Lone Marselia Werness Bekkeheien

Synthesizing Photo-Realistic images from a Marine Simulator via Generative Adversarial Networks

Master's thesis in Cybernetics and Robotics
Supervisor: Anastasios Lekkas
June 2020

Master's thesis

Lone Marselia Werness Bekkeheien

Synthesizing Photo-Realistic images from a Marine Simulator via Generative Adversarial Networks

Master's thesis in Cybernetics and Robotics
Supervisor: Anastasios Lekkas
June 2020

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Norwegian University of
Science and Technology

Synthesizing Photo-Realistic images from a Marine Simulator via Generative Adversarial Networks

Lone Marselia Werness Bekkeheien

Master of Science in Cybernetics and Robotics

Supervisor: Anastasios Lekkas, ITK

Submission date: June 2020

Norwegian University of Science and Technology

Department of Engineering Cybernetics

Preface

This thesis represents my work at the Norwegian University of Science and Technology (NTNU) as part of the study program Master of Science in Cybernetics and Robotics. The work has been carried out under the supervision of Anastasios Lekkas during the spring semester of 2020, which has been a great inspiration! This thesis aims to synthesize photo-realistic images from a marine simulator via GANs to improve the training of detection algorithms in the marine environment. The project has been performed in cooperation with DNV GL, who provided me with a marine simulator and drone footage of the autonomous ReVolt vessel. Martin Skaldebø and Albert Sans at the Department of Marine Technology at NTNU have also contributed with insightful discussion.
Some of the theory is taken from my project thesis, [1], and it is listed below.

- Subsection 3.1.1, except for the figures.
- Section 3.2, but the content has been changed, and figures added.
- From the beginning of section 3.3 till 3.3.1, but the content has been changed, and figures added.

The following resources were utilized in this master thesis:

- The ReVolt vessel Unity simulator by DNV GL.
- Jun-Yan's cycleGAN implementation [2].
- Matterport's Mask R-CNN implementation [3].
- Arteaga's LIME implementation [4].

- Google Colaboratory is providing single 12GB NVIDIA Tesla K80 GPU.
- Google Disk for saving checkpoints.
- Google forms to perform the 'Visual Turing Test'.
- Draw.IO to create figures.

All the implementations of this thesis are performed in Python, except the simulator scripts, which utilizes C-sharp. The following Python libraries are utilized:

- *Pandas* for data manipulation.
- *NumPy* for manipulation and math functions for arrays and metrics.
- *Matplotlib* for creating plots.
- *PyTorch* for implementing cycleGAN.
- *TensorFlow* for implementing Mask R-CNN.
- *Keras* for implementing Mask R-CNN.
- *Scikit-Learn* for implementing LIME.
- *Scikit-image* for image processing.
- *OpenCV* for image and video processing.

Trondheim, 19-06-2020

Lone Marselia Werness Bekkeheien

Acknowledgement

Several people deserve my thanks for their help during this work and my time at NTNU.
First of all, I want to thank my supervisor, Anastasios Lekkas, for pointing me in the right direction as well as motivating me with his engagement in the field of machine learning. From DNV GL, I would like to thank Tom Arne Pedersen for supplementing me with the marine simulator and drone footage needed for this project. Furthermore, I would also like to thank Martin Skaldebø and Albert Sans at the Department of Marine Technology for insightful discussion.
My years at NTNU has been challenging, and the support from my parents Heidi Bekkeheien and Jan Egil Werness, as well as my sister, June Bekkeheien, has been priceless.
Finally, I would like to thank everyone who participated in the 'Visual Turing Test'.

L.M.W.B

Abstract

Obtaining vast amounts of quality real-world data is expensive; therefore, it is reasonable to train detection algorithms in a simulated environment. However, there is a difference between simulated- and real-world environment referred to as the *reality gap*. Consequently, a vision-based algorithm trained in a simulated environment could generate failure when transferring its knowledge to real-life. Thus, this thesis aims to utilize the generative adversarial network (GAN) to improve the data quality acquired by a marine simulator and make it more realistic to achieve better detection algorithms in the marine environment. Cycle GAN (cycleGAN) is used to generate photo-realistic images based on a simulator of the autonomous ReVolt vessel, followed by training mask regional convolutional neural network (Mask R-CNN) in the simulated- and generated environment. The resulting models are tested in the real-world environment, and their predictions are explained by explainable artificial intelligence. The results show that training Mask R-CNN in a cycleGAN generated environment generalizes better to the real-world environment than the simulator trained model. Moreover, the explainable artificial intelligence revealed that the model trained in the generated environment base its prediction on more correct features than the simulator trained model when tested in the real-world environment. Thus, cycleGAN has proved its ability to improve the data quality acquired by a marine simulator and achieve better Mask R-CNN predictions in the real-life.

Sammendrag

Norwegian translation of the abstract.

Det er dyrt å skaffe enorme mengder data fra den virkelig verden. Derfor er det rimelig å trene deteksjonsalgoritmer i et simulert miljø. Imidlertid er det en forskjell mellom simulert miljø og den virkelige verden referert til som *realitetsgapet*. Følgelig kan en visjonsbasert algoritme trent i et simulert miljø generere feil når kunnskapen overføres til den virkelige verden. Dermed har denne oppgaven som mål å bruke generative motsigende nettverk (GAN) til å forbedre datakvaliteten anskaffet av en marin simulator og gjøre den mer realistisk for å oppnå bedre deteksjonsalgoritmer i det marine miljøet. Syklus GAN (cycleGAN) brukes til å generere fotorealistiske bilder basert på en simulator av det autonome ReVolt-fartøyet, etterfulgt av å trene et maske regionalt innviklet nevralt nettverk (Mask R-CNN) i det simulerte- og genererte miljøet. De resulterende modellene blir testet i det virkelige miljøet, og deres prediksjoner blir forklart med forklarbar kunstig intelligens. Resultatene viser at trening av Mask R-CNN i et cycleGAN-generert miljø generaliserer bedre til det virkelige miljøet enn den simulator-trente modellen. Videre avslørte forklarbar kunstig intelligens at modellen som ble trent i det genererte miljøet, baserer sin prediksjon på mer korrekte funksjoner enn den simulator-trente modellen når den ble testet i det virkelige miljøet. Dermed har cycleGAN bevist sin evne til å forbedre datakvaliteten anskaffet av en marin simulator og oppnå bedre Mask R-CNN prediksjoner i det virkelige miljøet.

Contents

List of Figures

Table of Abbreviations

Abbreviation	Description
GAN	Generative Adversarial Network
CycleGAN	Cycle GAN
cGAN	Conditional GAN
S+U	Simulated + Unsupervised
CoGAN	Coupled Generative Adversarial Network
SimGAN	Simulated Generative Adversarial Network
VAE	Variational Autoencoder
CIFAR10	Canadian Institute For Advanced Research 10
ML	Machine Learning
DP	Dynamic Positioning
ROV	Remotely Operated Underwater Vehicle
RL	Reinforcement Learning
DL	Deep Learning
CNN	Convolutional Neural Network
MC-lab	Marine Cybernetics Laboratory
NTNU	Norwegian University of Technology and Science

Abbreviation	Description
R-CNN	Region-based CNN
MLP	Multi-Layer Perceptron
PNG	Portable Network Graphic
BN	Batch Normalization
ReLU Rectified Linear Unit	
FCs	Fully Connected Layers
FCN	Fully Convolutional Networks
RoI	Region of Interest
RPN	Region Proposal Network
AI	Artificial Intelligence
FAIR	Facebook AI Research
SVM	Support Vector Machine
XAI	Explainable AI
LIME	Local Interpretable Model-agnostic Explaination
DOF	Degrees of Freedom
CPU	Central Processing Unit
FPN	Feature Pyramid Network
LiDAR	Light Detection and Ranging
GUI	Graphical User Interface
SGD	Stochastic Gradient Descent
NN	Neural Network
ANN	Artificial Neural Network
KKT	Karush-Kuhn-Tucker
LTU	Linear Threshold Unit
UDA	Unsupervised Domain Adaption
LSGAN	Least Square GAN

Table of Symbols

Symbol	Description
$\mathbf{x}$	Vector of input variables
$\mathbf{t}$	$\mathbf{x}$'s corresponding target values
$\phi(\mathbf{x})$	Basis function of input variables
$\mathbf{w}$	Vector of weights
$\mathbf{y}$	Predicted output based on $\mathbf{x}$
b	Bias parameter
$J(\boldsymbol{\theta})$	Model dependent cost function
p_{data}	Actual data variables
$p_{model}(\mathbf{y}\|\mathbf{x})$	Model distribution
$E(\mathbf{w})$	The negative logarithm of the likelihood function
g	The activation function for a unit in NN mapping from $\mathbb{R}^m$ to $\mathbb{R}^n$
f	A function mapping from $\mathbb{R}^n$ to $\mathbb{R}$
z	Is $g(\mathbf{x})$ mapped by f
$s(i, j)$	Convolution between a kernel and an image, where i, j represents rows and columns respectively
$\mathbb{E}$	Expected value

Symbol	Description
I	An image
K	A kernel, also called a filter
G	Generator
D	Discriminator
$V(D,G)$	Value function of an iterative two-player minmax game
z_{latent}	Latent space vector
$p_{z_{latetnt}}$	Input noise variables
L_{cGAN}	The cGAN's objective function
L_{L1}	L1 distance
L_{cyc}	Cycle consistency loss
$L_{GAN}(G, D_Y, X, Y)$	Objective function for $G : X \rightarrow Y$ in cycleGAN
$L_{GAN}(F, D_X, Y, X)$	Objective function for $F : Y \rightarrow X$ in cycleGAN
$L(F, D_X, Y, X)$	CycleGAN's objective function
g_L	LIME explanation model
G_L	Class of interpretable models
$\Omega(g_L)$	A complexity measure of g_L
z'_L	Perturbed sample
z_L	Perturbed sample in the original representation
$\pi_x(z_L)$	Proximity measure between x and z_L
Z	Dataset of preturbed samples and labels

Symbol	Description
$\mathscr{L}$	Fidelity functions
$\mathscr{L}(f, g_L, \pi_x)$	Locality-aware loss
$\xi(x)$	LIME explanation of input instance x
D_L	Distance
σ	Kernel width

Chapter 1

Introduction

Section 1.1 presents the motivation for this master thesis. Previous work is presented in Section 1.2, followed by Section 1.3 presenting the main objective and the approach that is used to reach the overall goal of the thesis. The contributions are presented in Section 1.4. Finally, Section 1.5 presents an overview of the report's structure.

1.1 Motivation

The word *autonomous* comes from the Greek roots "autos" meaning self, and "nomos" meaning law, and refers to a system functioning separately or independently. Today, many operations are becoming, or already have come to a degree, autonomous. For offshore applications, for instance, dynamic positioning (DP) has existed for decades and is used to automatically maintain a vessel's position and heading by using thrusters and propellers. Utilizing DP in remotely operated underwater vehicles (ROVs), for example, removes the sensitivity to human errors, which is one of the advantages of autonomous operations. Another more modern application that has recently received attention by both industry and academia is fully autonomous ships. The DNV GL's ReVolt vessel is an ongoing project of a fully battery-powered and autonomous vessel for transferring road freight volumes to waterways. Since the vessel is fully autonomous, it requires no crew, which leads to higher vessel container capacity due to no accommodation deck. Transportation of containers will, therefore, be more time- and cost-efficient. Besides the vessel itself, automated cargo handling, automated shore-based charging, and automated mooring facilities are also being developed in the ReVolt project [24].

Machine learning (ML) is expected to play a central part in the autonomous development. Deep neural networks (NNs) can solve challenging tasks if enough labelled data is available. The data for training an ML algorithm can be difficult and time-consuming to obtain manually because heaps amounts of quality data are needed to train a successful model. To train an algorithm for predicting when a vehicle is destroyed, for instance, would require a vast amount of data of vehicles crashing, and it is impractical to crash many vehicles to gather data. It is, therefore, better to use a simulator to collect the relevant data for training the model, since a simulator can create heaps amounts of data in a short amount of time. On the other hand, it is difficult to generate quality data with a simulator due to the difference in representation between the simulated- and real-world environment, which is called *the reality gap*. Due to the reality gap, a model for vision-based operations trained in a simulated environment could generate failure when transferring its knowledge to real-life. It is, therefore, of interest for operators in the market to generate robust techniques for transferring knowledge between domains and reducing the reality gap. For instance, the navigation algorithm for the autonomous ReVolt vessel is trained in a simulated environment. When the vessel is navigated with this trained model in the fjord of Trondheim for autonomously transferring containers, it could potentially crash or navigate to an incorrect port due to the reality gap. Therefore it makes sense to create more realistic simulators to make the transfer learning less prone to errors to enable more systems to become autonomous.

This master thesis explores the possibility of minimizing the reality gap by utilizing the technology of GANs. The network is applied to images extracted from a simulated environment of the ReVolt vessel as well as actual images of the real-world environment. If an object detection algorithm can detect the ReVolt Vessel with better accuracy after training on synthesized photo-realistic images from the simulator via GANs than just in the simulated environment, it could potentially help improve the autonomy of the ReVolt project and contribute in the shift towards more autonomous systems in general.

1.2 Previous Work

GANs were first introduced by Ian Goodfellow and other researchers at the University of Montreal in 2014 [25]. GANs consist of two neural networks, the generative and the discriminative, which work as adversarials. A generative network aims to learn the actual data distribution of the training data, and then use this distribution to generate new data that looks like it comes from the same distribution. The discriminator aims to predict which data comes from the training dataset and which comes from the generated dataset. Thus, GAN aims to generate data that looks like it is generated by the same set of rules as the training data. To reach this goal, an unknown probabilistic distribution function that explains why some data are more likely to originate from the training dataset and others are not, is to be found. In [25] the authors proposed extensions to the method like conditioning the generative model.

Conditional GANs (cGANs) have been utilized to perform image-to-image translation between domains in [12]. In an unconditional GAN, only the generator is conditioned on an input image. On the other hand, in a cGAN, both the generator and discriminator are conditioned on the input image. Here the term conditioned means feeding the network extra information as input that is used to learn from; that is, the additional information is conditioning the learning. The paper has used cGAN to generate realistic street views from semantic labelled images and photos from sketches, among other applications.

Another type of GAN, called the coupled GAN (coGAN), has been used to perform domain translation between a video game and reality by semantic labelling [26]. The authors achieved good results in real-time, but it simply does not make sense to render high-quality images from the video games, semantic labelling them, and then converting it to something else. Semantically labelled images should be used directly instead. Therefore, having a simulator that provides semantic images directly and then use this to turn it into photo-realistic images makes more sense.

A cycled type of GAN called the cycleGAN has also been used to perform domain translation between a video game and reality by utilizing semantic labelling [13]. The cycleGAN learns the mapping from both domain 1 to domain 2 with a discriminator and generator network, and the

inverse mapping from domain 2 back to domain 1 with another pair of generator and discriminator network. Due to the cycleGAN introducing the inverse mapping, it also introduces a cycle consistency loss to enforce the two generators and discriminators to generate correct data between the two domains consistently. Even though rendering high-quality images from the video game, semantic labelling them and converting it to something else does not make sense as mentioned above, the paper also achieves great results of mapping between domains like seasons, aerial photos to Google Maps photos and semantic labels to street views.
The method of cycleGAN from [13] was also utilized in transfer learning for underwater operations in [27]. Two datasets of rendered and real images of a subsea panel are used in [27]. The first dataset contains images of a subsea panel on the bottom of a marine cybernetics laboratory (MC-lab) at NTNU. In contrast, the second dataset consists of images of a subsea panel on the bottom of a fjord outside Trondheim. The real images are retrieved from a video stream filming the subsea panel, while the rendered images are from a software blender. The second dataset is more detailed and has more noise than the first dataset. In the results for the first dataset, the panel in the output image changes position relative to the input image, that is, the annotation is not preserved. The panel's position does not change as drastically for the second dataset, which could mean that more details are better for the CycleGAN. There is room for improvements in the achieved results in this paper.

Apple has also developed a GAN method for reducing the reality gap between simulated- and real-world environments, called simulated GAN (simGAN). Their resulting images have preserved the annotation and improved the realism of the synthetic images. They have also managed to avoid generating new artifacts and make the training more stable. A more stable training was achieved by updating the discriminator using a history of refined images rather than only the ones from the current refiner network [16]. Even though this led to a more stable training, it is also more computationally expensive, especially if the network is to run over several epochs with huge batch size. Generating new artifacts is avoided by limiting the discriminator's receptive field to local fields instead of the whole image.

All of the papers presented above, except the cGAN, use unpaired data for training, which means that the images in one domain do not have corresponding images in the other domain and vice

versa. Some of the papers are researched more and compared in Section 3 to see if the methods can be utilized in the implementation of this master thesis.
The different GANs mentioned above have been utilized in a wide range of applications due to their generality and simplicity, and the most relevant applications are introduced below.

