UNIVERSITY OF BERGEN
DEPARTMENT OF INFORMATICS

MASTER'S THESIS

Generative Adversarial Networks for Annotating Images of Otoliths

Author: Emir Zamwa

Supervisor: Dr. Troels Arnfred Bojesen

January 30, 2023

UNIVERSITY OF BERGEN

Abstract

Faculty of Mathematics and Natural Sciences
Department of Informatics: Machine Learning

Master of Informatics

Generative Adversarial Networks for Annotating Images of Otoliths

by Emir Zamwa

This thesis explores the use of generative adversarial networks (GANs) for annotating images of otoliths to determine the age of fish. The proposed solution not only provides accurate age determinations, but also visual representations of the otolith images with growth rings marked with dots, making it applicable as explainable artificial intelligence. The convolutional neural network models I propose are based on Pix2Pix GANs and Wasserstein GANs, with the latter showing the success in my experiments. The successful models achieve an accuracy of 82.8% and 81.5% in age determination, including an offset of ± 2 from the real ages of the dataset.

Acknowledgements

"I would first and foremost express gratitude towards my fellow master's students who became good friends of mine during my period of studying at the University of Bergen. This includes Knut Holager, Mathias Madslien, Halvor Barndon and Hans Martin Johansen, in addition to John Isak Fjellvang and Johanna Jøsang. We have been through ups and downs, struggles and joyful moments, endless goals and ambitions of how much we can put in our lunch sandwiches, and let us not forget; the Covid-19 pandemic, together.

I had to postpone my thesis a whole semester because of an acting job in Oslo for the TV series Blodsbrødre / Gangs of Oslo (TV 2 Norway and Netflix globally). This resulted in me finishing one semester after you guys, and the comparison of being and not being in your company was like night and day. So a big thanks goes to you guys.

I want to say a huge thanks to my advisor Dr. Troels Arnfred Bojesen, for his amazing availability, feedbacks, support and humor. Him just as the person he is, occasionally surprising us with his delicious carrot cakes, was alone a big motivation for me. I want to thank Ketil Malde, who provided the topic of this thesis. I also thank Dr. Côme Denechaud for giving me access to his database of images and annotations on otoliths, and providing his knowledge on the topic. Without these, there would be no thesis.

I want to thank my mom as well, for all the support she gives me on everything I do on this planet named Earth that flies through the universe.

And last but not least, I want to thank me.
I want to thank me for believing in me.
I want to thank me for doing all this hard work.
I want to thank me for never quitting.
And I want to thank me for just being me at all times."

- Emir Zamwa
January 30, 2023

Contents

List of Figures

List of Tables

List of Abbreviations

AI	**A**rtificial **I**ntelligence
ML	**M**achine **L**earning
GAN	**G**enerative **A**dversarial **N**etwork
CGAN	**C**onditional **G**enerative **A**dversarial **N**etwork
ANN	**A**rtificial **N**eural **N**etwork
CNN	**C**onvolutional **N**eural **N**etwork
MLP	**M**ulti **L**ayered **P**erceptron
ReLU	**R**ectified **L**inear **U**nit
Sigmoid/σ	**Logistic Activation Function**
Tanh	**Hyperbolic Tangent**
RGB	**R**ed **G**reen **B**lue
Conv	**Convolutional**
FC	**Fully-Connected**
WGAN	**W**asserstein **GAN**
WGAN-GP	**W**asserstein **GAN** with **G**radient **P**enalty
GD	**G**radient **D**escent
SGD	**S**tochastic **G**radient **D**escent
BCE	**B**inary **C**ross **E**ntropy
Batch**N**orm	**B**atch **N**ormalization
Instance**N**orm	**I**nstance **N**ormalization

Chapter 1

Introduction

1.1 Overview

ARTIFICIAL INTELLIGENCE (**AI**) has been able to carry out an increasing number of human tasks over the past ten years thanks to Big Data, algorithm improvements, and the continuous advancement of computing power. Using *deep learning*, which is a subfield of *machine learning* (**ML**), has allowed for abilities like pattern-recognition, generalization, and experience-based learning [14]. While deep learning have become advanced in finding patterns and *recognizing* things, it was not good at *creating* them ten years ago. Since then, there has been many advancements in *generative* techniques. A few examples that have gained massive media attention in 2022 include *DALL-E* [65, 19] and *ChatGPT* [80, 46], in addition to *Midjourney* [43]. The increased popularity and interest of generative techniques originally began to grow after the introduction of *generative adversarial networks* (**GANs**), presented in 2014 by Goodfellow et al. [31]. This is because GANs gave machines "The gift of imagination" [28].

The idea introduced by GANs represents a competition between two *artificial neural networks*. These methods have enabled researchers to produce wholly computer-generated images of everything from people's faces to art that look realistic. Additionally, they have also allowed for the creation of contentious "deepfake" media [26, 74]. GANs can be used to mimic any data distribution whether it is images, texts, videoes or sounds. In this thesis specifically, we will look at the possibility of using GANs to help marine researchers determine the age of fish.

Many studies are conducted on fish to help understand how the species population react to environmental stressors such as climate change, human exploitation and how the population develops as a reaction to commercial and recreational fishing [16, 23]. The studies are conducted on different aspects of a fish's life such as growth, mortality and maturity. Scientists specifically look at the fish's age and rates of growth. This can be done by studying the ear bones called the *otoliths*, which are located in the ear of all fish except for sharks, rays and lampreys [12].

The technique involve counting natural growth rings on otoliths. As the fish grows, new rings are developed on their otoliths as a result, dependent on seasonal changes and their access to nutrition. Counting these rings on the extracted otolith results in the approximate age of the fish. This technique is very similar to aging trees by counting growth rings on the tree stump.

The growth rings on otoliths are today mostly counted manually by hand under microscopes after they are processed, which can be tedious and slow. The goal of this thesis is to research the possibility of using GANs to analyze images of processed otoliths and visually annotate where these rings are located automatically – potentially skipping the laboring manual process.

By using the proposed GANs of this thesis for annotations, not only can the age of the fish be determined, but the method will in addition also allow us to comprehend and potentially trust the results created by the machine learning model, which can be referred to as an *explainable AI* [77] – making it possible for marine researchers to determine if the results are reasonable.

1.2 The Otolith

Otoliths are calcium carbonate crystals located in the ear stone or ear bone of a fish as shown in Figure 1.1 as *H*.

It allows fish to hear and sense vibrations in their surroundings, and helps them to perceive directional indicators such as balance, gravity and movement. The size and shape of otoliths varies greatly between fish species. In fact, most species can be distinguished from one another just by their otoliths [24]. A few different shapes are shown in Figure 1.3. In this thesis however, we will study otoliths from the Northeast Arctic cod (Gadus morhua) shown in Figure 1.2.

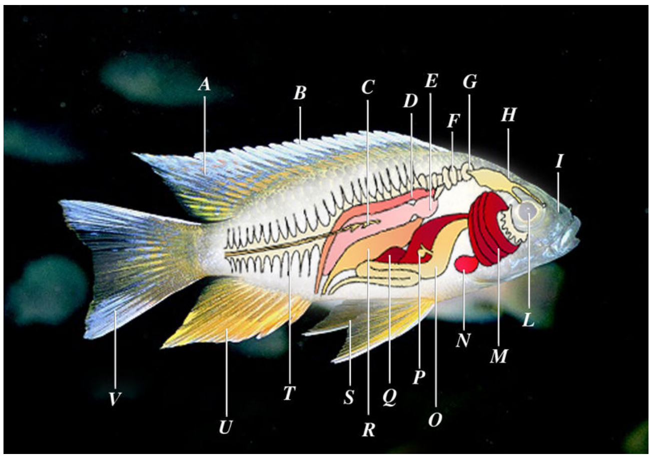


Figure 1.1: Location of the ear bone/otolith (*H*). Figure taken from Marrabbio2 [47].

Figure 1.2: Atlantic cod, Gadus morhua. Figure taken from Boulart [7].

As explained by Canada [12], there are three pairs of otoliths in each fish, including one large pair (the sagittae) and two small pairs (the lapilli and the asteriscii). The smaller pairs are about the size of the tip of a pin. The otoliths shown in Figure 1.3

are the larger *sagittae* pairs, which are usually used for determining age. These are the ones we will be focusing on in this thesis.

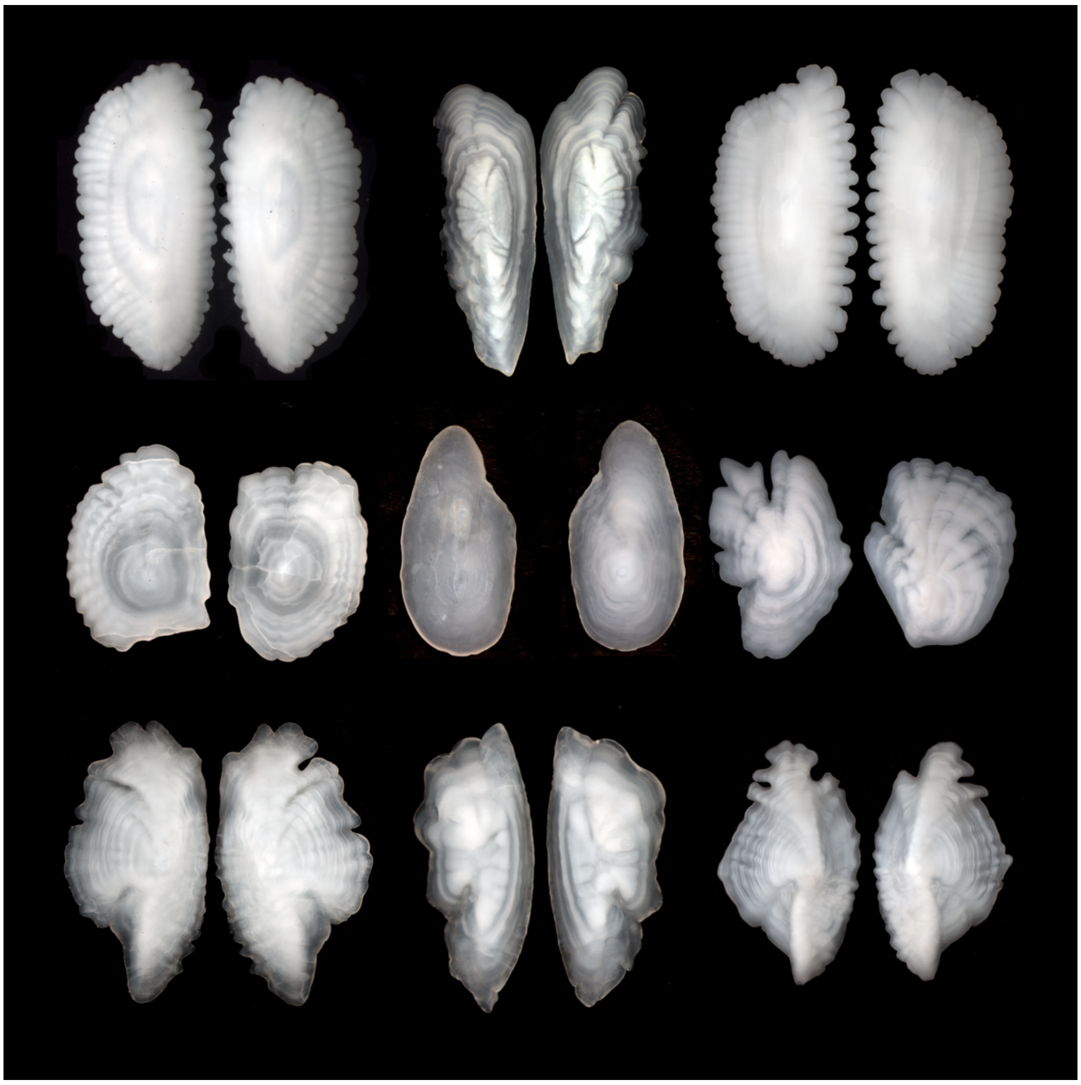


Figure 1.3: Otolith sagittae pairs (two per individual fish) from an assortment of Bering Sea fish species. Walleye pollock (top left) and Pacific cod (top right) are among the species shown in the figure. Note: otolith sizes are not on a relative scale. Figure taken from Fisheries [25].

1.2.1 Growth Rings

Otoliths exhibit both periods of low growth (winter) and high growth (summer) as they develop along with the rest of the fish (spring to autumn). Darker "narrow bands" called *checks* and lighter "wide bands" are produced by this pattern as shown in Figure 1.4.

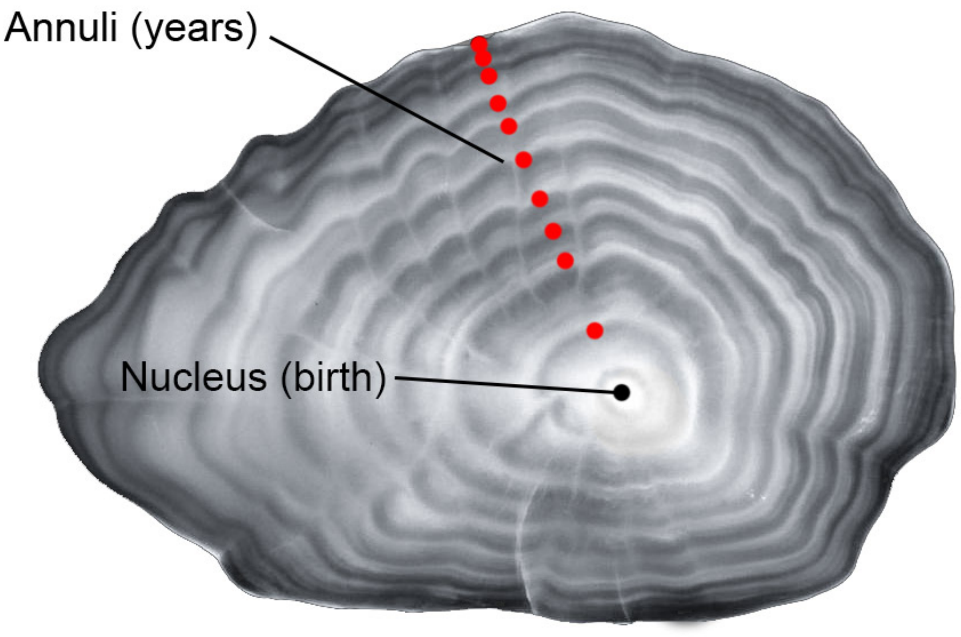


Figure 1.4: Visual growth rings on an otolith. Otoliths are usually sliced into thin crosswise sections with a low-speed saw for better visibility of the pattern of the bands. Red dots indicating each annuli. Figure taken from Brazier [8].

Checks are regarded as one winter's worth of growth. *Annuli* is the collective term for a narrow and a wide band together which represent one year's growth. These are counted to determine the fish's overall age.

According to Brazier [9], using otoliths is in most cases the most accurate way to determine the age of a fish, but this technique require the fish to be dead which sometimes is not preferable in scientific studies. For more details, see "Fish Hearing" [9] where Brazier explains the otoliths to a greater depth.

1.3 Computer Vision Assisted Age Determination

Computer vision is a field of AI and computer science that focuses on the ability of computers to interpret and understand visual data from the world around them. This involves developing specialized algorithms and machine learning models that can automatically detect, recognize, and track objects, faces, and other features in images and videos, as well as understand the context and meaning of the visual data [63]. Deep learning is widely used in computer vision to analyze images.

1.3.1 The Current Technology Sought To Further Expand Upon

In the field of age determination of fish, computer vision is already experimented with, and applied by marine researchers. An example is a program called DeepOtolith (`http://otoliths.ath.hcmr.gr/`) [58]. DeepOtolith takes an image of an otolith as input, then the ML model performs its algorithm on the image, and outputs the age prediction. This is done by using artificial neural networks to extract features from the input. A simple figure of the program is visualized in Figure 1.5.

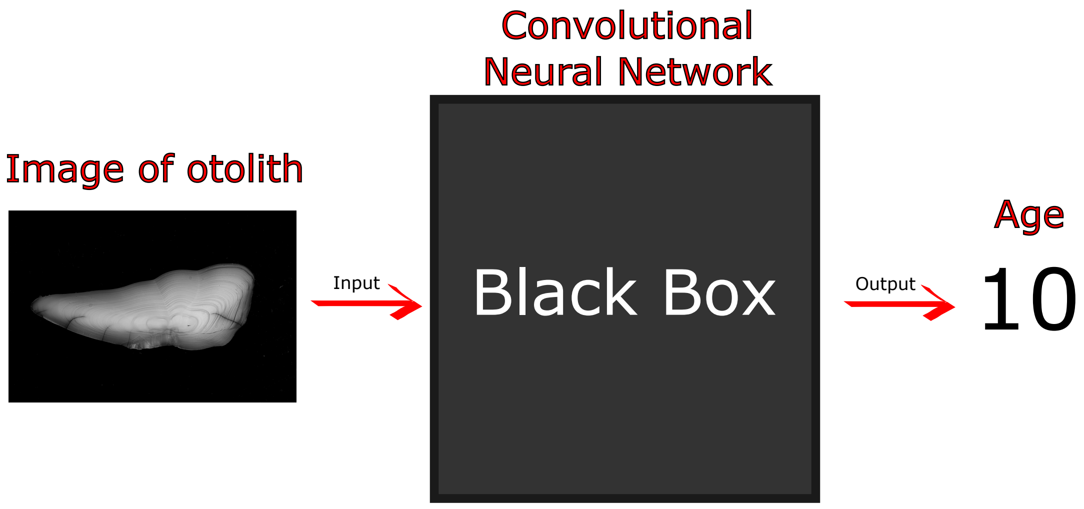


Figure 1.5: Simplified DeepOtolith architecture. Otolith image is used as input. A convolutional neural network then analyzes the image and outputs an age prediction. Otolith example used is for the purpose of visualization.

The goal of DeepOtolith is to predict the age of the fish from the input image by training the model in a *supervised learning*[1] fashion, where a dataset of otolith images and their respective ages are used for training. The problem is, the only way to determine if the results of the trained model are correct is by inspecting the otoliths manually.

The program itself applies a *convolutional neural network* that analyzes the input image and predicts an age as output, but everything else that happens in the program, including calculations and processes, happens under the hood without any transparency or explanation to the user. This can be referred to as a *black box*.

The question that arises is how does the program conclude an age for a specific otolith, like it has determined the age 10 for the otolith in Figure 1.5? Is there a simpler way to visualize the results instead of manually inspecting each otolith under a microscope or otolith image by hand for double-checking purposes?

1.3.2 Envisioned Solution

The goal of my thesis is to utilize a GAN to visually annotate where growth rings are located on otoliths. By using GAN as a tool to train a *generative model*, I will research the possibility of copying the manual by-hand process of marking dots on otolith growth rings for age determination by replacing this labor with machine learning. The GAN's *generator* (the generative model) will instead present a visualization of where each growth ring is located on the otolith image by marking them with dots as an explainable AI.

The training will utilize a dataset containing processed otolith images and their respective annotations' data. The GAN's generator will be trained to output dots by trying to fool the GAN's *discriminator* (the discriminative model), making it believe these dots are real when compared to images and human annotations. When the GAN training is complete, the generator trained by using the discriminator as a

[1]Supervised Learning is when a model receives guidance while learning from a labeled dataset. Unsupervised Learning is in contrast when the model is trained using unlabeled data without any supervision.

tool, will then be used to annotate unseen otolith images. The envisioned solution is visualized in Figure 1.6.

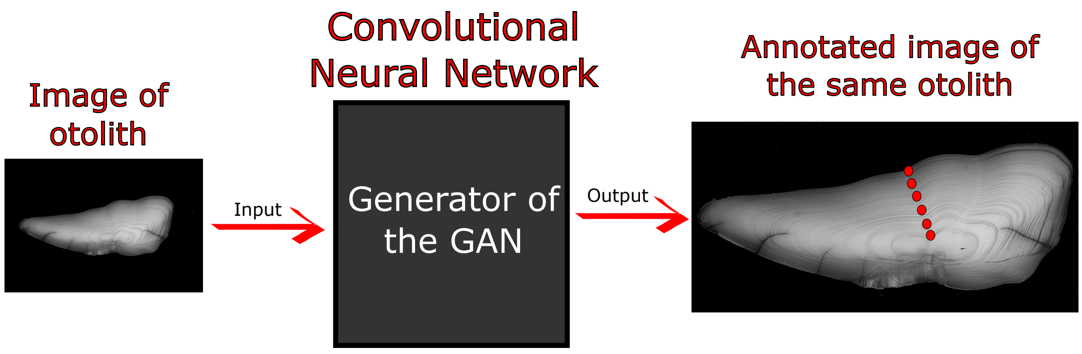


Image of otolith
Convolutional Neural Network
Annotated image of the same otolith
Input
Generator of the GAN
Output

Figure 1.6: Envisioned solution where a GAN's generator annotates growth rings on an otolith. Otolith image as input. Dots on annuli as output. By counting these dots either manually or with an algorithm, the approximate age will be determined. Note: image sizes are not to scale, and only different for the purpose of visualization.

As seen in Figure 1.6, an otolith image is used as input to the generator, and the envisioned output is visual dots on where the annuli are located on the otolith. With this solution, such a program can be used to annotate each growth ring otoliths, giving the researchers a way to visually see dots on each annuli without marking them manually. By using an algorithm to count the generated dots, the program will also provide the age determination of the fish as well. In addition to the age determinations, the resulting images can also be used to determine growth rates of fish by looking at the distance between each growth ring, potentially providing marine researchers more valuable information with this solution.

1.4 The Research Question

"What is the potential of using generative adversarial networks in developing a generative machine learning model for accurately annotating images of otoliths?"

1.5 The Chapters of the Thesis

- **Chapter 2 - Theoretical Background:** This chapter presents the technical background theory related to generative adversarial networks, where I will describe the concepts used in this thesis.
- **Chapter 3 - Related Works:** Where I will give a brief review of relevant literature to the study, and describe the basis work for which this thesis seeks to further expand upon.
- **Chapter 4 - Materials and Methods:** This chapter will explain the materials, methodical approach and considerations taken throughout the work.
- **Chapter 5 - Results:** Where I will provide the results of the experiments.

- **Chapter 6 - Discussion and Future Works:** Where I will analyze the results and discusses their implication. I will interpret the findings and address the limitations of the work, and describe the actions that can be done further in light of this research.
- **Chapter 7 - Conclusion:** Lastly, I will summarize how the main findings answered the research question.

Chapter 2

Theoretical Background

This chapter presents the theoretical background of GANs and their deep learning fundamentals and concepts.

2.1 Discriminative vs. Generative Models

In the field of machine learning, *discriminative* models and *generative* models are two broad classes of algorithms that can be used to learn the underlying structure of a given dataset [32]. These compute classifiers using various methods with varying degrees of statistical modeling. Following Jebara [39], we can define them as follows:

- A **discriminative model** is a model of the conditional probability $P(Y|X = x)$ of the target Y, given an observation x.
- A **generative model** is a statistical model of the joint probability distribution $P(X, Y)$ on given observable variable X and target variable Y. Can also be written as $P(X|Y = y)$, or $P(X)$ if there are no labels.

Generative models attempt to learn the probability distribution of a dataset, in order to generate *new* samples that are similar to the ones in the original dataset. This is typically done by learning the underlying patterns and relationships within the data and using that information to create new samples.

Discriminative models, on the other hand, learns to make predictions about the class or category of a given input by directly modeling the relationship between the input and the output. This means that they focus on learning the boundary between different classes in the data, rather than learning the underlying probability distribution of the data.

A generative model could for example learn to generate images of cats that look like real photos of cats, while the discriminative model could learn to tell a photo of a cat from a photo of a dog.

More formally, we want to move from an observation x to a label y in the context of classification, or probability distribution on labels. Without using a probability distribution, we can compute this directly using a distribution-free classifier. We can estimate the likelihood of a label given an observation, $P(Y|X = x)$ (using a discriminative model), and then base classification on that. Or, we can estimate the joint distribution, $P(X, Y)$ (using a generative model), compute the conditional probability $P(Y|X = x)$, and then base classification on that. These are more indirect, but more probabilistic, allowing for the application of more domain knowledge and probability theory [83]. Depending on the specific problem, different approaches are used in practice, and hybrids can combine the best aspects of several approaches.

Some examples of popular discriminative models are [32]:

- Logistic Regression
- Scalar Vector Machine
- Nearest Neighbor Search
- Conditional Random Fields
- Decision Trees and Random Forest
- Discriminator of GAN

Some examples of popular generative models are [64]:

- Naïve Bayes
- Bayesian Networks
- Markov Random Fields
- Hidden Markov Models
- Latent Dirichlet Allocation
- Generator of GAN

Both generative and discriminative models can be effective for different types of learning tasks, and which one is more appropriate for a given problem will depend on the specific characteristics of the data and the goals of the model.

2.2 Generative Adversarial Networks

The power of GANs lies in the competition between two artificial neural networks – one being generative and one being discriminative. It mimics the back-and-forth between two characters who are constantly attempting to outwit one another – a *generator* and a *discriminator*. The same set of data is used to train both networks, where the discriminator in addition use generated data from the generator.

2.2.1 Introduction

The generator is tasked with creating as *real*, data-like artificial outputs as possible. The discriminator attempts to distinguish between real and generated data from its knowledge gained when trained with authentic data from the original dataset. The generator modifies its parameters for producing new data in light of those findings. This competition continues until the discriminator is unable to distinguish between the authentic, real data and the ones created by the generator. An example case can be the creation of cat images as shown in Figure 2.1.

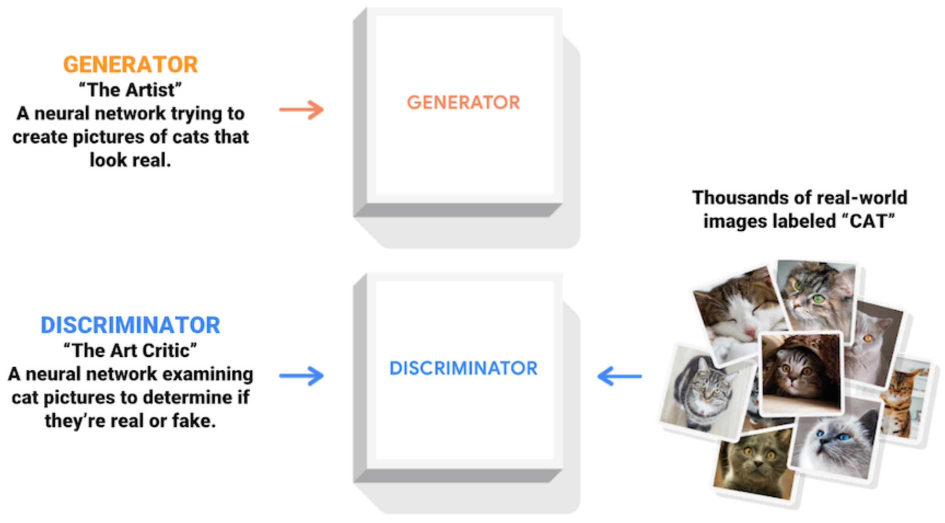

GENERATOR
"The Artist"
A neural network trying to create pictures of cats that look real.
GENERATOR
Thousands of real-world images labeled "CAT"
DISCRIMINATOR
"The Art Critic"
A neural network examining cat pictures to determine if they're real or fake.
DISCRIMINATOR

Figure 2.1: Simple example of the GAN concept in a case of cat images. Generator as "The Artist" in this case. Discriminator as "The Art Critic". Figure taken from TensorFlow [72].

Here, the generator will *generate* fake images of cats, while the discriminator decides whether they *are* real or fake. During *training*, the discriminator gets better at telling them apart as the attempts progresses, while the generator gets better at producing realistic-looking images of cats. When the discriminator no longer can distinguish between real and fake images of cats, the process has reached equilibrium as shown in Figure 2.2.

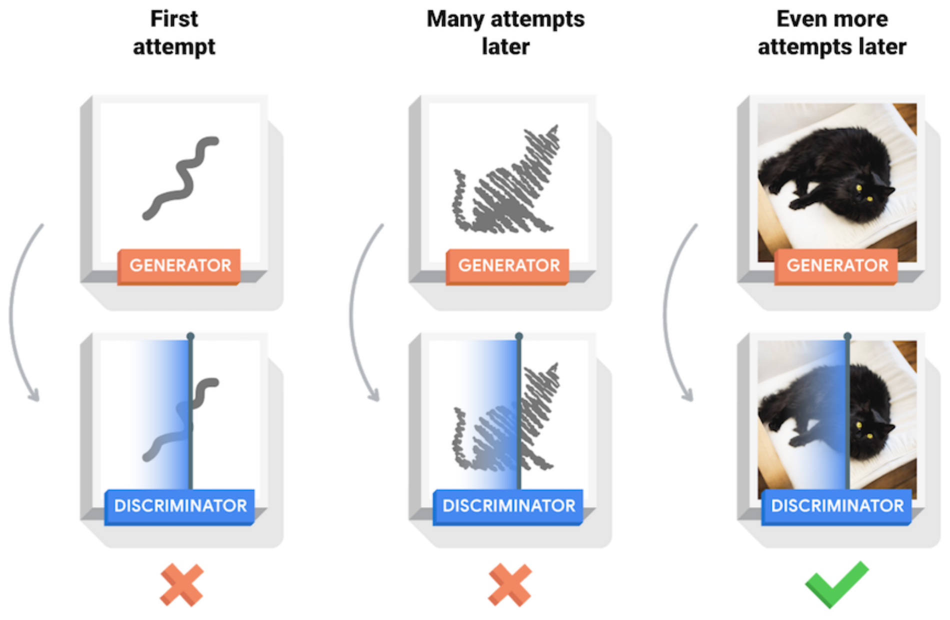


Figure 2.2: Simple example of a GAN's training process. Generator at the top creating fake images of cats. Discriminator at the bottom deciding if they are real or fake. Figure taken from TensorFlow [73].

2.2.2 General Architecture

To further describe the GAN in more general terms, we can visualize the standard architecture as shown in Figure 2.3:

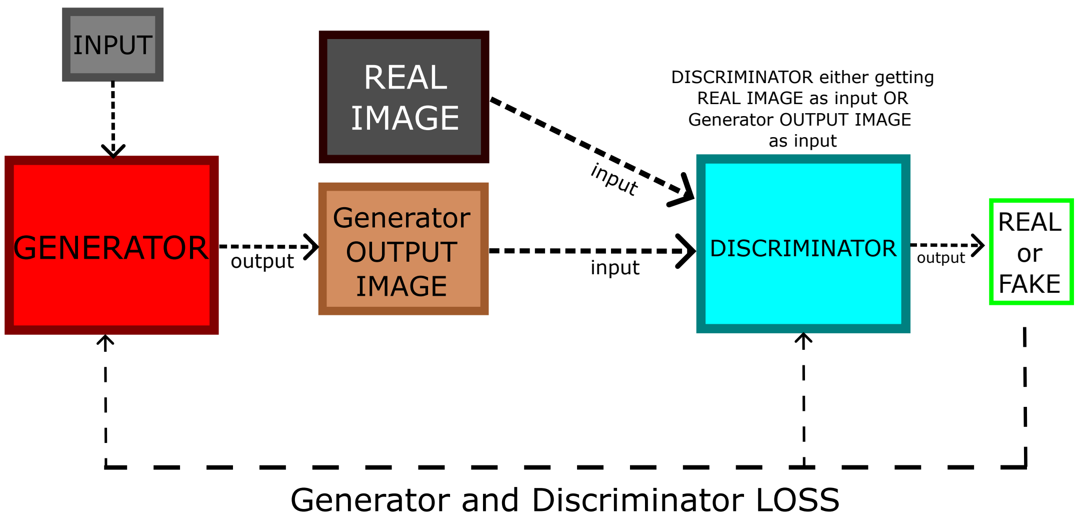

INPUT
REAL IMAGE
DISCRIMINATOR either getting REAL IMAGE as input OR Generator OUTPUT IMAGE as input
input
GENERATOR
output
Generator OUTPUT IMAGE
input
DISCRIMINATOR
output
REAL or FAKE
Generator and Discriminator LOSS

Figure 2.3: Standard GAN architecture, consisting of a generator and a discriminator. The generator has an arbitrary input, for example noise. "REAL IMAGE" are images from the original dataset that the discriminator has as input (if not "Generator OUTPUT IMAGE" is used, which is the generated image from the generator, different from its input). Discriminator output "REAL" or "FAKE" denotes whether the "Generator OUTPUT IMAGE" or the "REAL IMAGE" *is* real or fake, further used for *backpropagation* in the generator and the discriminator.

The generator must have an input. This input is usually random noise in general GANs, but it can for example also be an image (subsection 2.2.4). As visualized in Figure 2.3, the generator begins to generate data from this input. This generated data is then sent to the discriminator as its input, where it is analyzed to determine how close it is to being classified as real. After this classification, a *generator loss* is calculated, that is sent back to the generator using *backpropagation* (section 2.5) where the model readjusts its features to be able to generate better outputs. The same goes for the discriminator, receiving *discriminator loss* so that it can get better at classification.

The discriminator then receives this newly generated data once more, and the process continues. This process keeps repeating as long as the discriminator keeps labeling the generated data as fakes, because each time data is labeled as fake and with each backpropagation, the quality of the generated data keeps improving. Eventually, the generator will become so accurate that it will be difficult for the discriminator to tell the difference between data generated by the generator and real data. This *training process* is further described in section 2.7.

2.2.3 Conditional GAN

The inability to *regulate* the kinds of images produced by standard GANs presents a difficulty. The generator simply begins with random noise and repeatedly produces images that eventually start to resemble the training images.

This is resolved by a *conditional* GAN (**CGAN**), which makes use of additional information like labeled data or class labels. This may lead to faster or more stable training while also possibly improving the quality of the generated images. For instance, a CGAN can be trained to generate and discriminate numbers when given images and labels of various integers between $0-9$. An image of number 1 with its label "one", 2 with its label "two" and so on. Either the generator or the discriminator, or both, can have its inputs *conditioned* with a class label. This gives the generator or the discriminator the ability to know what they respectively are generating or discriminating. A standard GAN (also known as an *unconditional* GAN) relies solely on mapping the data in the latent space to that of the generated images in the absence of such a condition [37]. Compared to Figure 2.3, a CGAN can be visualized as shown in Figure 2.4.

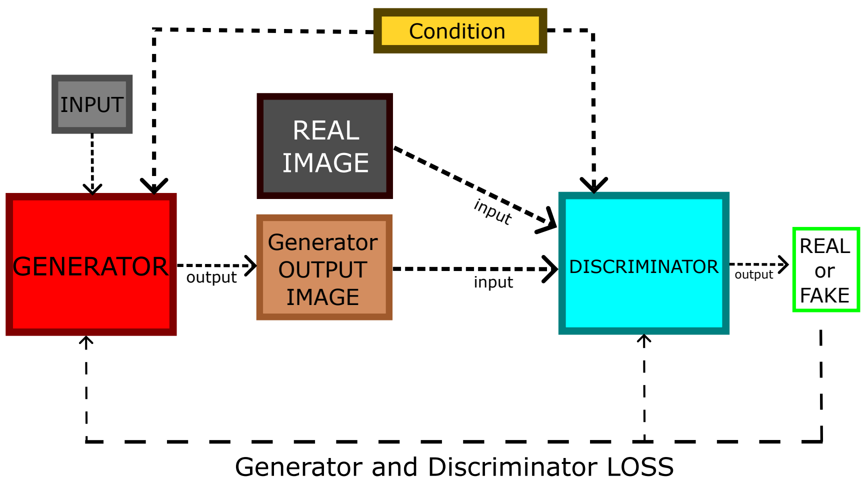


Figure 2.4: CGAN architecture, where both the inputs of the generator and the discriminator are conditioned with additional information, seen as the yellow box.

2.2.4 Image-to-Image Translation

My research question resolves around using GANs to produce images with annotations on where an annuli is located on an otolith, as visualized in Figure 1.6. The fundamental concept in the approach towards a solution was the use of *image-to-image translation* in my experiments. This technique is a conditional generation framework that translates an input image into a different output image, like for example inputting a grayscale image and outputting the same image in colors. The generation of the output image is dependent on an input, in this case, a *source* image. When given a source image and a conditioned *target* image, the discriminator must decide whether the generated image is a plausible transformation of the source image by comparing them to the target, real images. The architecture is shown in Figure 2.5.

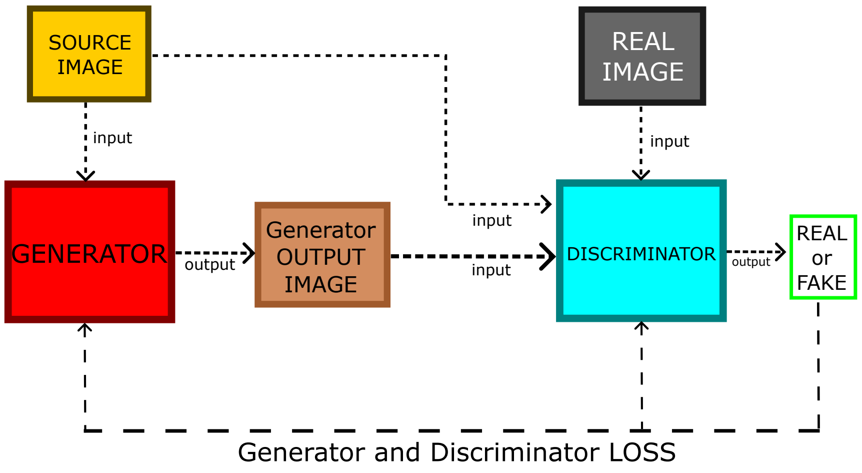


Figure 2.5: Image-to-image translation example, where the inputs are conditioned with a source image shown as the yellow box. Note that the "REAL IMAGE" and the "Generator OUTPUT IMAGE" are different from the "SOURCE IMAGE".

There are many ways of building an image-to-image GAN. This framework was originally presented by Isola et al. [38] called *Pix2Pix*, where the generator consists of a *U-Net* and the discriminator being a *PatchGAN*. We will further describe Pix2Pix in section 2.6, but before we can do that, we have to understand the building blocks of the generator and the discriminator. They consist of *artificial neural networks*, more specifically *convolutional neural networks*. These two concepts will be further described in the next two sections (section 2.3 and section 2.4).

2.3 Artificial Neural Networks

Image your brain, this incredible organ and all the amazing and advanced operations it is capable of doing. Artificial neural networks (**ANNs**) are a digital representation that is inspired by the same theoretical concepts of biological neural networks of a brain.

Our brains consist of billions of neurons, interconnected and linked together to form a highly complex network where information travels. Each neuron takes up new information, processes it, and then transmits electrical and chemical signals to other neurons throughout this network – resulting in our intelligence. A typical neuron is shown in Figure 2.6.

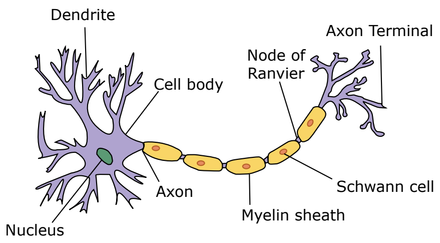


Figure 2.6: Typical neuron in a brain. Figure taken from Dhp1080 [17].

Now imagine that each of these neurons, these cells in your brain, instead is a *function*, that takes in the outputs of other functions, and sends out its signal as numbers. This digital representation of an artificial neuron is shown in Figure 2.7, and is also called a *perceptron*.

