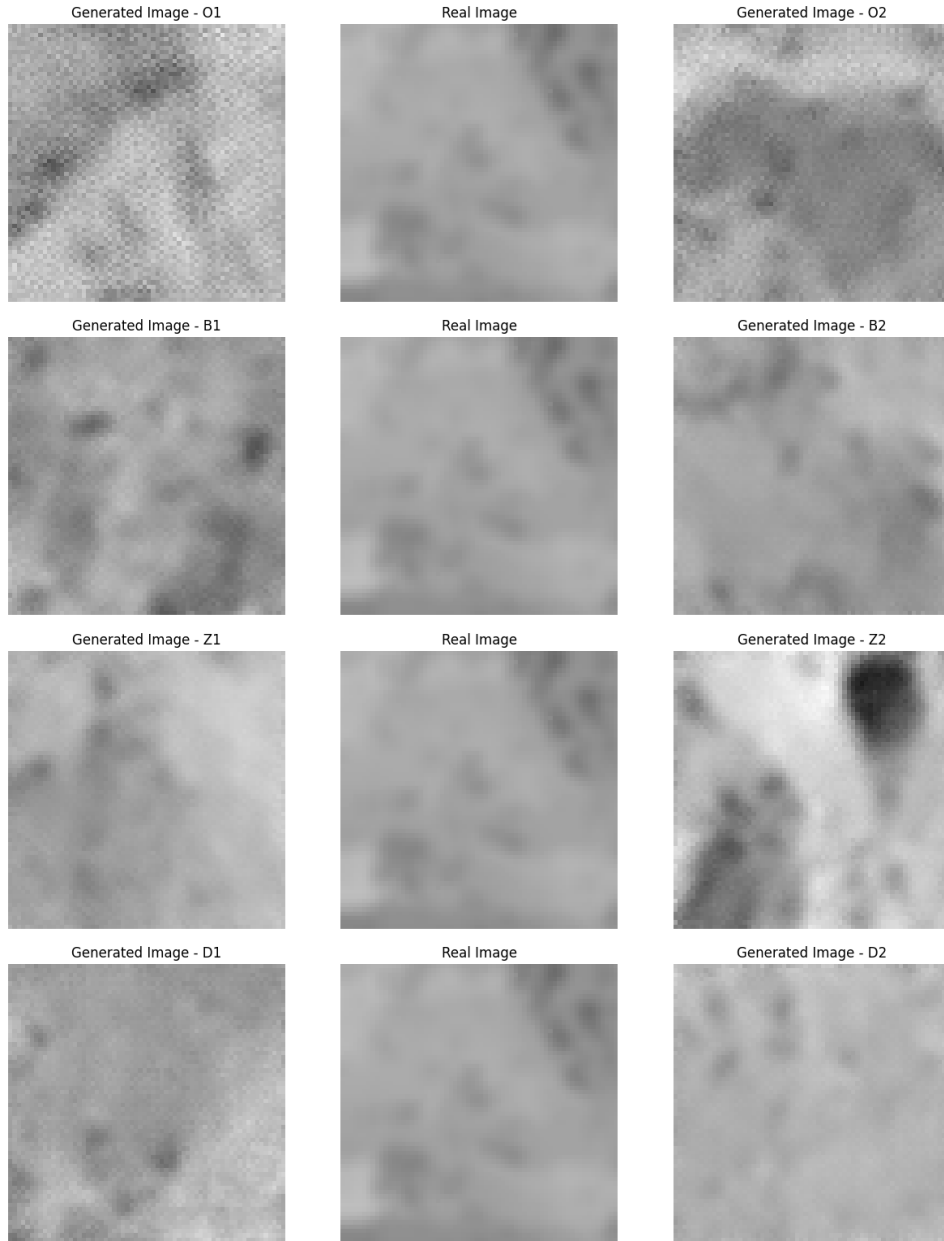


Figure 4.4: A few generated images compared to a real image (center row). The labels of the generated images refer to the parameter settings in Table 3.2.

images. In Figure 4.5 we see these histograms. Overall, the generated images tend to have a larger variance and a greater mean value than the real images. Though, they do all seem to have very similar distributions. O1 has the mean value closest to the real mean value, while B2 seems to have