The cycleGAN has not only been utilized for transfer learning in underwater operations but also in an attempt to address the reality gap between simulated and real-world environment for robot learning [28]. Due to the complexity of setting up a real robot and calibrate the dynamics across simulation and reality, the paper attempts to evaluate the method in two different simulated environments. The environments are both of a robotic arm that is to lift a cube, but the backgrounds and the cubes are of different colors. One of the domains is a bit more "complex" than the other due to a textured background. The paper achieves positive transfer results, but it also notices how low resolution and data imbalance affect the generator negatively. By data imbalance, the paper refers to having few images of the "complex" simulator and more of the "simpler" simulator. Future work for this paper is to use cycleGAN to minimize the reality gap between actual simulated- and real-world environment.
Robotic grasping is one of the fields affected by the fact that models trained purely on simulated data often fails to generalize to the real-world. GraspGAN has been developed to address the reality gap problem of deep robotic grasping in [29]. The paper reveals how the framework reduces the number of needed real-world samples by up to 50 times to achieve a level of performance.

LiDAR, which stands for Light Detection and Ranging, is a sensor utilized for environment perception of automated driving vehicles. LiDAR is utilized in the DNV GL's autonomous ReVolt vessel project. Heaps amount of sensor data is needed to train a Deep Learning model (DL) for autonomously drive a vehicle, which is complex and expensive to obtain. Using a LiDAR simulator to gather the data makes the process easier, but the trained model has trouble generalizing in the real-world environment. CycleGAN is used in [30] to solve the sensor modelling problem for LiDAR to produce realistic LiDAR from simulated LiDAR. The paper's results illustrate a high potential of the proposed approach.

Other than robotic applications, GANs have also been utilized in medical applications. The cGAN has been utilized to synthesize radiological images of the spine to be used in *In Silico* Trial in [31]. In Silico Trial is an individualized computer simulation utilized in the development of medicinal equipment and product. The first training dataset consists of paired data of semantically labelled images of the spine and actual images of the spine. The resulting generative framework created convincing synthetic planar X-rays. The second dataset consisted of paired sagittal and coronal images of the same patient. At first glance, the GANs managed to generate acceptable X-ray sagittal images from the coronal images and vice versa. However, a closer inspection of the generated images revealed several anatomical inaccuracies [31].

Other papers that have attempted to reduce the reality gap by utilizing GANs are the following [32, 33, 34]. Thus, GANs have been researched extensively, although it has not been utilized to improve the data quality by a marine simulator to make it more realistic to achieve better detection algorithms in the marine environment.

1.3 Objective and Approach

Obtaining quality real-world data is expensive; therefore, it is reasonable to train detection algorithms in a simulated environment. As mentioned in Section 1.1, there exists a reality gap between the simulated- and real-world environment, which makes the transfer learning prone to errors. The main objective of this work is thus to evaluate the possibility of utilizing GAN to improve the data quality acquired by a marine simulator and make it more realistic to achieve better detection algorithms in the marine environment.
A step-by-step approach is used to arrive at the final goal as follows.

1. Investigate different ways within GANs for minimizing the reality gap between simulated- and real-world environments. Discuss the candidates and choose one to use in the system. Step 1 is performed in Section 3.3, where cycleGAN is chosen.

2. Extract data from a marine simulator and obtain data from the real-world environment. The simulator that is utilized in this master thesis is of DNV GL's autonomous ReVolt vessel. The images of the real-world environment are retrieved from drone footage of the vessel in the fjord of Trondheim.

3. Adapt and apply cycleGAN to the dataset from the simulated- and real-world environment.

4. Evaluate the added value of cycleGAN's generated images.
   The evaluation is first performed by a 'Visual Turing Test' where 50 participants are asked to label images as real or fake. Secondly, evaluation is done by training Mask R-CNN in the simulated- and generated environment. The performance of these two models trained in different environments is tested in the real-world environment. These quantitative results are then validated by explainable AI, more specific, by the local interpretable model-agnostic explanation (LIME).

Figure 1.1 presents an overview of the system utilized to reach the main objective of this thesis. Acquisition of both simulated and real-world data is performed in Section 4.1. CycleGAN is trained on both the simulated- and real-world data to acquire generated data, described more thoroughly in Section 4.2. Further, model 1 and 2 in Figure 1.1 is achieved by training Mask R-

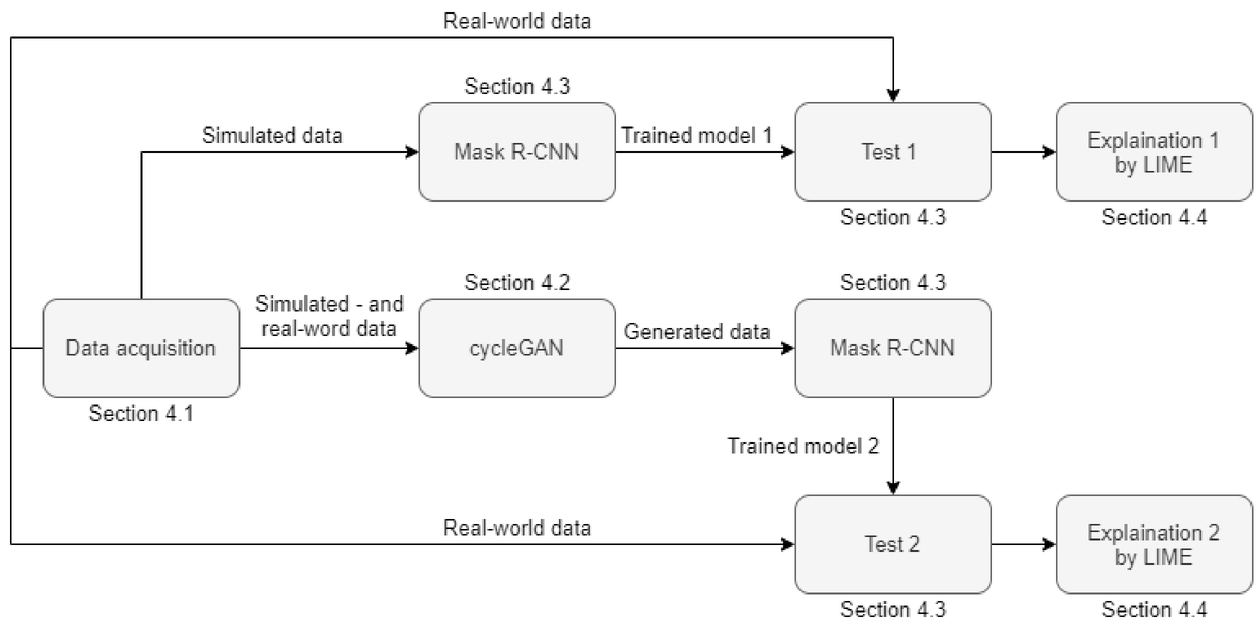


Figure 1.1: Block diagram of the system utilized to reach the main objective.

CNN on simulated- and generated data, respectively, described in Section 4.3. Finally, model 1 and 2 is tested on real-world data, referred to as test 1 and test 2, respectively, which is also described in Section 4.3. The overall objective is thus, to compare test 1 and test 2. The predictions made in test 1 and test 2 are explained by LIME, resulting in explanation 1 and 2 in Figure 1.1 respectively. Section 4.4 describes how LIME is integrated in the system.

1.4 Contributions

The main contribution of this thesis is the applications of ML frameworks in marine environments. As mentioned in the above section, cycleGAN is utilized to evaluate the possibility of improving the data quality acquired by a marine simulator and make it more realistic to achieve better detection algorithms in the marine environment. The different contributions of this thesis are listed below.

- To the author's best knowledge, this is the first time GAN is utilized to potentially improve visual-based learning algorithms for an autonomous vessel.
- To the author's best knowledge, this is also the first time GAN is used for synthesizing photo-realistic images from a marine simulator at sea-level.

- Mask R-CNN detected the ReVolt Vessel with better accuracy after training on synthesized photo-realistic images from the simulator via GANs than just in the simulated environment, which could potentially help improve the autonomy of the ReVolt project and contribute in the shift towards more autonomous systems in general.

1.5 Structure of the Report

The rest of the report is structured as follows:
The thesis starts by presenting the utilized software and equipment in Section 2. Theory about ML, including NNs, GANs, object detection and XAI is written in Section 3. This section starts with an introduction to ML, followed by a subsection on NNs. Next, different GANs and object detection algorithms are introduced and discussed. Section 3 also states why cycleGAN and Mask R-CNN is utilized in the system of this thesis. The design and implementation of the system, consisting of data acquisition, cycleGAN, Mask R-CNN and LIME, are explained in Section 4. The results are presented and discussed in Section 5, while Section 6 gives concluding remarks as well as making suggestions for further work.

Chapter 2

Software and Equipment

2.1 Python

The chosen frameworks for this thesis are implemented using Python, a high-level, general-purpose programming language. Python includes many libraries, and some of them are developed specifically for AI applications. On the other hand, the Python language uses a large amount of memory, which could be problematic for memory-intensive tasks. The execution of Python is also considered to be slow. Even though there are some drawbacks, PyTorch, Keras, and TensorFlow are libraries in the Python framework well suited for programming NNs and are therefore utilized in the implementations of this thesis.

2.1.1 PyTorch

PyTorch can be seen as a Python front end to the Torch engine, which provides the ability to define mathematical functions and compute their gradients [35]. The library is well suited for experimenting with new DL architectures because the source code in PyTorch is intuitive with a close correspondence with the mathematics in the networks.

2.1.2 Keras

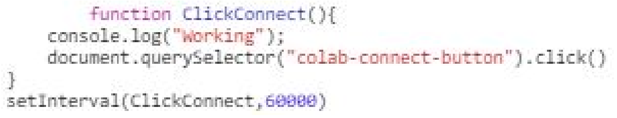
Keras is a high-level NN library written in Python [36]. Due to the library's user-friendliness, it enables easy and fast prototyping. Keras can also run on top of other lower-level libraries like TensorFlow.

2.1.3 TensorFlow

TensorFlow is an interface for expressing machine learning algorithms and an implementation for executing such algorithms [37]. The library is written in Python, C++, and CUDA. The system of TensorFlow is flexible due to its comprehensive ecosystem of libraries, tools, and community resources.

2.2 Google Colaboratory

Google Colaboratory is a free Jupyter notebook cloud service providing a single 12GB NVIDIA Tesla K80 GPU. Since it is a Jupyter notebook environment, it supports Python programming language. Google Colaboratory is utilized for training the implemented networks on online GPU and RAM. This way, the computationally expensive code can run on a GPU in Google's cloud instead of running locally. Google Colaboratory is also very easy to integrate with Google Disk, which is desirable to save checkpoints while training for hours. The free version of Google Colaboratory (the only version available outside the United States) can run for 12 hours, and it is, therefore, advisable to keep checkpoints to be able to continue the training after it has stopped. The GPU in Google Colaboratory is not guaranteed and not unlimited. The DevTools console command function displayed in Figure 2.1 is advisable for Google Colaboratory to stay connected when running for hours.



```
    function ClickConnect(){
  console.log("Working");
  document.querySelector("colab-connect-button").click()
}
setInterval(ClickConnect,60000)
```

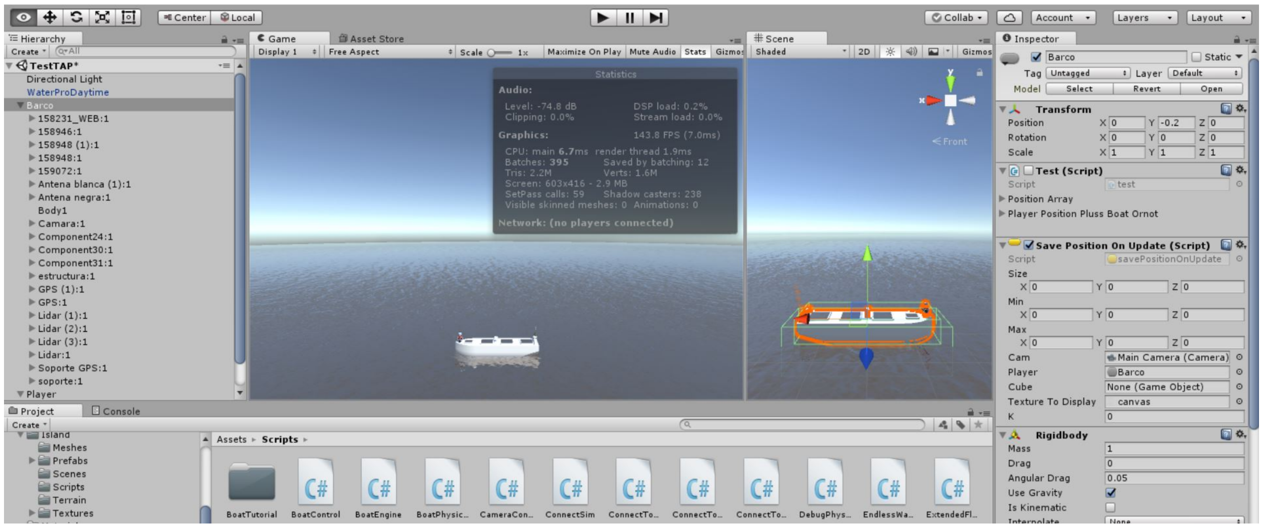
Figure 2.1: JavaScript command for Google Colaboratory to stay connected.

2.3 Unity Real-Time Development Platform

The cross-platform game engine, Unity, is developed by Unity Technologies [38]. The platform can be used to create 3D virtual- and augmented reality games as well as simulations, among other applications. The rendering of graphics in Unity's game view uses the available processor of the hosting device, which in this case, is a central processing unit (CPU). This thesis utilizes the Unity platform for the simulations.

2.3.1 Unity Simulator

This master thesis uses a 3D simulator of DNV GL's autonomous ReVolt vessel. A screenshot of the Unity engine simulator is visualized in Figure 2.2. It can be seen that the Unity platform



```
Statistics
Audio:
  Level: -74.8 dB            DSP load: 0.2%
  Clipping: 0.0%             Stream load: 0.0%
Graphics:                    143.8 FPS (7.0ms)
  CPU: main 6.7ms  render thread 1.9ms
  Batches: 395          Saved by batching: 12
  Tris: 2.2M            Verts: 1.6M
  Screen: 603x416 - 2.9 MB
  SetPass calls: 59     Shadow casters: 238
  Visible skinned meshes: 0  Animations: 0
Network: (no players connected)
```

```
Barco                          Static
Tag Untagged        Layer Default
Model   Select   Revert   Open
Transform
  Position   X 0   Y -0.2   Z 0
  Rotation   X 0   Y 0      Z 0
  Scale      X 1   Y 1      Z 1
Test (Script)
  Script     test
  Position Array
  Player Position Pluss Boat Ornot
Save Position On Update (Script)
  Script     savePositionOnUpdate
  Size
    X 0   Y 0   Z 0
  Min
    X 0   Y 0   Z 0
  Max
    X 0   Y 0   Z 0
  Cam                Main Camera (Camera)
  Player             Barco
  Cube               None (Game Object)
  Texture To Display canvas
  K                  0
Rigidbody
  Mass           1
  Drag           0
  Angular Drag   0.05
  Use Gravity    ✓
  Is Kinematic
  Interpolate    None
```

Figure 2.2: A screenshot of the Unity Simulator utilized in this thesis.

has a Game view that is rendered from the camera in the game and is representative of the final published game, which in this case, is the final simulation. Figure 2.2 also displays the platform's Scene view, which is the interactive view of the simulations that are created. The Scene view is utilized to select and position cameras, lights, game objects, and scenery. Each game object in Unity has its Inspector window as shown for the vessel, called *Barco* in this project, to the right in Figure 2.2. The Inspector window displays information of the game object like its attached components and their properties and scripts. This information can be modified to change the object's functionality. The Console window shows messages generated by Unity like errors, warnings, and debug logs. The figure also displays a script folder with C-sharp scripts that can be attached to, for instance, game objects or cameras.

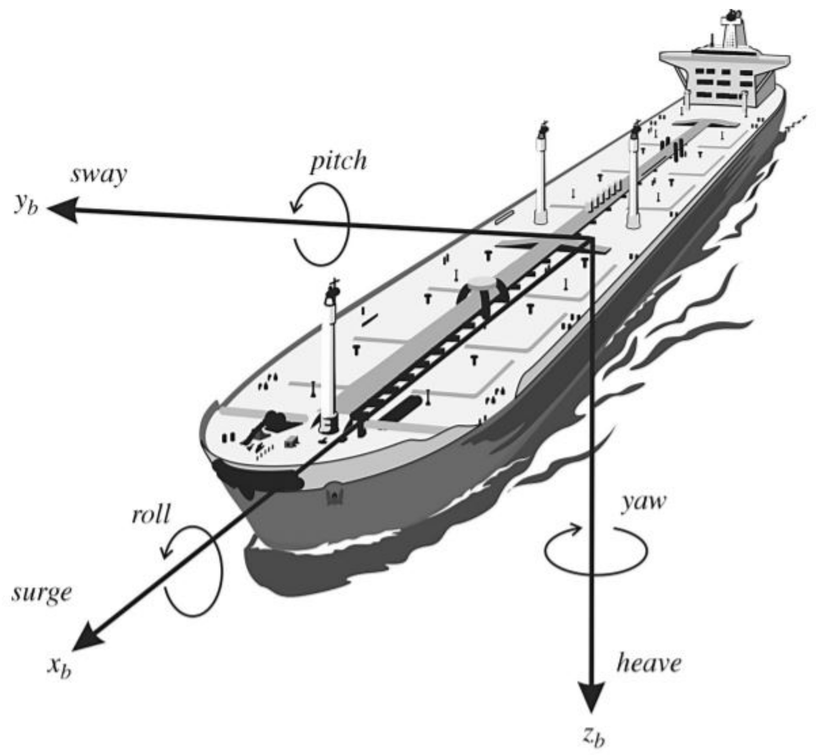


Figure 2.3: Motion in six degrees of freedom (DOF) [5].

