UiT The Arctic University of Norway

Faculty of Science and Technology

Department of Mathematics and Statistics

A study of generative adversarial networks to improve classification of microscopic foraminifera

Eirik Agnalt Østmo

Master's thesis in mathematics and education, year 8-13 – MAT-3907 – June 2020

Cover image: The cover image displays artificial planktic foraminifera that are generated by the generative adversarial network (GAN) of section 4.5. The smooth transitions between samples is one of the many interesting properties a GAN can learn unsupervised.

Abstract

Foraminifera are single-celled organisms with shells that live in the marine environment and can be found abundantly as fossils in e.g. sediment cores. The assemblages of different species and their numbers serves as an important source of data for marine, geological, climate and environmental research.

Steps towards automatic classification of foraminifera using deep learning (DL) models have been made (Johansen and Sørensen, 2020), and this thesis sets out to improve the accuracy of their proposed model. The recent advances of DL models such as generative adversarial networks (GANs) (Goodfellow *et al.*, 2014), and their ability to model high-dimensional distributions such as real-world images, are used to achieve this objective.

GANs are studied and explored from a theoretical and empirical standpoint to uncover how they can be used to generate images of foraminifera. A multi-scale gradient GAN is implemented, tested and trained to learn the distributions of four high-level classes of a recent foraminifera dataset (Johansen and Sørensen, 2020), both conditionally and unconditionally. The conditional images are assessed by an expert and a deep learning classification model and is found to contain mostly valuable characteristics, although some artificial artifacts are introduced. The unconditional images measured a Fréchet Inception distance of 47.1.

From the conditionally learned distributions a total of 10 000 images are sampled from the four distributions. These images are used to augment the original foraminifera training set in an attempt to improve the classification accuracy of (Johansen and Sørensen, 2020). Due to limitations of computational resources, the experiments were carried out with images of resolution 128×128. The synthetic image augmentation lead to an improvement in mean accuracy from $97.3 \pm 0.4\%$ to $97.4 \pm 0.7\%$ and an improvement in best achieved accuracy from 97.7% to 98.5%.

Acknowledgements

First and foremost I would like to express my sincerest gratitude to my supervisors: To PhD Thomas Haugland Johansen for answering my questions and pointing me in the right direction during my exploration of the field of study. Thank you for the technical support you have provided and for our valuable discussions. To professor Fred Godtliebsen for giving me the opportunity to write this thesis and for your motivational and valuable feedback.

I also want to thank the two marine geologists that has contributed to this thesis: Steffen Aagaard Sørensen for qualitatively assessing my synthetic foraminifera, and Christine Tømmervik Kollsgård for our useful discussions and your feedback on the sections concerning foraminifera.

To my fiancee Ingeborg, thank you for the love, support and patience you have shown me during the work of this thesis.

I wish to thank my classmates at UiT for the fellowship and good times we have had these past five years. A special thanks goes to Idunn that joined me when I wanted to change my secondary subject from chemistry to physics during freshman year, and has been my study companion ever since.

To my friends and family, thank you for your support. and to my housemate Kristine, thank you for cheering me all the way to the finish line.

Lastly, I dedicate a special thanks to my childhood friend Edvard, for your support and regular video calls during the months of home office during the corona pandemic.

This would not be possible without any of you.

Eirik Agnalt Østmo
Tromsø, May 2020

Contents

List of Tables

List of Figures

Notation

This section provides a concise reference describing notation used throughout this document. The typesetting is done in LaTeXwith notation template from (Goodfellow, 2016a).

Numbers and Arrays

a	A scalar (integer or real)
$\boldsymbol{a}$	A vector
$\boldsymbol{A}$	A matrix
A	A tensor
$\boldsymbol{I}_n$	Identity matrix with n rows and n columns
$\boldsymbol{I}$	Identity matrix with dimensionality implied by context
$\boldsymbol{e}^{(i)}$	Standard basis vector $[0, \dots, 0, 1, 0, \dots, 0]$ with a 1 at position i
$\text{diag}(\boldsymbol{a})$	A square, diagonal matrix with diagonal entries given by $\boldsymbol{a}$
a	A scalar random variable
$\mathbf{a}$	A vector-valued random variable
$\mathbf{A}$	A matrix-valued random variable

Sets

$\mathbb{A}$	A set
$\mathbb{R}$	The set of real numbers
$\{0, 1\}$	The set containing 0 and 1
$\{0, 1, \dots, n\}$	The set of all integers between 0 and n
$[a, b]$	The real interval including a and b
$(a, b]$	The real interval excluding a but including b
$\mathbb{A} \backslash \mathbb{B}$	Set subtraction, i.e., the set containing the elements of $\mathbb{A}$ that are not in $\mathbb{B}$

Indexing

a_i	Element i of vector $\boldsymbol{a}$, with indexing starting at 1
a_{-i}	All elements of vector $\boldsymbol{a}$ except for element i
$A_{i,j}$	Element i, j of matrix $\boldsymbol{A}$
$\boldsymbol{A}_{i,:}$	Row i of matrix $\boldsymbol{A}$
$\boldsymbol{A}_{:,i}$	Column i of matrix $\boldsymbol{A}$
$\mathsf{A}_{i,j,k}$	Element (i, j, k) of a 3-D tensor A
$\mathsf{A}_{:,:,i}$	2-D slice of a 3-D tensor
a_i	Element i of the random vector $\mathbf{a}$

Linear Algebra Operations

$\boldsymbol{A}^\top$	Transpose of matrix $\boldsymbol{A}$
$\boldsymbol{A}^+$	Moore-Penrose pseudoinverse of $\boldsymbol{A}$
$\boldsymbol{A} \odot \boldsymbol{B}$	Element-wise (Hadamard) product of $\boldsymbol{A}$ and $\boldsymbol{B}$
$\boldsymbol{A} * \boldsymbol{B}$	Convolution of the kernel $\boldsymbol{A}$ over the matrix $\boldsymbol{B}$
$\det(\boldsymbol{A})$	Determinant of $\boldsymbol{A}$

Calculus

$\frac{dy}{dx}$	Derivative of y with respect to x
$\frac{\partial y}{\partial x}$	Partial derivative of y with respect to x
$\nabla_{\boldsymbol{x}} y$	Gradient of y with respect to $\boldsymbol{x}$
$\nabla_{\boldsymbol{X}} y$	Matrix derivatives of y with respect to $\boldsymbol{X}$
$\nabla_{\mathbf{X}} y$	Tensor containing derivatives of y with respect to $\mathbf{X}$
$\frac{\partial f}{\partial \boldsymbol{x}}$	Jacobian matrix $\boldsymbol{J} \in \mathbb{R}^{m \times n}$ of $f : \mathbb{R}^n \to \mathbb{R}^m$
$\nabla_{\boldsymbol{x}}^2 f(\boldsymbol{x})$ or $\boldsymbol{H}(f)(\boldsymbol{x})$	The Hessian matrix of f at input point $\boldsymbol{x}$
$\int f(\boldsymbol{x}) d\boldsymbol{x}$	Definite integral over the entire domain of $\boldsymbol{x}$
$\int_{\mathbb{S}} f(\boldsymbol{x}) d\boldsymbol{x}$	Definite integral with respect to $\boldsymbol{x}$ over the set $\mathbb{S}$

Probability and Information Theory

$\mathrm{a} \perp \mathrm{b}$	The random variables a and b are independent
$\mathrm{a} \perp \mathrm{b} \mid \mathrm{c}$	They are conditionally independent given c
$P(\mathrm{a})$	A probability distribution over a discrete variable
$p(\mathrm{a})$	A probability distribution over a continuous variable, or over a variable whose type has not been specified
$\mathrm{a} \sim P$	Random variable a has distribution P
$\mathbb{E}_{\mathrm{x} \sim P}[f(x)]$ or $\mathbb{E} f(x)$	Expectation of $f(x)$ with respect to $P(\mathrm{x})$
$\mathrm{Var}(f(x))$	Variance of $f(x)$ under $P(\mathrm{x})$
$\mathrm{Cov}(f(x), g(x))$	Covariance of $f(x)$ and $g(x)$ under $P(\mathrm{x})$
$H(\mathrm{x})$	Shannon entropy of the random variable x
$D_{\mathrm{KL}}(P \| Q)$	Kullback-Leibler divergence of P and Q
$\mathcal{N}(\boldsymbol{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma})$	Gaussian distribution over $\boldsymbol{x}$ with mean $\boldsymbol{\mu}$ and covariance $\boldsymbol{\Sigma}$

Functions

$f : \mathbb{A} \to \mathbb{B}$	The function f with domain $\mathbb{A}$ and range $\mathbb{B}$
$f \circ g$	Composition of the functions f and g
$f(\boldsymbol{x}; \boldsymbol{\theta})$	A function of $\boldsymbol{x}$ parametrized by $\boldsymbol{\theta}$. (Sometimes $f(\boldsymbol{x})$ is used but the argument $\boldsymbol{\theta}$ is omitted to lighten the notation)
$\log x$	Natural logarithm of x
$\sigma(x)$	Logistic sigmoid, $\dfrac{1}{1+\exp(-x)}$
$\|\|\boldsymbol{x}\|\|_p$	L^p norm of $\boldsymbol{x}$
$\|\|\boldsymbol{x}\|\|$	L^2 norm of $\boldsymbol{x}$
x^+	Positive part of x, i.e., $\max(0, x)$
$\mathbf{1}_{\text{condition}}$	is 1 if the condition is true, 0 otherwise

Sometimes a function f whose argument is a scalar is used but applied to a vector, matrix, or tensor: $f(\boldsymbol{x})$, $f(\boldsymbol{X})$, or $f(\mathsf{X})$. This denotes the application of f to the array element-wise. For example, if $\mathsf{C} = \sigma(\mathsf{X})$, then $\mathsf{C}_{i,j,k} = \sigma(\mathsf{X}_{i,j,k})$ for all valid values of i, j and k.

Datasets and Distributions

p_{data}	The data generating distribution
$\hat{p}_{\text{data}}$	The empirical distribution defined by the training set
$\mathbb{X}$	A set of training examples
$\boldsymbol{x}_i$	The i-th example (input) from a dataset
y_i or $\boldsymbol{y}_i$	The target associated with $\boldsymbol{x}_i$ for supervised learning
$\boldsymbol{X}$	The $m \times n$ matrix with input example $\boldsymbol{x}_i$ in row $\boldsymbol{X}_{i,:}$

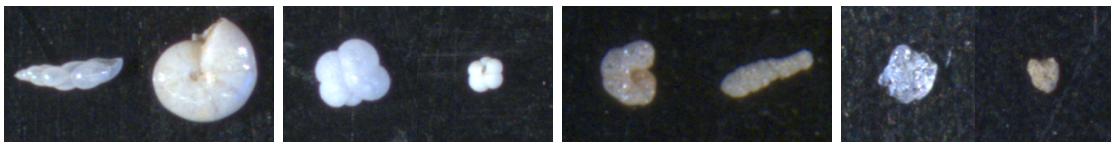
Chapter 1

Introduction

1.1 What are foraminifera?

Foraminifera are single-celled organisms (figure 1.1) that live in the marine environment. Although they are single-celled they often produce a shell (test) with one or multiple chambers encapsulating the organism. The shells of the foraminifera are made of minerals from the environment of the species, commonly calcium carbonate ($CaCO_3$) or agglutinated sediment particles. If the conditions are right (e.g. not too acidic), the shells of the foraminifera are preserved as fossils. These fossils can be found in the sediment of the sea floor, and are today an important source of information for scientists reconstructing the ancient environment of our planet and for petroleum exploration (O'neill, 1996).

Most marine foraminifera are **benthic** and thus live on or within the sediment of the sea floor, while a smaller variety are **planktic** which live and float in the water column at different depths. In total there are over 50 000 recognized species and subspecies of foraminifera, both living (10 000) and fossilized (40 000) (Hayward *et al.*, 2020). Sizes usually vary from 0.05 mm to 0.5 mm, although some species



(a) Calcareous benthic (b) Planktic (c) Agglutinated benthic (d) Sediment grain

Figure 1.1: Specimen of foraminifera and sediment grains that are extracted from sediment cores. The images are from the foraminifera dataset of (Johansen and Sørensen, 2020)

can grow up to several centimeters (Marshall, 2010).

1.1.1 Importance in research

Foraminifera are present in most marine sediment and have become the most studied group of fossils worldwide (Hayward *et al.*, 2020). The study of preserved foraminifera shells are of great importance in e.g. biostratigraphy, paleoenvironmental studies and isotope geochemistry.

The utility of foraminifera comes from the information that can be obtained from studying the foraminifer assemblages in e.g. sediment cores. A sediment core is a cross section of the seabed that reveal the layers of sediment that has been deposited over millions of years. One cubic centimeter of sediment may contain hundreds of foraminifera (Sabbatini *et al.*, 2014) making them the most abundant shelled organism in many marine environments (Hayward *et al.*, 2020). As different species thrive in different living conditions, the relative numbers of e.g. benthic and planktic species, the ratio of shell types or shell chemistry may provide important information regarding e.g. salinity and temperature (Sabbatini *et al.*, 2014).

To illustrate the utility of foraminifera in the context of climate research, one may consider the ratio of different isotopes in the chemical composition of foraminifera shells. The isotope ratios in the shell is thought to reflect the chemistry of the water in which the foraminifera grew (Sabbatini *et al.*, 2014). This ratio may thus reveal important information about the environment at the time. Calcareous shells of some foraminifera contain carbonate (CO_3^-) that was formed from e.g. carbon dioxide (CO_2) from the atmosphere, that was dissolved in the water at the time of formation of the shell. Analyzing the ratios of stable carbon isotopes in foraminifera shells may therefore give information about the amount of carbon and CO_2 in the atmosphere at the time. In a similar manner, the ratios of oxygen isotopes found in the chemical bonds of the shell can give an indication of how much of earths water that was trapped in ice (Riebeek, 2005). As the amount of ice gives strong evidence of the global temperature scientists can use information from foraminifera to estimate the global temperature millions of years back in time.

1.2 Classification of foraminifera

Due to the information that can be deduced from assemblages of foraminifera, statistical counting and classification of species is an important source of data. The work of picking, counting, identifying and classifying foraminifera is very time consuming and demands great resources. This job is performed manually using microscopes by trained expertise, as the foraminifera species are often difficult to tell

apart. In classification different species are recognized due to their morphological differences, such as shape, texture and gloss. Often they must be examined from different sides to determine their correct class, or even distinguish them from grains of sand.

Machine learning models based on deep learning has shown promising results in the progress of automating this manual classification process. Johansen and Sørensen (2020) provided a new labeled dataset (see figure 1.1) of four high-level classes: sediment grains, calcareous benthic, planktic and agglutinated benthic foraminifera. Using a large convolutional neural network (CNN) they managed to classify the samples with an accuacy of 98.5 %. Through Monte Carlo dropout the researchers uncovered the difficult cases in the dataset, and found that there were two scenarios: (1) The model was uncertain about the prediction, or (2) the model was certain, but the prediction was incorrect. A goal moving forward in this area of research is to improve the overall classification accuracy and to improve edge case classification and confidence.

1.2.1 Improving classification using generative models

As classification of foraminifera is of great importance to researchers steps towards an automatic classification procedure is highly desirable. Improvements to the current deep learning models could be an increase of classification accuracy and a reduction of edge case uncertainty. In addition to extending the domain of current classifiers to include e.g. foraminifera subspecies or micro plastics.

To succeed in the aforementioned improvements two possible strategies are proposed: (i) to fine tune, improve and extend the current classification models and (ii) to get access to more numerous and diverse training data. It is likely that the solution involves a combination of these two strategies. This thesis will follow the strategy of (ii) and try to synthetically produce relevant training data that could be used to improve the existing classification model of Johansen and Sørensen (2020).

In the recent years a new type generative model, and training approach, has had huge success and was described by Facebook's AI research director Yann LeCun as *"[...] the most interesting idea in the last 10 years in ML"* (LeCun, 2016). The novelty was the generative adversarial network (Goodfellow *et al.*, 2014) where two deep learning models are trained as opponents to produce synthetic data from a given distribution. This approach is called adversarial training and is used in this thesis to create synthetic images of foraminifera that could be used to augment the foraminifera dataset and thus improve classification.

This approach has in recent years been used to improve CNN-based classification of medical images where numerous and diverse training data are known to be

scarce. Concrete applications that have yielded good results are e.g. liver lesion classification (Frid-Adar *et al.*, 2018), detection of brain tumors from MRI-images (Bowles *et al.*, 2018) and generation of synthetic PET images of Alzheimer's decease at different stages (Islam and Zhang, 2020).

1.3 Contributions

This thesis sets out to contribute to the research towards developing an automatic foraminifera classifier by exploring GANs and their ability to synthetically generate images. The key contributions of this thesis are:

- An in-depth review of the recent advances in the field of generative adversarial networks.
- Novel insights on the utilization of GANs to synthetically generate images of foraminifera and sediments.
- Improved accuracy on the classification of foraminifera by GAN-based image augmentation.
- An expert assessment of synthetically generated images of foraminifera.
- Visualization of over- and underfitting in a GAN, and artifacts that may occur on images of foraminifera.
- Experimental results that suggest the instability of Fréchet Inception distance for evaluation of GAN images produced from small datasets.
- A novel implementation in `Tensorflow 2.1` of a multi-scale gradient GAN (Karnewar and Iyengar, 2019) based on the progressively growing architecture (Karras *et al.*, 2017).

1.4 Motivation and hypothesis

The motivation for this thesis emerges from the recent advances in the field of deep learning, particularly the models referred to as generative adversarial networks (GANs). Promising results from e.g. medical applications have suggested GANs have the ability to improve CNN-based classification models by generating additional synthetic training data. This approach seems promising to further improve the classification of fossil foraminifera.

In addition, as this is a thesis in education, another aim is to explore a subject that is relevant for the Norwegian school system. In Norway from the fall of 2020 a new

curriculum takes effect, and with it comes a new focus on algorithms and programming in mathematics education. Students are to have insight to how mathematics are used in a day to day basis, in society, science and technology (Norwegian Ministry of Education and Research, 2019). They shall have knowledge to judge how algorithms are used in society in a critical manner. As deep learning systems, and application of GANs such as deepfakes[1], are becoming more integrated in technology and society, knowledge about this topic is of relevance to the Norwegian school system.

With this motivation in mind the threefold objective of this thesis is presented.

1. To explore and study the branch of deep learning models concerning generative adversarial networks (GANs), by presenting the key aspects and challenges of these models and how these challenges can be addressed.
2. To create synthetic images of foraminifera by using a generative adversarial network to learn the distribution of the foraminifera image dataset.
3. To improve the accuracy of the classification model proposed in (Johansen and Sørensen, 2020) by using synthetic images retrieved through adversarial training.

With this in mind the following hypothesis is proposed for the main experiment in this thesis:

> Augmenting the training set of the foraminifera classifier (Johansen and Sørensen, 2020) with synthetic images from a generative adversarial network will improve the classification accuracy of the model.

1.5 Thesis outline

This thesis consists of five chapters: 1 Introduction, 2 Background theory, 3 Generative adversarial networks, 4 Experiments and 5 Final discussion and concluding remarks.

Chapter 1 presents the context, research task, motivation and objectives of this thesis.

Chapter 2 gives the reader an introduction to the field of deep learning, image data, the task and techniques of computer vision and the research that this thesis builds on. This background is essential prerequisite knowledge for chapter 3 that

[1]Deepfakes are synthetic media where a person in an image or video is replaced with someone else using deep learning algorithms. The result is e.g. a highly realistic video of a president saying or doing the actions of another person: `https://youtu.be/cQ54GDm1eL0`

aims to study and explore the generative adversarial networks that are used in the experiments.

Chapter 3 addresses the first objective of this thesis by reviewing and exploring the advances in the field of GANs. The study is conducted from a mainly theoretical and technical point of view and covers GANs from their introduction in 2014 to the modern state-of-the-art models. The chapter presents some key insights and challenges of the models, as well as how some of these challenges can be addressed. The chapter closes by presenting some of today's popular techniques of evaluating GAN images. This chapter constitutes large parts of the methodology that is used in the experiments.

Chapter 4 is built up by conducting subsequent experiments that on their own address the objectives of this thesis, as well as serving as intermediate steps towards testing the final hypothesis of this thesis. Each experiment is introduced with their own intermediate objective or hypothesis, before the experimental setup and results are described. Each experiment is rounded off with a discussion and some closing remarks that constitute the foundation for the succeeding experiment.

The experiments begin with a continuation of the study and exploration of GANs, but now from a more empirical standpoint. Section 4.1 illustrates the rise and fall of the first popular deep convolutional GAN architecture, before several experiments using a more robust multi-scale gradient GAN (MSG-GAN) are conducted. Section 4.2 describes the implementation of the MSG-GAN in `Tensorflow 2.1` that is used in the following experiments. Section 4.3 aims at finding the best configuration of the MSG-GAN by testing it on a familiar dataset from the GAN literature. The experiment in section 4.4 use the MSG-GAN to generate synthetic images of foraminifera unconditionally, while experiment 4.5 aims at generating synthetic images of foraminifera class conditionally. This experiment leads to valuable insights of the reliability of GAN evaluation measures. Thereafter two experiments are then conducted to investigate the artifacts (section 4.6) and quality (section 4.7) of the generated images. All experiments lead up to the final experiment in section 4.8 that will test the main hypothesis.

Chapter 5 provides a summary and final discussion of the experiments, suggestions for future work and some concluding remarks.

Chapter 2

Background theory

Before the objectives of this thesis can be addressed, and experiments to test the hypothesis can be conducted, it is essential with a solid theoretical background. This chapter presents some key insights to the field of deep learning and provides the theoretical background needed to explore generative adversarial networks in chapter 3 and 4.

This chapter of theoretical background starts off with some basic principles of machine learning. The statistics and machine learning basics presented in this chapter will not be a complete walk-through, but rather a reminder of some of the important intuitions and results. Proceeding in the chapter the fundamentals of deep learning are presented before the task of computer vision is introduced. The background theory of deep learning and computer vision will provide the necessary prerequisites for working with images, classification and generative adversarial networks.

2.1 Machine learning basics

Machine learning (ML) is the study of computer algorithms that improve automatically by processing data or through experience (Mitchell *et al.*, 1997). It is considered a subfield of **artificial intelligence** (AI) and builds on mathematical and statistical methods. Common problems for machine learning algorithms are related to classification, inference, prediction, segmentation and automatic decision making. A common strategy of machine learning is to analyze a set of observed samples, often referred to as **training data**, with a pattern recognizing model that gain insight to solve the specific problem. More advanced machine learning techniques use complex models of **deep learning** to solve problems of e.g. computer

vision, natural language processing or image generation. More on these models later.

2.1.1 Maximum likelihood

To introduce the notation and some important concepts the principle of maximum likelihood is briefly introduced. **Maximum likelihood** is an important principle when the objective is to find a function that are can estimate a data generating distribution $p_{\text{data}}(\mathbf{x})$.

Consider a set $\mathbb{X} = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n\}$ of independent variables sampled from p_{data}. Using these observed values one wishes to approximate the data generating distribution p_{data} using a parameterized model $p_{\text{model}}(\mathbf{x}; \boldsymbol{\theta})$. The objective is that the model shall estimate the true probability of any observed variables $\boldsymbol{x}$, so $p_{\text{model}}(\boldsymbol{x}; \boldsymbol{\theta}) \to p_{\text{data}}(\boldsymbol{x})$. A suitable model can solve this problem given the right parameter values.

The principle of maximum likelihood suggests to choose values of $\boldsymbol{\theta}$ so the probability of observing the samples in $\mathbb{X}$ are maximized given the model that is chosen. This gives the maximum likelihood parameters $\boldsymbol{\theta}_{\text{ML}}$

$$\boldsymbol{\theta}_{\text{ML}} = \arg\max_{\boldsymbol{\theta}} p_{\text{model}}(\mathbb{X}; \boldsymbol{\theta}) \tag{2.1}$$

$$= \arg\max_{\boldsymbol{\theta}} \prod_{i=1}^{n} p_{\text{model}}(\boldsymbol{x}_i; \boldsymbol{\theta}) \tag{2.2}$$

In practice $\boldsymbol{\theta}_{\text{ML}}$ is often found using an optimization algorithm to solve the equivalent log-transformed problem, so the product of the probabilities conveniently are transformed to the sum of log-probabilities.

$$\boldsymbol{\theta}_{\text{ML}} = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{n} \log p_{\text{model}}(\boldsymbol{x}_i; \boldsymbol{\theta}) \tag{2.3}$$

Scaling the optimization problem by $\frac{1}{n}$ does not change the solution, but lets us express it as an expectation with respect to the observed distribution $\hat{p}_{\text{data}}$ from $\mathbb{X}$ (Goodfellow *et al.*, 2016), so

$$\boldsymbol{\theta}_{\text{ML}} = \arg\max_{\boldsymbol{\theta}} \mathbb{E}_{\mathbf{x} \sim \hat{p}_{\text{data}}} [\log p_{\text{model}}(\boldsymbol{x}; \boldsymbol{\theta})] \tag{2.4}$$

The optimization problem of maximum likelihood can be interpreted as minimizing the dissimilarity between the empirical distribution $\hat{p}_{\text{data}}$ and the model distribution

p_{model}, when the dissimilarity is measured by the Kullback-Lieber (KL) divergence[1].

$$D_{\text{KL}}(\hat{p}_{\text{data}} \| p_{\text{model}}) = \mathbb{E}_{\mathbf{x} \sim \hat{p}_{\text{data}}}[\log \hat{p}_{\text{data}}(\boldsymbol{x}) - \log p_{\text{model}}(\boldsymbol{x})] \tag{2.5}$$

By convention optimization problems in machine learning are often formulated as a minimization of a loss or cost function. Following this convention the problem of finding the maximum likehood estimator of equation 2.4 is the same as minimizing the cost function

$$J(\boldsymbol{\theta}) = -\mathbb{E}_{\mathbf{x} \sim \hat{p}_{\text{data}}}[\log p_{\text{model}}(\boldsymbol{x})] \tag{2.6}$$

2.1.2 Gradient descent

The principle of maximum likelihood can be applied to obtain a cost function to minimize when the goal is to fit a model $f(\boldsymbol{x}; \boldsymbol{\theta})$ to a training set. Machine learning algorithms often use gradient based methods to find the minimum of a cost function $J(\boldsymbol{\theta})$. Given that the derivative exists local extrema can be found by solving $\frac{\partial J(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} = 0$.

Often these solutions must be found iteratively using an optimization algorithm. A simple optimization algorithm that tries to find a minimum of J is **gradient descent** that use the average gradient of the cost function, evaluated at all training samples, with respect to the parameters $\boldsymbol{\theta}$.

In practice it is more suitable to compute the average gradient of a **loss function** L evaluated on small subsets of samples $\mathbb{B} = \{\boldsymbol{x}_i\}_1^b$ of training data, with respect to the parameters $\boldsymbol{\theta}$ of the model. This method is known as **stochastic gradient descent** (SGD), and each subset of samples is known as a **minibatch**. The **loss** is the cost function evaluated on a minibatch of training samples. The gradients of the loss function gives the slope of L, and hence map out the local topology of the loss surface. Following the gradients in the negative direction usually gives a reasonable path down towards a minimum of the cost function. The parameters $\boldsymbol{\theta}$ of the model are updated by adding a fraction μ of the gradients of the loss function. The parameter updates of SGD are given by

$$\hat{\boldsymbol{g}} = \frac{1}{b} \sum_{i=1}^{b} \nabla_{\boldsymbol{\theta}^{(t)}} L(f(\boldsymbol{x}_i; \boldsymbol{\theta}^{(t)})) \tag{2.7}$$

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \mu \hat{\boldsymbol{g}} \tag{2.8}$$

where $\hat{\boldsymbol{g}}$ is the estimated gradient of iteration t of the minibatch $\mathbb{B}$, and μ is the **learning rate** which give how large step the algorithm should take in the negative direction of the gradients.

[1]Minimizing the KL divergence is the same as minimizing the cross-entropy between the distributions (Goodfellow *et al.*, 2016).

2.1.3 Momentum and Adam

Optimization and learning of the parameters using SGD can often be slow and unstable. Common culprits are the stochasticity introduced by the random sampling of minibatches and small or vanishing gradients of the loss function.

To improve the optimization equation 2.8 can be modified to speed up convergence. Some common measures are **adaptive learning rate** and **momentum** (Qian, 1999). Conceptually adaptive learning rate adjusts μ during optimization so bigger steps are taken when the algorithm performs well (Theodoridis and Koutroumbas, 2008). Momentum accelerates learning by adding a fraction of the previous parameter update to the current update, so oscillations in optimization are dampened.

These principles for faster convergence have been improved and adopted in more advanced optimizing algorithms such as *adaptive gradient algorithm* **AdaGrad** (Duchi *et al.*, 2011) and *root mean square propagation* **RMSProp** (Tieleman and Hinton, 2012).

The optimizer algorithm that has become one of the most popular is the *adaptive moments estimation* optimizer **Adam** (Kingma and Ba, 2014). The Adam optimizer calculates the running average of the first moment (mean) and the second moment (uncentered variance) of the gradients (equation 2.7), performs correction of bias before updating the parameters. Optimization through Adam is summarized by the following operations performed elementwise on each minibatch:

Estimating the first and second moment

$$\boldsymbol{m}^{(t+1)} = \beta_1 \boldsymbol{m}^{(t)} + (1 - \beta_1)\hat{\boldsymbol{g}} \tag{2.9}$$

$$\boldsymbol{v}^{(t+1)} = \beta_2 \boldsymbol{v}^{(t)} + (1 - \beta_2)\hat{\boldsymbol{g}} \odot \hat{\boldsymbol{g}} \tag{2.10}$$

performing correction of bias

$$\hat{\boldsymbol{m}}^{(t)} = \frac{\boldsymbol{m}^{(t)}}{1 - \beta_1^{(t)}} \tag{2.11}$$

$$\hat{\boldsymbol{v}}^{(t)} = \frac{\boldsymbol{v}^{(t)}}{1 - \beta_2^{(t)}} \tag{2.12}$$

and updating the parameters

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \mu \frac{\hat{\boldsymbol{m}}^{(t)}}{\sqrt{\hat{\boldsymbol{v}}^{(t)} + \epsilon}} \tag{2.13}$$

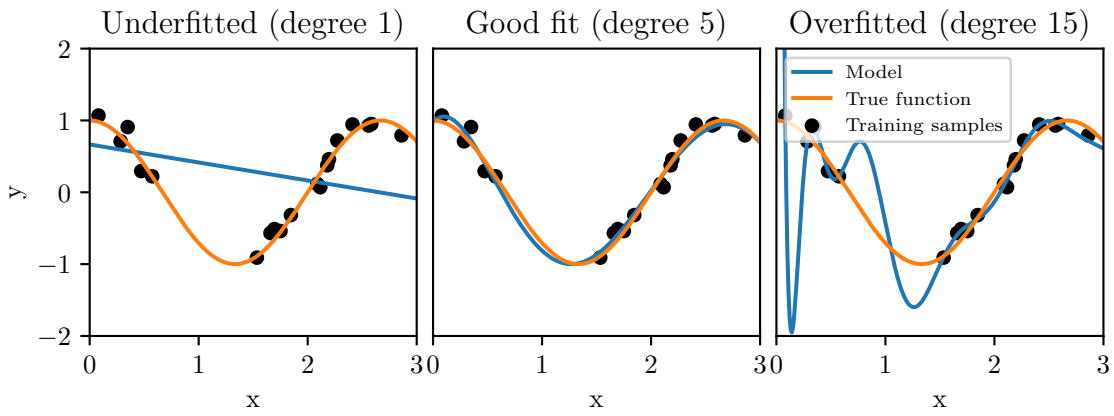


Figure 2.1: Polynomial regressions of different degrees are used to model the function $y = \cos(\frac{3\pi}{4}x)$. Degree 1 have too low, and degree 15 have too large capacity, so it underfits and overfits to the training data. The result is a model that will not generalize well. A polynomial of degree 5 is suitable and models the true function well.

Where β_1 and β_2 are the exponential decay rates for the moment estimates $\boldsymbol{m}$ and $\boldsymbol{v}$ in the range $[0, 1)$, and ϵ is a small constant for numeric stability. Kingma and Ba (2014) suggest the values $\beta_1 = 0.999$, $\beta_2 = 0.9$ and $\epsilon = 10^{-8}$ as default for the Adam algorithm.

2.1.4 Overfitting and underfitting

Though a model is optimized using a training set it is not guaranteed that it will **generalize** well to unseen data. If a model has too large **capacity** chances are that the model learns the training data, and not the underlying structure that it is supposed to model (figure 2.1). This is a common problem in machine learning and is referred to as **overfitting**. If the model has too small capacity or is not trained sufficiently it will not be able to model the underlying structure and is **underfit**.

The problem of overfitting can be discovered by calculating the **generalization error** also known as **test error** by testing the model's performance on data samples than was collected separately from the training data. When one dataset is used for training it is often separated into a **training split** and **test split** to reliably test the model's performance on separate samples. Some popular training techniques require an additional **validation split**, but more on this later (section 2.4.1).

```
[[  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0  82 187  26   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   1   0   0   1   0   0 179 240 237 255 240 139  83  64  43  60  54   0   1]
 [  0   0   0   0   0   0   0   0   0   1   0   0   1   0  58 239 222 234 238 246 252 254 255 248 255 187   0   0]
 [  0   0   0   0   0   0   0   0   0   0   2   3   0   0 194 239 226 237 235 232 230 234 234 233 249 171   0   0]
 [  0   0   0   0   0   0   0   0   0   1   1   0   0  10 255 226 242 239 238 239 240 239 242 238 248 192   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0 172 245 229 240 241 240 241 243 243 241 227 250 209   0   0]
 [  0   0   0   0   0   0   0   0   0   6   5   0  62 255 230 236 239 241 242 241 242 242 238 238 242 253   0   0]
 [  0   0   0   0   0   0   0   0   0   3   0   0 255 235 228 244 241 241 244 243 243 244 243 239 235 255  22   0]
 [  0   0   0   0   0   0   0   0   0   0   0 246 228 220 245 243 237 241 242 242 242 243 239 237 235 253 106   0]
 [  0   0   3   4   4   2   1   0   0  18 243 228 231 241 243 237 238 242 241 240 240 240 235 237 236 246 234   0]
 [  1   0   0   0   0   0   0   0  22 255 238 227 238 239 237 241 241 237 236 238 239 239 239 239 239 237 255   0]
 [  0   0   0   0   0  25  83 168 255 225 225 235 228 230 227 225 227 231 232 237 240 236 238 239 239 235 251  62]
 [  0 165 225 220 224 255 255 233 229 223 227 228 231 232 235 237 233 230 228 230 233 232 235 233 234 235 255  58]
 [ 52 251 221 226 227 225 225 225 226 226 225 227 231 229 232 239 245 250 251 252 254 254 252 254 252 235 255   0]
 [ 31 208 230 233 233 237 236 236 241 235 241 247 251 254 242 236 233 227 219 202 193 189 186 181 171 165 190  42]
 [ 77 199 172 188 199 202 218 219 220 229 234 222 213 209 207 210 203 184 152 171 165 162 162 167 168 157 192  78]
 [  0  45 101 140 159 174 182 186 185 188 195 197 188 175 133  70  19   0   0 209 231 218 222 224 227 217 229  93]
 [  0   0   0   0   0   0   2  24  37  45  32  18  11   0   0   0   0   0   0  72  51  53  37  34  29  31   5   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0]]
```

Figure 2.2: Shows how a 28×28 gray-scale image of a shoe can be represented as values in a matrix. Each entry in the matrix is the pixel value $[0, 255]$ that gives the amount of white in the image. The image data is from the fashion MNIST dataset (Xiao *et al.*, 2017).

2.1.5 Image data

As this thesis comprises image data it can be useful with a clarification on what image data is to a computer. An image consists of pixels composed in a grid. In a gray-scale image every pixel is usually represented by an integer value in the range of $[0, 255]$. The value represents the luminance of each pixel. The grid of pixel values in a gray-scale image are often represented in a matrix $\boldsymbol{A}$, where each entry $A_{i,j}$ corresponds to the pixel value in row i and column j (see figure 2.2).

In color images (RGB), every pixel is represented using three numbers. The numbers (r, g, b) are in the range $[0, 255]$ and specifies the amount of red, green and blue light respectively. Mixing the proportions of red, green and blue (RGB) allows it to display $256^3 = 16\,777\,216$ different colors. The numbers describing one pixel in an image are often referred to as **channels**. In an RGB image a 3-D matrix is needed to represent the numbers. Multi-dimensional matrices are sometimes referred to as **tensors** and are denoted A. In figure 2.3 the 2-D tensor slices $\mathsf{A}_{:,:,1}$, $\mathsf{A}_{:,:,2}$ and $\mathsf{A}_{:,:,3}$ yielding the red, green and blue channels respectively are displayed along with the complete image tensor A. Even though color images typically are represented by 3-D tensors, they are still considered to be a 2-D data types. More specifically, multi-channel 2-D data (Goodfellow *et al.*, 2016). The 3-D equivalent

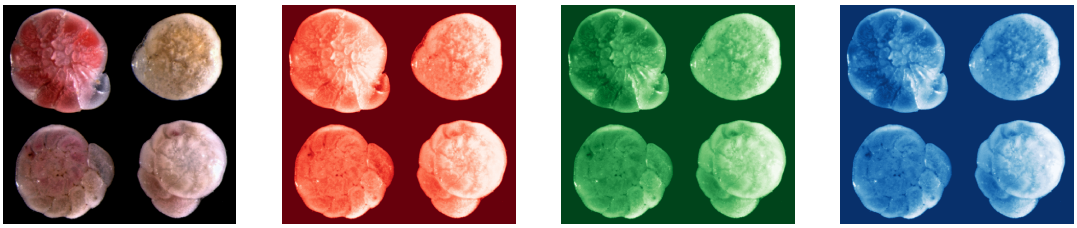


Figure 2.3: RGB image of benthic foraminifera (Ammonia beccarii) from the North Sea. The three "layers" of red, green and blue channels are shown to the right. These three together make up the complete RGB color image. The original image to the left is borrowed from (Commons, 2013)

.

of multi-channel color images could be video data, where the height and width of each video frame give the two first axis and time the last axis.

2.2 Neural networks

Neural networks are at the heart of deep learning, and can be considered the most fundamental architecture of the generative models used in the experiments in chapter 4. Neural networks are introduced by motivating and presenting their basic building block, the **perceptron**, before extending the concept to multilayer perceptrons and convolutional neural networks.

2.2.1 The perceptron

The perceptron have been developed partially after inspiration from the human brain and its biological neurons that transmits electric signals. This section presents some of the intuition and biological motivation that modern perceptrons and artificial neural networks originates from.

2.2.1.1 A mathematical model of the biological neuron

The human brain consists of more than 86 billions neurons that are connected in a large network. These **biological neurons** are electrically excitable cells that propagates signals to other neurons using connections called **synapses**. The neurons propagate the electrical signal forward to other neurons if the stimulus above a certain threshold.

This neuroscientific model of a biological neuron has inspired the **perceptron**, a simple mathematical model that have become the basic building block of modern

deep learning models. The predecessor of the perceptron were simple **linear functions** that associate a set of n input values $x_1, x_2, \ldots, x_n$ with an output y. To perform this mapping the model would learn a set of weights $w_1, w_2, \ldots, w_n$ using e.g. maximum likelihood (section 2.1.1). The model computes the output $f(\boldsymbol{x}, \boldsymbol{w}) = x_1 w_1 + x_2 w_2 + \cdots + x_n w_n$.