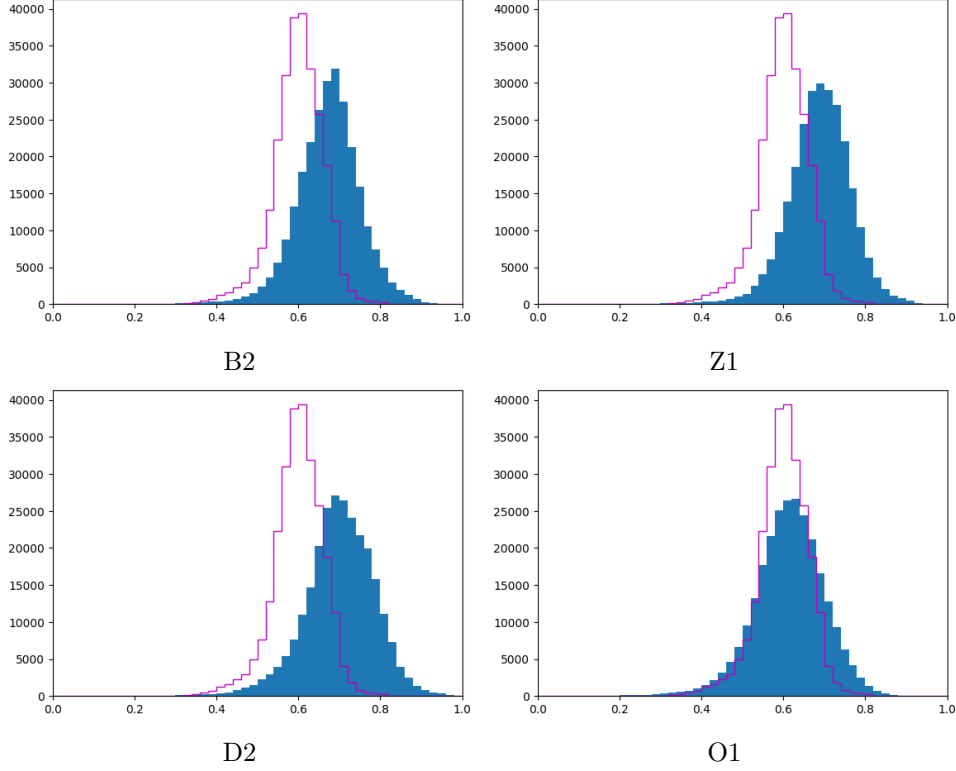


Figure 4.5: Histograms of the generated images (blue bars) together with the histogram of the real images (purple steps).

a variance that is closest to the real variance. The distribution of D2 looks to be the furthest from the real distribution. In Figure A.3 we see the histograms of the remaining generated images.

Considering the autocorrelation of a few images, we see in Figure 4.6 that the autocorrelations of B2 and Z1 both have similar behaviors to the autocorrelation for the real image. Looking at the autocorrelation of an O1 image, an irregular behavior can be seen, especially in the MD. This is likely a consequence of the checkerboard artifacts seen in the generated images.

Note that autocorrelation is computed for one image and does not represent the entirety of the 64 images in each set. In Figure A.5, another six image's autocorrelations from each set are shown, including two autocorrelations from D2 that show similar behavior to those of B2 and Z1. Considering all autocorrelation plots in Figure 4.6 and A.5, all sets of generated data decrease slower in the MD in most cases. This is also true for the real images and can be seen in Figure 4.6 and A.5, but also in Figure 3.2 which shows an autocorrelation for a full-sized image (1000 by 1000). Here, the number of