2.3.2 C-Sharp

C-sharp is an object-oriented, general-purpose, multi-paradigm, intuitive programming language. C-sharp scripts can be added to the game object, the ReVolt vessel, in Unity to give it physical movements in roll, pitch, yaw, heave, sway and surge, as illustrated in Figure 2.3. Due to simplicity, the vessel only has two DOF, which are sway and surge. The movements in two DOF enables for rendering vast amounts of distinctive vessel images, which has been performed by C-sharp scripts.

Chapter 3

Theory

Parts of this section is taken from Section 3 and 4 in [1].

3.1 Introduction

A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E.
—Tom Mitchell, 1997 [39].

This master thesis concerns the problem of reducing the reality gap between a simulated environment and the real-world environment. Relating this problem to Mitchell's definition of ML, the problem itself is the task T, the measured performance, P, is how well the transformation between the different environment is, and the data or images of the real and simulated environment used to train the algorithm is the experience, E.
Today ML is an important part of smart technologies that have a huge impact on human's everyday lives. ML is, for example, used in spam filters for e-mail, applications like Uber for minimizing the costumers waiting time, Facebook for personalizing the user's news feed, Google's search engine for finding what the user is looking for and the list goes on. The technology of ML comes handy when the problem is too hard to program by hand. For instance, if a spam filter is made without the use of ML, rules for what kind of e-mails that should be flagged as spam has

to be written explicitly, which is too time-consuming.

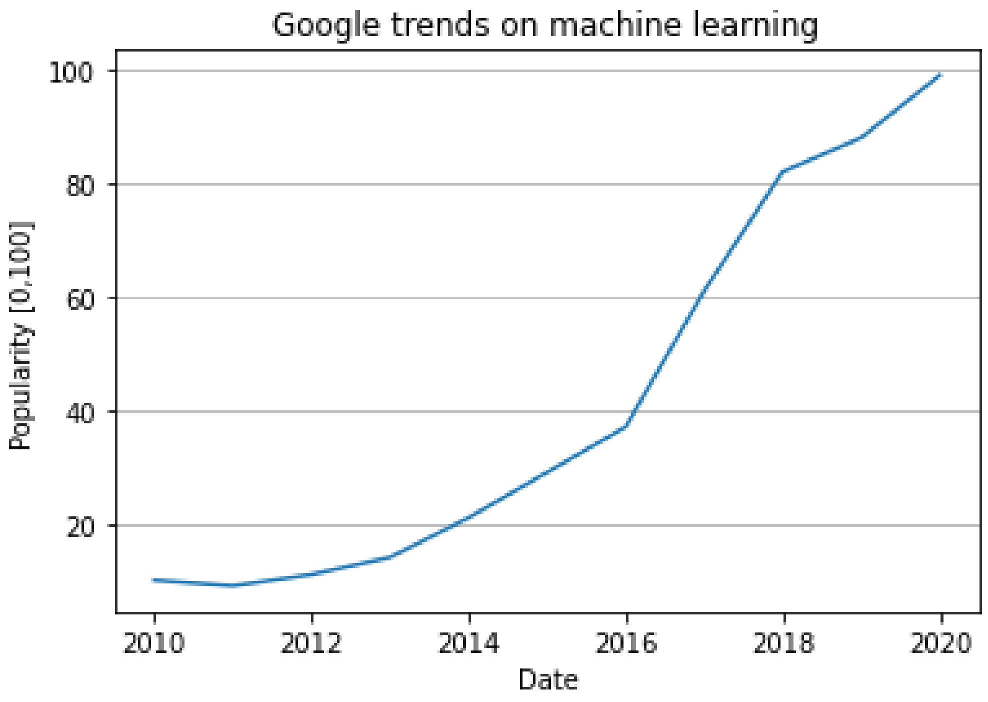


Figure 3.1: Google trends worldwide on machine learning over the past decade.

Figure 3.1 illustrates how the popularity, on a scale from 0 to 100, in ML has increased worldwide over the past decade. The data for the plot in Figure 3.1 is taken from Google Trends [40].
The learning algorithms for achieving the goal of ML described by Mitchell can be divided into different categories.

3.1.1 Learning Algorithms

The ML algorithms can be categorized into three main categories. The first category is based on whether or not the correct answers to the problems are known in the training period.

Supervised learning

Supervised learning algorithms have both the input and output data available in the training set; that is, the training data is labelled. Two typical types of supervised learning are classification and regression. The e-mail spam filter is a typical classification example, where the training data consists of different e-mails with corresponding labels classifying them as spam e-mails or not. After the training period, the spam filter should be able to recognize e-mails that are spam and e-mails that are not spam. A typical regression task is to predict the price of a car based on different kinds of features. The training data for this regression task consists of sets of features for different cars with their corresponding price. The trained model should be able to receive features of a certain car and output the predicted price of it. Figure 3.2 visualize how

Data with Label
Supervised Learning Model
Mapping

Figure 3.2: Supervised learning algorithm.

a supervised learning algorithm processes labelled data to predict the mapping. The mapping can either be a class, as for a classification problem, or a value like a price, as for a regression problem.

Unsupervised learning

Unsupervised learning algorithms take data without a label as input. Some important unsupervised learning algorithms are clustering and visualization and dimensionality reduction. The clustering algorithm is trying to group the input data in clusters based on similarity. For instance, you can use a clustering algorithm to detect what kind of groups are visiting an online newspaper, and at what times. This information can further be used to target specific readers at different times. Visualization algorithms can be used to input data and get a graphical representation as output. In unsupervised learning, the algorithms try to understand and learn from the data without the solution given in the training data, as for supervised learning. Figure 3.3

Data without Label
Usupervised Learning Model
Classes

Figure 3.3: Unsupervised learning algorithm.

illustrates how an unsupervised learning algorithm takes unlabeled data is input and predicts a class for this data. The algorithm discovers structures in the data and groups it. It is possible to make a hybrid of supervised- and unsupervised learning, which is called semi-supervised learning. Algorithms like this have a dataset consisting of a mix between unlabeled- and labelled data.

Reinforcement learning

Reinforcement learning (RL) algorithms learn based on rewards received on the last action performed [39]. RL algorithms can also work in a changing environment, which is typically used for an algorithm that plays a game. The player is then called an agent, the game is the environment, and the player is selecting action based on rewards from previous actions, as displayed in figure 3.4. The algorithm plays a certain game until the optimal policy is reached.

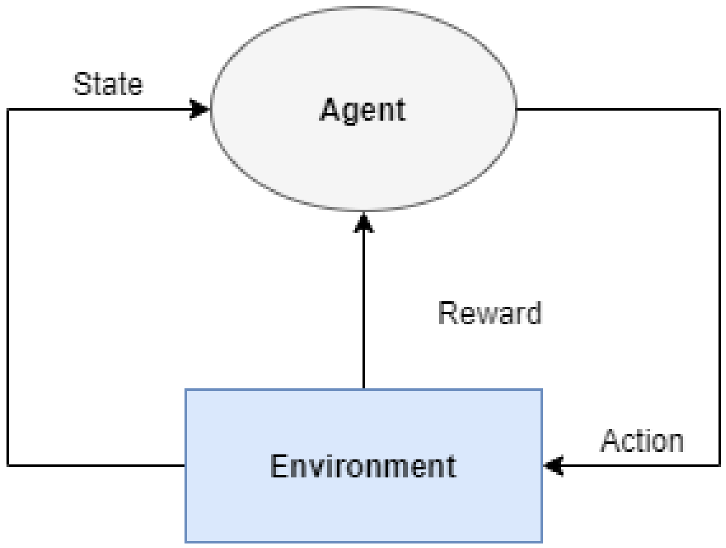


Figure 3.4: Reinforcement learning algorithm.

Supervised and unsupervised algorithms are used in the system of this thesis.
Today there exists more advanced branches in ML, and NNs is one of them. The field of NN makes it possible to solve more complex tasks better, like for instance image recognition with state of the art performance. The following section presents some main classification algorithms in NN that is utilized in this thesis.

3.2 Neural Networks

Parts of this subsection is taken from [1].

The artificial NNs (ANNs) have truly evolved from a collaboration between neuroscientists and computer scientists, looking into what intelligence is [41]. Tomaso Paggio is a professor at Massachusetts Institue of Technology and started looking at the problem of intelligence around 1990 [41]. He was making computer vision algorithms for detecting faces and people in street scenes. He started collaborating with experimental neuroscientists about how the brain detects faces and people. Neuroscientists have known that the brain's visual system is built up by a hierarchy of areas since the 1960s [41]. Paggio tried to mimic the brain in his model for visual recognition. In Paggio's model, the low levels recognize edges and lines, and the higher ones could turn the edges and lines into object parts and then objects.

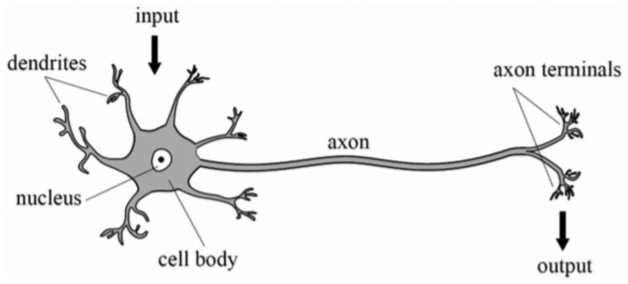


Figure 3.5: Biological neuron [6].

The brain is built up by 100 billion neurons communicating [41]. Figure 3.5 illustrates a biological neuron where dendrites receive the signals, the cell body processes the signals, and the axon sends the signals to other neurons. The neurons in the brain are the ones doing the recognition, and from communicating it among each other, the neurons in the highest level can detect the objects. The word "neural" in NNs comes from the neurons in the brain. The NN is a set of algorithms loosely modelled after the human brain. Figure 3.6 illustrates an artificial neuron from a neural network, which looks quite similar to the biological neuron in Figure 3.5. An example of

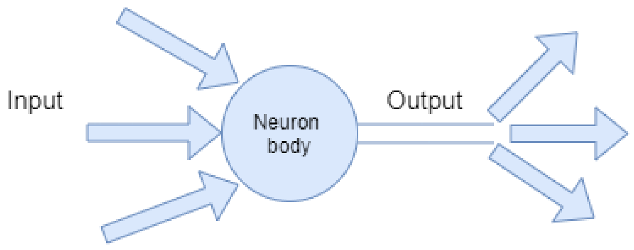


Figure 3.6: Artificial neuron.

a NN can be one for recognizing a person in an image, as what Paggio looked into. The overall goal is to map the input which is an image with a person in it, to the output which is the recognized person.

In 3.1, the ML algorithms acquired knowledge by extracting patterns from raw data. Features for representing the task had to be decided and provided to the ML algorithm. Sometimes these features might be hard to find. To make the algorithm recognize a face, one of the features to look at is the nose. But it is quite hard to explain what a nose looks like. When designing the features, it is also needed to separate the *factors of variation* that explain the observed data [42]. These factors can be looked at as abstractions that are needed to make sense of variability in the data. For instance, when the algorithm is trying to recognize the face from the picture, factors like the angle of the face and brightness of the sun are important. The pixel's colors might look a bit different than they are by night, for instance. Due to this, factors of variation that is important for the specific task needs to be found and taken into consideration. The only problem is that these factors might be quite difficult to extract, and therefore when looking at this as one mapping, the task is very complex. By dividing this mapping into smaller, nested mappings, the task is less complex. The process of division is exactly what Paggio did when he decided to build his vision recognition model based on the brain's hierarchy of areas. The mapping of input to output is done by processing the input by a set of functions, and then pass the output to the next layer.

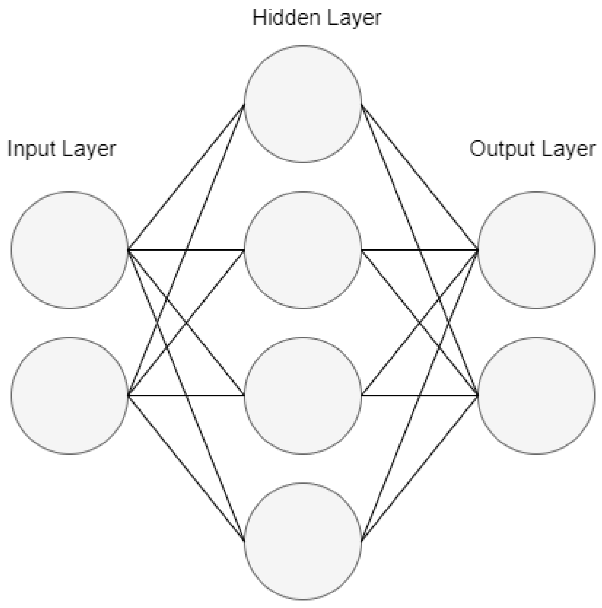


Figure 3.7: Neural network.

A layer, in a NN, represents the state of the computer's memory after executing another set of instructions in parallel [42]. The very first layer of a NN is called the input layer, and the last layer is called the output layer, while the layers in between are called hidden layers, visualized in Figure 3.7. The first layer's task can be to identify edges by comparing the brightness of neighbouring pixels. The output from the first layer is given to the second layer, and the second layer can look for corners and extended contours. By using the second layers explanation of the image by corners and contours, the third layer can find specific collections of contours and corners, which result in entire parts of objects. The fourth layer can use the third layer's description of the image by object parts to recognize the different objects in the image. The overall task of mapping the input image to the output as a recognized object by dividing the mapping into smaller, nested mappings are now solved. This is how NNs generally work by learning a concept at each layer and communicating it to the other layers.

The difference between DL and NNs is the "deep" part, which means that DL has more learned concepts or a greater amount of compositions than NNs [42]. The definition of how many learned concepts or compositions that are needed to be a deep NN is a bit vague. DL is a branch in ML which learns to represent the world by a nested hierarchy of concepts, where each concept is represented by simpler concepts [42]. This way, DL achieves great power and flexibility. The rest of Section 3.2 looks at deep feedforward- and convolutional networks.

3.2.1 Deep Feedforward Networks

To explain what deep feedforward networks are, this section starts by looking at what a perceptron is. The perceptron was truly invented by Frank Rosenblatt in 1957 [43], and is one of the simplest ANN architectures. Rosenblatt's perceptron contributed to the first popularity wave of ANN [39]. A visual representation of Rosenblatt's perceptron is displayed in Figure 3.9. The neurons in a perceptron has numbered inputs, $x_1, x_2, ..., x_n$, with weights, $w_1, w_2, .., w_n$. A linear threshold unit (LTU) sums the weighted inputs and puts the result in a step function. b represents the bias, which is the offset to the origin. The step function is typically a Heaviside function, where the output, y, is dependent on the weighted sum of the inputs. Perceptrons are based on linear models, which means that they cannot learn XOR functionality, for instance. The XOR learning inability, among other limitations the perceptron has, is pointed out by Marvin Minsky and Papert Seymour in the book *Perceptrons: An Introduction to Computational Geometry* [44].

Figure 3.8: Frank Rosenblatt [7].