McCulloch and Pitts (1943) proposed a mathematical model of brain function that could perform binary classification by testing whether $f(\boldsymbol{x}, \boldsymbol{w})$ was positive or negative. The test was done using a step function, and the weights were set manually. This model, often referred to as the **McCulloch-Pitts neuron**, was important inspiration for the modern perceptron. Rosenblatt (1958, 1962) introduced a similar model that could learn the weights needed to perform the classification automatically.

2.2.1.2 The modern perceptron

The modern perceptron is a simple mathematical function that serves as the fundamental building block in the neural networks used in deep learning. The perceptron performs an inner product operation between an input vector $\boldsymbol{x}$ and parameter weights $\boldsymbol{w}$. A bias term b is added to the inner product to create a **potential** that is no longer bounded to the origin[2]. The bias can be thought of as the threshold needed to propagate a signal forward (section 2.2.1.1). The potential is usually evaluated using a nonlinear function $g(\cdot)$ known as the **activation function**. The output of the activation function is often referred to as the **activation**. The operation performed by the perceptron is shown in equation 2.14 and figure 2.4.

$$g(\boldsymbol{w}^\top \boldsymbol{x} + b) = y \tag{2.14}$$

A common type of activation function in simple perceptrons are the continuous and differentiable functions from the family of **sigmoid functions**. When replacing the step function with e.g. the **logistic sigmoid function**

$$\sigma(x) = \frac{1}{1 + \exp(-x)} \tag{2.15}$$

the input x gets squished to the range of $(0, 1)$. If the perceptron is used for binary classification, and output of the is manipulated to be either 1 (true class) or 0 (false class), using the sigmoid function as a nonlinear lets us interpret the output as the probability of an input $\boldsymbol{x}$ being of class 1. This result can also be motivated using the logistic regression model for classification (Alpaydin, 2014).

[2]In the context of linear functions (section 2.2.1.1) the bias b states where the hyperplane will intersect the axis when $\boldsymbol{x} = 0$.

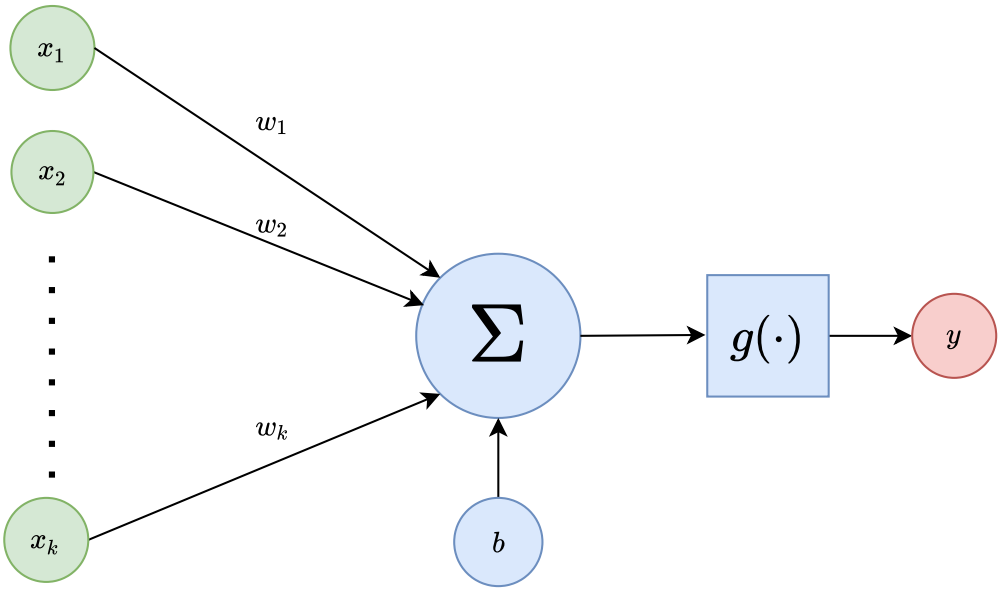


Figure 2.4: Shows the operations of a basic perceptron. The inputs $\boldsymbol{x}$ are multiplied with the weights $\boldsymbol{w}$ and summed with a bias b before it is sent through a nonlinear activation function g to produce the output y.

The basic perceptron is essentially a linear function that sends its potential through a nonlinear activation function. Multiple of these simple models can be combined and stacked in layers to form arbitrarily large **neural networks** of connected perceptrons. The stacked perceptrons pass their activations forward to form to create a powerful and versatile model called a **feedforward neural network** or **multilayer perceptron** (MLP).

2.2.2 Feedforward neural network

The multilayer perceptron is the base architecture of deep learning. It is often referred to as a neural network due to how the stacked perceptrons (neurons) are interconnected. The network can be considered a function f that approximates a function f^*. The non-linearity introduced by the activation function in each perceptron makes it possible for a large enough MLP to represent any function (Goodfellow *et al.*, 2016). For example, if $y = f^*(\boldsymbol{x})$ is a classifier that performs the mapping of the input $\boldsymbol{x}$ to a category y, the network that defines $\boldsymbol{y} = f(\boldsymbol{x}; \boldsymbol{\theta})$ is a function approximator that learns the values of $\boldsymbol{\theta}$ to give similar results as $f^*(\boldsymbol{x})$.

Stripped down to its core components it consists of an **input layer**, **hidden layers** and an **output layer**. See figure 2.5. In the input layer a vector $\boldsymbol{x}^{(0)}$ with n features is passed into the MLP. The input layer is connected to the first

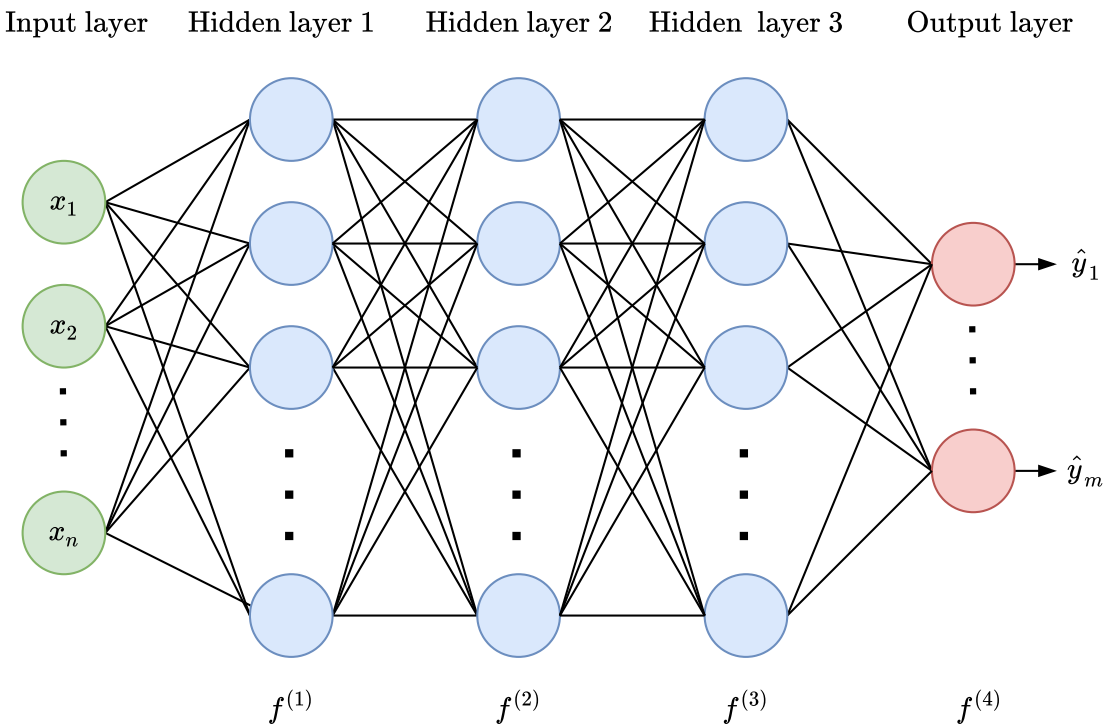


Figure 2.5: A multilayer perceptron with n inputs, 3 hidden layers and m outputs. The inputs $\boldsymbol{x}^{(0)} = [x_1, x_2, \ldots x_n]$ are sent through the hidden layers. Every layer produces activations that are propagated forward. The final layer produces the output $\hat{\boldsymbol{y}} = [\hat{y}_1, \ldots, \hat{y}_2]$.

layer, $f^{(1)}$, which consists of stacked perceptrons referred to as **units**. Every unit has n weights $\boldsymbol{w}$ and a bias b. The units of the first layer are connected to the second layer, $f^{(2)}$. Every connection between the units in layer $f^{(l)}$ and $f^{(l-1)}$ in figure 2.5 represents the weights that are multiplied with the activations from layer $(l-1)$ used to compute the potential in each unit in layer l. All the layers between the input layer and the output layer is referred to as **hidden layers** because the desired output of these layers are not specified by the training data. The output layer yields m activations that constructs a vector $\hat{\boldsymbol{y}}$ that gives the output of the network so $\hat{\boldsymbol{y}} \approx f^*(\boldsymbol{x})$.

Every layer $f^{(l)}$ in the network processes an input $\boldsymbol{x}^{(l-1)}$ by performing the operation described in equation 2.14 and figure 2.4 with the weights $\boldsymbol{w}$ and bias b of every unit. These are the trainable parameters and they are collected in $\boldsymbol{\theta}^{(l)} = \{\boldsymbol{W}^{(l)}, \boldsymbol{b}^{(l)}\}$ so the operations of equation 2.14 can be performed in parallel in every layer. In general layer l performs

$$f^{(l)}(\boldsymbol{x}^{(l-1)}; \boldsymbol{\theta}^{(l)}) = g^{(l)}(\boldsymbol{W}^{(l)}\boldsymbol{x}^{(l-1)} + \boldsymbol{b}^{(l)}), \quad l = 1, \ldots, L \tag{2.16}$$

Where $\boldsymbol{W}^{(l)} \in \mathbb{R}^{k_l \times k_{l-1}}$, $\boldsymbol{b}^{(l)} \in \mathbb{R}^{k_l}$ and $g^{(l)}(\cdot)$ is the vector-valued activation function of layer l. The specified input $\boldsymbol{x}^{(0)}$ give the input dimension k_0. The number of layers L is referred to as the **depth** of the network.

As every layer $f^{(l)}$ acts as a function (equation 2.16) that propagates its output to the next layer, the whole network $f(\boldsymbol{x}; \boldsymbol{\theta})$ can be represented by a nested function. Let the composite function be given by $\circ$ so $f^{(2)}(f^{(1)}(\boldsymbol{x})) = f^{(2)} \circ f^{(1)}$. The general feedforward neural networks is then

$$f(\boldsymbol{x}; \boldsymbol{\theta}) = f^{(L)} \circ f^{(L-1)} \circ \cdots \circ f^{(2)} \circ f^{(1)}(\boldsymbol{x}) \tag{2.17}$$

2.2.3 Learning the parameters

For the feedforward neural network to give fruitful results it must learn the set of trainable parameters $\boldsymbol{\theta} = \{\boldsymbol{W}^{(l)}, \boldsymbol{b}^{(l)}\}_{l=1}^{L}$ that is needed to compute output like $f^*(\boldsymbol{x})$. The learning of these parameters are done through an iterative optimization procedure often referred to as **training**.

When the training is supervised the network is provided with data pairs from a training set $\mathbb{X} = \{\boldsymbol{x}_i, \boldsymbol{y}_i\}_1^N$. $\boldsymbol{x}_i$ is a vector with features (explanatory variables) and $\boldsymbol{y}_i$ is the desired output for the given feature vector. The idea behind the training procedure is to compare the output of the network $f(\boldsymbol{x}; \boldsymbol{\theta}) = \hat{\boldsymbol{y}}$ with the ideal output of $f^*(\boldsymbol{x}) = \boldsymbol{y}$ to find out how the network should change $\boldsymbol{\theta}$ to make the $\hat{\boldsymbol{y}}$ more similar to $\boldsymbol{y}$. To do this a **loss function** typically quantifies a dissimilarity measure between the two outputs, and the parameters of the network are updated using a gradient based optimization algorithm.

The loss function of the network must be determined for the specific problem the model should solve. Often the goal of the a network is to model a probability distribution p_{data} that produced the set of training examples $\mathbb{X}$. Given the training pairs the distribution to model is $p(\boldsymbol{y}|\boldsymbol{x};\boldsymbol{\theta})$, so finding the parameters $\boldsymbol{\theta}$ can be done by the principle of maximum likelihood. In this scenario the cross-entropy between the training data and the model's prediction becomes the objective function to minimize (Goodfellow *et al.*, 2016).

$$J(\boldsymbol{\theta}) = -\mathbb{E}_{\mathbf{x}\sim\hat{p}_{\text{data}}}[\log f(\boldsymbol{x};\boldsymbol{\theta})] \tag{2.18}$$

To minimize the objective function in equation 2.18 the gradients $\nabla_{\boldsymbol{\theta}} J$ should be computed and used to update the weights according to gradient descent or another optimization algorithm like SGD or Adam. The most common procedure of computing these gradients is the **backpropagation algorithm**. The details of the algorithm is beyond the scope of this thesis, but it is derived in detail and for the general case in Goodfellow *et al.* (2016, p. 204-218). The intuition behind the algorithm is to recursively apply the chain rule of calculus to the objective function (e.g. equation 2.16) obtaining the gradients of the weights and biases of every layer in the neural network. Once the gradients $\nabla_{\boldsymbol{\theta}} J$ are obtained for the training set[3] $\mathbb{X}$ they are used to update the parameters $\boldsymbol{\theta}$. Using gradient descent the update becomes

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \mu \nabla_{\boldsymbol{\theta}^{(t)}} J \tag{2.19}$$

where μ is the learning rate that is used to scale the gradients so the parameter update only is a small step in the negative direction of the gradient.

2.3 Convolutional neural networks

A **convolutional neural network** (CNN) (LeCun *et al.*, 1989) is a feedforward neural network where the inner product operation of one or more of the perceptron layers are replaced by convolution operations. Before going into the details it is useful with some motivation for the CNN.

2.3.1 The biology of computer vision

When looking at the raw data of the shoe image in figure 2.2 it is possible to recognize the contours of a shoe even though you are looking at a grid of numbers. The biology of human vision makes it possible to recognize edges and some degree of texture, so the shoe image can be comprehended, even when it is displayed as raw data. In computer vision it has become essential to have models that can

[3]More commonly a minibatch $\mathbb{B}$ of the training set when using SGD or Adam

detect features such as edges, textures and colors when they examine raw data, so the models in turn can interpreted them as more complex shapes and objects. This hierarchical approach of tackling computer vision have been inspired by biology of vision (Goodfellow *et al.*, 2016; Bouvrie, 2006).

Images are perceived when light hits the retina – the light sensitive tissue in the back of the eye. Hubel and Wiesel (1959) investigated the visual system of cats and discovered that neurons early in the cat's visual system responded strongly to vertical, horizontal and oblique light patterns. Recognizing simple shapes and textures early in a computer vision model is clever because the simple patterns can be composed together to more complex patterns later in the model. This is the goal of using **convolutional layers** in a feedforward neural network.

2.3.2 The convolutional operator

The basis for the **convolutional neural network** (CNN) is the **convolution operator**. The convolution operator is useful for pattern recognition on grid-like data such as time series (1-D), images (2-D) and volumetric data (3-D) from e.g. CT scans. As this thesis concerns mostly images the focus of this section will be the two-dimensional discrete convolution operator.

The convolution operation can be thought of as passing a **filter** over an **input** producing an output referred to as the **feature map**. The filter is often referred to as a **kernel**, not to be confused with the kernels known from the "kernel trick" (Theodoridis and Koutroumbas, 2008).

Let the input $I(i, j)$ be an image, and $K(i, j)$ the kernel (filter). The discrete convolution in two dimensions is defined

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n) \tag{2.20}$$

where $*$ denotes the convolution operator. Figure 2.6 illustrates this 2-D discrete convolution operation.

Note that in practice many machine learning libraries implement a variant of the convolution operation called **cross-correlation**, and still refer to it as a convolution operation. The difference is that the kernel of equation 2.20 is flipped up-down and left-right before the element-wise multiplication. Conceptually this does not change much, and the reader is referred to Goodfellow *et al.* (2016) for details.

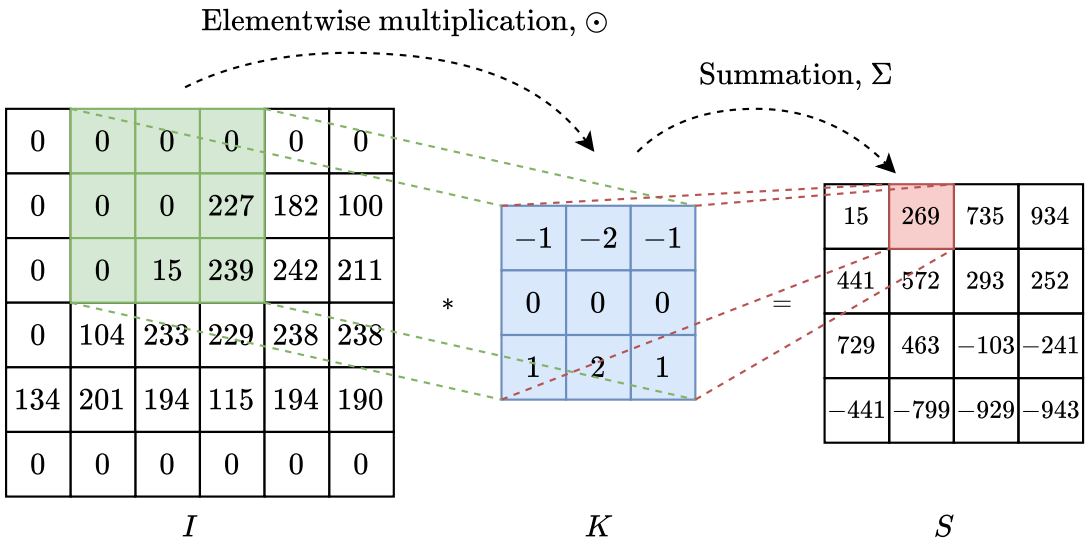


Figure 2.6: Illustration of the convolution $K * I$. The kernel $K(i, j)$ is convolved over the input image $I(i, j)$ to produce the feature map $S(i, j)$. As there is no padding around the input, this is a "valid" convolution resulting a smaller output.

2.3.3 Convolutional layers

In a convolutional layer, the operation performed on a two dimensional input in a neural network is

$$f^{(l)}(\boldsymbol{x}^{(l-1)}; \boldsymbol{\theta}^{(l)}) = g^{(l)}(\boldsymbol{K}^{(l)} * \boldsymbol{X}^{(l-1)} + \boldsymbol{b}^{(l)}) \tag{2.21}$$

where $\boldsymbol{\theta}^{(l)} = \{\boldsymbol{K}^{(l)}, \boldsymbol{b}^{(l)}\}$. For 2-D image data (height, width) $\boldsymbol{X}^{(0)} \in \mathbb{R}^{h \times w}$, the kernel matrix has dimensions so $\boldsymbol{K} \in \mathbb{R}^{h_l \times w_l}$. In practice implementations of the convolutional operation of a layer may differ slightly from equation 2.20 and 2.21. The variations are due to the practice that convolutions often are performed over multiple channels (e.g. RGB-channels or multiple feature maps) and over batches of inputs in parallel. Details of the variations are elaborated in (Goodfellow *et al.*, 2016, p. 347-358).

Considering a simple case, convolving a filter over an image can be used to extract features of an image in a computationally efficient fashion. To illustrate this property consider convolving the following matrices over an input image:

$$\boldsymbol{K}_h = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \qquad \boldsymbol{K}_v = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \tag{2.22}$$

The matrices are filters that corresponds to respectively extracting horizontal and vertical edges of an image. Figure 2.7 illustrates the effect of filter $\boldsymbol{K}_h$ and $\boldsymbol{K}_v$

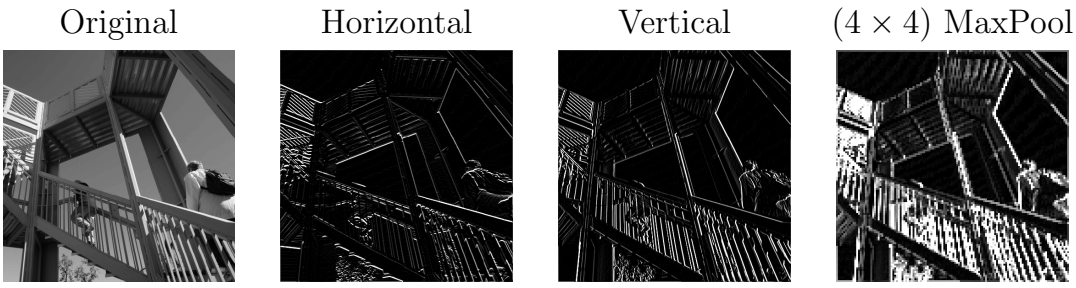


Figure 2.7: The three first images illustrate the effect of applying two simple 3×3 filters to an input image, extracting the horizontal and vertical features of the image respectively. The last image illustrate the effect of a 4×4 max pooling layer after applying the vertical filter. Photocredit Hillebrand (2016).

when applied to an input image. The results are feature maps where characteristics associated with horizontal and vertical edges are emphasized.

2.3.4 Motivation

The images of figure 2.7 are 512×512 pixels and the filters are 3×3. The operation requires $512^2 \times (9+8) = 4\,456\,448$ floating point operations (9 multiplications and 8 additions per output pixel). Producing the same result using matrix multiplication in an MLP would require $512^4 = 68\,719\,476\,736$ floating point operations. This illustrates the computational benefits of using convolutional layers.

To further emphasize the motivation for convolutional layers in feedforward neural networks their following properties (Goodfellow *et al.*, 2016) are highlighted:

- sparse interactions – filters of few parameters can be used to extract meaningful features such as edges or textures.
- parameter sharing – the same weights of the filter are used to compute multiple output values (pixels) when it convolves over an input.
- equivariance to translation – if the input changes, the output changes correspondingly

2.3.5 Pooling

One of the motivations for using convolutions are, as illustrated in figure 2.7, to extract certain features of the data. To amplify or reinforce the presence of such features pooling layers are often combined with convolutional layers.

A pooling layer can make the result of a convolutional layer become more **invariant** to small local changes. This means that the pooling layer produces a similar output even though the inputs are changed by a small amount. This is useful when *"[...] we care more about whether some feature is present than exactly where it is."* (Goodfellow *et al.*, 2016, p. 342).

A popular pooling function that achieves the aforementioned result is the **max pooling** function which extracts the maximum value of neighboring data points. Using the image in figure 2.7 as an example the max pooling function examines e.g. a 4×4 grid of the image and returns the maximum value of the pixels in the current grid. The result is a down-scaled version of the input image. In the case of a (4×4) pooling operation the output is $\frac{1}{4}$-th of the original resolution.

2.3.6 Variations of convolutional layers

To this point only the basic convolutional layer is presented, but there are many useful variations. In figure 2.8 and the following sections some popular variants will be presented briefly.

2.3.6.1 Padding

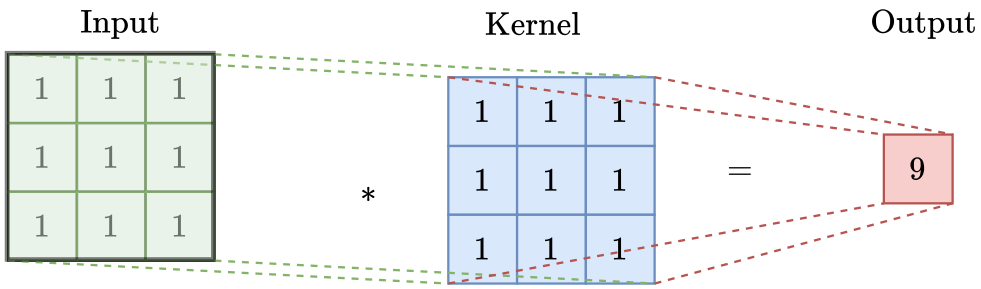
Examining the illustration in figure 2.6 it is clear that the output "image" is of a smaller resolution that the input. This output is produced by a **valid convolution**. To prevent this effect **padding** can be added to the input. The padding is usually zeros around the edges of the input image, so the output will be of the same size as the input. This is referred to as **same convolutions**.

2.3.6.2 Strides

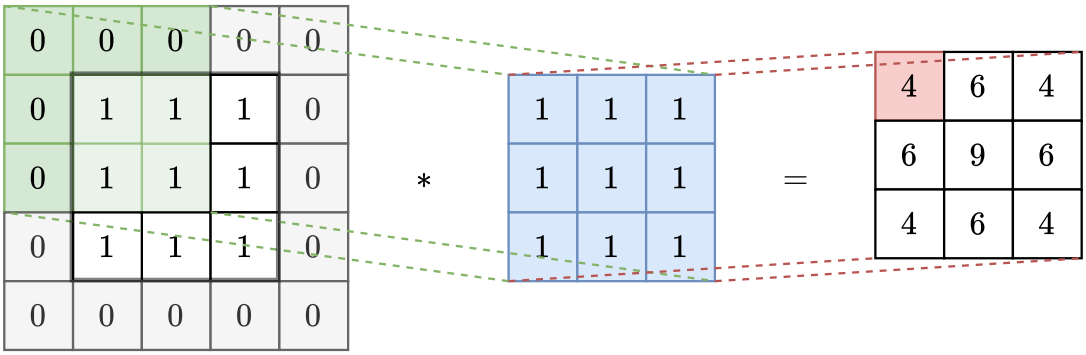
In all convolutions that are considered so far, the filter moves one pixel at the time during the convolution operation. This is referred to as a **stride** of 1. Moving the filter over multiple pixels when convolving over an image results in an output that is smaller than the input. Using a stride > 1 is useful for convolutional layers that in addition should perform a down-scaling operation, because the output produced will be smaller than the input.

2.3.7 Learning the filters

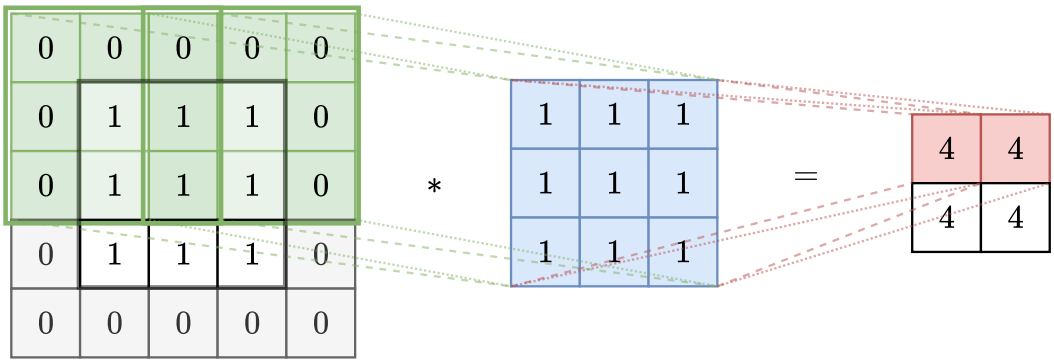
The filters of a convolutional layer are learned by the learning algorithm. Specifically that the model adapts the filters, so it learns useful features for solving the problem at hand. The learning of the filter weights can be done in a similar fashion as MLP's – by computing the gradients using backpropagation and updating the



(a) Convolution with stride 1 and "valid" padding



(b) Convolution with stride 1 and "same" padding



(c) Convolution with stride 2 and 1 padding

Figure 2.8: An illustration of three popular variants of the convolution operation. (a) "Valid" padding applies no zero-padding to the input, and the output will thus be of lower dimensionality. (b) "Same" padding adds enough padding around the edges of the input so the output is of the same dimensionality as the output. (c) A stride of 2 refers to the number of places the filter moves when convolving over the image.

weights using e.g. gradient descent. For an intuition for how this is done it can be useful to think of convolution as a linear transformation that can be described by matrix multiplication (Goodfellow *et al.*, 2016). The matrices involved are the input matrix which is unraveled to a vector $\boldsymbol{x}^*$, and a matrix $\boldsymbol{K}^*$ that is a function of the convolution kernel $\boldsymbol{K}$. When the convolution operation can be performed using matrix multiplication, $\boldsymbol{K}^*$ can also be transposed. This matrix transpose is needed to make use of the chain rule to backpropagate errors and to compute the gradients backwards in the network. Details of the backpropagation in CNNs are beyond the scope of this thesis, but can be found in Bouvrie (2006).

2.3.8 Transposed convolutions

The transposed matrix $\boldsymbol{K}^{*^\top}$ that is defined by the convolution kernel is in addition to be useful in backpropagation also used in various generative models such as autoencoders and generative adversarial networks (GANs) (Goodfellow *et al.*, 2014) that this thesis concerns (Goodfellow *et al.*, 2016).

Layers using $\boldsymbol{K}^{*^\top}$ in a convolutional layer are often referred to as **transposed convolutions**, **fractionally-strided convolutions** or **deconvolutional layers**. The name "deconvolution" is used because the operation is associated with doing the opposite of a convolutional layer, specifically going from a scalar to a matrix, while learning the filter that produces this result (figure 2.9). The term deconvolution can be misleading because transpose convolutions are not the inverse of convolutions. Therefore the more correct term "transposed convolution" will be used in this thesis to avoid this misconception.

In applications of convolutional networks where the goal is to go from a low dimensional input, or even a single vector to a higher dimensional output, there is a need for upsampling. Upsampling an image can be done using a method like nearest neighbor interpolation (Theodoridis and Koutroumbas, 2008), but might not yield sufficient results. Using transposed convolutions for upsampling allows the upsampling layer to learn useful upsampling-filters to provide better up-scaled results Goodfellow *et al.* (2016). Layers performing transposed convolutions provide a learnable upsampling that is adaptive to the specific problem.

Like normal convolutions transposed convolutions can be thought of as passing a filter over an input to produce an output. The operations that are performed is a little different, but similar. The process is displayed in figure 2.9.

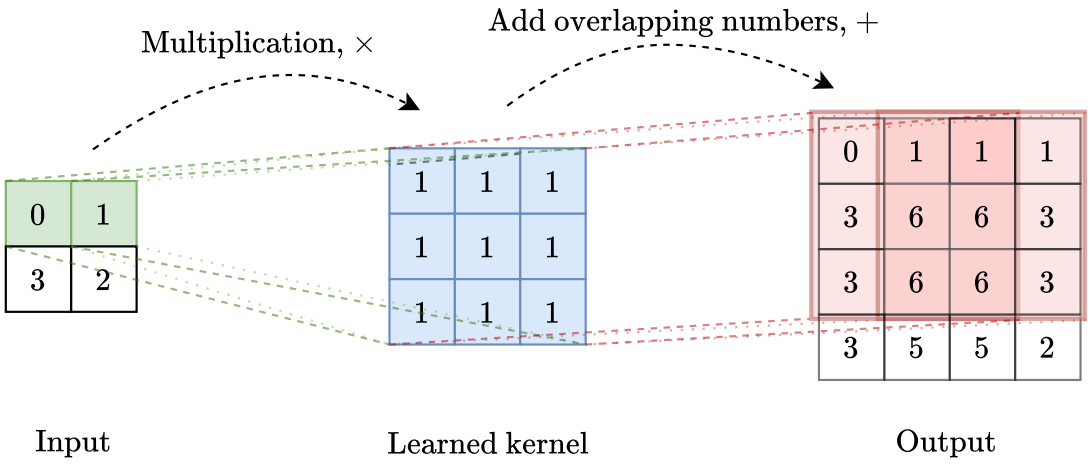


Figure 2.9: Shows the operation of a valid transpose convolution with stride of 1. Each value of the input is multiplied with the learned filter and mapped to the corresponding position of the output. The values of the overlapping areas are accumulated to produce one output value. In this example values near the edges of the output get accumulated less than the values near the center. In practice this is not a problem because the kernels learn to adjust for this effect.

2.4 Regularization

Regularization are strategies used in machine learning designed to reduce test error, possibly at the expense of increased training error.

2.4.1 Early stopping

One technique of regularization that can help a model generalize better is the method of **early stopping**. The idea behind early stopping is to terminate training before the model overfits to the training data. The technique requires a separate set of **validation samples** like the test set. These samples are not used for training, but to identify when the model's performance drops due to overfitting, and thus when to stop training. Figure 2.10 illustrates that training should be stopped when the validation error increases. This regularization technique is very effective in tasks like classification when it is easy to measure the model's performance. As will be discussed later this is not necessarily the case for the tasks of generative models.

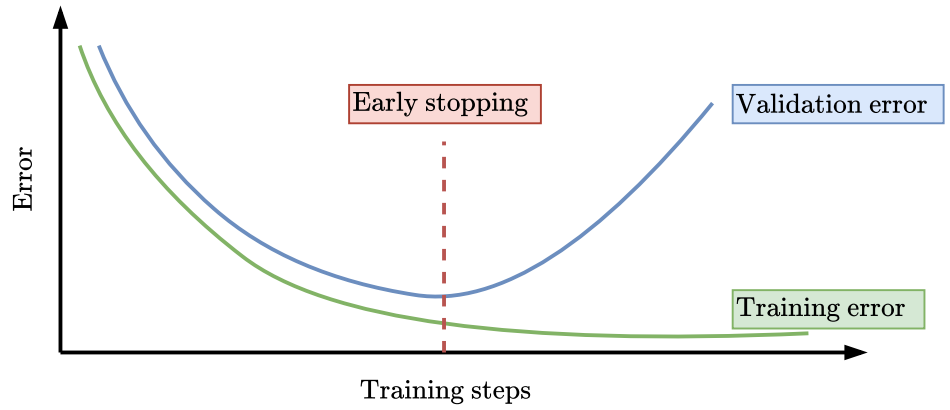


Figure 2.10: An illustration of when to terminate training. Training should be terminated when the model's performance measured on the validation set increases. After this point the model will overfit to the training set.

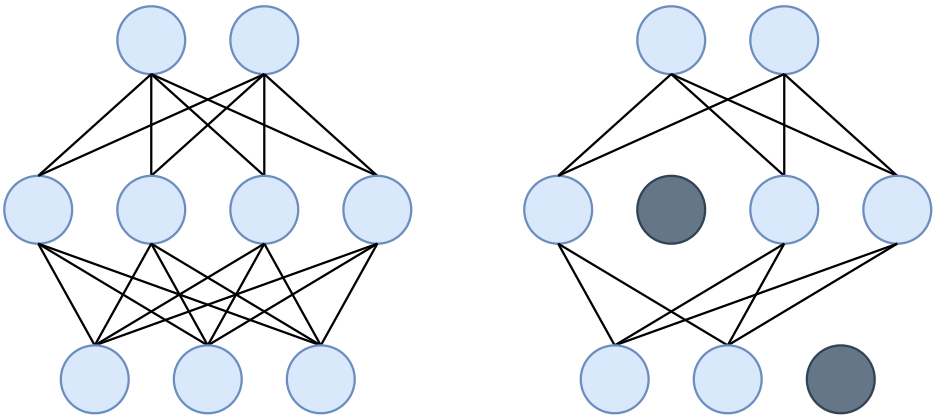
2.4.2 Dropout

Another technique to achieve regularization is to prevent a network to become too reliant on specific units in the layers or features of the input. The strategy of **dropout** (Srivastava *et al.*, 2014) is to remove units and its connections from the network during training with a probability e.g. $p = 0.2$ (figure 2.11) and then do the forward pass, backpropagation and optimization as usual. This strategy will intuitively force the network to learn a form of redundancy so it still performs well without a selection of the units (Goodfellow *et al.*, 2016). When the network is not training all units should be present.

2.4.3 Batch normalization

In section 2.2.2 it was illustrated that deep neural networks can be represented by composite functions (equation 2.17), and in section 2.2.3 that during training the gradient tells how each parameter should be updated. This update procedure is complicated in practice because every layer must constantly adapt to the changes in all the other layers of the network. Ioffe and Szegedy (2015) argues that these changes are covariate shifts in the layers' learned distributions, and that it makes it hard to train models efficiently. They show that a normalization operation to fix the means and variances of the layer inputs help remedy this covariate shift. The normalization is done over each minibatch during training and is referred to as **batch normalization** or **batchnorm**.

The batch normalization operation is done over the minibatch $\mathbb{B} = \{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_b\}$ of vectors $\boldsymbol{x}_i = [x_1, \ldots, x_m]$. The normalized minibatch is given by the normalized



(a) Without dropout (b) With dropout

Figure 2.11: An illustration of a neural network with and without dropout regularization applied. In (b) some inputs and hidden units are dropped during training, so the network must learn to manage without them.

vectors $\boldsymbol{x}'_i$ with elements

$$x'_j = \frac{x_j - \mathbb{E}[x_j]}{\sqrt{\text{Var}[x_j]}} \tag{2.23}$$

where the expectation $\mathbb{E}[\cdot]$ and variance $\text{Var}[\cdot]$ is computed over the minibatch $\mathbb{B}$.

It should be noted that normalizing the activations from a layer can constrain what the layer can represent. To ensure that a layer has the expressive power that is needed each normalized activation x'_j is scaled and shifted with a pair of parameters $\{\gamma_j, \beta_j\}$ that is learned through backpropagation.

$$y_j = \gamma_j x'_j + \beta_j \tag{2.24}$$

This additional transformation allows the new variable to have any mean and variance. This could in principle undo the normalization in equation 2.23 if it was the optimal thing to do. Still, Ioffe and Szegedy (2015) demonstrates that the reparameterization done by equation 2.23 and 2.24 allows a state-of-the-art classification network to learn with 14 times fewer training steps achieving better results.

2.5 Classification of foraminifera using a CNN

The final objective of this thesis is to improve the classification model of Johansen and Sørensen (2020). Their classification model is a large convolutional neural

Table 2.1: A high-level summary of the foraminifera classifier of (Johansen and Sørensen, 2020). This configuration was found through an extensive hyperparameter search that tested 72 permutations of units per layer.

Layer type	Input dim.	Output dim.
VGG16	$224 \times 224 \times 3$	$7 \times 7 \times 3$
Dense (ReLU)	25088	512
Dense (ReLU)	512	64
Dense (softmax)	64	4

network that is trained to classify images of foraminifera. The model utilizes **transfer learning**, a technique that allows a model to reuse parts of e.g. a pretrained convolutional neural network to boost its performance. The parts that can be reused are the learned convolutional filters that can detect abstract features. It has been shown that the utilization of abstract feature-detecting filters from one domain can be useful in a different one (Yosinski *et al.*, 2014).

To build the foraminifera classifier Johansen and Sørensen (2020) used the pretrained weights (filters) of the convolutional blocks of the VGG16 model (Simonyan and Zisserman, 2014) that was trained on the ImageNet (Deng *et al.*, 2009) dataset[4]. The convolutional blocks of VGG16 was used as a feature extractor to obtain the visually relevant features of the foraminifera images. These features were used as inputs to a fully connected neural network with three layers. A simplified model of the foraminifera classifier is displayed in figure 2.12. All layers employed the rectified linear unit (ReLU) activation function (equation 2.25), except the final layer that used the softmax function (equation 2.26) to map the outputs to probabilities associated with a generalized Bernoulli distribution. A high-level summary of the foraminifera classifier is given in table 2.1.

$$\mathrm{ReLU}(x) = \max(0, x) \tag{2.25}$$

$$\mathrm{softmax}(\boldsymbol{x})_i = \frac{\exp(x_i)}{\sum_{j=1}^{n} \exp(x_j)} \tag{2.26}$$

To further improve classification of foraminifera the last two layers of the VGG16 network was retrained to fine-tune the feature extractor to the dataset of foraminifera. Early stopping (section 2.4.1) and dropout (section 2.4.2) was implemented to improve generalization, and classical image augmentation was used to synthetically increase the number of images in the dataset. The augmentation was performed

[4]ImageNet is a large dataset consisting of 14 million real-world images.