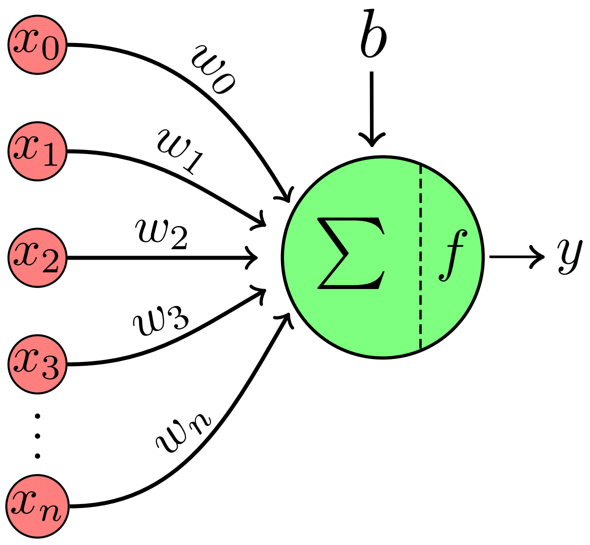


Figure 2.7: Artificial neuron, also called perceptron. $[x_0, x_1, \ldots, x_n]$ denoting the inputs with their corresponding weights $[w_0, w_1, \ldots, w_n]$. b denoting the bias, f the activation function and y the output. Figure taken from MartinThoma [48], and edited.

2.3.1 Perceptron

The perceptron is the fundamental block of an ANN. As seen in Figure 2.7, the perceptron has inputs $[x_0, x_1, \ldots, x_n]$. Each input x_i has its weight w_i. Every input scaled by corresponding weight, $w_i x_i$, goes into the neuron where some calculations happen, before the final number goes out as an output y, formulated in the following way for every perceptron [60]:

$$y = f(\sum_{i=1}^{n} w_i x_i + b), \tag{2.1}$$

where b is the bias and n is the number of the dimension of the input space. The output value y, called the *activation*, is the weighted sum of the input and weights plus the bias, transformed by f – the *activation function* (subsection 2.3.3).

Now one perceptron alone cannot solve complex tasks however, but connecting many perceptrons together in layers forming an ANN can. This forms a network called *multi-layered perceptron* (**MLP**).

2.3.2 Multi-Layered Perceptron

Simple MLP networks can consist of just a few layers, but having many layers forms a deeper neural network. This is where the term *deep learning* originated from. A visualization of a simple MLP is shown in Figure 2.8.

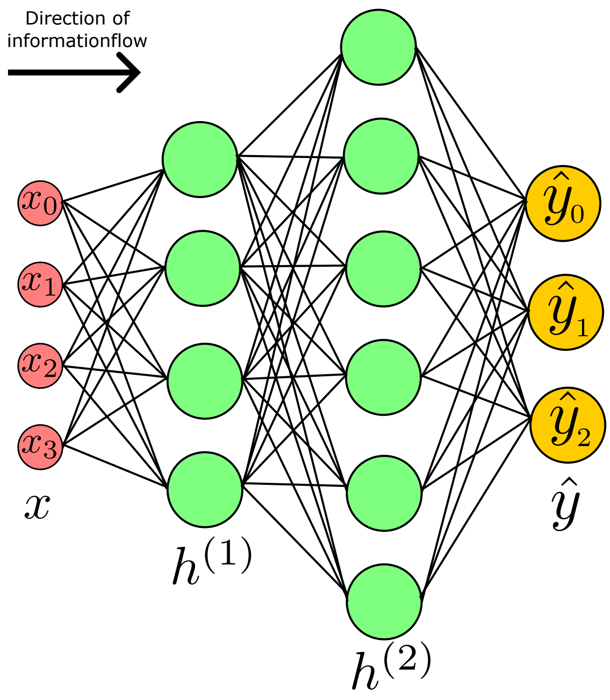


Figure 2.8: Simple example of an MLP. This feed-forward MLP has four inputs (x) in the input layer, two hidden layers $(h^{(1)}$ and $h^{(2)})$, and three outputs in the output layer $(\hat{y})$.

This MLP shows a standard ANN *architecture* – consisting of an input layer and an output layer, with user-defined amount of hidden layers in-between [60], also called fully-connected layers (subsection 2.4.4). Because the data flows from the input to the output layer, and each layer is a function of the previous layer, an MLP is a type of network known as a *feed-forward* neural network. Depending on the input, different neurons in the MLP will activate at varying intensities, producing various outputs. We can express the MLP in Figure 2.8 as follows [30]:

$$h^{(1)} = f^{(1)}(W^{(1)T}x + b^{(1)}), \tag{2.2}$$

$$h^{(2)} = f^{(2)}(W^{(2)T}h^{(1)} + b^{(2)}), \tag{2.3}$$

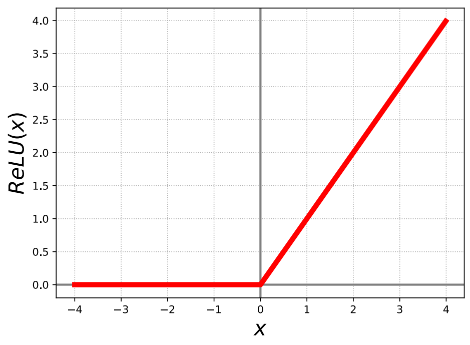
$$\hat{y} = f^{(3)}(W^{(3)T}h^{(2)} + b^{(3)}), \tag{2.4}$$

where each hidden layer h is a vector of *non-linear* activations, meaning the y's from Equation 2.1. W^T is a transposed matrix of the weights (necessary in order for the dot product to be possible), f is an activation function applied element-wise, b is a vector of biases, and $\hat{y}$ is a vector of the outputs. Let us now understand what these *activation functions* are.

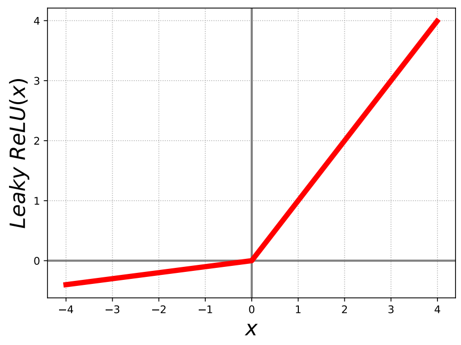
2.3.3 Activation Functions

Any MLP made up only of linear functions would only represent a linear mapping of input to output, regardless of the depth. As a result, activation functions are used to add *non-linearity* to the neuron outputs and give the MLP the ability to approximate more challenging functions. A network cannot represent non-linear functions if the hidden layers do not have any activation functions, but thanks to them, it can.

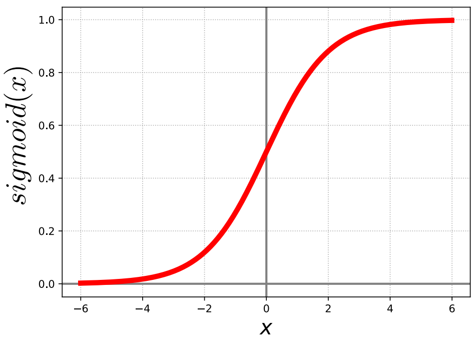
There are many kinds of activation functions in use [5], but depending on the task at hand, low computational cost, differentiability, and zero-centeredness are generally desired properties. Rectified linear unit (**ReLU**), the logistic activation function (**sigmoid**), and the hyperbolic tangent (**tanh**) are the three most popular and widely used activation functions and also the ones I use in my experiments, in addition to **LeakyReLU**. These are shown in Figure 2.9.



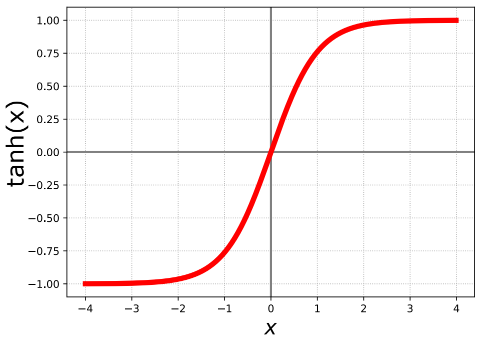
(a) ReLU



(b) LeakyReLU



(c) Sigmoid/σ



(d) Tanh

Figure 2.9: The activation functions used in the thesis.

ReLU and LeakyReLU

ReLU, as shown in Figure 2.9a, is a piecewise linear function that, if the input is positive, outputs the input directly; if not, it deactivates the neurons and outputs zero, formulated as follows for input x:

$$\text{ReLU}(x) = \max(0, x). \tag{2.5}$$

Because of this, the ReLU function results in being computationally efficient when compared to sigmoid or tanh. In addition to the advantage of being computationally efficient, ReLU also overcomes the *vanishing gradient problem* (subsection 2.5.2), which describes a situation where a deeper MLP is unable to propagate useful gradient information of the output end of the model back to the layers near the input end [11]. Since it is simpler to train and frequently results in better performance, it has evolved into the standard activation function for many types of neural networks.

The negative side of ReLU however, is the occasional instance of the *dying ReLU* *problem*, where during the backpropagation process, biases and weights for the neurons that are not above zero never get updated. This can create dead neurons that never contribute to the result, hence the term *dying* ReLU problem. To solve this issue, there is an improved version of ReLU called the LeakyReLU function, as shown in Figure 2.9b, formulated as follows:

$$\text{LeakyReLU}(x) = \max(0.1x, x). \tag{2.6}$$

Here, there is a tiny positive slope in the negative area of the function so that all neurons, instead of being either completely activated or deactivated, rather are *more* or *less* activated. This can be a safer choice in regard to dead neurons.

Sigmoid and Tanh

The sigmoid (σ) function, as shown in Figure 2.9c, accepts any real values as an input and produces output values between 0 and 1, formulated as follows:

$$\sigma(x) = \frac{1}{(1 + e^{-x})}. \tag{2.7}$$

The balancing feature of this activation function is that the output value will be closer to 1 the *larger* (more positive) the input value, and closer to 0 the *smaller* (more negative) the input value. σ is mostly used where predicting the probability in the output is the goal, since probability is in the range of 0 to 1. Therefore, it is commonly used in the output layer of a neural network.

An advantage of σ is that it provides a smooth gradient because of the function being differentiable. It prevents jumps in the output values [5].

The tanh function as shown in Figure 2.9d, is very similar to the σ function – having the same shape. The only difference between them is that tanh has the range of -1 to 1 compared to σ's 0 to 1, formulated as follows:

$$\tanh(x) = \frac{(e^{x} - e^{-x})}{(e^{x} + e^{-x})}. \tag{2.8}$$

The advantage here is that tanh is zero-centered, making it easy to interpret the output values as strongly positive, neutral or strongly negative.

The simple ANN I have presented in this chapter describe the basic concepts and how machine learning architectures are made. In computer-vision, these networks are further expanded, where the architecture is made up of convolutional neural networks. These deep learning networks are built using the same concepts, but their building blocks and layers are different. Now that the fundamentals are presented, I will in the next section move on to describe convolutional neural networks to give an understanding of the building blocks of the GANs used in this thesis.

2.4 Convolutional Neural Networks

Convolutional neural networks (**CNNs**) are a subset of ANNs that are primarily employed in image-related machine learning tasks [79]. Images are after all just a collection of numbers indicating the brightness for each pixel, where the amount of pixels becomes the input vector in an MLP. A black and white (grayscale) image of resolution 1024×1024 would require $1 \times 1024 \times 1024 = 1048576$ weights per pixel in the first hidden layer of the MLP containing one neuron. If the same image was to be colored, it would have three times as many weights for that one neuron because of the red-green-blue (**RGB**) color channels.

Now imagine having multiple neurons and multiple hidden layers, and a larger input image; when a neural network has more layers, the number of parameters increase quickly, hogging the resources and memory of even a modern computer to the point where training would not be possible [68]. Therefore, it is highly recommended in computer vision or image recognition tasks to use CNNs [27].

Another advantage with using CNNs, is that we utilize the equivariant information in the image, meaning that the pattern of a cat is the same whether it is in the center of the image or in the corner. In addition to this, we assume that the data shows some form of locality, meaning that pixels that are near each other typically are stronger correlated than the pixels further away from each other. By utilizing these assumptions, much fewer parameters are needed and learning may become easier, more stable and faster compared to the MLP example.

The modern CNN is constructed around three main parts, but can still be a CNN by only using the first: a *convolutional layer*, a *pooling layer*, in conjunction with a fully-connected layer. These layers used together address the MLP problem and gives fewer learnable parameters. A CNN example is shown in Figure 2.10.

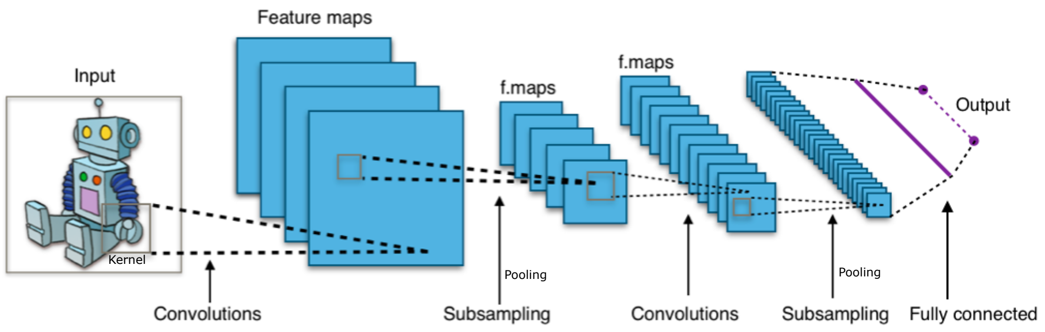


Figure 2.10: A typical CNN consisting of five layers. Two convolutional layers, two pooling layers and one fully-connected layer. Figure taken from Aphex34 [2].

2.4.1 The Convolutional Layer

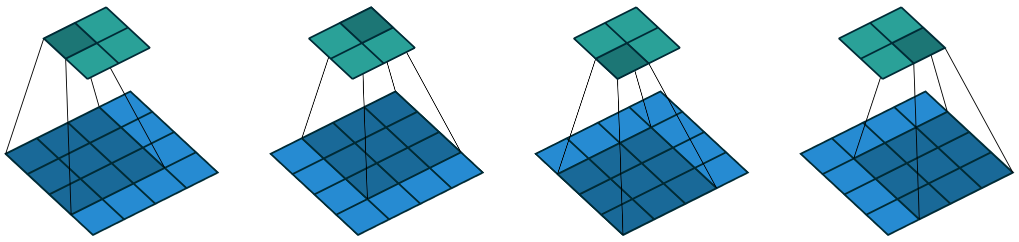
The convolutional (**Conv**) layer is the fundamental building block of a CNN. The conv layer is responsible for extracting features from the input image and reducing the dimensionality of the data by creating a condensed representation of the input. They are an essential component of CNNs, which enables CNNs to achieve state-of-the-art performance on a variety of tasks such as image classification, object detection, and semantic segmentation (chapter 3).

Conv layers are composed of a set of filters, also known as *kernels*. Each kernel is a small matrix of weights that is applied to a local region of the input image, known as the *receptive field*. The use of kernels allows conv layers to learn spatial hierarchies of features, with lower-level kernels learning basic patterns such as edges and corners, and higher-level kernels learning more complex patterns such as shapes and objects. By using convolutions, we can possibly be able to detect each annuli of an otolith. The output of the convolutional layer is also typically down-sampled, or reduced in resolution, which helps to further reduce the dimensionality of the data and improve the robustness of the model to small variations in the input.

The convolution operation involves element-wise multiplication of the weights in the kernel with the corresponding input pixels, followed by summation of the results to produce a single output value for each receptive field. This process is repeated for every receptive field in the input image, producing a set of output values that form a feature map. If the input image is a 2-dimensional image $I(x, y)$, for example like the robot in Figure 2.10, a convolution operation can be expressed as follows [30]:

$$(I * K)(x, y) = \sum_{i=1}^{n} \sum_{j=1}^{n} I(i, j) K(x - i, y - j), \tag{2.9}$$

where n is the size of the convolutional kernel and $I(i, j)$ and $K(x - i, x - j)$ represent the input pixels and kernel weights at position (i, j), respectively. Note that the kernel does not have to be quadratic. The output from the convolution operation is then usually fed through an activation function like ReLU, discussed in subsection 2.3.3. The number of horizontal positions the kernel moves in the input between each calculation is represented by a value called *stride*. If the kernel cannot move any further horizontally, starting from the left for example, it will, if possible, move vertically down equal to the stride value, and start over. This operation is visualized in Figure 2.11, but in practice can be done in no particular order or in parallel:



(a) 1st operation **(b)** 2nd operation **(c)** 3rd operation **(d)** 4th operation

Figure 2.11: Convolution with *stride* = 1 and *kernel* = 3 × 3 on input image of 4 × 4 pixels. Figures taken from Dumoulin and Visin [20].

An example of the calculation is visualized in Figure 2.12, with a vertical edge detection kernel:

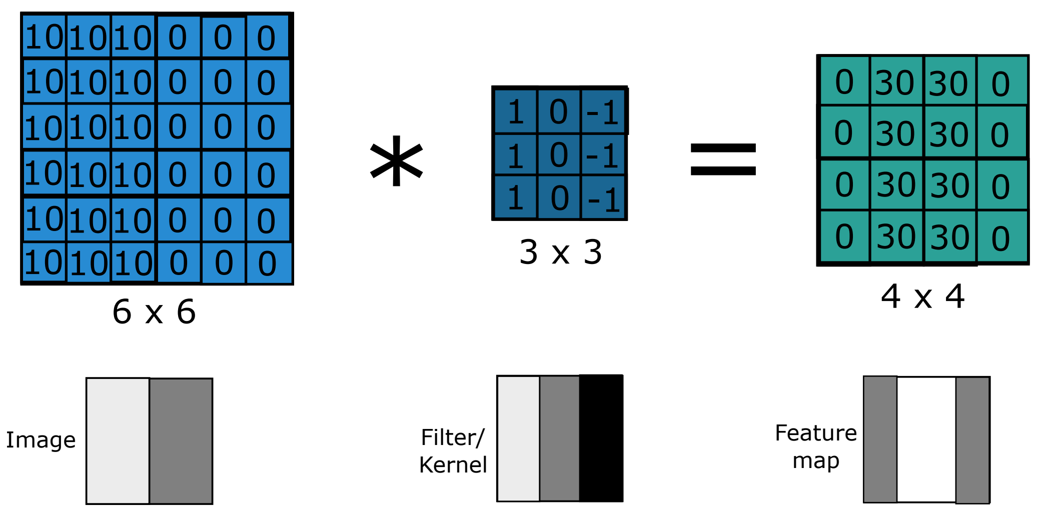


Figure 2.12: Example of a convolution. The *kernel* is applied repeatedly across the input image, in the same manner as Figure 2.11 with *stride* = 1. The figures at the bottom shows the image, kernel and output (feature map, activation) as color-gradings. Example taken from Ng [51] and edited.

One key advantage of conv layers is that they are highly efficient and can learn to recognize patterns in images using relatively little data. This is because the same set of weights is applied to every receptive field in the input image, allowing the network to learn local patterns without having to store separate weights for each individual input pixel. This also allows conv layers to generalize to new inputs.

Padding

Padding refers to the amount of extra pixels that is added to an image when it is processed by the kernel in a convolution. The padding value can be set by the user, but usually is set to zero. In Figure 2.13, we see an example of an image zero-padded with one pixel, adding an extra border.

Padding is an important concept in CNNs, as it allows for the preservation of spatial information and helps to ensure that the output feature map has the same dimensions as the input image. This can be useful for maintaining the spatial structure of the data and for ensuring that the output of a conv layer easily can be fed into subsequent layers in the network.

Furthermore, padding can be added to a CNN-processed image for more precise image analysis [15], like for example if the edge of the image contains important details, padding is recommended.

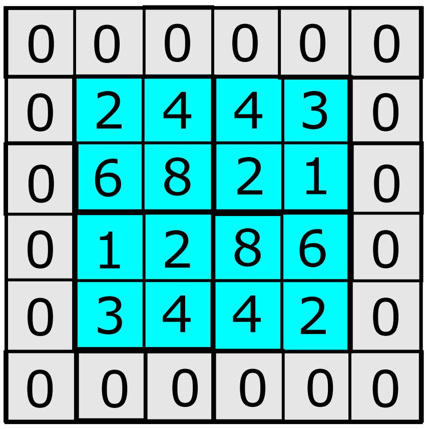
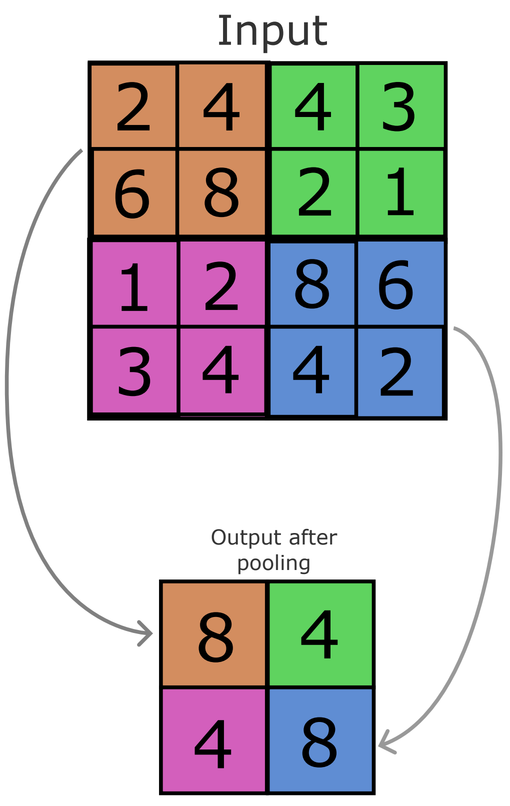


Figure 2.13: Zero-padding added to a 4×4 image. The resolution with padding equals 6×6. Blue box is the image, gray is the padded border.

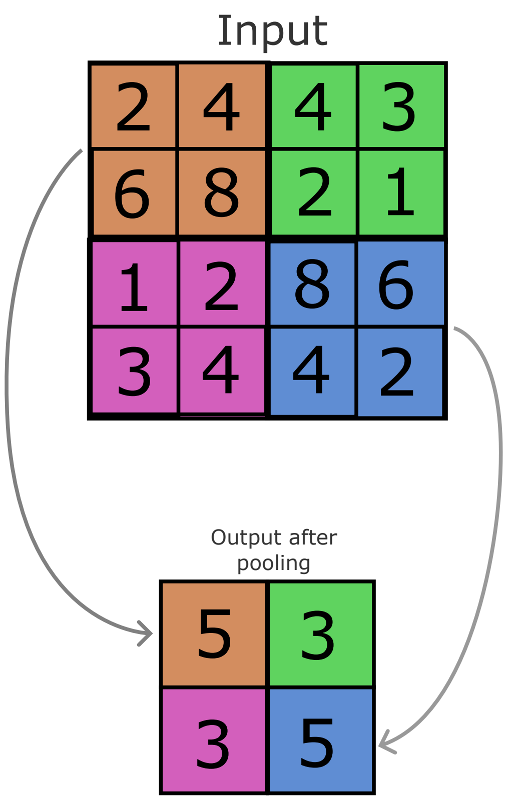
2.4.2 The Pooling Layer

To further reduce the feature maps' parameters, pooling layers are introduced. Pooling layers summarize the features generated by the conv layers. Reduced parameter feature maps are created as a result, and they are also more resistant to changes in spatial location in earlier layers [45]. In simple terms, the pooling layer *reduces* the height and width of its input.

Convolution operations are learned, whereas pooling is a fixed function. *Average pooling* and *max pooling*, which, respectively, compute the average or maximum values of all patches in the given feature map. They are two of the most frequently used pooling operations. They are visualized in Figure 2.14:



(a) Max Pooling



(b) Average Pooling

Figure 2.14: Pooling operations with 2×2 layer of *stride* $= 2$.

Pooling operations are mainly used to decrease computational costs of CNNs, but this again can be at the cost of potentially loosing important features or details of the input. By using pooling, we reduce the spatial resolution and can lose a feature's exact position if not techniques to preserve these are used to compensate. Most commonly used, is a pooling layer of 2×2 with a *stride* $= 2$, applied in Figure 2.14.

Average pooling adds a small amount of translation invariance, which means that changing the image by a small amount has little impact on the values of the majority of pooled outputs. Max pooling extracts more pronounced features, like edges, whereas average pooling extracts features more smoothly.

2.4.3 Convolution instead of Pooling

As stated in subsection 2.4.2, pooling operations are mainly used to decrease computational costs of CNNs, at the cost of loosing detail of inputs. To negate this problem, pooling layers can be replaced by conv layers. The advantage of the conv layer is that it can learn certain properties when downscaling its input that pooling layers cannot. The fixed operation of pooling can instead be learned by the strided convolution, with the only downside being increased number of parameters. The paper *Striving for Simplicity: The All Convolutional Net* by Springenberg et al. [69] demonstrates this by building their network of only conv layers, improving the overall accuracy of a model with the same width and depth. They state that "when pooling is replaced by an additional convolutional layer with stride $r = 2$, performance stabilizes and even improves on the base model" [69].

2.4.4 The Fully-Connected Layer

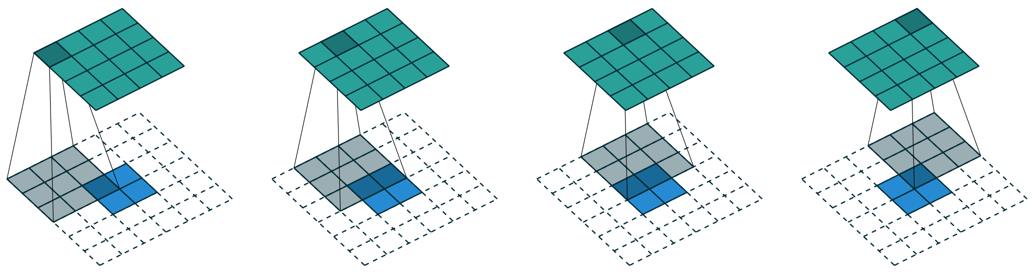
The fully-connected (**FC**) layer "flattens" the output of the preceding layers into a single vector that can be used as an input for the following layer. In essence, the FC layer is learning a non-linear function in the meaningful, low-dimensional, and somewhat invariant feature space that the conv layers have provided. Connecting multiple FC layers in a neural network will simply just become a standard feed-forward network as the neural network visualized in Figure 2.8. FC layers are usually used in the last few layers of a CNN towards the output in classification.

Fully-Connected Layer as a Convolutional Layer

One advantage of using a conv layer instead of an FC layer is that it can handle arbitrary input sizes, which means that it can work with inputs of different dimensions, such as images of different sizes. This allows the CNN to automatically adapt to the size of the input, without the need to specify the input size in advance, like we have to in an FC layer. Converting the FC layers into conv layers makes the network scalable. Therefore, designing a CNN by substituting all FC layers with conv layers will result in a more flexible network architecture [13].

2.4.5 Transposed Convolutions

A transposed convolution is a similar operation to a standard convolution, but rather than mapping the input to a lower resolution output, the transposed convolution does the opposite – mapping the input to a higher resolution output. The strided kernel that slides over the input, similar to Figure 2.11, is shown in Figure 2.15:



(a) 1st operation **(b)** 2nd operation **(c)** 3rd operation **(d)** 4th operation

Figure 2.15: First four transposed convolution operations with *stride* $= 1$ and *kernel* $= 3 \times 3$ on input image of 2×2 pixels. Empty cells in the figure are zero-padded in a temporary matrix. Figures taken from Dumoulin and Visin [21].

An example of the transposed convolution is visualized in Figure 2.16:

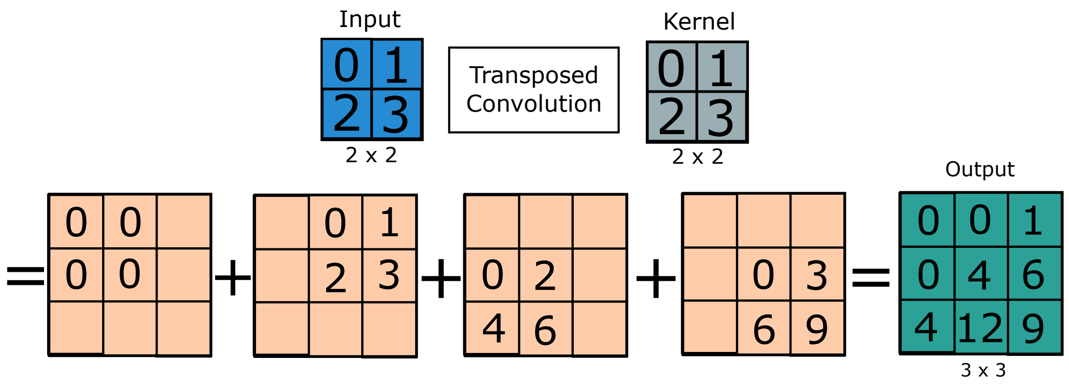


Figure 2.16: Example of a transposed convolution. The *kernel* is applied repeatedly across the input image, in the same manner as Figure 2.15 with *stride* = 1. Figure taken from Zhang et al. [91], and edited.

In a standard CNN, the input is downsampled and features are extracted towards the output like for classification. But in my GAN case however, we want to generate a *new* image from the input image in addition to extracting important features. That is where transposed convolutions comes into play. These layers are used when we want to upsample the previous layers after they are downsampled. And by building a network consisting of only conv layers, replacing pooling layers and fc layers, the GANs of the experiments become flexible.

Now that we understand the building blocks of the GANs used in this thesis, we will in the next section move on to describe *how* the parameters of neural networks are updated.

2.5 Gradient-based Optimization

Gradient-based optimization is a widely used method for training machine learning models by minimizing a given loss function. In order to efficiently compute the gradients of a network, we use an algorithm called *backpropagation*. It uses the chain rule of calculus and works by computing the error at the output layer and then propagating it backwards through the network to compute gradients with respect to each weight in the network. By propagating the error backwards, it allows the algorithm to be efficient and handle large neural networks [42]. Optimization algorithms such as *mini-batch stochastic gradient descent* (**SGD**) and *Adam* employ backpropagation to find the optimal set of model parameters that minimize the loss function.

2.5.1 Mini-batch Stochastic Gradient Descent

SGD is one of the most popular optimization algorithms used in practice [31]. It is a variant of standard gradient descent (**GD**) in which the model parameters are updated based on a small subset of the training data, rather than the entire dataset. This is known as a *mini-batch*. Using one mini-batch, we modify the network's parameters and then repeat on a fresh mini-batch. When we have completed this process on all the data, we refer to it as an *epoch*, and then start over [30]. By sampling the training data randomly, noise is also added to the optimization process, providing regularization (subsection 2.8.1).

The update of the parameters is controlled by a variable called the *learning rate*. The learning rate, which is typically between 0 and 1, determines how much the optimizing algorithm will scale a parameter. A high learning rate will take large steps towards the optimal solution, but it may overshoot or oscillate. On the other hand, a low learning rate will take smaller steps, but it may converge slowly or get stuck in local minima. The update rule for mini-batch SGD is given by [91]:

$$w_{t+1} = w_t - \eta \nabla L_{b_t}(w_t), \tag{2.10}$$

where w_t is the weight vector at iteration t, η is the learning rate, $\nabla L_{b_t}(w_t)$ is the gradient of the loss function L with respect to the weights w evaluated on a random subset (mini-batch) b_t of the training data at iteration t.

With the approach of SGD, not only the time per iteration is basically a constant, but the memory needed to compute the gradient is also constant [53]. So while each iteration of the SGD is less accurate than the standard GD, the fact that each iteration is cheap, allows us to do more of them. The resulting efficiency and frequent model parameter updates of SGD makes it more scalable than a standard GD, making it a good choice when working with larger datasets.

Momentum

A popular technique that can be used to improve the convergence of mini-batch SGD is *momentum*, which involves adding a *velocity* to the update step [71]. This term is based on the past weight updates and helps to smooth out the optimization process by incorporating a moving average of the gradients so that the update will step in the direction of the velocities rather than just the gradient currently in effect. It allows the optimization to move smoothly through the valleys and ravines of the loss function. This can help the model to escape from local minima and improve convergence on noisy or ill-conditioned datasets. Adding momentum to Equation 2.10 can be formulated as follows [49]:

$$\begin{aligned} V_{t+1} &= \gamma V_t + \eta \nabla L_{b_t}(w_t), \\ w_{t+1} &= w_t - V_{t+1}, \end{aligned} \tag{2.11}$$

where V_t is the velocity vector at iteration t and γ is the momentum hyperparameter (a user-defined value between 0 and 1). The velocity term keeps track of the past gradients and helps to smooth out the movement of the weight vector. When the momentum parameter is set to a high value, the previous gradients have a greater influence on the current update and the model is less sensitive to the noise in the gradients computed on the mini-batch.

Adam Optimizer

One of the disadvantages of mini-batch SGD, with or without momentum, is that it is sensitive to the learning rate, which can lead to slow convergence or divergence of the algorithm, depending on the value. The learning rate is often set through trial and error, which can be time-consuming and may not lead to optimal performance. Furthermore, traditional SGD can have difficulty dealing with sparse gradients, which are common in large-scale machine learning problems [40]. Additionally, the algorithm does not have an automatic balancing feature of the gradients, causing it to become sensitive to their scales, affecting performance.

To overcome these limitations, an optimization algorithm called Adam was proposed [41]. Adam, which stands for Adaptive Moment Estimation, is an extension of SGD that uses moving averages of the gradient and squared gradient to adapt the learning rate. It is similar in concept to momentum, but not exactly the same. Whereas momentum described in section 2.5.1 can be seen as a ball running down a slope, Adam behaves like a heavy ball with friction, which thus prefers flat minima in the error surface [34]. This helps to stabilize the learning process and improve the model's generalization ability, in addition to being more resistant to the vanishing gradient problem (subsection 2.5.2). The moving averages are formulated as follows [62]:

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1)\nabla L_{b_t}(w_t), \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2)(\nabla L_{b_t}(w_t))^{\odot 2}, \end{aligned} \tag{2.12}$$

where m_t and v_t are estimates of the first moment and the second moment (the mean and the uncentered variance, respectively), and $\odot 2$ denotes that the squaring of the gradient is done element-wise. The parameters β_1 and β_2 of the algorithm regulate the decay rates of these moving averages. These two terms are similar to the momentum term in the sense that it is a way to give more importance to recent gradients. Using Equation 2.12, the final update rule for the Adam optimizer is given by [62]:

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{v_t} + \epsilon} m_t, \tag{2.13}$$

where ϵ is a smoothing term to avoid division by zero.

2.5.2 Vanishing Gradient Problem

The vanishing gradient problem refers to the difficulty in training deep neural networks when the gradients of the weights with respect to the loss function have very small values. This can occur when the weights are updated using the backpropagation algorithm, as the gradients are multiplied by the weights on the way back through the network. If the gradients are small, the weights will be updated by only a small amount, and the network may not learn effectively. It can also make the training process very slow or even prevent it from converging.

Gradient-based optimization is essential in the training of machine learning models, with the algorithms mentioned being popularly adopted in practice. In this thesis, mini-batch SGD in conjunction with Adam optimizer is employed in all the experiments.

2.6 Pix2Pix GAN

Pix2Pix is an image-to-image translation CGAN that learns a mapping from input images to output images. The framework was presented by Isola et al. [38] in 2016, consisting of a *U-Net* generator and a *PatchGAN* discriminator.

2.6.1 U-Net Generator

For the generator in a GAN to be able to produce new images from input images as an image-to-image translator, its architecture is made of a network called *U-Net*. U-Net was initially developed by Ronneberger, Fischer, and Brox [61] for biomedical image analysis. The developed U-Net is shown in Figure 2.17. The generators of the GANs in my experiments uses a modified version of this fundamental architecture.

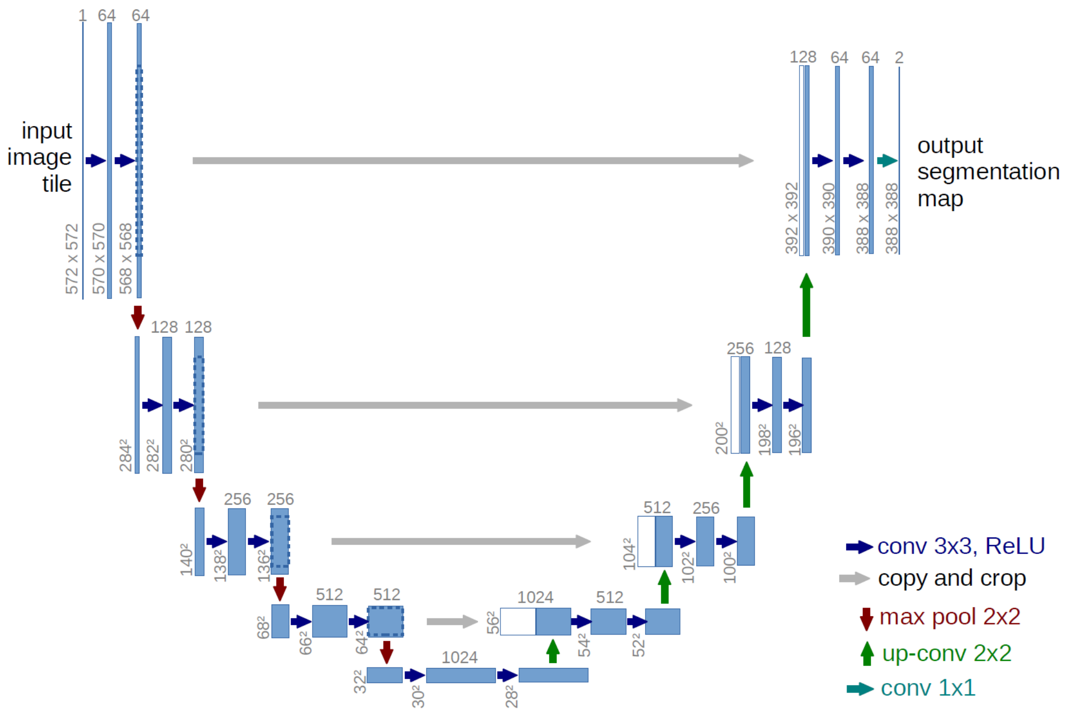


Figure 2.17: U-Net architecture (example for 32x32 pixels in the lowest resolution). Each blue box corresponds to a multi-channel feature map. The number of channels is denoted on top of the box. The x-y-size is provided at the lower left edge of the box. White boxes represent copied feature maps. The arrows denote the different operations. Figure taken from Olaf Ronneberger [52].

The U-Net architecture is composed of two parts: a contracting path and an expanding path (encoder, decoder). The contracting path is a series of convolutional and max pooling layers that reduce the spatial resolution of the input image, while the expanding path is a series of convolutional and transposed convolutional layers that increase the spatial resolution of the output. The two paths are connected by *skip connections*, which concatenate activations from the contracting path to the expanding path at corresponding spatial resolution levels. This architecture allows for extracting features as the input image is encoded/downsampled, and then creating a new image based on the features as it is decoded/upsampled.

The use of skip connections (seen as "copy and crop" in Figure 2.17) allows the U-Net to propagate information from shallower layers of the network to deeper layers, which helps the network to learn more complex and detailed features of the input image. This is particularly useful for tasks such as image segmentation, where it is important to preserve the spatial resolution of the input image and maintain the fine-grained details of the image. Skip connections can also help the network to learn more efficiently and to avoid the vanishing gradient problem. Skip connections provide an additional pathway for the gradients to flow back through the network, which can help to stabilize the training process and improve the network's ability to learn [61].

Another key feature of the U-Net is its ability to use relatively small convolutional filters, which helps to reduce the number of trainable parameters in the network and make it more efficient. This is particularly useful for tasks such as medical image analysis (section 3.2), where the amount of available training data is often limited.

By using U-Net in the GANs of the experiments, not only will we be able to extract important features of the otolith images efficiently, but also be able to generate new, transformed images as a result of the learned complex and detailed features of the input otolith images.