When flaws like the ones mentioned in Minsky- and Seymour's book where known, it backlashed against the NN approach. Some of the perceptron's limitations, like learning the XOR functionality, can be fixed by introducing the *multi-layer perceptron* (MLP). An MLP consists of stacked perceptrons and is also called feedforward NN. The network is called feedforward because the information flows from the input $\mathbf{x}$, through the intermediate layers with the computations used to define the approximated function f, and at the end, the information goes to the output $\mathbf{y}$. The approximated function, f, is formed by each layer's sub-function, where every layer's function uses the previous layer's function. The layers between the input and the output layers are called

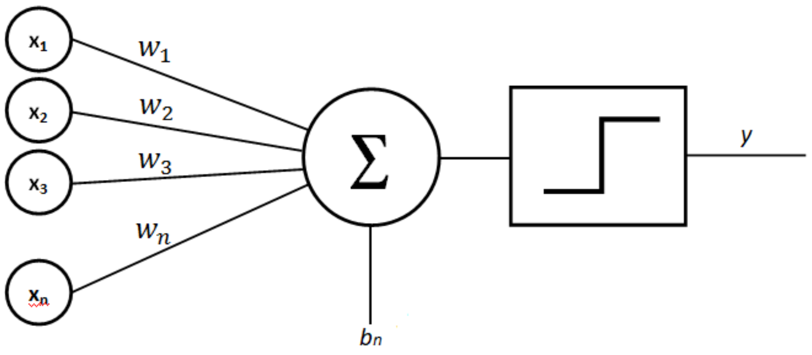


Figure 3.9: Rosenblatt's perceptron [7].

hidden layers because the input does not include a description of what each layer should do to create the output. The hidden layers contain hidden units, and the output of every unit in one layer is connected to the input of every unit in the next layer. Having this connection between the units means that the feedforward network has fully connected layers. Figure 3.7 visualize a fully connected feedforward network. The algorithm itself has to choose what each layer should be to find the best-approximated function. The overall goal of a feedforward network is to approximate some function, f^*, to generate the most accurate prediction of the output based on the input.

$$y = f(\mathbf{x};\boldsymbol{\theta},\mathbf{w}) = \phi(\mathbf{x};\boldsymbol{\theta})^T\mathbf{w}, \tag{3.1}$$

is a mathematical representation of the feedforward NN. The parameters $\boldsymbol{\theta}$ are used to learn the hidden layer's function, ϕ. $\mathbf{w}$ is mapping the learned function, $\phi(\mathbf{x})$, to the output, y. The hidden layers functions can be called *activation functions*. The activation function computes the layer's values. In today's NNs, it is normally recommended to use the rectified linear unit (ReLU) as an activation function. A ReLU computes a linear function of the inputs and outputs the result if it is positive, and 0 otherwise [42]. For the feedforward network to be able to learn, the gradients of complicated functions are needed. This is called *gradient-based learning*.

To train NNs, iterative, gradient-based optimizers that derive the cost function to a low value are usually used [42]. That is, the training algorithm is based on using the gradient to descend the cost function for the feedforward NN. An efficient way of computing the gradient is built on the mathematical chain-rule concept. This principle is called the *back-propagation* algorithm, which was presented in 1986 by David E. Rumelhart and other researchers [45]. After this algorithm was introduced, NNs gained popularity and had a peak in the 1990s. Today's feedforward NN has approximately the same back-propagation and approach to gradient descend as in the 1980s. The feedforward NN itself does not use the back-propagation algorithm, but the back-propagation algorithm uses the feedforward NN. The network is used to feed forward the values from input to output, and then the back-propagation algorithm calculates the error and propagates it back to the previous layers. That is, the algorithm goes through the network in reverse to measure each layer's error contribution from each connection. The gradient descend is used after the back-propagation algorithm to adjust the weighted connections to reduce the overall error. The error that propagates back is found by utilizing a cost function.

For NNs, the cost function is usually defined as the cross-entropy between the training data and the model's predictions plus a regularization term. The regularization term in the cost function is used to make the model generalize well. That is, to make the model not only perform well on the training data but also on new instances, avoiding overfitting. Due to this, the regularization term in the loss function penalizes for large weights. One type of regularization that can be used by a broad family of models is called *dropout*. This technique to avoid overfitting randomly ignores or dropout some hidden units in a given layer.

Saying that the NNs are trained using the cross-entropy error is equivalent to the negative log-likelihood. To be able to compensate for the error, the negative log-likelihood is minimized. The minimization can be done by using a gradient descend algorithm.

$$J(\boldsymbol{\theta}) = -\mathbb{E}_{x,y\sim\hat{p}_{data}} \log p_{model}(\mathbf{y}|\mathbf{x}) \tag{3.2}$$

Equation (3.2) is a general form of the cost function for a NN. The form of this cost function depends on the model, indicated by p_{model}. The symbol $\mathbb{E}$ stands for expected value over the subscript's probability distribution.

It is desired to find the gradient, $\nabla J(\boldsymbol{\theta})$, of the cost function with respect to the parameters. The evaluation of the gradient is done by the back-propagation algorithm. Each layer has to change its weights according to a back-propagated error message from the next layer and calculate an error message for the previous layer. The error is calculated using the gradient, and this is done efficiently with the back-propagation algorithm. For instance, if the input and output vectors

$\mathbf{x} \in \mathbb{R}^m$ and $\mathbf{y} \in \mathbb{R}^n$ respectively, and g maps from $\mathbb{R}^m$ to $\mathbb{R}^n$, then g is the activation function defined for each hidden unit. A function f maps from $\mathbb{R}^n$ to $\mathbb{R}$, $\mathbf{y} = g(\mathbf{x})$ and $z = f(\mathbf{y})$. The chain rule can then be used in the back-propagation algorithm to get the gradient as follows.

$$\nabla z = \frac{\partial \mathbf{y}}{\partial \mathbf{x}}^T \nabla z \tag{3.3}$$

Equation (3.3) shows that the gradient can be computed using the chain rule, which means that the gradient of a variable $\mathbf{x}$ is computed by multiplying the Jacobian matrix $\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$ by the gradient ∇z for each node in the graph [42]. This technique is used to find $\nabla J(\boldsymbol{\theta})$. Normally the back-propagation algorithm is applied to tensors rather than vectors, but the principle is the same as in (3.3) by doing some rearranging in the tensor before running the algorithm.

The SGD is used to perform learning utilizing the gradient found by the back-propagation algorithm. The SGD algorithm finds an estimate of the gradient by finding the average gradient on a minibatch of m examples drawn independent and identical distributed from the data generating distribution [42]. The learning rate is a crucial part of the SGD, and it is gradually decreasing throughout the algorithm until a certain iteration is reached. The iterations after the reached iteration have a constant learning rate.

Batch Normalization (BN) is usually used in the gradient descend algorithm to optimize learning. BN addresses the problem that the distribution of each layer's input changes during training [39]. This happens because the parameters of the previous layers change. The BN operation is done before the activation function in each layer. The operation includes scaling and shifting of the layer's input. BN makes it possible for the NNs to have a larger learning rate and make them less sensitive to weight initialization. Without using the BN, the exploding gradient problem could occur, which can make the learning unstable. The gradient is found by looking at the difference between the predicted values and the actual values, which means that if the error is large, the gradient gets big and could "explode." The opposite of the exploding gradient is called the vanishing gradient and occurs when the gradient is vanishingly small, which could prevent the weights from changing its value. In the worst case, the network will stop training.

How the feedforward neural network operates and how it learns by using gradient-based learning has now been presented. The deep feedforward network is a fully connected network, and therefore it has many parameters to tune for complex data. The introduction of Section 3.2 mentions an example of recognizing a person in an image. This works fine with deep feedforward NN if the image is small, but with larger images, the network breaks down [39]. To perform image recognition on larger images, a specialized kind of deep feedforward network called convolutional networks can be used.

3.2.2 Convolutional Networks

The human perception of differentiating objects seems effortless, but for a computer, this task is extremely complex. The perception happens outside the human's consciousness, within specialized visual, auditory, and other sensory modules in our brains [39]. This type of NNs, which also goes under the name *convolutional NN* (CNNs), is specialized in processing data that has a known grid-like topology [42]. The CNNs are typically used for processing images, which has a 2D grid of pixels and has occurred from studying the visual cortex of the brain [39]. These networks perform better on image recognition for large images than the deep feedforward NN because it has partially connected layers. This way the CNN has fewer parameters to tune than the feedforward NN. The CNNs also uses convolution instead of matrix multiplication, as the deep feedforward NNs uses, in at least one of the layers. The neurons in the first convolutional layer are not connected to every pixel in the input image. Instead, it is just connected to neurons located in a small rectangle of the input image [39]. Every convolutional layer's neurons in the CNN is only connected to a small rectangle of the neurons in the previous layer. For a layer to have the same height and width as the previous layer, *zero padding* is used around the input.
The input of a convolution in ML is usually a multidimensional array of data. The kernel is usually a multidimensional array of parameters that the learning algorithm adapts. The kernel represents the neuron's weights, and can also be called a *filter*. A multidimensional array will, from now on, be called a *tensor*. The discrete convolution between a filter and an image can be represented as done below.

$$s(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n) K(m,n) \tag{3.4}$$

In (3.4), the input is the image I, and the kernel is also two-dimensional and is represented by K. The convolution has a commutative property because the kernel in this example is flipped relative to the input. This is a property that is not needed for the implementation of NNs. Due to this, the term convolution is used, but the networks are normally using *cross-correlation*, which is the same as convolution, except it does not do the flip operation. In this thesis, the same convention is used, and it is specified if the kernel is flipped. A mathematical representation of the convolution without a flipped kernel, the cross-correlation, is explicitly expressed as follows:

$$s(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n). \tag{3.5}$$

This convolution is used in the layers to recognize patterns like edges, object parts, full objects, along with others. The kernel is convolved with the input in the convolutional layer, and the result, $s(i,j)$, is the output that is given to the next layer. A layer of neurons using the same filter provides a *feature map* where the pixels in the image that are similar to the filter are highlighted [39]. Multiple feature maps like this are stacked upon each other and compose a 3D convolutional layer. The layer does multiple convolutions simultaneously to its input, where each feature map has its weights and bias [39]. An image also consists of layers, which are called channels, where a colored image has three layers. One layer for red, one for green and one for blue.
To understand how this discrete convolution works in practice, it can be looked at as matrix multiplication, where the matrix has certain constraints depending on the input.

The interaction between input and output in CNNs is referred to as *sparse interactions* or *sparse weights*. Compared to traditional NNs where every input interacts with every output, convolutional networks have a small kernel that only occupies some of the important pixels, which improves the efficiency.
Another difference from traditional NNs is the concept of *parameter sharing*, which means that each parameter can be used for more than one function in a model in convolutional networks. As mentioned earlier, traditional NNs use matrix multiplication between the input matrix and the weight matrix. This means that each element in the weight matrix is only multiplied with

one of the input elements, which can be called *tied weights*. This means that a set of parameters needs to be learned for every location. With CNNs, on the other hand, each parameter of the kernel is used by every input parameter, with a few exceptions. By using this method, the network only needs to learn one set of parameters, which reduces the storage.
The CNNs also have a *equvariant representation*, which means that if the input changes, the output changes the same way. For example, if the input is shifted by time, the output is the same as before but shifted the same amount of time as the input.
A convolutional layer typically consists of three stages. The first stage does the convolutions in parallel and provides its output, a set of linear activation, to the next stage. Stage two feeds the linear activation into a nonlinear activation function, sometimes called the detector stage. In the last stage, a pooling function is used to for instance make the representation more robust against small translations in the input, called invariance. This becomes handy when it is desired to check if a feature is present rather than exactly where it is. Pooling can also be used to reduce the size of the representation to speed up the computation and to make it possible to use input of different sizes. Different pooling functions exist to make the wanted output for the specific task. A typical pooling function can be max or mean. Say, for instance, the max pooling kernel is of size 2×2, and the input is of size 4×4. When the pooling kernel moves over each quadruple of pixels, it only keeps the maximum value. The pooling kernel that is sent to the next layer will, therefore, consist of 4-pixel elements, where each represents the maximum of its quadruple in the input. Seventy-five percent of the input is dropped in the output. The neurons in the pooling layer do not have any weights. The pooling layer's task is only to optimize the convolutional layer. A clustering algorithm can, for example, be used to dynamically pool features together.

The NNs and basic ML algorithms, described in Section 3.1, go under the category of discriminative modelling. A discriminative model can learn from labelled training data to predict the label of new data. In the next subsection, generative modelling is introduced, and together with discriminative modelling, it is possible to generate new data from a set of training data. This type of algorithm utilizing generative- and discriminative modelling is called GAN and is used in the system of this thesis. Different candidate GANs are also proposed and discussed in the subsection below.

3.3 Generative Adversarial Network (GAN)

As written in a paper by Ian Goodfellow among others,
In the proposed adversarial nets framework, the generative model is pitted against an adversary: a discriminative model that learns to determine whether a sample is from the model distribution or the data distribution [25].
GANs were first introduced by Ian Goodfellow and other researchers at the University of Montreal in 2014 [25]. Ian Goodfellow, shown in Figure 3.10, can, in some way, be looked at as the father of GAN.

Generative Models

The input for a generative model is usually unlabeled, but it can also be labelled. The output of the generative model is a set of pixels, for instance, that have a high chance of being predicted to belong to one of the classes in the training dataset. With generative modelling, it is possible to figure out how the data was generated, and not only make predictions on it as with discriminative models. The generative model is also probabilistic, which means that it does not produce the same output every time. This is accomplished by introducing a random element. A generative model aims are to learn the true data distribution of the training data, and then use this distribution to generate new data that looks like it comes from the same distribution.
The Naive Bayes model builds on the naive assumption that each feature is independent of every other feature [10]. This kind of model estimates the probability of seeing each feature independently. The probability of the Naive Bayes model to generate an observation is calculated by multiplying the probability of the appearance of each feature itself. This model can work well as a generative model for features that are reasonable to expect to be independent. The Naive Bayes model does not work well on raw image data, however. This is because the pixels, which are the features, are not independent of each other. Creating a genera-

Figure 3.10: Ian Goodfellow [8]

tive model for images involves DL for the model to find the features itself and learn the unknown probabilistic distribution function for the features in the training dataset.
Representational learning is used in a generative model to represent an observation, for instance an image, in a lower-dimensional *latent* space. The generator finds a function to map a pixel point in the latent space to a point in the high-dimensional image [10]. Using a latent space representation of the image instead of using raw pixels simplifies the problem and provides better performance. Humans also use a latent space representation when playing the Catch Phrase game, for instance. This is a word guessing game where the player wants their partners to guess the word on the player's card without saying the word itself. For instance, if the word is "apple," the player can describe "batches" of pixels like its shape, color among others, and assume the partners have an idea of how these batches look like in pixels. The player's partners are then mapping the explanation in the latent space to pixels to generate an image of an apple. Latent space also comes handy for manipulations of an image, which we can call latent arithmetic's [10]. For instance, it is possible to change the latent space vector of an image of a smiling person to make the decoded image be the same person but sad. This same technique can be used to morph between two faces. This way, complex problems can be solved by dividing it into smaller and easier problems.

Deep Generative Models

Variational autoencoders (VAE) is a famous and fundamental generative DL model [10]. The VAE is explained by starting to look at a regular autoencoder. An autoencoder can be used to do the mapping from a high dimensional space to a low dimensional latent space and vice versa, as illustrated in Figure 3.11. An image can be encoded to a representation vector in the latent space, which again can be decoded to an image in the high dimensional space. When decoding back to the high dimensional space, some information is lost, which means that it is hard to reconstruct the input image. An autoencoder network can be trained to find weights that minimize this loss of information. Comparing the autoencoder to known models, it can be seen that the decoder is a bit similar to the generative model, and the encoder is a bit similar to the discriminative model. The implementation is also quite the same. To create the autoencoder, a third model has to be implemented where the encoder model's output is taken as input in the decoder model. When this third model is created, it needs to be compiled with an optimizer and loss function, just as for the NN in Section 3.2. The model is fitted to a training dataset with corresponding labels. This model takes an image, passes it through the encoder, and back through the decoder to reconstruct the image again. There are some drawbacks with the autoencoder which can be solved with the VAE.

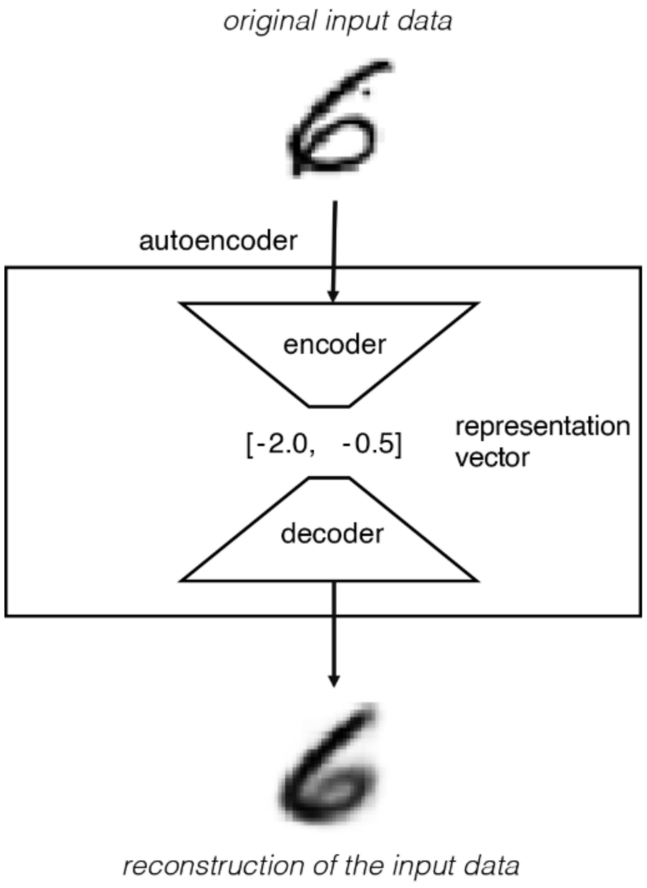


Figure 3.11: Autoencoder diagram [9]

Each image is in the VAE mapped to a multi-rate normal distribution around a point in the latent space instead of just being mapped to a point in the latent space, as was the case for the autoencoder [10]. This introduces a randomness to the model, which results in a point in the latent space that has not been seen by the decoder before to still be decoded to a well-formed image. VAE works fine as a generative model when a low dimensional latent space, for instance, of two dimensions, can be used. With higher dimensions of the latent space, the VAE has trouble performing well. VAE can be used to generate "fake" images of people based on a training dataset with vast amounts of images of "real" people. The images have quite low quality and resolution, but it is still possible to see that it is an image of a person. To generate images of people with a higher resolution, GANs can be utilized.