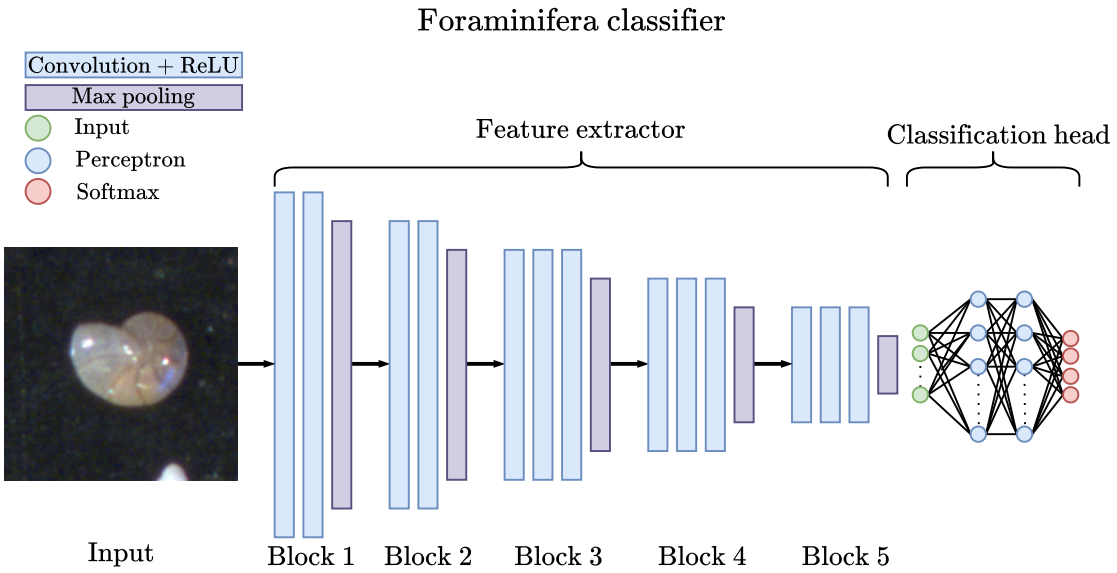


Figure 2.12: A simplified model of the foraminifera classifier. The first part of the classifier uses the pretrained VGG16 (Simonyan and Zisserman, 2014) model to extract relevant features from the foraminifera images. The classification head is a fully connected network that performs classification to four classes.

by horizontal flipping, 90-degree rotations, and randomly adjusting brightness, contrast and saturation with ±10% and hue with ±5%.

Chapter 3

Generative adversarial networks

This chapter presents an in-depth study of the deep learning models referred to as generative adversarial networks (GANs). The study addresses the first objective of this thesis using mainly a theoretical approach. In this chapter the framework, key aspects, as well as challenges and proposed solutions of GANs are presented and explored. The theory and methodology that is comprised in this chapter makes up the theoretical framework upon which the experiments of chapter 4 are based.

The chapter begins by addressing some key challenges of generative models before the GAN framework is presented along with some common challenges of GANs. From there the evolution of GANs is followed until some very recent discoveries in the field of research. Lastly, the chapter is concluded with a general overview, some final notes on GANs and how GAN images can be evaluated.

3.1 Challenges of generative models

Generative models take training samples from a distribution p_{data} and learn to represent an estimate of the distribution. Generative models can represent a probability distribution over one, two or multiple variables. Some models allow evaluation of the probability distribution explicitly and other allow only implicit interaction with the learned distribution, such as sampling data.

3.1.1 The curse of dimensionality

Simple distributions can be estimated using e.g. Parzen window estimation (Rosenblatt, 1956; Parzen, 1962), where superpositions of kernel functions[1] centered at

[1]Not convolutional kernels

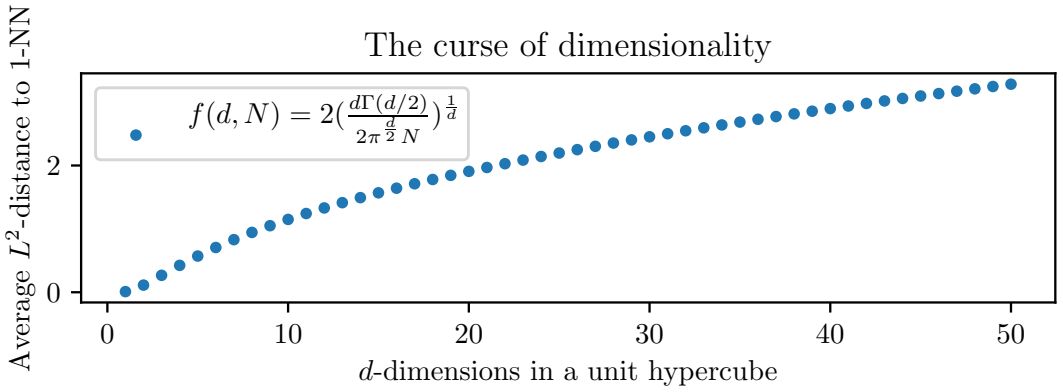


Figure 3.1: The average Euclidean distance to the nearest neighbor for $N = 100$ points drawn from a uniform distribution in a d-dimensional unit hypercube. The plot illustrates the curse of dimensionality because for a fixed number of points the average distance the closest point increases as d increases.

the training data points are used to estimate $\hat{p}_{\text{data}}$. The idea is simple, but falls short when data becomes high-dimensional with many variables.

This problem is often referred to as the **curse of dimensionality** (Bellman, 1961). To illustrate the problem consider a one-dimensional distribution on an interval that is considered densely populated with N equidistant points. For a corresponding two-dimensional distribution on a square, N^2 equidistant points are needed to achieve the same population density. For a 3-D cube N^3 points are needed and so on. Friedman (1989) showed that for N points of a uniform distribution in a d-dimensional unit hypercube the average Euclidean distance to the nearest neighbor is given by

$$f(d, N) = 2\left(\frac{d\Gamma(d/2)}{2\pi^{\frac{d}{2}}N}\right)^{\frac{1}{d}} \tag{3.1}$$

as the dimensionality increases. How the average distance increases is illustrated in figure 3.1. As the dimensionality of a distribution increases, more samples are required to describe the distribution. This constitutes a major problem of learning or estimating high-dimensional distributions, because real data samples or observations are often limited.

As discussed in section 2.1.5 images can be considered as high-dimensional data, because every image is $(h \times w \times c)$-dimensional. To have a model learn this high-dimensional distribution of pixel values would be near impossible if the pixel values were not highly correlated. This means that the dimensionality of real-world image datasets only appear to be artificially high. Once the theme of a real-world image

dataset is fixed the images must follow real life restrictions, e.g. a human face has eyes, nose and a mouth, and cannot have free form. Gong *et al.* (2019) suggests that the true intrinsic dimensionality of high-resolution real-world images are in the range of $[10, 20]$.

Learning this lower-dimensional representation and a mapping to the full-resolution image is still not an easy task. Generative models based on approaches of deep learning and convolutional models have shown an increasing success on the problem of generating images over the recent years. Models based on convolutional Boltzmann machines (Desjardins and Bengio, 2008), variational autoencoders (Kingma and Welling, 2013) and generative adversarial networks (GANs) (Goodfellow *et al.*, 2014) are some of the more successful. Especially models based on GANs have become very popular, achieving to generate high-resolution credible images (Karras *et al.*, 2019a; Brock *et al.*, 2018).

3.1.2 Creating multi-modal outputs

Real-world data distributions are complex and multi-modal with multiple peaks of likely outcomes. A well performing generative model whose goal is to learn the real-world distribution and sample from it, must produce samples that represent modes of the data generating distribution. Depending on the model, one input may correspond to many acceptable solutions. Many traditional models, for instance some that reduce the mean squared error, tend to average over multiple acceptable solutions (Goodfellow, 2016b). In the case of generating images this results in blurry images (Lotter *et al.*, 2015) that are not suitable for many applications of generative models.

Moving forward in this thesis the focus will be to investigate the generative adversarial networks that has shown in particular to be able to produce sharp images from high-dimensional multi-modal distributions such as real-world image datasets.

3.2 Generative adversarial networks

Generative adversarial networks (GANs) are a type of generative models that learn through **adversarial training**. GANs are neural networks that are constructed in clever ways to yield models that overcome much of the challenges of generative models.

3.2.1 The GAN framework

A generative adversarial network is set up as a game between two players: a **generator** and a **discriminator**. The generator tries to create samples that are similar i.e. from the same distribution as the training data. The discriminator tries to predict which samples are fake (from the generator), and which are real (from the training samples). As the game proceeds the discriminator must learn how to distinguish between real and fake samples. The generator must then learn to improve the quality of the fake samples.

The classic analogy of the GAN setup is the police and the forger (Goodfellow, 2016b). The generator is the forger that tries to make fake money, and the discriminator is the police trying to allow legitimate money and stop forged money. In this game the forger wants to make increasingly better forges, and the police must learn to distinguish between real and fake. To make the analogy more correct it should be specified that the forger never actually sees any money it can replicate. It must learn the distribution of real samples without ever examining a real one.

To be more precise the discriminator is a function D that takes a sample (real or fake) $\boldsymbol{x}$ as input, and computes its output using its parameters $\boldsymbol{\theta}^{(D)}$. The generator is a function G that takes a latent variable $\mathbf{z}$ as input. This input can be considered a seed for the generated sample $G(\mathbf{z}) = \hat{\boldsymbol{x}}$. The goal is that the generator function learns the mapping from the latent variable $\mathbf{z} \sim Z$ to the distribution of the data, so $G : \mathbf{z} \rightarrow p_{\text{data}}$. Both functions D and G are usually represented by deep neural networks and are differentiable with respect to their parameters $\boldsymbol{\theta}^{(D)}$ and $\boldsymbol{\theta}^{(G)}$, so the backpropagation algorithm can be applied. The model setup is illustrated in figure 3.2

3.2.2 Learning in the GAN framework

For the functions to learn useful parameters during the game the players need an objective to optimize.

3.2.2.1 Mini-max game

The simplest version is a zero-sum game with a value function $V(\boldsymbol{\theta}^{(G)}, \boldsymbol{\theta}^{(D)})$ that determines the discriminators reward. In the zero-sum game the generator receives the opposite reward $-V(\boldsymbol{\theta}^{(G)}, \boldsymbol{\theta}^{(D)})$. When the game proceeds both players want to learn their parameters $\boldsymbol{\theta}^{(G)}$ and $\boldsymbol{\theta}^{(D)}$ to maximize their reward.

Provided enough capacity the game will *in theory* end when the generator converges to the optimal generator G^* (Goodfellow *et al.*, 2014, 2016) and the solution of the

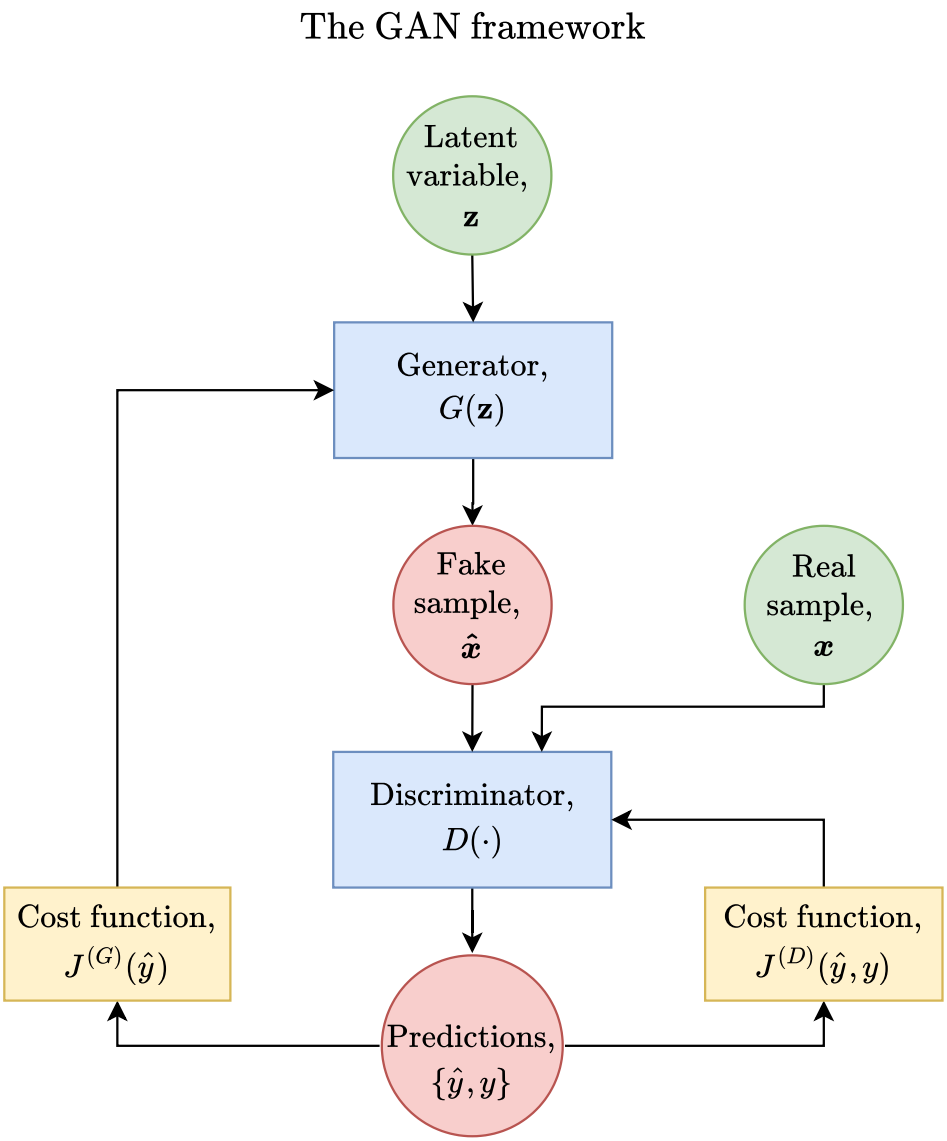


Figure 3.2: The overall structure of a simple generative adversarial network. A latent variable $\boldsymbol{z}$ is the seed of the fake sample produced by the generator G. The fake sample and a real sample are fed to the discriminator D which tries to predict whether the samples are real or fake. The discriminator's output is in the range of $(0, 1)$. The prediction is close to 0 if it predicts a fake sample, and close to 1 for a real sample. The evaluation of the fake sample is the argument of the generator's cost function $J^{(G)}$ and both outputs are arguments of the discriminator's cost function $J^{(D)}$. The cost of the discriminator and generator are used in backpropagation so they can improve.

mini-max game becomes

$$G^* = \arg\min_G \max_D V(G, D) \tag{3.2}$$

When the discriminator wants to maximize the probability of classifying the training data (labels 1) and the fake samples (labels 0) correctly the discriminator's reward, and the value function of the game, is the negative standard cross-entropy cost (equation 2.18) for the discriminator $V(G, D) = -J^{(D)}$ (Goodfellow, 2016b). The discriminator is a binary classifier with a sigmoid output of range $(0, 1)$, so the cost becomes

$$J^{(D)}(\boldsymbol{\theta}^{(D)}, \boldsymbol{\theta}^{(G)}) = -\frac{1}{2}\mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}}[\log D(\boldsymbol{x})] - \frac{1}{2}\mathbb{E}_{\mathbf{z}}[\log(1 - D(G(\mathbf{z})))] \tag{3.3}$$

Using $-J^{(D)}$ as value function in the game, the discriminator learns to distinguish between fake and real samples. Simultaneously the generator learns to produce samples that ultimately become indistinguishable from the real data. At this point the generator has converged to G^* and discriminator outputs $D(\boldsymbol{x}) = \frac{1}{2}$ for all samples (Goodfellow *et al.*, 2016). The game has reached its Nash equilibrium (Nash *et al.*, 1950), where none of the players can improve their strategy to win. Now the discriminator can be discarded and the generator is a generative model that can sample from $p_{\text{model}} = p_{\text{data}}$ (Goodfellow *et al.*, 2014).

It should be noted that these results only apply in theory. In practice the learning of parameters, and convergence of the models, may be very challenging. More on this later.

3.2.2.2 Vanishing gradients in the mini-max game

In the mini-max game the discriminator minimizes, and the generator maximizes, the same cross-entropy cost (equation 3.3). This causes a problem when the generator is learning its parameters. When the discriminator successfully classifies the fake samples as fake the generator's gradients vanish (figure 3.3) because the cost function saturates.

A simple, and somewhat heuristically motivated, solution to this problem would be to flip the targets in the generator's cross-entropy cost function (equation 3.3), so the cost is greater when the discriminator is performing well. The cost of the generator then becomes

$$J^{(G)}(\boldsymbol{\theta}^{(D)}, \boldsymbol{\theta}^{(G)}) = -\frac{1}{2}\mathbb{E}_{\mathbf{z}}\log[D(G(\mathbf{z}))] \tag{3.4}$$

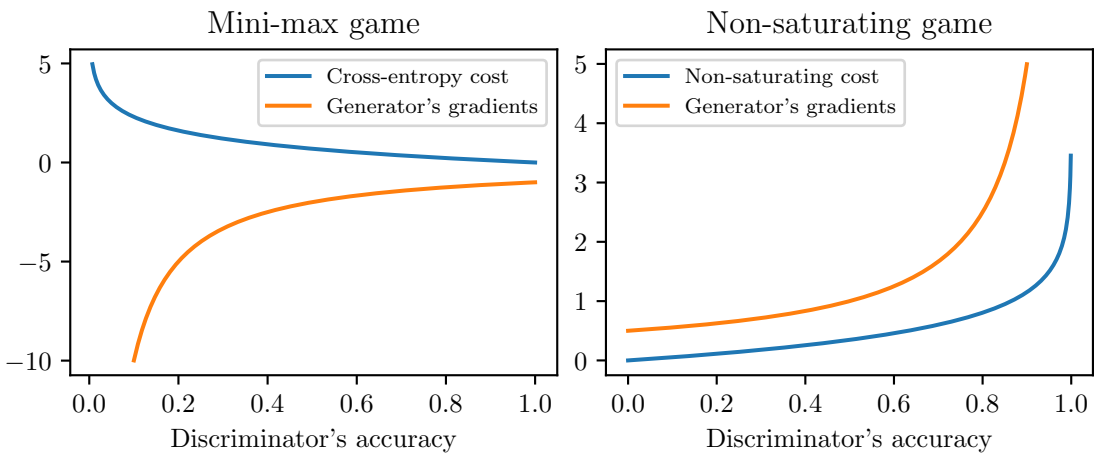


Figure 3.3: An illustration of the vanishing gradient problem in the mini-max game (left). When the generator minimizes the cross-entropy cost (equation 3.3), but the discriminator rejects the fake samples with confidence (high accuracy), the generator's gradients vanish. A simple solution is proposed in the non-saturating game (right), where the target in the generator's cost function is flipped (equation 3.4). In this game the generator's gradients become stronger when the discriminator is performing well (high accuracy). In this game the generator's learning does not halt even if it is performing poorly.

This is known as the **non-saturating cost** for the generator. This cost function gives the generator stronger gradients (figure 3.3), and allows it to learn its parameters faster, even though the discriminator is performing well. This modification ensures that the generator tries to generate samples that have a high probability of being real, rather than a low probability of being fake as in the original mini-max game (equation 3.3) (Fedus *et al.*, 2017).

The GAN framework is open for many different cost functions. The cost function for the discriminator (equation 3.3) and the generator (equation 3.4) were the first to become popular and provide good results (Goodfellow *et al.*, 2014; Radford *et al.*, 2015; Denton *et al.*, 2015).

3.2.3 Learning the distribution of a circle

The generator of a GAN can learn a distribution without ever observing the training data directly. Everything it learns is through the judgements of the discriminator. To demonstrate how this happens, and to concretize how a GAN learns, this section illustrates through an example how the generator can learn the distribution of the unit circle.

The GAN in this example consists of a generator and a discriminator represented

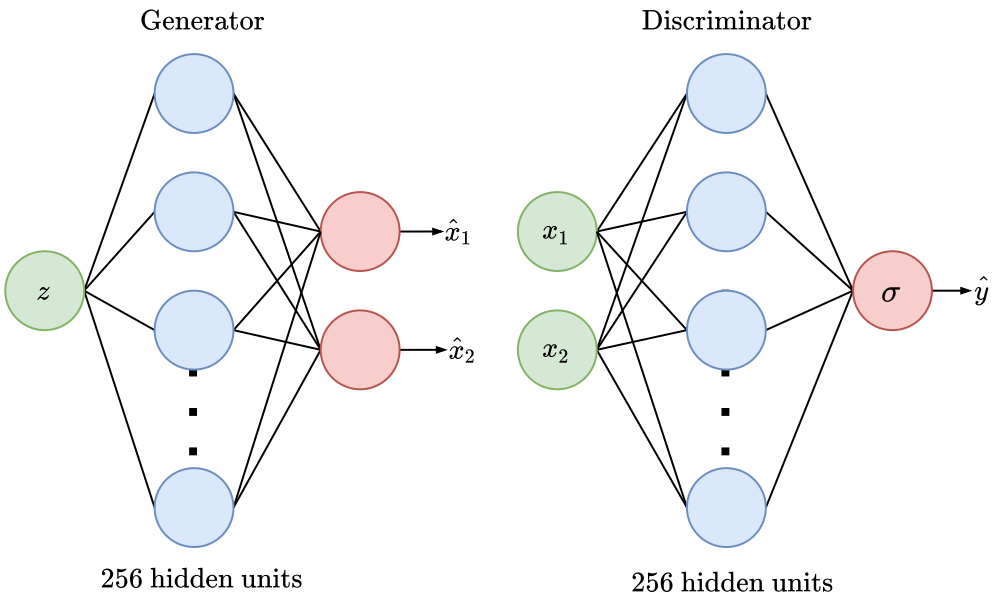


Figure 3.4: The architecture of the generator and discriminator that was used in the generative adversarial network that learned the distribution of the unit circle in section 3.2.3.

by two multi-layer perceptrons with one hidden layer each consisting of 256 hidden units. Both models use the ReLU activation function (equation 2.25) in all hidden units. The generator and discriminator are presented in figure figure 3.4

The input of the generator is a scalar z $\in Z \sim$ Uniform$(-1, 1)$ and the output layer uses an identity (linear) activation function, so the layer only performs matrix multiplication and adds the bias without sending it through a non-linearity. The discriminator takes as input a vector $\boldsymbol{x} = [x_1, x_2]$ and the output layer uses a sigmoid activation function $\sigma(\boldsymbol{x}) = y$ to produce an output of range $(0, 1)$. The cost is computed using the binary cross-entropy cost function (equation 3.3) for the discriminator and the non-saturating cost for the generator (equation 3.4).

As stated the goal of this GAN is to learn the distribution of the unit circle. To do this 300 points are sampled randomly from the true distribution, and used as training data. The GAN is trained on minibatches of 32 samples each training step according to the following training scheme:

1. The generator samples 32 numbers from Z and generates fake samples by performing the forward pass.
2. The discriminator is trained on 32 real training samples and the 32 fake samples. The real samples get labeled as 1s and the fake as 0s.

3. The generator is trained through the discriminator evaluating the fake samples from $G(\mathrm{z})$ and then computing the cost using 1s as labels (flipped targets equation 3.4).

Additional details on the training scheme and the code to implement a basic GAN using the high-level API of `Keras` is presented in appendix B.

In figure 3.5 and figure 3.6 it is shown how the GAN training evolves during the 720 epochs of training. As the training proceeds the GAN learns the mapping from z to the circumference of the unit circle. In figure 3.6 the discriminator's decision boundary is displayed as a contour map. Red areas indicate that the discriminator classifies everything within as fake, and in the blue, everything is classified as real. As both models learn the mapping simultaneously the results are poor in the beginning, but after around epoch 550 the results improve. At this point the generator has learned to produce samples that are almost indistinguishable from the real samples, and the discriminator outputs $D(\boldsymbol{x}) \approx 0.5$ for all samples (figure 3.5). Although the discriminator outputs 0.5 it is not guaranteed that the GAN has converged. The generator might continue to learn a better representation of the distribution, and the discriminator may learn a sharper decision boundary.

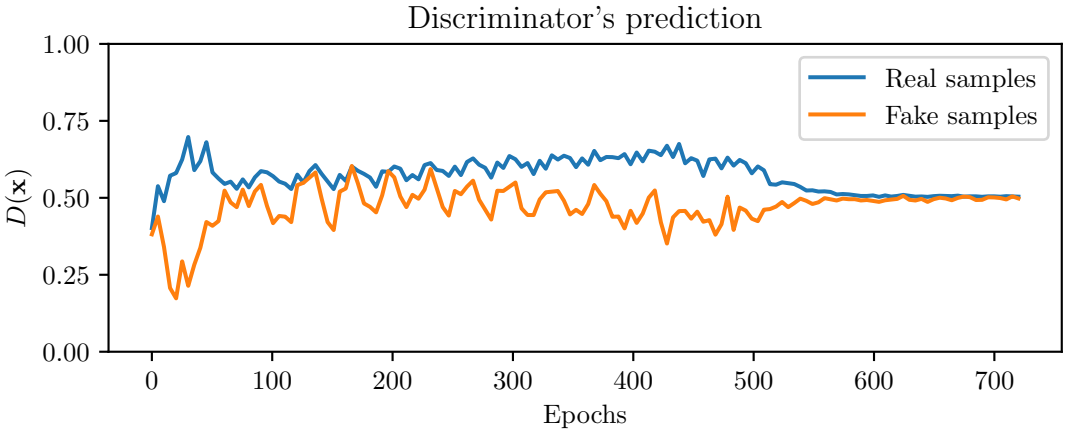


Figure 3.5: Shows how the discriminator's average predictions of real and fake samples evolves during training. In the beginning the predictions fluctuates, but after epoch 550 the discriminator is unable to distinguish between the real and fake samples and outputs ≈ 0.5 for all samples.

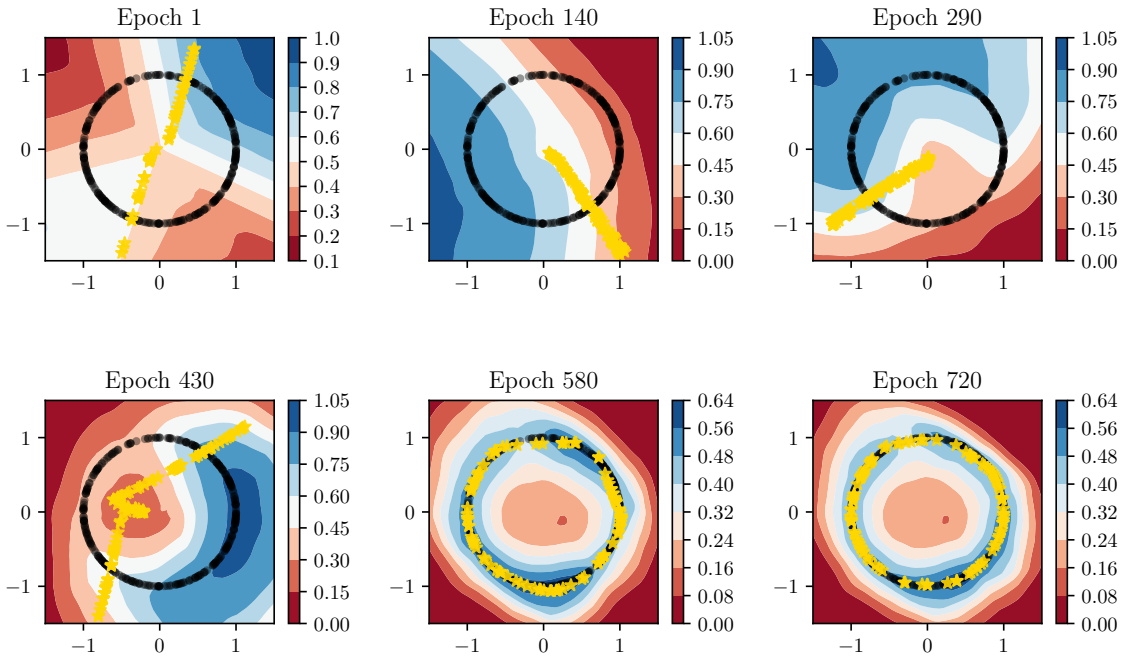


Figure 3.6: A visualization of what the generator and discriminator learns during training. The contour map shows the learned decision boundary of the discriminator, and the yellow stars show 100 mappings of the generator. The 300 black dots in a circle formation are the training points that the discriminator use for training.

3.2.4 Interpolation in latent space

To get a better understanding of what the generator really has learned one can try to illustrate the mapping G performs from the latent z-space to the generated samples $\hat{\boldsymbol{x}}$. This can be performed by feeding in evenly distributed values of the latent space, and visualize where they are mapped to. Figure 3.7 shows an interpolation in the latent z-space with values from -1 to 1 plotted at $y = 0$. These latent points are used as seeds to the generator that has learned to map them to its distribution p_{model}. One interesting observation is that the points that are close in the latent space are also close in the learned output space. This property indicates that the generator has learned a continuous mapping from $\text{z} \rightarrow G(\text{z})$. Goodfellow *et al.* (2014) and Radford *et al.* (2015) demonstrated that this also applies for higher dimensional data such as images. This property allows well performing GANs to sample smooth transitions between images by interpolating in the latent $\boldsymbol{z}$-space of the generator. This will be demonstrated in the experiments of chapter 4.

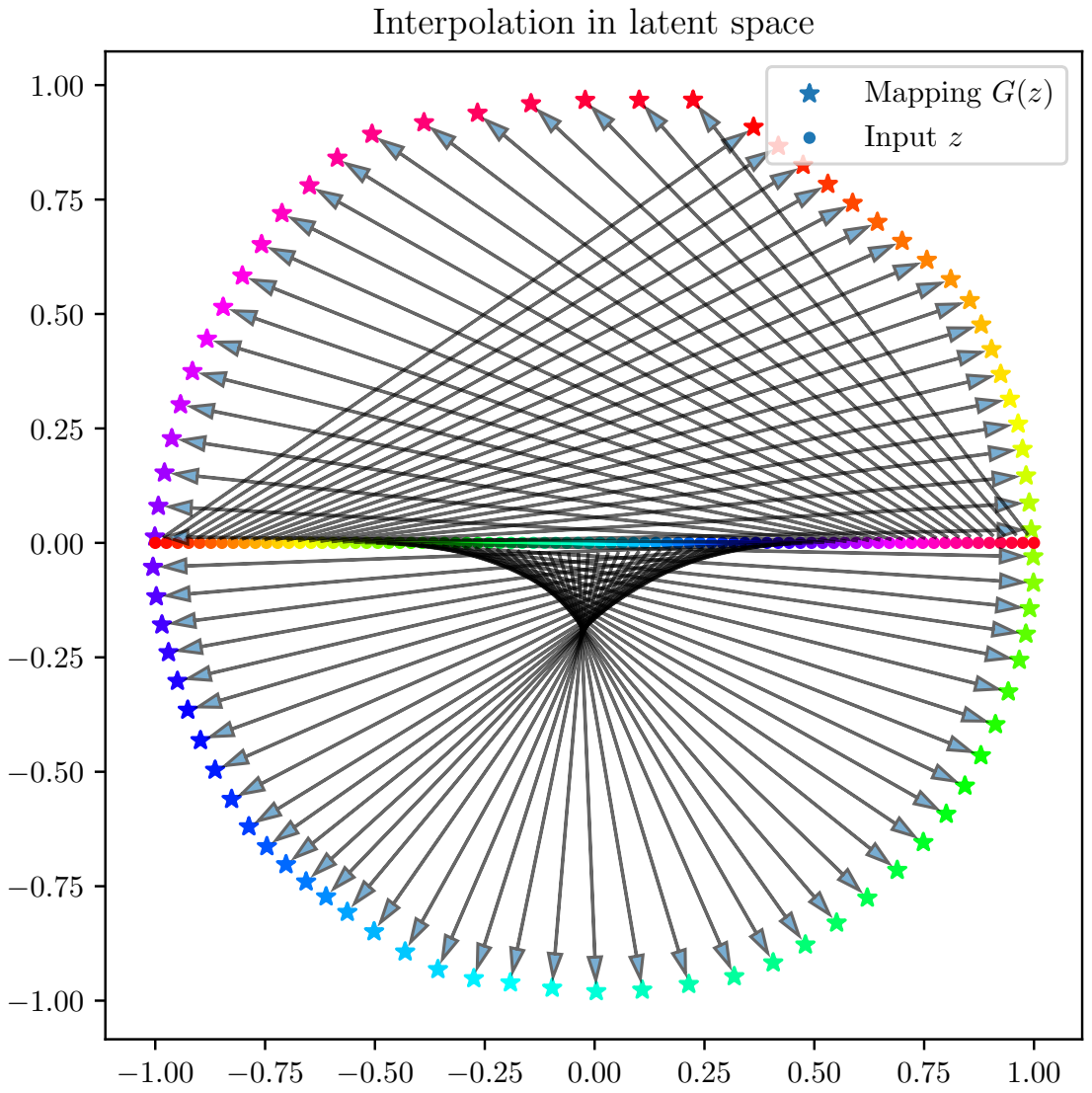


Figure 3.7: The mapping learned by the generator. 80 input points z are linearly spaced in the range $[-1, 1]$ and are passed into the generator $G(\mathrm{z})$. The arrows and colors show where each input end up after the mapping. The pattern formed by the arrows is caused by the non-linear stretching and bending of the input space that the generator has learned when mapping inputs to the circumference of the circle.

3.3 Challenges of generative adversarial networks

The GAN framework has several computational advantages over other generative models and have provided good results on many different tasks (Goodfellow, 2016b). Despite the progress in GANs the recent years there are still major challenges related to convergence and training. In this section some of the major challenges one are faced with when working with GANs will be presented.

3.3.1 Training instability

Figure 3.5 and 3.6 shows how the discriminator's predictions oscillates somewhat during training in the example of the unit circle GAN in section 3.2.3. It was observed in the aforementioned example that the training stabilized after a while, but it was not guaranteed that the GAN had converged. For GANs it is common that the performance oscillates a lot during training, and especially for more complex datasets.

This, along with the challenge of determining convergence, are training related issues that are common to GANs and are considered open fields of research. Later in this thesis some possible solutions to these problems will be addressed, but first the culprit needs to investigated.

The mini-max-game example in section 3.2.2.1 converges in theory using simultaneous gradient descent, but this result is not guaranteed in practice. In practice the updates during training are made in parameter space, so the value surface is not necessarily convex (Goodfellow *et al.*, 2014). This implies that when the two models are trained simultaneously with conflicting interests the improvement of one comes at the expense of the other.

Most deep learning models are optimized by finding a minimum of a cost function. This is not always an easy task, but an optimization algorithm usually make steady progress downhill towards a local minimum (Goodfellow, 2016b). In GANs on the other hand the optimization algorithm is required to find the equilibrium in a game between two players with opposing goals. Finding the equilibrium is hard and the adversarial players often undo each other's progress without making any overall progress.

To illustrate the problem consider the mini-max game of two players controlling a single scalar value each, x and y. The value function of the game is

$$V(x, y) = xy \tag{3.5}$$

where player 1 wants to maximize V by controlling x, and player 2 wants to minimize V by controlling y. Visualizing the surface of the value function in

three dimensions (figure 3.8) gives an idea of why finding the equilibrium can be a difficult task in optimization. The equilibrium is a saddle point at $x = y = 0$ with 0 gradient. If the optimization algorithm were to follow the gradients on the value surface the model would never converge because simultaneous gradient descent forms a circular trajectory (Goodfellow, 2016b).

This example illustrates that games do not always converge for simultaneous gradient descent. GANs sometimes converges, but *"[...] there is no theoretical prediction as of whether simultaneous gradient descent [for GANs] should converge or not."* (Goodfellow, 2016b, p. 49).

In practice GAN training do not always follow a circular trajectory like in figure 3.8. Another scenario is that the gradients cause the generator to spiral outwards (Bailey and Piliouras, 2018), or that the stochasticity of training orients the gradients in the correct direction, but the learning rate is too large, so the optimization algorithm overshoots the equilibrium (Mescheder *et al.*, 2018). All these scenarios make it hard for GANs to converge and are some of the reasons that convergence for GANs are often considered a more fleeting than stable state.

This may give an intuition of why it can be helpful to average out the oscillations in training. Yazıcı *et al.* (2018) show theoretically and experimentally that keeping an **exponential moving average** (EMA) of the GAN's weights outside the training loop will help dampen the amplitude of oscillations, and thus contribute to stabilize GAN training. An exponential moving average of the weights $\boldsymbol{\theta}$ at iteration t are defined as

$$\boldsymbol{\theta}_{\text{EMA}}^{(t)} = \alpha \boldsymbol{\theta}_{\text{EMA}}^{(t-1)} + (1 - \alpha)\boldsymbol{\theta}^{(t)} \tag{3.6}$$

where $\alpha \in [0, 1)$ is the rate of decay in the EMA. Common values for α are usually close to 1 like 0.999 and 0.9999. The effect of EMA for different decay values on simple oscillations are illustrated in figure 3.9.

3.3.2 Mode collapse

Real-life data distributions have often multiple modes representing ranges of data values that are more likely to occur than other (section 3.1.2). It is desirable that a generative model is able to reproduce the whole range of possible outputs from the distribution. In the case of GANs this means that the model should be able to sample from the whole distribution, not just provide small subset of the distribution.