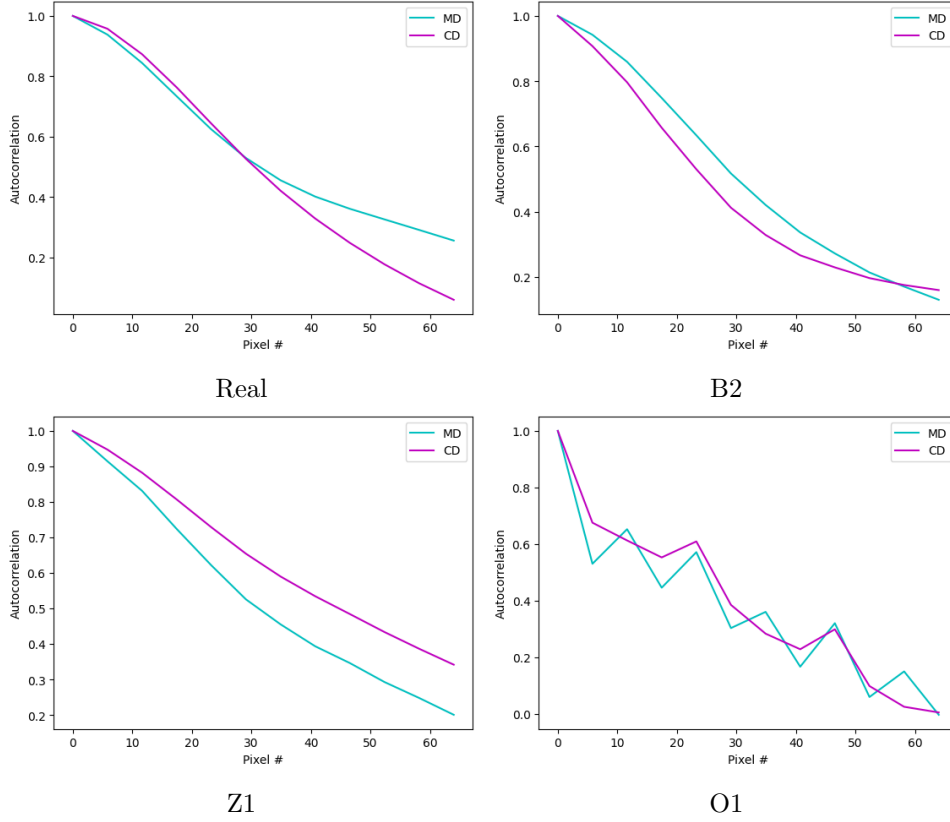


Figure 4.6: Autocorrelation for a few of the generated and real images. The CD curve is the mean value of all row-wise autocorrelations, while the MD curve is the mean value of all column-wise autocorrelation.

lags is 12, which gives a lag length of approximately 5 pixels. This is about the same lag length as in Figure 3.2.

Looking at the MDS plot in Figure 4.7, we see that only O1 seems to lay close to the real images. Just as for the histograms, this could be a consequence of B2, Z1, and D2 having greater mean values than both the real images and images of type O1. The Wasserstein distance was plotted against the Euclidean distance in Figure A.4 to check that they coincide.

Concluding from these methods, B2 seems to be the run that performed the better. The fact that the mean value of B2 seems to be slightly off is not really an issue because this can be adjusted. In Figure 4.8, the generated images of type B2 have been shifted in mean value to have the same mean value as the real images. It can be seen that the overall shades of the images now match better with the real images.

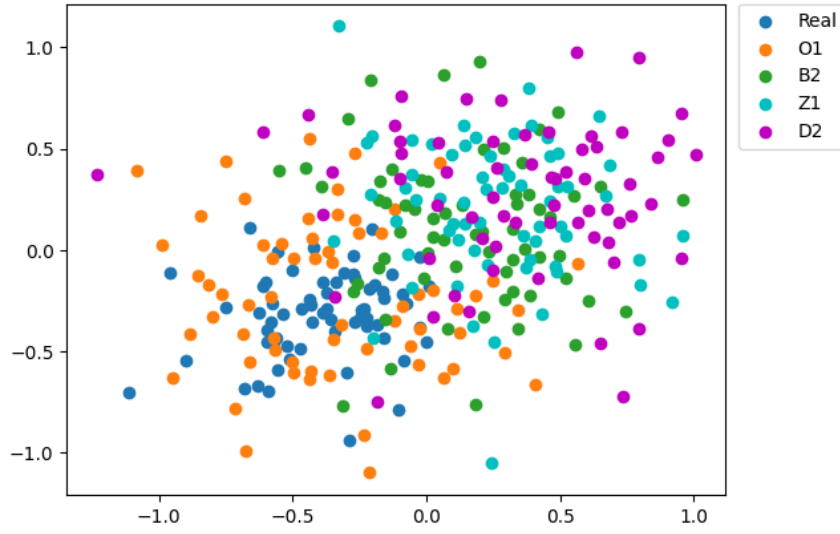


Figure 4.7: MDS for four runs of generated images together with the real images.