A GAN consists of two networks, one called the generative network, and the other called the discriminative network. These two networks are adversarial, thus the name GANs. The networks are generative because they are probabilistic, which implies that the model does not produce the same output every time. The generative network generates samples from random noise, while the discriminative network takes both the actual data and the generated samples as input and labels it as "fake" or "real." The generative network wants to generate adequately "fake" data to "fool" the discriminative network to label it as "real," while the discriminative network does not want to be "fooled." This is how the networks are adversarials. If enough data are available, the GAN can converge. The goal is to find an unknown probabilistic distribution that explains why some images are more likely to be found in the training dataset, and others a not [9].
An example of a GAN can be an iterative two-player minmax game with the value function $V(G,D)$, illustrated in 3.6 [25]. The generator, G, minimizes its loss when its generated samples from a latent space vector, z_{latent}, get a probability of one from the discriminator, D, that is $D(G(z_{latent})) = 1$. The discriminator, on the other hand, minimizes its loss when the probability is one for "real" data, $D(x) = 1$, and zero for the generated samples, $D(G(z_{latent})) = 0$. The main principle of GAN is the alternating training of the discriminator and generator, and the aim is to reach convergence. The convergence requires enough capacity, computation time and data.

$$\min_G \max_D V(D,G) = \mathbb{E}_{x \sim p_{data}}\left[\log(D(x))\right] + \mathbb{E}_{z_{latent} \sim p_{z_{latent}}}\left[\log(1 - D(G(z_{latent})))\right] \tag{3.6}$$

In Equation (3.6), p_{data} stands for the actual data variables, and $p_{z_{latent}}$ stands for the noise variables in the latent space.

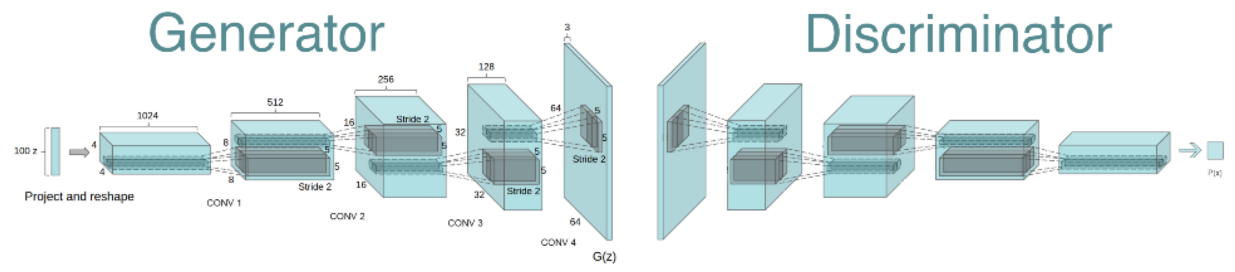


Figure 3.12: The layered architecture of the generator and discriminator of a GAN [10].

In the implementation, GANs are used for generating images GANs. The goal is to be able to generate new sets of pixels that look like they have been generated by the same set of rules as the pixel sets in the training data. The discriminator in the GAN is built with a convolutional architecture, similarly as CNN in Section 3.2. That is, starting with an image, getting the feature maps, and downsampling the image [25]. For the discriminator network, there is just one probability in the output. This probability is one if the discriminator labels the image as "real", and zero if it is labelled as a generated sampled data.
The generative network is built on a transposed convolutional architecture, which is in a way opposite to the convolutional architecture. The generative network does not start with an image but with a latent vector and ends up with an image. *Transposed* convolutional layers are used to increase the dimensionality from the dimensions of the latent vector to the dimensions of the image to be generated. The layers of the generator and discriminator are illustrated in Figure 3.12. The above figure reveals how the generator maps a vector to an image, and the discriminator maps an image to a probability that the image is "real" or "fake" [10]. The generator generates the images only based on the vector, and not by using the training data. The dimensions in 3.12 is for an example of a GANs implementation done for the Canadian Institute For Advanced Research 10 (CIFAR10) dataset [10] and may differ a bit in the implementation in Section 4, but the concept is the same. After GANs was introduced in 2014 [25], many different types of GANs have been proposed.

3.3.1 Conditional GAN

The cGAN is used for image-to-image translation in [12]. The paper is associated with a *pix2pix* software, which has been used of a large number of internet users for experiments like, for instance, art [46]. The cGAN is similar to the GAN explained above but in the conditional setting. Meaning that cGAN learns a conditional generative model instead of just a generative model. The cGAN represented in [12] learns a mapping from observed image x and random noise vector z_{latent}, to y, $G : \{x, z_{latent}\} \rightarrow y$. This network requires paired training data, illustrated to the left in Figure 3.13.

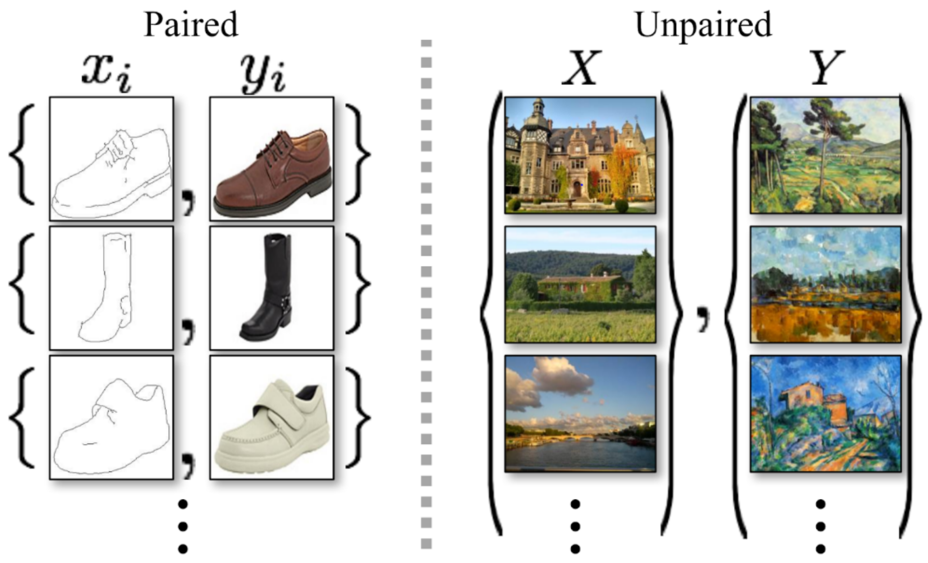


Figure 3.13: Example of paired and unpaired training data from [11].

The training of a cGAN consists of a generator, G, and a discriminator, D. Let's look at an example of mapping from the edged shoe domain, x, to the actual shoe domain, y, shown in the paired training dataset in Figure 3.13, utilizing cGAN. The generator takes an edged image of a shoe together with a random noise vector z_{latent}, as input, and generates a synthesized image $G : \{x, z_{latent}\} \rightarrow y$. The discriminator takes both the image synthesized by the generator as well as the edged image, x, as input to determine if the synthesized image is "real" or "fake." In an unconditional GAN, only the discriminator sees the edged image. The discriminator also takes the ground truth image of the shoe, y, as input together with x to determine if it is real or fake.

The objective function for a cGAN can be expressed as

$$L_{cGAN}(G,D) = \mathbb{E}_{x,y}\left[\log D(x,y)\right] + \mathbb{E}_{x,z_{latent}}\left[\log(1-D(x,G(x,z_{latent})))\right]. \tag{3.7}$$

In Equation (3.7) it can be seen that the generator is conditioned by x in the last term.

$$L_{L1}(G) = \mathbb{E}_{x,y,z_{latent}}\left[\parallel y - G(x,z_{latent}) \parallel_1\right]. \tag{3.8}$$

The L1 distance [1] in Equation (3.8) is used together with (3.7), resulting in the following objective function for the cGAN,

$$\min_G \max_D L_{cGAN}(G,D) + \lambda L_{L1}(G). \tag{3.9}$$

The generator's job in a cGAN is not only to fool the discriminator, but also to be as near the ground truth as possible. The L1 distance is therefore used to keep track and regulate how far the generator is from the ground truth. Figure 3.14 illustrates how an edged input image of a

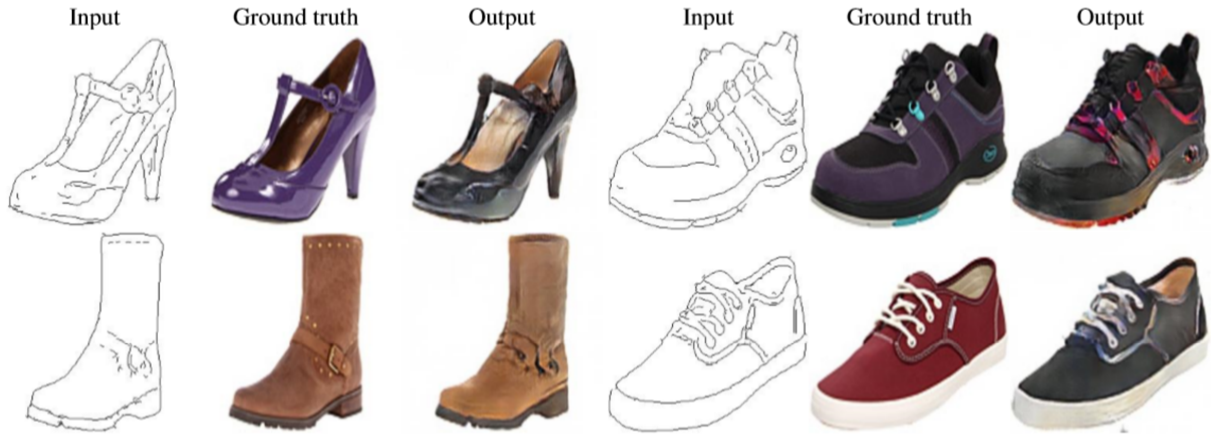


Figure 3.14: Example results from a cGAN automatically detecting edges→shoes, compared to ground truth [12].

shoe is mapped to a shoe that looks a lot like the ground truth.

If paired training data is available, cGAN among several other approaches [47, 48, 49] can be applied for image transformation. It is possible to create a paired dataset by semantically labelling the ground truth image, and using the labelled image as input in the network, for instance. Se-

[1]The L1 distance is the sum of the magnitudes of the vectors in a space.

mantically labelling images is time-consuming and is, therefore, not an optimal choice. A network that can process unpaired training data is thus desirable, which can be done in a cycle-GAN.

3.3.2 Cycle GAN

CycleGAN is an approach to learn the mapping between an input image and an output image using unpaired training data and is presented in [13]. An example of an unpaired training dataset is illustrated to the right in Figure 3.13. For many tasks paired training data is not available, and for these tasks, CycleGAN could be a decent choice. It is possible to create a paired training dataset of unpaired data, but this is time-consuming. CycleGAN can perform image-to-image translation, that is, converting an image between different representations of a scene. Resulting image translation between scenes done by CycleGAN is displayed in Figure 3.15. The

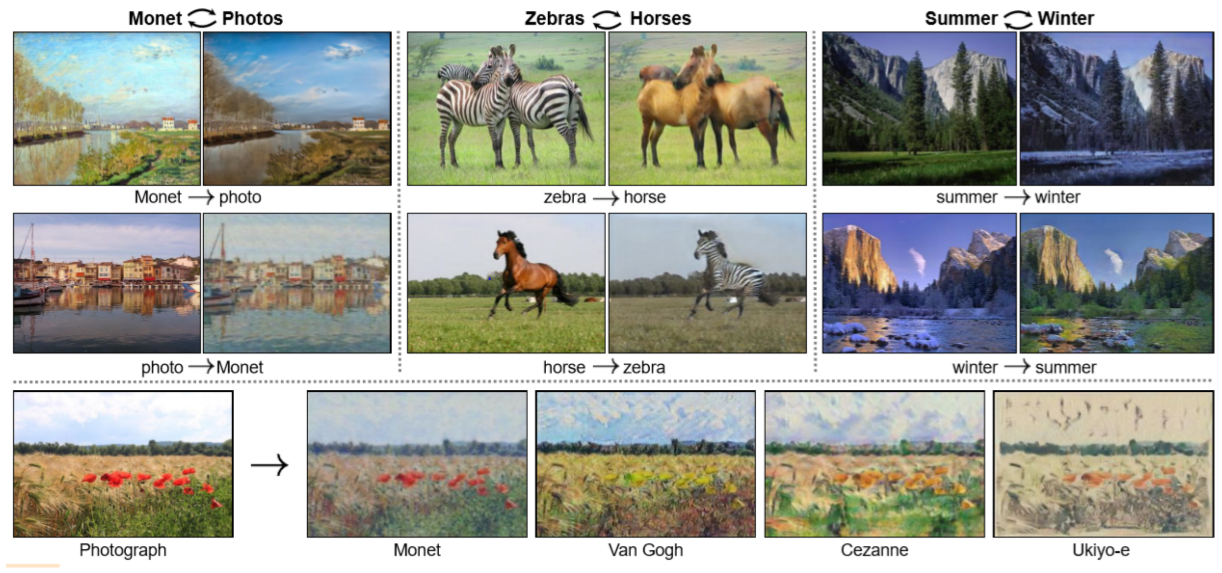


Figure 3.15: CycleGAN results achieved by [13].

network can translate an image of zebras to an image of horses and vice versa, as illustrated in Figure 3.15. CycleGAN manages to work around the paired dataset problem by introducing a mapping function F in addition to the G. The architecture of this type of GAN with an additional mapping function is displayed in Figure 3.16. The added mapping function F also introduces an additional discriminator D_Y, which is labeling y and $G(x)$ as either fake or real. If $G(x)$ is labeled as fake, this function needs to become better at mapping images from domain X to Y.

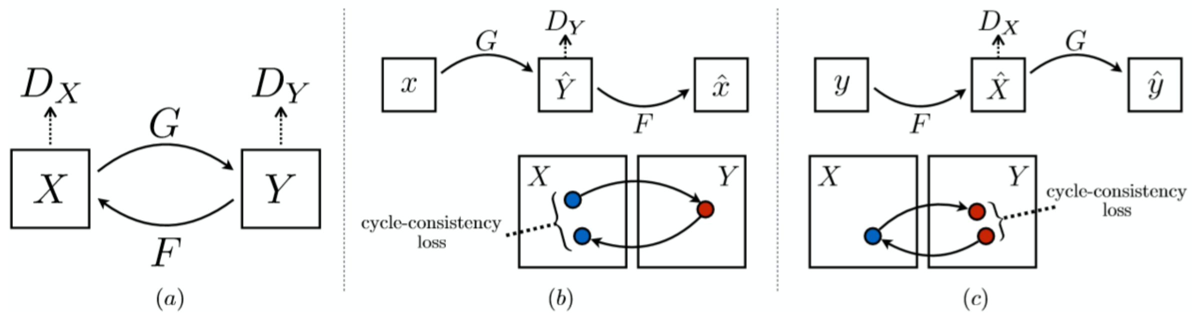


Figure 3.16: The architecture of a CycleGAN [13].

The same accounts for D_X, that is, D_X are labeling x and $F(y)$ as either fake or real. If $F(y)$ is labeled as fake, then the mapping function F needs to improve its performance to be able to "trick" D_X. Thus, by introducing a second mapping function, which indicates that a second discriminator is also needed, CycleGAN manages to do image-to-image translation without the need for paired training data. Sub figure (b) and (c) in 3.16 illustrates how a cycle-consistency loss occurs between the mappings. This loss is pushing G and F to be consistent with each other. The architecture in Figure 3.16 (b) results in the following objective function for $G: X \rightarrow Y$ and its discriminator D_Y,

$$L_{GAN}(G, D_Y, X, Y) = \mathbb{E}_{y\sim p_{data(y)}}\left[\log(D_Y(y))\right] + \mathbb{E}_{x\sim p_{data(x)}}\left[\log(1 - D_Y(G(x))\right]. \tag{3.10}$$

The objective function for $F: Y \rightarrow X$ illustrated in 3.16 (c) and its discriminator D_X is,

$$L_{GAN}(F, D_X, Y, X) = \mathbb{E}_{x\sim p_{data(x)}}\left[\log(D_X(x))\right] + \mathbb{E}_{y\sim p_{data(y)}}\left[\log(1 - D_X(F(y))\right]. \tag{3.11}$$

A *cycle consistency loss* is introduced to make the mapping functions cycle-consistent, and can be expressed as follows:

$$L_{cyc}(G, F) = \mathbb{E}_{x\sim p_{data(x)}}[\parallel F(G(X)) - x \parallel_1] + \mathbb{E}_{y\sim p_{data(y)}}[\parallel G(F(y)) - y \parallel_1]. \tag{3.12}$$

If the mapping functions are cycle consistent then the cycle-consistency loss illustrated in (b) and (c) in Figure 3.16 is zero. This means that $x \rightarrow G(x) \rightarrow F(G(x)) \approx x$ displayed in (b) in 3.16.