A common problem in GANs is when the generator is not able to reproduce the full distribution, only a small subset. This is called the **Helvetica scenario** and is caused by **mode collapse**, in the generator. In mode collapse the generator maps

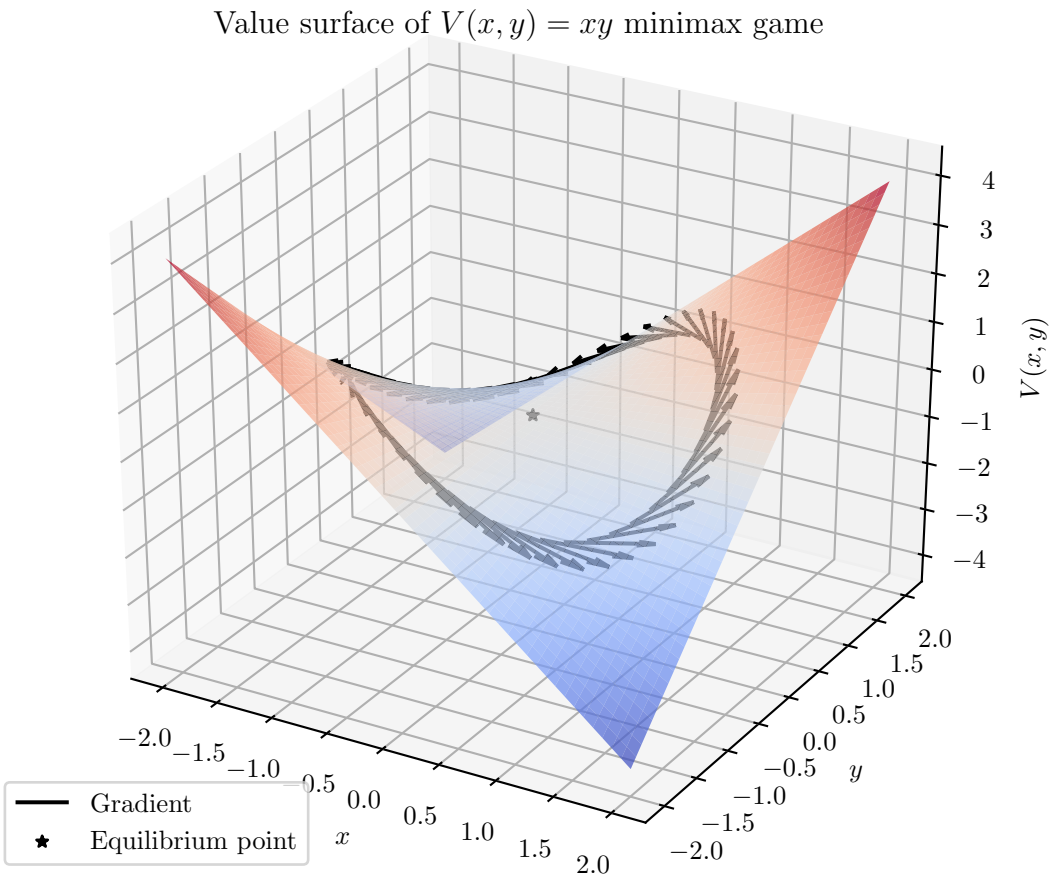


Figure 3.8: The trajectory and the gradients given by simultaneous gradient descent with infinitesimal small steps on the value surface of the game $\min_x \max_y V(x, y) = xy$. The players will orbit the equilibrium point and the game will never converge if the optimization algorithm were to follow the gradients of the value surface.

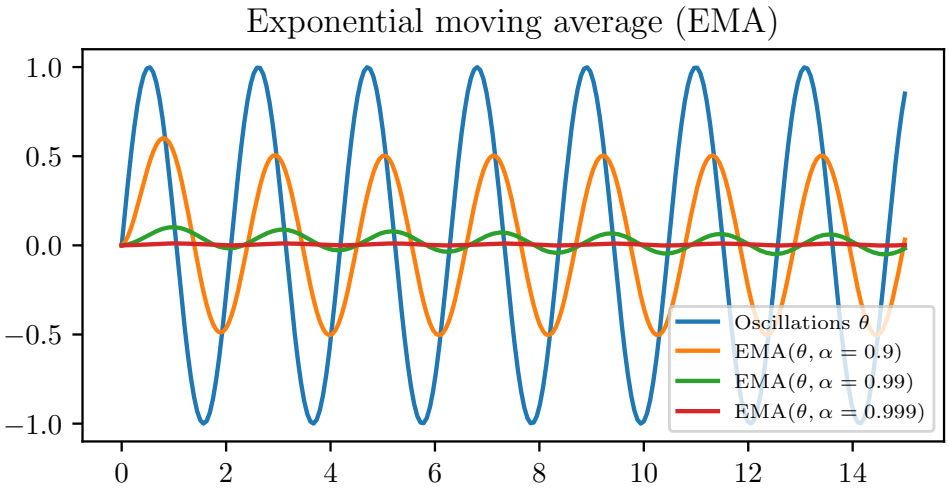


Figure 3.9: The plot illustrates the effect of equation 3.6 on sinusoidal oscillations with different decay rates α. Decay rates close to 1 dampen the amplitude of the oscillations more.

too many values of $\mathbf{z}$ into the same $\hat{\boldsymbol{x}}$ so p_{model} has too little diversity to represent p_{data}. In practice this causes the fake samples to appear too similar with little or no variation.

Mode collapse happens when the generator and discriminator are not synchronized well. This usually becomes a problem when the generator is trained more than the discriminator, and the generator overfits to the low-grade discriminator. To get an intuition of how this happens consider the simplified and extreme case where the generator is updated extensively without any updates to the discriminator. The generator will learn to map all inputs to the one output that fools the discriminator the most. In this case the generator will collapse p_{model} into the same single point $\hat{\boldsymbol{x}} = \boldsymbol{x}$ that the discriminator believes is realistic (Goodfellow, 2016b). When the discriminator then learns that this point is also fake, gradient descent will not be able to produce sensible gradients that recover the models from the collapsed mode (Salimans *et al.*, 2016).

In a less extreme example not all modes are dropped. During training of the generator one can imagine that the fake samples are drawn towards their nearest neighbor as illustrated in figure 3.10. As training proceeds the fake points get closer to their nearest real neighbor. When this happens, every fake sample has a nearby real sample, though it is not guaranteed that every real sample has a nearby fake sample (Li, 2019). This can cause some modes of the data generating distribution to be dropped.

Li and Malik (2018) proposed an intuitive fix to this problem for generative models. Instead of letting the generated samples go towards their nearest neighbor, they let the real samples pull the generated samples towards themselves, to avoid dropping modes. This solves the problem of mode collapse, but the solution uses implicit maximum likelihood estimation (Li and Malik, 2018) and is no longer adversarial training. Furthermore their model does not provide qualitative better results than GANs so this solution will not be considered as a sufficient solution to mode collapse in GANs.

3.3.3 Addressing the challenges of GANs

In the field of generative adversarial networks there have been few theoretical results that solve the challenges one are faced with when training GANs. Training instability, convergence and mode collapse are still missing solid theoretical results needed to solve the problems for good.

Despite this there has been progress in the field of GANs the recent years. The progress have mainly been techniques and practices that are proven useful in some cases. Some of these remedies have some theoretical basis, but other are just considered tricks worth trying out. Before addressing these remedies, the task of synthetic image generation using the deep convolutional GANs is first presented.

3.4 Deep convolutional GANs

Synthetic image generation is a major task of GANs that has shown great improvements over the recent years. The task is also of great relevance to this thesis as one objective is to generate synthetic images of foraminifera.

3.4.1 Early deep convolutional GANs

When the GAN framework was launched in 2014 the idea of combining the generator and discriminator with convolutional layers was proposed. The strength of this methodology is to use the pattern recognition capabilities of the well-developed convolutional neural networks in combination with the GAN framework. Goodfellow *et al.* (2014) showed that using convolutional layers in the discriminator and transposed convolutions in the generator produce qualitatively better fake samples than using a fully connected model (MLP). Still, the generated images were noisy and incomprehensible, with great room for improvement.

Denton *et al.* (2015) improved the convolutional GAN architecture by extending it with a Laplacian pyramid (Burt and Adelson, 1983) approach. This Laplacian

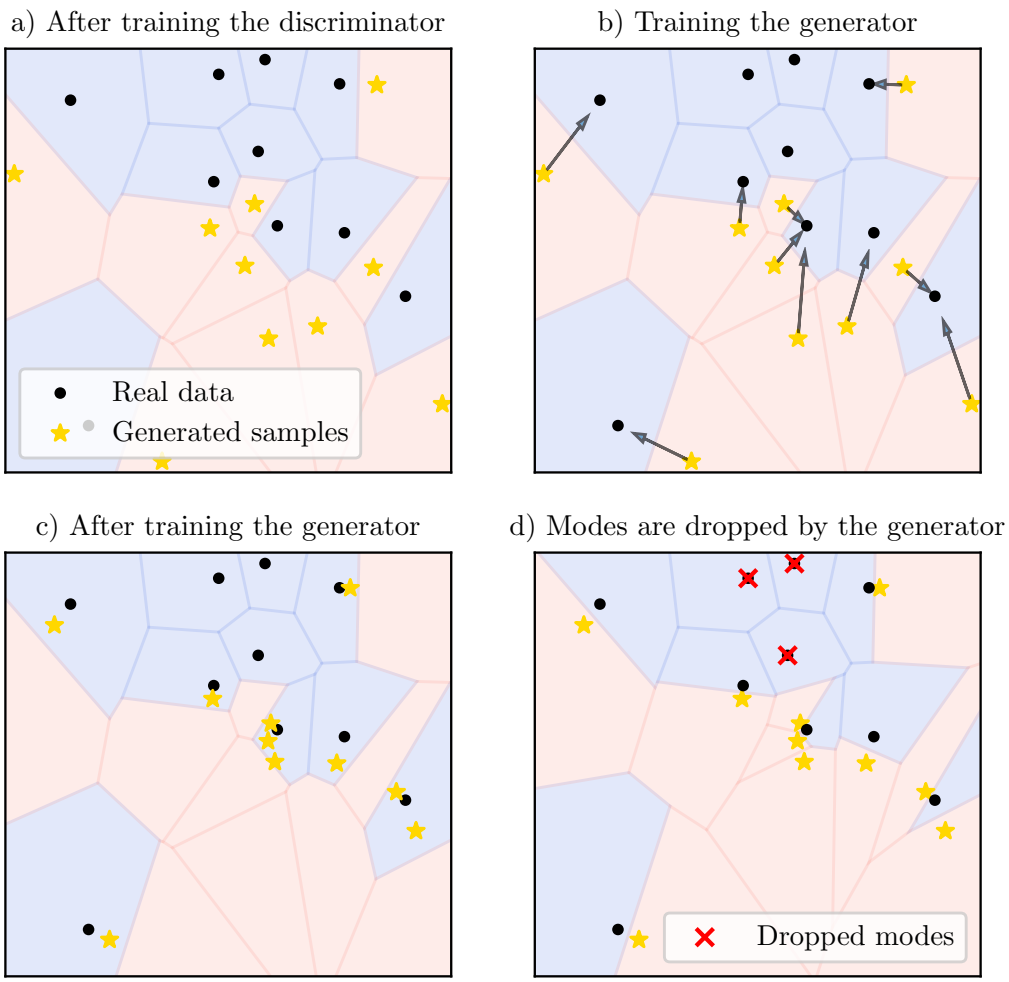


Figure 3.10: An illustration of why modes are dropped during in mode collapse. The discriminator and generator are trained on a toy dataset of points (•). a) After the discriminator is trained all the blue regions are classified as real, and all the red as fake. b) While training the generator the fake samples get closer to the nearby samples of the real distribution resulting in the scenario of c). d) After the discriminator is trained again new discrimination regions are produced. As each generated point gets pushed towards its nearest neighbor it is not guaranteed that all real samples have a nearby fake sample (Li, 2019). Thus, modes of the real distribution may be dropped by the generator.

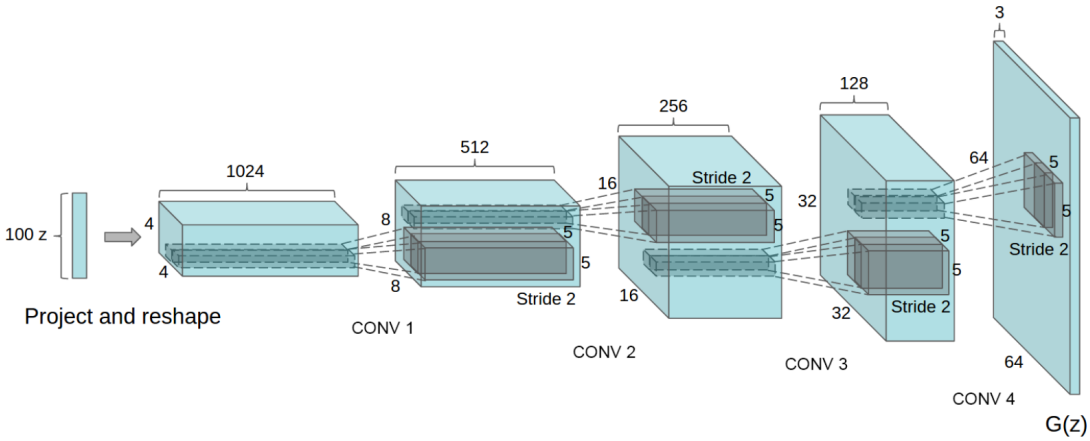


Figure 3.11: An illustration of the generator of a DCGAN. The latent vector $\mathbf{z}$ is reshaped and projected to a small (4×4) image that is scaled up by learned upscaling filters of transposed convolutions of stride > 1. The result is a (64×64) image with 3 channels (RGB). The figure is from (Radford *et al.*, 2015).

pyramid GAN (LAPGAN) generated images using multiple GANs in multiple stages. First a coarse image is generated, then different convolutional GANs apply increasingly more detail to the image. Using this approach LAPGAN were able to generate plausible looking scenes at resolutions of (32×32) and (64×64) pixels.

3.4.2 DCGAN architecture

The first GAN that was able to produce high quality images at resolutions of (64×64) pixels in one shot was the deep convolutional generative adversarial network (DCGAN) of Radford *et al.* (2015).

The DCGAN architecture is built on the approach of LAPGAN, by starting with a low resolution image and gradually scaling it up, but without using Laplacian pyramids or multiple GANs.

The approach of Radford *et al.* (2015) uses transposed convolutions with strides > 1 to learn upscaling filers from one resolution to the next. In the beginning a 100-dimensional vector $\mathbf{z}$ from a uniform distribution is projected and reshaped to a (4×4) image. This low resolution image is upscaled using the learned filters to (8×8), (16×16), (32×32) and finally (64×64) pixels. Figure 3.11 illustrates the architecture of a DCGAN model that was used in (Radford *et al.*, 2015). The discriminator of a DCGAN is conceptually similar to a reversed generator only it uses convolutions with strides > 1 to downsample the image instead of transposed convolutions to upscale it. Details are covered in the following section and in

(Radford *et al.*, 2015).

3.4.3 Architectural guidelines

Radford *et al.* (2015) developed some architectural guidelines to achieve stable training for deep convolutional GANs. These were long considered the standard of GAN image generation and has influenced many architectures since.

Strided convolutions. DCGANs should use the all-convolutional architecture from Springenberg *et al.* (2014). In this architecture all spatial pooling (and unpooling) functions such as maxpooling (section 2.3.5) are replaced with strided convolutions. In the discriminator this allows the network to learn its own downsample filters using strided convolutions. Similarly, the generator uses strided transposed convolutions so the network learns its own upsample filters to produce images.

Batch normalization in the discriminator and generator stabilizes training (Radford *et al.*, 2015). When using batch normalization, the input of each unit is normalized to have mean 0 and variance 1. This takes care of problems caused by poor weight initialization and helps gradients flow through the network in deeper models. Radford *et al.* (2015) found that this was critical to prevent mode collapse in the generator but could also cause oscillation and model instability if batchnorm was applied to all layers. To bypass this problem batchnorm should not be applied to the output of the generator and the input of the discriminator.

The generator used the **hyperbolic tangent** (tanh) function as activation function in the output layer. Radford *et al.* (2015) observed that this allowed the generator to learn the saturation and coverage of the color space more quickly. The tanh function (equation 3.7) is the shifted and scaled sigmoid function and produces outputs in the range of $[-1, 1]$. All training images should consequently be scaled and shifted to this range before training.

$$\tanh(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)} \tag{3.7}$$

In addition the following should be noted on the DCGAN architecture:

- For optimization they used Adam instead of stochastic gradient descent.
- There were no fully connected hidden layers in deep models.
- They used ReLU activation function (equation 2.25) in all layers of the generator, except for the output layer, that used Tanh (equation 3.7).

- LeakyReLU (equation 3.8) with leakiness $a = 0.02$ was used as activation function in all the layers of the discriminator.

$$\text{LeakyReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ ax & \text{otherwise} \end{cases} \tag{3.8}$$

3.4.4 Challenges of the DCGAN architecture

The DCGAN architecture provided guidelines for developing more stable generative adversarial networks. Let it be clear that "more stable" is relative to the early GAN models (Goodfellow *et al.*, 2014; Denton *et al.*, 2015) that were notoriously unstable. The DCGAN architecture addressed the instability problems of the early GANs, but did not perfect them. Radford *et al.* (2015) report also of mode collapse in DCGANs, especially when the models are trained for longer. In addition DCGANs are like most GANs sensitive to the choice of learning rate, optimizer and size of dataset.

3.5 Wasserstein GAN

The cost function that is considered up to this point is the cross-entropy cost (equation 3.3) with the heuristic non-saturating modification to the generator's cost function presented in section 3.2.2.1. This heuristic modification has no theoretical justification, and is only motivated by the desire of a strong gradient of each player when the adversary is "winning" Goodfellow (2016b).

This section comprises the buildup of the **Wasserstein loss** (Arjovsky *et al.*, 2017) as a new cost function for GANs that has theoretical advantages over the cross-entropy cost function.

3.5.1 Wasserstein distance

Wasserstein distance (Vaserstein, 1969) also known as the **Earth-mover distance** (Rubner *et al.*, 2000) is a metric that measures the distance between probability distributions. If one imagines that two distributions $p_r = p_{\text{data}}$ and $p_g = p_{\text{model}}$ are piles of dirt, the EM distance would be the cost of the most effective way to transform one distribution into the shape of the other distribution (Arjovsky *et al.*, 2017). The cost is calculated by the amount of dirt that should be moved multiplied with the Euclidean distance to move the dirt.

Consider the two discrete distributions in figure 3.12. To compute the EM distance one would need to find the optimal way to transform P_g into P_r and examine

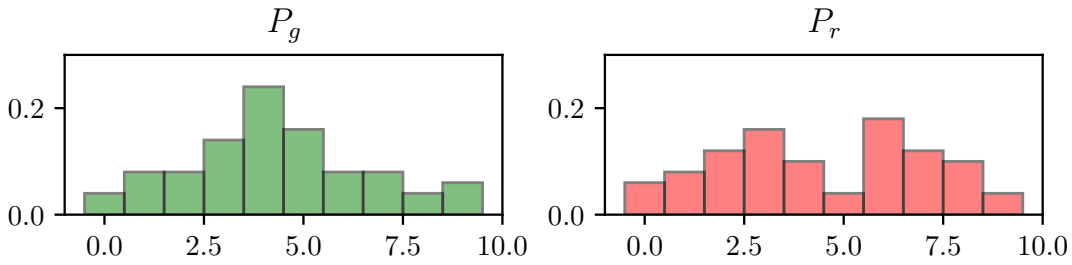


Figure 3.12: Two example distributions to illustrate the starting distribution P_g and the goal distribution P_r in the Earth mover distance example of section 3.5.1.

how much of the distribution, and how far, it should be moved. Assuming that the piles are of equal mass, i.e. for probability distributions that they sum to 1, there are infinitely many ways transform P_g into P_r. Each transport plan $\gamma(x, y)$ states how the proportion of dirt should be redistributed from x to y, so as long as $\sum_x \gamma(x, y) = P_r(y)$ and $\sum_y \gamma(x, y) = P_g(x)$ the transport plan is valid. Equivalently γ is a joint probability distribution when $\gamma \in \mathbb{P}(P_g, P_r)$, where $\mathbb{P}(P_g, P_r)$ is the set of all distributions with marginals P_g and P_r respectively (Herrmann, 2017). To obtain the EM distance every value (x, y) of the optimal transport plan γ^* is multiplied with the Euclidian distance $\|x - y\|$ between the points. More formally the Earth mover distance (Wasserstein-1 distance) is defined by

$$\begin{aligned} W(P_g, P_r) &= \sum_{x,y} \|x - y\| \gamma^*(x, y) \\ &= \inf_{\gamma \in \mathbb{P}(P_g, P_r)} \mathbb{E}_{(x,y)\sim\gamma}[\|x - y\|] \end{aligned} \tag{3.9}$$

where the **infimum** (inf) is the greatest lower bound of the set indicating that the only transport plan of interest is the optimal one, and hence yields the smallest cost.

As there are an infinite number of possible transport plans, finding the optimal is an optimization problem in itself. For the above example with the distributions of figure 3.12 the optimal transport plan can be found using linear programming[2]. Figure 3.13 show the optimal transport plan with all the required transportations $\gamma^*(x, y)$ and figure 3.14 illustrates the result after the optimal transportation is executed.

[2]Details are presented in (Herrmann, 2017)

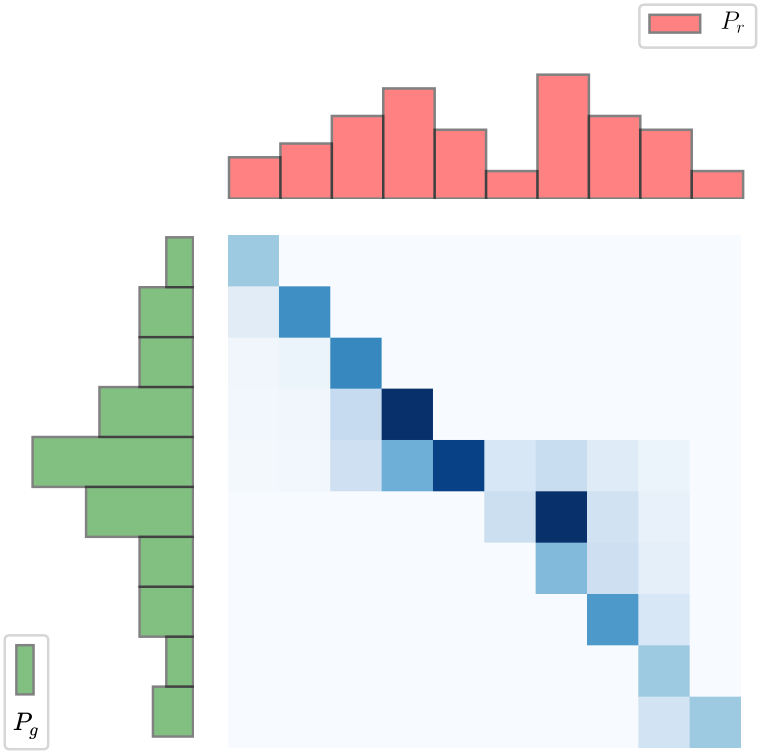


Figure 3.13: The optimal transportation plan $\gamma^*(x, y)$ illustrated as a joint distribution with marginals P_g and P_r. Darker squares indicates that a greater proportion is moved, and the position of the square give where it should be moved from and to.

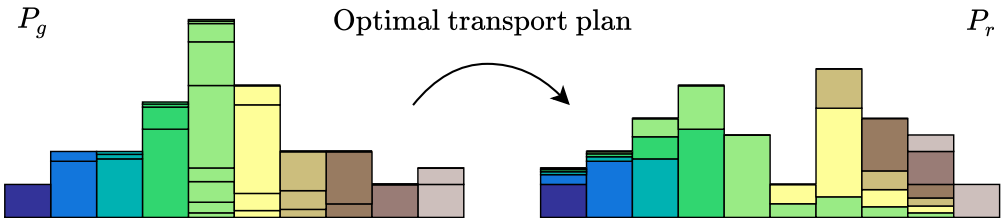


Figure 3.14: The result of the optimal transport plan $\gamma^*(x, y)$. The colored pieces of the distribution indicates where each piece ends up.

3.5.2 Advantages of the Wasserstein distance

As illustrated the Wasserstein distance measures the difference between two distributions. This is useful in GANs because the loss function should essentially compare the learned distribution of the generator with the distribution of the training samples. The discriminator in early GANs used the cross-entropy loss (equation 3.3) to do this comparison.

In section 2.2.3 the loss was computed during training so the optimization algorithm could use the gradients of the loss function to update the learned distribution p_g, and hence make it more similar to the objective distribution p_r. Arjovsky *et al.* (2017) argues that the weaker the distance measure between p_g and p_r, the easier it is to learn the continuous mapping from $p_g \rightarrow p_r$ because it is easier for the distribution sequence to converge. Hence using a weak distance measure as loss function may give stronger gradients when learning a distribution. Arjovsky *et al.* (2017) proves that the Wasserstein distance is a weaker distance measure than the Kullbach-Lieber divergence[3] and Jensen-Shannon divergence[4].

This suggests that the Wasserstein distance is a compelling loss function for generative adversarial networks.

3.5.3 Towards a Wasserstein loss function

Though the Wasserstein distance is compelling to use as loss function it is not straight forward. Equation 3.9 call for the infimum of all possible joint distributions $\inf_{\gamma \sim \mathbb{P}(p_g, p_r)}$. This is intractable to compute, so Arjovsky *et al.* (2017) use the Kantorovich-Rubinstein duality (Kantorovich and Rubinstein, 1958) to express $W(p_g, p_r)$ using the supremum sup, also known as the least upper bound, instead.

$$W(p_g, p_r) = \frac{1}{k} \sup_{\|f\|_L \leq k} \mathbb{E}_{x \sim p_g}[f(x)] - \mathbb{E}_{x \sim p_r}[f(x)] \tag{3.10}$$

This formulation calls for the maximum over all functions $\|f\|_L \leq k$, meaning that f is k-Lipschitz continuous.

[3]The Kullbach-Lieber divergence is minimized when training a GAN using maximum likelihood (Goodfellow, 2016b)

[4]The Jensen-Shannon divergence is minimized in the original GAN mini-max game (Goodfellow *et al.*, 2014).

3.5.3.1 Lipschitz continuity

A real valued function $f : \mathbb{R} \to \mathbb{R}$ is k-**Lipschitz continuous** when there exists a constant $k \geq 0$ such that for all $x_1, x_2 \in \mathbb{R}$

$$|f(x_1) - f(x_2)| \leq k|x_1 - x_2| \tag{3.11}$$

A Lipschitz continuous function is always continuous, but not necessarily differentiable (Sauer, 2012). This means that functions, such as $f(x) = |x|$, that are continuous everywhere but not differentiable at $x = 0$, still can satisfy the Lipschitz continuity condition. Intuitively this is because Lipschitz continuity is related to the how rapid the function value changes over a small interval of the domain.

3.5.4 From discriminator to critic

In equation 3.10 the function f helps measure the Wasserstein distance between the distributions p_g and p_r and is required to be k-Lipschitz continuous.

Arjovsky *et al.* (2017) proposed a modification of the GAN scheme where the discriminator is the function f that helps the generator learn the distribution p_r. In this version f is parametrized by w and is assumably from a family of functions $\{f_w\}_{w \in \mathbb{W}}$ that are k-Lipschitz continuous. Now the Wasserstein distance for a GAN can be achieved by

$$W(p_r, p_g) = \max_{w \in \mathbb{W}} \mathbb{E}_{x \sim p_r}[f_w(x)] - \mathbb{E}_{\mathrm{z} \sim p(\mathrm{z})}[f_w(g_\theta(\mathrm{z}))] \tag{3.12}$$

This equation serves as the loss function in the **Wasserstein GAN** (WGAN). In this version of the GAN game the discriminator no longer classifies real and fake samples but is rather considered a **critic** that helps the generator produce good fake samples. When the critic computes the Wasserstein distance it is possible to back-propagate through equation 3.12 by estimating $\mathbb{E}_{\mathrm{z} \sim p(\mathrm{z})}[\nabla_\theta f_w(g_\theta(\mathrm{z}))]$.

Using more familiar terms with $G(\mathbf{z})$ as the generator and $D(\boldsymbol{x})$ as the critic (formerly the discriminator), the value function of this modified GAN scheme becomes

$$V(D, G) = \min_G \max_{D \in \mathbb{D}} \mathbb{E}_{\boldsymbol{x} \sim p_{\text{data}}}[D(\boldsymbol{x})] - \mathbb{E}_{\mathbf{z}}[D(G(\mathbf{z}))] \tag{3.13}$$

where $\mathbb{D}$ is a set of 1-Lipschitz continuous functions. This value function gives the loss function of the critic

$$\begin{aligned} L^{(D)} &= -\mathbb{E}_{\boldsymbol{x} \sim p_{\text{data}}}[D(\boldsymbol{x})] + \mathbb{E}_{\mathbf{z}}[D(G(\mathbf{z}))] \\ &= -\mathbb{E}_{\boldsymbol{x} \sim p_{\text{data}}}[D(\boldsymbol{x})] + \mathbb{E}_{\hat{\boldsymbol{x}} \sim p_{\text{model}}}[D(\hat{\boldsymbol{x}})] \end{aligned} \tag{3.14}$$

and the generator

$$L^{(G)} = -\mathbb{E}_{\hat{\boldsymbol{x}} \sim p_{\text{model}}}[D(\hat{\boldsymbol{x}})] \tag{3.15}$$

To enforce the Lipschiz constraint in equation 3.13 Arjovsky *et al.* (2017) proposes to clip the weights of the critic to lie in a compact space $\mathbb{W} = [-c, c]$. Though this approach is simple and *"[...] clearly a terrible way to enforce a Lipschitz constraint."*(Arjovsky *et al.*, 2017, p. 7), it yields good empirical results.

3.5.5 Gradient penalty on Wasserstein GANs

Gulrajani *et al.* (2017) demonstrated the downsides of clipping the weights in Wasserstein GANs (WGANs) and found it to cause convergence failure and optimization difficulties in deep WGANs.

As an alternative approach a soft version of the weight clipping is proposed to encourage the critic to be 1-Lipschitz. This soft approach is motivated by the desire of having gradients with norm ≤ 1 everywhere for a 1-Lipschitz differentiable function. Gulrajani *et al.* (2017) proposes a gradient penalty to constrain the gradient norm of the critic's output with respect to its input. The penalty is evaluated randomly along straight lines between pairs of points $\{\boldsymbol{x}, \hat{\boldsymbol{x}}\}$, from p_r and p_g respectively, where the critic should have unit gradient. Evaluating the penalty along these lines effectively smooth out the space between the distributions p_r and p_g (Fedus *et al.*, 2017). The gradient penalty (GP) is

$$\text{GP} = \lambda \mathbb{E}_{\hat{\boldsymbol{x}} \sim p_{\text{model}}}[(\|\nabla D(\alpha \boldsymbol{x} + (1-\alpha)\hat{\boldsymbol{x}})\|_2 - 1)^2] \tag{3.16}$$

where $\alpha \sim U(0, 1)$ determine the random points to evaluate the gradient norm, and λ is the penalty coefficient (suggested to $\lambda = 10$). The GP term is added to the Wasserstein critic loss in equation 3.14 to enforce the Lipschitz constrain.

$$L_{\text{WGAN-GP}} = L_{\text{WGAN}} + \text{GP} \tag{3.17}$$

where L_{WGAN} is the discriminator's Wasserstein loss from equation 3.14. It should be noted that gradient penalty is not valid when batch normalization is used in the critic. This is because equation 3.16 penalize the critic's gradient norm with respect to each input independently, and not the entire batch.

3.6 Progressively growing GANs

The GANs considered in this section builds on the intuition that a complex mapping from a latent variable $\mathbf{z}$ to high-resolution (e.g. 1024^2) images $\hat{\boldsymbol{x}}$ is easier to learn step by step at different scales, rather than learning all scales simultaneously. This

idea is the same as Denton *et al.* (2015) used in LAPGANs (section 3.4.1), and others has experimented with in GANs (Durugkar *et al.*, 2016; Zhang *et al.*, 2017; Wang *et al.*, 2017).

The aforementioned GANs that build on this idea use multiple generators and discriminators that operate in a hierarchy or on different spatial resolutions. The progressively growing GAN (ProGAN) of Karras *et al.* (2017) uses only one discriminator and generator to produce high quality 1024×1024 images.

3.6.1 ProGAN architecture

The progressive growing of images start with a latent variable $\mathbf{z}$ drawn randomly from the distribution of the 512-dimensional surface of a 513-hypersphere, i.e. $\mathbf{z} \sim \mathbb{S}^{512} = \{\mathbf{z} \in \mathbb{R}^{512} : \|\mathbf{z}\| = 1\}$. The latent variable is used by the generator G to form a (4×4) resolution image. The discriminator[5] D is trained in turn on real and fake images of the same resolution as the generator produces. As training advances layers are added to G and D to incrementally increase the resolution of the images to (8×8), (16×16) up to e.g. (1024×1024) pixels. This is similar to the DCGAN approach (section 3.4.2), but G and D are trained separately on each spatial resolution before more layers are added. Models of the discriminator and generator as training proceeds are illustrated in figure 3.15. Details on how the new layers are introduced are presented in (Karras *et al.*, 2017).

3.6.2 Normalization and a remedy to mode collapse

After DCGAN's success (section 3.4.3) most GAN models use a variant of batch normalization (section 2.4.3) to prevent training instability due to escalation of signal magnitudes in the models, and to reduce the risk of mode collapse. The progressively growing GAN (Karras *et al.*, 2017) is designed for multiple loss functions including the Wasserstein distance (equation 3.14) with gradient penalty (equation 3.16) and is therefore not able to use batchnorm. ProGANs use different techniques to replace the need for batchnorm in the generator and discriminator.

3.6.2.1 Normalization

Though it has not been under much consideration up to this point, careful weight initialization is an important measure to achieve stable training in deep learning. In short, weight initialization influences i) training time and ii) risk of exploding or vanishing gradients. A popular approach is to initialize the weights randomly from

[5]Optionally "the critic" as ProGAN sometimes uses the WGAN-GP loss function from equation 3.14 and 3.16.

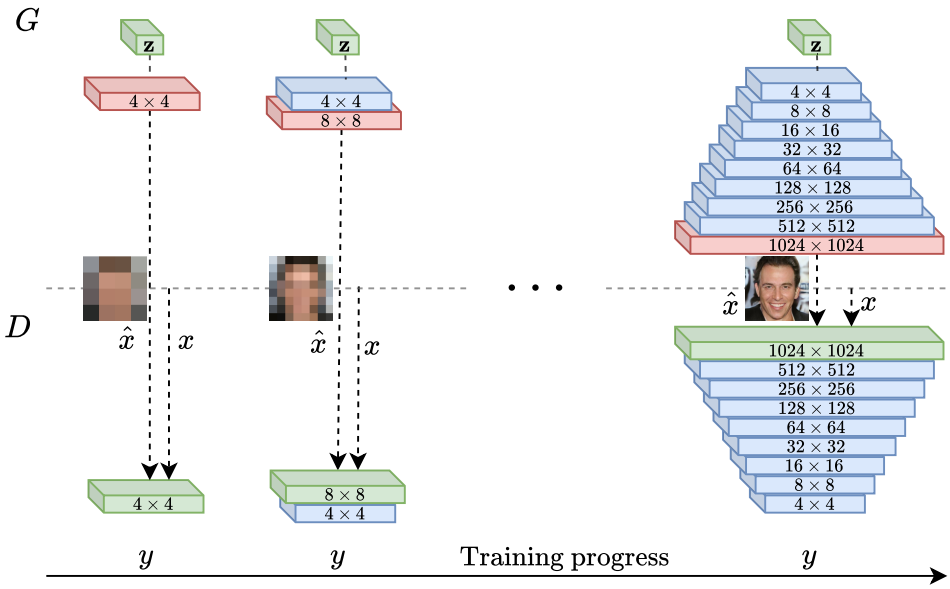


Figure 3.15: The architecture and training process in progressively growing GANs (ProGANs). The discriminator is trained on small resolution images (real $\boldsymbol{x}$ and fake $\hat{\boldsymbol{x}}$) before higher resolutions are added incrementally. The figure is reproduced from (Karras *et al.*, 2017).

$\mathcal{N}(0, 1)$ and scale them using He's initialization constant c (He *et al.*, 2015)

$$w_i' = c \cdot w_i, \quad \text{where } c = \sqrt{\frac{2}{n_{\text{inputs}}}} \tag{3.18}$$

where n_{inputs} are the number of inputs to the layer.

In ProGANs, instead of initializing the weights carefully, they are initialized from $\mathcal{N}(0, 1)$ and scaled using equation 3.18 each time the layer is ran. Karras *et al.* (2017) refers to this as **equalized learning rate** because it ensures that the dynamic range of all parameters is the same when using optimizers such as Adam and RMSProp. This equal dynamic range ensures that the time it takes to learn a parameter with a large range is the same as with a small range, helping speed up and stabilize the training.

In addition ProGAN normalizes the feature vector of each pixel (across all feature maps) to have unit length after each convolutional layer. This is to prevent exploding activations as a result of competition between the generator and discriminator. Karras *et al.* (2017) refer to this as **pixelwise normalization**, and it is performed

on the multi-channel image $\mathbf{A}$

$$\mathbf{A}'_{i,j,:} = \frac{\mathbf{A}_{i,j,:}}{\sqrt{\frac{1}{n}\sum_{k=0}^{n-1}(\mathbf{A}_{i,j,k})^2 + \epsilon}} \tag{3.19}$$

where n is the number of feature maps and $\epsilon = 10^{-8}$ is a small constant for numeric stability. It was found that this fairly heavy constraint on the generator did not limit its performance (Karras *et al.*, 2017).

3.6.2.2 Variation

To prevent mode collapse in ProGANs and to increase variation in the synthetic images Karras *et al.* (2017) builds on the idea of **minibatch discrimination** (Salimans *et al.*, 2016). The idea is that the discriminator examines multiple samples in combination, instead of just one by one, to prevent mode collapse. In ProGANs this idea constitutes by letting the discriminator have easy access to the average standard deviation of all pixel values across all channels of the images of each minibatch. When this statistic is computed it is concatenated as a constant feature map across the all samples in the minibatch. By doing this the discriminator can easily detect if the generator produces a minibatch of little variation (small standard deviation), and hence similar images. This encourages the generator to capture more variation of the data generating distribution (Karras *et al.*, 2017).

3.6.3 Restricting the discriminator

The progressively growing GAN was introduced with an additional loss term that was added to the WGAN-GP loss (equation 3.17). This additional loss was introduced to restrict the discriminator from drifting too far away from zero, thus stabilizing the training process (Karras *et al.*, 2017). The loss was introduced without further discussion of its importance but have been adopted by other GAN models that build on the ProGAN architecture. The drift term is given by

$$\text{drift} = \epsilon_{\text{drift}}\mathbb{E}_{\boldsymbol{x}\in\mathbb{P}_r}[D(\boldsymbol{x})^2] \tag{3.20}$$

where $\epsilon_{\text{drift}} = 0.001$. The complete loss becomes

$$L_{\text{drift}} = L_{\text{WGAN-GP}} + \text{drift} \tag{3.21}$$

3.7 Multi-scale gradient learning in GANs

The previously introduced DCGAN and ProGAN both learn the distribution of images by gradually increasing the resolution of images, thus increasing the

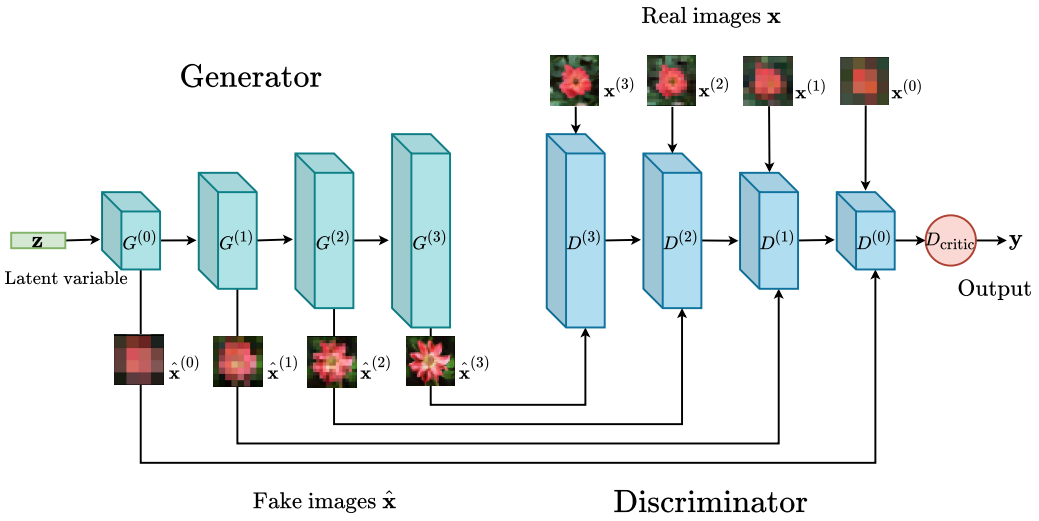


Figure 3.16: A model of a multi-scale gradient GAN of depth 4 based on the progressively growing GAN architecture. Each block of the generator G upscales and performs convolutions on the input volume. Each block of the discriminator downscales and performs convolutions on the input volume. The generator is forced to be able to produce RGB images at each resolution (equation 3.24) because the discriminator has access to these intermediate outputs. The connections between G and D allow the gradients to flow to G at multiple resolution scales during backpropagation. This results in faster convergence and more stable training.

complexity of the distribution. Their main difference is that ProGAN explicitly train the GAN on low resolution images before introducing higher resolution images, while DCGAN has only access to the full resolution targets while training.