2.6.2 PatchGAN Discriminator

When using image-to-image translation with a U-Net generator, the discriminator can also be modified into a *PatchGAN* as presented by Isola et al. [38], which instead of deciding that its input image is real of fake for the whole image, it can instead decide that it is real or fake for different patches of that image. It compares two images, one from the source domain (the conditioned image which the generator also has as input in my case) and one from the target domain (either the generated image or a real image), and estimates the likelihood that the latter image is a fake reproduction of the former. The discriminator output is visualized in Figure 2.18.

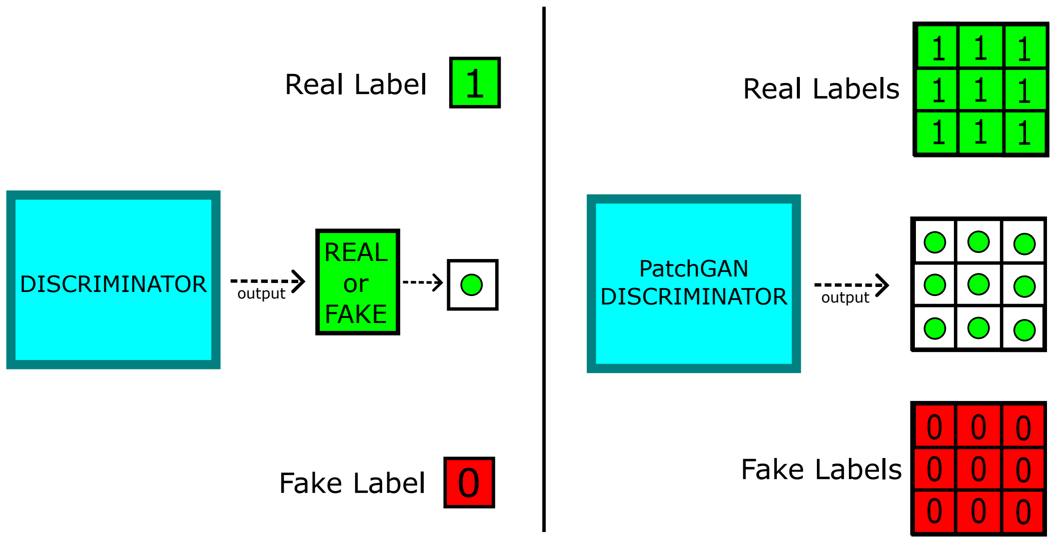


Figure 2.18: PatchGAN Discriminator. Standard discriminator to the left. PatchGAN discriminator to the right, where the discriminator output is a matrix of real/fake determinations for different patches of the input image instead of determining real/fake for the whole image.

The PatchGAN discriminator has several advantages over other types of discriminators. One advantage is that it can capture fine-grained details in the input image, which is important for tasks such as image synthesis where it is important to generate realistic-looking images. Another advantage is that it can handle input images of varying sizes, which is useful for tasks where the input images may have different dimensions.

To use a PatchGAN discriminator, the input image is first divided into a grid of overlapping patches. The patches are then processed independently by the discriminator network, which outputs a probability from its σ activation in the last layer for each patch indicating whether the patch is real or fake. These patch-level probabilities are then combined by averaging all the responses and to produce a final image-level probability that indicates whether the entire input image is real or fake [38].

PatchGAN discriminator uses an $n \times m$ output vector rather than a scalar output to provide probabilities of "real or fake" outcomes. The $n \times m$ size can vary depending on the size of the input image, but each output vector represents a region of the input image that is for example 70×70 pixels in size (not the entire input size), and this size is a fixed representation depending on how architecture is built. This results in the discriminator giving more feedback to the generator during training.

The Pix2Pix architecture with PatchGAN discriminator is one way of developing an image-to-image translation GAN. It was used in my main experiments of this thesis, but because of a few problems in the results which I will explain in chapter 4, another version of image-to-image translation architecture has been experimented with. The architecture is called *Wasserstein* GAN (**WGAN**), where instead of using a PatchGAN discriminator the way it is explained above, a Wasserstein *critic* is used that classifies the input image differently. This critic has been experimented both with a patch architecture and without. When using the patch architecture in the critic, the combined responses are averaged to the closest real number without the use of a σ activation function. The WGAN will be further explained in section 2.7.2.

2.7 Training Generative Adversarial Networks

As I introduced in section 2.2, the power of GAN lies in the competition between the generator and the discriminator that constantly tries to outwit one-another. The objective of a traditional GAN can be formalized as a two-player minimax game between the generator and the discriminator. The goal of the generator is to minimize the probability that the discriminator will correctly classify a generated sample as fake, while the goal of the discriminator is to maximize the probability of correctly classifying generated samples as fake.

Following Goodfellow et al. [31], we can define the framework as follows, where G and D represents the generator and the discriminator, respectively. To learn the generator's distribution p_g over data y, we define a prior on an input noise variables p_x, then represent a mapping to a data space as $G(x; \theta_g)$, where G is a differentiable function represented by a network with parameters θ_g. We also define a second network $D(y; \theta_d)$ that outputs a single scalar. $D(y)$ represents the probability that y came from the training dataset rather than p_g. We train D to maximize the probability of assigning the correct label to both training examples and samples from G. We simultaneously train G to minimize $\log(1 - D(G(x)))$. This GAN loss function can be expressed as the following optimization problem, where G and D plays the two-player min-max game with value function $V(D, G)$:

$$\min_G \max_D V(D,G) = \mathbb{E}_{y\sim p_{\text{data}}}[\log D(y)] + \mathbb{E}_{x\sim p_x}[\log(1 - D(G(x)))]. \tag{2.14}$$

Here, p_{data} and p_x are the distributions of the training data and the generator input, respectively. The input x is usually noise modeled as uniform or Gaussian, but in my case, x is an otolith image.

This min-max equation is modified if we are using a CGAN, where the only difference between them would be a conditional probability c used. The training process here is similar to that of a regular GAN, but with the additional input information, the condition, incorporated into both the generator and the discriminator networks. The generator is trained to generate samples that match the given condition, while the discriminator is trained to distinguish between the generated samples and real samples that match the condition. The objective function for a CGAN can be expressed as follows:

$$\min_G \max_D V(D,G) = \mathbb{E}_{y\sim p_{\text{data}}}[\log D(y|c)] + \mathbb{E}_{x\sim p_x}[\log(1 - D(G(x|c)))]. \tag{2.15}$$

Here, c represents the condition (e.g. the description of the object to be generated), and the other variables have the same meanings as in Equation 2.14.

As explained in subsection 2.2.4, the image-to-image CGAN architecture consists of a source image and a target image. In my experiments, the generator is not conditioned, and the source image x is an otolith image while the target image is y. The otolith image is also used as the condition for the discriminator. In the context of this thesis' experiments, the function of the CGAN can be expressed as follows:

$$\min_G \max_D V(D,G) = \mathbb{E}_{x\sim p_{\text{data}}}[\mathbb{E}_{y\sim p_{\text{data}}}[\log D(y|x)] + \log(1 - D(G(x)))], \tag{2.16}$$

where the discriminator is conditioned on the same image x as the input image of the generator. Note that both x and y are now from the distribution p_{data} because both the target images and the otolith images comes from the same training dataset.

The training of the model happens in two steps, where either the generator or the discriminator is fixed while the other performs a gradient step, and then vice-versa. This is described in the following way using Equation 2.16 [67]:

Step 1: Fix, or freeze, the generator and perform a gradient step to maximize the probability of assigning the correct label to both training examples and samples from the generator, with respect to the discriminator:

$$\mathbb{E}_{x\sim p_{\text{data}}}[\mathbb{E}_{y\sim p_{\text{data}}}[\log D(y|x)] + \log(1 - D(G(x)))]. \tag{2.17}$$

Step 2: Fix the discriminator and perform a gradient step to:

(in theory)

$$\min_G \mathbb{E}_{x\sim p_{\text{data}}}[\log(1 - D(G(x)))], \tag{2.18}$$

(in practice)

$$\max_G \mathbb{E}_{x\sim p_{\text{data}}}[\log D(G(x))]. \tag{2.19}$$

Goodfellow et al. [31] states that in practice, Equation 2.14 may not provide sufficient gradients for G to learn well. This also applies to Equation 2.15 and Equation 2.16. Early in learning, when G is poor, D can reject samples with high confidence because they are clearly different from the training data. In this case, $\log(1 - D(G(x)))$ saturates. Rather than training G to minimize $\log(1 - D(G(x)))$, we can train G to maximize $\log D(G(x))$. This objective function results in the same fixed point of the dynamics of G and D but provides much stronger gradients early in learning. The training process of traditional GANs is visualized in Figure 2.19.

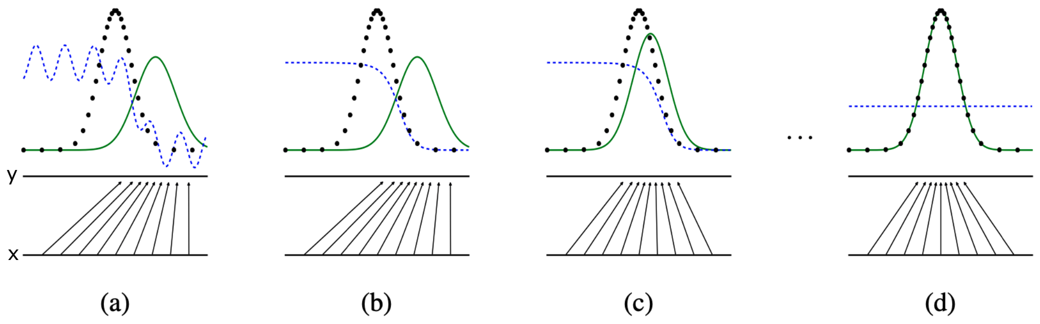


Figure 2.19: The process of training traditional GANs. Black dotted line denoting the data distribution p_{data}, green solid line denoting generative distribution (data made from the generator) p_g, blue dashed line denoting discriminative distribution (outputs from the discriminator) $D(y)$. The lower horizontal line is the domain from which x is sampled. The horizontal line above is part of the domain of y. The upward arrows show how the mapping $y = G(x)$ imposes the non-uniform distribution p_g on transformed samples. Figures **(a)** through **(d)** showing the process of training in the beginning towards the end. Figure taken from Goodfellow et al. [31] and edited.

As seen in Figure 2.19, the goal is to make p_{data} and p_g equal. This figure by Goodfellow et al. [31] shows an example of an adversarial pair near convergence, where p_x is similar to p_{data} in Figure 2.19a, and D is a partially accurate classifier. In Figure 2.19b, D is further trained to discriminate samples from the data, converging to:

$$D^*(y) = \frac{p_{\text{data}}(y)}{p_{\text{data}}(y) + p_g(x)}. \tag{2.20}$$

In Figure 2.19c, when the G has been updated, the gradients of D has guided $G(x)$ to point to regions that are more likely to be classified as real data (as seen by the arrows). In Figure 2.19d, after many steps of training, if both the G and the D have enough capacity, they will reach the goal of $p_{\text{data}} = p_g$, where both cannot improve any further. The discriminator is unable to differentiate between the two distributions, i.e. $D(y) = 0.5$. Goodfellow et al. describes this in further detail in reference [31].

2.7.1 Challenges when Training GANs

Many GAN models have a few major problems they suffer from during training. A few examples of these are presented below:

- **Unstable Convergence or Non-Convergence:** The parameters of the model may oscillate when training so that they destabilize and never converge. This can make it challenging to tune the network hyperparameters and achieve good performance.
- **Mode Collapse:** Where the model fails to generalize properly, missing entire modes (variety of samples) from the input data, rather than producing a diverse range of outputs. This can lead to the generated samples being less realistic and less useful. This may occur, for example, if the generator learns more quickly than the discriminator. The best generator would only produce elements of Equation 2.19 if the discriminator D were held constant. Therefore, for instance, if we are training a GAN with the *MNIST Dataset*[1] [85], the discriminator somehow favors the digit 0 slightly more than other digits, the generator may take advantage of the opportunity to generate only the digit 0, then be unable to escape the local minimum after the discriminator improves.
- **Vanishing Gradient:** When using GANs, the vanishing gradient problem can occur in both the generator and the discriminator networks. In the generator network, the vanishing gradient problem can prevent the network from learning to generate high-quality images, as the gradients may be too small to effectively update the weights and improve the output. In the discriminator network, the vanishing gradient problem can prevent the network from effectively learning to distinguish real and fake images, leading to poor performance.

  In contrast to the last bullet point, if the discriminator learns information faster than the generator, it can be able to distinguish fakes perfectly. In this scenario, the generator gradients would be very close to zero because the generator might be stuck with a very high loss no matter which way it changes its parameters. This is an instance of the vanishing gradient problem where the generator is unable to learn [82].
- **Difficulty in Evaluating Performance:** GANs are often evaluated using subjective metrics, such as asking human evaluators to compare the generated samples to real samples. Using the loss as the sole metric performance often is not accurate. This can make it difficult to compare different GANs and to know how well a GAN is performing.
- **Resource-heavy Procedure:** In general, when working with CNNs for image classification and generation, training requires lots of resources and often takes

[1]A large database of handwritten digits that is commonly used for training various image processing systems

several hours to finish depending on hardware capabilities. Training on a computer with just a CPU would make the training phase last even longer. Therefore, training GANs on GPU's is highly recommended – mitigating the time used in this phase.

2.7.2 Alternative GAN Loss Functions

In the beginning of section 2.7, I introduced the training procedure of the traditional GAN and its standard min-max loss function in Equation 2.14. In the experiments of this thesis, alternative GAN loss functions has been used. These are presented below.

Pix2Pix Loss

In section 2.6, I presented the Pix2Pix GAN, where the generator is a U-Net and the discriminator is a PatchGAN. The loss function of Pix2Pix is defined as follows, using a source image x, target image y and condition c:

$$\mathcal{L}_{\text{CGAN}}(G, D) = \mathbb{E}_{y,c}[\log D(y, c)] + \mathbb{E}_{y,x,c}[\log(1 - D(y, G(c, x)))]. \tag{2.21}$$

In my case, the generator is not conditioned, and the discriminator is conditioned on the source image:

$$\mathcal{L}_{\text{CGAN}}(G, D) = \mathbb{E}_{y,x}[\log D(y, x) + \log(1 - D(y, G(x)))]. \tag{2.22}$$

The loss in standard min-max GAN and CGAN is essentially an adversarial loss of *Binary Cross-Entropy* (**BCE**) [29]. In Pix2Pix, the generator is also mixed with L1 loss, also known as ℓ^1 penalty, defined as follows in my case of using x as source image and y as target image [6]:

$$\mathcal{L}_{\ell^1}(G) = \mathbb{E}_{x,y}[\|y - G(x)\|_1]. \tag{2.23}$$

ℓ^1 norm (absolute error), a regularization technique (subsection 2.8.1), is used to minimize the generator's error. It measures the distance between the generated images and the target images, telling how similar the generated images are to the target images. The generator's BCE loss function when additionally combined with an ℓ^1 penalty enables the generator to create images that are not only close to the truth, but also trick the discriminator by allowing it to produce structurally similar images to the target images [66].

This combined with Equation 2.22, the final loss function for the generator is given as [6]:

$$G^* = \arg\min_G \max_D \mathcal{L}_{\text{CGAN}}(G, D) + \lambda \mathcal{L}_{\ell^1}(G), \tag{2.24}$$

where $\mathcal{L}_{\text{CGAN}}(G, D)$ is Equation 2.22 and λ denoting the hyperparameter controlling the ℓ^1 penalty.

Wasserstein Loss

To tackle the problem of mode collapse mentioned in subsection 2.7.1, another loss function has also been tested in the experiments. It is called the Wasserstein loss (Wasserstein distance), presented by Arjovsky, Chintala, and Bottou in the paper *Wasserstein GAN* [3]. The Wasserstein distance, also known as the Earth Mover's distance [81], is a measure of the distance between two probability distributions. It is defined as the minimum amount of work needed to transform one distribution into the other, where the work is defined as the amount of mass moved multiplied by the distance it is moved. In WGAN, the generator is trained to minimize the Wasserstein distance between the distribution of the generated samples and the distribution of the real samples, while the discriminator is trained to maximize this distance.

In this setting, the discriminator is often called the *critic*, where instead of directly discriminating its inputs to tell whether an image is real or fake, the critic instead scores the *realness* or the *fakeness* of its inputs by outputting a real number. This change can result in more stable training with less sensitivity to model architecture and configurations of hyperparameters [10]. The loss relates more on the *quality* of the generated images of the generator.

The use of the Wasserstein loss has several advantages over other loss functions used in GANs. Unlike the cross-entropy loss, the Wasserstein loss is differentiable with respect to the generator's inputs. This can improve the stability and convergence of GAN training, and has been shown to produce better results in some tasks [1]. The Wasserstein loss is more sensitive to the mode collapse problem, resulting in better variety in sample generation which we will see in the results of the experiments (chapter 5).

In WGANs, there is a requirement called the Lipschitz continuity or $1-$Lipschitz constraint [84]. It refers to a property of a function which states that the function's output changes by at most a fixed amount (the Lipschitz constant) for a given change in the input. A function that is Lipschitz continuous with a Lipschitz constant of 1 is called a $1-$Lipschitz function. The Wasserstein loss is formulated as follows, where f_w is the $1-$Lipschitz function and w the weights:

$$\mathcal{L}_{\text{WGAN}} = \max_{w \in \mathcal{W}} \mathbb{E}_{y \sim p_{\text{data}}}[f_w(y)] - \mathbb{E}_{x \sim p_x}[f_w(G(x))]. \tag{2.25}$$

The first part of the equation above represents the real data of the network, while the second part represents the data generated by the generator. Because it wants the generated data to be as accurate as possible, the generator network seeks to reduce the gap between real data and generated data. The max value refers to the constraint on the critic. Compared to the GANs discussed earlier, the critic of the Wasserstein GAN does not have a sigmoid layer that limits the values between 0 and 1. Instead, it returns a real value, a score, that allows it to act less *strictly*, hence the name critic. The Wasserstein loss in my case, using the same f_w as described above, also being conditional, is formulated as follows:

$$\mathcal{L}_{\text{WCGAN}} = \mathbb{E}_{x \sim p_{\text{data}}}[\max_{w \in \mathcal{W}} \mathbb{E}_{y \sim p_{\text{data}}}[f_w(y|x)] - f_w(G(x))]. \tag{2.26}$$

Wasserstein Loss with Gradient Penalty

The Wasserstein loss may be able to improve training stability, but there are cases where it results in poor sample generation or fails to converge, or result in vanishing gradients as mentioned in subsection 2.7.1. To tackle these problems, *gradient penalty* can be applied. Wasserstein loss with gradient penalty (**WGAN-GP**) can help to improve the stability and convergence of the GAN during training [33].

As stated earlier, the critic network in a WGAN is required to be a $1-$Lipschitz function in order to ensure the stability of the training process. This constraint can be imposed by using a gradient penalty term in the loss function for the critic. The gradient penalty term is calculated as the mean squared norm of the gradients of the critic output with respect to the input, multiplied by a weighting factor. This term is added to the loss function in order to penalize the critic for violating the Lipschitz constraint. This encourages the critic to be a $1-$Lipschitz function and prevent the training process from becoming unstable, which has been shown to produce better results in some tasks [3, 35]. The Wasserstein loss with gradient penalty can be formulated as:

$$\mathcal{L}_{\text{WCGAN-GP}} = \mathcal{L}_{\text{WCGAN}} + \lambda \mathbb{E}_{y,x}[(||\nabla_{y,x} f(y,x)||_2 - 1)^2], \tag{2.27}$$

where $\mathcal{L}_{\text{WCGAN}}$ is Equation 2.26 and λ is the penalty coefficient that weight the gradient penalty term.

2.8 Overfitting vs. Underfitting

In deep learning, we aim to build a model that have both good accuracy and flexibility. The goal is that it can generalize well, meaning by not either *overfitting* or *underfitting* to the data. Visualization in Figure 2.20 for easier understanding.

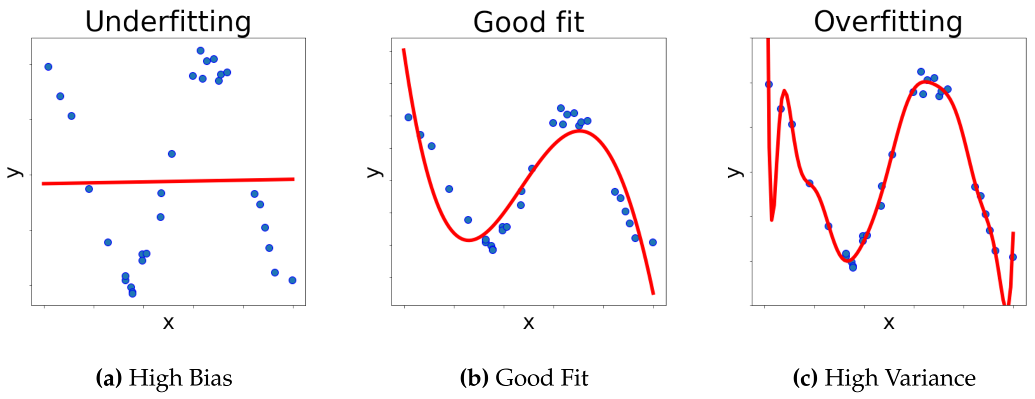


Figure 2.20: Bias-Variance Trade-off. Blue dots representing samples. Red line representing the fitness of the model. In Figure 2.20a, we have an example of a model that has high bias, resulting on underfitting. In Figure 2.20c, we have an example of a model that has high variance, resulting in overfitting.

By generalizing well, it can result a fitness example as Figure 2.20b. Introducing high *bias*[2] however, will result in a model that pays little attention to *training* data and oversimplifies the model by underfitting, as seen in Figure 2.20a. In my GAN case, this can result in vanishing gradients for the generator as described in subsection 2.7.1, and can also result in mode collapse.

Introducing high *variance*[3] results in a model that pays too much attention to

[2]The difference between the average prediction of our model and its target values.
[3]The variability of model prediction due to changes in training data.

training data and overfits to it, therefore cannot perform well on *testing* data, seen in Figure 2.20c. In my GAN case, this can result in randomly generated images that only resembles the samples from the training set.

2.8.1 Regularization

The task of training deep neural networks can be slow and prone to overfitting. As a result, research in deep learning is constantly looking for ways to solve these issues. According to Kukačka, Golkov, and Cremers [44], regularization is any additional technique with the aim of enhancing the model's generalization performance. In other words, when a deep learning model is presented with brand-new data from the problem domain, regularization is a set of strategies that can prevent overfitting in neural networks and hence increase its performance.

Batch Normalization

One of these techniques is batch normalization, also known as **BatchNorm**, which is used in the Pix2Pix experiments. In the field of deep learning, it is currently a widely used method. It can accelerate neural network learning and prevent overfitting by decreasing what is known as the *internal covariate shift* [36]. This refers to the change in the distribution of the inputs to a layer of a neural network as the parameters of the network are updated during training. The use of BatchNorm can mitigate internal covariate shift by normalizing the inputs to a layer using the statistics of the current mini-batch, rather than the entire training dataset, and enables the network to shift and adjust the distribution thanks to the two learnable parameters, γ and β. This helps to keep the inputs to a layer more stable, formulated as follows [56]:

$$
\begin{aligned}
\mu &= \frac{1}{b}\sum_{i=1}^{b} x^{(i)}, \\
s^2 &= \frac{1}{b}\sum_{i=1}^{b} (x^{(i)} - \mu)^2, \\
x_{\text{new}}^{(i)} &= \frac{x^{(i)} - \mu}{\sqrt{s^2 + \epsilon}}, \\
z^{(i)} &= \gamma x_{\text{new}}^{(i)} + \beta,
\end{aligned}
\tag{2.28}
$$

where μ is the mean over the mini-batch that has b training instances x, s is an estimate of standard deviation, $x_{new}^{(i)}$ is the zero-centered and normalized input $x^{(i)}$, ϵ is a small number to avoid division by zero, and γ and β are the scaling and shifting parameters done on the batch normalized activations $z^{(i)}$, respectively.

Equation 2.28, which in practice is done element-wise, is typically applied to the inputs of a layer, before the activation function is applied. The idea is to standardize the inputs to have a mean of 0 and a standard deviation of 1, which can help stabilize the training process and improve the overall performance of the network. This is done by computing the mean and standard deviation of the inputs for each mini-batch during training, and using those values to normalize the inputs before they are passed through the layer. During inference, the mean and standard deviation of the entire training dataset is used.

Instance Normalization

When using BatchNorm, a large mini-batch size is recommended, being greater than one. When using smaller batch sizes, an introduction of errors may occur in the training because the mean and variance estimates can become noisy. Therefore, when using smaller batch sizes, *instance normalization* (**InstanceNorm**) is recommended and can perform better.

In contrast to BatchNorm, InstanceNorm normalizes the inputs to a layer of a neural network for each individual sample, i.e. it calculates the mean and standard deviation of the input values for each sample separately, and normalizes the inputs using these statistics. This allows the network to learn more robustly by normalizing the inputs for each sample individually, rather than using statistics from the entire batch. This can be particularly useful when the samples in a batch have different distributions or scales. In my Wasserstein GAN experiments, all the BatchNorm layers are replaced with InstanceNorm for the purpose of experimenting with different types of regularization techniques.

Dropout

Another popular regularization technique used in the experiments, is dropout, which Srivastava et al. first proposed in 2014 [70]. Dropout provides regularization by temporarily omitting a certain number of neurons for a particular layer during training, which adds noise to the training process. Dropout is visualized in Figure 2.21.

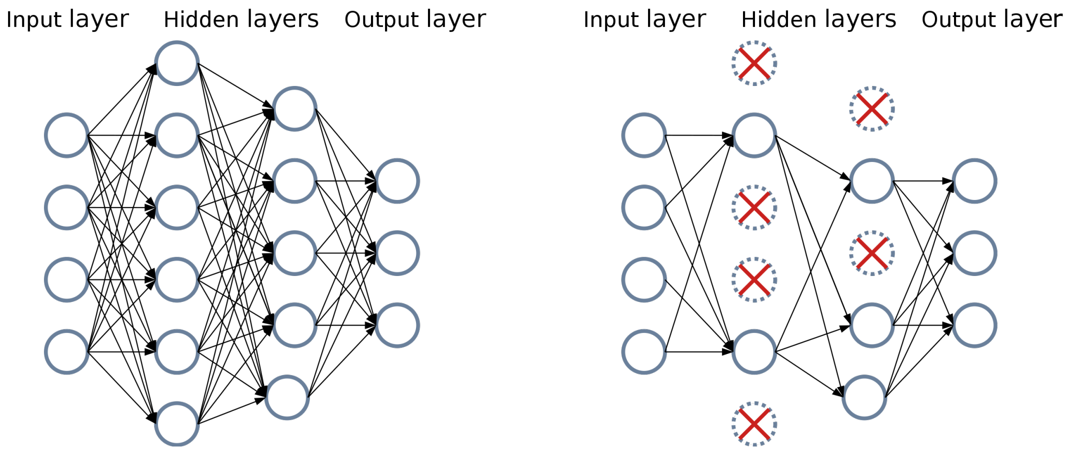


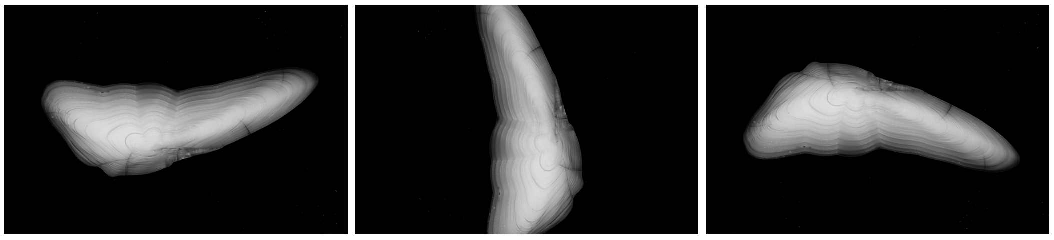
Figure 2.21: Dropout, a regularization technique, where certain numbers of neurons in layers are omitted during training. This adds noise to the training process and can reduce overfitting. Figure taken from Dyrmann [22].

In each iteration, a group of neurons will be dropped, determined by a probability called the dropout rate. As a result, neurons will affect the output differently for each iteration. The capacity of the model will decrease through random sub-sampling of layer outputs, thereby lowering overfitting. In addition, dropout forces the network to develop redundancy so that the workload of the network does not only rest on a few neurons. To ensure accurate and reliable results during testing, however, all neurons are kept.

Data Augmentation

Another regularization technique is data augmentation. This regularization technique is directly applied to the training dataset by applying *transformations* to the data, being images in my case. Often when dealing with smaller datasets, data augmentation is used to expand the amount of samples of the data, resulting in more data being available for training. More data is acquired by applying these transformations to the images because we can use both the original images in addition to the transformed images. This results in another advantage of the technique, which is that it can help to reduce overfitting and improve the performance of the model by providing it with more diverse and representative data to learn from. Data augmentation can assist the model in learning more reliable and generalizable features which can help to increase the model's ability to generalize to new data, which can be especially helpful when the original dataset is small.

Many methods of image transformations are available – meaning whatever edits done to one image sample will expand the number of samples. Two main transformation techniques are used in my experiments: rotation and vertical flipping, where each maps every pixel of the images into a new representation. In addition, cropping is also done in some cases. Each of the transformation techniques used are visualized in Figure 2.22.



(a) Original Image. **(b)** Rotation. **(c)** Vertical axis flipping.

Figure 2.22: Transformation techniques done to original image **(a)**. Image **(b)** is rotating the image either −90 or 90 degrees. Image **(c)** is flipping the image on its vertical axis.

2.8.2 Training-Validation-Testing Split

The lowest generalization error is what we aim for when working with a machine learning model. This refers to the ability that the models performs well on previously seen data, but also on *new*, unforeseen data. It is typical to divide the data into three portions, the training, validation, and test datasets, and to measure the models' performance on each of them.

Let us say the data consists of for example 10000 images. After we have prepared and processed our images, we must split them into the sets described above. In this example case, 8000 images will be used for the training set. And the last 2000 images will be divided into 1000 images for validation, and another 1000 images for testing; resulting in a split of 80% - 10% - 10%. These sets cannot overlap each other, otherwise when testing our model, the data will be biased.

The training dataset is utilized, as the name implies, when the model is being trained. The purpose of this dataset is to optimize the parameters of the model such that it can learn and solve the task at hand. The validation dataset, on the other hand, is employed for the purpose of providing an unbiased assessment of the model's performance. This can be done either during the training phase, or after the training phase has been completed. The validation dataset can also be used to direct the training process by tuning the hyperparameters of the model. Finally, the test dataset, which is a set of never-before seen data, is employed when the model has finished training and all the hyperparameter tuning has been completed. The purpose of using this dataset is to obtain an unbiased estimation of the generalization error of the trained model [30], in my case being the trained GAN.

Chapter 3

Related Works

This section presents a few relevant related works. We will look at applications of deep learning on fish age determination, and GANs that are used for annotations and segmentations of images.

3.1 Deep Learning Applications on Fish Age Determination

As presented in subsection 1.3.1, the current technology sought to further expand upon in this thesis, is DeepOtolith, who Ketil Malde that provided the topic of this thesis has co-worked on [58]. DeepOtolith use otolith photos and CNNs to automatically estimate fish ages. It currently contains classifiers for three fish species, Greenland halibut (Reinhardtius hippoglossoides), Atlantic salmon (Salmo salar) and Greek red mullet (Mullus barbatus). Images of these fish species are used as input in the program, where the output is the age prediction. Different CNN architectures has been used to predict age for the species, and the accuracies that was obtained varies. For predicting age on Greenland Halibut, a VGG19 CNN model was used. The trained model attained a root mean square error of 1.69 years between age prediction and age read by experts. For the Atlantic salmon, an implementation of EfficientNetB4 CNN was utilized. The prediction of sea age obtained an accuracy of 86.99%, while the predictive accuracy of river age was 63.2%. For the Greek Red Mullet, the Inception v3 CNN model was implemented. The ages predicted correctly was 64.4%.

Politikos et al. has also investigated the feasibility of using deep learning to provide an automatic estimation of fish age from otolith images in their paper "Automating fish age estimation combining otolith images and deep learning: The role of multitask learning" [57]. On top of using CNN models, they also proposed an enhanced multi-task learning[1] network to better estimate fish age. According to the paper, the results showed that the network without multi-task learning predicted fish age correctly by 64.4%, attaining high performance for younger age groups (ages between 0 and 1). The network with multi-task learning increased the correctness reaching 69.2% and also proved efficient for older age groups (ages between 2 to 5+) [57].

Deep learning-based computer vision algorithms have shown promising results for age determination applications [54, 50], but despite the increased attention for these applications, the utilization of deep learning in this field is still relatively untouched. Especially when it comes to solutions that are applicable as explainable AI's.

[1]Multi-task learning is a subfield of machine learning in which multiple learning tasks are solved at the same time, while exploiting commonalities and differences across tasks. This can result in improved learning efficiency and prediction accuracy for the task-specific models, when compared to training the models separately [86].

3.2 GAN Applications for Annotation and Segmentation

In the field of fish age determination, the use of GANs has not been an approach towards a solution. But there are a few research studies conducted where GANs has been implemented for image segmentations and annotations, especially in the medical field.

Dimitrakopoulos, Sfikas, and Nikou has in the paper “ISING-GAN: Annotated Data Augmentation with a Spatially Constrained Generative Adversarial Network” [18] utilized Ising-GAN for data augmentation, oriented towards use with medical imaging sets where a localization/segmentation annotation is available. Their model can produce artificial annotations where tuple generation of synthetic images and corresponding segmentation masks localizes an object, tissue or organ of interest. Dimitrakopoulos, Sfikas, and Nikou has compared standard GAN with ResGAN, Ising-GAN and Ising-ResGAN for their experiments. Their results have shown that the Ising model-based GANs have the best performance. The Ising-based smoothing term forces the synthetic annotations and their synthesized image counterpart to be visually coherent, with numerical and qualitative results validating the usefulness of the model.

In the paper *HistoStarGAN: A Unified Approach to Stain Normalisation, Stain Transfer and Stain Invariant Segmentation in Renal Histopathology* by Vasiljević et al. [78], an approach towards synthetic generation of virtual stain transfer has been experimented with in the area of Computational Pathology. They have proposed a HistoStarGAN model as a framework that performs stain transfer between multiple staining, stain normalization and stain invariant segmentation. Their HistoStarGAN model builds upon StarGANv2 in addition to CycleGANs. The results vary in their experiments, showing promising performance results when compared to UDA-GAN which also uses CycleGAN models for data augmentation. The paper states that "To illustrate the capabilities of our approach, as well as the potential risks in the microscopy domain, inspired by applications in natural images, we generated KidneyArtPathology, a fully annotated artificial image dataset for renal pathology."

Wu, Zou, and Yang has in the paper “U-GAN: Generative Adversarial Networks with U-Net for Retinal Vessel Segmentation” [90] researched the use of U-Net generator and several discriminators, one being PatchGAN, for their automatic detection of retinal diseases. Their use of GAN presents a method that generates a precise map of retinal blood vessels, with results achieving segmentation accuracy of 96.15%.

Many of these works would have been interesting to apply in regard to my research question. But due to the lack of time and access to available hardware, I did not have the opportunity to experiment with these solutions.

Chapter 4

Materials and Methods

This chapter describes the steps taken to answer the research question in section 1.4. In summary, this chapter will first examine the data itself as well as the instruments and techniques used to prepare it for the training of GANs, and then move on to further describe the experiments.

4.1 Otolith Extraction, Image and Data Acquisition Process

All images of otoliths used in this thesis, if not referenced otherwise, was personally provided by Dr. Côme Denechaud, Research Scientist in the Demersal Fish Research Group at Havforskningsinstituttet (Norway). These images, in addition to their respective data of annuli coordinates, were used for training and testing in my envisioned solution presented in subsection 1.3.2. The dataset was originally developed for growth chronology analysis in the paper "A century of fish growth in relation to climate change, population dynamics and exploitation" by Denechaud et al. [16]. The processing of the otoliths, done by Dr. Côme Denechaud, are described below.

4.1.1 Step 1 of the process:

The otoliths were collected from an archive where they were stored in individual dry envelopes. They were then embedded in black epoxy into a tray fitted to a cutting machine, with their core (identified as a groove on the surface) aligned along the cutting axis as shown in Figure 4.1. Then a second layer of epoxy covers the otoliths before cutting, hence why they need to be aligned beforehand.

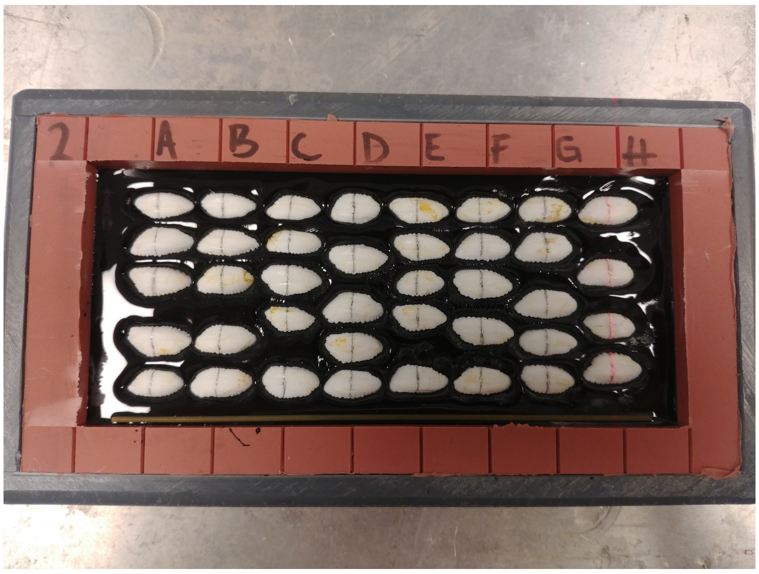
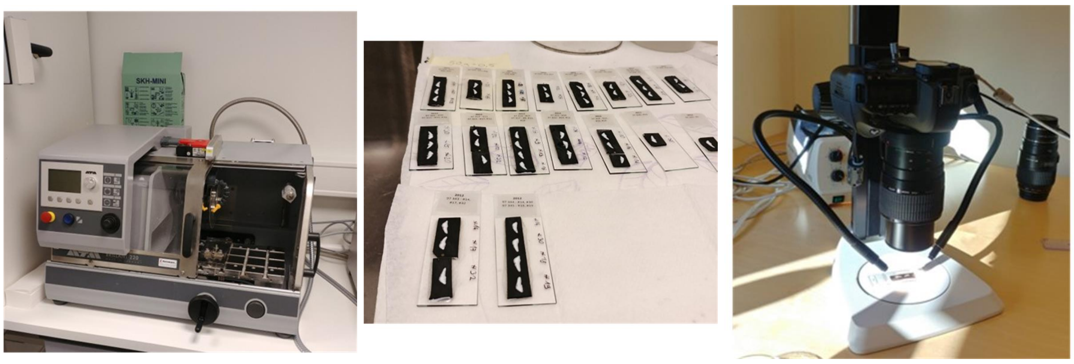


Figure 4.1: First step of processing otoliths, where they were aligned and embedded in black epoxy into a tray fitted for a cutting machine.

4.1.2 Step 2, 3 and 4 of the process

The machine cuts thin sections of about 1mm thickness using a double diamond circular blade following the tray lines under which the otoliths were embedded as shown in Figure 4.2a. These thin sections were then cleaned and stuck to a microscope glass slide using a drop of clear epoxy, leaving the surface untouched in case of future sampling of material as shown in Figure 4.2b. The images were acquired using high-resolution setup composed of a high-end Canon DSLR camera fitted with a macro lens on a vertical support, and externally adjusted lighting as shown in Figure 4.2c.