The same accounts for y, and is shown in (c) in 3.16. In Equation (3.12), $\| . \|_1$ represents the L1 norm. The resulting objective function by taking Equation (3.10), (3.11) and (3.12) into account is then

$$L(G, F, D_X, D_Y) = L_{GAN}(G, D_Y, X, Y) + L_{GAN}(F, D_X, Y, X) + \lambda L_{cyc}(G, F). \tag{3.13}$$

The aim of the CycleGAN is to solve

$$\min_{G,F} \max_{D_X, D_Y} L(G, F, D_X, D_Y) = L_{GAN}(G, D_Y, X, Y) + L_{GAN}(F, D_X, Y, X) + \lambda L_{cyc}(G, F). \tag{3.14}$$

This is quite similar to the standard GAN represented in (3.6), but with some adjustments to make it work with unpaired data. CycleGAN is not the only type of GAN that can work with unpaired training data, and another type will be described in Section 3.3.3.

3.3.3 Coupled GAN

A coupled generative adversarial network (CoGAN) to tackle the unpaired setting by learning a joint distribution of multi-domain images is introduced in [14]. The joint distribution is learned by enforcing a weight-sharing that limits the network capacity. The authors reveals successful image transformation with CoGAN between domains for color and depth images, and also on face images with different attributes. Although the framework CoGAN can be used for multi-image domains, the paper focuses on the case of two image domains. The network consists of a pair of GAN, illustrated in Figure 3.17, where each network is responsible for synthesizing images in one domain. The two GANs are forced to share weights during training, which re-

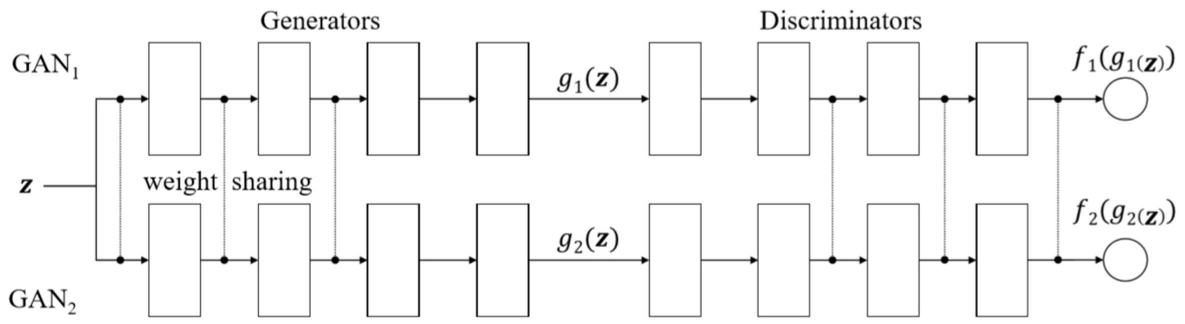


Figure 3.17: The architecture of a CoGAN [14].

sults in the GANs learning to synthesize pairs of corresponding images. Figure 3.17 illustrates how each of the two GANs, GAN_1 and GAN_2, have a generator $g(\mathbf{z})$ for synthesizing realistic images in each domain, and a corresponding discriminator $f(g(\mathbf{z}))$ for classifying if an image is real or synthesized. The vector $\mathbf{z}$ is a random vector like the z_{latent} described above. The weights are tied for the first layers in the generators, which are the layers responsible for decoding high-level semantics, and the last layers in the discriminators responsible for encoding high-level semantics. The joint distribution is learned through this weight-sharing, and it is how the paired-training data setting is tackled. Figure 3.18 displays the resulting images from using

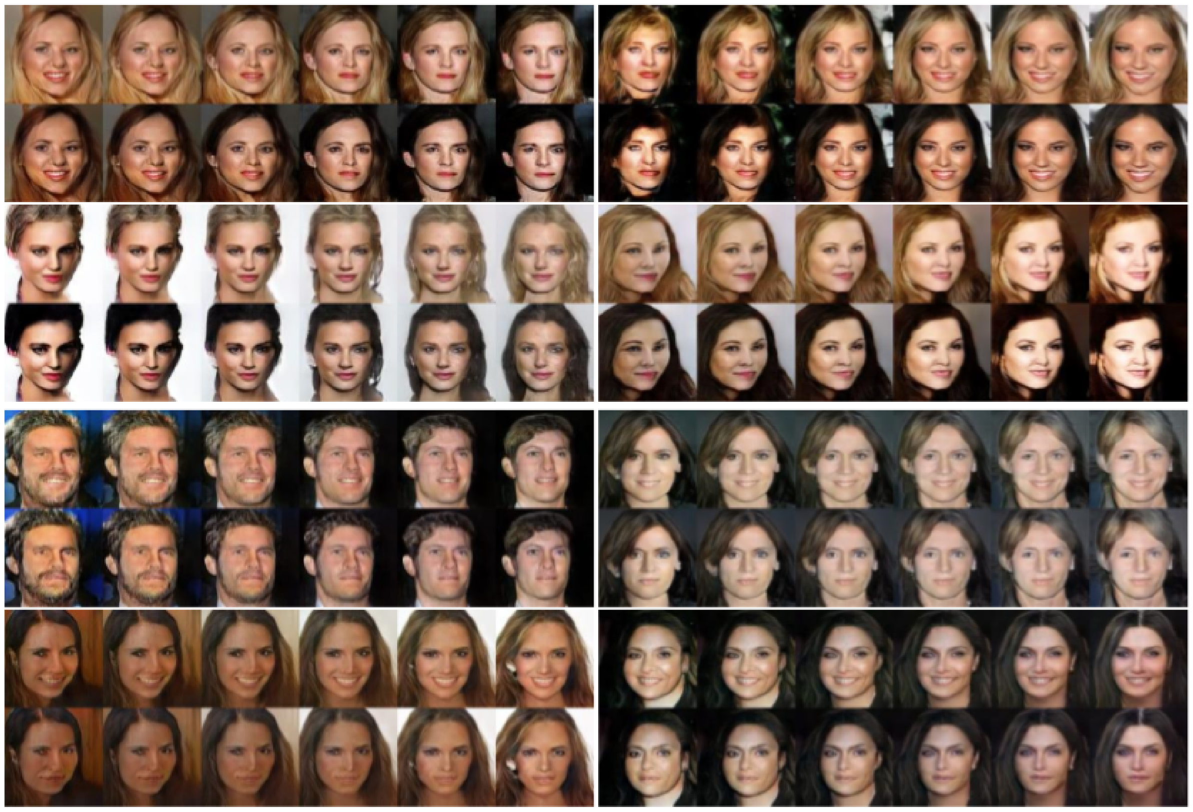


Figure 3.18: Resulting generated images of faces with different attributes utilizing CoGAN [14].

CoGAN. The generated images are pairs of corresponding faces from the same person with and without an attribute. The different attributes represented in Figure 3.18 are pair face generation results for blond-hair and smiling from top to bottom. For every image pair, the first row contains the images with the attribute, and the second is without it. The paper also shows that CoGAN can be applied in the Unsupervised Domain Adaption (UDA), which concerns adapting a classifier trained in one domain to classify samples in a new domain where there is no labelled example in the new domain for re-training the classifier [14].

An image-to-image translation based on the CoGAN framework from [14] and an assumption of the existence of a shared latent-space is proposed in [15]. The framework represented in [15] manages to work around the paired training data issue by assuming a pair of corresponding images (x_1, x_2) in two different domains χ_1 and χ_2 can be mapped to the same latent code z in a shared-latent space Z, shown in Figure 3.19. The encoding functions E_1 and E_2 are mapping the images to the latent code and two generator functions, G_1 and G_2, maps the latent code back to the images. Representing an image in a latent space is simpler than representing the images as the actual image. The features of the image are simpler to represent in the latent space than in the image space and are therefore also simpler to extract. The authors illustrates how this shared-latent space assumption results in an image-to-image translation between domains without any paired training data, displayed in Figure 3.20. Even though the paper reveals positive results, there were also two limitations discovered. The translation model is unimodal due to the Gaussian latent space assumption, which means that there only exists one correct answer. Possible unstable training due to the saddle point searching problem is the second limitation of the proposed framework. The saddle point searching problem is the issue of distinguishing saddle points from minimums and maximums. Another framework for performing image-to-image translation between two different domains is the simGAN.

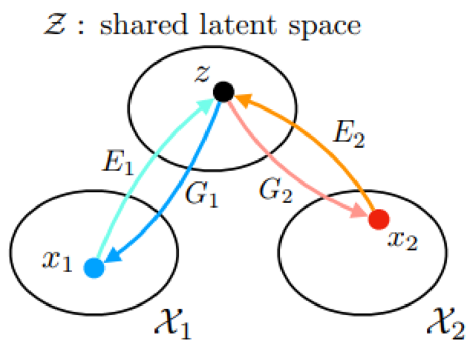


Figure 3.19: The shared latent space assumption [15].

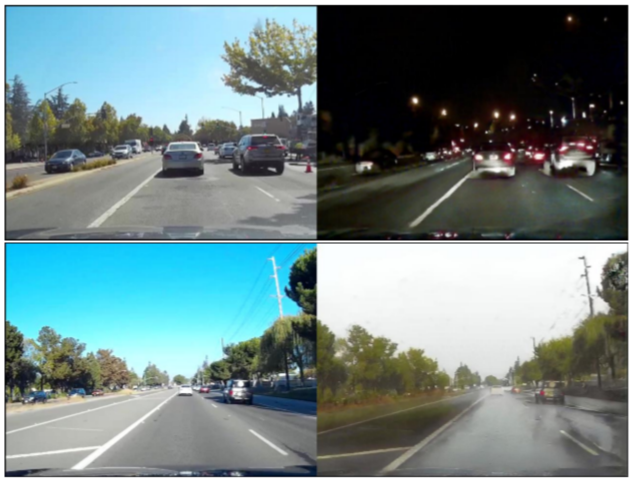


Figure 3.20: Street scene image translation results [15].

3.3.4 Simulated GAN

Apple introduces a method for Simulated + Unsupervised (S+U) learning, which they term simGAN [16].The task is to learn a model to improve the realism of a simulator's output using unlabeled real data while preserving the annotation information of the simulator [16]. The S+U learning uses an adversarial network together with synthetic images from a simulator as input. The annotation information should be preserved for the training of ML models. An example of annotation information of an image of an eye is the gaze direction. Figure 3.21 illustrates an overview of the architecture of a simGAN framework. The figure displays how synthetic images

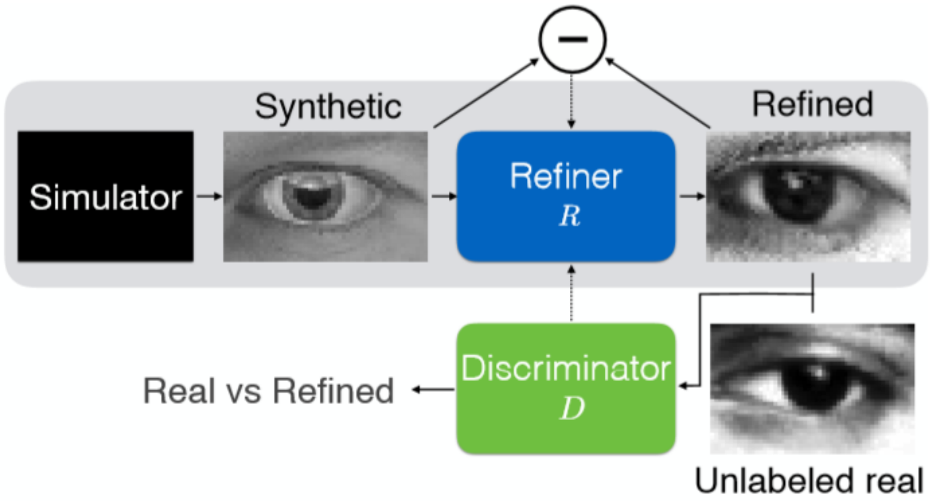


Figure 3.21: The SimGAN architecture [16].

are rendered from a simulator and are input to a refiner neural network, R, which are minimizing the combination of local adversarial loss and a "self-regularization" term. The local adversarial loss tries to "fool" the discriminator, D, that classifies an image as real or refined. The self-regularization term is minimizing the difference between synthetic and refined images. The adversarial loss that trains the refiner network is similar to the regular GANs [25] and should result in the refined images being quite similar to the real images. The refiner network aims to generate refined images that the discriminator cannot distinguish from real images, while the discriminator does not want to be "fooled" by the refiner network. This is how the adversarial loss can train the refiner network and how the architecture in 3.21 is similar to GANs in [25]. The training of the GAN framework is known to be unstable as well as to introduce artifacts [50]. Apple addresses the problem of unstable training by updating the discriminator using a history of refined images instead of only the ones from the current refiner network. The discriminator's receptive field is limited to local fields instead of the whole image to not introduce artifacts, which means that each image has multiple local adversarial losses. The paper reveals state-of-the-art results without any labelled data. The SimGAN has not been used to refine videos instead of images, which are set as future work.

3.3.5 Discussion

The cGAN was tested with weight sharing, which worked fine on grey-scaled images but was outperformed by the CoGAN. Testing cGAN with weight-sharing for color and depth image domains turned out to not even be feasible [14]. Another drawback with cGAN is that paired training data is needed, which is not naturally accessible for this thesis and is time-consuming to generate. CoGAN, on the other hand, does not need paired training data and illustrated promising results even though it had some drawbacks. SimGAN does not need paired training data either, and the results seemed positive. Comparing CoGAN with SimGAN, SimGAN has managed to stabilize the training, where CoGAN could result in unstable training due to the saddle point problem, as stated above. CycleGAN, as with coGAN and simGAN, does not need paired training data either. The three GANs, simGAN, coGAN, and cycleGAN are, therefore, candidates for minimizing the reality gap, which is the purpose of this master thesis.

A comparison of cycleGAN, coGAN, and simGAN, among others where the input is semantically labelled images is performed by [13]. Figure 3.22 displays how different GANs can generate photos that are closer to ground truth by taking semantically labelled images as input.

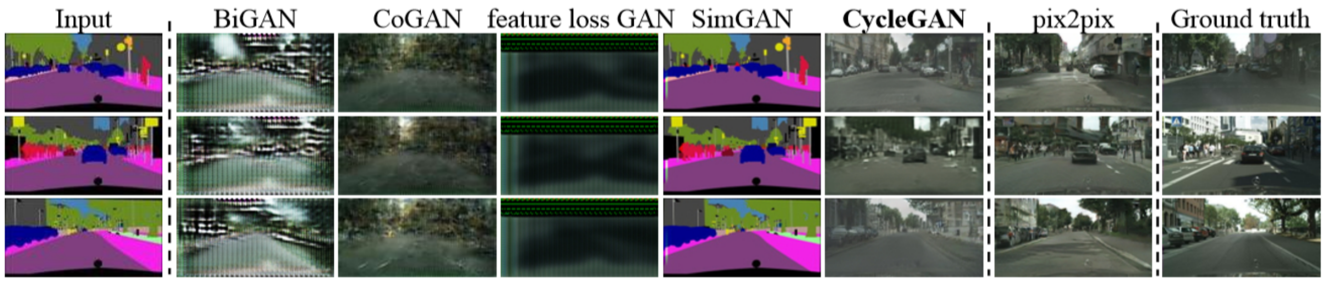


Figure 3.22: Different GANs for mapping labels to photos [13].

Figure 3.22, indicates that the generated images performed by cycleGAN are closer to the ground truth than the images generated by coGAN and simGAN; hence cycleGAN is utilized for this thesis although this comparison is not fair for the simGAN due to the input image. Figure 3.21, illustrates that the simGAN performs well when the input image has more details and is closer to the ground truth. A detailed simulator is accessible for this thesis, and simGAN could, therefore, still be ubteresting to use for minimizing the reality gap.

Object detection and instance segmentation algorithms are utilized in this thesis to check if a model trained on the generated images achieves better accuracy when tested on the real-world images than a model trained on the actual simulator images. The next subsection presents and discusses candidate detection algorithms, and one is chosen for the system. The chosen algorithm is implemented in Section 4.3.