In Figure 4.9, we see the histogram and MDS for the shifted B2. In the histogram, it can be seen that the distribution matches quite well with the real images' histogram, but the variance is still slightly too large. In the MDS graph, both the real images and B2 seem to center around the same area as opposed to before the shift of the mean value. The points for B1 are possibly a bit more spread out than the real images, but overall they seem to lay quite close in distribution. The correlation in the images will stay the same and is because of this not plotted.

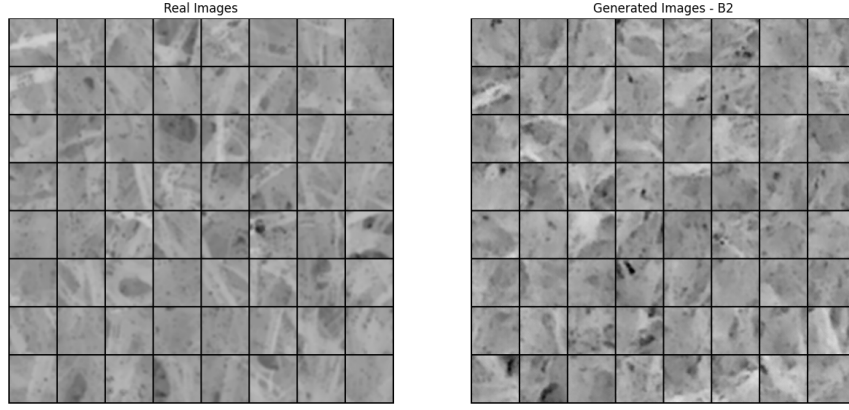


Figure 4.8: Real and generated images of parameter settings B2, where the generated images have been shifted to have the same mean value as the real images.

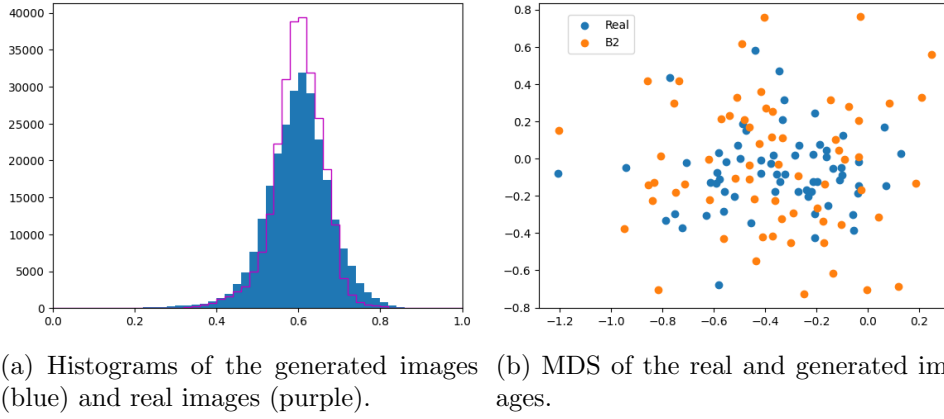


Figure 4.9: Histogram and MDS for the generated images of parameter settings B2, where the generated images have been shifted to have the same mean value as the real images.

Chapter 5

Discussion

The ideal outcome of the study would be to be able to generate images of paperboard surfaces that captured the entire span of information in the images. The size of these, after dividing them into 16 smaller images, was 250 by 250. This is a quite large image size for GAN, which typically takes the input size of 64 by 64. Considering the number of pixels in a 64 by 64 image we have 4096, while in a 250 by 250 image, there are 62500 pixels. There are thus 15.26 times more pixels in a 250 by 250 image than in a 64 by 64 image.

A challenge with generating larger images is the training time. It increases quite rapidly, firstly because the input is 15.26 times larger, but also because the neural network has to be larger architecturally with a larger input. Finding parameter settings that work is more difficult because only training a few epochs to test for convergence takes several hours. This does not leave space for much trial and parameter tuning. Even with a lot of research and knowledge about the subject, the reality is that at least some parameter tuning has to be done. If hyper-parameters that do not lead to a failure within the first few epochs are found, it might still lead to a failure a few epochs later. In any case, training 500 epochs, which is considered to be the minimum epochs needed for the problem, would take several weeks.