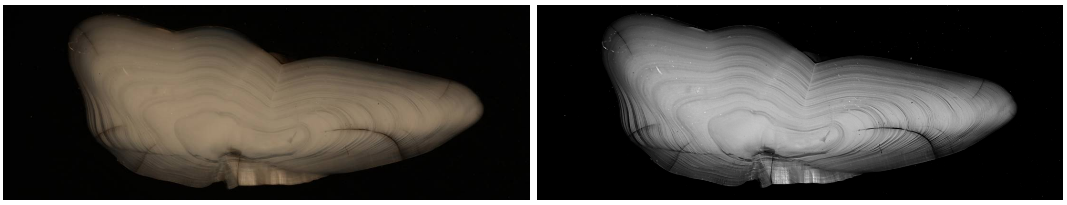


(a) Cutting machine. **(b)** Otoliths after cutting. **(c)** Camera setup.

Figure 4.2: Step 2, 3 and 4 of how the otoliths were processed and taken photographs of. **(a)** showing the machine that cuts the processed otoliths into thin sections. **(b)** showing the cut and cleaned sections stuck into microscope glass. **(c)** showing how the camera is set up for taking images of the otoliths.

4.1.3 Step 5 of the process

The RAW images were then enhanced using a set of Photoshop macros aimed at facilitating interpretation by removing color artifacts (conversion to grayscale) and enhancing contrast and brightness to better differentiate the winter and summer growth rings (subsection 1.2.1). Note that this step was applied for easier human interpretation of the images, and may not be necessary in the context of using ANNs where the model may be able to differentiate the features of the images regardless.



(a) RAW image of the photographed otolith. **(b)** Edited image of the photographed otolith.

Figure 4.3: Step 5 of the process, showing the RAW image of the photographed otoliths in **(a)**, and the edited and enhanced image of that otolith in **(b)**.

4.1.4 Step 6 of the process

Each image (in total of 4096) was then manually inspected by Dr. Côme Denechaud, where each otolith and their respective coordinates for annuli locations in addition to other information were noted in a table as a *CSV* file, shown in Table 4.1.

Sample	Quality	Age	Cohort	Core	I	Year	Increment	X	Y
N-1953_51	1	10	1943	2274.5	core			2354	2172
N-1953_51	1	10	1943	2274.5	1	1944.0	259.1	2395	2045
N-1953_51	1	10	1943	2274.5	2	1945.0	262.1	2435	1916
N-1953_51	1	10	1943	2274.5	3	1946.0	146.3	2458	1845
N-1953_51	1	10	1943	2274.5	4	1947.0	235.1	2495	1730
N-1953_51	1	10	1943	2274.5	5	1948.0	347.0	2549	1560
N-1953_51	1	10	1943	2274.5	6	1949.0	182.8	2577	1470
N-1953_51	1	10	1943	2274.5	7	1950.0	208.8	2609	1368
N-1953_51	1	10	1943	2274.5	8	1951.0	142.7	2632	1298
N-1953_51	1	10	1943	2274.5	9	1952.0	223.5	2666	1188
N-1953_51	1	10	1943	2274.5	10	1953.0	170.1	2693	1105
N-1953_51	1	10	1943	2274.5	edge		68.3	2704	1072
N-1953_52	1	10	1943	1804.2	core			2114	2126
N-1953_52	1	10	1943	1804.2	1	1944.0	276.3	2135	1986
N-1953_52	1	10	1943	1804.2	2	1945.0	569.5	2177	1696
N-1953_52	1	10	1943	1804.2	3	1946.0	405.0	2207	1490
N-1953_52	1	10	1943	1804.2	4	1947.0	238.7	2225	1369
N-1953_52	1	10	1943	1804.2	5	1948.0	220.9	2242	1256
N-1953_52	1	10	1943	1804.2	6	1949.0	192.4	2256	1159
N-1953_52	1	10	1943	1804.2	7	1950.0	152.2	2268	1081
N-1953_52	1	10	1943	1804.2	8	1951.0	198.1	2282	980
N-1953_52	1	10	1943	1804.2	9	1952.0	151.9	2294	903
N-1953_52	1	10	1943	1804.2	10	1953.0	130.4	2303	837
N-1953_52	1	10	1943	1804.2	edge		60.8	2308	806
...	...	...	...	...	...	...	...	...	...

Table 4.1: Table of annotations data of processed otoliths.

The X and Y denotes the pixel coordinates in the image on where the annuli for the samples are. Each image has a resolution of 5616×3744, in addition to a few images with resolution of 5760 × 3600. The different ages and their occurrences are shown in Table 4.2.

Age	8	9	10	11	7	12	13	14	18	16	19	21	**Total:**
Occurrences	3023	507	281	136	85	42	11	5	2	1	1	1	**4096**

Table 4.2: The ages of the fish in the dataset and their occurrences. Most of the fish in the dataset are 8 years old; only a few has reached ages past 13.

4.2 Further Data Preprocessing

In order for the annotations data and images to be applicable in the machine learning experiments, further preprocessing and data preparation was applied.

4.2.1 Annotations Data Feature Extraction

The features extracted from the original data were the sample names (for each image file), ages and annuli coordinates. These were extracted into a dataset as shown in Table 4.3.

Sample	**Age**	**X0**	**Y0**	**Z0**	**X1**	**Y1**	**Z1**	**X2**	**Y2**	**Z2**	**...**
N-1953_51	10	2395	2045	1	2435	1916	1	2458	1845	1	...
N-1953_52	10	2135	1986	1	2177	1696	1	2207	1490	1	...
N-1953_53	10	2165	2056	1	2215	1907	1	2255	1787	1	...
N-1953_01	8	3229	1717	1	3156	1551	1	3110	1444	1	...
N-1953_02	8	2894	1916	1	2817	1802	1	2770	1733	1	...
N-1953_03	8	2858	1887	1	2872	1750	1	2886	1616	1	...
N-1953_04	8	2860	1703	1	2864	1556	1	2867	1456	1	...
N-1953_05	8	3121	1920	1	3061	1819	1	2999	1714	1	...
N-1953_06	8	3021	2092	1	2973	1964	1	2928	1840	1	...
...	...	...	...	...	...	...	...	...	...	...	...

Table 4.3: Extracted features in a prepared dataset. X's denoting the x-coordinates of annuli and Y's denoting the y-coordinates of annuli for each sample. Z's denoting whether there is a coordinate/annuli present or not (True = 1, or False = 0).

In order for the GAN to learn effectively, it only required the coordinates' data of the annuli, which allowed the GAN to learn where the annuli are located in the images. The rest of the data in the prepared dataset such as the image sample names and the age of the fish were only used in further preprocessing, such as allowing each image to be loaded from the disk and age for visualization purposes.

4.2.2 Image Preprocessing

An example of an otolith image from the original dataset is shown in Figure 4.4.

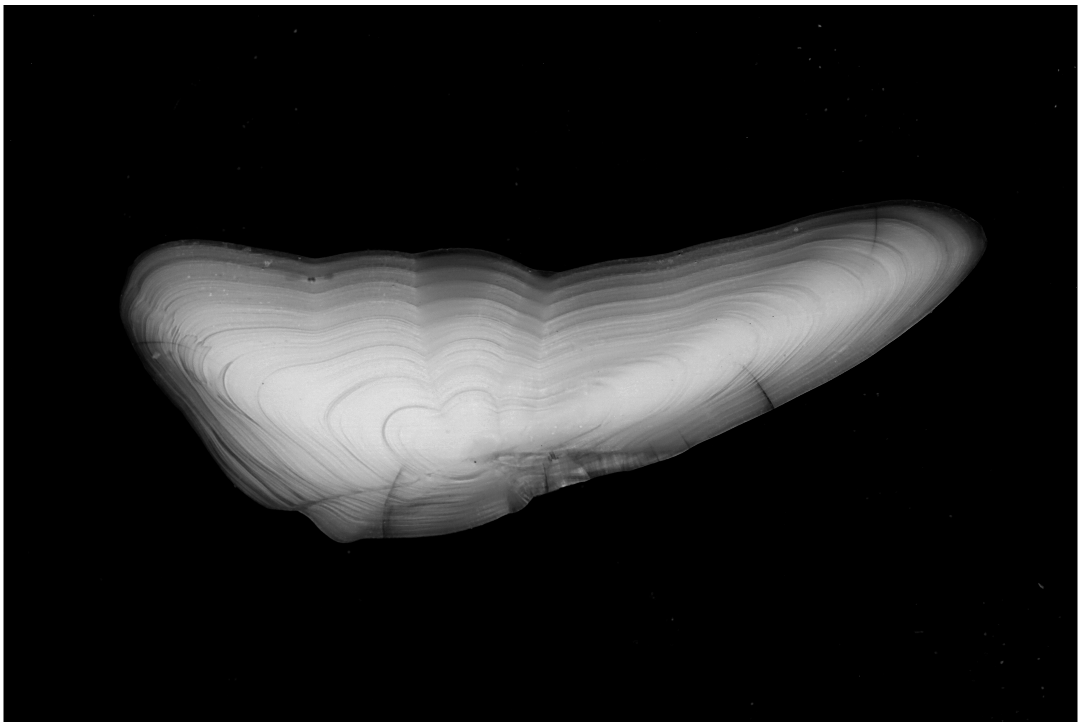


Figure 4.4: An example of an otolith image from the original dataset.

Many preprocessing steps were done to the images provided for the thesis to reduce memory demands and increase performance of the GANs. These steps are described below:

1. The entire dataset of 4096 images had a total size of 11.91GB on disk, with each image averaging around 3.3MB in size. In order to save memory and improve training speeds, each image was compressed using the Python library Pillow [88], reducing the total size of the dataset to 1.53GB on disk, with each image averaging around 348KB in size. The compression did reduce some sharpness and brightness of the images, but overall did not remove important details. Due to lack of time and hardware, model performance differences were not tested on non-compressed images.

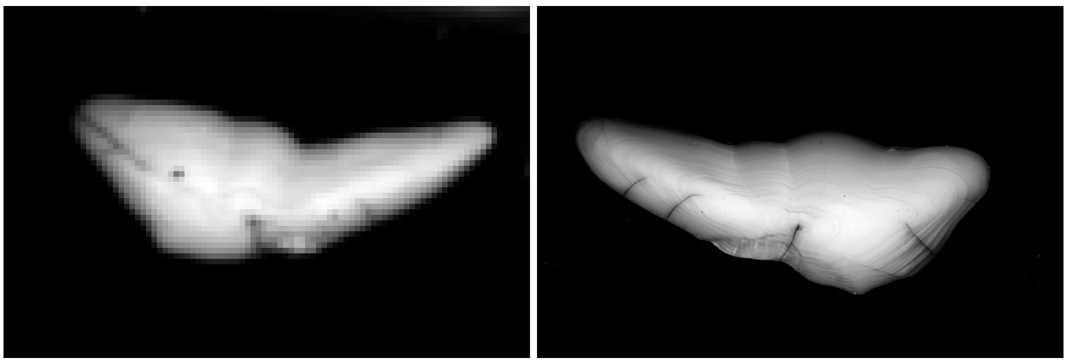
2. Most of the images in the dataset had already been converted into grayscale, but there were a few exceptions. To ensure that all the images were grayscale, I converted any remaining RGB images (3 channels) into grayscale (1 channel).

3. As previously mentioned, every image in the dataset had dimensions of 5616×3744 with a few samples of 5760×3600. These large images are demanding for CNNs and require significant computational resources to process. Due to the lack of available resources, two resized versions of the dataset were created. One "smaller" dataset consisting of images that was downscaled to dimensions of 64×96, which were used for initial model development and testing. And one "bigger" dataset consisting of images that was downscaled to dimensions of 512×768, which were used for final model evaluations. A few experiments were also done where the images from the bigger dataset also were square-cropped to the sizes 512×512 and 128×128. The 128×128 crops were mostly centered around where the real annuli were located in the dataset, but the crops did in some cases also crop a few annuli from the images. This will

be seen in chapter 5, where the results show that the real age of an otolith is 6, even though, according to Table 4.2, there were no age occurances under 7. "Bigger" and "smaller" will be the referred names of the datasets with their respective image resolutions further down the thesis, unless specified otherwise.

4. As described in subsection 2.8.2, it is common practice to divide datasets into three parts: a training set, a validation set, and a testing set. For each of the two resized datasets described in the previous step, this train-val-test split was applied. The total of 4096 images were divided into 3277 samples for training, 410 for validation, and 409 samples for testing.

A sample of an image from the smaller dataset and the bigger dataset as described in step 3 above, are shown in Figure 4.5.



(a) Sample with resolution of 64 × 96 **(b)** Sample with resolution of 512 × 768

Figure 4.5: Two samples of otolith images. One from the smaller dataset as seen in Figure 4.5a, and one from the bigger dataset as seen in Figure 4.5b.

As shown in Figure 4.5a, the images from the smaller dataset did not retain many details after the downscaling process. The annuli of the otoliths were mostly blurred out and not visible, making it difficult to determine whether a dot was placed correctly or not by the GAN. The smaller dataset was mainly utilized for testing the GAN in an efficient and non-demanding training procedure to assess whether there was any indication of learning. The images of the bigger dataset, shown in Figure 4.5b, were downscaled in a way that preserved sufficient details such that the annuli were visible, while still keeping the computational burden manageable. With these images, it was possible to evaluate the accuracy of dot placement by the GANs.

4.3 Main Experiment

This section describes how the main experiment was conducted, with its data preparation, GAN architecture, training settings and performance measures.

4.3.1 Image-to-Image Data Preparation

The final dataset used in the experiments consisted of images and tabular data of coordinates for the locations of dots on each image. An example of an otolith with its corresponding annuli coordinates is shown in Figure 4.6, where the coordinates of the dots were extracted from the tabular dataset, presented in Table 4.3, and drawn on the image.

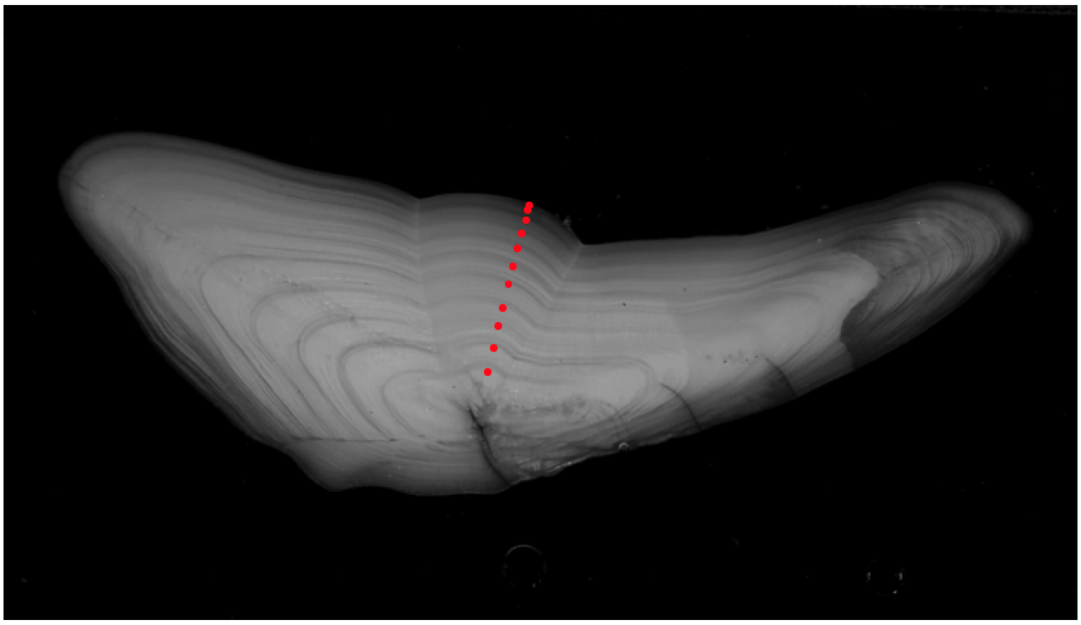


Figure 4.6: An example of an otolith image from the bigger dataset, where coordinates of dot locations from the tabular dataset in Table 4.3 is drawn on the image, showing where the annuli are located.

To ensure that the GAN was strictly image-based, further preparation of the data was necessary. The objective was to train the GAN's generator to produce dots based solely on images, which was achieved by preparing the current data in the following manner:

1. For each otolith sample, the corresponding dot coordinates were extracted from the tabular data and concatenated, resulting in each sample being a pair of an otolith image and its dot coordinates in an array [Otolith image, Dot coordinates], where the first index represented the image and the second index represented the correct dot coordinates.

2. Each training sample from the smaller and the bigger datasets, consisting of an otolith image and annuli coordinates, were transformed into *tensors*[1]. The otolith images were normalized and the dot coordinates were corrected to match the dimensions of the resized otoliths images. These samples were saved locally to the computer's disk (0.59MB per smaller sample, 1.6MB per bigger sample) in order to save time and improve training performance, rather than having the program load and perform the transformations for each sample while training. This allowed the program to directly load and use each sample or mini-batch of samples from the disk during training and testing.

3. The otolith image in each sample served as the *source* image for training. Its coordinates were used to create the corresponding *target* image. For each sample, a new black image of the same dimensions as the otolith image was created as the target, containing white dots at the coordinates obtained from the sample. The white dots in the target images had larger extent than only being the size of 1 pixel each. They were created as squares with an area of 9 pixels each.

[1]Tensors are simply mathematical objects that can be used to describe physical properties, just like scalars and vectors [89, 76].

A source image and a target image from the bigger dataset is illustrated in Figure 4.7.

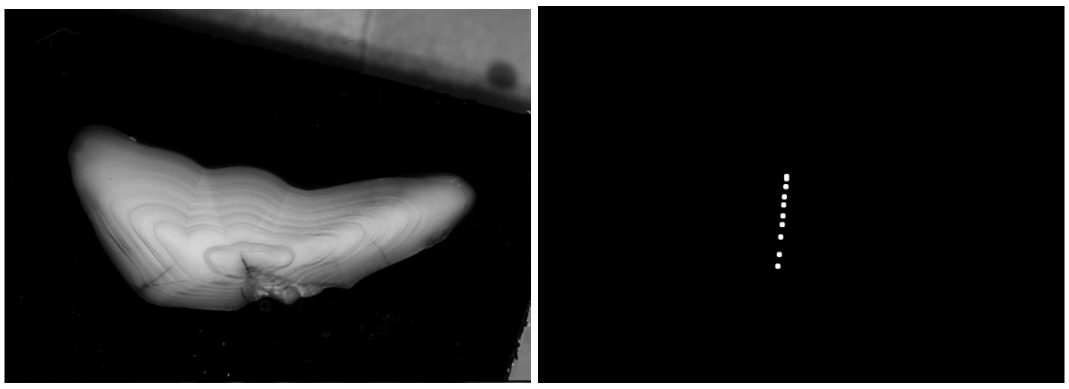

(a) Source: Otolith image sample. **(b)** Target: Respective dot image.

Figure 4.7: A pair of an otolith image and its dot image from the bigger dataset, where the otolith image is the source and the dot image is the target. The dot image is made in the same dimensions as the otolith image, where the white dots are drawn according to the sample coordinates from the dataset. Note that the dots have a larger extent than only being of a size of 1 pixel each.

Visualizing these pairs on top of each other will result in an otolith image with dots as shown in Figure 4.6.

4.3.2 GAN Architecture

The goal was to train the GAN's generator to produce *dot images* by providing it with otolith images as input (source). The GAN's discriminator then used either the generated dot image from the generator or a real dot image (target) as its input, conditioned on the corresponding otolith image (source). The architecture is visualized in Figure 4.8.

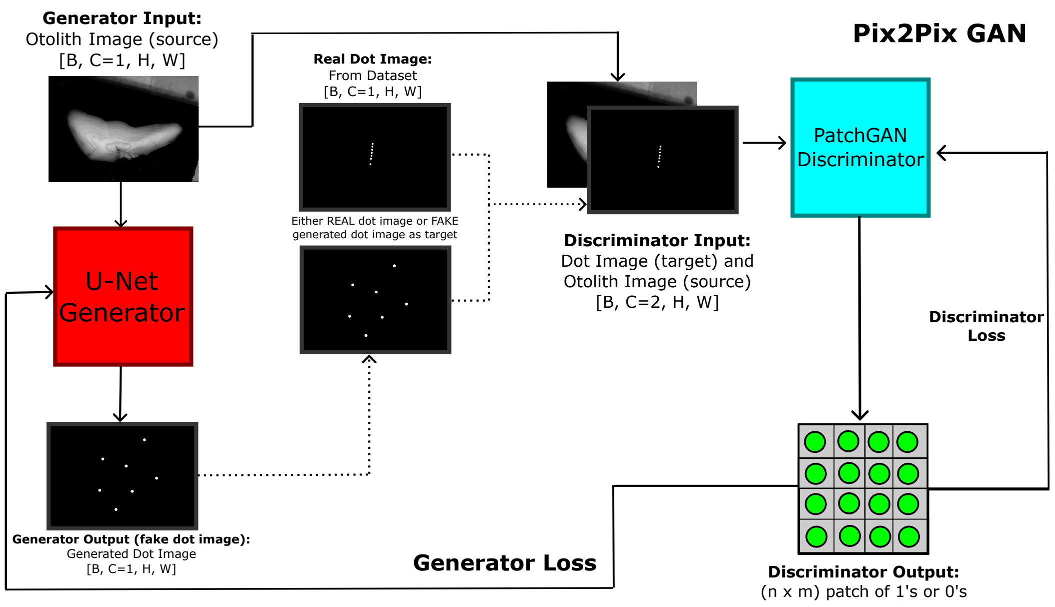


Figure 4.8: The GAN of the main experiment. Otolith (source) image as generator input, where the output is a generated dot image. Same otolith image (source) and either a real dot image from the dataset or the fake generated dot image (target) as discriminator input, where the output is a $n \times m$ patch of $1's$ or $0's$. B denoting the batch size, C for channels (2 for discriminator because its input is both the source and target images concatenated), H and W are the height and width of the images.

The U-Net generator architecture was similar to that shown in Figure 2.17 in subsection 2.6.1, with a few modifications to the input and output dimensions. In addition, all the pooling layers were converted to conv layers, resulting in a fully convolutional architecture. The PatchGAN discriminator used in the experiment was also similar to the one described in the paper by Isola et al. [38], with a few alterations such as the conversion to a fully convolutional network and modifications to the dimensions of the input and output patch. Several patch sizes were tested in the experiments. A few CNN architectures used in the experiments are shown in Appendix A.

4.3.3 Experiment Settings

The GAN architectures and all the machine learning code of the experiments were implemented using the Python library, PyTorch [55].

Data Augmentation

The data augmentations performed on the samples are described in section 2.8.1. Each sample loaded from the training data had a 50% probability of being transformed. These transformations included rotation or flipping, with probabilities specified in Table 4.4. The same transformations were applied to both the otolith images and the dot images. The 50% of samples that were transformed had a new 50% probability of being vertically flipped and a 50% probability of being rotated by either −90 or 90 degrees.

Data Augmentation (50% of occurring upon loading a sample)	**Probability of transforms**
Rotating the pair 90 degrees or −90 degrees	50%
Flipping the pair on their vertical axis	50%

Table 4.4: Table of data augmentation details of training samples.

The table above shows the data augmentations used in the experiments summarized in Table 4.6. In the general experimentation, multiple distributions and mixtures of the ones shown in Table 4.4 were tested with.

Hyperparameters

Multiple trainings were performed with different hyperparameter configurations. All the notable hyperparameters used in the experiments are summarized in Table 4.5. In all the experiments, a *manual cuda seed* [75] of 9 was applied when training with GPU for reproducibility purposes.

Hyperparameters	**Values**	**Details**
Image Resolutions	64×96 or 512×768, and 512×512 or 128×128	From the smaller and bigge datasets. Also tested square cropping the images from the bigger dataset to 512×512 and 128×128.
Optimizer	Stochastic Gradient Descent with Adam	$\beta_1 = 0.5$, $\beta_2 = 0.999$
Mini-Batch Size	1, 4, 8, 32, 64	Depending on image sizes and resource capabilities.
Loss Functions	BCE-Loss and L1-Loss for Pix2Pix, and Wasserstein Loss with gradient penalty for Wasserstein GAN	Both the generator and the discriminator use BCE-loss, where the generator in addition use L1-loss in Pix2Pix. The λ, ℓ^1 penalty, was set to 100 or 200. When using Wasserstein GAN, the Wasserstein loss was used with gradient penalty, where the penalty λ was set to 10.
Learning Rate	0.001, 0.0001, 0.0002, 0.0003	With a learning rate decay of 0.5 every 10th, 25th or 50th epoch when using Pix2Pix.
LeakyReLU Slope:	0.1, 0.2	
Patch size of discriminator/critic	512×768, 64×96, 62×94, 13×13, 6×10, ...	Many patch sizes have been tested, ranging from close to 1×1 to the whole image for the purpose of experimentation. The patch sizes varied depending on how deep the discriminator architectures were.
Epochs	50, 100, 150, 200, 500, 750	Number of epochs for the different training phases.

Table 4.5: Hyperparameters used in the experiments.

Training Scheme

Within the timeframe of when hardware was available (section 4.6), a series of training experiments were conducted using the same methodology. Both the hyperparameters in the experiments and the number of layers used in the generator and discriminator networks varied. A summary of the hyperparameters used are shown in Table 4.5.

The training was conducted in the following way:

- For every mini-batch, calculate the losses of both the generator and discriminator. These losses are logged, and printed every user-defined iterations in the terminal for visualization.
- For every user-defined iterations, run the model on 5 samples from the validation dataset and save the images, giving a visual representation of how the model is performing during the training, which can be manually inspected in a qualitative manner.
- When training is finished, testing phase is initialized, where the final trained model (which is saved to the disk after training) is loaded and tested on the unseen test data. For each sample, a visualization image of the otoliths are saved, where one has real dots drawn on the image, and one has generated dots on the image. In addition, an algorithm (section 4.4) is applied on the images, calculating the number of dots in each image, resulting in the age of the otoliths.
- Plots over losses are also saved on disk.
- Histograms of age determinations (amount of dots) are saved on disk.

4.4 Dot Counting Algorithm

Given the fact that the experiments were strictly image based, a way of counting the generated dots was applied. When the generated dot images are visualized on top of the otolith images, as shown in Figure 4.6, these dots had to be counted to determine the age of the otolith. This was partially done by using the Python package *OpenCV* (cv2), which is an open source computer vision library [87].

By using cv2, we can find all the contours of an image, in my case being generated dots. Then, by calculating the sum of brightness values of all the pixels inside each contour, we can determine whether one dot is bright enough to be considered as a dot. Lastly, the sum of the bright, generated dots gives us the count, being the age. These steps are visualized in a simple way in Figure 4.9.

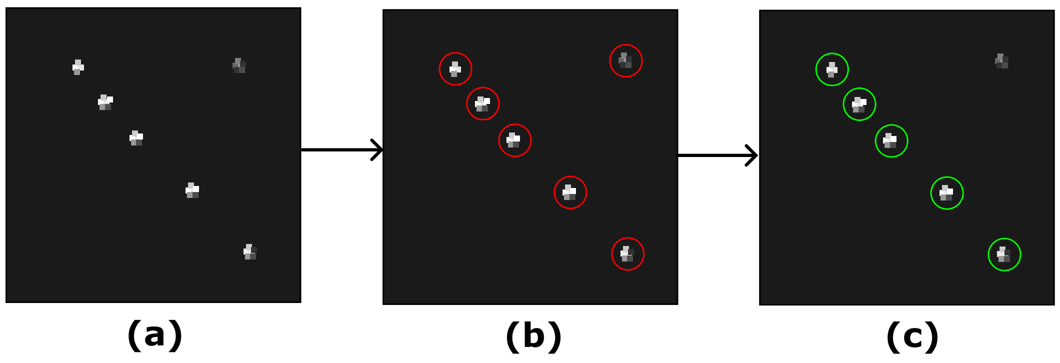


Figure 4.9: A simple visualization of the dot counting algorithm. In **(a)**, we see the dot image produced by the generator. In **(b)**, the image is processed by cv2 to find all contours of the image, shown as red circles. For each contour, all the pixels inside are used to calculate the sum of brightness denoting one dot. If this sum is higher than a brightness threshold, it would then be counted as a dot. In **(c)**, we see the resulting image where all the contours, except one, are marked to be bright enough to be counted as dots, shown as green circles.

The max brightness of a white pixel in a grayscale image is 255 (where 0 is completely black). The area of dots used in the training data was set to 3 pixels, meaning that the area of one *perfect* dot was $3 \times 3 = 9$ pixels. The "dots" were practically squares in the dot images used in the training. If all the pixels of one dot area of 9 was at its brightest, the sum of brightness would equal 2295. However, the generated dots were not always perfect, where the areas of the dots varied, and their brightness varied. Therefore, a threshold of 600 was set, where each dot that had a sum of brightness larger than 600 was counted as a dot. The value sum of 600 was chosen because this value was the lowest reasonable brightness I personally thought one dot should have to be counted as a "bright enough" dot when looking at the images.

4.5 Other Ideas and Experiments

In the quest of implementing a solution to the research question of the thesis, different ideas were considered. While the first idea (subsection 4.5.1) was not ultimately included in the final analysis, it proved to be an interesting experiment during the brainstorming and trial-and-error phase of the thesis. After this, the main experiment of using Pix2Pix GAN with U-Net generator and PatchGAN discriminator was pursued to address the research question. After encountering issues with mode collapse when using Pix2Pix, I began experimenting with Wasserstein GANs, which ultimately yielded more promising and satisfactory results, ending in being the best performing model.

4.5.1 Tabular GAN

The initial idea involved using a CGAN to generate tabular data, specifically a numeric table of coordinates. The generator would take an otolith image as input and generate corresponding coordinates, which the discriminator would then evaluate as either real or fake based on the same otolith image as input. The architecture of this GAN is illustrated in Figure 4.10.

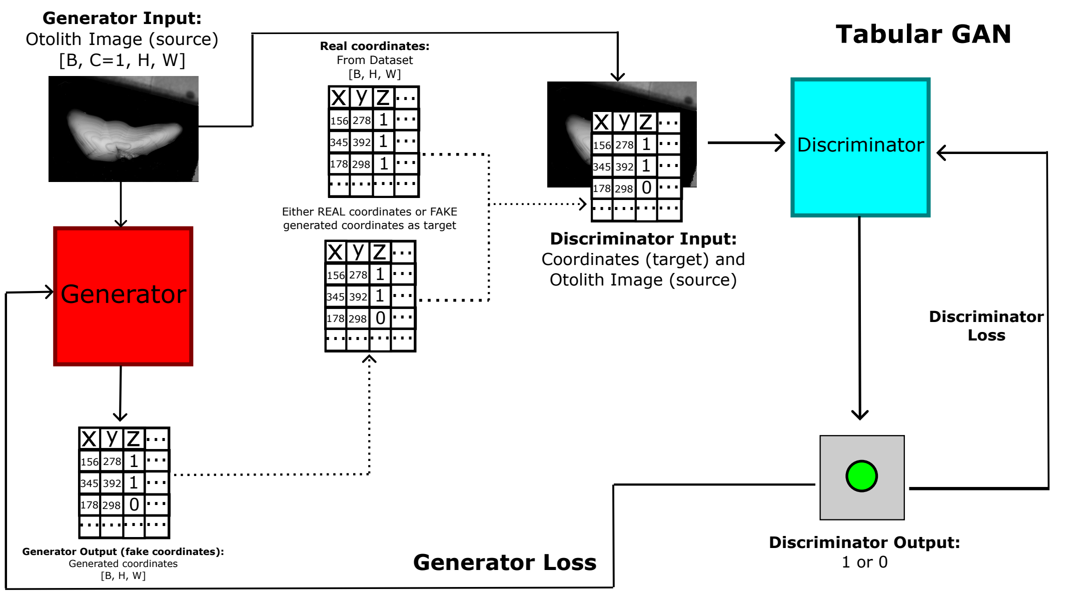


Figure 4.10: The Tabular GAN idea. Otolith (source) image as generator input, where the output is a generated table of coordinates. Same otolith image (source) and either a real coordinates from the dataset or the fake generated coordinates (target) as discriminator input, where the output is 1 or 0. *B* denoting the batch size. *H* for height and *W* for width.

This idea was not trained and tested properly because of architecture difficulties and hardware availability.

4.5.2 Wasserstein GAN

The WGAN experiment was conducted in a manner similar to the original Pix2Pix experiments. The architecture was similar, but the discriminator was changed to a critic that scored the realness or fakeness of its input. The BatchNorm layers were also replaced with InstanceNorm. This GAN was trained using Wasserstein loss with gradient penalty, where the generator was updated every five iterations and the critic was updated each iteration. This design showed promising results in addressing the mode collapse issues of Pix2Pix, ultimately yielding the best performing design with more accurate results compared to the original Pix2Pix experiments. The architecture is shown in Figure 4.11

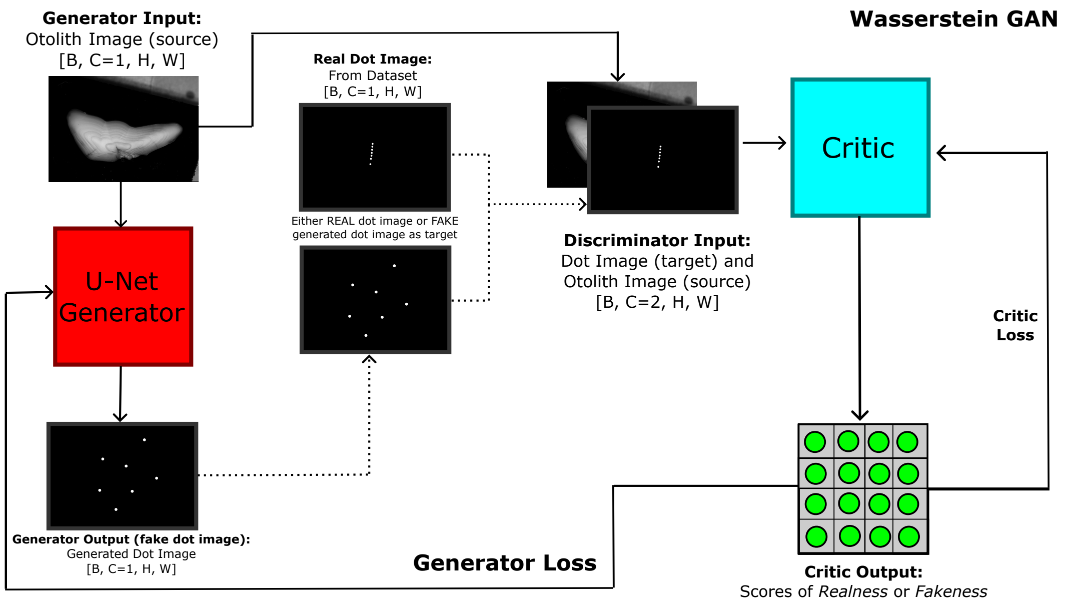


Figure 4.11: The Wasserstein GAN of the last experiment. Otolith (source) image as generator input, where the output is a generated dot image. Same otolith image (source) and either a real dot image from the dataset or the fake generated dot image (target) as critic input, where the output is a matrix denoting scores of realness or fakeness of the input. *B* denoting the batch size, *C* for channels (2 for critic because its input is both the source and target images concatenated), *H* for height and *W* for width.

Note that the critic of the WGAN is visualized as a patch. This is because it, in addition to mainly being a non-patch, was also experimented as a patch architecture similar to the PatchGAN discriminator of Pix2Pix – differing in loss function.

4.6 Hardware Availability

In order to run the models with with their image processing capabilities, strong hardware was required (subsection 2.7.1). Due to the limitations of my personal laptop, all experiments were reliant on remote hardware when it was available. The University of Bergen provided access to two remote servers, named Janus and Birget, with varying levels of availability. Their specifications are detailed below:

- **Janus:** Partially available between 1. November 2021 – 15. December 2021. And partially available 22. October 2022 – 10. November 2022.
  - CPU: Intel(R) Core(TM) *i*9-7900*X* CPU @ 3.30GHz
  - GPU: 1× GeForce RTX 2080 Ti 11GB
- **Birget:** Partially available between 5. November 2022 – 23. December 2022.
  - CPU: AMD EPYC 7742 64-Core Processor
  - GPU: 8× *A*100-SXM-80GB *(One used when available)*

All the experiments were conducted on these servers when available.

4.7 Summary of Relevant Experiment Runs

ID	Settings	Details
1	**GAN:** Pix2Pix **CNN Architectures:** Figure A.1 **Image Resolution:** 64 × 96 **Mini-Batch Size:** 64 **Learning Rate:** 0.0003 **LeakyReLU Slope:** 0.1 **Epochs:** 200 **Patch Size:** 6 × 10	No data augmentation was applied. This experiment was the only experiment out of the four in this table where the smaller dataset was used. From the testing with the smaller dataset, this experiment in particular gave promising results that made way to move on to test the GAN on the bigger dataset.
2	**GAN:** Pix2Pix **CNN Architectures:** Figure A.2 **Image Resolution:** 512 × 768 **Mini-Batch Size:** 8 **Learning Rate:** 0.0003 **LeakyReLU Slope:** 0.2 **Epochs:** 500 **Patch Size:** 62 × 94	With all data augmentations applied.
3	**GAN:** Wasserstein **CNN Architectures:** Figure A.3 **Image Resolution:** 512 × 512 **Mini-Batch Size:** 8 **Learning Rate:** 0.0002 **LeakyReLU Slope:** 0.2 **Epochs:** 750 **Patch Size:** 13 × 13	With all data augmentations. Images cropped to the size 512 × 512.
4	**GAN:** Wasserstein **CNN Architectures:** Figure A.4 **Image Resolution:** 128 × 128 **Mini-Batch Size:** 32 **Learning Rate:** 0.0002 **LeakyReLU Slope:** 0.2 **Epochs:** 750 **Patch Size:** No patch architecture	With all data augmentations, where all images were cropped to the size 128 × 128.

Table 4.6: Summary of relevant runs for different experiments.

Chapter 5

Results

In this chapter, I present the results of the experiments described in Chapter 4. I trained several models using different hyperparameter configurations and architectures, as shown in Table 4.5. The candidate models that were selected for this chapter are listed in Table 4.6. The results shown in this chapter are based on the validation and test data from both the smaller and bigger datasets (subsection 4.2.2). In chapter 6, I will further discuss the performance of the models and interpret the results.

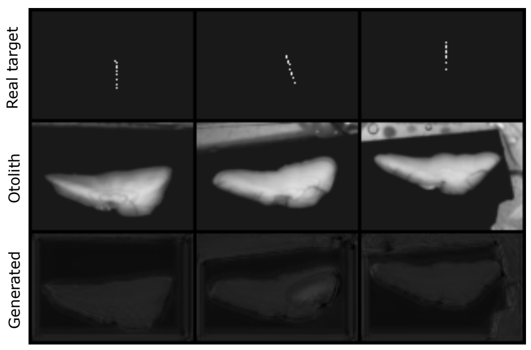
5.1 Main Experiment: Pix2Pix GAN

The Pix2Pix experiment was initially conducted using the smaller dataset of images with resolution of 64×96. After a learning trend was established using this dataset, I moved on to using the bigger dataset of images with resolution of 512×768.