3.4 Object Detection and Instance Segmentation

Object detection is a more complex task than object classification, explained in Section 3.2, due to the required accurate localization of objects. Various candidate object locations often called "proposals," must be processed, and these candidates must be refined to accomplish precise localization. Solutions to finding accurate localization of objects often compromise speed, accuracy or simplicity.

Instance segmentation requires the object detection part described above as well as accurately segmenting each instance. In semantic segmentation, the goal is to classify each pixel into a

fixed set of categories without differentiating object instances [20].
This section starts by looking into different models for object detection like R-CNN, Fast R-CNN, and Faster R-CNN. Followed by reviewing a unified model between semantic segmentation and Faster R-CNN called Mask R-CNN. At the end of this section, the different methods are compared, and the final methodologies that are used in this thesis are stated.

3.4.1 R-CNN

A Region-based CNN (R-CNN) approach to bounding-box object detection is introduced in [17]. The R-CNN extracts a manageable amount of region proposals and then evaluate CNNs, described in Section 3.2, independently on each region of interest (RoI). After computing the CNN features, a class-specific linear support vector machine (SVM) [2], is utilized to classify each region. A visual representation of the R-CNN is illustrated in Figure 3.23, where the process is divided into four steps. The first step is just the input image, and then this image is divided

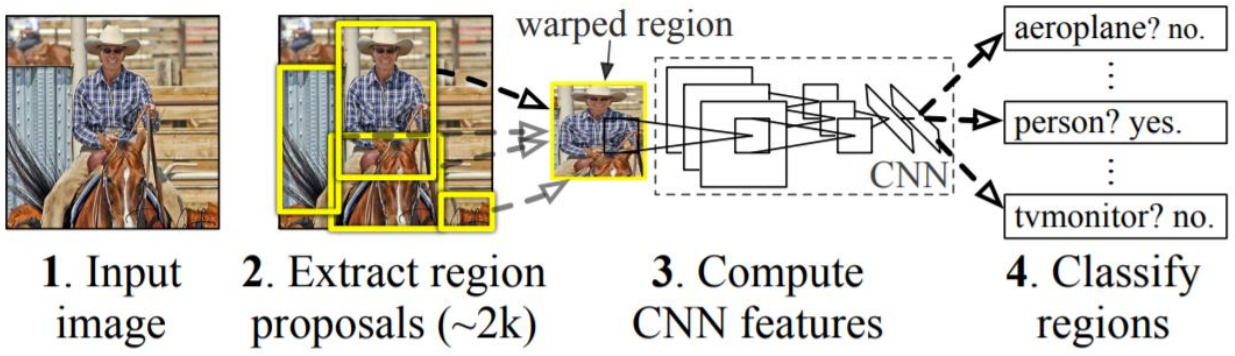


Figure 3.23: R-CNN architecture [17].

into regions in step two. In step three, the CNN features for each RoI are computed, and on the fourth step the different regions are classified. Although R-CNN achieves exemplary results, the framework still has notable drawbacks like expensive training in space and time. The training is a multi-stage pipeline, and the object detection itself is slow. The R-CNN framework was introduced in 2014, a year later, Microsoft Research introduced an improved method called Fast R-CNN.

[2]SVM is a supervised machine learning algorithm, which can be utilized for both classification and regression problems. The algorithm is kernel based meaning that the input data does not need to be linearly separable.

3.4.2 Fast R-CNN

A Fast R-CNN for object detection is introduced in [18]. One of the improvements of Fast R-CNN compared to R-CNN lies in the name itself; it is faster. Fast R-CNN also improves the algorithm's detection accuracy. Figure 3.24 displays the architecture of the Fast R-CNN algorithm. A fully convolutional network takes an image as well as multiple object proposals, RoIs, as input. The whole image is first processed by several convolutional and max pooling layers to produce the convolutional (conv) feature map. The RoIs are then individually pooled by an RoI pooling layer that utilizes max pooling, into a fixed-sized feature map. Fully connected layers (FCs) are then utilized to map the fixed-size feature maps into RoI feature vectors. Each RoI has two output vectors, that is, softmax class probabilities and per-class bounding-box regression offsets. These two output vectors are obtained by feeding the RoI feature vector into a sequence of FCs. The architecture in Figure 3.24 is trained end-to-end with a multi-task loss. Each training RoI is

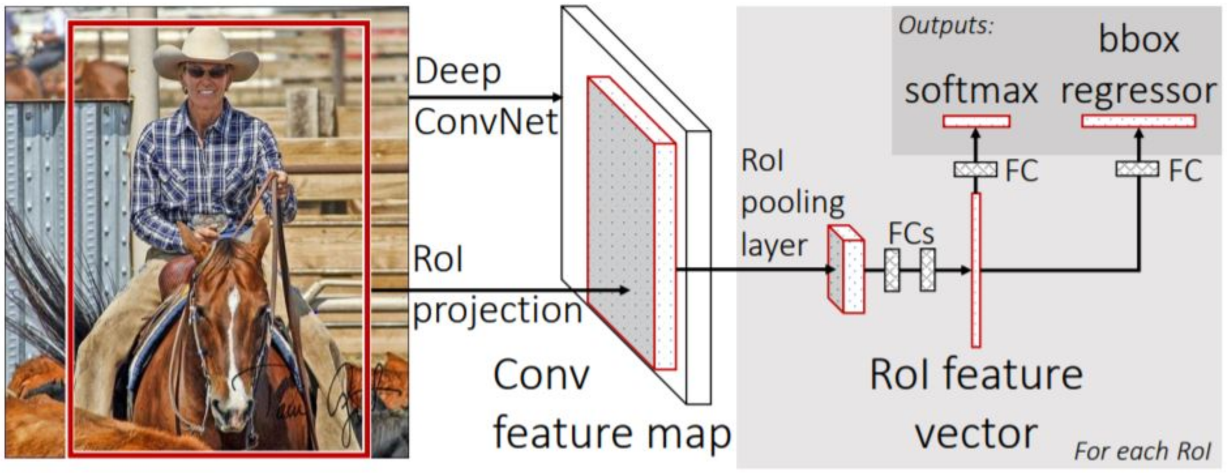


Figure 3.24: Fast R-CNN architecture [18].

labelled with a ground-truth class and bounding box regression target. The multi-task loss is used on each RoI to train for classification and bounding-box regression jointly [18]. The multi-task training makes it possible to avoid managing a pipeline of sequentially-trained tasks, as for the R-CNN. Although Fast R-CNN is an improvement of the R-CNN, there is still room for improvement. The region proposals are the computational bottleneck in the Fast R-CNN object detection system.

3.4.3 Faster R-CNN

A method that improves the Fast R-CNN by changing the way the region proposals are computed is introduced in [19]. The method is called Faster R-CNN and is faster than the Fast R-CNN due to the calculation of region proposals are performed with deep nets called *Region Proposal Networks* (RPNs). The framework uses the conv feature map used in the Fast R-CNN to generate region proposals by constructing RPNs on top of these conv features. The RPNs are constructed by adding one conv layer to encode every conv map position into a feature vector and another layer that outputs an objectness score and regressed bounds for a number of region proposals for each conv map position. In the unification between RPNs and Fast R-CNN, a simple training scheme is proposed. The training scheme alternates between fine-tuning for the region proposal and then for the object detection, while the proposals are kept fixed. The conv features are shared between the region proposal task and the object detection task in this training scheme, and it converges quickly. By using RPNs, the Faster R-CNN manages to compute the region proposals faster than Fast R-CNN. Figure 3.25 illustrates the RPN's architecture, where a

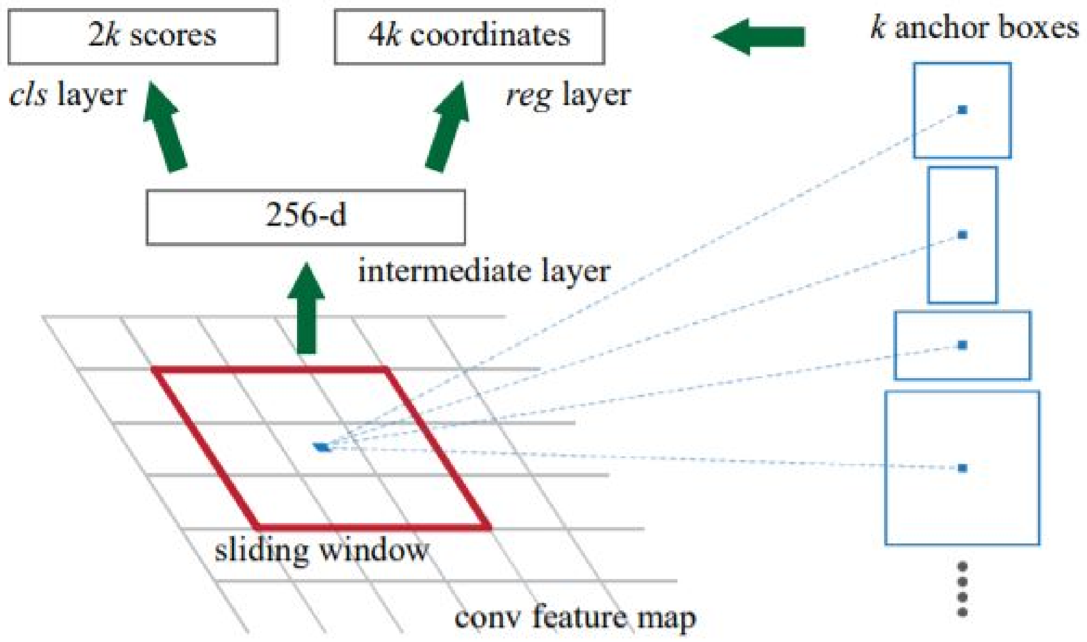


Figure 3.25: Faster R-CNN's RPN architecture [19].

small network slides over the conv feature map to generate region proposals. This small network is fully connected to a sliding window on the conv feature map. A lower dimension vector, 256-d

in the figure, is mapped from each sliding window and is fed into two fully connected layers. The two layers are a box-classification layer and a box-regression layer. The k anchor boxes in the figure are rectangular object proposals. The fully-connected layers are shared across all spatial locations due to the fact that the small network operates in a sliding-window fashion.
Even though Faster R-CNN is an improved version of the Fast R-CNN in many applications with increased accuracy and decreased computational time, there is still room for extensions to the model which has been done in the Mask R-CNN model.

3.4.4 Mask R-CNN

In 2018, Facebook AI Research (FAIR) presented a framework for object instance segmentation that efficiently detects objects in an image while simultaneously generating high-quality segmentation masks for each instance [20]. The Mask R-CNN framework achieves state-of-the-art results on a range of tasks in the field of object detection. The researches merged two state-of-the-art models together and played around with linear algebra. Mask R-CNN combines elements from the object detection task of classifying individual objects and using bounding boxes to localize them and the semantic segmentation task of classifying each pixel into a fixed set of categories without differentiating object instances. An example of Mask R-CNN for instance segmentation done on an image of boys playing football can be seen in Figure 3.26.

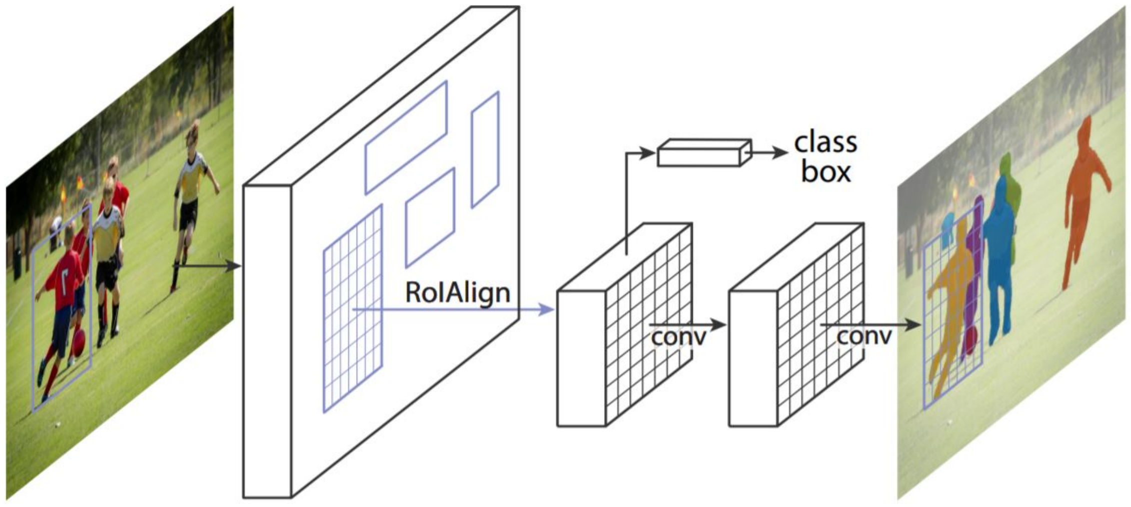


Figure 3.26: Mask R-CNN for instance segmentation [20].

The figure displays an extension of the Faster R-CNN by adding a branch for predicting segmentation masks on each RoI in parallel with the Faster R-CNN's existing branch for classification and bounding box regression [20]. The predicting segmentation masks performed on each RoI in a pixel-to-pixel manner consists of small, fully convolutional networks (FCN). The mask branch only adds a small computational overhead to the Faster R-CNN, which results in the Mask R-CNN being a fast system. The RoIPool layer in the Faster R-CNN is replaced by the quantization-free RoIAlign layer to enable the Mask R-CNN's pixel-to-pixel alignment. The RoIAlign layer that can be seen in Figure 3.26, preserves the exact spatial locations and improves the mask accuracy by relatively 40 percent [20]. The FCN has also been changed by decoupling mask and class predictions, that is, predicting a mask for each class independently and relying on the RoI classification branch for each small FCN to predict the category.

3.4.5 Discussion

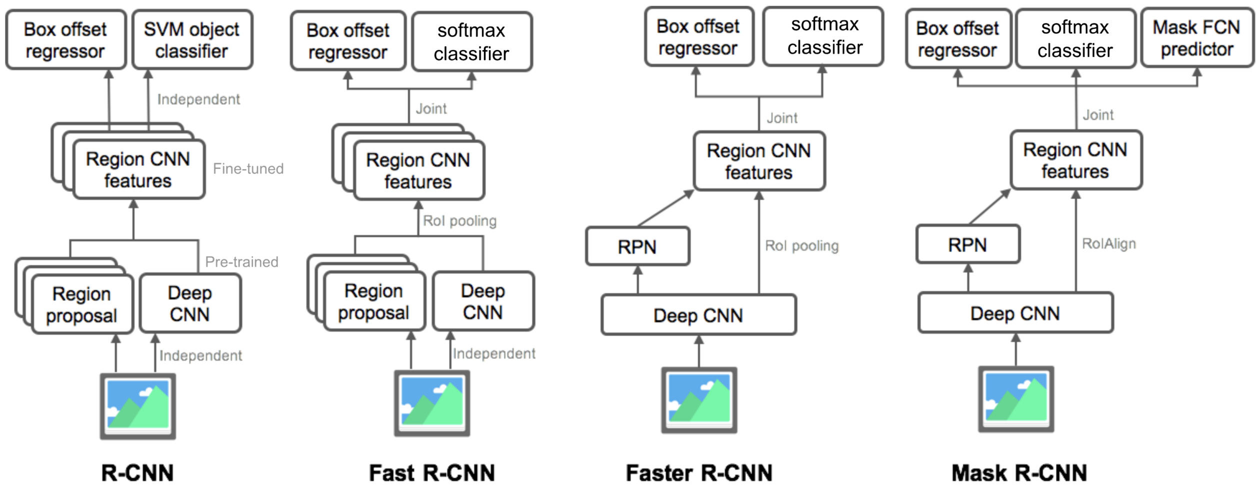


Figure 3.27: Comparison of models in the R-CNN family [21].

Figure 3.27 illustrates an architecture-based comparison between the models in the R-CNN family. It can be seen that the difference between R-CNN and Fast R-CNN is the SVM object classifier in the R-CNN, which is replaced by the softmax classifier in the Fast R-CNN. Going from Fast R-CNN to Faster R-CNN is achieved by removing the region proposal stage performed in parallel with the CNN and adding an RPN stage in sequence with the CNN. Adding a mask FCN predictor in parallel with the box offset regressor and the softmax classifier in the Faster R-CNN architecture results in the Mask R-CNN architecture. It is known that the computational speed

is increasing going from R-CNN to Faster R-CNN. The Mask R-CNN adds a small computational overhead to the Faster R-CNN, which results in the Mask R-CNN still being a fast system. It is also known that the accuracy is increasing in several applications going from R-CNN to Mask R-CNN. Even though the Mask R-CNN is an extension of the Faster R-CNN, it is still simple to train and it is easy to generalize to different tasks [20]. Due to the state-of-the-art results achieved by the Mask R-CNN algorithm, it is utilized in this thesis for checking if the generated simulator frames have added value compared to the original simulator frames.
The cycleGAN is chosen as the methodology for synthesizing photo-realistic images from the ReVolt simulator. The methodology for evaluating the added value of synthesizing these photo-realistic images with GANs is performed by utilizing Mask R-CNN. XAI is utilized to check the quality of the quantitative results from the Mask R-CNN, and its theory is described in the below section.