A checkerboard effect can be seen in some of the generated images. On some generated images it is solely seen on the edges. A way to treat this would be to use an upsampling layer and a convolutional layer instead of a deconvolutional layer [7]. Because a kernel size of four together with a stride of two was used, this was not expected to be a large problem. Though, it is something to consider and something that possibly could improve the results.

DCGAN is only one out of several types of GAN. This is one of the simpler types of GANs. The learning in a DCGAN can be seen as a game, where each player, the generator and the discriminator, has the power during its

entire round. Because of this, DCGAN does not work that well with larger image sizes. Too much will happen in a round, and this will make it more difficult for the other player to catch up.

The best results of the partial images were for B2, where parameter settings B were used. The deviation from the default hyper-parameters was that $\beta_1 = 0.3$ instead of $\beta_1 = 0.5$. This means that using a lower forgetting factor for past gradients improved the training. The previous steps of the gradients will then matter more for future steps. A small β_1 will make the Adam algorithm more similar to another algorithm called the RMSProp algorithm. This is, just like Adam, a popular optimization algorithm so it is not surprising for a lower β_1 to improve the learning.

Chapter 6

Conclusion

To conclude this thesis: The size of the images have a great effect on GAN and specifically on DCGAN. Training on 64 by 64 images gave good results with all tested parameter setting, and have the potential to generate images that are completely indistinguishable from the real images. The final results of this thesis were not far from this.

For the 250 by 250, no results were obtained. Training with larger images is more difficult and also unexplored. For the 64 by 64 images, it was easy to find parameter settings that had been tested to work for others, but for 250 by 250 sized images, nothing like this was found. GAN is very sensitive to changes in parameters, and without a ground to start out on this becomes a large issue.

6.1 Further Studies

It is possible that GAN is not the better deep generative model for the problem of 250 by 250-sized images. Other models worth trying are variational autoencoders as well as diffusion models.

A variational autoencoder, or VAE, has the same architecture as an autoencoder, but different goals and mathematical formulations. An autoencoder, for that part, is an artificial neural network that finds features in unlabeled data (unsupervised learning) [11]. Their architecture is encoder-decoder. They produce codes for the input data and are trained so that the decoded outputs resemble the input as much as possible.

A VAE can be explained as an autoencoder that uses a variational Bayesian approach to the encoding [11]. They have a generator that is not invertible, so the loss cannot be computed directly. Instead, maximum likelihood

training is used. We have

$$p_{\theta}(x) = \frac{p_{\theta}(x|z)p_{\mathcal{Z}}(z)}{p_{\theta}(z|x)} \quad (6.1)$$

which is called the evidence. Recall that θ are the parameters of the generator and $p_{\mathcal{Z}}(z)$ is a known distribution. $p_{\theta}(z|x)$, the posterior, is what is approximated using variational inference.

Using diffusion models would be another option. This is a quite new method that has been proven to perform well [12]. A diffusion model is a Markov chain where Gaussian noise is gradually added to it [13]. The process is then reversed by training a neural network to recover the original data. Just as VAEs, diffusion models use likelihoods for the optimization.

VAEs and diffusion models are not based on a two-player game where the players take turns, as DCGAN is. Because of this, these types of deep generative models may work better with larger image sizes.

Other types of GANs that train in a different way than DCGAN could perform better when the input images are larger. An example is StyleGAN [14]. StyleGAN has been shown to be able to generate with high quality for images as large as 1024 by 1024.