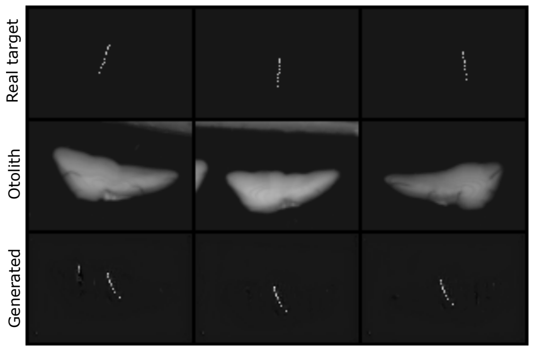
5.1.1 Training and Testing Pix2Pix GAN with the Smaller Dataset

Training

Experiment 1, shown in Table 4.6, was conducted using the smaller dataset. The following figures show the performance of the model during training, validated using the validation set. The top row contains the real dot images, the middle row contains the otolith images, and the bottom row contains the generated dot images produced by the generator. This format applies to all figures showing results in this chapter, unless otherwise specified.



(a) First epoch.



(b) Last epoch.

Figure 5.1: Results from the training with images from the smaller dataset (Pix2Pix GAN), where the model is tested on the validation data. Top row contains the real dot images, middle row the otolith images, and bottom row the generated dot images from the generator. Image **(a)** shows the results from the first epoch. Image **(b)** shows the results from the last epoch.

As shown in Figure 5.1, a trend of dot generation can be seen once the model has been trained. However, as discussed in subsection 4.2.2, the smaller dataset lack the detail necessary to accurately interpret the placement of the generated dots on the annuli. Therefore, I did not visualize the generated dots on top of the otoliths. This experiment was only used in the building phase of the model architecture. The plot of the losses for the generator and discriminator is shown in Figure 5.2.

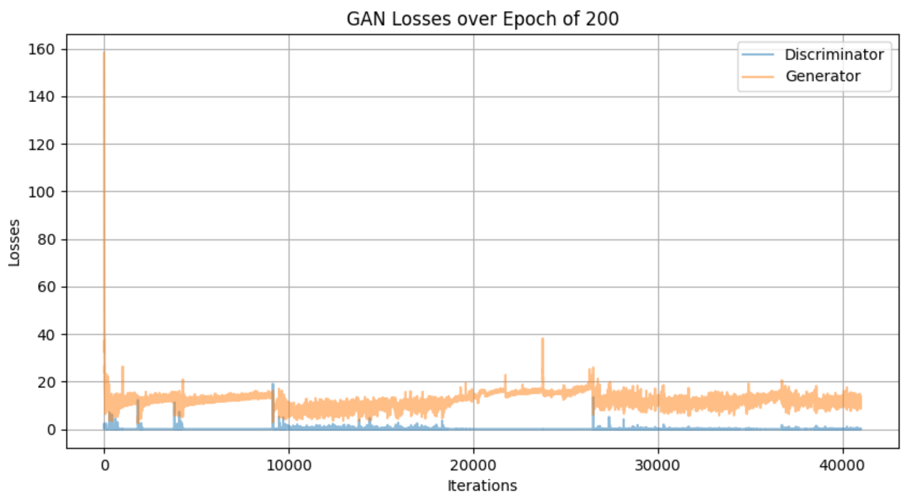


Figure 5.2: Plot over the losses of the generator and the discriminator in the experiment using the smaller dataset (Pix2Pix GAN).

Testing on Unseen Data

The figure below shows the performance of the model on the test set, the unseen data.

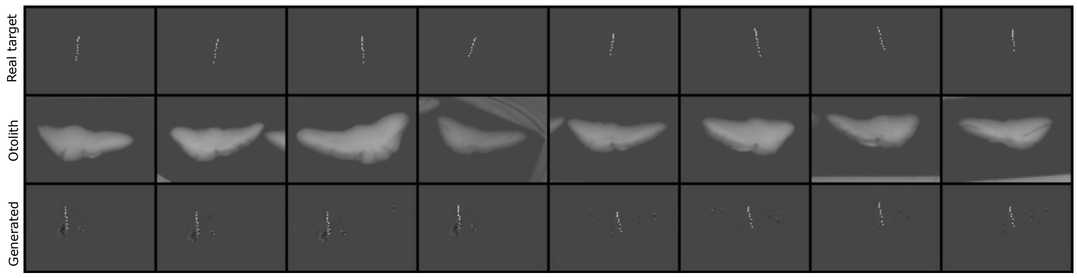


Figure 5.3: Results from the testing with the smaller dataset (Pix2Pix GAN) using the test data. Note that these images have a gray tone and low contrast when compared to the other images of the experiments. This issue was caused during visualization when developing the architecture, which was later fixed in the experiments.

Once I observed that the model demonstrated evidence of learning and was able to generate dots, I proceeded to experiment with the architecture using the bigger dataset containing larger images.

5.1.2 Training and Testing Pix2Pix with the Bigger Dataset

Training

In the following figures, we see the results of experiment 2 from Table 4.6. Figure 5.4 and Figure 5.5 present the results from the first epoch and the final epoch, respectively.

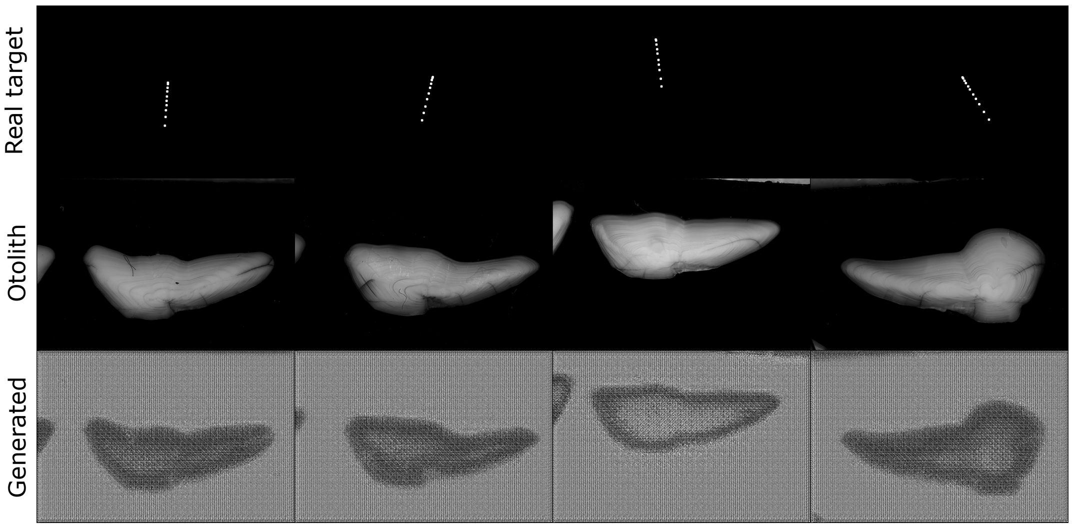


Figure 5.4: First epoch of training with the bigger dataset (Pix2Pix GAN).

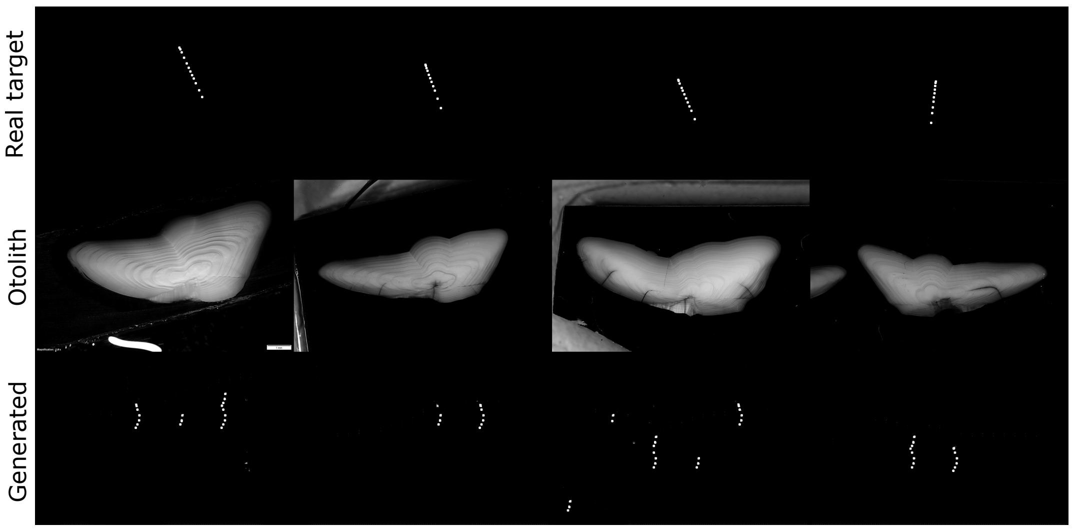


Figure 5.5: Last epoch of training with the bigger dataset (Pix2Pix GAN).

As shown in Figure 5.5, the generated dots are positioned in multiple locations in the images. These dots are visualized on top of the otoliths when the model is evaluated on the test set in section 5.1.2. The plot of the losses is shown in Figure 5.6.

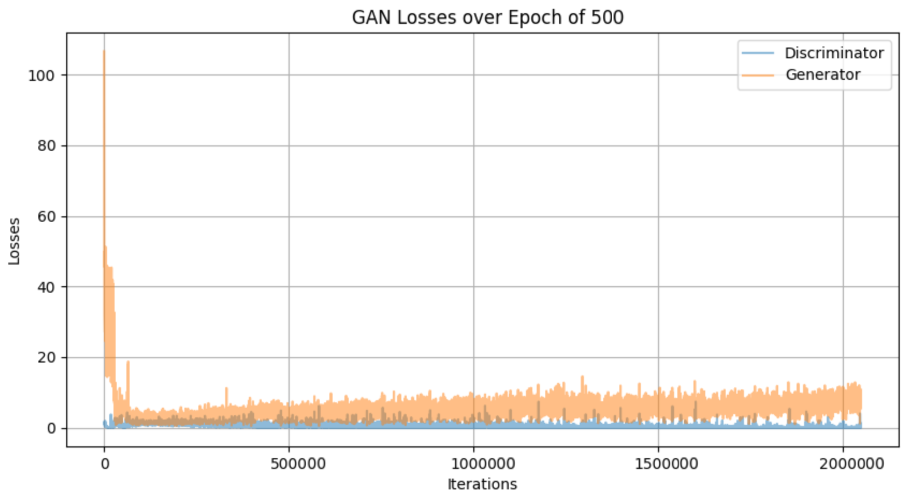


Figure 5.6: Plot over the losses of the generator and the discriminator in the experiment using the bigger dataset (Pix2Pix GAN).

Testing on Unseen Data

Figure 5.7 displays the results of the Pix2Pix model's performance on the test data.

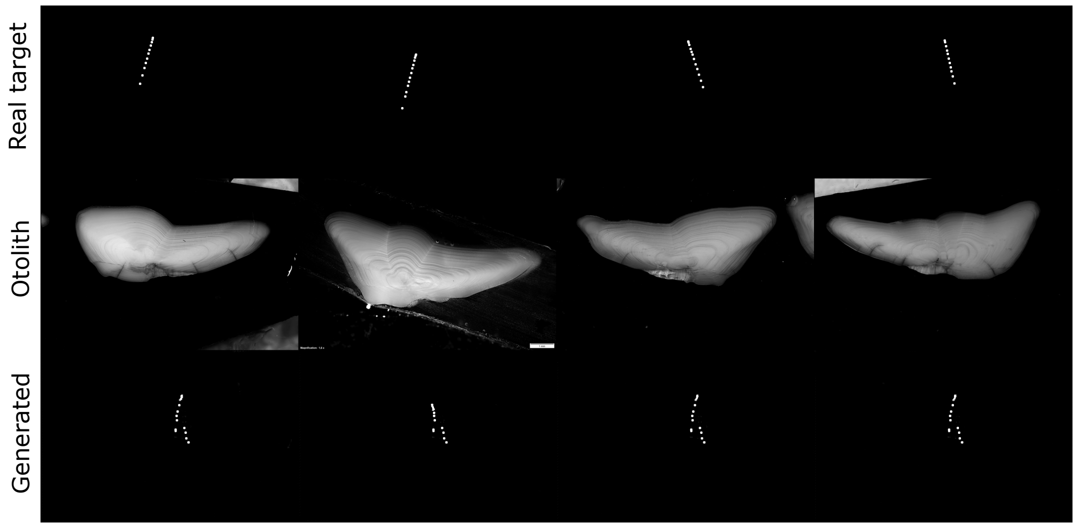


Figure 5.7: Results from the testing with the bigger dataset (Pix2Pix GAN) using the test data.

Figure 5.8 presents a few results where the dots are overlaid on top of the otoliths. The images on the left show the real dots from the dataset overlaid on the otoliths, and the images on the right show the dots generated by the generator on the otoliths, both from the test data. This format applies to all subsequent results where the dots are visualized on top of the otoliths, unless otherwise specified.

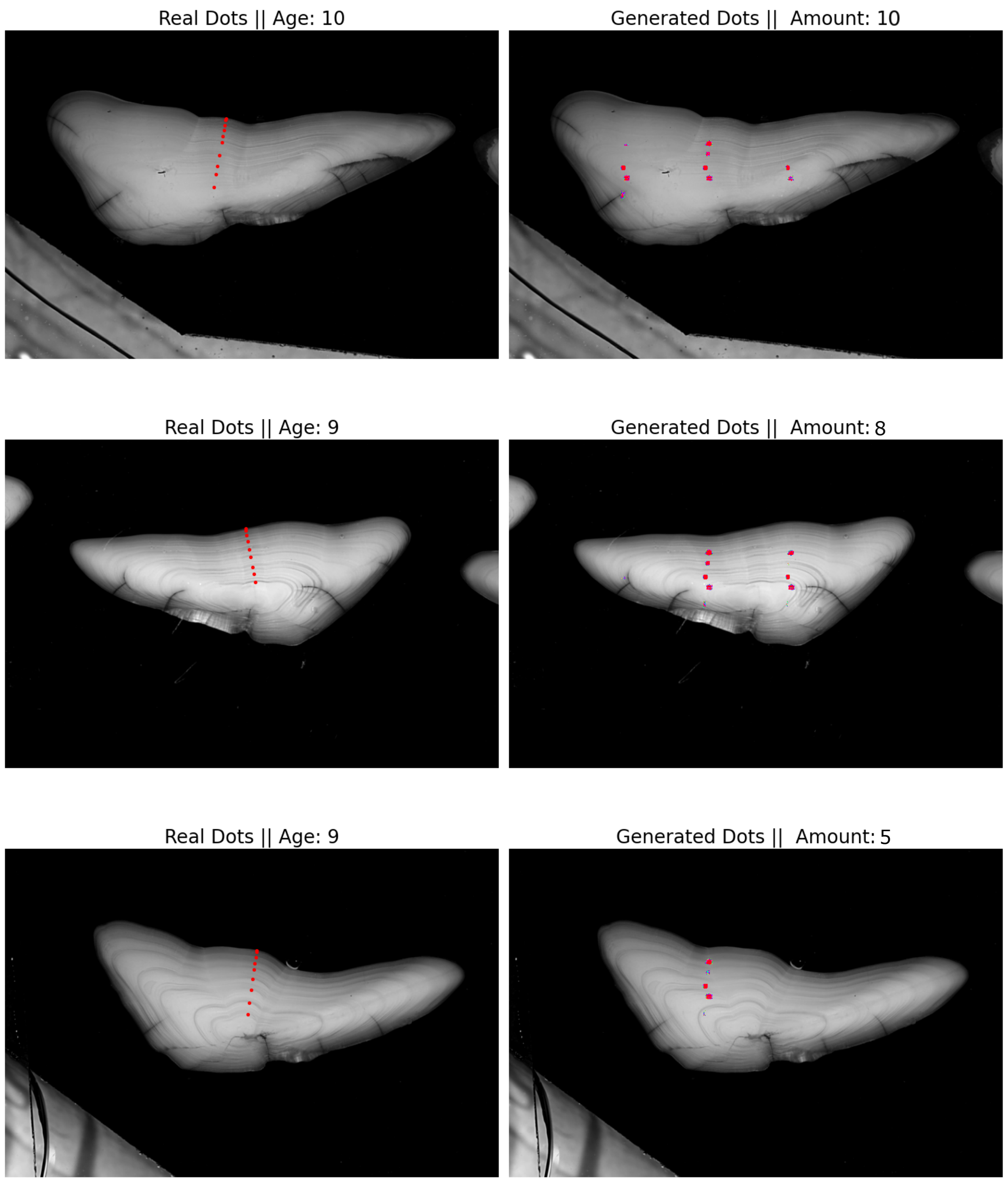

Real Dots || Age: 10
Generated Dots || Amount: 10
Real Dots || Age: 9
Generated Dots || Amount: 8
Real Dots || Age: 9
Generated Dots || Amount: 5

Figure 5.8: Results from the testing with the bigger dataset (Pix2Pix GAN), where the dots are visualized on top of the otoliths. Left images contain the real dots from the dataset, the right images contain the dots generated by the generator, both from the testing data.

As seen in Figure 5.8, the initial experiment of using Pix2Pix GANs yielded poor results, both in the generation of dots and in the results of the dot counting algorithm. Because of these poor results, no further statistics or visualizations are presented for the Pix2Pix experiments. The failure of Pix2Pix led to the investigation of alternative solutions. In the next section, I will present the results of the final experiments using the Wasserstein GANs.

5.2 Last Experiment: Wasserstein GAN

This section presents the results of experiments 3 and 4, as summarized in Table 4.6. These experiments were performed using the bigger dataset of images of resolution 512 × 768, which were square-cropped to 512 × 512 and 128 × 128.

5.2.1 Training Results

The results of the training using 512 × 512 images are shown in Figure 5.9 and Figure 5.10, while the results using 128 × 128 images are shown in Figure 5.11 and Figure 5.12. These results are derived from the validation data.

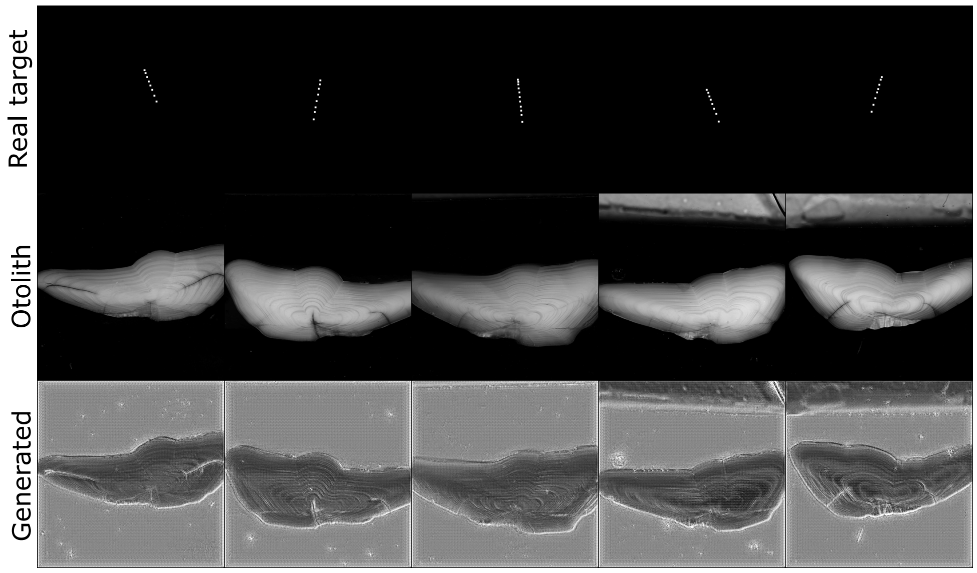


Figure 5.9: First epoch. Results from the training with Wasserstein GAN with 512 × 512 images, where the models are tested on the validation data.

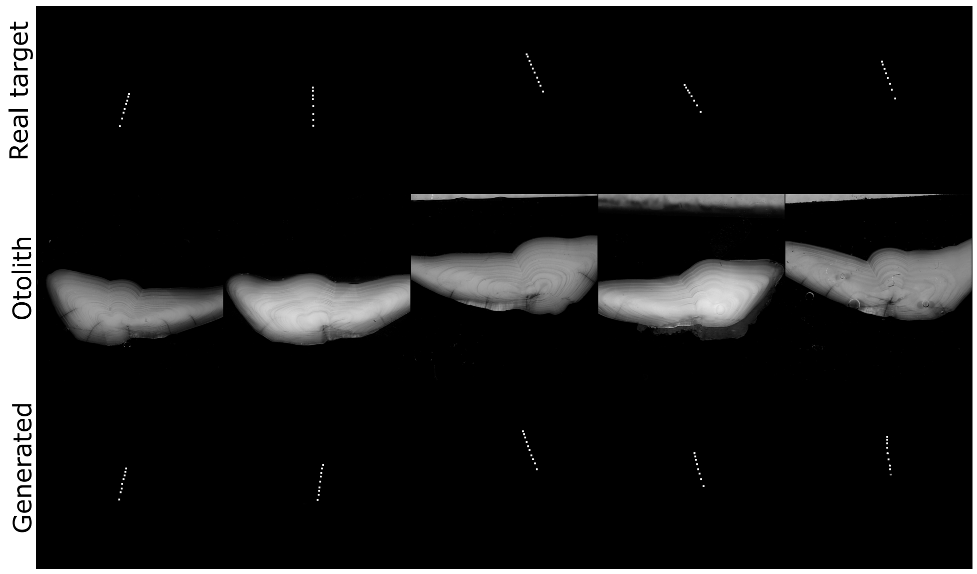


Figure 5.10: Last epoch of training with images of 512 × 512 (Wasserstein GAN).

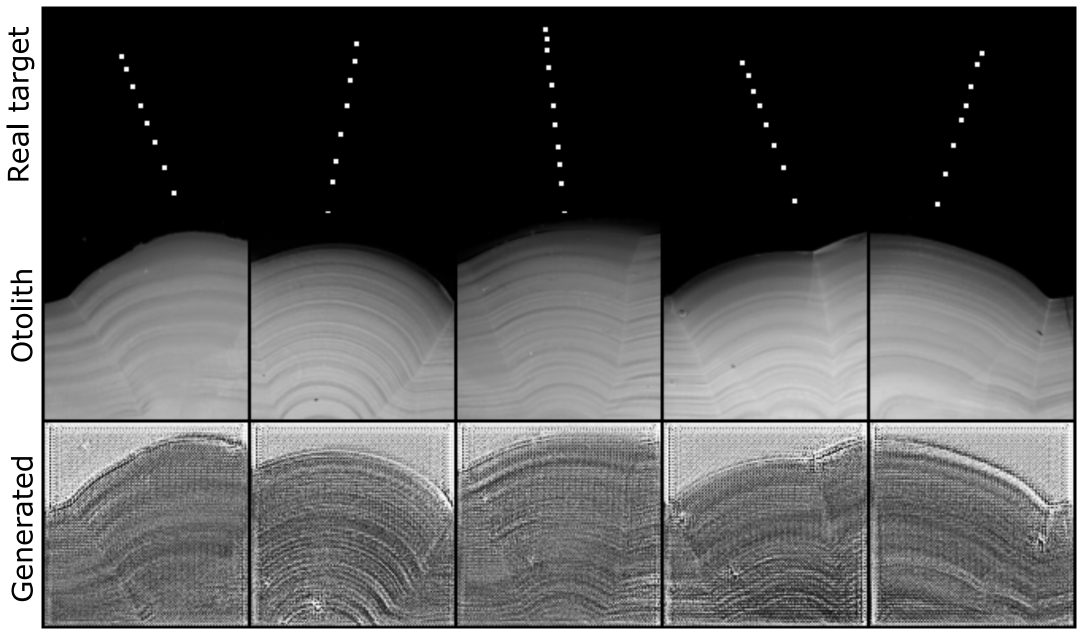


Figure 5.11: First epoch of training with images of 128 × 128 (Wasserstein GAN).

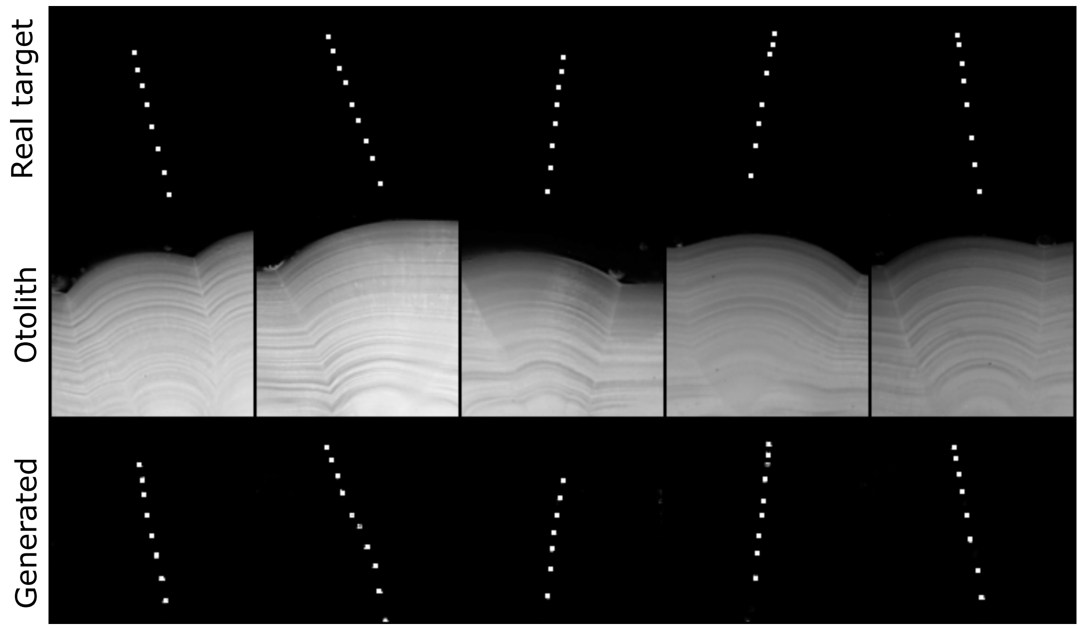


Figure 5.12: Last epoch of training with images of 128 × 128 (Wasserstein GAN).

The plots over the losses are shown in Figure 5.13 and Figure 5.14.

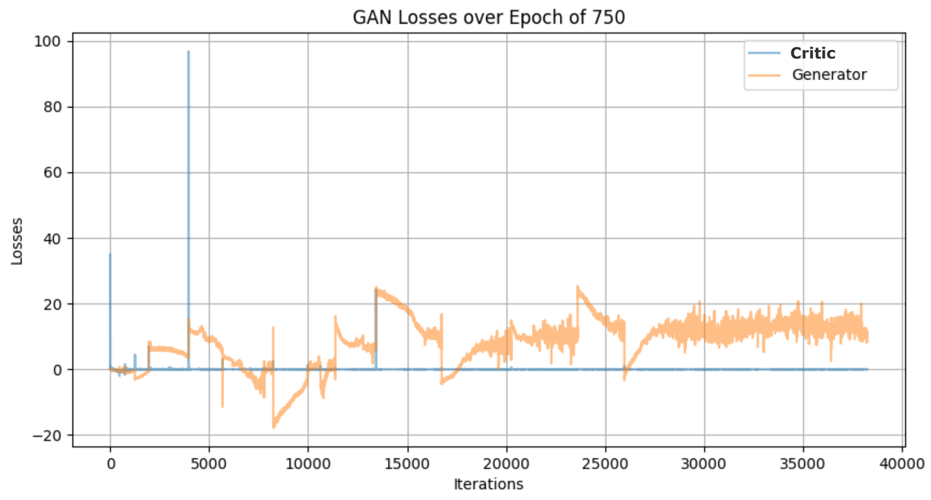


Figure 5.13: Plot over the losses of the generator and the critic using 512 × 512 images (Wasserstein GAN).

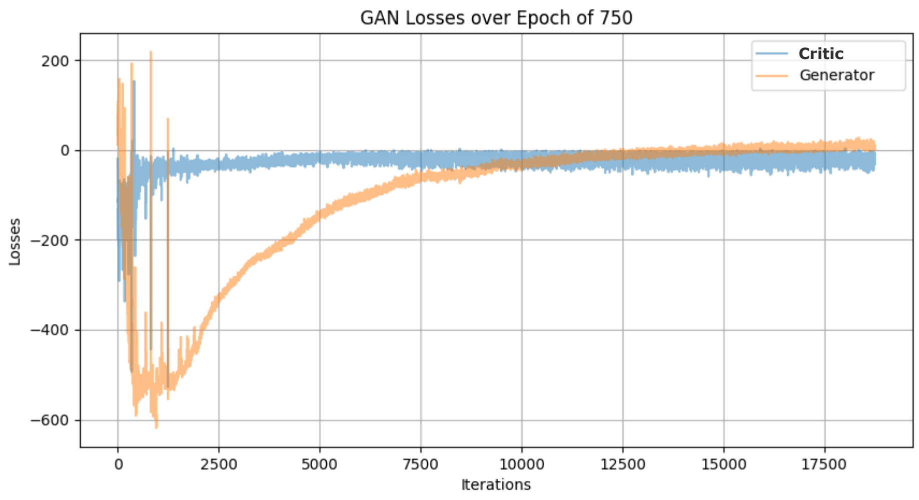


Figure 5.14: Plot over the losses of the generator and the critic using 128 × 128 images (Wasserstein GAN).

5.2.2 Testing Results

The following figures show the models' performance on the testing data, with the dots overlaid on the otolith images. Figure 5.15 shows the results for the 512 × 512 images, while Figure 5.16 shows the results for the 128 × 128 images.

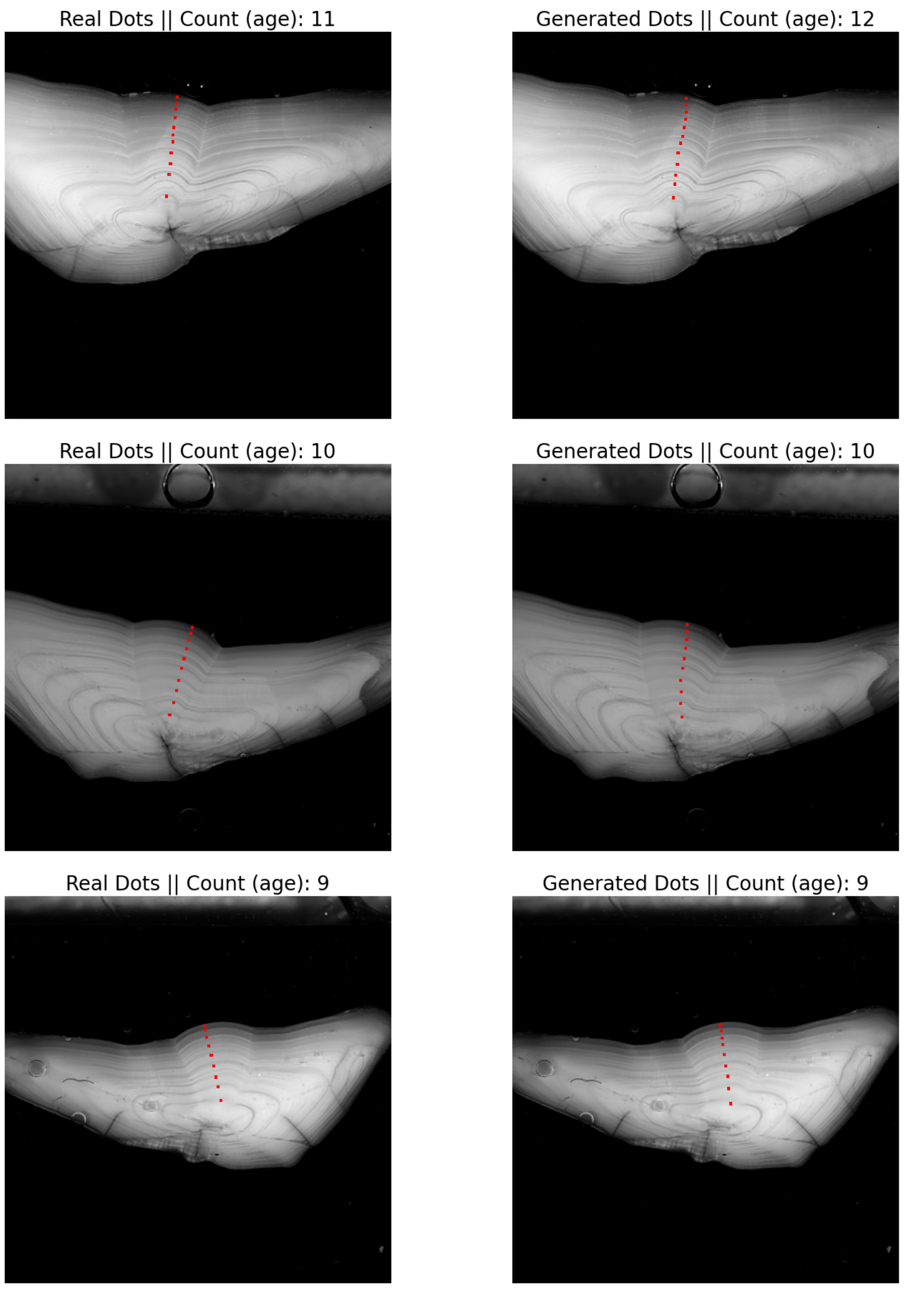

Real Dots || Count (age): 11
Generated Dots || Count (age): 12
Real Dots || Count (age): 10
Generated Dots || Count (age): 10
Real Dots || Count (age): 9
Generated Dots || Count (age): 9

Figure 5.15: Results from the testing with images of 512 × 512 (Wasserstein GAN), where the dots are visualized on top of the otoliths.

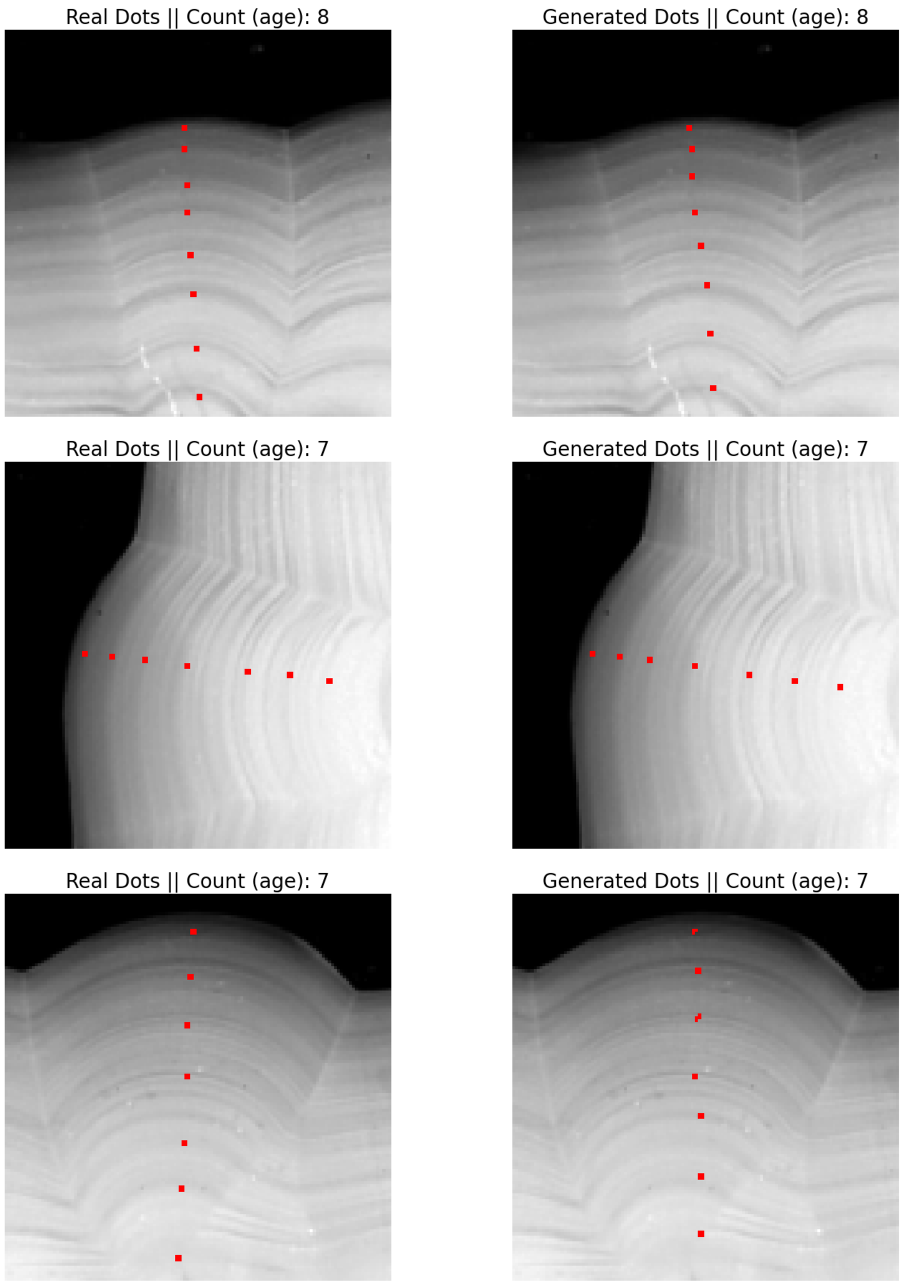
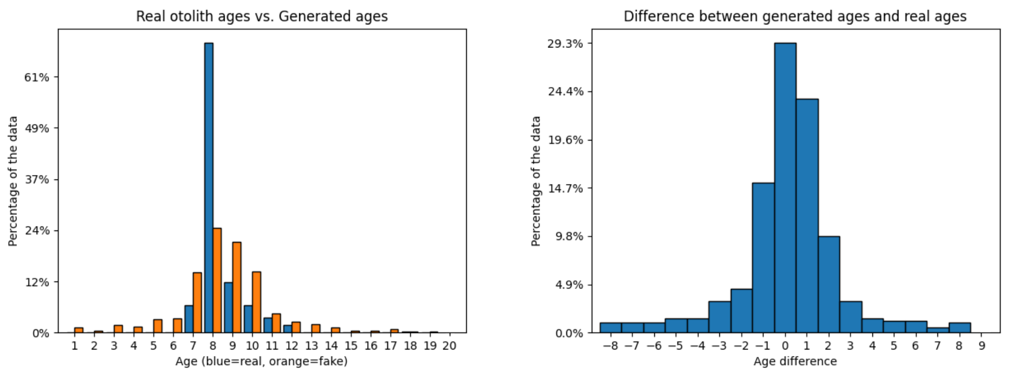

Real Dots || Count (age): 8
Generated Dots || Count (age): 8
Real Dots || Count (age): 7
Generated Dots || Count (age): 7
Real Dots || Count (age): 7
Generated Dots || Count (age): 7

Figure 5.16: Results from the testing with images of 128 $\times$ 128 (Wasserstein GAN), where the dots are visualized on top of the otoliths.

The results using Wasserstein GANs were clearly better than the Pix2Pix GAN results, as seen in Figure 5.15 and Figure 5.16. Additional results from the experiments with the Wasserstein GANs can be found in Appendix B. The histograms below display the age data obtained from the dot counting algorithm described in section 4.4.

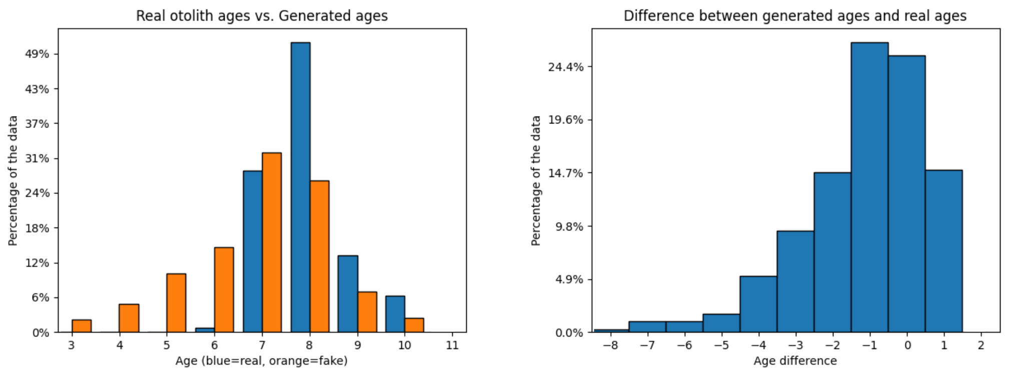


(a) Real and generated ages distribution.