In this section the concept of multi-scale gradient (MSG) based learning in GANs (Karnewar and Iyengar, 2019) is introduced, which allow the generator and discriminator to learn a distribution at multiple scales simultaneously. Intuitively the modification is a mix between the DCGAN approach, where training takes place in all scales simultaneously, and the ProGAN approach where the discriminator has access to real and fake samples at different resolution scales. A conceptual illustration model of the GAN architecture is illustrated in figure 3.16.

Consider the generator $G(\mathbf{z})$ of a multi-scale gradient GAN. It consists of k blocks $G^{(i)}$ that produce an output volume $\mathbf{A}^{(i)} \in \mathbb{R}^{2^{i+2} \times 2^{i+2} \times c}$ much like the DCGAN (figure 3.11), where c is the number of channels. The initial block $G^{(0)}(\mathbf{z})$ take in a latent variable e.g. $\mathbf{z} \sim \mathbb{S}^{512}$ and hence produce an image $G^{(0)} : \mathbf{z} \to \mathbf{A}^{(0)}$, where $\mathbf{A}^{(0)} \in \mathbb{R}^{4 \times 4 \times 512}$. In general $G^{(i)} : \mathbf{A}^{(i-1)} \to \mathbf{A}^{(i)}$ for $i \in \mathbb{N}$. Using this notation G

can be considered a composite function that produces a synthetic image $\hat{\boldsymbol{x}}$

$$G(\mathbf{z}) = G^{(k)} \circ G^{(k-1)} \circ \cdots G^{(i)} \circ \cdots G^{(1)} \circ G^{(0)}(\mathbf{z}) = \hat{\boldsymbol{x}} \tag{3.22}$$

Each block of the MSG-GAN generator must be able to produce an RGB ($c = 3$) image, so learning can take place at different scales simultaneously. Therefore a function r is defined so it converts each activation volume $\mathbf{A}^{(i)}$ to an RGB image $\hat{\boldsymbol{x}}^{(i)}$.

$$r^{(i)} : \mathbf{A}^{(i)} \rightarrow \hat{\boldsymbol{x}}^{(i)}$$

$$\text{where, } \hat{\boldsymbol{x}}^{(i)} \in \{\mathbb{R}^{2^{i+2} \times 2^{i+2} \times 3} | \, 0 \leq \hat{\boldsymbol{x}}_{i,j,k} \leq 1\} \tag{3.23}$$

$$\text{hence, } r^{(i)}(G^{(i)}(\mathbf{z})) = r^{(i)}(\mathbf{A}^{(i)}) = \hat{\boldsymbol{x}}^{(i)} \tag{3.24}$$

The discriminator is built up in a similar hierarchical fashion as the generator, only in reverse (figure 3.16). The initial block of the discriminator $D^{(0)}(\boldsymbol{x})$ takes in the highest resolution image (real or fake) and use a function $r'^{(k)}$, similar to r but inversely[6], to produce features from the raw RGB input. These features are handled by the block to produce an activation volume $\mathbf{B}^{(0)}$ where the height and width are downsampled by a factor of 2. To this activation volume the features $r'(\boldsymbol{x})$ from an RGB image of corresponding size is concatenated using a combine function ϕ, so $\mathbf{B}^{(1)'} = \phi(\mathbf{B}^{(0)}, r'^{(1)}(\boldsymbol{x}^{(k-1)}))$. In general the output activation volume $\mathbf{B}^{(j)'}$ from $D^{(j)}$, where $j = k - i$ is defined

$$\begin{aligned} \mathbf{B}^{(j)'} &= D^{(j)}(\phi(\mathbf{B}^{(j-1)'}, r'^{(j)}(\boldsymbol{x}^{(k-j)}))) \\ &= D^{(j)}(\phi(\mathbf{B}^{(j-1)'}, r'^{(j)}(\boldsymbol{x}^{(i)}))) \end{aligned} \tag{3.25}$$

The final block of the discriminator $D_{critic}(\cdot)$ computes the critic score based on $\mathbf{B}^{(k)'}$ which indirectly contains information of images on all scales due to the connections in equation 3.25. The discriminator as a composite function becomes

$$D(\boldsymbol{x}, \boldsymbol{x}^{(0)}, \ldots, \boldsymbol{x}^{(k-1)}) = D_{\text{critic}} \circ D^{(k)}(\cdot, \boldsymbol{x}^{(0)}) \circ D^{k-1}(\cdot, \boldsymbol{x}^{(1)}) \circ \cdots D^{0}(\cdot, \boldsymbol{x})) \tag{3.26}$$

In equation 3.26 where the discriminator takes in a new $\boldsymbol{x}^{(i)}$ there essentially is a connection between the generator and discriminator. These connections allow the gradients from the discriminator to flow to the generator at multiple scales simultaneously during backpropagation. This lets the generator be able to learn all scales simultaneously, and is not required to be trained to convergence on each resolution separately as in the progressively growing GAN (section 3.6).

[6]The function r' is not a real inverse of r, but conceptually they perform the opposite operations.

Karnewar and Iyengar (2019) shows experimentally that the MSG modification is robust to different base architectures, learning rates, loss functions[7] and training sets.

3.8 Final notes on GANs

Up to this point some major steps of the evolution of generative adversarial networks has been covered. Starting from the early GANs which were unstable and hard to train, moving forward to the success of the deep convolutional GANs, Wasserstein loss and progressively growing GANs.

These variants of GANs are individual steps towards achieving a more stable training for GANs. DCGANs (section 3.4.2) and ProGANs (section 3.6) that are from a family of convolutional GAN architectures, that build images from a low resolution and up, which gives them stability. The heuristically motivated non-saturating loss (section 3.2.2.2) and the more theoretically rooted Wasserstein GANs (section 3.5) with gradient penalty (section 3.5.5) explored better loss functions that provide a stronger gradient for the generator. And lastly the multi-scale gradient GAN (section 3.7) that allow the model to learn from multiple resolutions simultaneously, providing a more stable training.

It should be emphasized that other approaches towards stable GANs producing high quality high resolution images has been made. StyleGAN (Karras *et al.*, 2019b) builds on principles from neural style transfer (Gatys *et al.*, 2016) to develop a GAN architecture that automatically learn unsupervised separation of high-level attributes and stochastic variation in synthetic images. This architecture has further been improved with StyleGAN2 (Karras *et al.*, 2019a), adopting elements from techniques such as progressively growing (section 3.6) and multi-scale gradient learning (section 3.7), to provide state-of-the-art results.

In total all these architectures, techniques and approaches are remedies to training instability, vanishing and exploding gradients, and mode collapse that were the major setbacks of the original GAN framework. All these problems are not permanently solved yet (Wang *et al.*, 2020), but huge steps have been made in the right direction.

[7] If a loss function with gradient penalty (e.g. equation 3.16) is used the penalties are averaged over all inputs (Karnewar and Iyengar, 2019)

3.9 Evaluating generative models

As GANs have evolved and the quality of synthetic images have become increasingly better, the need for reliable quantitative methods for evaluating and comparing image quality has followed.

As the probability distribution p_g that the generator learns cannot be explicitly represented it is difficult to evaluate it directly. The early GAN papers used Gaussian Parzen windows to estimate the learned distribution so the performance could be judged using log-likelihood estimates. This method is unreliable for high-dimensional data, but it was the *"best method available"* at the time (Goodfellow *et al.*, 2014, p. 6).

Since then several different evaluation metrics have been proposed. Denton *et al.* (2015) used human evaluation of real and fake images to compare the performance of different GANs. Arjovsky *et al.* (2017) found that the Wasserstein distance (section 3.5.1) computed by the WGAN loss function corresponded reasonably well with image quality. A good performance metric for GANs must be reliable on different models and data types, as well as being computationally efficient to compute. Steps towards this goal has been made, but the problem of GAN evaluation is still considered open (Wang *et al.*, 2020).

3.9.1 Inception Score (IS)

One possible approach concerning images that gained some popularity is the Inception Score. To get an automatic and reliable evaluation of the generated images Salimans *et al.* (2016) proposed to use a pretrained image classifier on the generated images. The model they used was Google's Inception model[8] trained on ImageNet (Deng *et al.*, 2009) with images from 1000 classes. When evaluating each generated sample $\hat{\boldsymbol{x}}$ on the Inception model the probability of the sample being from each of the classes, the conditional label distribution $p(y|\hat{\boldsymbol{x}})$, is returned.

A well performing generator produces images with meaningful objects. Evaluated by the Inception model $p(y|\hat{\boldsymbol{x}})$ should have low entropy, corresponding to recognizable objects in the image. If an image has high entropy in the conditional label distribution, the classifier does not recognize any particular objects in the image, and returns a small probability for multiple classes.

A well performing generator is not only measured by the "objectness" of $\hat{\boldsymbol{x}}$, but also that it is able to reproduce the variety of the data generating distribution (section 3.1.2). Therefore the marginal distribution $p(y) = \int_{\mathbf{z}} p(y|\hat{\boldsymbol{x}} = G(\mathbf{z}))d\mathbf{z}$

[8]The Inception V3 model is a deep convolutional network that achieved state-of-the-art classification results on ImageNet (Deng *et al.*, 2009) in 2015.

should have high entropy. These two requirements are summed up by the proposed **inception score** (IS) (Salimans *et al.*, 2016)

$$\mathrm{IS}(G) = \exp(\mathbb{E}_{\boldsymbol{x}\sim p_g} D_{\mathrm{KL}}(p(y|\boldsymbol{x})\|p(y))) \tag{3.27}$$

Equation 3.27 compares the conditional label distribution with the marginal distribution of all labels using the Kullback-Leiber divergence. When comparing Inception score for different models a higher score is better, meaning that the samples have a high degree of "objectness" and are diverse over different classes.

When IS measures the variety of the images it only has access to the generated samples, and can thus only penalize the model for not producing all classes of the distribution. As IS do not have access to any real images, it cannot penalize the model for modes that are dropped within the classes.

3.9.2 Fréchet inception distance (FID)

An alternative to the Inception score that takes use of fake *and* real images and has proven to be robust to noise is the Fréchet inception distance (FID) (Heusel *et al.*, 2017). FID is like IS based on the Inception model, but instead of using the classifications from the output layer of the model, the activations of the last hidden layer are used. These activations are the 2048-dimensional features that carry important information about the recognizable characteristics of the image. The statistics of the activations from the generated images are, unlike in the IS score, compared with the statistics of the real images upon evaluation.

It is assumed that the activations from the last hidden layer follow a multidimensional Gaussian distribution so $P_r \sim \mathcal{N}(\boldsymbol{\mu}_r, \boldsymbol{\Sigma}_r)$ and $P_g \sim \mathcal{N}(\boldsymbol{\mu}_g, \boldsymbol{\Sigma}_g)$. The Frétchet distance (Fréchet, 1957) also known as the Wasserstein-2 distance[9] (Vaserstein, 1969) between the two distributions can be calculated using equation 3.28.

$$\mathrm{FID}(P_r, P_g) = \|\boldsymbol{\mu}_r - \boldsymbol{\mu}_g\|_2^2 + \mathrm{Tr}(\boldsymbol{\Sigma}_r + \boldsymbol{\Sigma}_g - 2(\boldsymbol{\Sigma}_r\boldsymbol{\Sigma}_g)^{\frac{1}{2}}) \tag{3.28}$$

where $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ are the mean and covariance matrix of the two distributions respectively. Tr is the trace of the matrix which computes the sum over the elements down the main diagonal.

The Frétchet Inception distance between the real and fake images has proven to correlate well to human evaluation of image quality (Heusel *et al.*, 2017; Lucic

[9]There is indeed a connection to the Wasserstein-1 distance considered in section 3.5.1. For the curious reader the difference is due to the Wasserstein-2 distance in addition uses the second moment when comparing two distributions. As the Wasserstein-1 distance has the "earth mover" intuition behind it, the Wasserstein-2 distance can be thought of as the minimum cord-length needed to join two forward-moving points on the two distributions together.

et al., 2018), and is therefore today the most popular metric for evaluating GANs. FID is, unlike IS, sensitive to the dropping of modes within a class, because FID compares the distribution of the generated samples with the distribution of the real samples.

Note that FID and IS are both metrics that only apply to image data. As variety in the distributions is an important characteristic to measure, both IS and FID are measured over 50 000 generated samples, and often the whole training set. The reliability of these measures for small training sets will be discussed in section 4.5.3.

Chapter 4

Experiments

Apart from the illustrative GAN example in section 3.2.3 there is up to this point only considered theoretical aspects and techniques of deep learning and GANs. Convolutional neural networks, that constitute the basis for synthetic image generation using deep convolutional GANs, have been studied along with the key theoretical and empirical aspects, challenges and solutions of GANs since the beginning in 2014.

The following chapter will continue to address the threefold objective of this thesis (section 1.4) only now taking a more empirical approach. Experiments are conducted to illustrate some key results and challenges of the deep convolutional GAN architecture and the progressively growing multi-scale gradient GAN. Different datasets from the GAN literature are used in the experiments, as the well as the foraminifera dataset that addresses this thesis' objective.

The first experiment is conducted to illustrate the success and failure of the DCGAN architecture using the MNIST, CelebA and Horse or human datasets. The second experiment is set up to find the most stable configuration for the MSG-GAN, and to compare performance with other models, using the CIFAR-10 dataset. Following, several experiments are conducted to explore the distributions that the MSG-GAN learns when it is trained on the foraminifera dataset both unconditionally and class conditionally. Two experiments are conducted to investigate artifacts and the quality of the images that is sampled from the learned distributions. The final experiment address the main hypothesis of this thesis, to test if synthetic image augmentation will improve the foraminifera classification model of Johansen and Sørensen (2020).

4.1 Preliminary experiments with a deep convolutional GAN

As a part of the exploration of GAN models a preliminary experiment is conducted to illustrate how the DCGAN architecture (section 3.4.2) can be used to produce images from a relatively simple dataset. This model is also used, without adjusting hyperparameters or the model's capacity, to investigate the stability and robustness of GAN training when using two more complex datasets. The main objective is not to achieve state-of-the-art results on either datasets but rather to gently introduce the task of synthetic image generation in this thesis.

4.1.1 Datasets

The main dataset of consideration is the MNIST dataset (LeCun *et al.*, 2010) consisting of 60 000 grayscale images of handwritten digits of resolution 28×28 pixels. This dataset has been analyzed countless of times, and has been used as a benchmark test for classification and image generation since its release in 2010.

The two more complex datasets are a selection of 2500 downscaled 28×28 RGB images of celebrity faces, *CelebA* (Liu *et al.*, 2015) and 500 horses from the *Horse or human* dataset (Moroney, 2019).

4.1.2 Experiment setup and implementation details

For this experiment a DCGAN is implemented to generate images of resolution 28×28. The image starts from a latent vector of length 100 that is sampled from a standard normal distribution $\mathcal{N}(0, 1)$. A fully connected perceptron layer produces $7 \cdot 7 \cdot 256 = 12\,544$ activations that are reshaped to 256 feature maps of size 7×7. These feature maps are upsampled using two (5×5) convolution transpose layers with strides of 2. All layers except the final employ batch normalization (section 2.4.3) and the leaky ReLU activation function (equation 3.8) with leakiness of 0.3. The final layer uses the hyperbolic tangent function (equation 3.7) as activation function.

The discriminator is a mirror image of the generator, only using convolutional layers instead of convolution transpose layers, and dropout (section 2.4.2) regularization (with $p = 0.3$) instead of batch normalization. The final discriminator layer use a fully connected layer with linear activation function.

The discriminator use the original GAN loss function from equation **??** and the generator use the non-saturating loss equation 3.4. The training is performed on minibatches of 256 samples for a total of 50 epochs.

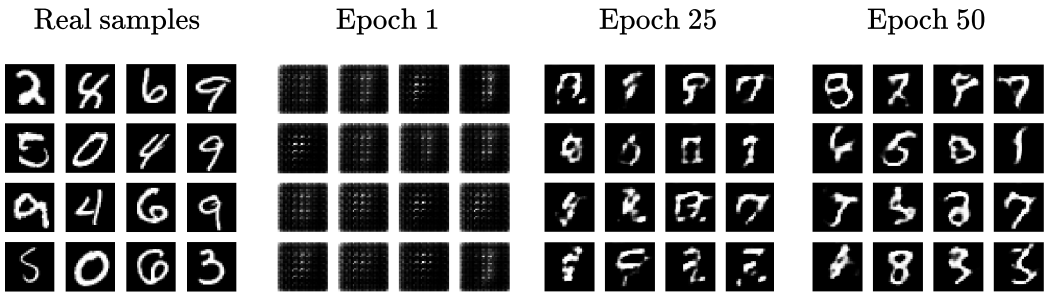


Figure 4.1: A small selection of real training data (left) from the MNIST dataset (LeCun *et al.*, 2010) and synthetic samples that are generated from the same latent variables at different stages of the training process. The checkerboard artifact can be observed early in the training are due to accumulations in the feature maps caused by overlap in the convolution transpose layers (Odena *et al.*, 2016) similar to figure 2.9.

4.1.3 Results

Here the results from the preliminary DCGAN experiments are presented. The Fréchet Inception distance and Inception score is not measured for any of the results as this experiment's objective is to introduce image generation with GANs. Figure 4.1 show a random selection of real samples from the MNIST dataset as well as generated images from various epochs during training. Figure 4.2 and 4.3 illustrate the instability of the DCGAN model as the generator collapse to one training sample.

4.1.4 Discussion

The results in figure 4.1 suggest that the DCGAN model has learned to reproduce the distribution of MNIST digits to some degree after 50 epochs. In the image from the first epoch of training on the MNIST dataset (figure 4.1) a checkerboard artifact can be observed in the generated images. These artifacts are due to the activations of the transposed convolutional layers accumulating in the feature maps (Odena *et al.*, 2016). This is especially prominent when transposed convolutions with strides > 1 are used in the generator, like in this DCGAN model. One can observe this accumulation also happening in figure 2.9. As training proceeds much of the checkerboard artifacts disappear as the generator learns filters that compensate for this accumulation.

As expected both experiments with color images collapsed to one mode during training. A plausible hypothesis is that the mode collapse is due to DCGAN's sensitivity to the choice of hyperparameters and model configurations. As the DC-

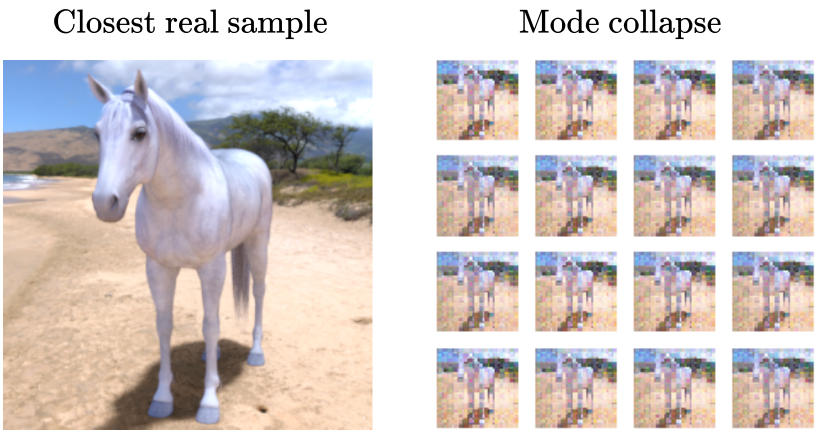


Figure 4.2: The result of training DCGAN on downscaled images of horses without tuning hyperparameters or model capacity. The generator has collapsed to one mode, that is easily recognized from the training samples. Note that all training samples were scaled down to 28×28 pixels before training.

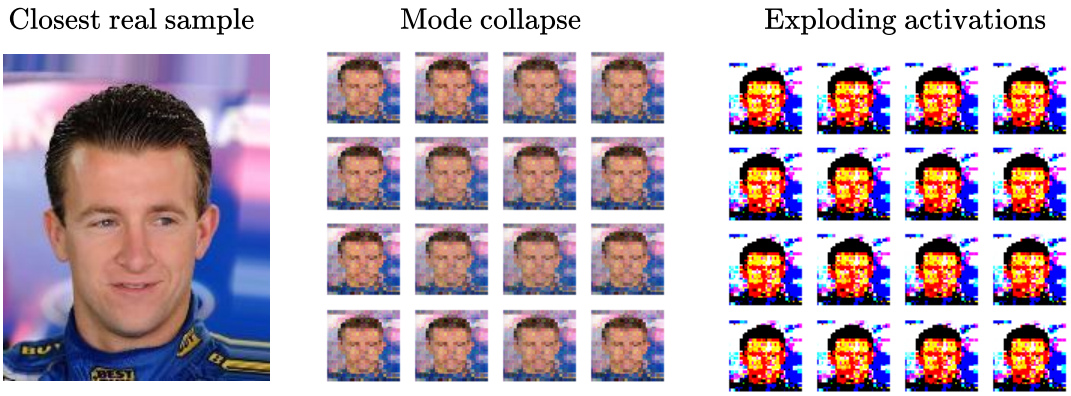


Figure 4.3: The result of training DCGAN on a selection of downscaled images from the CelebA dataset (Liu *et al.*, 2015). The generator has collapsed onto one mode of the distribution and is not able to produce anything else. As training proceeds, signs of exploding activations can be observed (right) due to competition of the models.

GAN model in this experiment have not been especially tailored for the complexity of these datasets training becomes unstable, and the generator only produce the image that fools the discriminator the most. Further investigations of why mode collapse happens have been presented in section 3.3.1.

As training proceeds in figure 4.3 (even though the generator has collapsed), the pixels become brighter and the contrast increases. This effect is possibly caused by exploding activations in the generator due to competition between the generator and discriminator (Karras *et al.*, 2017). One measure against this artifact is the pixelwise normalization from the progressively growing GANs section 3.6. The normalization is performed on all pixels across all feature maps of the generator.

A possible solution to why the generator does not recover from mode collapse might be the choice of loss function of the generator. Figure 3.3 illustrates how the gradients of the generator explodes as the discriminator's accuracy increases. Exploding gradients points in uninformative directions so the generator do not recover from the mode collapse (Salimans *et al.*, 2016; Arjovsky *et al.*, 2017).

4.1.5 Closing remarks

In this DCGAN experiment several interesting observations was made when training the same GAN on different datasets with different degree of complexity. On the least complex MNIST dataset the generator learned a meaningful distribution, but when the complexity of the datasets increased the model's performance dropped. By training the DCGAN on color images without adapting the capacity or hyper-parameters the model collapsed to one mode. Though the generator collapsed the results lead to valuable insight on the task of image-generation with GANs.

4.2 Method and setup of the multi-scale gradient GAN

One explicit goal of this thesis is to use a generative adversarial network to generate synthetic images of foraminifera. The foraminifera images are from a dataset of 2637 RGB color images of dimensions $224 \times 224 \times 3$. To generate realistic images of this dimensionality is a computationally demanding task that require a robust model and days of training on fast GPUs.

To avoid the need for additional hyperparameter searches and to reduce the risk of unstable training, the model that is chosen for this experiment is a multi-scale gradient GAN described in section 3.7 that builds on the progressively growing GAN architecture from section 3.6. This model has shown to provide more stable

training of the GAN, also when there are few images in the training set. Karnewar and Iyengar (2019) demonstrated the stability of an MSG-GAN on a dataset of 3000 256×256 RGB images. The foraminifera dataset consists of even fewer samples distributed over four classes, so generating realistic foraminifera images is nonetheless a challenging task.

Before tackling the problem of generating synthetic images the experimental setup and implementation details of the MSG-GAN is described and its performance is tested. The performance is demonstrated and compared with other models by generating synthetic images from the CIFAR-10 dataset familiar from the GAN- and computer vision literature.

4.2.1 Implementation details of the MSG-GAN model

The model that is presented here builds on the progressively growing GAN (Karras *et al.*, 2017) and MSG-GAN (Karnewar *et al.*, 2019). The implementation is similar to the MSG-ProGAN described in (Karnewar *et al.*, 2019) but implemented in `Tensorflow 2.1` using `Keras`.

The GAN consists of a generator and a discriminator that are represented by separate models. Each model consists of sequential blocks that perform the operations required to manipulate an input volume of a specific dimensionality. The models consist of k blocks that performs an upsample or downsample procedure followed by convolutional operations.

To be able to use the GAN on datasets with different resolutions blocks are added to, or removed from, the generator and discriminator respectively. Each additional block in the models results in doubling the resolution of the images. The number of blocks k in the models is referred to as the **depth**. The resolution of a generator of depth k is therefore $(2^{k+1} \times 2^{k+1})$. A detailed figure of the architecture of the generator and the discriminator is given in figure 4.4.

All layers of the models with learnable parameters are implemented to incorporate equalized learning rate (section 3.6.2.1), where the weights are scaled by $\sqrt{2/n_{\text{inputs}}}$ at runtime according to equation 3.18. All (3×3) and (4×4) convolutional layers in all blocks of both models use the leaky ReLU (equation 3.8) for activation function with a leakiness of 0.2. The final layer of the discriminator and generator and the (1×1) convolutions r and r', use linear activation functions. All convolutional layers use a stride of 1.

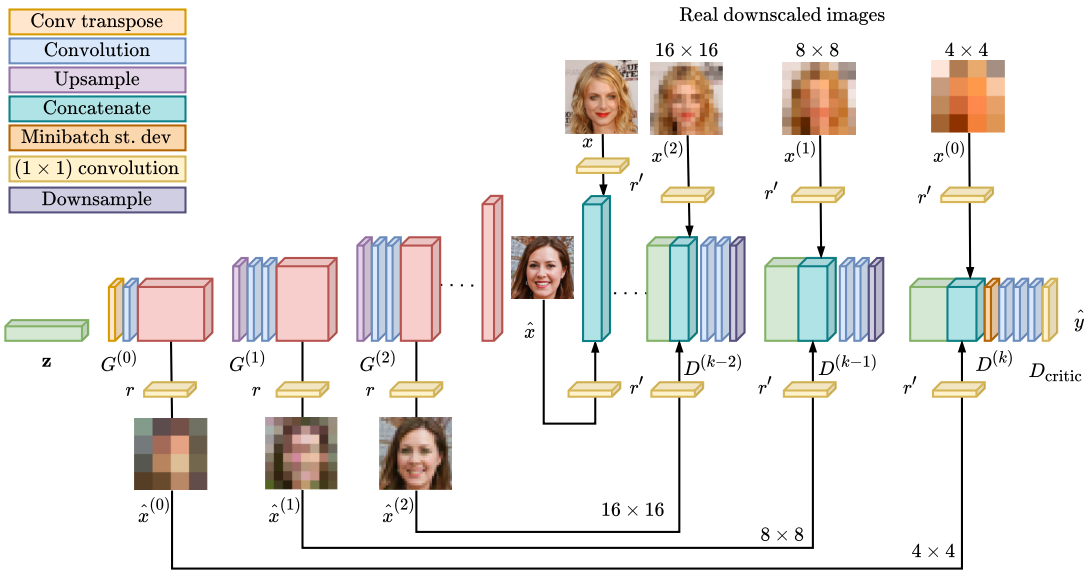


Figure 4.4: A detailed model of the implementation of a multi-scale gradient GAN built on the progressively growing GAN architecture. Note that there is no progressively growing of images like the original ProGAN as all scales are learned simultaneously like DCGAN.

4.2.1.1 The generator

The generator takes in a 512-D latent variable sampled from the surface of a hypersphere. This is achieved by sampling 512 random variables $z_i \sim \mathcal{N}(0, 1)$ to form a 512-D vector $\mathbf{z}$, and performing hyperspherical normalization

$$\frac{\mathbf{z}}{\|\mathbf{z}\|_2} \sim \mathbb{S}^{512} \tag{4.1}$$

(Muller, 1959). This latent variable serves as input to the generator's initial block that in turn propagates its activations forward through the network to create the final image.

The initial block of the generator transforms $\mathbf{z}$ to 512 feature maps of size (4×4) by a convolution transpose layer with kernel size (4×4) and "valid" padding. This is in turn sent to a (3×3) convolutional layer with "same" padding before pixel normalization (equation 3.19) is performed across all feature maps. All subsequent blocks of the generator consist of: an upsampling layer performing nearest neighbor interpolation to double the resolution of the image, followed by two (3×3) convolutional layers with "same" padding. Pixel normalization is performed after each (3×3) convolution to prevent exploding activations. Details and dimensions of each block is presented in table 4.1.

For each block of the generator there are also a corresponding function r referred to as **RGB converter** that produces an RGB image from the output volume of the block. The RGB converter consists of one (1×1) convolutional layer with linear activation function that learns the mapping from the c feature maps of each block to the 3 channels of RGB images. Note that this convolutional layer also incorporates equalized learning rate.

4.2.1.2 The discriminator

Each block of the discriminator takes in an RGB image of resolution $2^{(k+1)} \times 2^{(k+1)}$ and uses a (1×1) convolution layer r' with a linear activation function to convert the image to the desired number of feature maps. These features are concatenated with the activation volume of the previous discriminator block (if any). The resulting volume is subject to two (3×3) convolutional layers with "same" padding before the feature maps are downsampled using an average pooling layer, that computes the average of every (2×2) pixel grid, yielding feature maps of half the dimensionality. These feature maps are propagated forward to the next block of the discriminator and the same operations are applied.

Finally the activation volume reaches the final block of the discriminator. This block is similar to the ones previously described, only a minibatch standard deviation layer described in section 3.6.2.2 is applied before the final convolutions of (3×3), (4×4) and (1×1) produce the discriminator's final prediction $\boldsymbol{y}$. The dimensionality of the activations of all layers are given in table 4.2.

4.2.2 Implementation of the training loop

Generic GAN training is illustrated in figure 3.2 and section 3.2.3. Following is additional details on how this training procedure is implemented in the MSG-GAN setup for the following experiments.

The GAN training process consists of the following steps:

1. Sample real and fake data at all scales from the generator and training set respectively.
2. Train the discriminator
   (a) Compute the loss from fake samples and real samples (equation 3.14), the gradient penalty (equation 3.16) and the drift loss (equation 3.20).
   (b) Backpropagate to retrieve the gradients w.r.t. the weights
   (c) Apply gradients.

Table 4.1: A detailed description of the blocks of the MSG-GAN generator used in the experiments. The shape of the output volume of every layer is specified (excluding the batch size). Model 2 is used to produce images of resolution 128×128 and use all layers. Model 1 produce images of size 32 and use only blocks from "Model 1" and up.

Block	Operation	Activation	Output shape
	Latent variable	Hypersphere norm.	$1 \times 1 \times 512$
1	(4×4) conv transpose	LeakyReLU	$4 \times 4 \times 512$
	(3×3) convolution	LeakyReLU	$4 \times 4 \times 512$
	Pixelwise normalization	-	$4 \times 4 \times 512$
2	Upsample	-	$8 \times 8 \times 512$
	(3×3) convolution	LeakyReLU	$8 \times 8 \times 512$
	Pixelwise normalization	-	$8 \times 8 \times 512$
	(3×3) convolution	LeakyReLU	$8 \times 8 \times 512$
	Pixelwise normalization	-	$8 \times 8 \times 512$
3	Upsample	-	$16 \times 16 \times 512$
	(3×3) convolution	LeakyReLU	$16 \times 16 \times 512$
	Pixelwise normalization	-	$16 \times 16 \times 512$
	(3×3) convolution	LeakyReLU	$16 \times 16 \times 512$
	Pixelwise normalization	-	$16 \times 16 \times 512$
4	Upsample	-	$32 \times 32 \times 512$
	(3×3) convolution	LeakyReLU	$32 \times 32 \times 512$
	Pixelwise normalization	-	$32 \times 32 \times 512$
	(3×3) convolution	LeakyReLU	$32 \times 32 \times 512$
	Pixelwise normalization	-	$32 \times 32 \times 512$
	Model 1 ↑		
5	Upsample	-	$64 \times 64 \times 512$
	(3×3) convolution	LeakyReLU	$64 \times 64 \times 256$
	Pixelwise normalization	-	$64 \times 64 \times 512$
	(3×3) convolution	LeakyReLU	$64 \times 64 \times 256$
	Pixelwise normalization	-	$64 \times 64 \times 512$
6	Upsample	-	$128 \times 128 \times 256$
	(3×3) convolution	LeakyReLU	$128 \times 128 \times 128$
	Pixelwise normalization	-	$128 \times 128 \times 512$
	(3×3) convolution	LeakyReLU	$128 \times 128 \times 128$
	Pixelwise normalization	-	$128 \times 128 \times 512$
	Model 2 ↑		

Table 4.2: A detailed description of the MSG-GAN discriminator. Model 2 is used to critic images of resolution 128×128 and use all blocks. Model 1 is used for images of size 32 and use only the blocks from "Model 1" and down.

Block	Operation	Activation	Output shape $h \times w \times c$
		Model 2 ↓	
1	RGB input		$128 \times 128 \times 3$
	(1×1) convolution, r'	Linear	$128 \times 128 \times 64$
	(3×3) convolution	LeakyReLU	$128 \times 128 \times 128$
	(3×3) convolution	LeakyReLU	$128 \times 128 \times 128$
	Average pooling	-	$64 \times 64 \times 128$
2	RGB input	-	$64 \times 64 \times 3$
	(1×1) convolution, r'	Linear	$64 \times 64 \times 128$
	Concatenate	-	$64 \times 64 \times 256$
	(3×3) convolution	LeakyReLU	$64 \times 64 \times 256$
	(3×3) convolution	LeakyReLU	$64 \times 64 \times 256$
	Average pooling	-	$32 \times 32 \times 256$
		Model 1 ↓	
3	RGB input	-	$32 \times 32 \times 3$
	(1×1) convolution, r'	Linear	$32 \times 32 \times 256$
	Concatenate	-	$32 \times 32 \times 512$
	(3×3) convolution	LeakyReLU	$32 \times 32 \times 512$
	(3×3) convolution	LeakyReLU	$32 \times 32 \times 256$
	Average pooling	-	$16 \times 16 \times 256$
4	RGB input	-	$16 \times 16 \times 3$
	(1×1) convolution, r'	Linear	$16 \times 16 \times 256$
	Concatenate	-	$16 \times 16 \times 512$
	(3×3) convolution	LeakyReLU	$16 \times 16 \times 512$
	(3×3) convolution	LeakyReLU	$16 \times 16 \times 256$
	Average pooling	-	$8 \times 8 \times 256$
5	RGB input	-	$8 \times 8 \times 3$
	(1×1) convolution, r'	Linear	$8 \times 8 \times 256$
	Concatenate	-	$8 \times 8 \times 512$
	(3×3) convolution	LeakyReLU	$8 \times 8 \times 512$
	(3×3) convolution	LeakyReLU	$8 \times 8 \times 256$
	Average pooling	-	$4 \times 4 \times 256$
6	RGB input	-	$4 \times 4 \times 3$
	(1×1) convolution, r'	Linear	$4 \times 4 \times 256$
	Concatenate	-	$4 \times 4 \times 512$
	(3×3) convolution	LeakyReLU	$4 \times 4 \times 512$
	(4×4) convolution	LeakyReLU	$1 \times 1 \times 512$
	(1×1) convolution	Linear	$1 \times 1 \times 1$

3. Train the generator
   (a) Sample fake data from the generator.
   (b) Use the discriminator to compute loss of the fake samples (equation 3.15)
   (c) Backpropagate to retrieve the gradients w.r.t. the weights.
   (d) Apply gradients.

Note that the discriminator and generator are trained as two separate models. This is important for adversarial training, so the generator is not aiming for a moving target.

In the training loop the operations of backpropagation are performed using `TensorFlow`'s automatic differentiation software. This software "records" the operations, that is performed on the weights during training by the computational graph, so the gradients can be evaluated numerically. The fake samples retrieved at stage 1 must be detached from the computational graph so gradients does not flow through the generator when the discriminator is training. The complete and detailed training process of the GAN discriminator is illustrated in figure 4.5.

After the gradients are retrieved through automatic differentiation (backpropagation) the Adam optimizer (section 2.13) is used with learning rate of 0.003, $\beta_1 = 0$, $\beta_2 = 0.99$ and $\epsilon = 10^{-8}$ to apply the gradients.

For additional stability during training, a copy of the generator, referred to as the **shadow generator**, is kept outside the training loop and updated with an exponential moving average of the generator's weights during training (section 3.3.1). The shadow generator is kept for evaluation and ultimately used for generating synthetic images after training. The EMA is updated according to equation 3.6.

4.2.3 Technical details

The models are implemented in Python 3.7.4 with `TensorFlow 2.1` and `Keras`. All custom blocks and layers such as the generator- and discriminator blocks, equalized learning rate-layers and normalization layers are implemented by subclassing from `tf.keras.Model` and `tf.keras.layers.Layer`. This allows the implementation to benefit from the high-level API of `Keras` while having the freedom to change low-level implementation details such as weight scaling at runtime and making a custom training loop.