Bibliography

- [1] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville and Yoshua Bengio. *Generative Adversarial Nets*. Proceedings of the 27th International Conference on Neural Information Processing Systems (NIPS 2014), 2014, pp. 2672–2680.
- [2] Aston Zhang, Zachary C. Lipton, Mu Li and Alexander J. Smola. *Dive into Deep Learning*. arXiv preprint arXiv:2106.11342, 2021.
- [3] Lars Ruthotto and Eldad Haber. *An introduction to deep generative modeling*. GAMM - Mitteilungen, 2021.
- [4] Ian Goodfellow, Yoshua Bengio and Aaron Courville. *Deep Learning*. MIT Press, 2016, <https://www.deeplearningbook.org/>.
- [5] Diederik P. Kingma and Jimmy Lei Ba. *ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION*. Conference paper at ICLR, 2015.
- [6] Nathan Inkawhich. *DCGAN TUTORIAL*. PyTorch, 2023, https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html#dcgan-tutorial.
- [7] Augustus Odena, Vincent Dumoulin and Chris Olah. *Deconvolution and Checkerboard Artifacts*. Distill, 2016, <https://distill.pub/2016/deconv-checkerboard/>.
- [8] Alec Radford, Luke Metz and Soumith Chintala. *UNSUPERVISED REPRESENTATION LEARNING WITH DEEP CONVOLUTIONAL GENERATIVE ADVERSARIAL NETWORKS*. 2016.
- [9] Tushar Mittal. *Tips On Training Your GANs Faster and Achieve Better Results*. Medium, 2019, <https://medium.com/intel-student-ambassadors/tips-on-training-your-gans-faster-and-achieve-better-results-9200354acaa5>.
- [10] John A. Lee and Michel Verleysen. *Nonlinear Dimensionality Reduction*. Springer, 2007, pp. 73.

- [11] David Charte, Francisco Charte, Salvador García, María J. del Jesus and Francisco Herrera. *A practical tutorial on autoencoders for nonlinear feature fusion: Taxonomy, models, software and guidelines*. Elsevier, 2017.
- [12] Jonathan Ho, Ajay Jain and Pieter Abbeel. *Denoising Diffusion Probabilistic Models*. 34th Conference on Neural Information Processing Systems (NeurIPS 2020), 2020.
- [13] Sergios Karagiannakos and Nikolas Adaloglou. *How diffusion models work: the math from scratch*. <https://theaisummer.com/>, 2022, <https://theaisummer.com/diffusion-models/>.
- [14] Synced. *NVIDIA Open-Sources Hyper-Realistic Face Generator StyleGAN*. Medium, 2019, <https://medium.com/syncedreview/nvidia-open-sources-hyper-realistic-face-generator-stylegan-f346e1a73826>.

Appendix A

Figures

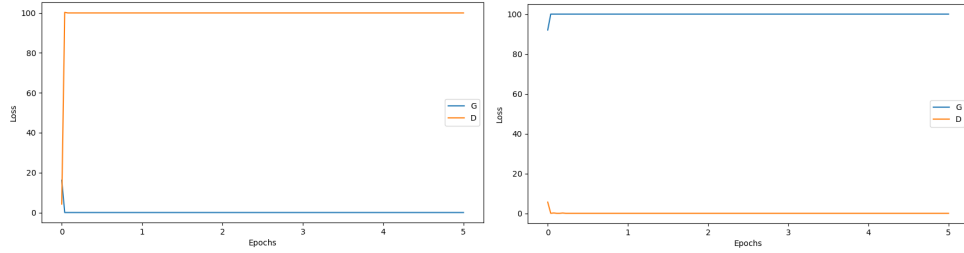


Figure A.1: Training process for the full-sized images. It can be seen that when we have the parameters for the first training (left), we have a generator that goes to zero, and when we have the parameters for the second training (right) the discriminator goes to zero, both within the first epoch. The blue curve (G) is the loss of the generator while the orange curve (D) is the loss of the discriminator.

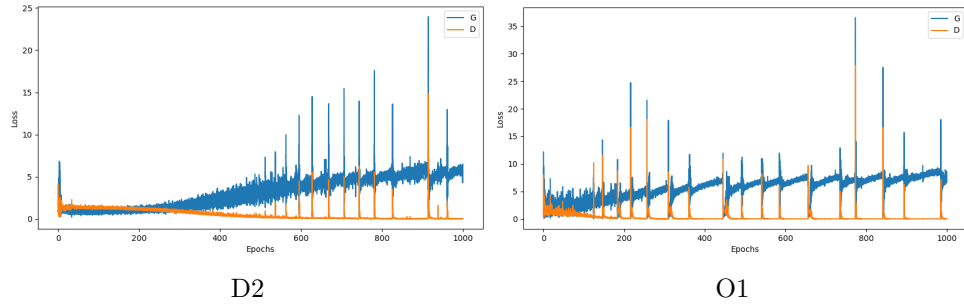


Figure A.2: Two training curves where quite a lot of fluctuation can be seen. The blue curve (G) is the loss of the generator while the orange curve (D) is the loss of the discriminator