(b) Real and generated ages difference.

Figure 5.17: Histograms over the resulting age data from the testing with 512 × 512 images (Wasserstein GAN). In Figure 5.17a, we see the distribution of the real data and the generated data, and their percentages of how many samples were within the different ages. Blue represents the real ages, orange the counted ages from the algorithm. In Figure 5.17b, we see the age differences between the generated ages and the real ages, showing us how many years off the generated ages were from the real ages.

When using images with a resolution of 512 × 512, the results of the dot counting algorithm appear to be slightly skewed towards positive values. This indicates that the amount of generated dots were more often higher than the actual ages. Examining the age differences in Figure 5.17b, 29.4% of the samples were classified correctly. 14.8% were off by −1 year, and 4.9% were off by −2 years, 23.9% were off by +1 year, 9.8% were off by +2 years from the correct ages. Considering differences of ±2 years from the results, the model's overall accuracy was 82.8%.



(a) Real and generated ages distribution.

(b) Real and generated ages difference.

Figure 5.18: Histograms over the resulting age data from the testing with 128 × 128 images (Wasserstein GAN), similar to Figure 5.17. Note that in the 128 × 128 images, some age occurrences was the age 6. As seen in Table 4.2, there are no age occurrences under 7 in the dataset. But these images are the cropped versions of the full-size images, where in some cases, an annuli was also cropped in the square-cropping process (subsection 4.2.2). This has also affected the age distribution between Figure 5.17a and Figure 5.18a given that they are not equal.

When using images with a resolution of 128×128, the results of the dot counting algorithm appear to be skewed towards negative values compared to those obtained using images with a resolution of 512×512, as shown in Figure 5.18b. This indicates that the amount of generated dot were more often lower than the actual ages. The model did correctly predict the dot counts for 25.1% of the samples. 26.9% of the generated dot counts were -1 year from the correct age, 14.8% were -2 years from the correct age, and 14.7% were $+1$ year from the correct age. Considering differences of ± 2 years from the results, the model's overall accuracy was 81.5%.

However, there were some issues with the results that impacted the accuracy of the estimated ages. These issues are discussed in the following section, subsection 5.2.3.

5.2.3 Issues in the Results of WGAN

Despite the promising results of the WGANs, as shown in Figure 5.15 and Figure 5.16, some results exhibited some issues, which will be described in the following figures.

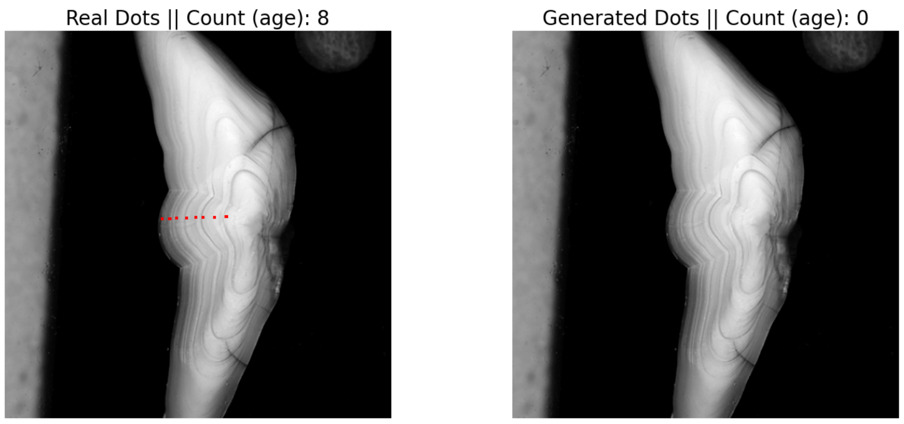


Figure 5.19: An example of an issue where the generator does not produce any dots, resulting in 0 in age.

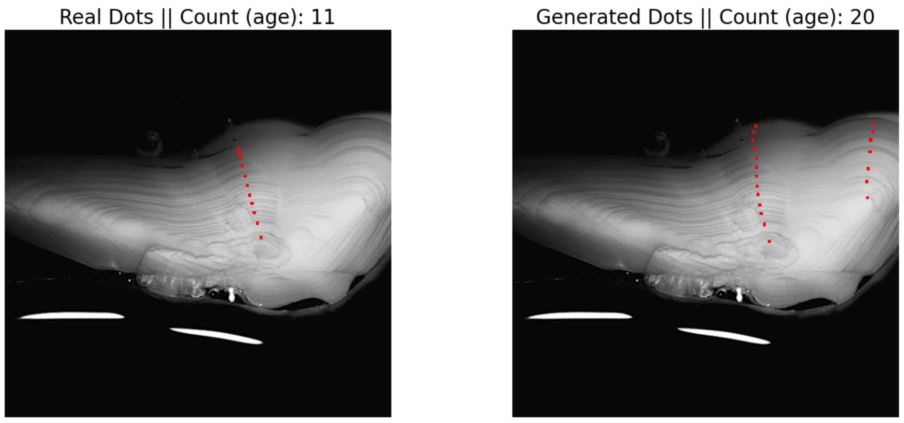

Real Dots || Count (age): 11
Generated Dots || Count (age): 20

Figure 5.20: Example of an issue where the generation of dots occurs on multiple places on the otolith. This in itself is not wrong, the dots are mostly sensibly placed, but the amount of dots representing the otolith age results in being wrong. This explains why the ages were skewed more to the right, as shown in Figure 5.17b.

I will in chapter 6 further discuss the results.

5.3 Bonus Experiment: Using an Image of a Tree Stump

As a fun side experiment, I wanted to test the trained Wasserstein GAN model on an image of a tree stump. After a few attempts, the model surprisingly did show *something*, in Figure 5.21. I will leave this result and its interpretation to you.

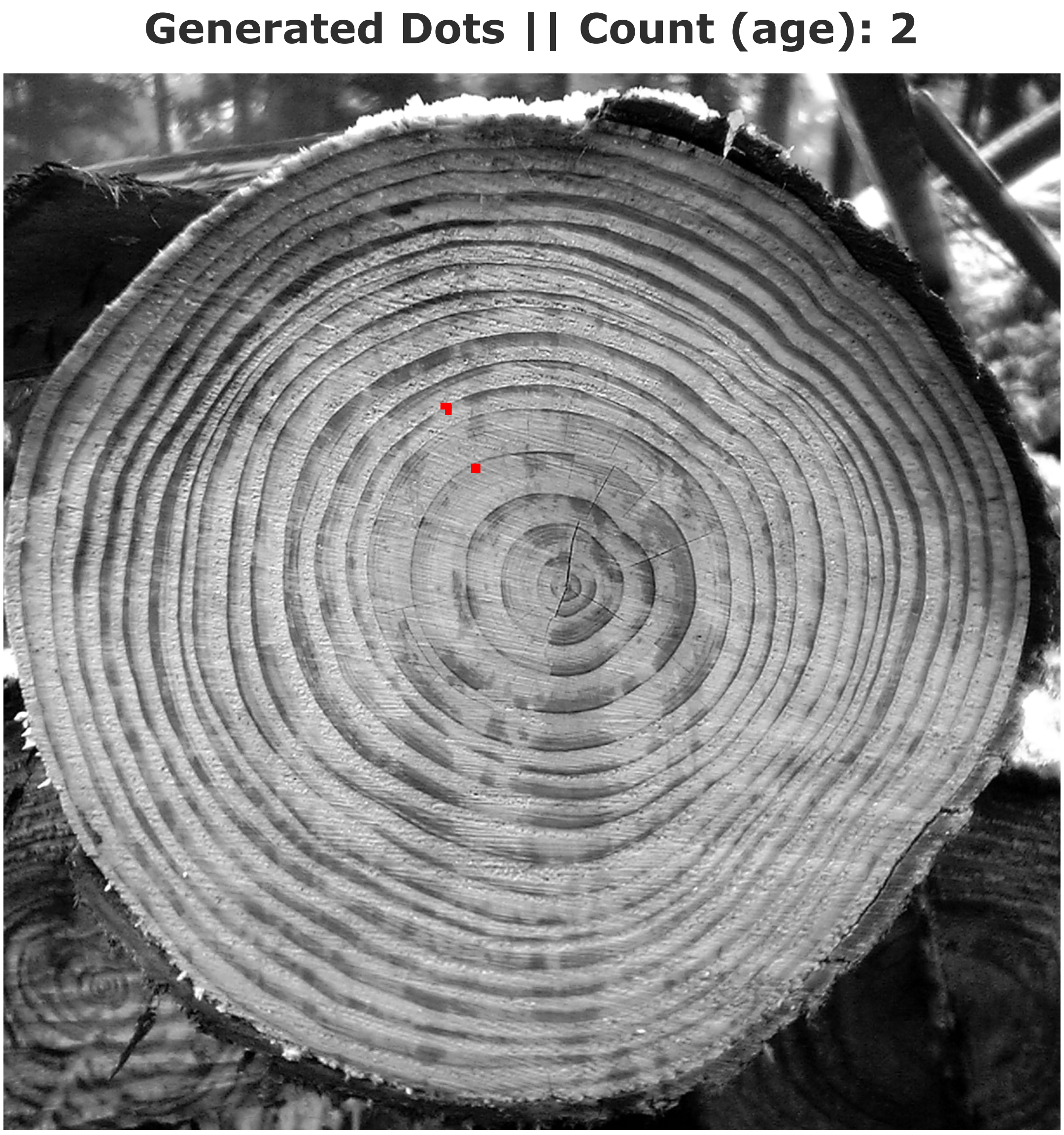


Figure 5.21: Result of the bonus experiment, where the trained Wasserstein GAN model is tested on an image of a tree stump. Counting the rings of the tree results in 19 years of age. Image of tree taken from aurorabreakup [4].

Chapter 6

Discussion and Future Works

This chapter will discuss the results of the experiments and their implications. The findings will be interpreted and the limitations of the work will be addressed in regard to the research question:

> *"What is the potential of using generative adversarial networks in developing a generative machine learning model for accurately annotating images of otoliths?"*

Furthermore, the actions that further can be done in light of this research will be described.

6.1 Interpretations

In the proposed solution to the research question, the goal was to use an image-to-image translation GAN, specifically a Pix2Pix GAN, as the primary experimental approach. However, the obtained results using this architecture were not precise and did not show evidence of well-generated dots. This led to the exploration of a slightly different image-to-image approach using Wasserstein GAN, where the loss was changed to Wasserstein loss and the discriminator of the network was replaced with a critic. The BatchNorm layers were also changed to InstanceNorm. The critic was tested both with a patch architecture and without. The performance of the Wasserstein GANs proved to be better than the Pix2Pix GANs, and more promising results were obtained with the approach.

6.1.1 The Experiment Models in General

The results obtained using Pix2Pix showed that the discriminators of the models generally performed well, effectively discriminating fake images from real ones (see Figure 5.2). In theory, the losses of both the generator and discriminator should be close to zero after many iterations, but as shown in the loss plots in Figure 5.2 and Figure 5.6, the generators never managed to reach this point and instead oscillated throughout training. Despite this, the quality of the generated images improved over time.

The Pix2Pix model using smaller images showed signs of promising results, as seen in Figure 5.3. This led to experimentation with larger images, which also exhibited a trend in dot production. However, further analysis revealed that the generated dots were generally weak, non-existent, or not meaningful. The primary issues with the Pix2Pix models were mode collapse and lack of variety in dot generation, as discussed in subsection 2.7.1. The fake dots generated by the Pix2Pix models often showed the same structure and dot patterns across multiple otolith images, and only by chance were they placed accurately on the annuli, as seen in Figure 5.7 and Figure 5.8.

The mode collapse problem was already evident in the results obtained using smaller 64 × 96 images, as shown in Figure 5.3. The first four results in the bottom left row are identical, as are the last four results on the right. There is no sign of variety. This problem also affected the results using larger images, as seen in Figure 5.7, where the generated dots are largely the same across all the images.

Despite multiple attempts to improve the Pix2Pix models using different hyperparameters and architectures, the results did not improve, continuing to suffer from mode collapse and inaccurate results. GANs are after all black box models, which can make it difficult to diagnose and fix problems. The poor results led to the use of the Wasserstein GAN as a final attempt to achieve the desired solution. The same GAN architecture as Pix2Pix was used, but the discriminator was replaced with a critic, BatchNorm layers replaced with InstanceNorm, and the Pix2Pix loss was replaced with the Wasserstein loss. The resulting model is shown in Figure 4.11.

The results from the Wasserstein GAN experiments outperformed those of the Pix2Pix GAN, effectively addressing the issue of mode collapse (see Figure 5.10, Figure 5.12, Figure 5.15, and Figure 5.16), as the theory states in regards to Wasserstein GANs (section 2.7.2). When comparing the results of the Wasserstein GAN to those of the Pix2Pix GAN, it is clear that the Wasserstein GAN produces much better, more accurate and meaningful dots, without any signs of mode collapse. Each generated dot image was unique. This was not immediately apparent from the loss plot, as shown in Figure 5.13 for 512 × 512 images. It was, however, more evident in Figure 5.14 for 128 × 128 images, where the generator loss starts with large negative scores, but gradually improves and shows signs of convergence.

In the context of this study, the only way to determine whether the GANs had learned to produce dots on otoliths was to examine the resulting generated images. The quality of the generated images can be evaluated based on the accuracy of dot placement on the annuli, as well as the calculated age of the otoliths based on the number of dots. The use of loss as the sole performance metric for evaluating the ability of the GANs to produce high-quality images may not be accurate, as stated in subsection 2.7.1. This is evident from the loss plots, where a poor-looking loss plot may be accompanied by good results, and vice versa. Many of the loss plots during the experiments were hard to interpret and often not useful in regard to analyzing performance. The trend of learning and quality improvement of the generated images were clear regardless of the loss plots.

6.1.2 Mode Collapse

It is surprising and interesting that the Wasserstein GAN superiorly outperformed the Pix2Pix GAN, considering that both CNN architectures were built and implemented very similarly, only differing normalization layers (see subsection 2.8.1). As stated earlier, the primary issue with the Pix2Pix models was mode collapse, which was not observed in the results of the Wasserstein models. One potential explanation for this difference in performance is the relative strengths of Pix2Pix GANs as image-to-image translation frameworks.

Pix2Pix GANs are typically used in situations where the source and target images are different versions of the same underlying image. Many applications of Pix2Pix GANs include generating filled-in versions of object outlines, such as buildings. For example, the source image may be an outline of a building facade, while the target image is a fully rendered version of that building. In this case, the generator would learn to produce an accurate image of the building by using the outline of the building facade as input. Another example is using non-satellite images from Google Maps as the source, with the corresponding aerial satellite photos as the target. These examples can be seen in the original paper *Image-to-Image Translation with Conditional Adversarial Networks* by Isola et al. [38].

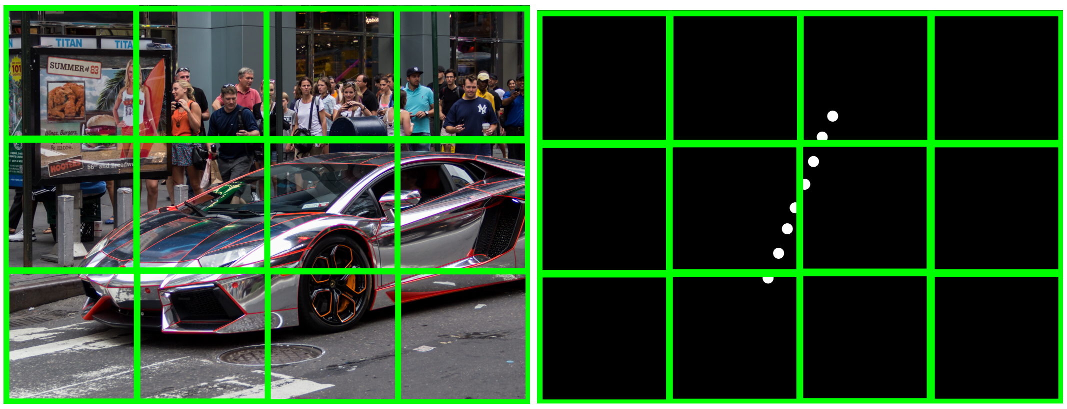
The key point is that Pix2Pix GANs may be best suited for scenarios where the source and target images are different versions of the same underlying image. In this study, however, the source images were otolith images, while the target images were completely different, being black images with white dots. The significant difference between the source and target images may have made it difficult for the Pix2Pix model to capture important features in the translation.

Another possible explanation for the superior performance of the Wasserstein GAN is the implementation of discrimination in the model. In Pix2Pix GANs, the discriminator classifies patches of the generated images as either true or false, whereas the critic in the Wasserstein GAN simply scores the quality of the generated images. Given that the dot images of the dataset were highly similar in structure, this strict binary classification of the Pix2Pix may have led to the mode collapse. The generator might learn to produce dots in a manner that is structurally acceptable enough for the discriminator, resulting in the discriminator often classifying them as correct, leading the generator to generate images that are largely identical. In contrast, the generator in the Wasserstein GAN does not have this opportunity, as the critic only provides a score and does not classify the generated images as strictly true or false. This way of discriminating the images may have forced the generator to try harder to produce higher-quality images by getting higher scores, leading to the observed lack of mode collapse and the generation of unique samples.

6.1.3 Patch vs. non-patch Architecture in the Discriminator/Critic

The performance of Pix2Pix GAN is highly dependent on the PatchGAN discriminator architecture and patch size. As shown in Table 4.5, a range of patch sizes were tested during the Pix2Pix experimentation. These included patch sizes close to 1×1 as well as patch sizes equal to the input image size. Results with patch sizes closer to 1×1 were generally poor, with the only learning signs being generation of darker or more black images resembling the general look of the target images. Larger patch sizes, closer to the input image size, resulted in even worse performance, with the generated images never showing any signs of dots or dark images resembling the target images. Further experimentation showed that patch sizes generally around 10% to 15% of the input image size produced the best results relative to the peak performance of the Pix2Pix GAN experiments.

In addition to being dependent on its patch architecture and size in its discriminator, Pix2Pix is also highly dependent on the source and target images used in the network. As previously discussed in subsection 6.1.2, the source and target images are usually different versions of the same underlying image in Pix2Pix applications, where in my case, the source and target images were completely different. The target images that the GAN's generator learned to produce were black images with white dots, which had a sparse structure. Only the areas with dots contained "important" information, while most of the image were black and empty, containing no information. As mentioned in subsection 2.6.2, in Pix2Pix, the input image for PatchGAN discriminators is divided into overlapping patches, which are processed independently. The patch-level probabilities are then combined and averaged to produce a final, image-level probability that indicates whether the image is real or fake. Given that the target images in the experiments mostly were "empty", may have caused issues in the classification, becoming a factor for the poor results of the Pix2Pix models. I will further explain using Figure 6.1.



(a) An information-rich image. **(b)** A target, dot image example.

Figure 6.1: Information-comparison of images. Image **(a)** showing an image containing several objects and a lot of information, while image **(b)** shows an example of a target, dot image, being mostly black. The green squares visualize different patches of the PatchGAN discriminator. Figure 6.1a taken from Rabich [59] and edited.

As seen in Figure 6.1a, the image is rich with information, including a street with a car, people, and commercial advertisements. Each patch, depicted by the green squares, contains distinct information making them differ from each other. The patch-level probabilities of this image are more evenly weighted in the final image-level probability that indicates whether the whole image is real or fake. In contrast, when examining Figure 6.1b for comparison, the dot images of the experiments contain valuable information primarily in the center of the images. If we examine the patches of this image, we can see that the majority of them are simply patches of "nothing," being black. The patch-level probabilities in this scenario would be biased towards these black patches as there are more of them, while the few patches containing white dots would not be as heavily weighted in the total image-level probability. This may have led the discriminator to conclude that the generated images were generally correct, given that they were mostly black, resulting in less attention being paid to the fine-grained details of the dots towards the center. This means that the generator in this case could easily learn to reproduce these black patches but may not have been as effective at reproducing the white dots, due to the unbalanced distribution of unique patches.

In the context of Pix2Pix, it would be an interesting experiment to implement a discriminator that prioritizes the probabilities of the patches towards the center of its input images, as this is where most of the information is located, assuming that the dots are always centered. By weighting these central patches and their probabilities more heavily than the outer patches, it may have an impact on performance and result in better generation of dots.

There is another interesting difference to be observed when examining the images in Figure 6.1. As previously discussed in the context of Figure 6.1a, there are numerous relevant structures present in the image. Each local patch contains sufficient information, structures, and details to the point where each patch in itself can be considered a standalone image. In this scenario, the Pix2Pix model is able to individually determine each of these patches *locally* without necessarily being dependent on the details of the rest of the image. In contrast, when using Figure 6.1b, the model only yields satisfactory results when it has a *global* overview of the entire image when determining the patches. This is because the entirety of the otolith is necessary for determining the appropriate number of dots. The importance of a global overview becomes apparent when considering a scenario in which dots are present in one patch, maybe these dots are continued into the next patch, but not in the subsequent patches, even though they *could* be present (negating the issue shown in Figure 5.20). This dependency on a global overview of the images may have been overlooked by the Pix2Pix models, resulting in a factor that impacted its performance. This may not have been an issue if the presented source and target images were different.

The use of a patch architecture was also investigated in the context of Wasserstein GANs, to determine its impact on performance. As shown in Table 4.6, experiment 3 used a patch architecture for the critic. During this experimentation, a non-patch critic was also tested with the same hyperparameters. However, the resulting model did not show any significant differences in performance. The experiment using the patch architecture was included in the list of relevant experiments, along with the final experiment where no patch architecture was used, to demonstrate that the performances of the Wasserstein GAN models were not sensitive to the use of a patch architecture in the same way that the Pix2Pix GAN models were.

6.1.4 Quality of the Generated Dots using Wasserstein GAN

There are two ways to evaluate the results of the successful experiments. The first is to assess the number of dots counted in each generated image and determine whether the generator has produced the correct amount. The second is to examine the resulting images and determine whether the dots generated by the generator are accurately placed on the otoliths' annuli. It is important to note that a correct count of dots does not necessarily imply accurate placement of those dots. Therefore, a comprehensive evaluation of the results must consider both the number and the placement of the dots.

Looking at the results in Figure 5.17 and Figure 5.18, we see that the Wasserstein models achieved an accuracy of 82.8% and 81.5%, with a ± 2 age difference, for the 512×512 and 128×128 images, respectively. These values are strictly from the algorithm that counts the *number* of dots and do not account for the quality of dot placement on the otoliths. To evaluate the accuracy of dot placement, we must mainly assess the results in a supervised manner by ourselves, unless further techniques are used to determine the accuracy. An example could be assessing the distance between each dot and determine whether they are placed in a line or in the same cluster, or other creative metrics that could help to determine the dot placement quality.

Examining the results in Figure 5.15, Figure 5.16 and in Appendix B, we can see that the majority of dots generated by the Wasserstein GANs are placed reasonably and accurately on the annuli of the otoliths. However, there are instances where the model has mistaken a check for an annuli, as shown in Figure B.1c, Figure B.1i and Figure B.1r. As discussed in subsection 1.2.1, an annuli is composed of both a narrow (check) and a wide band, and only represents one year's growth when these are considered together. The structural similarity between checks and annuli may have caused the model to place dots on the checks.

The quality of the dots did in general get better during the training, but even after the total epochs of the experiments, there were still signs of dot contours in the dot images that I personally would not count as a dot due to their total brightness. More training could lead to a model that would generate less non-bright dots and instead produce higher quality ones.

6.2 Limitations and Future Works

The main limitation of this work has been the access to resources and hardware. As presented in subsection 2.7.1, when working with GANs for image classification and generation, training requires lots of resources and often takes several hours to finish. Many of the training experiments, especially the ones that were run for 750 epochs, took over 25 hours to finish each. Considering the availability of the hardware, as presented in section 4.6, only the timeframes within the availability periods gave me partial access for experimentation with models. For future works, it is recommended to have full access to hardware and GPU's when experimenting. This way, multiple architectures can be experimented with, and in addition, more hyperparameter tuning can be applied. Having more time could lead to experimenting with even higher resolution images and the use of higher number of epochs to see whether there would be better convergence in the training.

The dataset used in this thesis consisted of 4096 samples, where these were split into training, validation and testing sets. In machine learning, the amount of data is important. The more, the better. In my case, to provide the models with extra samples, a few data augmentation techniques were applied, mainly including -90 and 90 degree rotations, horizontal flipping and a few tests with cropping. I recommend using even more augmentation techniques in future works. There are many augmentation techniques that can be applied, i.e. more range in the degrees of rotation, adding noise to the images and random cropping of the images. Implementing these may give the models more variety in samples, also expanding the samples in the relatively small dataset.

The dataset of otolith images, and their respective dot coordinates have provided valuable information for the models of the thesis. As seen in the real images and dot images, the dots are mostly placed in a straight line for every otoliths. The models have been able to catch the structural placements of the dots, and the testing results show that they have learned to place these dots in the same manner as in the original dataset. However, it would be interesting to see how the models can perform with a dataset consisting of more variety in dot placement on otoliths. Instead of having only dots in straight lines from the core to the shortest edge of the otoliths, the dots could for example be more scattered around the otolith. This way, the resulting dots could have more variety, and the samples of the dataset could be more dissimilar to each other. But then again, this could lead to a more difficult interpretation for a human observer, hence result in a less explainable approach.

The main experiment of the thesis was using Pix2Pix GANs in the pursuit of answering the research question. The results using this architecture did not show promising results within my timeframe of the thesis. This led to the experimentation of using Wasserstein GANs instead, which outperformed the Pix2Pix results in a superior way. Because of these results, and the discussion in subsection 6.1.2

and subsection 6.1.3, I recommend experimenting more using Wasserstein GANs in future works for these types of problems. The Wasserstein GANs did not struggle with mode collapse which the Pix2Pix GANs did, being the main problem of the original idea of using Pix2Pix in the envisioned solution.

Chapter 7

Conclusion

The aim of my thesis was to investigate the potential of using GANs to learn a generative model for annotating images of otoliths and to explore the feasibility of this approach as an explainable AI. To do this, I trained different GAN architectures on a dataset of processed otolith images and preprocessed annotations data, and evaluated the performance of the trained models on a held-out test set. I also analyzed the visual annotations produced by the models to understand how they were able to identify the growth rings in the otolith images.

The results of the experiments indicate that GANs can be applied effectively to image annotation tasks. I was able to develop models that were able to generate dots that were accurately placed on the annuli of otoliths. In addition, the models were able to count the dots and provide visual representations of the results, making them applicable as explainable AI's.

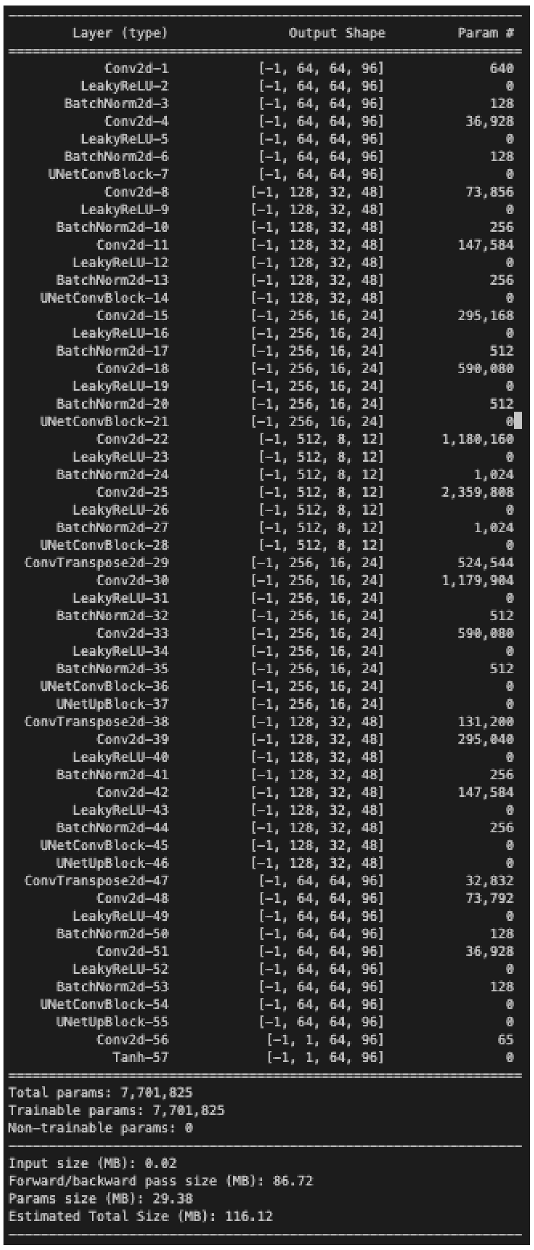
The experiments showed that Wasserstein GANs outperformed Pix2Pix GANs in this task, achieving an age accuracy of 82.8% on images of resolution 512×512, and 81.5% on images of resolution 128×128, including an age offset of ± 2. The quality of dot placement on otoliths was promising, but only human judgement can give the final accuracy evaluation of the placements.

I hope that this research will provide insights into the potential of GANs for learning generative models for image annotation tasks and give a better understanding of their potential as explainable AI's. By exploring various GAN architectures and training settings, and evaluating their effectiveness for this task, the research can contribute to the development of new GAN-based methods for annotating images of otoliths and other similar image annotation tasks.

Appendix A

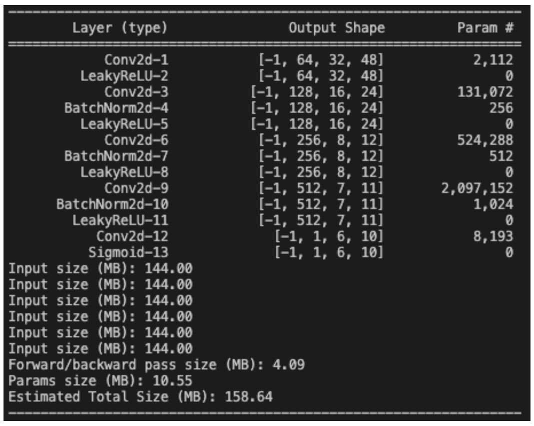
CNN Architectures

In this appendix, the CNN architectures of the different experiments described in this thesis (Table 4.6) are shown. The figures show both the generator and the discriminator/critic architectures. The key takeaways are the amount and type of layers in each architecture, and the output layers of the discriminators/critics. These output layers denote the size of the patches (if used) in the architectures. As seen in the last experiment (Figure A.4), no patch architecture was used in the experiment. These figures can also be used as a source for reproducibility purposes.



```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 64, 64, 96]             640
         LeakyReLU-2           [-1, 64, 64, 96]               0
       BatchNorm2d-3           [-1, 64, 64, 96]             128
            Conv2d-4           [-1, 64, 64, 96]          36,928
         LeakyReLU-5           [-1, 64, 64, 96]               0
       BatchNorm2d-6           [-1, 64, 64, 96]             128
     UNetConvBlock-7           [-1, 64, 64, 96]               0
            Conv2d-8          [-1, 128, 32, 48]          73,856
         LeakyReLU-9          [-1, 128, 32, 48]               0
      BatchNorm2d-10          [-1, 128, 32, 48]             256
           Conv2d-11          [-1, 128, 32, 48]         147,584
        LeakyReLU-12          [-1, 128, 32, 48]               0
      BatchNorm2d-13          [-1, 128, 32, 48]             256
    UNetConvBlock-14          [-1, 128, 32, 48]               0
           Conv2d-15          [-1, 256, 16, 24]         295,168
        LeakyReLU-16          [-1, 256, 16, 24]               0
      BatchNorm2d-17          [-1, 256, 16, 24]             512
           Conv2d-18          [-1, 256, 16, 24]         590,080
        LeakyReLU-19          [-1, 256, 16, 24]               0
      BatchNorm2d-20          [-1, 256, 16, 24]             512
    UNetConvBlock-21          [-1, 256, 16, 24]               0
           Conv2d-22           [-1, 512, 8, 12]       1,180,160
        LeakyReLU-23           [-1, 512, 8, 12]               0
      BatchNorm2d-24           [-1, 512, 8, 12]           1,024
           Conv2d-25           [-1, 512, 8, 12]       2,359,808
        LeakyReLU-26           [-1, 512, 8, 12]               0
      BatchNorm2d-27           [-1, 512, 8, 12]           1,024
    UNetConvBlock-28           [-1, 512, 8, 12]               0
  ConvTranspose2d-29          [-1, 256, 16, 24]         524,544
           Conv2d-30          [-1, 256, 16, 24]       1,179,904
        LeakyReLU-31          [-1, 256, 16, 24]               0
      BatchNorm2d-32          [-1, 256, 16, 24]             512
           Conv2d-33          [-1, 256, 16, 24]         590,080
        LeakyReLU-34          [-1, 256, 16, 24]               0
      BatchNorm2d-35          [-1, 256, 16, 24]             512
    UNetConvBlock-36          [-1, 256, 16, 24]               0
      UNetUpBlock-37          [-1, 256, 16, 24]               0
  ConvTranspose2d-38          [-1, 128, 32, 48]         131,200
           Conv2d-39          [-1, 128, 32, 48]         295,040
        LeakyReLU-40          [-1, 128, 32, 48]               0
      BatchNorm2d-41          [-1, 128, 32, 48]             256
           Conv2d-42          [-1, 128, 32, 48]         147,584
        LeakyReLU-43          [-1, 128, 32, 48]               0
      BatchNorm2d-44          [-1, 128, 32, 48]             256
    UNetConvBlock-45          [-1, 128, 32, 48]               0
      UNetUpBlock-46          [-1, 128, 32, 48]               0
  ConvTranspose2d-47           [-1, 64, 64, 96]          32,832
           Conv2d-48           [-1, 64, 64, 96]          73,792
        LeakyReLU-49           [-1, 64, 64, 96]               0
      BatchNorm2d-50           [-1, 64, 64, 96]             128
           Conv2d-51           [-1, 64, 64, 96]          36,928
        LeakyReLU-52           [-1, 64, 64, 96]               0
      BatchNorm2d-53           [-1, 64, 64, 96]             128
    UNetConvBlock-54           [-1, 64, 64, 96]               0
      UNetUpBlock-55           [-1, 64, 64, 96]               0
           Conv2d-56            [-1, 1, 64, 96]              65
             Tanh-57            [-1, 1, 64, 96]               0
================================================================
Total params: 7,701,825
Trainable params: 7,701,825
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.02
Forward/backward pass size (MB): 86.72
Params size (MB): 29.38
Estimated Total Size (MB): 116.12
----------------------------------------------------------------
```

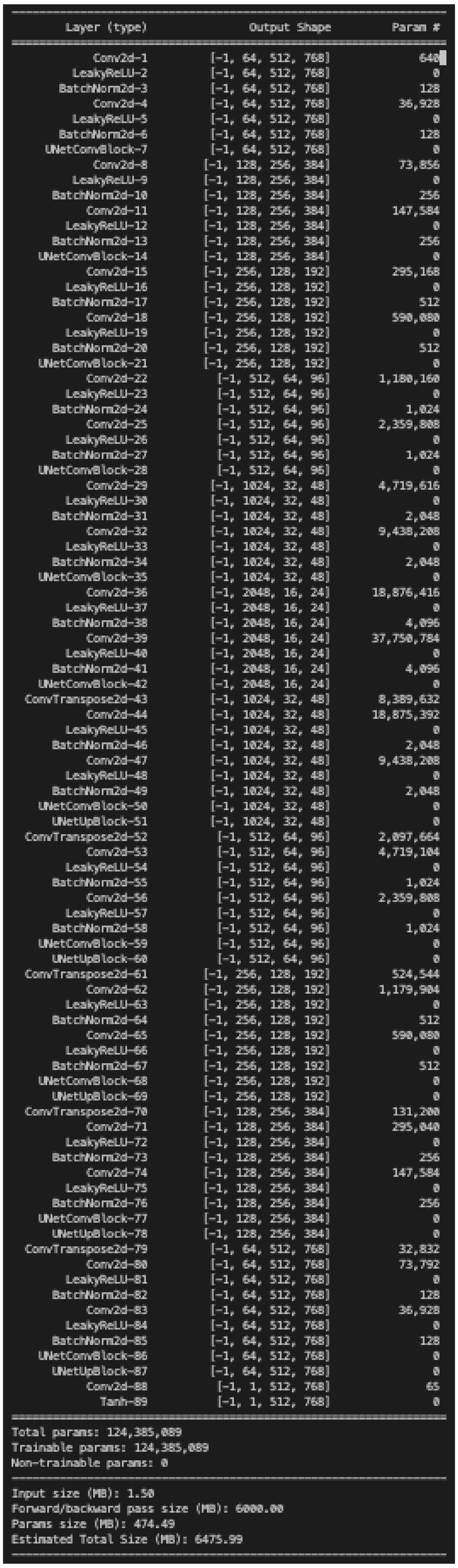
(a) Generator



```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 64, 32, 48]           2,112
         LeakyReLU-2           [-1, 64, 32, 48]               0
            Conv2d-3          [-1, 128, 16, 24]         131,072
       BatchNorm2d-4          [-1, 128, 16, 24]             256
         LeakyReLU-5          [-1, 128, 16, 24]               0
            Conv2d-6           [-1, 256, 8, 12]         524,288
       BatchNorm2d-7           [-1, 256, 8, 12]             512
         LeakyReLU-8           [-1, 256, 8, 12]               0
            Conv2d-9           [-1, 512, 7, 11]       2,097,152
      BatchNorm2d-10           [-1, 512, 7, 11]           1,024
        LeakyReLU-11           [-1, 512, 7, 11]               0
           Conv2d-12             [-1, 1, 6, 10]           8,193
          Sigmoid-13             [-1, 1, 6, 10]               0
Input size (MB): 144.00
Input size (MB): 144.00
Input size (MB): 144.00
Input size (MB): 144.00
Input size (MB): 144.00
Input size (MB): 144.00
Forward/backward pass size (MB): 4.09
Params size (MB): 10.55
Estimated Total Size (MB): 158.64
----------------------------------------------------------------
```