For computational performance the decorator `@tf.function` is used on computationally demanding tasks such as backpropagation and computing the gradient penalty. This decorator compiles a function block to a computational graph for

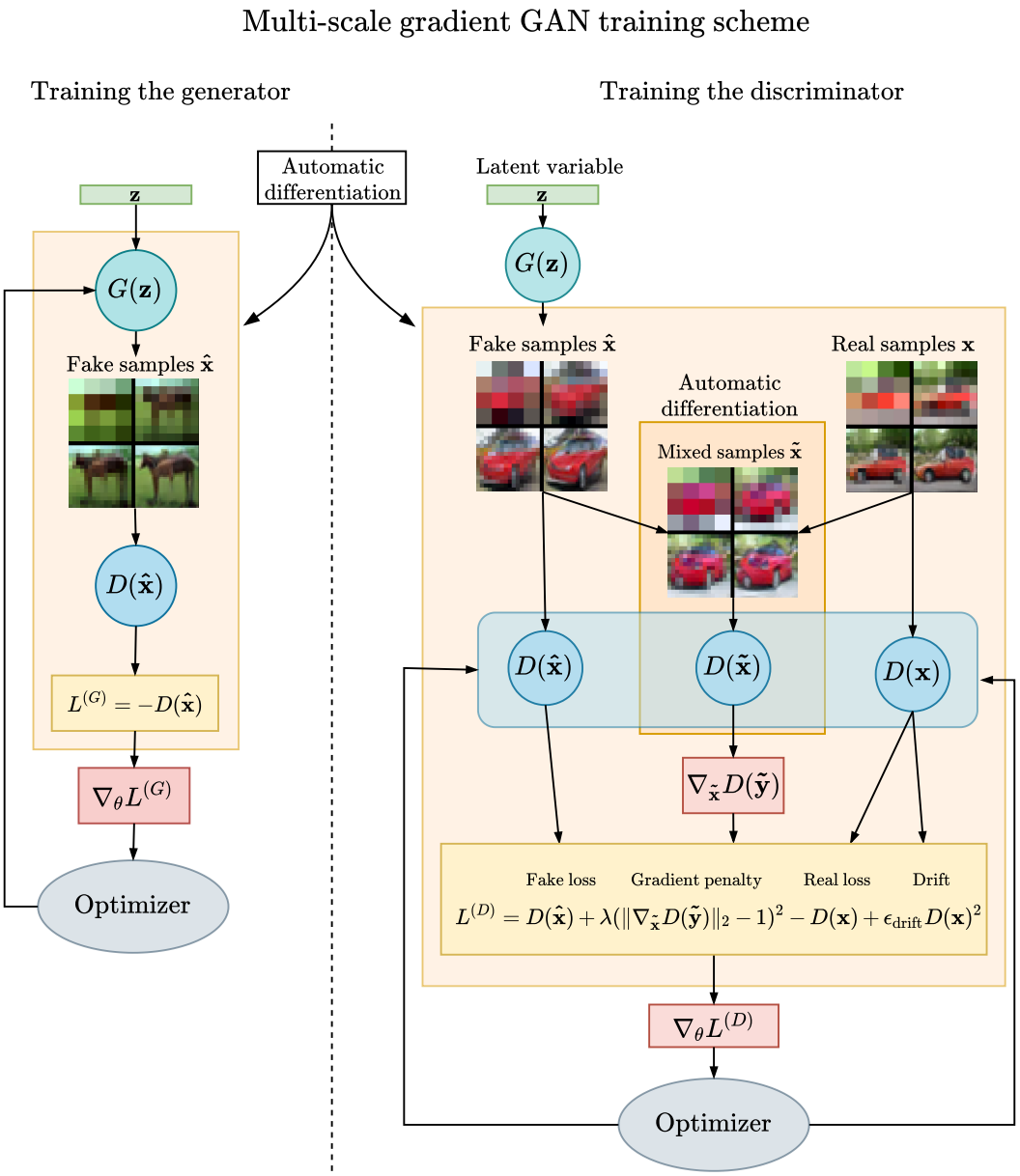


Figure 4.5: Illustration of the computational graph and training process of the discriminator and generator. Training D starts by sampling multiple resolution images from the G and training set. These samples are given to D to be evaluated and this makes up the fake and real loss of the loss $L^{(D)}$. The gradient penalty is computed by evaluating the norm of the gradients of the discriminator w.r.t. a mixture of real and fake samples. Training G is done generating fake samples and evaluating them using D. Automatic differentiation software capture the gradients for backpropagation during the training process. These gradients are applied to the models according to the Adam optimizer (equation 2.13).

faster execution on GPU.

The training loop is implemented as described in section 4.2.2 with `Tensorflow`'s function `tf.stop_gradients()` to detach the fake images from the computational graph when the discriminator is training. `Tensorflow`'s `tf.GradientTape` is used to record operations of the forward pass for automatic differentiation in backpropagation and gradient penalty. No preprocessing is performed on the training images except downscaling the images to the correct resolutions of the discriminator, and rescaling the pixel values to the range $[-1, 1]$. See appendix A for additional details on the source code.

The GAN training is performed on one Nvidia GPU with 11 GB RAM and on 8 CPUs with 32 GB of RAM.

4.3 Model validation and testing on real-world images

Before the model is trained on the foraminifera dataset it is tested by investigating its performance on the popular CIFAR-10 datasets, and comparing the results with other GANs. In addition, two different versions of the GAN setup is tested to find the best configuration, one with and one without EMA in the generator.

This experiment is an unconditional learning experiment for the GAN model. The generator will thus have no information of the image's class when it generates an image. The goal is that the generator will learn the distribution of the whole dataset including all classes. In such an unconditional experiment it is likely that the GAN will produce images reminiscent of mixtures of images from different classes.

4.3.1 The CIFAR-10 dataset

The model is tested on real-world images from the CIFAR-10 dataset (Krizhevsky *et al.*, 2009) that is a popular benchmark dataset for GANs. The dataset consists of a total of 60 000 32×32 RGB images evenly distributed over 10 different classes: *airplane, automobile, bird, cat, deer, dog, frog, horse, ship* and *truck*. All of the classes are exclusive, so no images overlap multiple categories, e.g. there are no pick-up trucks in either of the car or truck class.

Random selection of real CIFAR10 images

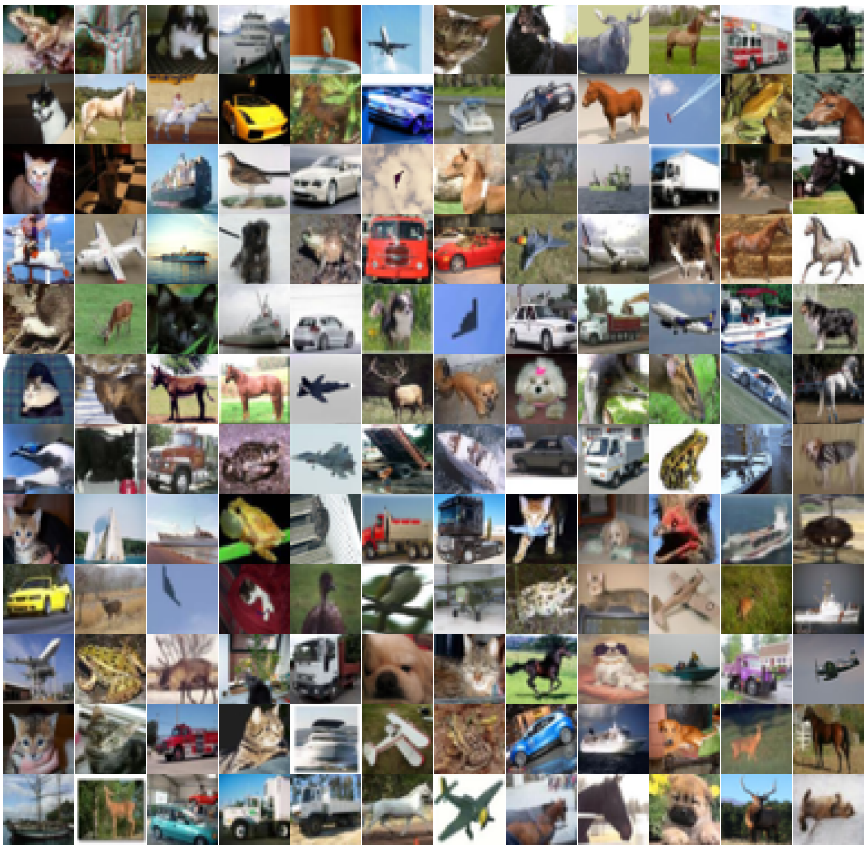


Figure 4.6: For comparison, 144 randomly selected CIFAR-10 images are presented. Each image display an object from one of the following classes *airplane, automobile, bird, cat, deer, dog, frog, horse, ship* and *truck*.

4.3.2 Experiment setup

For this experiment the multi-scale gradient GAN based on the progressively growing GAN described in section 4.2 is used. It implemented with a depth of 4 to produce 32×32 images and is trained on minibatches of size 32. The relatively small minibatch size is due to a limitation of computational resources.

To find the best evaluation configuration for the GAN, a model with exponential moving averages of weights, and one without, are tested. In **model A** no EMA is used, so the weights of the generator are always updated. In **model B** an EMA with a decay of 0.999 of the generator's weights is used when images are generated for the results.

Both models are trained for a total of 200 epochs, where each epoch the discriminators are shown all 60 000 real images. When the training is complete the discriminators have been shown a total of 12 million real images. To evaluate the models 50 000 images are generated from both models and evaluated using the Inception score (IS) and Fréchet Inception distance (FID). The code for evaluating samples using the Inception model is based on the library of `tensorflow_gan` that uses the official implementation of the Inception V3 model.

As a measure of stability and convergence of the GAN the mean squared error (MSE) is reported between images generated at all scales from the same seed of consecutive epochs as is suggested in (Yazıcı *et al.*, 2018). The MSE is averaged over 36 samples.

To illustrate some of the mappings that the generator has learned interpolations in the 512-D latent space of the generator visualized. The interpolations are performed by sampling two random points in the latent space and linearly interpolating across all dimensions. The interpolated points are normalized to lie on the surface of a hypersphere (equation 4.1) before they are used as input to the generator to yield images.

4.3.3 Results

Following the results of the aforementioned experiment with the MSG-GAN trained unconditionally on the CIFAR-10 dataset is presented. A random selection of generated images from model A and B are displayed in figure 4.7 and figure 4.8 respectively. Inception scores and Fréchet Inception distances are presented in table 4.3. A selection of interpolation transitions is displayed in figure 4.9.

Fake CIFAR-10 images from model A

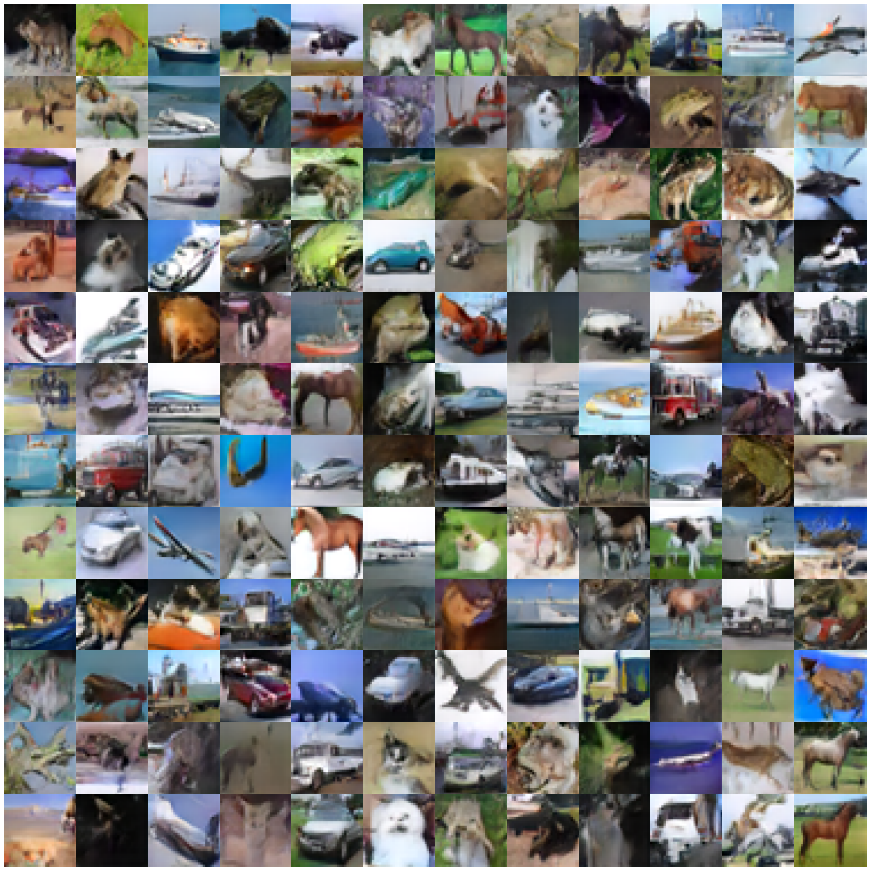


Figure 4.7: 144 randomly selected images generated from MSG-GAN model A, trained unconditionally on the CIFAR-10 dataset for 200 epochs. These images correspond to an inception score of 6.99.

Fake CIFAR-10 images from model B

Figure 4.8: 144 randomly selected images generated from MSG-GAN model B (with EMA in generator), trained unconditionally on the CIFAR-10 dataset for 200 epochs. These images correspond to an inception score of 7.17.

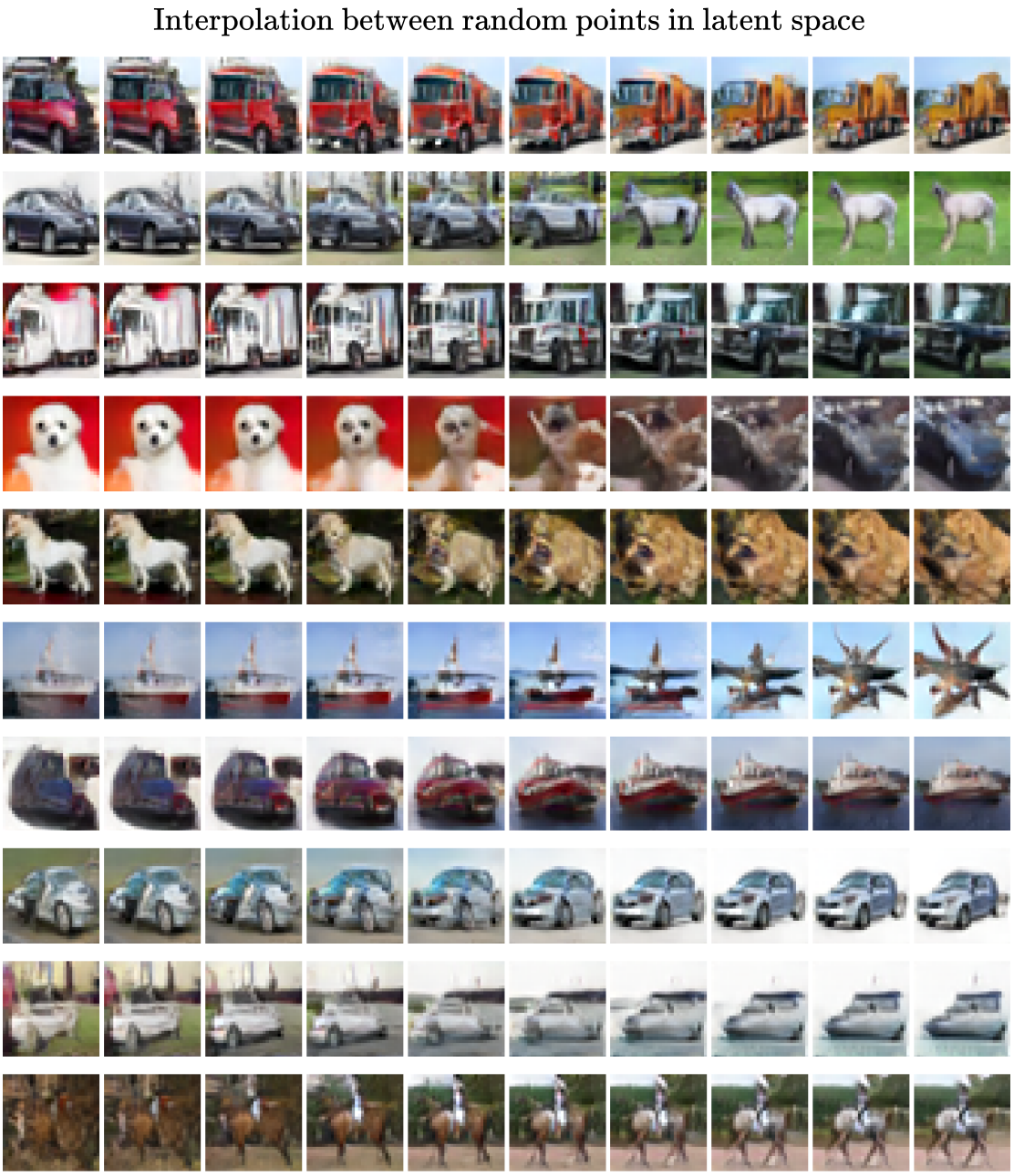


Figure 4.9: Interpolation between random points in latent space of model B (with EMA). This experiments show the smooth transitions between classes that the model has learned. The interpolated images change smoothly in both color and shape. It it not easy to recognize all "ghosts" caused by between-class transitions.

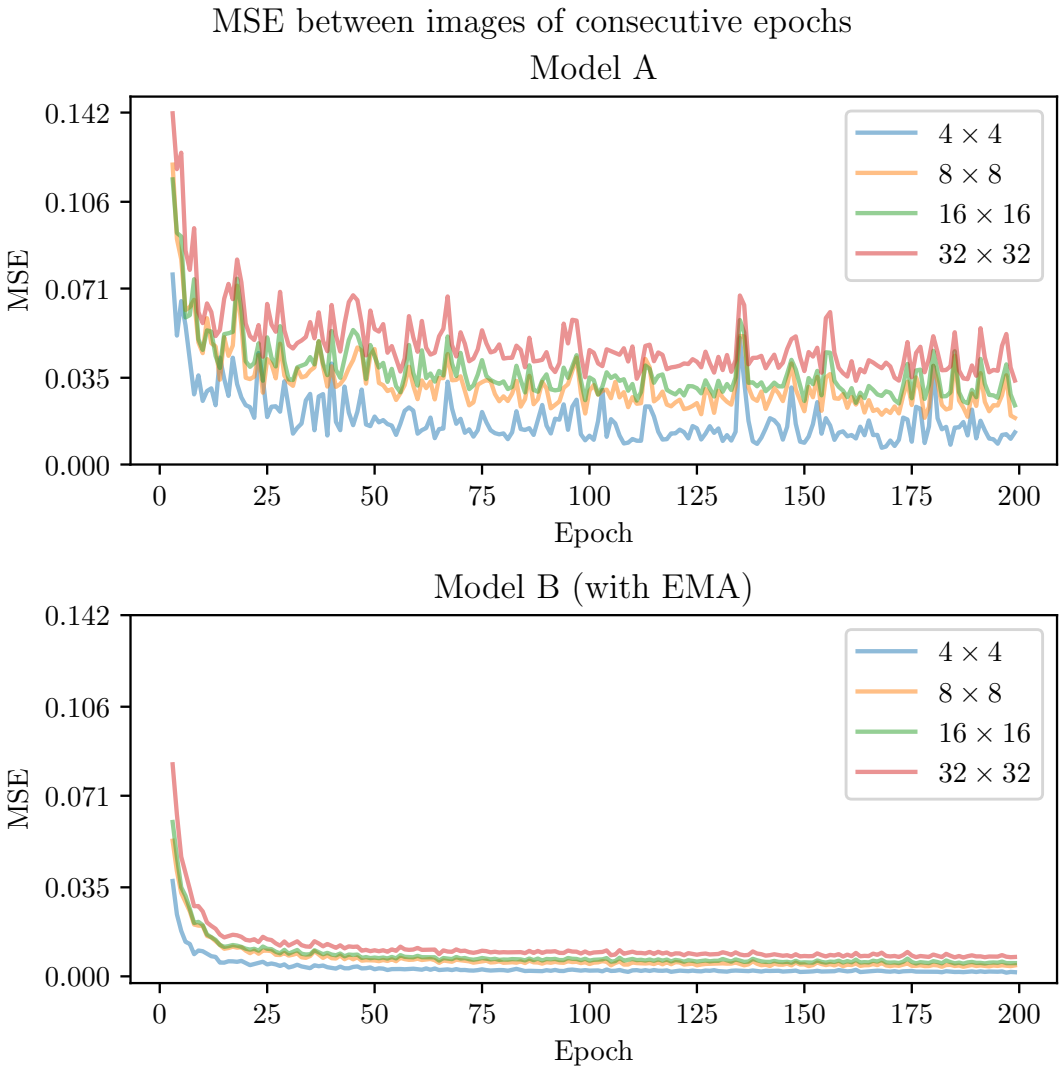


Figure 4.10: The plots display the mean squared error (MSE) between images generated from the same latent variables in consecutive epochs as a measure of stability. During the first few iterations there is much variation from epoch to epoch. As training proceeds the variation is somewhat constant apart from small variations in both models. As expected, there is less variation in the low-resolution images. The effect of exponential averaging over the generator weights becomes apparent when comparing model A and model B.

Table 4.3: Evaluation of model A and model B, compared to other recognized GAN methods. Inception score (IS), where a higher score is better, is the standard evaluation measure of the CIFAR-10 dataset. For FID a lower value is better.

Model	IS(↑)	FID (↓)
Real images	11.36	0
NCSN (Song and Ermon, 2019)	8.87	25.32
ProGAN (Karras *et al.*, 2017)	8.80	-
MSG-GAN (Karnewar *et al.*, 2019)	7.96	-
WGAN-GP (Gulrajani *et al.*, 2017)	7.86	29.30
Model B (with EMA)	7.17	26.32
Model A	6.99	31.72
Improved GAN (Salimans *et al.*, 2016)	6.86	-
DCGAN (conditional) (Radford *et al.*, 2015)	6.58	-

4.3.4 Discussion

A qualitative evaluation of image samples from the generators suggest that neither of model A or B suffers from mode collapse. This is also suggested by the reasonably low FID value, as FID is very sensitive to the dropping of modes (Lucic *et al.*, 2018). Many of the images from both models look plausible, although there are multiple images that are difficult to recognize as real objects.

When comparing the scores of model A and B from table 4.3 and the plot of mean-squared errors of figure 4.10 the effect of exponential moving averaging in the generator becomes apparent. The EMA causes both an improved performance in IS and FID, and more stable training. EMA smooths out the oscillations in GAN training, so the results become less dependent on the specific point the when training is terminated.

Both models A and B get an Inception score below both WGAN-GP (Gulrajani *et al.*, 2017), ProGAN (Karras *et al.*, 2017) and the original MSG-GAN (Karnewar *et al.*, 2019) that they were based upon. The lower IS could be due to scarcity of computational resources in the experiment. The GPUs that was used in the experiment allows only a batch size of 32 due to limited GPU memory, while e.g. WGAN-GP used hardware that allowed a batch size of 64. In the training of GANs, the batch size has a big impact on training results. Brock *et al.* (2018, p. 3) illustrated the *"tremendous benefits"* of increasing the batch size, showing that a batch size increased by a factor of 8 gave an increased IS of 46 %.

The IS measures the "objectness" of the images, and rewards images that are easy to define into one class. The slightly lower IS could therefore be due to the models

producing images that are a mixture of multiple classes, yielding "ghosts" that are difficult to classify as one class. This hypothesis is supported by the results from figure 4.9. Karras *et al.* (2017) achieved ProGAN's impressive IS of 8.80 by increasing the gradient norm target of 1 in the gradient penalty (equation 3.16) to 750, essentially making the discriminator a 750-Lipschitz continuous function that prefer much faster transitions between images (Karras *et al.*, 2017). This adjustment is not performed in the described CIFAR-10 experiment as the modification would requires computationally expensive hyperparameter searches.

Model B especially achieves relatively low FID scores. FID is not the most common measure for the CIFAR-10 dataset but is generally considered as the recommended measure of GAN images (Lucic *et al.*, 2018). Lucic *et al.* (2018) showed that FID is very sensitive to intraclass mode dropping, opposed to IS that only detects interclass mode dropping. In general FID measures how dissimilar the generated image distribution is to the real image distribution, considering mainly important visual artifacts. This suggests that even though the model did not capture the objectness in the images, it still learned important characteristics of the CIFAR-10 dataset, and that EMA of generator weighs contributed significantly.

4.3.5 Closing remarks

This experiment has demonstrated that the MSG-GAN model is able to learn important visual artifacts of the training distribution, despite the limited resources of the training setup. Model A achieves an IS of 6.99 and FID 31.72, while model B achieves 7.17 and FID 26.32. The experiment show that EMA in the generator gives several benefits concerning training and performance, and suggests that model B is a more suitable configuration for training in the following experiments with foraminifera.

4.4 Generating synthetic foraminifera unconditionally

The final objective of this thesis is to improve the accuracy and confidence of the foraminifera classifier of Johansen and Sørensen (2020) by using synthetic images of foraminifera. A first step towards this objective is to uncover the potential of GANs with the limited foraminifera dataset. To achieve this an experiment is conducted to train a GAN to learn the distribution of foraminifera unconditionally. The objective is to investigate how well a GAN is able to learn the distribution of the foraminifera dataset. Sampling from an unconditional distribution of foraminifera will not yield synthetic images that can be used for supervised learning (due to the

Table 4.4: The number of training samples in the different classes and splits of the foraminifera dataset.

	Agglutinated	Benthic	Planktic	Sediment	Total
Train	138	623	505	843	2109
Validation	18	78	63	106	265
Test	17	78	63	105	263
Total	173	779	631	1054	**2637**

lack of labels), but it will help uncover the potential of GANs applied to images of foraminifera.

4.4.1 The foraminifera dataset

The foraminifera dataset was introduced by Johansen and Sørensen (2020) as a part of their work towards developing a deep learning model that can detect and classify microscopic foraminifera. The dataset consists of specimen from sediment cores retrieved from the Arctic Barents Sea region, with sediments influenced by Atlantic, Arctic, polar and coastal waters. Foraminifera specimen and sediments were picked out in sizes ranging from 100 µm to 1000 µm and photographed through a microscope. The images were in turn refined and preprocessed to yield a dataset consisting of 2637 images of resolution 224×224 from four high-level classes; planktic, calcareous benthic, agglutinated benthic and sediment. The dataset was divided into training, validation and test set using an 80/10/10 split. The number samples of each class in the different dataset splits are presented in table 4.4 and figure 4.11 show samples from the different classes.

4.4.2 Experiment setup

For the unconditional foraminifera GAN experiment the MSG-GAN model described in section 4.2 is used with a depth of $k = 6$. Exponential running average technique (section 3.3.1) is used as well, as this has shown to yield higher quality images when generating synthetic images (section 4.3).

The model used in this experiment has the exact same configuration as model B that was used in the CIFAR-10 experiment (section 4.3), only with two additional blocks added in the discriminator and the generator. This configuration yields images of resolution 128×128. Ideally the generated images would be of resolution 224×224, as this is the original resolution of the foraminifera dataset, but due to limitations on GPU memory it is only possible to produce images of size 128×128.

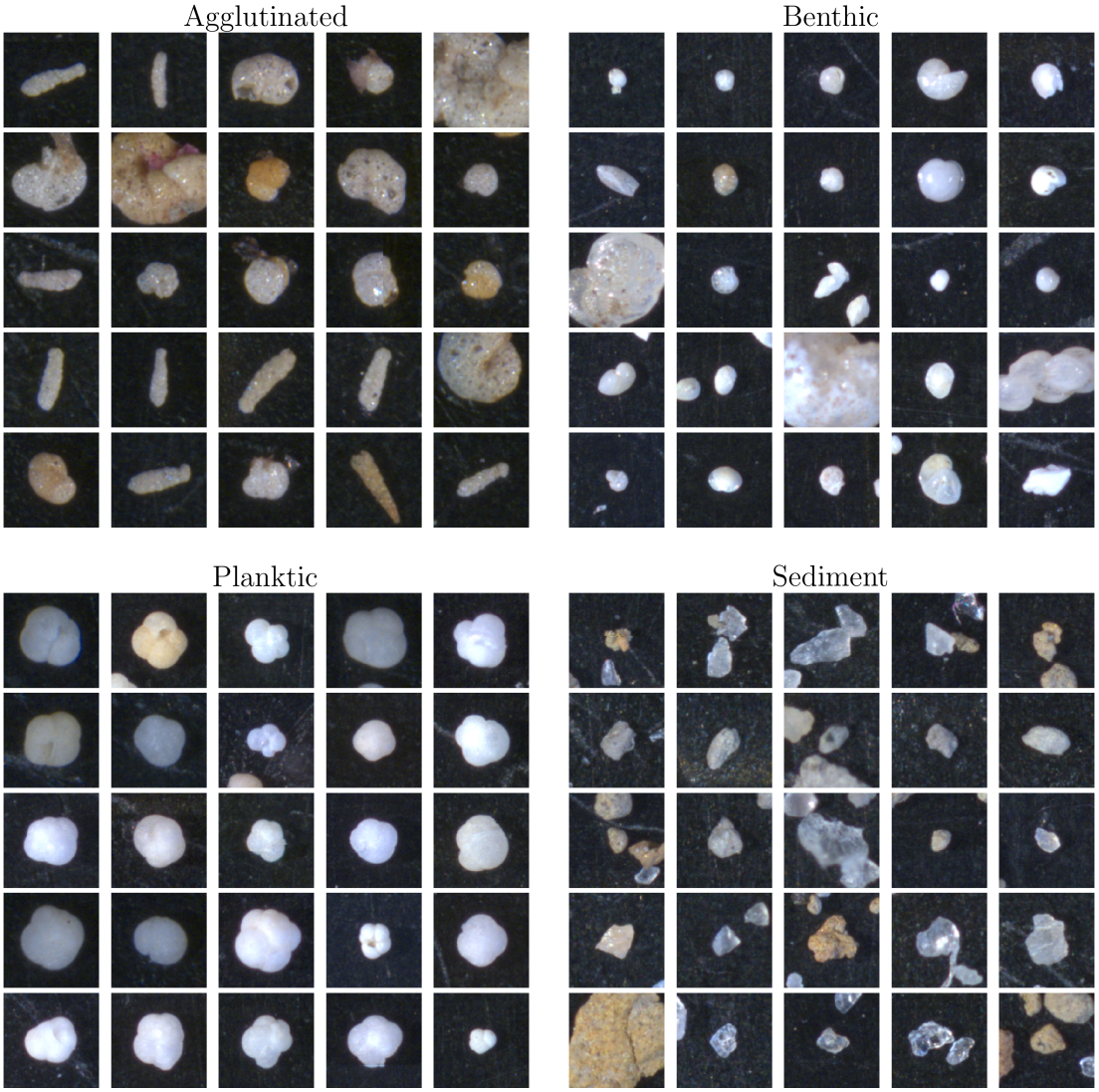


Figure 4.11: A random selection of 25 image samples from each of the four classes of the foraminifera dataset: *agglutinated*, *benthic*, *planktic* and *sediment*. The images are scaled down to resolution 128×128.

At this resolution a batch size of 8 is used.

As the unconditionally generated images will not be used for supervised training of the foraminifera classifier, and to illustrate the full potential of GANs on foraminifera, all 2637 images from the foraminifera dataset is used for training, including the validation set and test set. All images are scaled down so the highest resolution is 128×128, and the pixel values are centered to the range $[-1, 1]$. No additional preprocessing or data augmentation is applied to the images.

For evaluation of the learned distribution 50 000 fake images are generated to measure the Fréchet Inception distance to the distribution of all training samples. For comparison the FID bias is measured for the distribution of training images using a similar method as Lucic *et al.* (2018): the training images are divided randomly into two sets and the FID between the sets are measured. This process is repeated 50 times, and the mean and standard deviation is reported.

To visualize the GAN training on the foraminifera dataset images at all scales from the same latent vector is evaluated during training. In addition, the MSE between images of consecutive epochs are reported to check for convergence. To visualize the distribution the GAN has learned a linear interpolation is performed between random points in the latent space.

4.4.3 Results

The GAN was trained for a total of 760 epochs over 42 hours, and shown in total 2 004 120 real images of foraminifera. A random selection of synthetically generated images are presented in figure 4.12. The unconditionally generated foraminifera measured a FID to the training data of 47.1 and the bias of the distribution is estimated to 11.58 ± 0.13. Figure 4.13 shows how the images generated at all scales from the same latent vector evolve during training, and figure 4.15 shows that the MSE stabilizes as training proceeds. Figure 4.14 shows the generated images from 10 interpolations between random points in the latent space of the unconditionally trained GAN.

4.4.4 Discussion

The results of figure 4.12 show random samples from the distribution that the generator has learned. The distribution seems to contain a large variety of samples, with realistic looking samples that could originate from all four classes. Figure 4.14 show that the generator has learned smooth transitions between images. Smooth image transitions when interpolating in latent space suggests that the generator has not "memorized" the real samples, but rather learned the mapping to a continuous

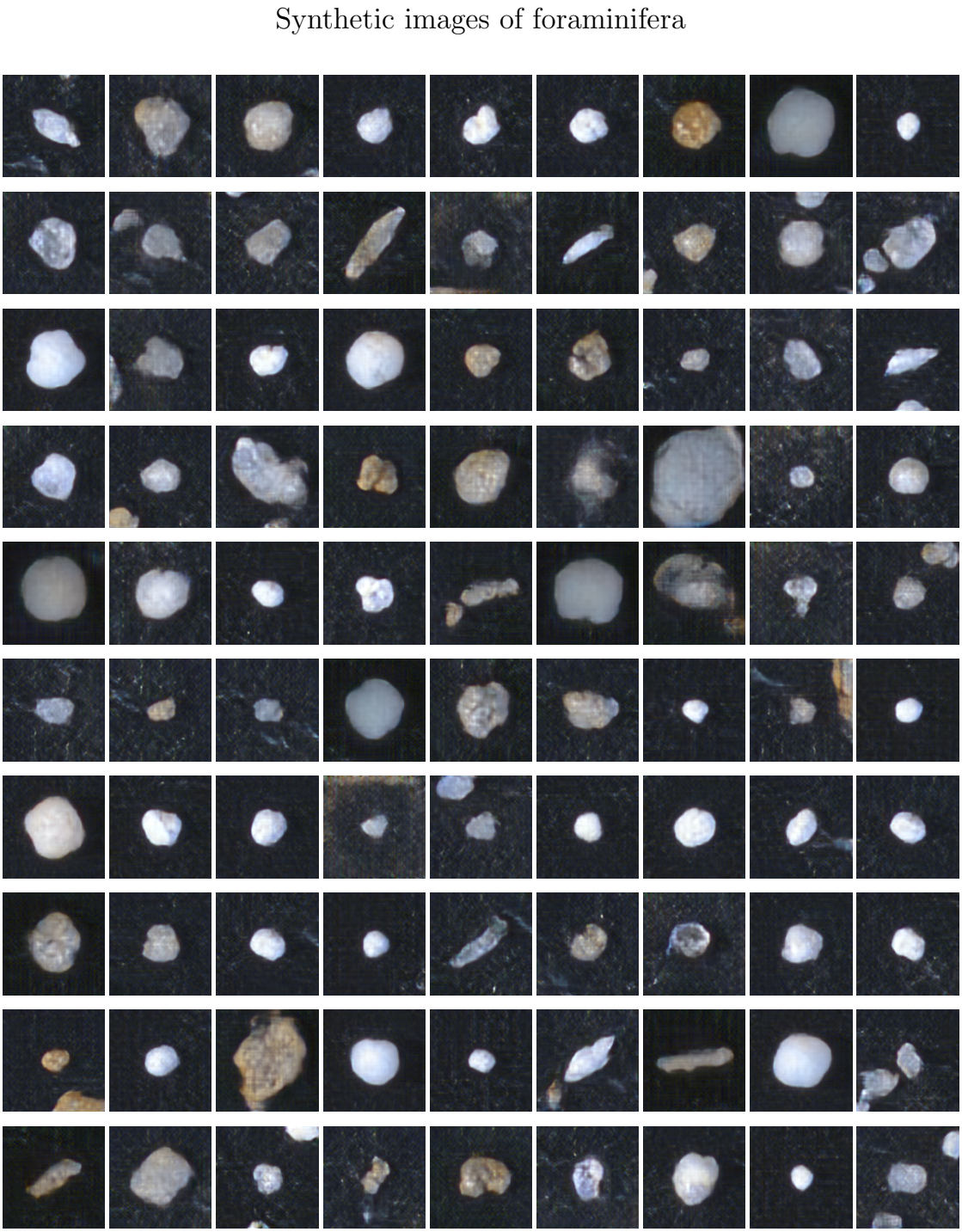


Figure 4.12: A random selection of unconditionally generated images of foraminifera after the GAN is trained for 760 epochs, and the discriminator is shown approximately 2 million real images.

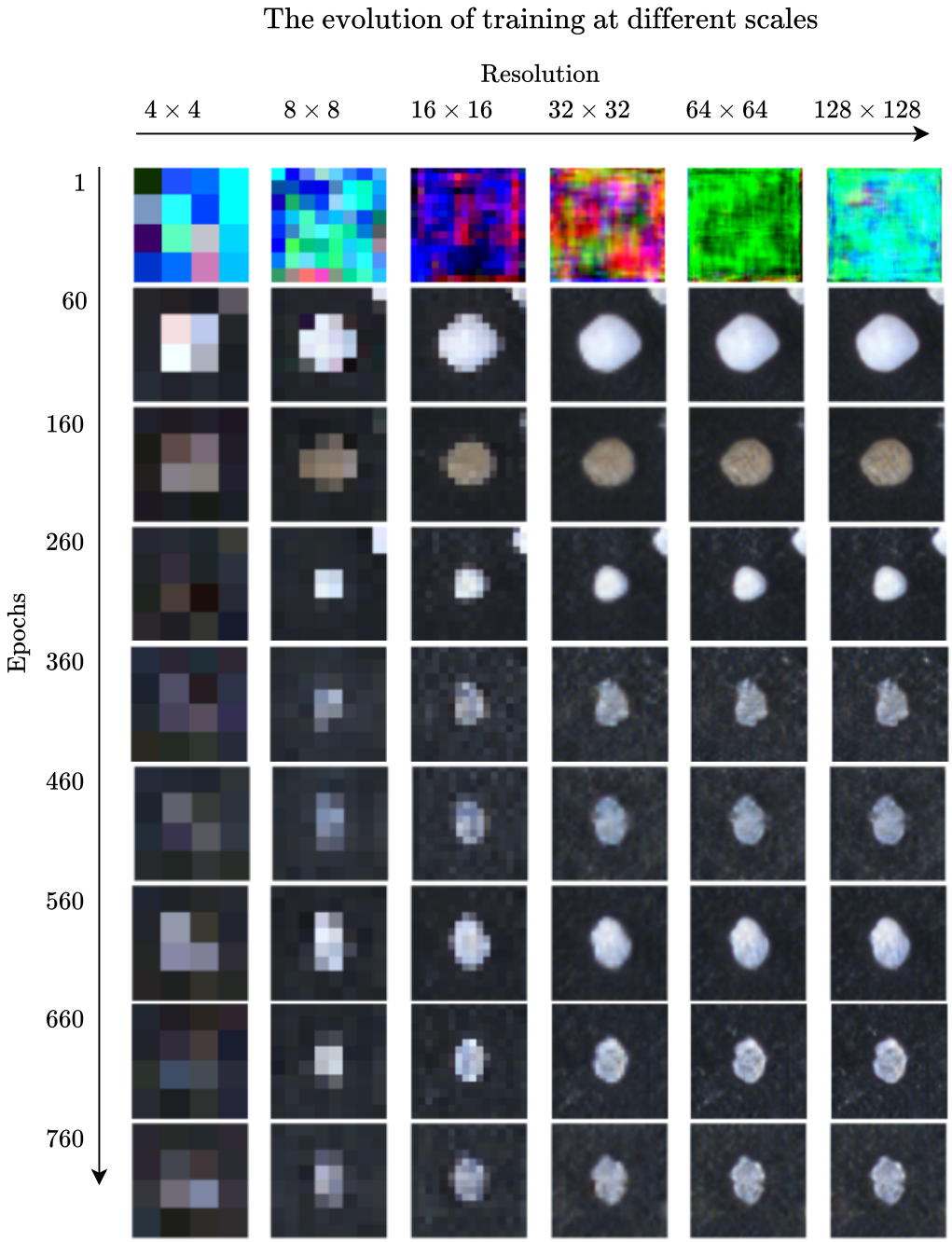


Figure 4.13: Images evaluated at different scales during training of the MSG-GAN unconditionally on the foraminifera dataset. In the beginning of training the images synchronize in color at all resolutions. As training proceeds the images at higher resolutions stabilize to one kind of sample and increasing amounts of detail are added.