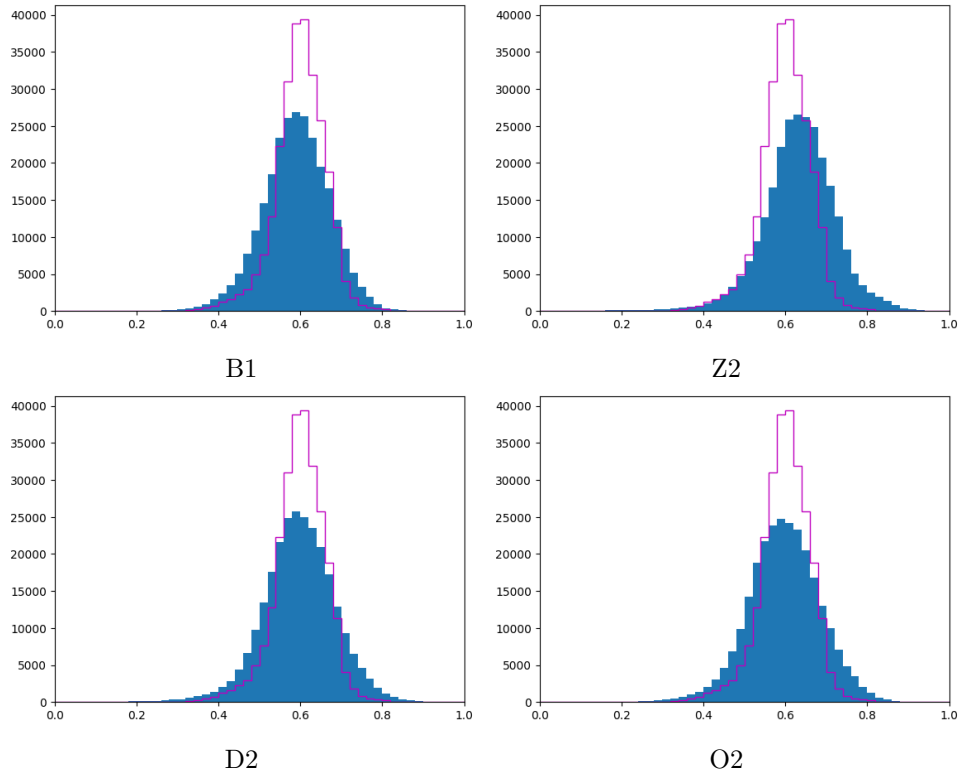


Figure A.3: Histograms of the generated images (blue bars) together with the histogram of the real images (purple steps).

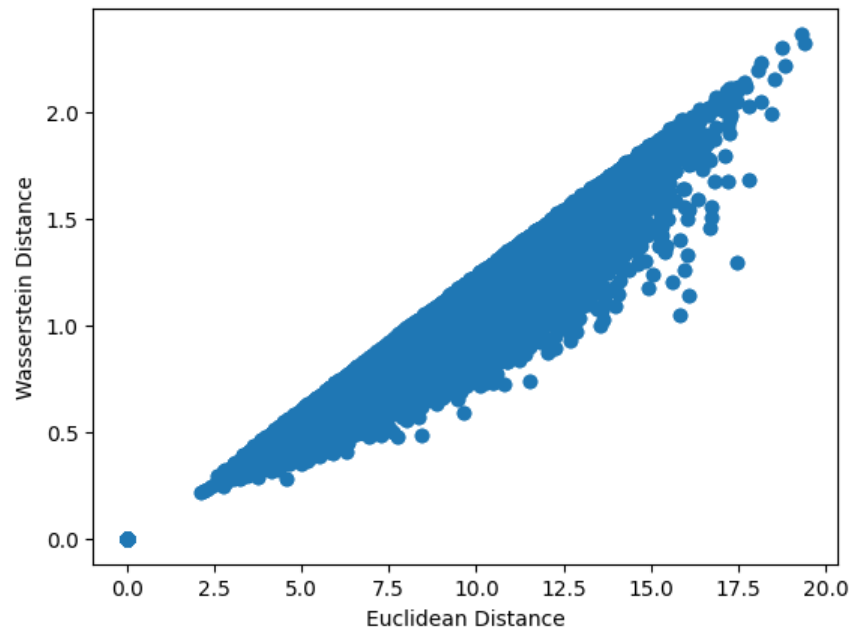


Figure A.4: The Wasserstein distance was plotted against the Euclidean distance for the distances between the real data, O1, B2, Z1 and D2.