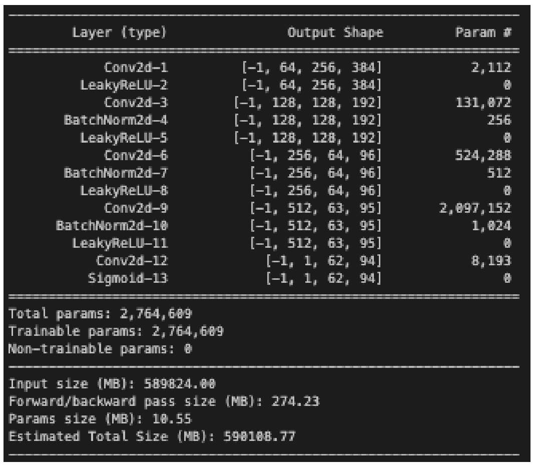
(b) Discriminator

Figure A.1: Pix2Pix GAN CNN Architecture for experiment 1 in Table 4.6.



```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1         [-1, 64, 512, 768]             640
         LeakyReLU-2         [-1, 64, 512, 768]               0
       BatchNorm2d-3         [-1, 64, 512, 768]             128
            Conv2d-4         [-1, 64, 512, 768]          36,928
         LeakyReLU-5         [-1, 64, 512, 768]               0
       BatchNorm2d-6         [-1, 64, 512, 768]             128
     UNetConvBlock-7         [-1, 64, 512, 768]               0
            Conv2d-8        [-1, 128, 256, 384]          73,856
         LeakyReLU-9        [-1, 128, 256, 384]               0
      BatchNorm2d-10        [-1, 128, 256, 384]             256
           Conv2d-11        [-1, 128, 256, 384]         147,584
        LeakyReLU-12        [-1, 128, 256, 384]               0
      BatchNorm2d-13        [-1, 128, 256, 384]             256
    UNetConvBlock-14        [-1, 128, 256, 384]               0
           Conv2d-15        [-1, 256, 128, 192]         295,168
        LeakyReLU-16        [-1, 256, 128, 192]               0
      BatchNorm2d-17        [-1, 256, 128, 192]             512
           Conv2d-18        [-1, 256, 128, 192]         590,080
        LeakyReLU-19        [-1, 256, 128, 192]               0
      BatchNorm2d-20        [-1, 256, 128, 192]             512
    UNetConvBlock-21        [-1, 256, 128, 192]               0
           Conv2d-22          [-1, 512, 64, 96]       1,180,160
        LeakyReLU-23          [-1, 512, 64, 96]               0
      BatchNorm2d-24          [-1, 512, 64, 96]           1,024
           Conv2d-25          [-1, 512, 64, 96]       2,359,808
        LeakyReLU-26          [-1, 512, 64, 96]               0
      BatchNorm2d-27          [-1, 512, 64, 96]           1,024
    UNetConvBlock-28          [-1, 512, 64, 96]               0
           Conv2d-29         [-1, 1024, 32, 48]       4,719,616
        LeakyReLU-30         [-1, 1024, 32, 48]               0
      BatchNorm2d-31         [-1, 1024, 32, 48]           2,048
           Conv2d-32         [-1, 1024, 32, 48]       9,438,208
        LeakyReLU-33         [-1, 1024, 32, 48]               0
      BatchNorm2d-34         [-1, 1024, 32, 48]           2,048
    UNetConvBlock-35         [-1, 1024, 32, 48]               0
           Conv2d-36         [-1, 2048, 16, 24]      18,876,416
        LeakyReLU-37         [-1, 2048, 16, 24]               0
      BatchNorm2d-38         [-1, 2048, 16, 24]           4,096
           Conv2d-39         [-1, 2048, 16, 24]      37,750,784
        LeakyReLU-40         [-1, 2048, 16, 24]               0
      BatchNorm2d-41         [-1, 2048, 16, 24]           4,096
    UNetConvBlock-42         [-1, 2048, 16, 24]               0
  ConvTranspose2d-43         [-1, 1024, 32, 48]       8,389,632
           Conv2d-44         [-1, 1024, 32, 48]      18,875,392
        LeakyReLU-45         [-1, 1024, 32, 48]               0
      BatchNorm2d-46         [-1, 1024, 32, 48]           2,048
           Conv2d-47         [-1, 1024, 32, 48]       9,438,208
        LeakyReLU-48         [-1, 1024, 32, 48]               0
      BatchNorm2d-49         [-1, 1024, 32, 48]           2,048
    UNetConvBlock-50         [-1, 1024, 32, 48]               0
      UNetUpBlock-51         [-1, 1024, 32, 48]               0
  ConvTranspose2d-52          [-1, 512, 64, 96]       2,097,664
           Conv2d-53          [-1, 512, 64, 96]       4,719,104
        LeakyReLU-54          [-1, 512, 64, 96]               0
      BatchNorm2d-55          [-1, 512, 64, 96]           1,024
           Conv2d-56          [-1, 512, 64, 96]       2,359,808
        LeakyReLU-57          [-1, 512, 64, 96]               0
      BatchNorm2d-58          [-1, 512, 64, 96]           1,024
    UNetConvBlock-59          [-1, 512, 64, 96]               0
      UNetUpBlock-60          [-1, 512, 64, 96]               0
  ConvTranspose2d-61        [-1, 256, 128, 192]         524,544
           Conv2d-62        [-1, 256, 128, 192]       1,179,904
        LeakyReLU-63        [-1, 256, 128, 192]               0
      BatchNorm2d-64        [-1, 256, 128, 192]             512
           Conv2d-65        [-1, 256, 128, 192]         590,080
        LeakyReLU-66        [-1, 256, 128, 192]               0
      BatchNorm2d-67        [-1, 256, 128, 192]             512
    UNetConvBlock-68        [-1, 256, 128, 192]               0
      UNetUpBlock-69        [-1, 256, 128, 192]               0
  ConvTranspose2d-70        [-1, 128, 256, 384]         131,200
           Conv2d-71        [-1, 128, 256, 384]         295,040
        LeakyReLU-72        [-1, 128, 256, 384]               0
      BatchNorm2d-73        [-1, 128, 256, 384]             256
           Conv2d-74        [-1, 128, 256, 384]         147,584
        LeakyReLU-75        [-1, 128, 256, 384]               0
      BatchNorm2d-76        [-1, 128, 256, 384]             256
    UNetConvBlock-77        [-1, 128, 256, 384]               0
      UNetUpBlock-78        [-1, 128, 256, 384]               0
  ConvTranspose2d-79         [-1, 64, 512, 768]          32,832
           Conv2d-80         [-1, 64, 512, 768]          73,792
        LeakyReLU-81         [-1, 64, 512, 768]               0
      BatchNorm2d-82         [-1, 64, 512, 768]             128
           Conv2d-83         [-1, 64, 512, 768]          36,928
        LeakyReLU-84         [-1, 64, 512, 768]               0
      BatchNorm2d-85         [-1, 64, 512, 768]             128
    UNetConvBlock-86         [-1, 64, 512, 768]               0
      UNetUpBlock-87         [-1, 64, 512, 768]               0
           Conv2d-88          [-1, 1, 512, 768]              65
             Tanh-89          [-1, 1, 512, 768]               0
================================================================
Total params: 124,385,089
Trainable params: 124,385,089
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 1.50
Forward/backward pass size (MB): 6000.00
Params size (MB): 474.49
Estimated Total Size (MB): 6475.99
----------------------------------------------------------------
```

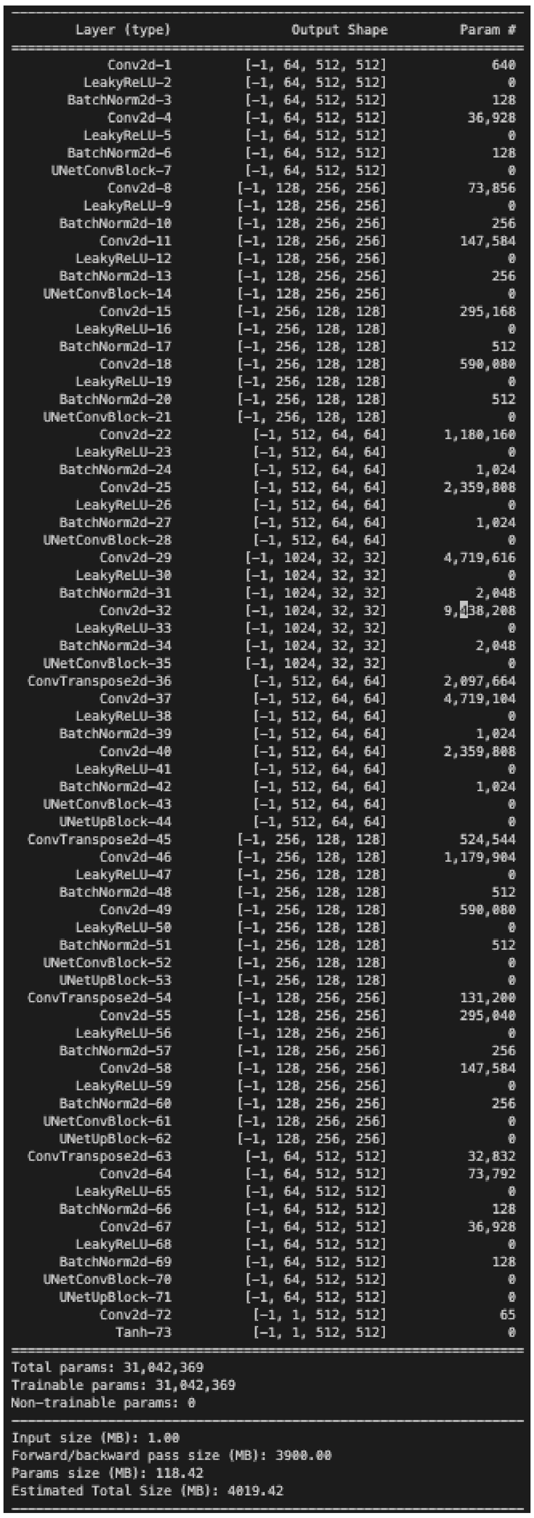
(a) Generator



```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1         [-1, 64, 256, 384]           2,112
         LeakyReLU-2         [-1, 64, 256, 384]               0
            Conv2d-3        [-1, 128, 128, 192]         131,072
       BatchNorm2d-4        [-1, 128, 128, 192]             256
         LeakyReLU-5        [-1, 128, 128, 192]               0
            Conv2d-6          [-1, 256, 64, 96]         524,288
       BatchNorm2d-7          [-1, 256, 64, 96]             512
         LeakyReLU-8          [-1, 256, 64, 96]               0
            Conv2d-9          [-1, 512, 63, 95]       2,097,152
      BatchNorm2d-10          [-1, 512, 63, 95]           1,024
        LeakyReLU-11          [-1, 512, 63, 95]               0
           Conv2d-12            [-1, 1, 62, 94]           8,193
          Sigmoid-13            [-1, 1, 62, 94]               0
================================================================
Total params: 2,764,609
Trainable params: 2,764,609
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 589824.00
Forward/backward pass size (MB): 274.23
Params size (MB): 10.55
Estimated Total Size (MB): 590108.77
----------------------------------------------------------------
```

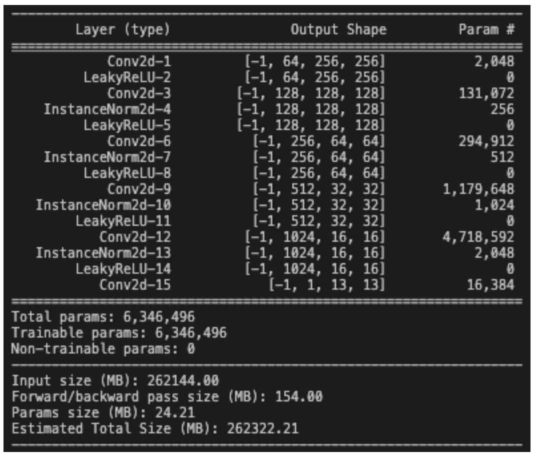
(b) Discriminator

Figure A.2: Pix2Pix GAN CNN Architecture for experiment 2 in Table 4.6.



```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1         [-1, 64, 512, 512]             640
         LeakyReLU-2         [-1, 64, 512, 512]               0
       BatchNorm2d-3         [-1, 64, 512, 512]             128
            Conv2d-4         [-1, 64, 512, 512]          36,928
         LeakyReLU-5         [-1, 64, 512, 512]               0
       BatchNorm2d-6         [-1, 64, 512, 512]             128
     UNetConvBlock-7         [-1, 64, 512, 512]               0
            Conv2d-8        [-1, 128, 256, 256]          73,856
         LeakyReLU-9        [-1, 128, 256, 256]               0
      BatchNorm2d-10        [-1, 128, 256, 256]             256
           Conv2d-11        [-1, 128, 256, 256]         147,584
        LeakyReLU-12        [-1, 128, 256, 256]               0
      BatchNorm2d-13        [-1, 128, 256, 256]             256
    UNetConvBlock-14        [-1, 128, 256, 256]               0
           Conv2d-15        [-1, 256, 128, 128]         295,168
        LeakyReLU-16        [-1, 256, 128, 128]               0
      BatchNorm2d-17        [-1, 256, 128, 128]             512
           Conv2d-18        [-1, 256, 128, 128]         590,080
        LeakyReLU-19        [-1, 256, 128, 128]               0
      BatchNorm2d-20        [-1, 256, 128, 128]             512
    UNetConvBlock-21        [-1, 256, 128, 128]               0
           Conv2d-22          [-1, 512, 64, 64]       1,180,160
        LeakyReLU-23          [-1, 512, 64, 64]               0
      BatchNorm2d-24          [-1, 512, 64, 64]           1,024
           Conv2d-25          [-1, 512, 64, 64]       2,359,808
        LeakyReLU-26          [-1, 512, 64, 64]               0
      BatchNorm2d-27          [-1, 512, 64, 64]           1,024
    UNetConvBlock-28          [-1, 512, 64, 64]               0
           Conv2d-29         [-1, 1024, 32, 32]       4,719,616
        LeakyReLU-30         [-1, 1024, 32, 32]               0
      BatchNorm2d-31         [-1, 1024, 32, 32]           2,048
           Conv2d-32         [-1, 1024, 32, 32]       9,438,208
        LeakyReLU-33         [-1, 1024, 32, 32]               0
      BatchNorm2d-34         [-1, 1024, 32, 32]           2,048
    UNetConvBlock-35         [-1, 1024, 32, 32]               0
  ConvTranspose2d-36          [-1, 512, 64, 64]       2,097,664
           Conv2d-37          [-1, 512, 64, 64]       4,719,104
        LeakyReLU-38          [-1, 512, 64, 64]               0
      BatchNorm2d-39          [-1, 512, 64, 64]           1,024
           Conv2d-40          [-1, 512, 64, 64]       2,359,808
        LeakyReLU-41          [-1, 512, 64, 64]               0
      BatchNorm2d-42          [-1, 512, 64, 64]           1,024
    UNetConvBlock-43          [-1, 512, 64, 64]               0
      UNetUpBlock-44          [-1, 512, 64, 64]               0
  ConvTranspose2d-45        [-1, 256, 128, 128]         524,544
           Conv2d-46        [-1, 256, 128, 128]       1,179,904
        LeakyReLU-47        [-1, 256, 128, 128]               0
      BatchNorm2d-48        [-1, 256, 128, 128]             512
           Conv2d-49        [-1, 256, 128, 128]         590,080
        LeakyReLU-50        [-1, 256, 128, 128]               0
      BatchNorm2d-51        [-1, 256, 128, 128]             512
    UNetConvBlock-52        [-1, 256, 128, 128]               0
      UNetUpBlock-53        [-1, 256, 128, 128]               0
  ConvTranspose2d-54        [-1, 128, 256, 256]         131,200
           Conv2d-55        [-1, 128, 256, 256]         295,040
        LeakyReLU-56        [-1, 128, 256, 256]               0
      BatchNorm2d-57        [-1, 128, 256, 256]             256
           Conv2d-58        [-1, 128, 256, 256]         147,584
        LeakyReLU-59        [-1, 128, 256, 256]               0
      BatchNorm2d-60        [-1, 128, 256, 256]             256
    UNetConvBlock-61        [-1, 128, 256, 256]               0
      UNetUpBlock-62        [-1, 128, 256, 256]               0
  ConvTranspose2d-63         [-1, 64, 512, 512]          32,832
           Conv2d-64         [-1, 64, 512, 512]          73,792
        LeakyReLU-65         [-1, 64, 512, 512]               0
      BatchNorm2d-66         [-1, 64, 512, 512]             128
           Conv2d-67         [-1, 64, 512, 512]          36,928
        LeakyReLU-68         [-1, 64, 512, 512]               0
      BatchNorm2d-69         [-1, 64, 512, 512]             128
    UNetConvBlock-70         [-1, 64, 512, 512]               0
      UNetUpBlock-71         [-1, 64, 512, 512]               0
           Conv2d-72          [-1, 1, 512, 512]              65
             Tanh-73          [-1, 1, 512, 512]               0
================================================================
Total params: 31,042,369
Trainable params: 31,042,369
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 1.00
Forward/backward pass size (MB): 3900.00
Params size (MB): 118.42
Estimated Total Size (MB): 4019.42
----------------------------------------------------------------
```

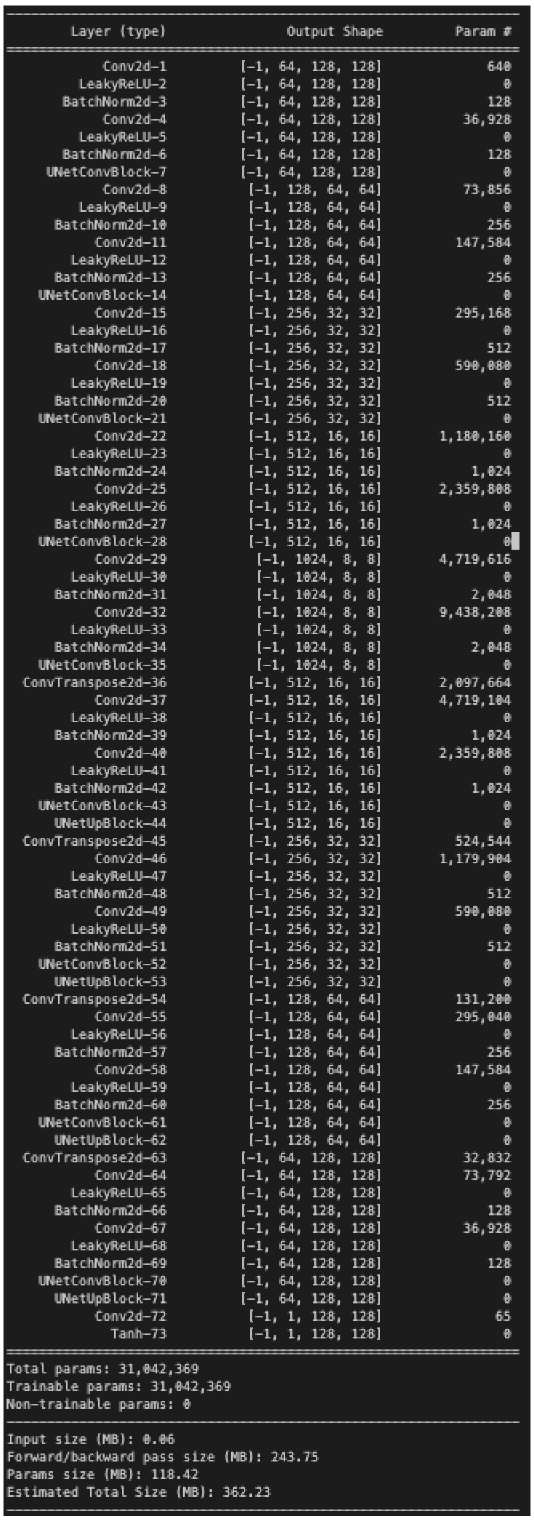
(a) Generator



```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1         [-1, 64, 256, 256]           2,048
         LeakyReLU-2         [-1, 64, 256, 256]               0
            Conv2d-3        [-1, 128, 128, 128]         131,072
    InstanceNorm2d-4        [-1, 128, 128, 128]             256
         LeakyReLU-5        [-1, 128, 128, 128]               0
            Conv2d-6          [-1, 256, 64, 64]         294,912
    InstanceNorm2d-7          [-1, 256, 64, 64]             512
         LeakyReLU-8          [-1, 256, 64, 64]               0
            Conv2d-9          [-1, 512, 32, 32]       1,179,648
   InstanceNorm2d-10          [-1, 512, 32, 32]           1,024
        LeakyReLU-11          [-1, 512, 32, 32]               0
           Conv2d-12         [-1, 1024, 16, 16]       4,718,592
   InstanceNorm2d-13         [-1, 1024, 16, 16]           2,048
        LeakyReLU-14         [-1, 1024, 16, 16]               0
           Conv2d-15            [-1, 1, 13, 13]          16,384
================================================================
Total params: 6,346,496
Trainable params: 6,346,496
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 262144.00
Forward/backward pass size (MB): 154.00
Params size (MB): 24.21
Estimated Total Size (MB): 262322.21
----------------------------------------------------------------
```

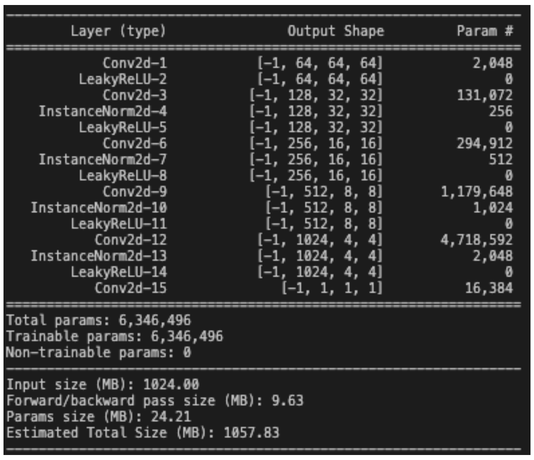
(b) Critic

Figure A.3: Wasserstein GAN CNN Architecture for experiment 3 in Table 4.6.



```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1        [-1, 64, 128, 128]             640
         LeakyReLU-2        [-1, 64, 128, 128]               0
       BatchNorm2d-3        [-1, 64, 128, 128]             128
            Conv2d-4        [-1, 64, 128, 128]          36,928
         LeakyReLU-5        [-1, 64, 128, 128]               0
       BatchNorm2d-6        [-1, 64, 128, 128]             128
     UNetConvBlock-7        [-1, 64, 128, 128]               0
            Conv2d-8         [-1, 128, 64, 64]          73,856
         LeakyReLU-9         [-1, 128, 64, 64]               0
      BatchNorm2d-10         [-1, 128, 64, 64]             256
           Conv2d-11         [-1, 128, 64, 64]         147,584
        LeakyReLU-12         [-1, 128, 64, 64]               0
      BatchNorm2d-13         [-1, 128, 64, 64]             256
    UNetConvBlock-14         [-1, 128, 64, 64]               0
           Conv2d-15         [-1, 256, 32, 32]         295,168
        LeakyReLU-16         [-1, 256, 32, 32]               0
      BatchNorm2d-17         [-1, 256, 32, 32]             512
           Conv2d-18         [-1, 256, 32, 32]         590,080
        LeakyReLU-19         [-1, 256, 32, 32]               0
      BatchNorm2d-20         [-1, 256, 32, 32]             512
    UNetConvBlock-21         [-1, 256, 32, 32]               0
           Conv2d-22         [-1, 512, 16, 16]       1,180,160
        LeakyReLU-23         [-1, 512, 16, 16]               0
      BatchNorm2d-24         [-1, 512, 16, 16]           1,024
           Conv2d-25         [-1, 512, 16, 16]       2,359,808
        LeakyReLU-26         [-1, 512, 16, 16]               0
      BatchNorm2d-27         [-1, 512, 16, 16]           1,024
    UNetConvBlock-28         [-1, 512, 16, 16]               0
           Conv2d-29          [-1, 1024, 8, 8]       4,719,616
        LeakyReLU-30          [-1, 1024, 8, 8]               0
      BatchNorm2d-31          [-1, 1024, 8, 8]           2,048
           Conv2d-32          [-1, 1024, 8, 8]       9,438,208
        LeakyReLU-33          [-1, 1024, 8, 8]               0
      BatchNorm2d-34          [-1, 1024, 8, 8]           2,048
    UNetConvBlock-35          [-1, 1024, 8, 8]               0
  ConvTranspose2d-36         [-1, 512, 16, 16]       2,097,664
           Conv2d-37         [-1, 512, 16, 16]       4,719,104
        LeakyReLU-38         [-1, 512, 16, 16]               0
      BatchNorm2d-39         [-1, 512, 16, 16]           1,024
           Conv2d-40         [-1, 512, 16, 16]       2,359,808
        LeakyReLU-41         [-1, 512, 16, 16]               0
      BatchNorm2d-42         [-1, 512, 16, 16]           1,024
    UNetConvBlock-43         [-1, 512, 16, 16]               0
      UNetUpBlock-44         [-1, 512, 16, 16]               0
  ConvTranspose2d-45         [-1, 256, 32, 32]         524,544
           Conv2d-46         [-1, 256, 32, 32]       1,179,904
        LeakyReLU-47         [-1, 256, 32, 32]               0
      BatchNorm2d-48         [-1, 256, 32, 32]             512
           Conv2d-49         [-1, 256, 32, 32]         590,080
        LeakyReLU-50         [-1, 256, 32, 32]               0
      BatchNorm2d-51         [-1, 256, 32, 32]             512
    UNetConvBlock-52         [-1, 256, 32, 32]               0
      UNetUpBlock-53         [-1, 256, 32, 32]               0
  ConvTranspose2d-54         [-1, 128, 64, 64]         131,200
           Conv2d-55         [-1, 128, 64, 64]         295,040
        LeakyReLU-56         [-1, 128, 64, 64]               0
      BatchNorm2d-57         [-1, 128, 64, 64]             256
           Conv2d-58         [-1, 128, 64, 64]         147,584
        LeakyReLU-59         [-1, 128, 64, 64]               0
      BatchNorm2d-60         [-1, 128, 64, 64]             256
    UNetConvBlock-61         [-1, 128, 64, 64]               0
      UNetUpBlock-62         [-1, 128, 64, 64]               0
  ConvTranspose2d-63        [-1, 64, 128, 128]          32,832
           Conv2d-64        [-1, 64, 128, 128]          73,792
        LeakyReLU-65        [-1, 64, 128, 128]               0
      BatchNorm2d-66        [-1, 64, 128, 128]             128
           Conv2d-67        [-1, 64, 128, 128]          36,928
        LeakyReLU-68        [-1, 64, 128, 128]               0
      BatchNorm2d-69        [-1, 64, 128, 128]             128
    UNetConvBlock-70        [-1, 64, 128, 128]               0
      UNetUpBlock-71        [-1, 64, 128, 128]               0
           Conv2d-72         [-1, 1, 128, 128]              65
             Tanh-73         [-1, 1, 128, 128]               0
================================================================
Total params: 31,042,369
Trainable params: 31,042,369
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.06
Forward/backward pass size (MB): 243.75
Params size (MB): 118.42
Estimated Total Size (MB): 362.23
----------------------------------------------------------------
```

(a) Generator



```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 64, 64, 64]           2,048
         LeakyReLU-2           [-1, 64, 64, 64]               0
            Conv2d-3          [-1, 128, 32, 32]         131,072
    InstanceNorm2d-4          [-1, 128, 32, 32]             256
         LeakyReLU-5          [-1, 128, 32, 32]               0
            Conv2d-6          [-1, 256, 16, 16]         294,912
    InstanceNorm2d-7          [-1, 256, 16, 16]             512
         LeakyReLU-8          [-1, 256, 16, 16]               0
            Conv2d-9            [-1, 512, 8, 8]       1,179,648
   InstanceNorm2d-10            [-1, 512, 8, 8]           1,024
        LeakyReLU-11            [-1, 512, 8, 8]               0
           Conv2d-12           [-1, 1024, 4, 4]       4,718,592
   InstanceNorm2d-13           [-1, 1024, 4, 4]           2,048
        LeakyReLU-14           [-1, 1024, 4, 4]               0
           Conv2d-15              [-1, 1, 1, 1]          16,384
================================================================
Total params: 6,346,496
Trainable params: 6,346,496
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 1024.00
Forward/backward pass size (MB): 9.63
Params size (MB): 24.21
Estimated Total Size (MB): 1057.83
----------------------------------------------------------------
```

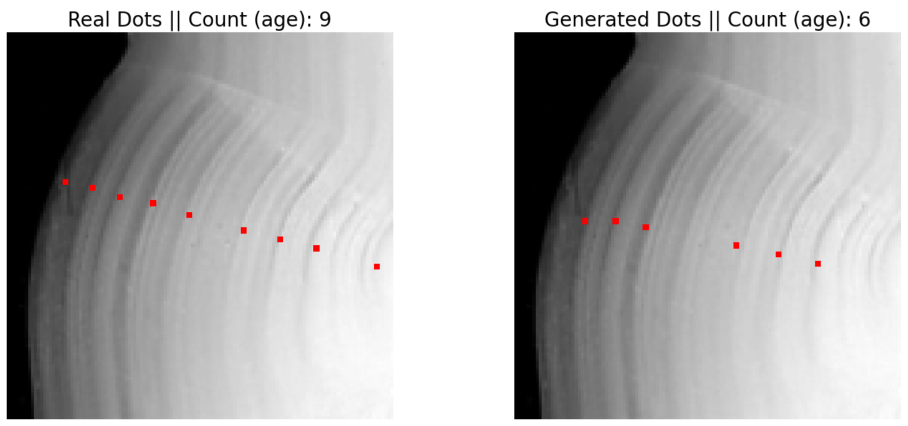
(b) Critic

Figure A.4: Wasserstein GAN CNN Architecture for experiment 4 in Table 4.6.

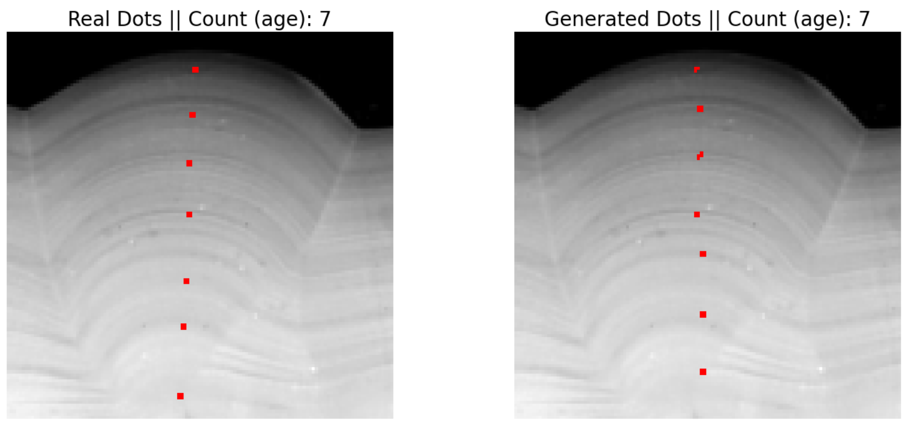
Appendix B

Testing Results

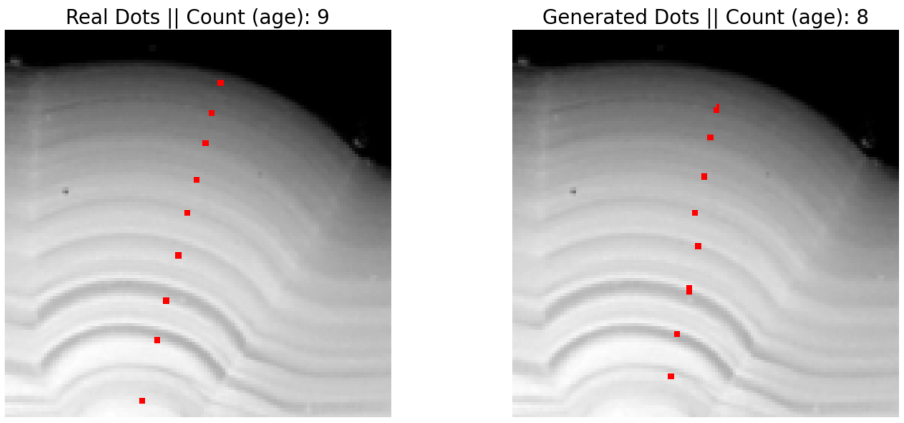
In this appendix, additional results from the last two successful experiments from Table 4.6 are shown. The last nine figures also includes the raw generated dot images in the middle column.



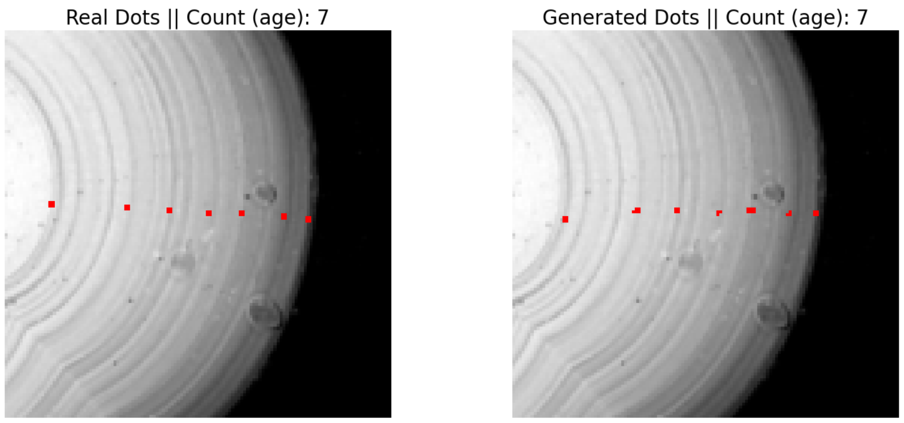
(a)



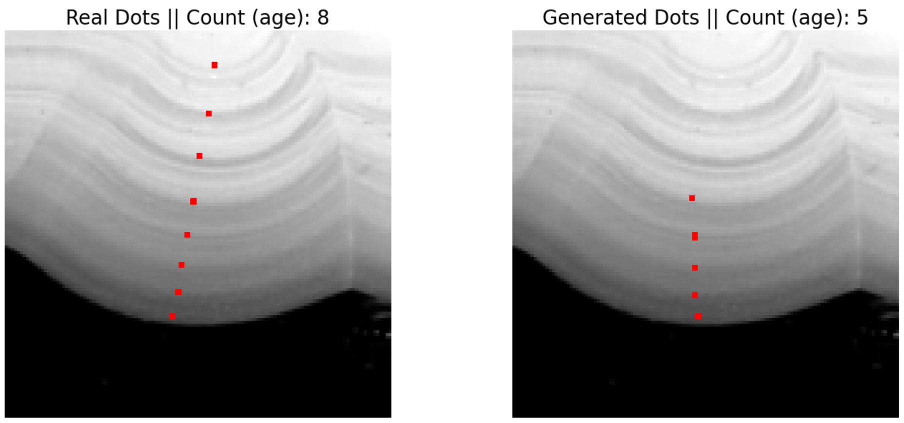
(b)



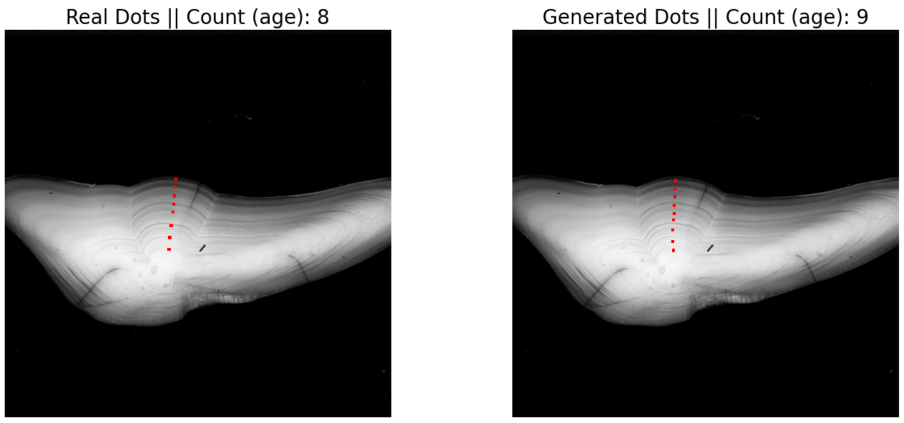
(c) Example of a result where a dot is placed on a check, as seen as the first dot counting from the core of the otolith.



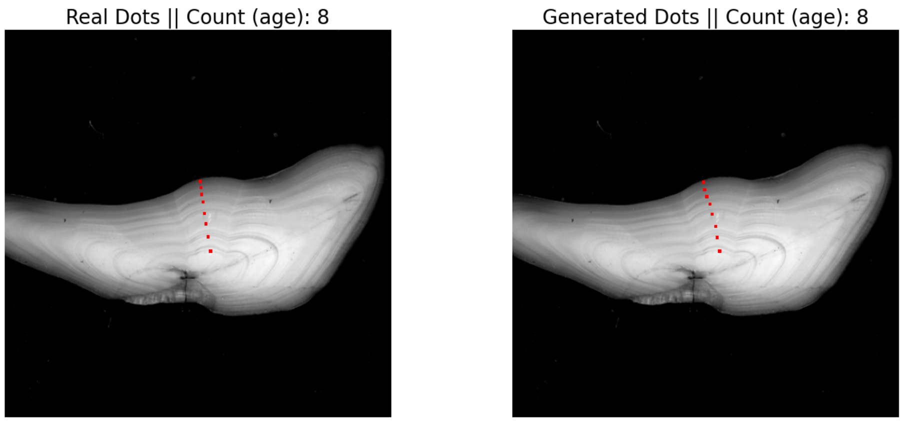
(d)



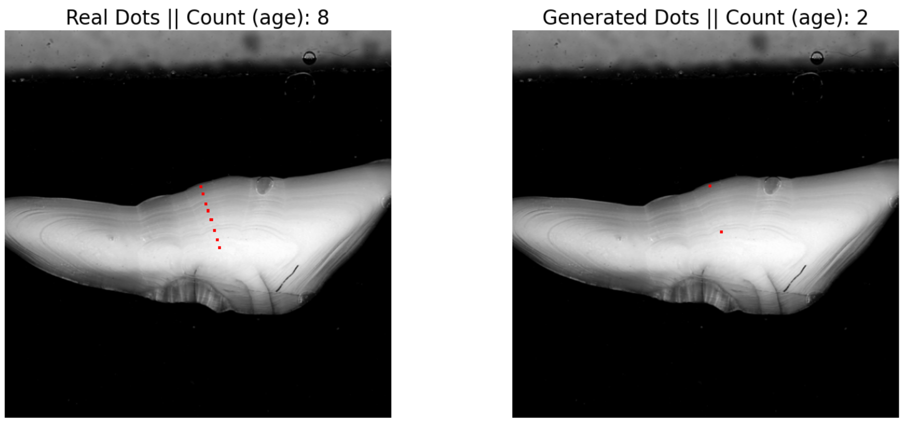
(e)


Real Dots || Count (age): 8
Generated Dots || Count (age): 9

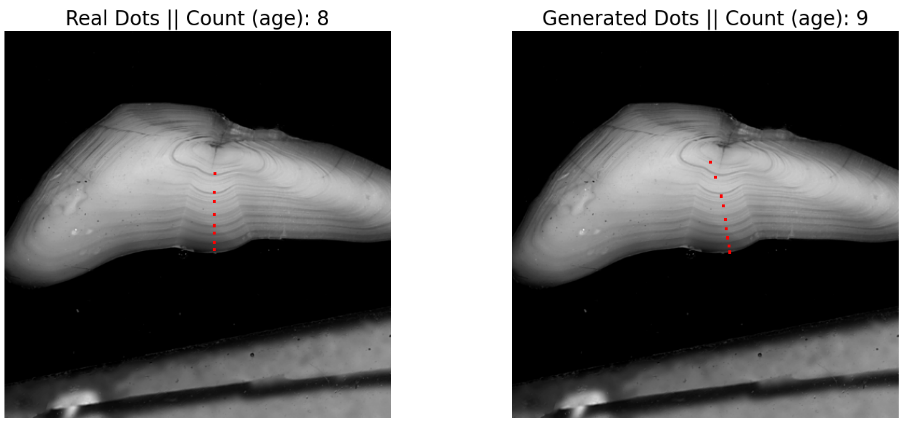
(f)


Real Dots || Count (age): 8
Generated Dots || Count (age): 8

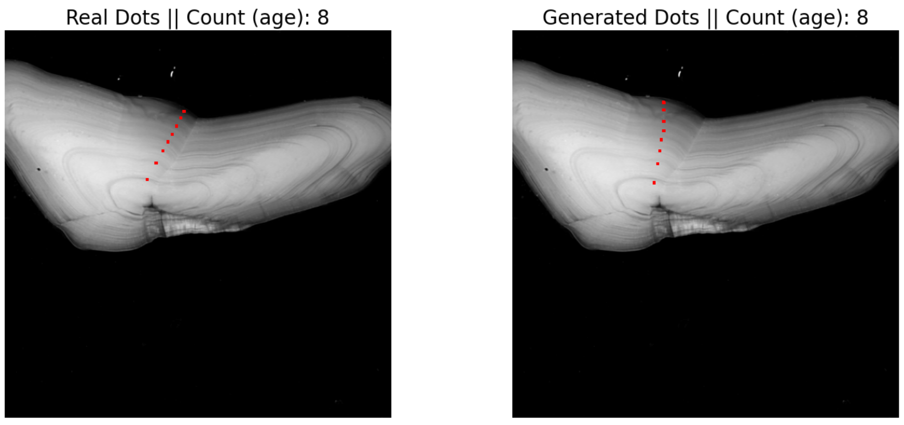
(g)


Real Dots || Count (age): 8
Generated Dots || Count (age): 2

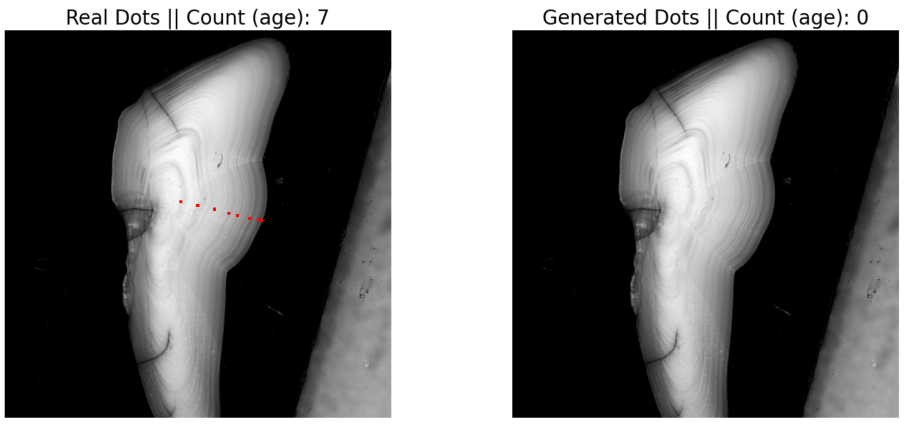
(h)



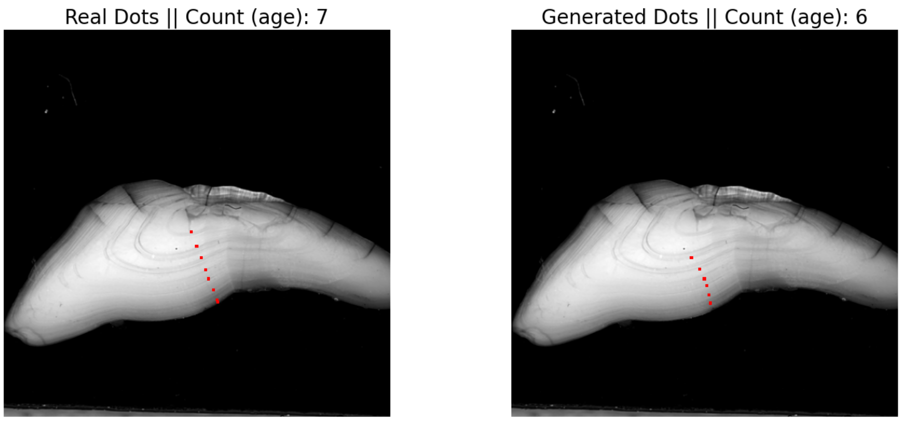
(i) Example of a result where a dot is placed on a check, as seen as the first dot counting from the core of the otolith.