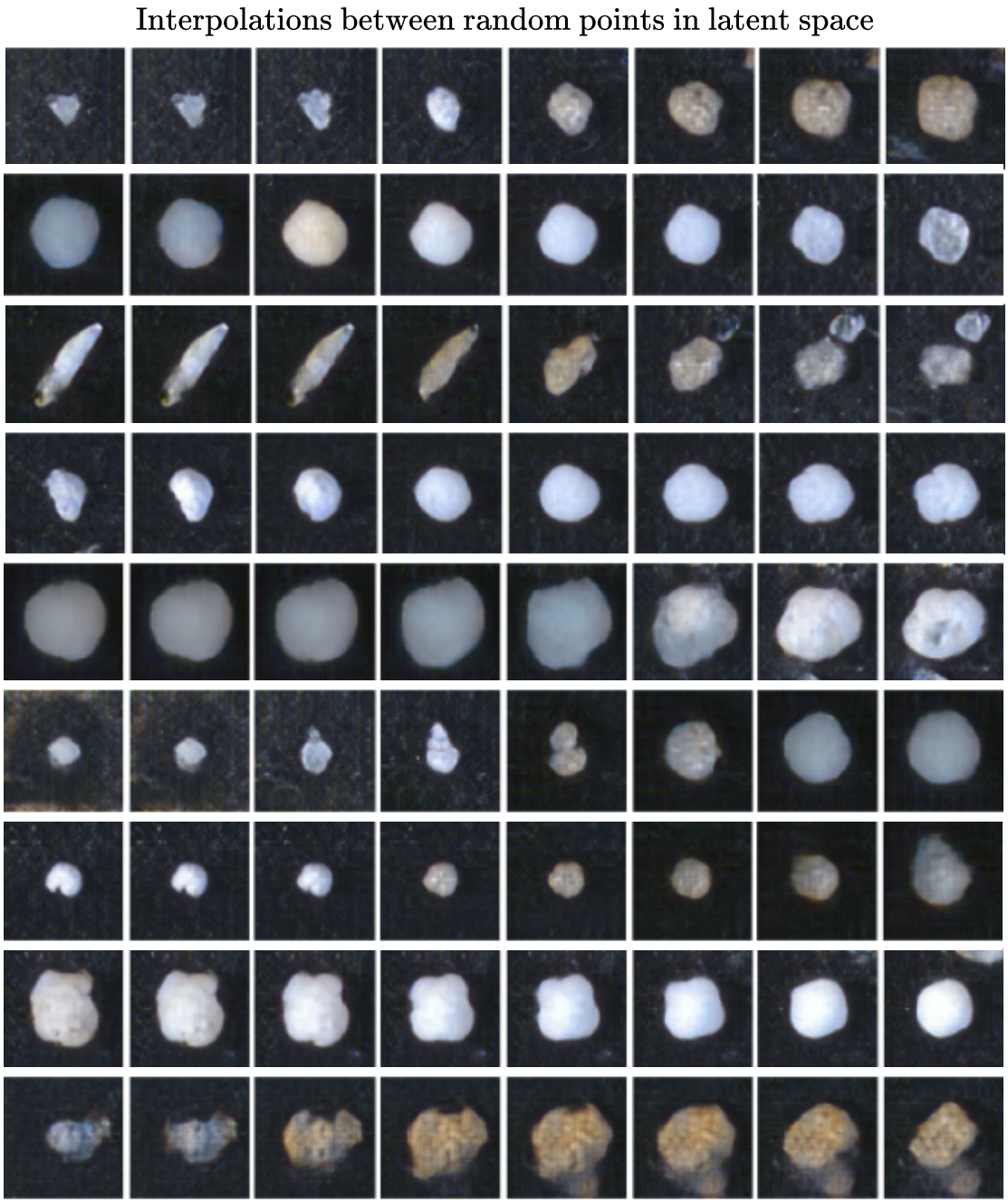


Figure 4.14: Synthetic images produced from interpolations in latent space of the unconditionally trained GAN on the foraminifera dataset. Every row shows the transition between the images of two random points. The generator has learned to produce smooth transitions between different species, sizes, textures, colors, shapes and numbers.

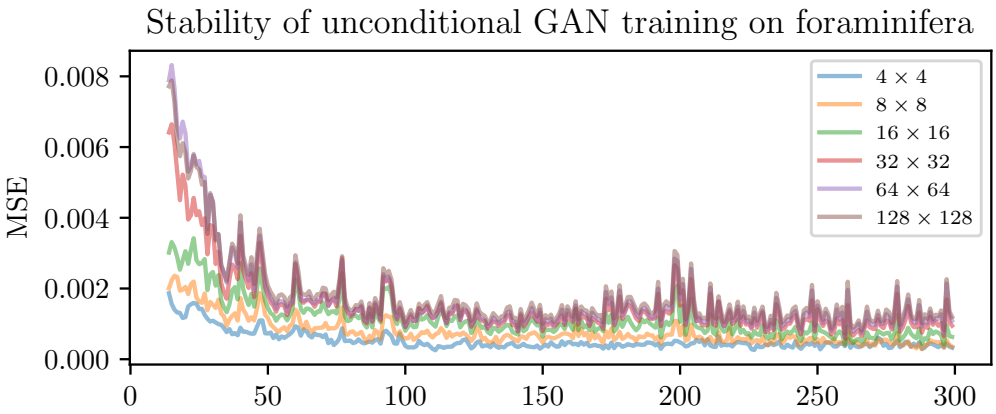


Figure 4.15: A plot of the mean squared error between images generated from the same latent variables from consecutive epochs. The plot shows how the images stabilize during training. The low resolution images stabilize faster as it is easier to learn their distribution. Note that the plot only shows MSE between epoch 15 and 300 as values before epoch 15 was too large for the scale, and values after 300 did not provide any new information.

distribution (Radford *et al.*, 2015).

The FID of 47.1 indicate that the generator suffers from a low degree of collapsed modes, and that it has learned characteristics of the distribution that are similar to the original distribution. The smooth transitions between generated images will sometimes result in "ghosts" that do not resemble any of the true classes. Some of these ghosts can be found in figure 4.12 and figure 4.14 (e.g. bottom row) and is likely to contribute to a higher FID value, as they do not exist in the original distribution.

Figure 4.13 show the training process and how the generator's mapping from one latent point to an image evolves. In the beginning (epoch 1) it produces only noise, when the parameters are not learned. During the first epochs the generator synchronizes the color of images at all resolution scales. After approximately 300 epochs the generator seems to settle for something that could originate from the sediment class. The stabilization of training is also suggested by the averaged MSE between images during training in figure 4.15. In the first few epochs the generator produces very different images, before the fluctuations in MSE settle between 0 and 0.002. This stabilization can be observed in the higher resolutions from epoch 360 to 760 in figure 4.13.

The MSE between images never reached 0 indicating that the generator had not yet converged when the training was stopped after 760 epochs. As elaborated in section 3.3.1 convergence in GANs is not guaranteed and small datasets does not

make it easier. It is possible that the generator would converge eventually, but when the measured MSE more or less has stabilized there is no way of knowing when the best time to stop training is. This has been a common and unsettled problem in GANs since the beginning, and only recently have there been any advances towards a better convergence measure (Grnarova *et al.*, 2019). More on this discussion later. For this experiment showing the discriminator 2 000 000 real images, yielded sufficient results to show the potential of GANs with foraminifera.

4.4.5 Closing remarks

This experiment is conducted to investigate how well a GAN is able to learn the distribution of the foraminifera dataset. The results were that the implemented MSG-GAN was able to learn the unconditional distribution of foraminifera. Though the resulting images of foraminifera are not perfect, containing e.g. artifacts from mixed species and classes, this experiments show the potential of using a generative adversarial network to produce synthetic images of foraminifera.

4.5 Generating foraminifera conditionally

As a measure to avoid artifacts from the mixing of characteristics of different species, and to produce foraminifera that can be used for supervised training, the following experiment is conducted to generate images of foraminifera class conditionally. The benefit of avoiding the mixture of species characteristics comes at the expense of more limited training data. On the other hand, the limited training data will be more homogeneous so the distribution may be easier to learn.

4.5.1 Hypothesis and experimental setup

This is the setup for a class conditional experiment with the MSG-GAN described in section 4.2 and each class of the foraminifera dataset (table 4.4). The hypothesis is that images generated from the class conditional distributions will be qualitatively more similar to the images from the real distributions of foraminifera classes than the unconditional images, and hence yield a lower FID.

This class conditional experiment is conducted in a similar fashion as the unconditional foraminifera experiment, only now one MSG-GAN is trained for each dataset. The GAN will be trained on the training and validation splits of the foraminifera dataset, however the test images are spared for later classification experiments. In lack of a more stable evaluation measure for small datasets[1] the Fréchet Inception

[1] A more in depth discussion on this will follow in section 4.5.3

Table 4.5: Fréchet Inception distance measures between the learned distributions and the training set. The FID scores in parenthesis are considered somewhat unreliable as they are measured from training sets containing < 2048 samples.

	Real images shown	FID (↓)
Unconditional foraminifera	2 M	47.1
Benthic (conditional)	1 M	(52.8)
Planktic (conditional)	1 M	(52.6)
Agglutinated (conditional)	300 k	(52.6)
Sediment (conditional)	1 M	(42.0)

distance is measured between 50 000 fake samples and the training samples of each class.

As the individual classes of the dataset contains a different amount of images, and as the risk of non-convergence is high, it would not yield a fair comparison to fix the number of training epochs. Instead this experiment fixes the number of real images the discriminator is shown, similar to how results are compared in (Karras *et al.*, 2017, 2019b; Karnewar and Iyengar, 2019). Each GAN is trained so the discriminator is shown approximately 1 million real samples, except the agglutinated GAN that is shown 300 000 real images. The reason for this is to avoid the discriminator to overfit to the significantly smaller training set (156 against 701, 568, 949). GAN overfitting on small datasets will be investigated further in section 4.6.

This experiment as well employs the MSG-GAN model with a depth of 6, exponential moving averages in the generator and a batch sizes of 8.

4.5.2 Results

Figure 4.16 show 25 randomly generated images from the generators trained on the training and validation splits of each class of the foraminifera dataset. Figure 4.17 and 4.18 shows illustrates some of the learned distributions by sampling from interpolations between random points in the latent space of the generators. The resulting FID scores are given in table 4.5.

4.5.3 Discussion

Figure 4.16 suggests that none of the conditional GANs collapsed during training despite the small datasets of each class. This result is rather surprising considering the very limited amount of training data each GAN had access to. The produced

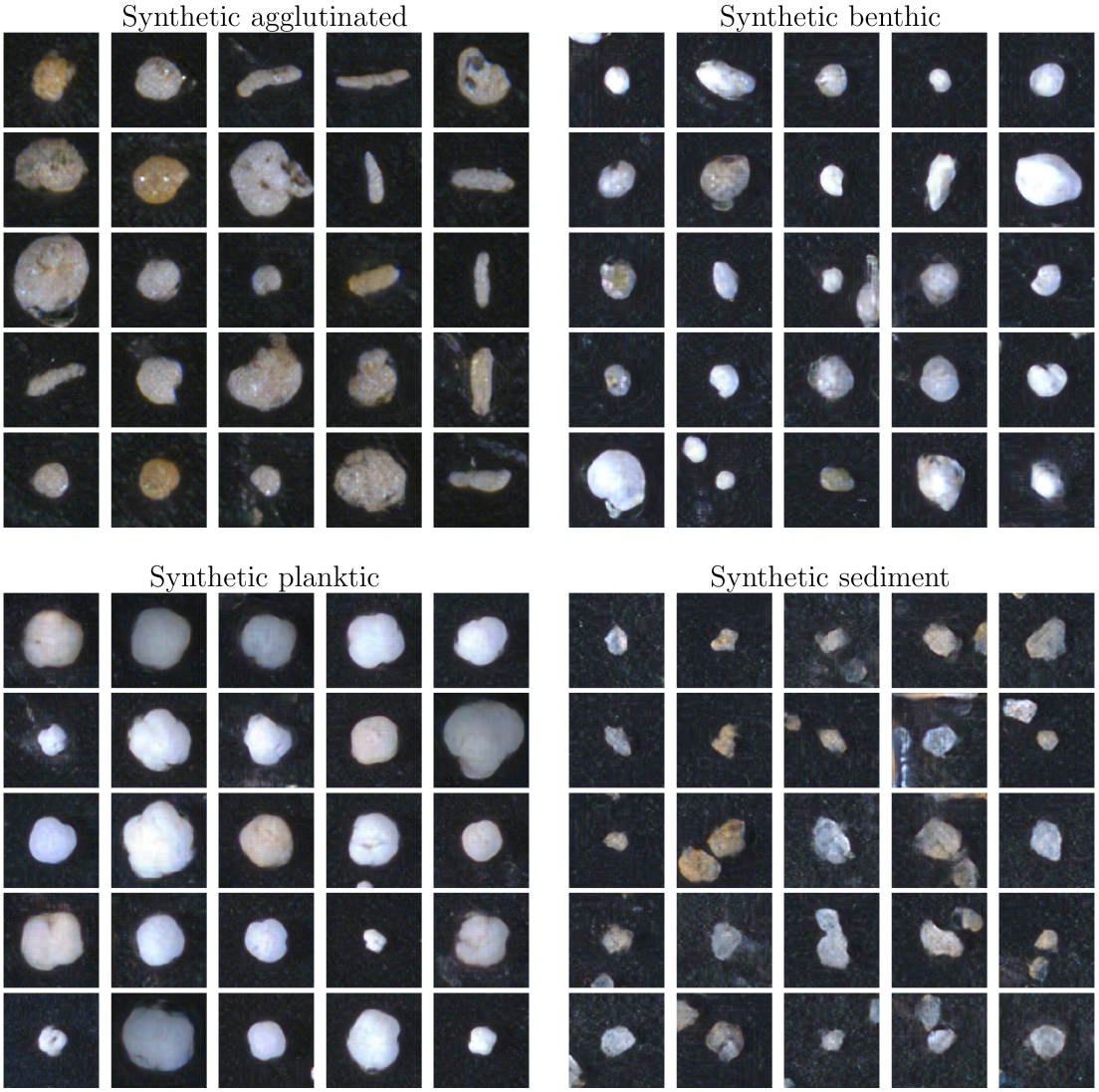
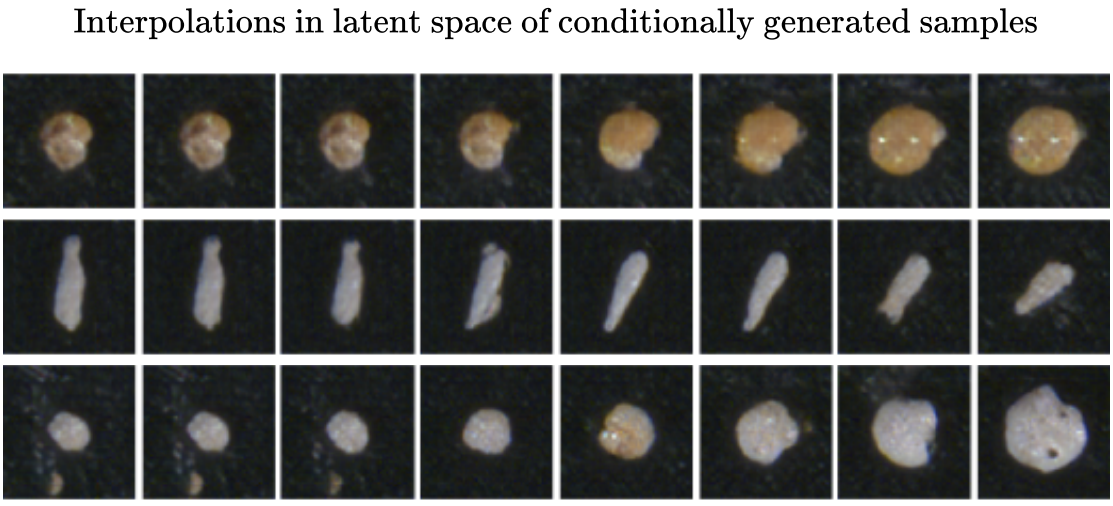
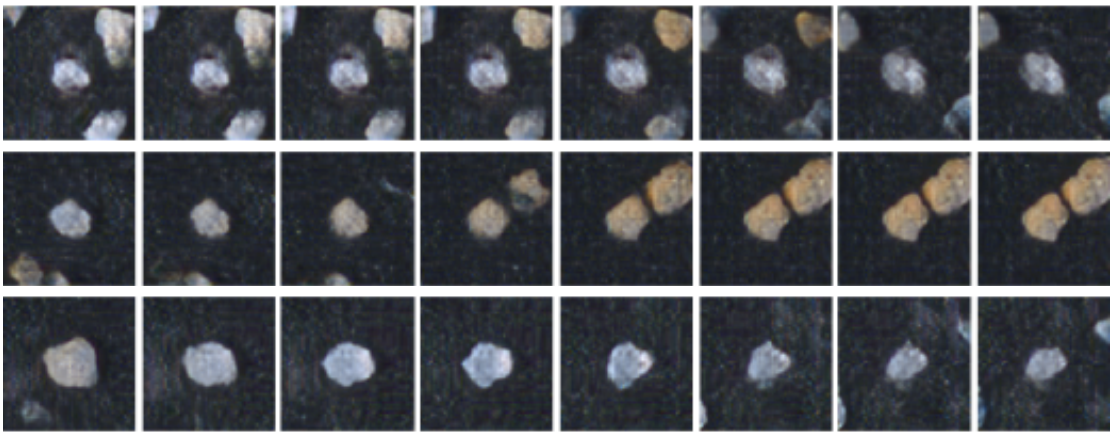


Figure 4.16: A random selection of 25 synthetically generated samples from each of the four classes of the foraminifera dataset: *agglutinated*, *benthic*, *planktic* and *sediment*. The generator has learned much of the diversity in each class

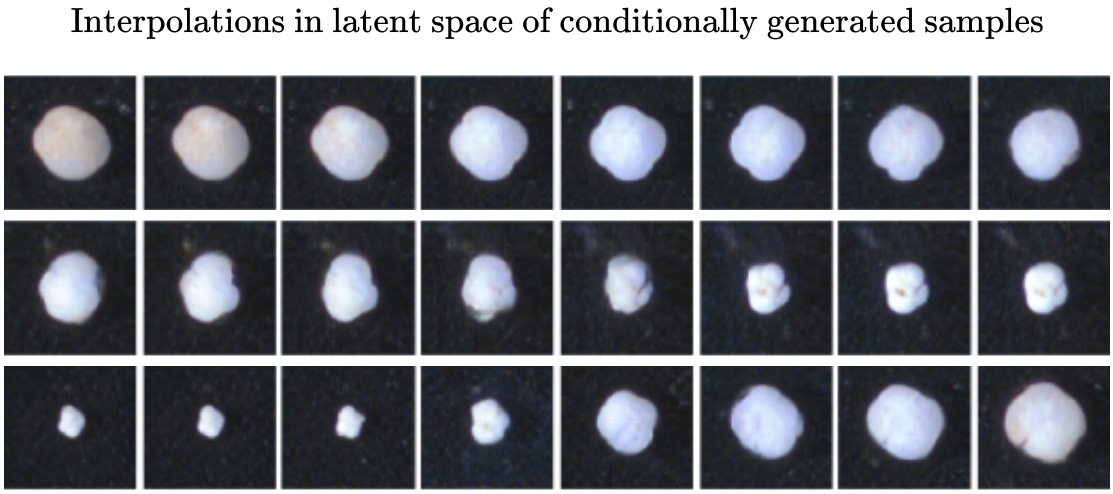


(a) Synthetic agglutinated

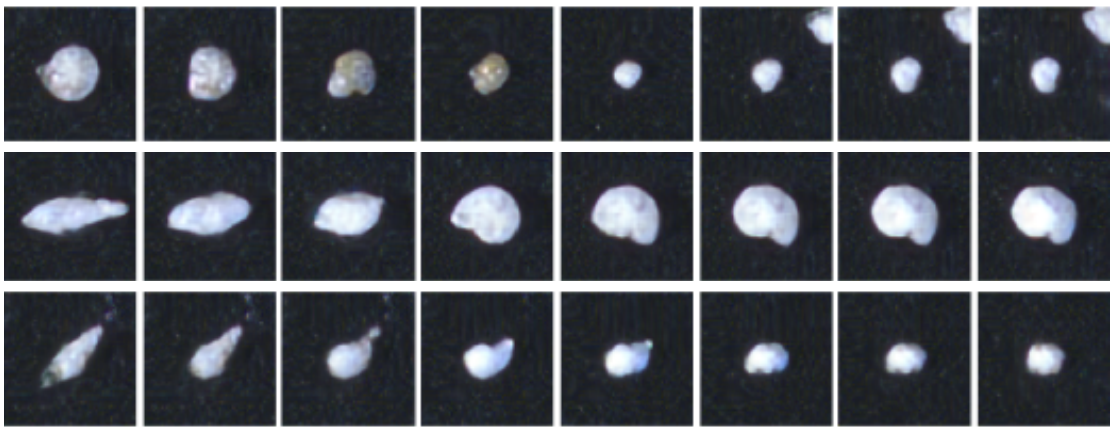


(b) Synthetic sediment grains

Figure 4.17: Images created by interpolating between random points in the latent space of the generators trained on agglutinated foraminifera (a) and sediment grains (b).



(a) Synthetic planktic



(b) Synthetic benthic

Figure 4.18: Images created by interpolating between random points in the latent space of the generators trained on planktic foraminifera (a) and benthic foraminifera (b).

images show a great variety in both size, texture and color. The MSG-GAN (section 4.2) is after all only implemented with one measure against mode collapse – the minibatch standard deviation layer (section 3.6.2.2) from the progressively growing GAN (section 3.6).

The interpolation between images in figure 4.17 and figure 4.18 illustrate that the generators have learned to produce relatively smooth transitions between variants of the specimen within each class. As each training set contains few samples the capacity of the GAN becomes larger relative to the size of the training set. This introduces an additional challenge that is rarely discussed in the GAN literature, namely overfitting causing memorization of the training set. A **memory GAN** would produce sharp transitions between samples when interpolating between images. From the results of interpolation in latent space of the conditional GANs it cannot be guaranteed that there is no degree of memorization in the GANs. The agglutinated GAN would be especially prone to overfitting due to the extra small training set of 156 samples. This issue will be further discussed in the experiment of section 4.6.

The FID values displayed in table 4.5 show that only the GAN trained on sediment grains was able to surpass the FID of the unconditional foraminifera GAN. This partly falsifies the hypothesis of the experiment that stated that the conditional GANs would outperform the unconditional GAN. There are multiple plausible reasons how this could happen, and they will be elaborated on in the following sections.

4.5.3.1 Reliability of FID

For convenience the key aspects of Fréchet Inception distance (section 3.9.2) is reiterated here. FID is the Fréchet distance measured between the distributions P_g and P_r of 2048 features have been extracted from Inception V3's `pool_3` layer, from 50 000 fake samples and the whole training set of images, respectively. These distributions are assumed to be multivariate Gaussian, so the FID can be calculated by

$$\mathrm{FID}(P_r, P_g) = \|\boldsymbol{\mu}_r - \boldsymbol{\mu}_g\|_2^2 + \mathrm{Tr}(\boldsymbol{\Sigma}_r + \boldsymbol{\Sigma}_g - 2(\boldsymbol{\Sigma}_r\boldsymbol{\Sigma}_g)^{\frac{1}{2}}) \tag{3.28}$$

However there are a couple of issues with this evaluation measure for GANs. (1) It has been pointed out that the Gaussian assumption might not hold in practice (Grnarova *et al.*, 2019; Borji, 2019) and probably not when the real distribution consists of few samples. Concerning sample sizes is considered a minimum that each distribution contains at least 2048 samples (Jean, 2018) for this assumption to hold. Noguchi and Harada (2019) pointed out that FID is hence unstable for small datasets and therefore not a good evaluation measure for the quality of GAN images from small training sets. Considering that variety is an important characteristic

that FID measures it becomes clearer that FID should not be recommended for small datasets.

(2) Even though FID has become the standard for evaluation of real-world GAN images, it is not necessarily a good evaluation measure for GANs in all domains. As FID uses the pretrained Inception model the features that it is able to detect in the images are heavily dependent on the data that the model is trained on. Even though the Inception model is known to extract robust features for FID evaluation (Lucic *et al.*, 2018; Heusel *et al.*, 2017), they might be unreliable if the domain FID is applied to is too different from the domain of Inception V3 (Grnarova *et al.*, 2019).

Whether the evaluation of microscope images of foraminifera is within the range of the Inception classifier and hence reliable to FID is uncertain and need more investigation. Possibly the Inception classifier is able to gather useful features, textures and shapes although it is not trained on microscope images. However, the likelihood of violation of the Gaussian assumption in equation 3.28 can be considered to be high, when the small training sets are taken into account. This might be some of the explanation of why all FIDs of the conditional experiments did not surpass the unconditional experiment.

4.5.3.2 When to stop training?

All experiments up to this point has been stopped rather arbitrary. The DCGAN and unconditional foraminifera experiments were stopped after a "*sufficient*" number of epochs. The training proceeded until the experiments proved a point or fulfilled the objective of the experiment. The CIFAR-10 experiment was stopped after the discriminator was shown 12 million images, so the results could be compared with previous work. This conditional foraminifera experiment however could, for the sake of the objective, continue until the fake images were as good as they could possibly to get.

For GANs there has been no common consensus of when training should come to a halt. Regularization techniques like early stopping (section 2.4.1) is not developed for GANs. The MSE technique employed in previous experiments (figure 4.10 and 4.15) do not give any indication on when to stop training when the MSE stabilizes instead of converging. The GAN loss function is complex and behaves in non-intuitive ways (Grnarova *et al.*, 2019), so it gives few clues of when to stop training. This, along with the challenge of giving GANs a fair evaluation of performance during training, makes the stopping criteria almost arbitrary for some GAN applications. Progress has although been made on this matter, as Grnarova

et al. (2019) showed that approximating the **duality gap**[2] from game theory can measure the similarity of a generated data and true data distribution. This convergence measure seems promising, but has unfortunately not been implemented in any experiments as the author was made aware of the novelty at a late stage of the experimental phase.

4.5.3.3 A grid-like artifact on synthetic samples

On some synthetic images of foraminifera a grid-like artifact can be observed in the texture of the images (figure 4.19). This is an artifact that is not present in the true distribution of images and is thus a clue to which images are real and fake. It is not easy to pinpoint exactly what causes this artifact as the generator and discriminator are deep and complex models.

In the preliminary DCGAN experiments a similar but different "checkerboard" artifact can be observed. This effect has already been discussed and is likely due to the overlapping of filters in the transposed convolution layers. A similar effect could be the reason in these experiments as well, but as the GAN architecture is different it is not likely. The generator used in this experiment builds on the progressively growing GAN that use an upscale layer followed by two convolutional layers to increase the resolution of an image. This method is designed to avoid the checkerboard artifact that can be caused by transpose convolutions (Odena *et al.*, 2016).

One observation that might give a clue on why the generator learns a grid-like texture on some samples is displayed in figure 4.19. It seems that the artifact is most prominent in rare or unique images of foraminifera, some of which that take up almost the whole image. When examining the training data and the fake samples with artifacts qualitatively, two relevant observations are made: (1) the real specimen that are similar to the generated ones with artifacts are relatively rare and unique. (2) in the unconditionally generated images (section 4.4) the artifact is more prominent in the agglutinated looking images. As the agglutinated images are the least numerous of the classes (156 against 701, 568, 949), one common characteristic for these observations is that the artifact appears in images that have few corresponding real samples. This might suggest that the incorrect grid-like texture in some images are due to underfitting and lack of numerous and continuous training data. In other words, the generator incorrectly assumes the texture on some images that it has not trained sufficiently on. If the artifact is due to underfitting, it is likely that it would disappear if the GAN is trained for longer. This hypothesis will be investigated further in the following experiment.

[2] A measure of the sub-optimality of a zero-sum game solution with respect to an equilibrium.

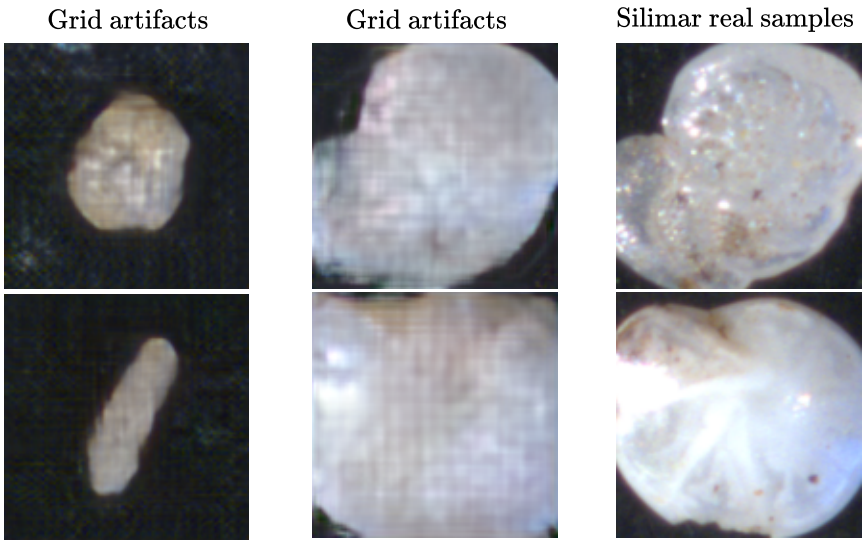


Figure 4.19: (a) Two agglutinated looking unconditionally generated foraminifera with grid artifacts (section 4.4). (b) Two benthic foraminifera that are generated conditionally with signs of the grid artifacts. (c) Two real benthic images are similar to the ones generate in (b).

4.5.4 Closing remarks

A conditional GAN experiment has been conducted on each of the classes of the foraminifera dataset. The hypothesis that the conditionally generated images would yield a lower (better) FID than the unconditionally generated images (FID 47.1) was partly falsified. Only the conditionally generated sediment grains improved FID to 42.0, when benthic, planktic and agglutinated foraminifera measured 52.8, 52.6 and 52.6, respectively. A discussion on why this may have happened suggests three possible explanations that all could be contributing: (1) FID is unreliable for small datasets, (2) there is no sufficient stopping criteria for GANs and (3) a grid artifact in the generated images that may originate from underfitting.

4.6 Underfitting and overfitting in GANs

To follow up the hypothesis introduced in the discussion of section 4.5.3 a short experiment is conducted to investigate the grid artifact observed in some images. The hypothesis is that the grid artifact appears on images that suffer from underfitting. In the previous experiments the underfitting is caused by (i) not enough training data or (ii) not enough training.

4.6.1 Experiment setup

To investigate the hypothesis the MSG-GAN model is trained on the training set of "agglutinated" foraminifera for 800, 2000 and 6400 epochs, showing the discriminator approximately 100 k, 300 k and 1 M real images, respectively. The agglutinated training set is the least numerous and contains only 156 training samples. This should make it plausible to provoke vast overfitting when the model is trained for too long (6400 epochs) and underfitting when the model is trained for too short (800 epochs). If the grid artifact is due to underfitting it should be prominent in the images from the underfitted generator, and not the overfitted one.

Images are generated after each of the given training configurations, and images from interpolations between points in latent space are displayed to visualize the learned mapping, and to uncover memorization of training data. The FID is measured after 2000 epochs and 6400 epochs by evaluating 50 000 fake samples against the whole agglutinated training set.

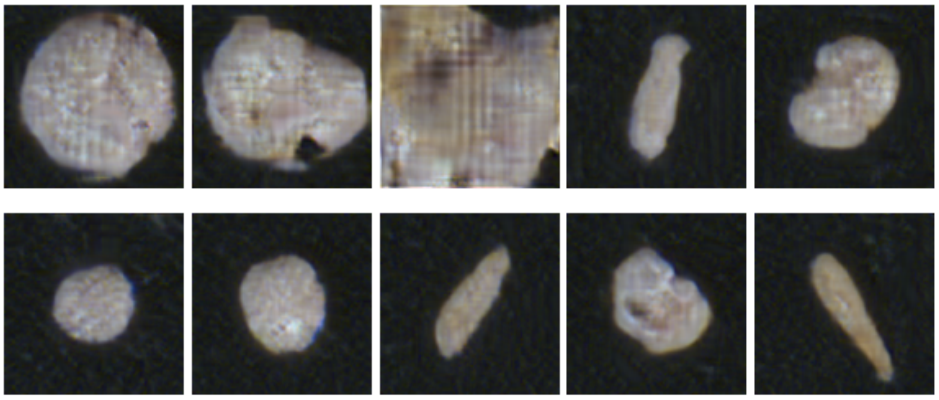
4.6.2 Results

A random selection generated samples after training for 800, 2000 and 6400 epochs are displayed in figure 4.20. Interpolations between random points in latent space after different amount of training is presented in figure 4.21. The FID is measured to 52.6 for the samples generated after 2000 epochs, and 32.0 after 6400 epochs.

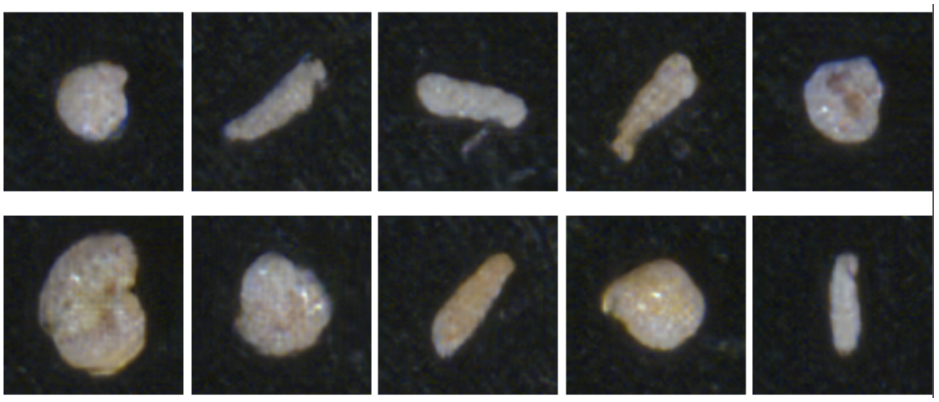
4.6.3 Discussion

Figure 4.20 confirms that the grid artifact produced by the generator in much degree is due to underfitting of the GAN. It can be assumed that the generator not yet has learned the textures of the agglutinated class, as the artifact disappears as the GAN is trained for longer. Adlam *et al.* (2019) suggest that poor performance of the generator is due to the discriminator being underfit. This coincides well with the results as the GAN is only trained for 800 epochs and showed 100 k real images.

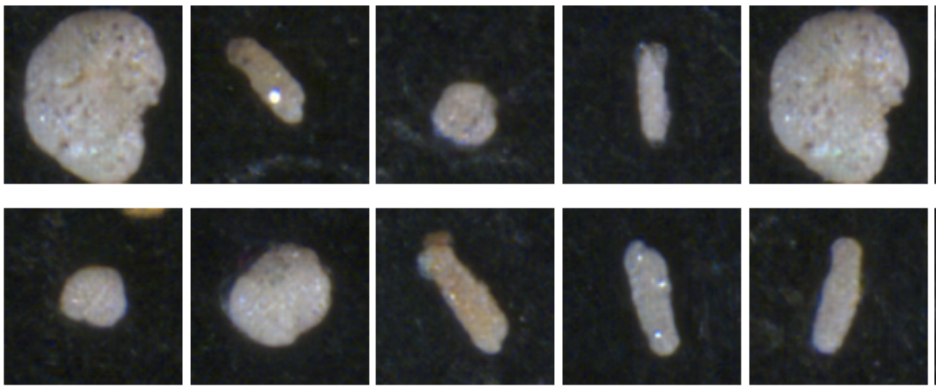
Figure 4.21 illustrates how overfitting can happen in GANs. The GAN has "memorized" the whole training set, so the generator do not produce smooth transitions between images, and the FID drops significantly to 32.0. A possible explanation on why this happens involves the relative capacity of the discriminator compared to the dataset it tries to learn (Adlam *et al.*, 2019). In practice the discriminator has the same capacity when it in this experiment tries to learn the distribution of 156 agglutinated images as it has when it tries to learn the full 2637 unconditional foraminifera images, or the 60 000 images of the CIFAR-10 dataset. This taken



(a) GAN trained for 800 epochs (underfit)

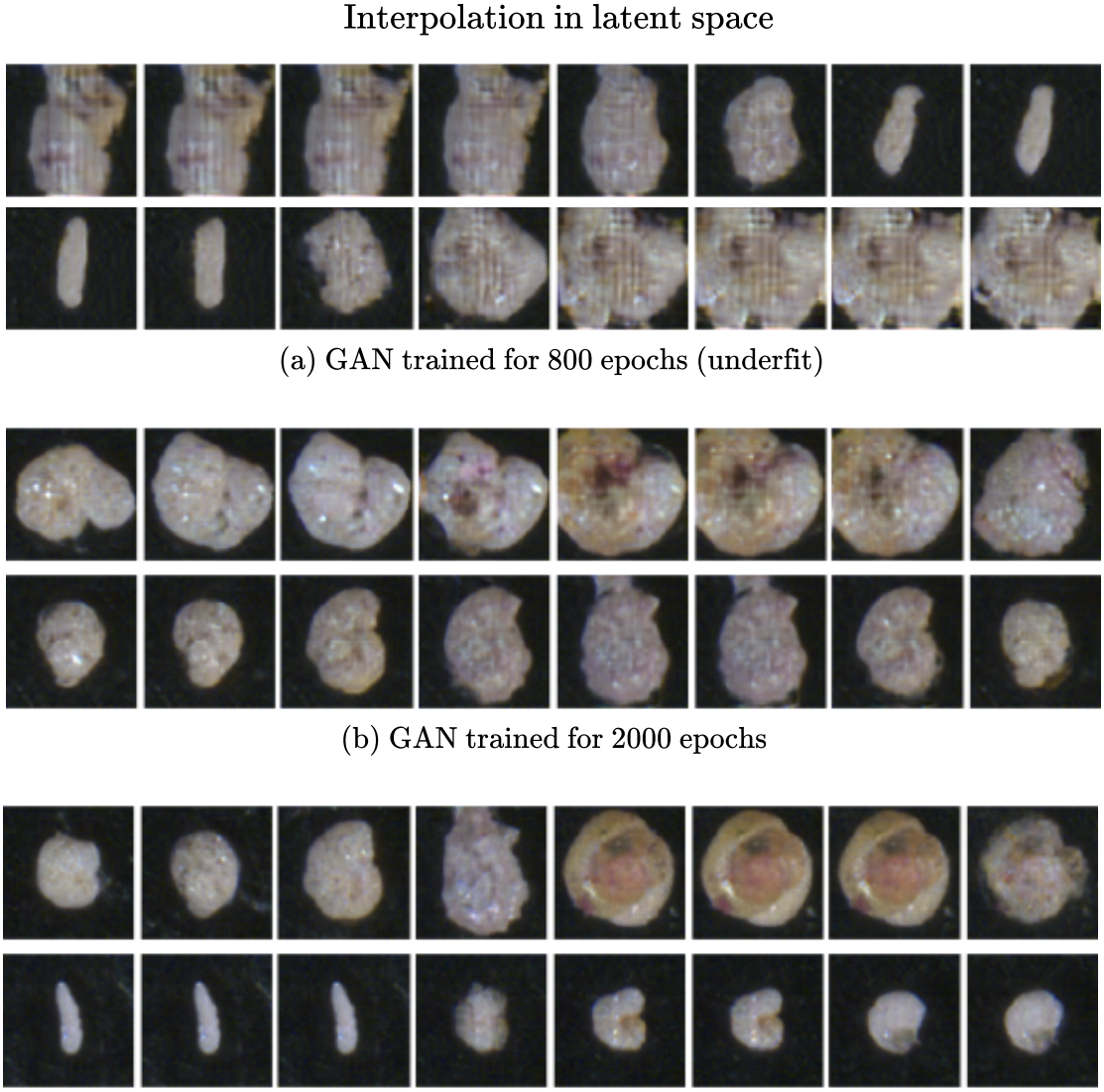


(b) GAN trained for 2000 epochs



(c) GAN trained for 6400 epochs (overfit)

Figure 4.20: All images are randomly generated from the same MSG-GAN trained for a different number of epochs on the 156 agglutinated samples. (a) Underfitting in the generator causes the grid artifact discussed earlier. This artifact disappears when the GAN is trained for longer (b). (c) illustrates severe overfitting in the generator. It produces two almost identical samples (figure (c) top row left and right) by chance (!). At this point the GAN has "memorized" all training samples.