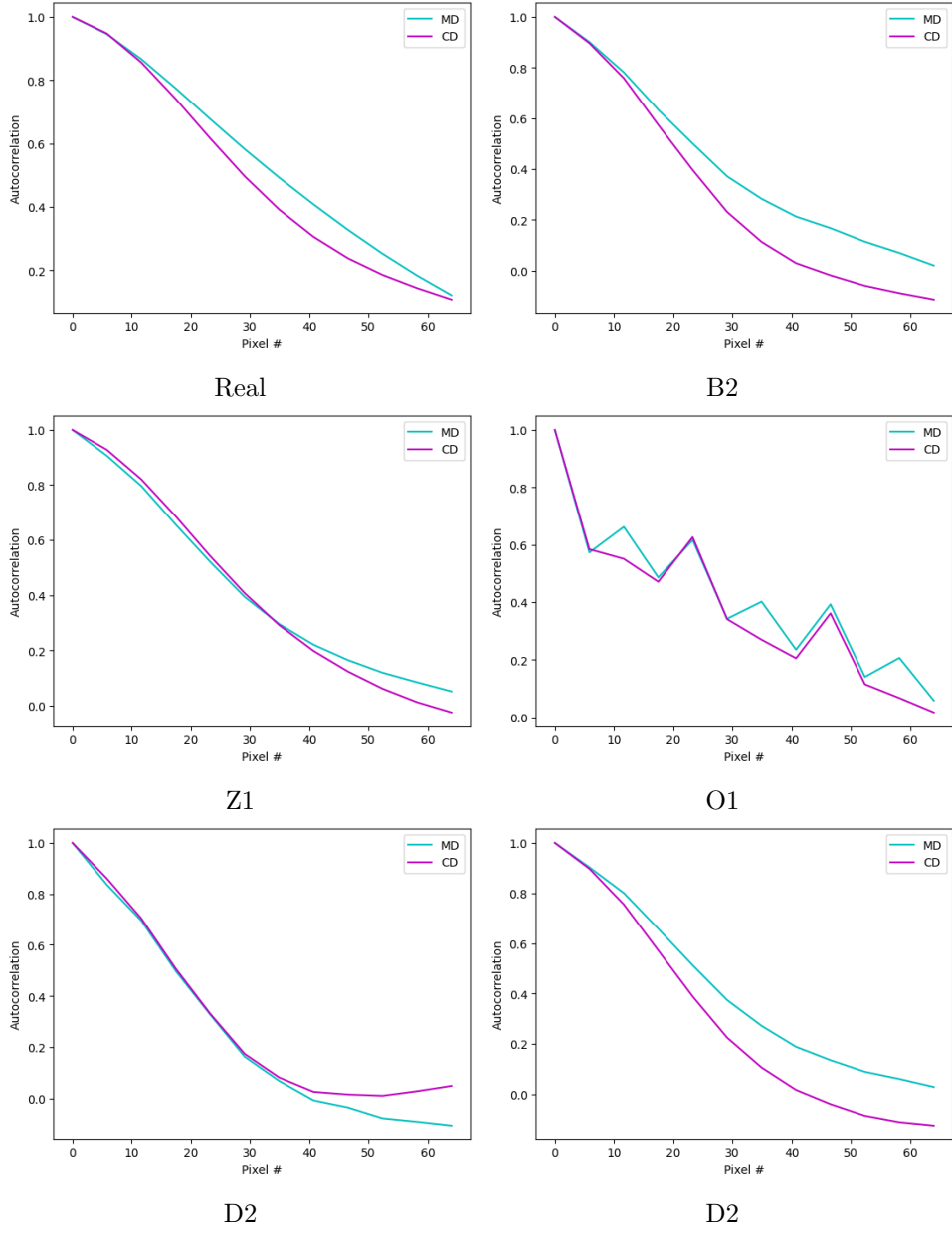


Figure A.5: Autocorrelation for a few of the generated and real images. The CD curve is the mean value of all row-wise autocorrelations, while the MD curve is the mean value of all column-wise autocorrelation.