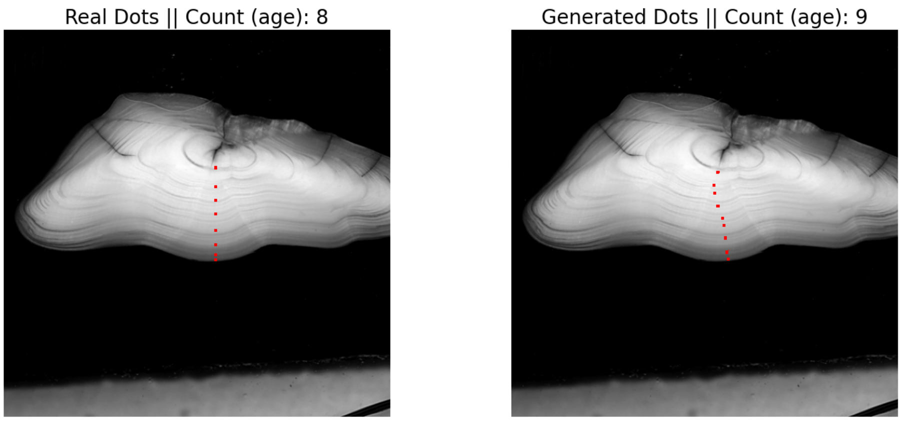
(j)



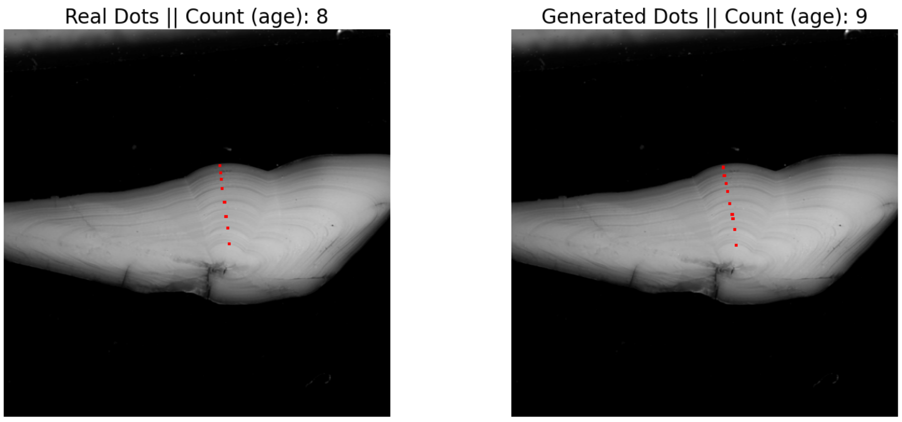
(k) Example of a result where no dots are placed on the otolith.


Real Dots || Count (age): 7
Generated Dots || Count (age): 6

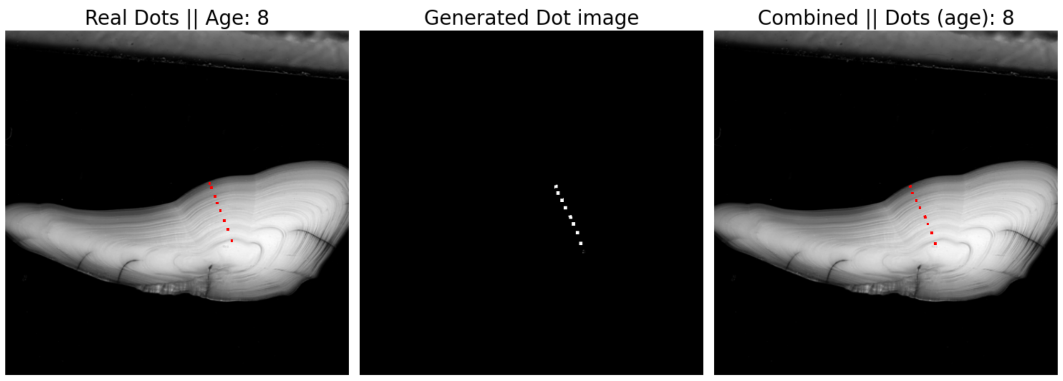
(l)


Real Dots || Count (age): 8
Generated Dots || Count (age): 9

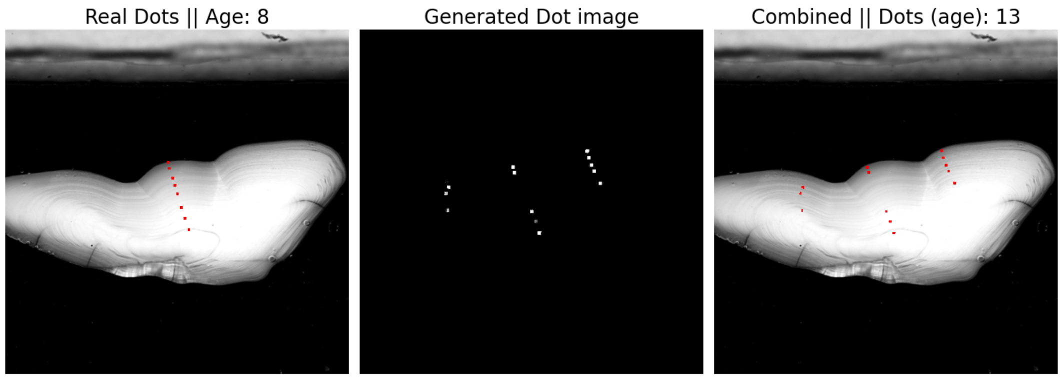
(m)


Real Dots || Count (age): 8
Generated Dots || Count (age): 9

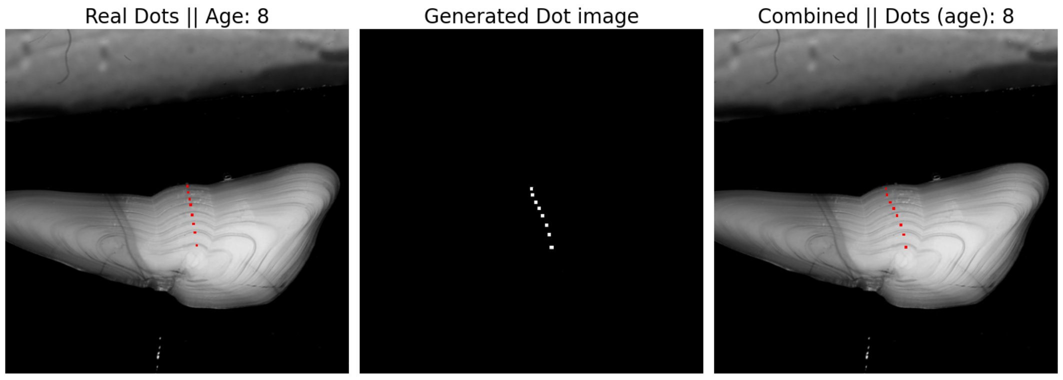
(n)


Real Dots || Age: 8
Generated Dot image
Combined || Dots (age): 8

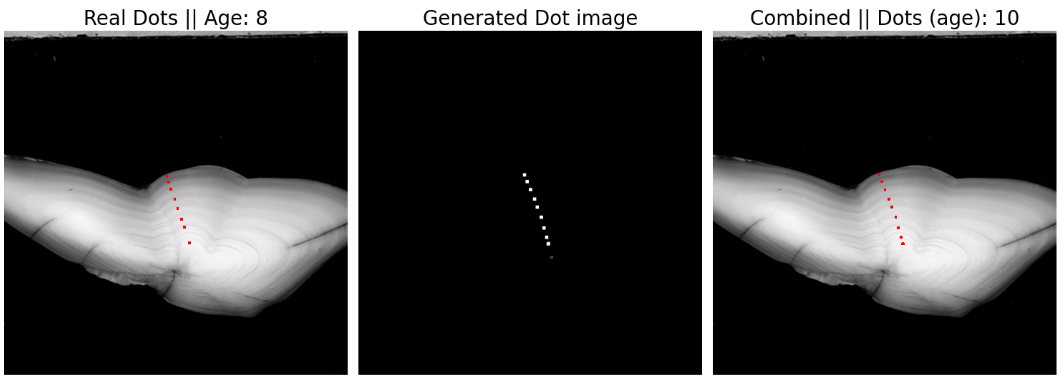
(o)


Real Dots || Age: 8
Generated Dot image
Combined || Dots (age): 13

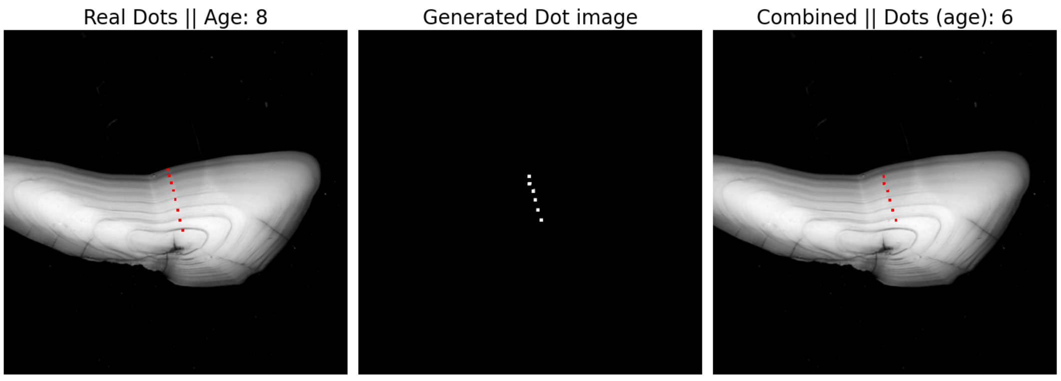
(p)


Real Dots || Age: 8
Generated Dot image
Combined || Dots (age): 8

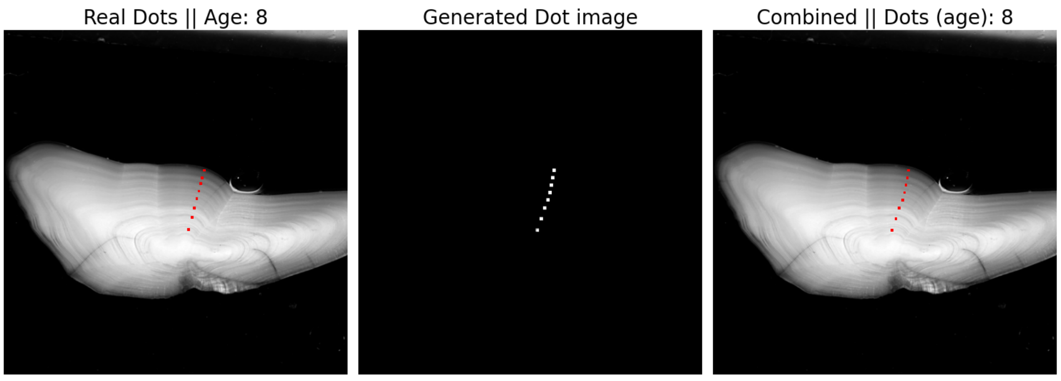
(q)



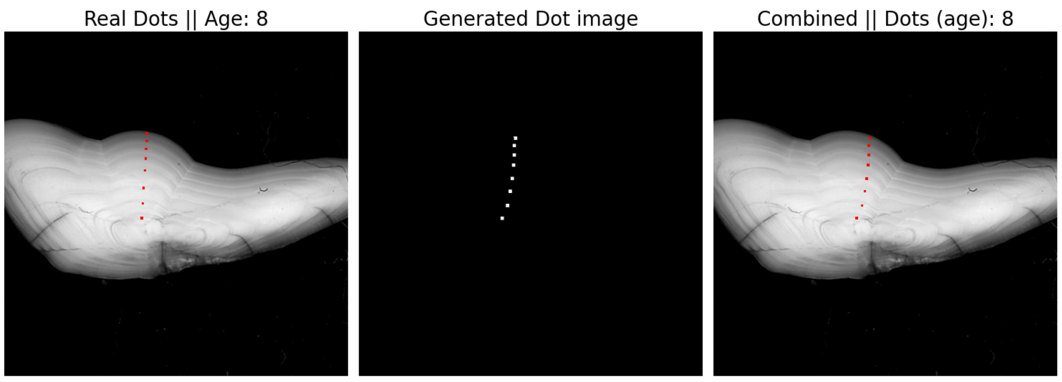
(r) Example of a result where a dot is placed on a check, as seen as the second dot counting from the core of the otolith. We also see that the first dot is barely visible in the dot image, but is still counted as a dot. It is however, not shown on top of the otolith. This issue might have been caused by having different thresholds in the visualization of dots overlaid on the otolith vs. the dot counting algorithm.



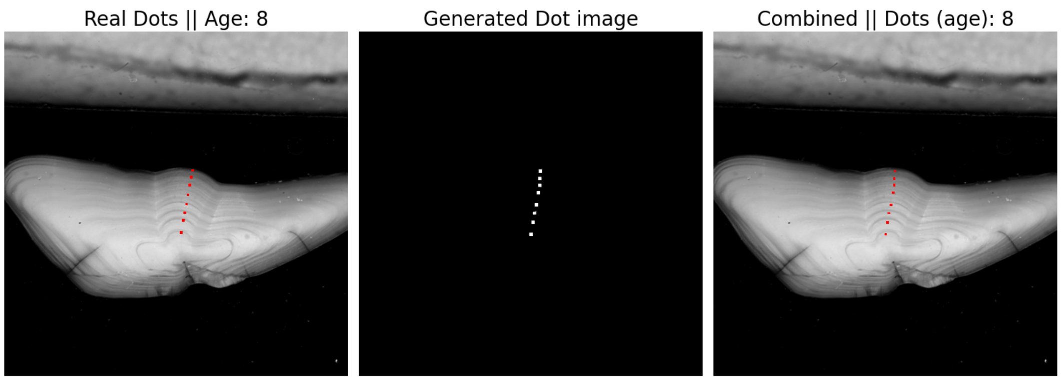
(s)



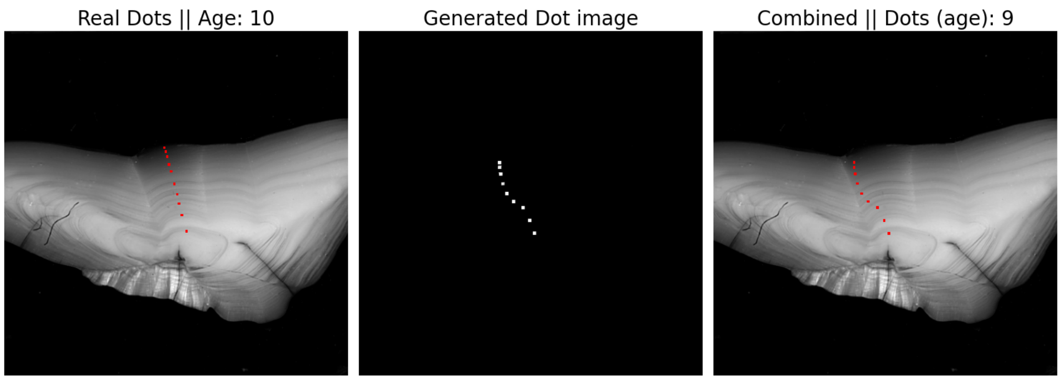
(t)



(u)



(v)



(w)

Figure B.1: Additional results from the successful experiments.

Bibliography

[1] Nikola Adaloglou. "How to stabilize GAN training". In: *Towards Data Science* (2020). URL: https://towardsdatascience.com/wasserstein-distance-gan-began-and-progressively-growing-gan-7e099f38da96.

[2] Aphex34. *Typical cnn*. https://upload.wikimedia.org/wikipedia/commons/6/63/Typical_cnn.png. [Accessed and edited: 8. September 2022] License: https://creativecommons.org/licenses/by-sa/4.0/deed.en. 2015.

[3] Martin Arjovsky, Soumith Chintala, and Léon Bottou. *Wasserstein GAN*. 2017. DOI: 10.48550/ARXIV.1701.07875. URL: https://arxiv.org/abs/1701.07875.

[4] aurorabreakup. *panoramio (755)*. https://commons.wikimedia.org/w/index.php?title=Main_Page&oldid=651894909. [Accessed and edited: 24. November 2022] License: https://creativecommons.org/licenses/by/3.0/deed.en. 2005.

[5] Pragati Baheti. "Activation Functions in Neural Networks". In: *V7Labs* (2022). URL: https://www.v7labs.com/blog/neural-networks-activation-functions.

[6] Nilesh Barla. "Pix2pix: Key Model Architecture Decisions". In: *Neptune* (2022). URL: https://neptune.ai/blog/pix2pix-key-model-architecture-decisions.

[7] Gervais et Boulart. *Gadus morhua Gervais*. https://upload.wikimedia.org/wikipedia/commons/4/4b/Gadus_morhua_Gervais.jpg. [Accessed: 19. September 2022] License: https://creativecommons.org/publicdomain/mark/1.0/deed.en. 2012.

[8] Bill Brazier. *Fish Hearing*. https://www.offthescaleangling.ie/wp-content/uploads/2020/01/Otolith-example-ageing_optimized.jpg. [Accessed: 24. August 2022] Permission for use granted by author. 2017.

[9] Bill Brazier. "Fish Hearing". In: *Off the Scale Magazine* (2017). URL: https://www.offthescaleangling.ie/the-science-bit/fish-hearing/.

[10] Jason Brownlee. "How to Develop a Wasserstein Generative Adversarial Network (WGAN) From Scratch". In: *Machine Learning Mastery* (2019). URL: https://machinelearningmastery.com/how-to-code-a-wasserstein-generative-adversarial-network-wgan-from-scratch/.

[11] Jason Brownlee. "How to Fix the Vanishing Gradients Problem Using the ReLU". In: *Machine Learning Mastery* (2020). URL: https://machinelearningmastery.com/how-to-fix-vanishing-gradients-using-the-rectified-linear-activation-function/.

[12] Government of Canada. "Otoliths, removal and ageing". In: *Otolith research lab, Government of Canada* (2018). URL: https://www.dfo-mpo.gc.ca/science/species-especes/otoliths/students/removal-prelevement-eng.html.

[13] Pau Carré Cardona. "How to convert fully connected layers into equivalent convolutional ones". In: *HBC Tech* (2016). URL: https://tech.hbc.com/2016-05-18-fully-connected-to-convolutional-conversion.html.

[14] B.J. Copeland. *artificial intelligence*. https://www.britannica.com/technology/artificial-intelligence. Encyclopedia Britannica, 2022.

[15] DeepAI. "Padding (Machine Learning)". In: *DeepAI* (2022). URL: https://deepai.org/machine-learning-glossary-and-terms/padding.

[16] Côme Denechaud et al. "A century of fish growth in relation to climate change, population dynamics and exploitation". In: *Global Change Biology* 26.10 (2020), pp. 5661–5678. DOI: https://doi.org/10.1111/gcb.15298. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/gcb.15298. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/gcb.15298.

[17] Dhp1080. *Neuron*. https://training.seer.cancer.gov/anatomy/nervous/tissue.html. [Accessed: 24. August 2022] License: https://creativecommons.org/licenses/by-sa/4.0/deed.en. 2019.

[18] P. Dimitrakopoulos, G. Sfikas, and C. Nikou. "ISING-GAN: Annotated Data Augmentation with a Spatially Constrained Generative Adversarial Network". In: *2020 IEEE 17th International Symposium on Biomedical Imaging (ISBI)*. 2020, pp. 1600–1603. DOI: 10.1109/ISBI45749.2020.9098618.

[19] Jason Dorrier. "OpenAI Says DALL-E Is Generating Over 2 Million Images a Day—and That's Just Table Stakes". In: *Singularity hub* (2022). URL: https://singularityhub.com/2022/10/03/openai-says-dall-e-is-generating-over-2-million-images-a-day-and-thats-just-table-stakes/.

[20] Vincent Dumoulin and Francesco Visin. *Convolution arithmetic - No padding no strides*. https://upload.wikimedia.org/wikipedia/commons/6/6c/Convolution_arithmetic_-_No_padding_no_strides.gif. [Accessed: 9. September 2022] License: https://commons.wikimedia.org/wiki/Category:Expat/MIT_License. 2016.

[21] Vincent Dumoulin and Francesco Visin. *Transposed Convolution arithmetic - No padding no strides*. https://upload.wikimedia.org/wikipedia/commons/0/07/Convolution_arithmetic_-_No_padding_no_strides_transposed.gif. [Accessed: 22. September 2022] License: https://commons.wikimedia.org/wiki/Category:Expat/MIT_License. 2016.

[22] Mads Dyrmann. *Neural Network Dropout*. https://upload.wikimedia.org/wikipedia/commons/thumb/6/66/Neural_Network_Dropout.svg/2560px-Neural_Network_Dropout.svg.png. [Accessed: 31. October 2022] License: https://creativecommons.org/licenses/by-sa/4.0/deed.en. 2021.

[23] NOAA Fisheries. "Age and Growth". In: *NOAA Fisheries* (2020). URL: https://www.fisheries.noaa.gov/national/science-data/age-and-growth.

[24] NOAA Fisheries. "Near-Infrared Technology Identifies Fish Species From Otoliths". In: *NOAA Fisheries* (2020). URL: https://www.fisheries.noaa.gov/feature-story/near-infrared-technology-identifies-fish-species-otoliths.

[25] NOAA Fisheries. *Near-Infrared Technology Identifies Fish Species From Otoliths*. https://media.fisheries.noaa.gov/dam-migration-miss/1280_s2kXntXlA241.png?1593183583. [Accessed: 24. August 2022] Permission for use granted by author. 2020.

[26] Joseph Foley. “18 deepfake examples that terrified and amused the internet”. In: *Creative Bloq* (2022). URL: https://www.creativebloq.com/features/deepfake-examples.

[27] Yulia Gavrilova. “Convolutional Neural Networks for Beginners”. In: *Serokell* (2021). URL: https://serokell.io/blog/introduction-to-convolutional-neural-networks.

[28] Martin Giles. “The GANfather: The man who’s given machines the gift of imagination”. In: *MIT Tehcnology Review* (2018). URL: https://www.technologyreview.com/2018/02/21/145289/the-ganfather-the-man-whos-given-machines-the-gift-of-imagination/.

[29] Daniel Godoy. “Understanding binary cross-entropy / log loss: a visual explanation”. In: *Towards Data Science* (2018). URL: https://towardsdatascience.com/understanding-binary-cross-entropy-log-loss-a-visual-explanation-a3ac6025181a.

[30] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. htttp://www.deeplearningbook.org. MIT Press, 2016.

[31] Ian J. Goodfellow et al. *Generative Adversarial Networks*. 2014. DOI: 10.48550/ARXIV.1406.2661. URL: https://arxiv.org/abs/1406.2661.

[32] Chirag Goyal. “Deep Understanding of Discriminative and Generative Models in Machine Learning”. In: *Analytics Vidhya* (2021). URL: https://www.analyticsvidhya.com/blog/2021/07/deep-understanding-of-discriminative-and-generative-models-in-machine-learning/.

[33] Ishaan Gulrajani et al. *Improved Training of Wasserstein GANs*. 2017. DOI: 10.48550/ARXIV.1704.00028. URL: https://arxiv.org/abs/1704.00028.

[34] Martin Heusel et al. “GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium”. In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon et al. Vol. 30. Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper/2017/file/8a1d694707eb0fefe65871369074926d-Paper.pdf.

[35] Jonathan Hui. “GAN - Spectral Normalization”. In: *Medium* (2020). URL: https://jonathan-hui.medium.com/gan-spectral-normalization-893b6a4e8f53.

[36] Sergey Ioffe and Christian Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. 2015. DOI: 10.48550/ARXIV.1502.03167. URL: https://arxiv.org/abs/1502.03167.

[37] Martin Isaksson. “Five GANs for Better Image Processing”. In: *Towards Data Science* (2021). URL: https://towardsdatascience.com/five-gans-for-better-image-processing-fabab88b370b.

[38] Phillip Isola et al. *Image-to-Image Translation with Conditional Adversarial Networks*. 2016. DOI: 10.48550/ARXIV.1611.07004. URL: https://arxiv.org/abs/1611.07004.

[39] Tony Jebara. *Machine Learning: Discriminative and Generative*. https://link.springer.com/book/10.1007/978-1-4419-9011-2. The Springer International Series in Engineering and Computer Science, 2004.

[40] Divakar Kapil. “Stochastic vs Batch Gradient Descent”. In: *Towards Data Science* (2019). URL: https://medium.com/@divakar_239/stochastic-vs-batch-gradient-descent-8820568eada1.

[41] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2014. DOI: 10.48550/ARXIV.1412.6980. URL: https://arxiv.org/abs/1412.6980.

[42] Simeon Kostadinov. "Understanding Backpropagation Algorithm". In: *Towards Data Science* (2019). URL: https://towardsdatascience.com/understanding-backpropagation-algorithm-7bb3aa2f95fd.

[43] Sri Krishna. "Midjourney founder says "the world needs more imagination"". In: *VentureBeat* (2022). URL: https://venturebeat.com/ai/midjourney-founder-says-the-world-needs-more-imagination/.

[44] Jan Kukačka, Vladimir Golkov, and Daniel Cremers. *Regularization for Deep Learning: A Taxonomy*. 2017. DOI: 10.48550/ARXIV.1710.10686. URL: https://arxiv.org/abs/1710.10686.

[45] Yann LeCun, Koray Kavukcuoglu, and Clément Farabet. "Convolutional networks and applications in vision". In: *International Symposium on Circuits and Systems (ISCAS 2010), May 30 - June 2, 2010, Paris, France*. IEEE, 2010, pp. 253–256. DOI: 10.1109/ISCAS.2010.5537907. URL: https://doi.org/10.1109/ISCAS.2010.5537907.

[46] German Lopez. "A Smarter Robot". In: *The New York Times* (2022). URL: https://www.nytimes.com/2022/12/08/briefing/ai-chatgpt-openai.html.

[47] Marrabbio2. *Fish anatomy*. https://upload.wikimedia.org/wikipedia/commons/f/f0/Fish_anatomy.jpg. [Accessed: 27. August 2022] License: https://creativecommons.org/publicdomain/zero/1.0/. 2006.

[48] MartinThoma. *Perceptron-unit*. https://hvidberrrg.github.io/deep_learning/activation_functions_in_artificial_neural_networks.html. [Accessed and edited: 24. August 2022] License: https://creativecommons.org/licenses/by-sa/4.0/deed.en. 2014.

[49] James McCaffrey. "Neural Network Momentum Using Python". In: *Visual Studio Magazine* (2017). URL: https://visualstudiomagazine.com/articles/2017/08/01/neural-network-momentum.aspx.

[50] B.R. Moore et al. "Development of deep learning approaches for automating age estimation of hoki and snapper". In: *Fisheries New Zealand* (2021). ISSN: 0165-7836. URL: https://docs.niwa.co.nz/library/public/FAR2021-69.pdf.

[51] Andrew Ng. *C4W1L02 Edge Detection Examples*. YouTube. 2017. URL: https://www.youtube.com/watch?v=XuD4C8vJzEQ.

[52] Thomas Brox Olaf Ronneberger Philipp Fischer. *U-Net: Convolutional Networks for Biomedical Image Segmentation*. https://lmb.informatik.uni-freiburg.de/people/ronneber/u-net/u-net-architecture.png. [Accessed: 15. September 2022] License: https://creativecommons.org/licenses/by/4.0/. 2015.

[53] Artem Oppermann. "Stochastic-, Batch-, and Mini-Batch Gradient Descent". In: *Towards Data Science* (2020). URL: https://towardsdatascience.com/stochastic-batch-and-mini-batch-gradient-descent-demystified-8b28978f7f5.

[54] Alba Ordonez et al. "Automatic Fish Age Determination across Different Otolith Image Labs Using Domain Adaptation". In: *Fishes* 7.2 (2022). ISSN: 2410-3888. DOI: 10.3390/fishes7020071. URL: https://www.mdpi.com/2410-3888/7/2/71.

[55] Adam Paszke et al. *PyTorch: An Imperative Style, High-Performance Deep Learning Library*. 2019. DOI: 10.48550/ARXIV.1912.01703. URL: https://arxiv.org/abs/1912.01703.

[56] Federico Peccia. "Batch normalization: theory and how to use it with Tensorflow". In: *Towards Data Science* (2018). URL: https://towardsdatascience.com/batch-normalization-theory-and-how-to-use-it-with-tensorflow-1892ca0173ad.

[57] Dimitris V. Politikos et al. "Automating fish age estimation combining otolith images and deep learning: The role of multitask learning". In: *Fisheries Research* 242 (2021), p. 106033. ISSN: 0165-7836. DOI: https://doi.org/10.1016/j.fishres.2021.106033. URL: https://www.sciencedirect.com/science/article/pii/S0165783621001612.

[58] Dimitris V. Politikos et al. "DeepOtolith v1.0: An Open-Source AI Platform for Automating Fish Age Reading from Otolith or Scale Images". In: *Fishes* 7.3 (2022). ISSN: 2410-3888. DOI: 10.3390/fishes7030121. URL: https://www.mdpi.com/2410-3888/7/3/121.

[59] Dietmar Rabich. *New York City (New York, USA), Times Square-Duffy Square – 2012 – 6380*. [Accessed and edited: 5. January 2023] License: https://creativecommons.org/licenses/by-sa/4.0/deed.en. 2012. URL: https://upload.wikimedia.org/wikipedia/commons/6/69/New_York_City_%28New_York%2C_USA%29%2C_Times_Square-Duffy_Square_---_2012_---_6380.jpg.

[60] Saman Razavi. "Deep Learning, Explained: Fundamentals, Explainability, and Bridgeability to Process-Based Modelling". In: *Environ. Model. Softw.* (2021). URL: https://doi.org/10.1016/j.envsoft.2021.105159.

[61] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. *U-Net: Convolutional Networks for Biomedical Image Segmentation*. 2015. DOI: 10.48550/ARXIV.1505.04597. URL: https://arxiv.org/abs/1505.04597.

[62] Sebastian Ruder. "An overview of gradient descent optimization algorithms". In: *Sebastian Ruder* (2016). URL: https://ruder.io/optimizing-gradient-descent/index.html#adam.

[63] Run:AI. "Deep Learning for Computer Vision". In: *Run:AI* (2022). URL: https://www.run.ai/guides/deep-learning-for-computer-vision.

[64] Deval Shah. "The Complete Guide to Generative Adversarial Networks (GANs)". In: *V7Labs* (2022). URL: https://www.v7labs.com/blog/generative-adversarial-networks-guide.

[65] Stephen Shankland. "Dall-E Opens Its AI Art Creation Tool to Everyone". In: *CNET* (2022). URL: https://www.cnet.com/tech/computing/dall-e-opens-its-ai-art-creation-tool-to-everyone/.

[66] Aditya Sharma. "Pix2Pix: Image-to-Image Translation in PyTorch and TensorFlow". In: *LearnOpenCV* (2021). URL: https://learnopencv.com/paired-image-to-image-translation-pix2pix/.

[67] SlideShare. "Tutorial on Theory and Application of Generative Adversarial Networks". In: *SlideShare, a Scribd company* (2017). URL: https://www.slideshare.net/mlreview/tutorial-on-theory-and-application-of-generative-adversarial-networks#.

[68] Vikas Solegaonkar. "Convolutional Neural Networks". In: *Towards Data Science* (2019). URL: https://towardsdatascience.com/convolutional-neural-networks-e5a6745b2810.

[69] Jost Tobias Springenberg et al. *Striving for Simplicity: The All Convolutional Net*. 2014. DOI: 10.48550/ARXIV.1412.6806. URL: https://arxiv.org/abs/1412.6806.

[70] Nitish Srivastava et al. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: *Journal of Machine Learning Research* 15.56 (2014), pp. 1929–1958. URL: http://jmlr.org/papers/v15/srivastava14a.html.

[71] Ilya Sutskever et al. "On the Importance of Initialization and Momentum in Deep Learning". In: *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*. ICML'13. Atlanta, GA, USA: JMLR.org, 2013, pp. III-1139-III–1147.

[72] TensorFlow. *Deep Convolutional Generative Adversarial Network*. https://www.tensorflow.org/tutorials/generative/images/gan1.png. [Accessed: 1. September 2022] License: https://creativecommons.org/licenses/by/4.0/. 2022.

[73] TensorFlow. *Deep Convolutional Generative Adversarial Network*. https://www.tensorflow.org/tutorials/generative/images/gan2.png. [Accessed: 1. September 2022] License: https://creativecommons.org/licenses/by/4.0/. 2022.

[74] Rob Toews. "Deepfakes Are Going To Wreak Havoc On Society. We Are Not Prepared". In: *Forbes* (2020). URL: https://www.forbes.com/sites/robtoews/2020/05/25/deepfakes-are-going-to-wreak-havoc-on-society-we-are-not-prepared/.

[75] *TORCH.CUDA*. https://pytorch.org/docs/stable/cuda.html. [Accessed: 16-January-2023].

[76] *TORCH.TENSOR*. https://pytorch.org/docs/stable/tensors.html. [Accessed: 4-January-2023].

[77] V. Turri. *What is Explainable AI?* Carnegie Mellon University's Software Engineering Institute Blog. Jan. 2022. URL: http://insights.sei.cmu.edu/blog/what-is-explainable-ai/.

[78] Jelica Vasiljević et al. *HistoStarGAN: A Unified Approach to Stain Normalisation, Stain Transfer and Stain Invariant Segmentation in Renal Histopathology*. 2022. DOI: 10.48550/ARXIV.2210.09798. URL: https://arxiv.org/abs/2210.09798.

[79] Wei Wang et al. "Development of convolutional neural network and its application in image classification: a survey". In: *Optical Engineering* 58.4 (2019), p. 040901. DOI: 10.1117/1.OE.58.4.040901. URL: https://doi.org/10.1117/1.OE.58.4.040901.

[80] Low De Wei. "This AI Chatbot Is Blowing People's Minds. Here's What It's Been Writing." In: *Bloomberg* (2022). URL: https://www.bloomberg.com/news/articles/2022-12-02/chatgpt-openai-s-new-essay-writing-chatbot-is-blowing-people-s-minds.

[81] Wikipedia contributors. *Earth mover's distance — Wikipedia, The Free Encyclopedia*. [Online; accessed 16-December-2022]. 2022. URL: https://en.wikipedia.org/w/index.php?title=Earth_mover%27s_distance&oldid=1121085069.

[82] Wikipedia contributors. *Generative adversarial network — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Generative_adversarial_network&oldid=1113247966. [Online; accessed 3-October-2022]. 2022.

[83] Wikipedia contributors. *Generative model — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Generative_model&oldid=1115984959. [Online; accessed 17-October-2022]. 2022.

[84] Wikipedia contributors. *Lipschitz continuity — Wikipedia, The Free Encyclopedia*. [Online; accessed 16-December-2022]. 2022. URL: https://en.wikipedia.org/w/index.php?title=Lipschitz_continuity&oldid=1122347867.

[85] Wikipedia contributors. *MNIST database — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=MNIST_database&oldid=1101449122. [Online; accessed 3-October-2022]. 2022.

[86] Wikipedia contributors. *Multi-task learning — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Multi-task_learning&oldid=1104310342. [Online; accessed 20-October-2022]. 2022.

[87] Wikipedia contributors. *OpenCV — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=OpenCV&oldid=1113479408. [Online; accessed 28-November-2022]. 2022.

[88] Wikipedia contributors. *Python Imaging Library — Wikipedia, The Free Encyclopedia*. [Online; accessed 20-December-2022]. 2022. URL: https://en.wikipedia.org/w/index.php?title=Python_Imaging_Library&oldid=1104375111.

[89] Wikipedia contributors. *Tensor — Wikipedia, The Free Encyclopedia*. [Online; accessed 20-December-2022]. 2022. URL: https://en.wikipedia.org/w/index.php?title=Tensor&oldid=1127664112.

[90] Cong Wu, Yixuan Zou, and Zhi Yang. "U-GAN: Generative Adversarial Networks with U-Net for Retinal Vessel Segmentation". In: *2019 14th International Conference on Computer Science and Education (ICCSE)*. 2019, pp. 642–646. DOI: 10.1109/ICCSE.2019.8845397.

[91] Aston Zhang et al. *Dive into Deep Learning*. 2021. DOI: 10.48550/ARXIV.2106.11342. URL: https://arxiv.org/abs/2106.11342.