(a) GAN trained for 800 epochs (underfit)

(b) GAN trained for 2000 epochs

(c) GAN trained for 6400 epochs (overfit)

Figure 4.21: Latent space interpolation on the MSG-GAN trained on agglutinated foraminifera for 800, 2000 and 6400 epochs. The grid artifact can be easily spotted in figure (a). The artifact disappears when the GAN is trained for longer (b). Figure (c) illustrated the "memory GAN" that has memorized all training samples and is not able to produce smooth transitions.

into consideration it becomes clear that the discriminator should have no problem to memorize all 156 training images, and thus reproduce training samples instead of generating new plausible agglutinated samples.

When the discriminator has trained for 2000 epochs it seems to produce images that are plausible, and do not suffer from too much memorization. It is not guaranteed that the GAN has not memorized the training set to some degree, but the results seems sufficient for the objective of this thesis.

4.6.4 Closing remarks

The grid-artifact that is observed in some images in the unconditional and conditional experiment is investigated and is found to presumably be caused by underfitting of the discriminator. The results show that the grid artifact disappears as the GAN is trained for 2000 epochs. When the model has trained for 6400 epochs on the smallest foraminifera dataset of 156 images, the GAN has to a large degree memorized all training samples due to the discriminator's relatively high capacity.

4.7 Assessment of conditionally generated foraminifera

The final objective of this thesis is to improve classification of the foraminifera classifier of (Johansen and Sørensen, 2020) (section 2.5) by using synthetic images. To verify that the synthetic images contain valuable information so that can be used to improve the classifier, a qualitatively and quantitatively assessment of the images is performed. The scientists behind the foraminifera classifier (Johansen and Sørensen, 2020) have contributed with an expert evaluation of the synthetic images, as well as providing an equivalent foraminifera classifier optimized for 128×128 resolution images. The goal is to test whether the generated images contain enough characteristic information to be classified correctly by a foraminifera expert and by the original foraminifera classifier.

4.7.1 Experiment setup

Two separate experiments are conducted to achieve the same objective. The first is a human expert assessing benthic and planktic generated images. The second is the foraminifera classifier that attempts to classify samples generated from all four classes.

Table 4.6: The form used by the foraminifera expert to assess benthic and planktic images. Each image 001-200 was classified as either benthic or planktic, and if the image was thought to be real or fake. The expert reported the confidence level of the desicion from 1-5.

Image #	Benthic?	Planktic?	Fake?	Confidence
001				
002				
003				
⋮				

4.7.1.1 Expert assessment of planktic and benthic foraminifera

In this assessment a marine geologist whose field of expertise is foraminifera is presented with a set of 100 real and 100 synthetic images of foraminifera and is challenged to classify them. The set consists of images that are randomly generated from the benthic and planktic class (section 4.5), and images that is randomly drawn from the training sets of the respective classes. From each class there are 50 real and 50 synthetic samples. The participant is asked to classify each image as benthic or planktic, as well as if the image is real or synthetic and the overall confidence level (1-5) of the decision. The results are recorded in a spreadsheet similar to the one displayed in table 4.6. The classification accuracy of the expert is reported on both the synthetic and the real images.

It should be noted that this setup of evaluation is very different from how foraminifera classification usually take place. Usually the classification is performed using a microscope that is able to zoom in to display much detail of each specimen. In addition a tiny needle is used so the specimen can be rotated and inspected from different sides.

4.7.1.2 Automatic assessment of conditionally generated foraminifera

To assess the conditionally generated foraminifera the classification model of Johansen and Sørensen (2020) is used to classify synthetic samples from all four classes. The corresponding author of (Johansen and Sørensen, 2020) have provided a trained and fine-tuned model of the original foraminifera classifier that is optimized for 128×128-pixel images. This model was obtained by training on downscaled images of the original dataset and choosing the best performing weights after 10 random weight initializations. After 10 runs the model achieved a mean accuracy

of $97.3 \pm 0.4\%$[3], with a best run at 97.7 %, on the test set.

To this model a total of 10 000 synthetic images generated from the conditionally trained GANs of section 4.5 was randomly chosen. The relative proportions of the original dataset are preserved, so 6.6 %, 29.5 %, 23.9 % and 40.0 % of the samples were from the agglutinated, benthic, planktic and sediment classes respectively. These images are classified by the 128×128-model and the accuracy is reported.

4.7.2 Results

The expert assessment of 200 foraminifera images yielded a mean accuracy of 90.0 % upon classification. On average the expert guessed correctly if the samples were real or generated in 92.5 % of the cases. When considering only the real images, the expert classified them correctly 99.0 % of the time. The generated images were classified correctly in 81.0 % of the trials. The confidence level for the fake samples were 3.7, and for the real samples 4.19. Using the trained classification model to assess 10 000 synthetic images a categorical accuracy of 93.4 % was achieved across all four classes.

4.7.3 Discussion

The results of assessing the synthetic images gives promising prospects for the final experiment of this thesis. The images contains enough information to be classified correctly in most cases, in both assessments performed in this experiment. 93.4 % accuracy of the CNN suggests that the generated images contain much of the relevant characteristics it uses for classification. As the foraminifera expert managed to correctly classify the synthetic samples in 81 % of the cases, the images clearly contain important information for human assessment as well. Albeit this accuracy is lower than the impressive 99 % on the real images it still may be enough for improving the deep learning based classification model.

When interviewed after the assessment, the expert reported that the synthetic images was usually detected due to a *"blurry outline"* and *"visible 'digital' striping"* (e.i. the grid-artifact), that became clear when zooming in on the images. These problems of "ghosts" and grid-artifacts have been addressed previously (section 4.6 and 4.4) and are familiar with GAN images, especially when the GAN is trained with limited data. The expert stated that most of the incorrectly classified generated samples were due to *"lack of defined chambers"* in planktic foraminifera. The expert noted that the *"lack of defined chambers made it look like a single chamber benthic"*,

[3]This is a performance drop from $98.8 \pm 0.2\%$ when the model is trained on full scale 224×224 resolution images.

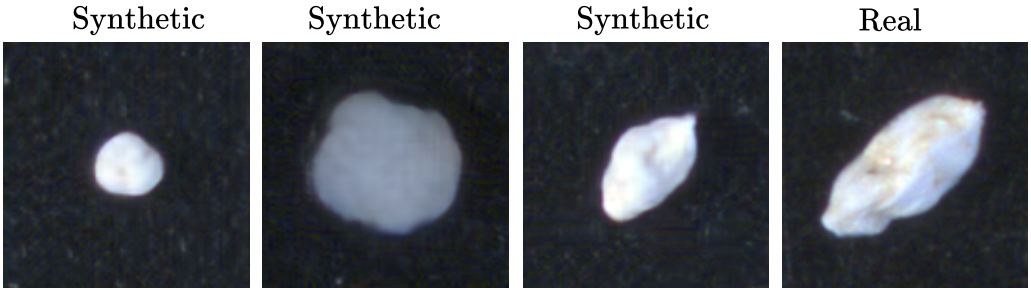


Figure 4.22: (a) and (b) show synthetically generated planktic foraminifera that was assumed to be single chamber benthic by the expert due to their lack of defined chambers. In (b) a more undefined and blurry outline of some synthetic foraminifera is visible. (c) show a highly realistic benthic foraminifer that has characteristics reminiscent of a specimen of acidic conditions. (d) show a similar real sample to (c) but has not been subject to acidic conditions.

when reviewing several of the misclassified generated samples from the planktic class (figure 4.22 (a) and (b)). This may suggest that the synthetic images might introduce some variants that are rare or that do not appear in nature. One example of a generated foraminifer that fooled the expert was the one displayed in figure 4.22 (c). The expert elaborated that the sample looks like a highly realistic benthic foraminifera, and that the smoothed surface of the foraminifer was reminiscent of a specimen that had been subject to acidic conditions. This example illustrates how generated data might as well introduce valuable variation into a scarce dataset.

4.7.4 Closing remarks

In this experiment the conditionally generated images of foraminifera from section 4.5 have been assessed by (1) an expert and (2) a CNN classification model. The results suggested that the synthetic foraminifera contains valuable information that can be used for classification. Although the images in most cases are credible they may sometimes introduce artifacts or characteristics that make classification harder.

4.8 Improving classification of foraminifera using synthetic data

The final objective of this thesis is to improve the classification model of (Johansen and Sørensen, 2020) described in section 2.5 by using synthetic foraminifera images

from a generative adversarial network. To this point several experiments have been conducted as intermediate steps towards this final objective. In this final experiment the main hypothesis related to the aforementioned objective is tested. The hypothesis is reiterated for convenience:

> Augmenting the training set of the foraminifera classifier (Johansen and Sørensen, 2020) with synthetic images from a generative adversarial network will improve the classification accuracy of the model.

As the computational resources available are somewhat limited, it is not possible to produce full 224×224 resolution synthetic images. The model that will be considered and attempted to improve will thus be the provided model of resolution 128×128 from section 4.7. This model have a mean accuracy on the foraminifera test split of $97.3 \pm 0.4\%$ after 10 runs, with a best achieved accuracy of 97.7 %.

The challenge of this objective should be emphasized as improving a models accuracy that already is close to 100 % is considered a very hard task. The model of (Johansen and Sørensen, 2020) is fine-tuned and optimized by performing computationally expensive hyperparameter searches. This experiment will not perform the same optimization process, only test the effect of augmenting additional synthetic images to the existing training set.

4.8.1 Experimental setup

To test the hypothesis the MSG-GAN described section 4.2 is used to generate images from each class in the foraminifera dataset. The same experimental setup is used as in the conditional experiment of section 4.5. To summarize the key points, an MSG-GAN is trained on the train and validation split of each class of the foraminifera dataset. The agglutinated GAN discriminator is shown approximately 300 k real images while the discriminators of the remaining classes are shown approximately 1 M real images.

The trained generators are used to produce synthetic images of each class which in turn are augmented to the training set of each class. In total 10 000 synthetic samples are produced: 660 agglutinated, 2950 benthic, 2390 planktic and 4000 sediment, so the original proportions of the classes in the dataset is preserved after augmentation. The 128×128 CNN classifier of section 2.5 is trained on the augmented dataset. In practice the training set is increased by a factor of ≈ 5 after augmentation. The classification model is trained in the exact same manner as the original model, only with the synthetically augmented training set.

The mean accuracy and standard deviation after training the model 10 times with random weight initializations is reported, along with the highest achieved accuracy.

4.8.2 Results

Augmenting the original dataset with 10 000 synthetically generated images and training the classifier from scratch 10 times from random weight initializations yielded the following result: The mean classification accuracy was improved with 0.1 percentage points from $97.3 \pm 0.4\%$ to $97.4 \pm 0.7\%$. The best achieved accuracy obtained during the 10 runs was improved by 0.8 percentage points from 97.7% to 98.5%.

4.8.3 Discussion

Considering the challenge of improving the already high accuracy of the model, the results of this experiment provides promising results of the approach. The experimental results confirms the main hypothesis of the thesis, although the improvement is marginal.

In section 4.6 and section 4.7 it was found that artifacts that was not present in the original distribution was introduced by the conditional GANs. A plausible outcome of this experiment was hence that these additional artifacts would introduce noise into the distribution, so the classifier would perform worse after the augmentation. Considering that the classifier was previously optimized for the original dataset it was no guarantee that the synthetic augmentation would lead to improved accuracy.

Although the mean accuracy improved after augmentation, the standard deviation of the classifications increased from 0.4 % to 0.7 %. This increase indicates that the final classification result is somewhat more reliant on the weight initialization of the model, and the stochastic processes that occurs during training. As the test set is relatively small (10 % of the original dataset, and 263 images), classification of one image more or less will have a considerable effect on the end test result. To further investigate the effect of synthetic data augmentation more experiments should be performed on bigger test sets, and different selections of generated images.

The experiment conducted here have used both the training and validation split of the dataset to train the conditional GANs. If any of the GANs to some degree have memorized any of the training samples, it might give an effect on the early stopping implemented in the CNN classifier of (Johansen and Sørensen, 2020). Conducting augmentation experiments with generated images from GANs that have not seen the validation split would yield interesting results, and possibly a model that can generalize better to unseen test data.

Previous results of related work of GAN based image augmentation showed a similar effect of synthetically augmenting scarce datasets. Frid-Adar *et al.* (2018) used an experimental setup that was similar to the one used in this experiment, but

they applied classical data augmentation techniques (cropping, flipping, rotations etc.) *before* training the conditional GANs. The experiment performed in this thesis applied the classical data augmentation *after* the GANs were trained, as this was a part of the training scheme of (Johansen and Sørensen, 2020). It would be interesting to measure the effect of applying classical augmentation techniques before training the GANs as the datasets were somewhat limited to start with. This approach would may yield a better performing conditional GAN that has learned a more continuous distribution and is less prone to overfitting as the training set increases due to the classically augmented images.

A final variant of this experiment that should be investigated further is the effect of the number of synthetic images that is added to the classifier. This experiment was conducted with 10 000 synthetic images that preserved the original class distribution of the dataset. The exact number of 10 000 images was used for convenience related to transfer speeds and computational resources, but it is not guaranteed that this is the optimal number of images. It may be that additional or fewer synthetic images augmented would yield a better result of the experiment.

Due to the limited scope and extent of this thesis, the proposed improvements and additional experiments must unfortunately be left for future work.

4.8.4 Closing remarks

This experiment have tested the hypothesis to fulfill the final objective of this thesis – to improve the foraminifera classification model of (Johansen and Sørensen, 2020) by synthetic data augmentation. Due to the limited computational resources available the hypothesis could only be tested on a downscaled version of the original classifier. Under these circumstances the hypothesis was confirmed by a marginal improvement in accuracy from $97.3 \pm 0.4\%$ to $97.4 \pm 0.7\%$ with a best run improved from 97.7% to 98.5%. These results provide promising results that encourage further investigation in the use of GANs to synthetically augment scarce datasets.

Chapter 5

Final discussion and concluding remarks

This chapter provides a final discussion and summary of the experiments that have been conducted. The experiments and results are discussed from an holistic approach on the basis of the objectives of the thesis. After an overall discussion, propositions for future work related to the experiments of this thesis are suggested. Finally some concluding remarks are made on the thesis as a whole.

5.1 Final discussion

The first objective of this thesis was to study the deep learning models of generative adversarial networks. The motivation for this objective was the recent advances in the field of study that allow GANs to learn high-dimensional distributions of e.g. images. This potential has brought the application of GANs to multiple domains in research and to applications in society. The intent of the study presented in chapter 3 was to provide an in-depth exploration, and theoretical background, of GANs that could be useful for research as well as the inclined teacher in the Norwegian education system. The experiments of section 4.1 and 4.3 continued the exploration from an empirical standpoint, showing off some of the challenges and learned features of GANs.

The potential of GANs to learn high-dimensional distributions led to the second objective of this thesis: To learn the distribution of foraminifera images, and use them to generate synthetic images of foraminifera. Provided the theoretical background from chapter 2 and 3 the implementation and methodology of the multi-scale gradient GAN was described, so it could be used to learn the distribution

of foraminifera. The experiments that followed showed off the interesting attributes that the GAN had learned when modeling the distribution of foraminifera images. Interpolations in the latent space of the generator displayed the smooth transitions that was learned within each class and between classes. Both the unconditional and conditional learning experiments, as well as the investigations and assessment that followed in section 4.6 and 4.7 illustrated that GANs have great potential to learn important attributes of a distribution unsupervised. These experiments, their results and their following discussions was measures that was taken towards achieving the second objective of this thesis.

Even though the experiments were justified by the second objective of this thesis, they were also necessary steps towards the final objective of this thesis. Section 4.8 provided an experiment to test if the conditionally generated images from the foraminifera dataset classes could improve the already high classification accuracy of the CNN model of (Johansen and Sørensen, 2020). The experiment was sustained to the extent that the computational resources available allowed, and within the scope of this thesis. The results were promising for further investigations and application of the technique to further develop and improve CNN-based foraminifera classifiers.

5.2 Future work

This section provides suggestions of future work that could be an extension of this thesis, or that relates generative adversarial networks to the long-term goal towards developing a reliable foraminifera classifier.

5.2.1 Direct extensions of this thesis

A direct continuation of the work of this thesis would be to implement the recently developed convergence criteria and goodness measures for GANs by (Grnarova *et al.*, 2019). This modification, along with a hyperparameter search and more computational resources, could provide better generated images and further uncovering of the potential of GAN applications on foraminifera images.

Other GAN architectures should also be investigated. The GAN of (Karras *et al.*, 2019a) uses principles from neural style transfer to learn high-level attributes and different "styles". This model may have potential to learn different "styles" that are related to the species of foraminifera in an unsupervised manner. It could as well be interesting to see if the effect of transfer learning could be useful in GANs and the domain of foraminifera. Noguchi and Harada (2019) have achieved promising results using a novel method of transferring knowledge from a pretrained generator to a new domain, yielding synthetic images from small training sets of

< 100 samples.

5.2.2 Towards the goal of an automatic foraminifera classifier

The foraminifera classifier under consideration in this thesis only classifies samples into four high-level classes. To achieve the long-term goal of being able to classify the specific species of a specimen, additional models and image datasets must be developed. When new datasets are developed and classification models are extended to improve discrimination between different species, performance may not be as good as in the model of (Johansen and Sørensen, 2020). Previous results (Frid-Adar *et al.*, 2018) may suggest that that the effect of augmenting the dataset with synthetic images is larger when the classifier is not performing as well to begin with. If this is the case in the future GANs can be applied in a similar manner as in this thesis, to synthetically augment the datasets and potentially improve classification with a greater effect.

If additional unlabeled images of sediment and foraminifera are provided, a slight modification to the GAN scheme could make use of GANs for semi-supervised learning (Salimans *et al.*, 2016; Odena, 2016; Dai *et al.*, 2017). The modification for semi-supervised learning use the GAN to learn important high-level attributes of the images in an unsupervised manner, and the discriminator is modified to both classify and discriminate based on very few labeled samples.

5.3 Concluding remarks

This thesis emerged from the recent advances of generative adversarial networks (GANs) and their potential to model high-dimensional distributions such as real-world images.

The advances of these models have been studied and explored from a theoretical and empirical standpoint. A multi-scale gradient GAN was implemented in `Tensorflow 2.1` and tested on the CIFAR-10 dataset to find the best model configuration. It was in turn trained to learn the distributions of four high-level classes of a recent foraminifera dataset, both conditionally and unconditionally. The conditional images were assessed by an expert and a deep learning classification model and was found to contain mostly valuable characteristics, although some artificial artifacts was introduced. The unconditional images measured a Fréchet Inception distance of 47.1.

From the conditionally learned distributions a total of 10 000 images was sampled

from the four distributions. These images were used to augment the original foraminifera training set in an attempt to improve the classification accuracy of (Johansen and Sørensen, 2020). Due to limitations of computational resources, the experiments were carried out with images of resolution 128×128. The synthetic image augmentation lead to an improvement in mean accuracy of $97.3 \pm 0.4\%$ to $97.4 \pm 0.7\%$ and an improvement in best achieved accuracy from 97.7% to 98.5%.

Bibliography

Adlam, B., Weill, C., and Kapoor, A. (2019). Investigating under and overfitting in wasserstein generative adversarial networks. *arXiv preprint arXiv:1910.14137*. 102

Alpaydin, E. (2014). *Introduction to Machine Learning. ISBN: 978-0–262-028189*. MIT Press. 14

Arjovsky, M., Chintala, S., and Bottou, L. (2017). Wasserstein gan. *arXiv preprint arXiv:1701.07875*. 50, 53, 54, 55, 62, 69

Bailey, J. P. and Piliouras, G. (2018). Multiplicative weights update in zero-sum games. In *Proceedings of the 2018 ACM Conference on Economics and Computation*, pages 321–338. 43

Bellman, R. E. (1961). *Adaptive control processes: a guided tour*. Princeton university press. 32

Borji, A. (2019). Pros and cons of gan evaluation measures. *Computer Vision and Image Understanding*, **179**, 41–65. 98

Bouvrie, J. (2006). Notes on convolutional neural networks. 19, 24

Bowles, C., Chen, L., Guerrero, R., Bentley, P., Gunn, R., Hammers, A., Dickie, D. A., Hernández, M. V., Wardlaw, J., and Rueckert, D. (2018). Gan augmentation: Augmenting training data using generative adversarial networks. *arXiv preprint arXiv:1810.10863*. 4

Brock, A., Donahue, J., and Simonyan, K. (2018). Large scale gan training for high fidelity natural image synthesis. *arXiv preprint arXiv:1809.11096*. 33, 84

Burt, P. and Adelson, E. (1983). The laplacian pyramid as a compact image code. *IEEE Transactions on communications*, **31**(4), 532–540. 46

Commons, W. (2013). File:ammonia beccarii.jpg — wikimedia commons, the free media repository. [Online; accessed 9-March-2020]. 13

Dai, Z., Yang, Z., Yang, F., Cohen, W. W., and Salakhutdinov, R. R. (2017). Good semi-supervised learning that requires a bad gan. In *Advances in neural information processing systems*, pages 6510–6520. 115

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. (2009). Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee. 28, 62

Denton, E. L., Chintala, S., Fergus, R., *et al.* (2015). Deep generative image models using a laplacian pyramid of adversarial networks. In *Advances in neural information processing systems*, pages 1486–1494. 37, 46, 50, 56, 62

Desjardins, G. and Bengio, Y. (2008). Empirical evaluation of convolutional rbms for vision. *DIRO, Université de Montréal*, pages 1–13. 33

Duchi, J., Hazan, E., and Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, **12**(Jul), 2121–2159. 10

Durugkar, I., Gemp, I., and Mahadevan, S. (2016). Generative multi-adversarial networks. *arXiv preprint arXiv:1611.01673*. 56

Fedus, W., Rosca, M., Lakshminarayanan, B., Dai, A. M., Mohamed, S., and Goodfellow, I. (2017). Many paths to equilibrium: Gans do not need to decrease a divergence at every step. *arXiv preprint arXiv:1710.08446*. 37, 55

Fréchet, M. (1957). Sur la distance de deux lois de probabilité. *COMPTES RENDUS HEBDOMADAIRES DES SEANCES DE L ACADEMIE DES SCIENCES*, **244**(6), 689–692. 63

Frid-Adar, M., Diamant, I., Klang, E., Amitai, M., Goldberger, J., and Greenspan, H. (2018). Gan-based synthetic medical image augmentation for increased cnn performance in liver lesion classification. *Neurocomputing*, **321**, 321–331. 4, 110, 115

Friedman, J. H. (1989). Regularized discriminant analysis. *Journal of the American statistical association*, **84**(405), 165–175. 32

Gatys, L. A., Ecker, A. S., and Bethge, M. (2016). Image style transfer using convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2414–2423. 61

Gong, S., Boddeti, V. N., and Jain, A. K. (2019). On the intrinsic dimensionality of image representations. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3987–3996. 33

Goodfellow, I. (2016a). Deep learning book notation. Avaliable at `https://github.com/goodfeli/dlbook_notation`. Accessed 1 December 2019. xv

Goodfellow, I. (2016b). Nips 2016 tutorial: Generative adversarial networks. *arXiv preprint arXiv:1701.00160*. 33, 34, 36, 42, 43, 45, 50, 53

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2014). Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680. iii, 3, 24, 33, 34, 36, 37, 40, 42, 46, 50, 53, 62

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. 8, 9, 12, 15, 18, 19, 20, 21, 22, 24, 26, 34, 36

Grnarova, P., Levy, K. Y., Lucchi, A., Perraudin, N., Goodfellow, I., Hofmann, T., and Krause, A. (2019). A domain agnostic measure for monitoring and evaluating gans. In *Advances in Neural Information Processing Systems*, pages 12069–12079. 93, 98, 99, 114

Gulrajani, I., Ahmed, F., Arjovsky, M., Dumoulin, V., and Courville, A. C. (2017). Improved training of wasserstein gans. In *Advances in neural information processing systems*, pages 5767–5777. 55, 84

Hayward, B., Cedhagen, T., Kaminski, M., and Gross, O. (2020). World foraminifera database. Avaliable at `http://www.marinespecies.org/foraminifera`. Accessed 22 March 2020]. 1, 2

He, K., Zhang, X., Ren, S., and Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034. 57

Herrmann, V. (2017). Wasserstein gan and the kantorovich-rubinstein duality. Avliable online at `https://vincentherrmann.github.io/blog/wasserstein/`. Accessed 3 April 2020. 51

Heusel, M., Ramsauer, H., Unterthiner, T., Nessler, B., and Hochreiter, S. (2017). Gans trained by a two time-scale update rule converge to a local nash equilibrium. In *Advances in neural information processing systems*, pages 6626–6637. 63, 99

Hillebrand, S. (2016). Accent to the top. Available at: `https://pixnio.com/people/accent-to-the-top`. [Accessed 24 March 2020]. 21

Hubel, D. H. and Wiesel, T. N. (1959). Receptive fields of single neurones in the cat's striate cortex. *The Journal of physiology*, **148**(3), 574–591. 19

Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*. 26, 27

Islam, J. and Zhang, Y. (2020). Gan-based synthetic brain pet image generation. *Brain Informatics*, **7**, 1–12. 4

Jean, N. (2018). Fréchet inception distance. `https://nealjean.com/ml/frechet-inception-distance/`. 98

Johansen, T. H. and Sørensen, S. A. (2020). Towards detection and classification of microscopic foraminifera using transfer learning. *arXiv preprint arXiv:2001.04782*. iii, 1, 3, 5, 27, 28, 65, 85, 86, 105, 106, 108, 109, 110, 111, 114, 115, 116

Kantorovich, L. V. and Rubinstein, G. S. (1958). On a space of completely additive functions. *Vestnik Leningrad. Univ*, **13**(7), 52–59. 53

Karnewar, A. and Iyengar, R. S. (2019). Msg-gan: Multi-scale gradients gan for more stable and synchronized multi-scale image synthesis. *arXiv preprint arXiv:1903.06048*. 4, 59, 60, 61, 70, 94

Karnewar, A., Wang, O., and Iyengar, R. S. (2019). Msg-gan: multi-scale gradient gan for stable image synthesis. *CoRR, abs/1903.06048*, **6**. 70, 84

Karras, T., Aila, T., Laine, S., and Lehtinen, J. (2017). Progressive growing of gans for improved quality, stability, and variation. *arXiv preprint arXiv:1710.10196*. 4, 56, 57, 58, 69, 70, 84, 85, 94

Karras, T., Laine, S., Aittala, M., Hellsten, J., Lehtinen, J., and Aila, T. (2019a). Analyzing and improving the image quality of stylegan. *arXiv preprint arXiv:1912.04958*. 33, 61, 114

Karras, T., Laine, S., and Aila, T. (2019b). A style-based generator architecture for generative adversarial networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4401–4410. 61, 94

Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*. 10, 11

Kingma, D. P. and Welling, M. (2013). Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*. 33

Krizhevsky, A., Hinton, G., *et al.* (2009). Learning multiple layers of features from tiny images. 77

LeCun, Y. (2016). What are some recent and potentially upcoming breakthroughs in deep learning? Avaliable at `https://www.quora.com/What-are-some-recent-and-potentially-upcoming-breakthroughs-in-deep-learning`. Accessed 9 April 2020. 3

LeCun, Y. *et al.* (1989). Generalization and network design strategies. *Connectionism in perspective*, **19**, 143–155. 18

LeCun, Y., Cortes, C., and Burges, C. (2010). Mnist handwritten digit database. `http://yann.lecun.com/exdb/mnist/`. 66, 67

Li, K. (2019). Overcomping mode collapse and the curse of dimensionality. Available at `https://drive.google.com/file/d/1PV4YN3OQprww4BCDwB9XWMUIz_mbdDab/view`. Accessed 26 March 2020. 45, 47

Li, K. and Malik, J. (2018). Implicit maximum likelihood estimation. *arXiv preprint arXiv:1809.09087*. 45, 46

Liu, Z., Luo, P., Wang, X., and Tang, X. (2015). Deep learning face attributes in the wild. In *Proceedings of International Conference on Computer Vision (ICCV)*. 66, 68

Lotter, W., Kreiman, G., and Cox, D. (2015). Unsupervised learning of visual structure using predictive generative networks. *arXiv preprint arXiv:1511.06380*. 33

Lucic, M., Kurach, K., Michalski, M., Gelly, S., and Bousquet, O. (2018). Are gans created equal? a large-scale study. In *Advances in neural information processing systems*, pages 700–709. 63, 84, 85, 88, 99

Marshall, M. (2010). "zoologger: 'living beach ball' is giant single cell". *New Scientist*. 2

McCulloch, W. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, **5**(4), 115–133. 14

Mescheder, L., Geiger, A., and Nowozin, S. (2018). Which training methods for gans do actually converge? *arXiv preprint arXiv:1801.04406*. 43

Mitchell, T. M. *et al.* (1997). Machine learning. 7

Moroney, L. (2019). Horses or humans dataset. 66

Muller, M. E. (1959). A note on a method for generating points uniformly on n-dimensional spheres. *Communications of the ACM*, **2**(4), 19–20. 71

Nash, J. F. *et al.* (1950). Equilibrium points in n-person games. *Proceedings of the national academy of sciences*, **36**(1), 48–49. 36

Noguchi, A. and Harada, T. (2019). Image generation from small datasets via batch statistics adaptation. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2750–2758. 98, 114

Norwegian Ministry of Education and Research (2019). Læreplan i fordypning i matematikk. Available at `https://data.udir.no/kl06/v201906/laereplaner-lk20/MAT07-02.pdf`. Accessed 26 February 2020. 5

Odena, A. (2016). Semi-supervised learning with generative adversarial networks. *arXiv preprint arXiv:1606.01583*. 115

Odena, A., Dumoulin, V., and Olah, C. (2016). Deconvolution and checkerboard artifacts. *Distill*. 67, 100

O'neill, B. J. (1996). Using microfossils in petroleum exploration. *The Paleontological Society Papers*, **2**, 237–246. 1

Parzen, E. (1962). On estimation of a probability density function and mode. *The annals of mathematical statistics*, **33**(3), 1065–1076. 31

Qian, N. (1999). On the momentum term in gradient descent learning algorithms. *Neural networks*, **12**(1), 145–151. 10

Radford, A., Metz, L., and Chintala, S. (2015). Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*. 37, 40, 48, 49, 50, 84, 92

Riebeek, H. (2005). Paleoclimatology: the oxygen balance. `https://earthobservatory.nasa.gov/features/Paleoclimatology_OxygenBalance`. 2

Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, **65**(6), 386–408. 14

Rosenblatt, F. (1962). Perceptrons: Principles of neurodynamics. 14

Rosenblatt, M. (1956). Remarks on some nonparametric estimates of a density function. annals of mathematical statistics. 31

Rubner, Y., Tomasi, C., and Guibas, L. J. (2000). The earth mover's distance as a metric for image retrieval. *International journal of computer vision*, **40**(2), 99–121. 50

Sabbatini, A., Morigi, C., Nardelli, M., and Negri, A. (2014). *Foraminifera*, pages 237–256. 2

Salimans, T., Goodfellow, I., Zaremba, W., Cheung, V., Radford, A., and Chen, X. (2016). Improved techniques for training gans. In *Advances in neural information processing systems*, pages 2234–2242. 45, 58, 62, 63, 69, 84, 115

Sauer, T. (2012). *Numerical analysis*. Always learning. Pearson, Boston, Mass, 2nd ed. edition. 54

Simonyan, K. and Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*. 28, 29

Song, Y. and Ermon, S. (2019). Generative modeling by estimating gradients of the data distribution. In *Advances in Neural Information Processing Systems*, pages 11895–11907. 84

Springenberg, J. T., Dosovitskiy, A., Brox, T., and Riedmiller, M. (2014). Striving for simplicity: The all convolutional net. *arXiv preprint arXiv:1412.6806*. 49

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, **15**(1), 1929–1958. 26

Theodoridis, S. and Koutroumbas, K. (2008). *Pattern Recognition, 4th Edition*. Academic Press, 1 edition. 10, 19, 24

Tieleman, T. and Hinton, G. (2012). Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, **4**(2), 26–31. 10

Vaserstein, L. N. (1969). Markov processes over denumerable products of spaces, describing large systems of automata. *Problemy Peredachi Informatsii*, **5**(3), 64–72. 50, 63

Wang, L., Chen, W., Yang, W., Bi, F., and Yu, F. R. (2020). A state-of-the-art review on image synthesis with generative adversarial networks. *IEEE Access*, **8**, 63514–63537. 61, 62

Wang, T.-C., Liu, M.-Y., Zhu, J.-Y., Tao, A., Kautz, J., and Catanzaro, B. (2017). High-resolution image synthesis and semantic manipulation with conditional gans. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8798–8807. 56

Xiao, H., Rasul, K., and Vollgraf, R. (2017). Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *CoRR*, **abs/1708.07747**. 12

Yazıcı, Y., Foo, C.-S., Winkler, S., Yap, K.-H., Piliouras, G., and Chandrasekhar, V. (2018). The unusual effectiveness of averaging in gan training. *arXiv preprint arXiv:1806.04498*. 43, 79

Yosinski, J., Clune, J., Bengio, Y., and Lipson, H. (2014). How transferable are features in deep neural networks? In *Advances in neural information processing systems*, pages 3320–3328. 28

Zhang, H., Xu, T., Li, H., Zhang, S., Wang, X., Huang, X., and Metaxas, D. N. (2017). Stackgan: Text to photo-realistic image synthesis with stacked generative adversarial networks. In *Proceedings of the IEEE international conference on computer vision*, pages 5907–5915. 56

Appendix A

Source code

For the sake of transparency and reproducibility the GAN code used to achieve the results of this thesis is uploaded to github: `agnalt/thesis`[1]

Do not hesitate to contact the author regarding any questions related to the source code or the work of this thesis.

[1] `https://github.com/agnalt/thesis`

Appendix B

Implementation details of a basic GAN

This appendix contains the source code for training the MLP GAN from section 3.2.3. Some readers may find this supplementary material useful to further understand the example of section 3.2.3. In addition, the code illustrates how the high-level API of `Keras` can be used to train a basic GAN model. The code is implemented in `Python 3.7.4` using `Tensorflow 2.1` and `Keras`. The code for generating the figures is not presented here, but can be found alongside with the source code for this thesis.

```
""" This module contains the code to train a basic GAN to learn the
distribution of the unit circle.

Use GPU for performance boost.

Eirik Agnalt Østmo
Tromsø, 2020 """


import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
import tensorflow.keras.layers as layers
from tqdm import tqdm

tf.random.set_seed(0)
np.random.seed(0)
```

```
def make_discriminator(n_inputs=2):
    """ GAN discriminator
    n_inputs: dimensionality of input data """

    model = tf.keras.Sequential()
    model.add(layers.Dense(256, activation='relu',
                           kernel_initializer='he_uniform',
                           input_dim=n_inputs))

    model.add(layers.Dense(1, activation='sigmoid'))

    # Compile the discriminator
    model.compile(loss='binary_crossentropy',
                  optimizer='adam', metrics=['accuracy'])

    return model


def make_generator(latent_dim, n_outputs=2):
    """ GAN generator
    latent_dim: dimensionality of input variable z
    n_outputs: dimensionality of output """

    model = tf.keras.Sequential()
    model.add(layers.Dense(256, activation="relu",
                           kernel_initializer="he_uniform",
                           input_dim=latent_dim))
    model.add(layers.Dense(n_outputs, activation="linear"))

    return model


def make_gan(generator, discriminator):
    """ Combined generator and discriminator. """

    # Disable training of discriminator as default
    discriminator.trainable = False

    # Create GAN
    model = tf.keras.Sequential()
    model.add(generator)
    model.add(discriminator)
```

```
    # Compile the GAN
    model.compile(loss='binary_crossentropy', optimizer='adam')

    return model


def real_distribution(n=200, r=1):
    """ Distribution of the real training data.
    A circle with radius r. """

    theta = np.linspace(0, 2*np.pi, n)

    x1 = r * np.sin(theta)
    x2 = r * np.cos(theta)

    return x1, x2


def generate_real_samples(n):
    """ Choose n random training points. """

    # Generate the population
    population = 1000000
    x1, x2 = real_distribution(population)
    X = np.c_[x1, x2]

    # Pick a subsample
    i = np.random.randint(0, population, n)
    X = X[i]

    # Generate class labels
    y = np.ones((n, 1))

    return X, y


def generate_latent_points(latent_dim, n):
    """ Sample points from latent space of the generator. """

    # Sample from uniform distribution
    x_input = np.random.uniform(-1, 1, latent_dim * n)
```

```
    # Make batch
    x_input = x_input.reshape(n, latent_dim)

    return x_input


def generate_fake_samples(generator, latent_dim, n):
    """ Generate fake samples with class labels """

    # Sample points from latent space
    x_input = generate_latent_points(latent_dim, n)

    # Forward pass
    X = generator.predict(x_input)

    # Generate labels for the discriminator
    y = np.zeros((n, 1))

    return X, y


def train(g_model, d_model, gan_model, latent_dim, training_data):
    """ The completion of one whole training epoch."""

    for batch in data:

        batch_size = batch.shape[0]

        # Real samples
        x_real = batch

        # Fake samples
        z = generate_latent_points(latent_dim, batch_size)
        x_fake = generator.predict(z)

        # Train discriminator
        # Send in real and fake samples with 1 and 0 as labels
        d_model.train_on_batch(x_real, np.ones((batch_size, 1)))
        d_model.train_on_batch(x_fake, np.zeros((batch_size, 1)))

        # Train the generator
        # Update the generator via the discriminator's error
        # with flipped labels
```

```
        gan_model.train_on_batch(z, np.ones((batch_size, 1)))

if __name__ =="__main__":

    # Make training dataset
    X, y = generate_real_samples(300)

    data = tf.data.Dataset.from_tensor_slices(X)
    data = data.batch(32)

    # Number of input dims
    latent_dim = 1

    # Create models
    discriminator = make_discriminator()
    generator = make_generator(latent_dim)
    gan_model = make_gan(generator, discriminator)

    # Training stats
    train_plot = []
    train_loss = []
    i = 0

    # Train the GAN
    epochs = 1200

    # Training loop
    for i in tqdm(range(i, i + epochs)):
        train(generator, discriminator, gan_model, latent_dim, data)

    # ... additional code to generate figures and evaluate the gan
    # can be found along with additional source code on github...
```

Index