3.5 Explainable AI (XAI)

Taking action in, for instance, the autonomous ReVolt vessel project based on a prediction, cannot be done without understanding the model's reason for making that prediction. If the model bases its prediction on the incorrect features, it could in worst-case be lethal. Thus, the system needs to assess *trust* to be deployed. In this master thesis, the explanation technique LIME is utilized.

3.5.1 LIME

LIME can explain the prediction of any classifier, thus the model-agnostic part in the name. It is often impossible to achieve a completely faithful explanation. An explanation must at least be locally faithful to be meaningful, although it is important to note that local fidelity does not imply global fidelity and vice versa. LIME learns an interpretable model locally around the prediction. The overall goal of LIME is to identify an **interpretable** model over the *interpretable representation* that is **locally faithful** to the classifier [22]. The interpretable explanations need to be understandable for the user, regardless of the features utilized by the model.
The original representation of the instance being explained is denoted as, $x \in \mathbb{R}^d$, and $x' \in \{0,1\}^{d'}$

is a binary vector for its corresponding interpretable representation. Further, an explanation defined as a model in a class of potentially interpretable models is represented by $g_L \in G_L$. The subscript L is utilized to differentiate the symbols from similar symbols used above in this section. Moreover $g_L \in \{0, 1\}$ acts over absence or presence of the interpretable components. Since g_L must be simple enough to be understood, a measure of the complexity of the interpretable model is defined as $\Omega(g_L)$.

The model that is to be explained is denoted $f : \mathbb{R}^d \to \mathbb{R}$. The prediction made by Mask R-CNN in this thesis, that is, the probability of a segment belonging to a certain class is defined by $f(x)$. Further, $\pi_x(z_L)$ is a proximity measure between the instances z_L and x used to define a locality around x. Finally, a function $\mathscr{L}(f, g_L, \pi_x)$ is a measure of how unfaithful g_L is in approximating f in the locality defined by π_x, where $\mathscr{L}$ represents the fidelity functions. The $\mathscr{L}(f, g_L, \pi_x)$ is minimized while $\Omega(g_L)$ is kept low enough to be interpretable by the user, which ensures both interpretability and local fidelity. Thus, LIME produce the following explanation:

$$\xi(x) = \underset{g_L \in G_L}{\arg\min}\, \mathscr{L}(f, g_L, \pi_x) + \Omega(g_L). \tag{3.15}$$

To minimize the locality-aware loss $\mathscr{L}(f, g_L, \pi_x)$ while keeping the explainer model-agnostic, the loss is approximated by drawing samples weighted by π_x. A perturbed sample $z'_L \in \{0,1\}^{d'}$, contains a fraction of non-zero elements drawn from x' uniformly at random. The perturbed sample recovered in the original representation, $z_L \in \mathbb{R}^d$, is given as input to the classifier f. The classified, $f(z_L)$, is then used as label for the explanation model. By repeating this process for a certain amount of perturbed samples, a dataset Z, of perturbed samples with associated labels is achieved. This dataset is utilized to optimize equation 3.15 and thus get the desired explanation $\xi(x)$.

In Equation (3.15), the explanation functions, G_L, the fidelity functions, $\mathscr{L}$, and the complexity measures Ω is to be chosen for the specific problem. LIME uses a class of sparse linear models as explanation, such that $g_L(z'_L) = w_g \cdot z'_L$, and performs the search using perturbations. Locally weighted square loss is utilized for $\mathscr{L}$, as defined in Equation (3.16).

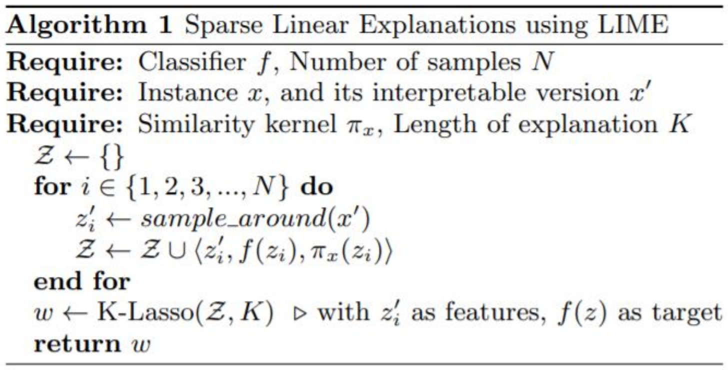
$$\mathscr{L}(f, g_L, \pi_x) = \sum_{z_L, z'_L \in Z} \pi_x(z_L)(f(z_L) - g_L(z'_L))^2 \tag{3.16}$$

The weight, $\pi_x(z_L) = \exp(-D_L(x, z_L)^2/\sigma^2)$, is an exponential kernel defined on the L2 distance [3] function, D_L, with width σ.

The explanation is ensured to be interpretable by letting it be a set of "super-pixels" (computed using any standard algorithm), and by setting a limit K on the amount of these pixels such that,

$$\Omega(g_L) = \infty \mathbb{1}\left[\| w_g \|_0 > K\right]. \tag{3.17}$$

The interpretable representation is thus a binary vector where 1 indicates the original super-pixel and 0 indicates a greyed out super-pixel. Directly solving (3.16) with the chosen Ω in Equation (3.17) is intractable. Thus, it is approximated by first selecting K features with Lasso regression [4] and then learning the weights via least squares (a procedure called K-Lasso in 3.28) [22]. Figure 3.28 is taken from the paper where the LIME algorithm is presented [22], where the subscript L is not needed to differentiate symbols. The complexity of the LIME algorithm depends



```
Algorithm 1 Sparse Linear Explanations using LIME
Require: Classifier $f$, Number of samples $N$
Require: Instance $x$, and its interpretable version $x'$
Require: Similarity kernel $\pi_x$, Length of explanation $K$
  $\mathcal{Z} \leftarrow \{\}$
  for $i \in \{1, 2, 3, ..., N\}$ do
      $z'_i \leftarrow sample\_around(x')$
      $\mathcal{Z} \leftarrow \mathcal{Z} \cup \langle z'_i, f(z_i), \pi_x(z_i)\rangle$
  end for
  $w \leftarrow$ K-Lasso$(\mathcal{Z}, K)$  ▷ with $z'_i$ as features, $f(z)$ as target
  return $w$
```

Figure 3.28: Pseudocode for the LIME algorithm [22].

on the time to compute the prediction $f(x)$ and on the number of samples N. Finally, faithfulness of the explanation on Z is estimated, and present to the user.

Figure 3.29 presents the primary intuition behind the LIME algorithm. The figure shows the actual instance, x, to be explained, by a bold red cross. The sample instances, perturbations, are illustrated by the other red crosses and blue circles. The samples in the vicinity of x have a high weight due to π_x, thus a more significant size in the figure, and vice versa. The red- and blue colored areas in Figure 3.29 represents the global decision boundary which is unknown to the

[3]The L2 distance is the shortest distance between two points and is calculated as the square root of the sum of the squared vector values.

[4]Lasso regression is a type of linear regression that uses shrinkage.

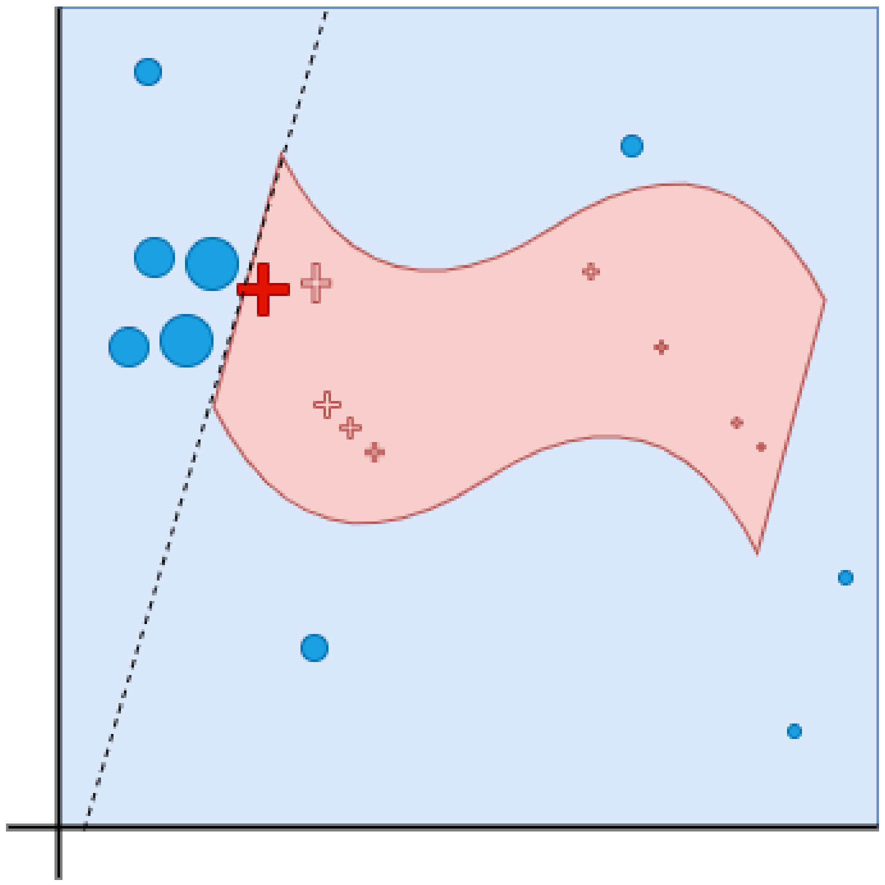


Figure 3.29: Toy example to present intuition for LIME.

LIME algorithm. The dotted line represent the linear explanation presented by LIME which is locally faithful. The line is determined based on the weighted samples, and as the figure illustrates, it is a local decision boundary for the perturbations around the instance x.

The LIME algorithm can only explain the prediction of one class at a time, and Figure 3.30 illustrates the explanation of the classes "Electric Guitar," "Acoustic Guitar" and "Labrador." The resulting explanations show that LIME sets the super-pixels that the model used to make the prediction as "active", and greys out the other pixels. Image (b) in 3.30 provides insight into why an acoustic guitar is predicted to be an electric guitar. Thus, LIME enhances trust in the classifier by showing that it is not acting unreasonably.

CycleGAN, Mask R-CNN and the LIME algorithm constitute the overall system of this thesis and its design and implementation is described in Section 4.

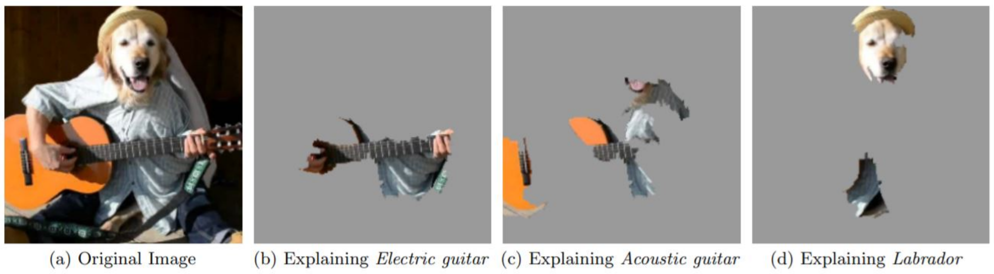

(a) Original Image
(b) Explaining *Electric guitar*
(c) Explaining *Acoustic guitar*
(d) Explaining *Labrador*

Figure 3.30: Explaining an image classification prediction made by Google's Inception NN. The top 3 classes predicted are "Electric Guitar" (p = 0.32), "Acoustic Guitar" (p = 0.24) and "Labrador" (p = 0.21) [22].

Chapter 4

System Design and Implementation

The cycleGAN, Mask R-CNN and LIME are the chosen algorithms for addressing the main objective of this thesis. The theory about these methods is in Section 3. This section presents the data acquisition from the simulator- and the real-world environment, architecture, dataset structure and the implementation of the system in connection with the overall goal of the thesis.

4.1 Data Acquisition

For this master thesis, two types of data are acquired, that is, rendered data of the ReVolt vessel from a simulated environment and actual data of the vessel from the real-world environment. These types of data are retrieved in different ways described more thoroughly in this section.

4.1.1 ReVolt Simulation Data

OBJ File

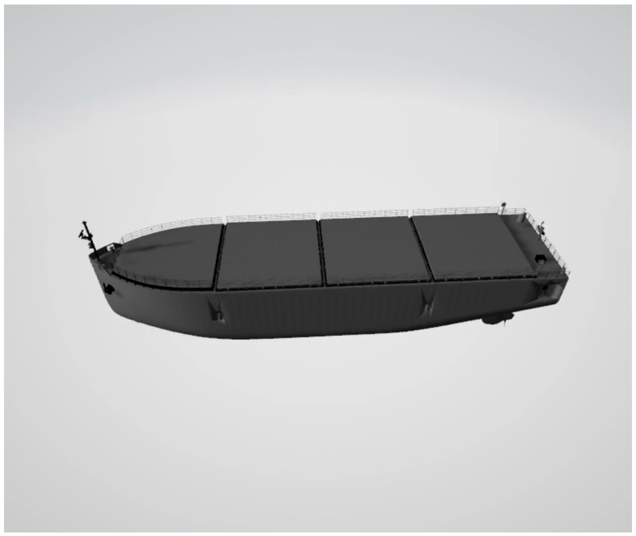


Figure 4.1: Snapshot from the OBJ file of the DNV GL's revolt vessel.

At the beginning of this thesis process, it was not possible to get the DNV GL 3D simulator of the autonomous ReVolt vessel, described in Section 2, up and running due to lack of code. As a result of this, it took some time to get the simulator data, but it was possible to get an OBJ file of the ReVolt vessel. OBJ is developed by Wavefront Technologies and is a format for storing 3D models. The file format contains 3D coordinates, texture, maps, and other object information and can be opened by various 3D image editing programs. This file includes the vessel in 3D with possibilities of rotating it in three DOF, that is, roll, pitch, and yaw illustrated in Figure 2.3. The rotation in three DOF made it possible to retrieve distinctive images from the OBJ file. The first images retrieved looked like the one in Figure 4.1. The OBJ file was then changed to make the vessel look more like the real-world environment, shown in Figure **??**, by making the vessel white and the background dark blue like the ocean. The function *mixed reality* in Windows 3D viewer enabled for the above-mentioned changes, as shown in Figure 4.2.

The OBJ file images are all from footage taken of the OBJ file with the Windows Xbox game bar while rotating the vessel in roll, pitch, and yaw. The footage is then processed in the open-source portable cross-platform media player software VLC Media Player to extract a certain amount of images per second of footage. The rendered images, displayed in Figure 4.1 and 4.2, manages to depict the vessel itself quite well, but they do not look very realistic.

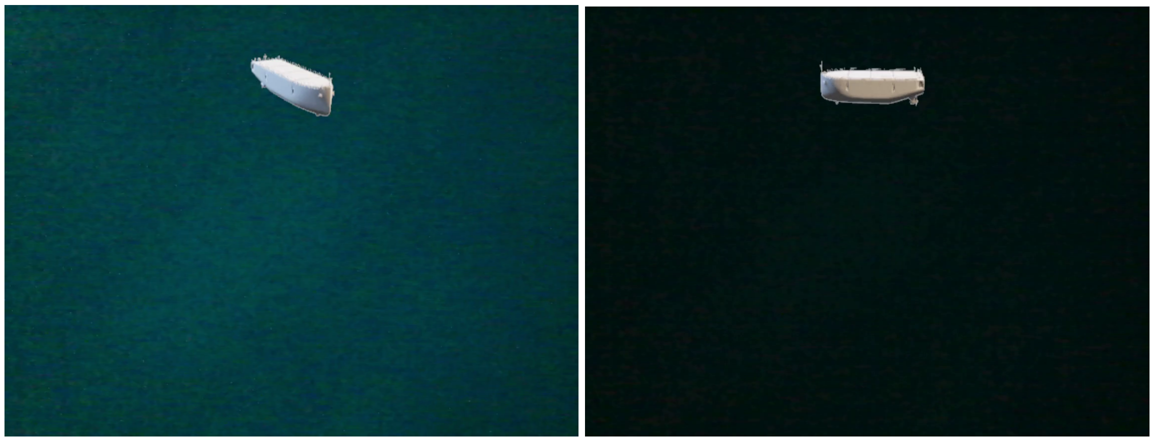


Figure 4.2: Mixed reality of the OBJ file with both light- and dark blue background.

Unity Simulator

The rendered images from the Unity simulator looks more realistic than the one from the OBJ file due to the vessel being submerged in the ocean. Furthermore, the images have a simulated sea and a visualization of the sky. The simulator was also changed in different ways, shown in Figure 4.4, to acquire more distinctive images. The Unity simulator images are rendered through a C-sharp script attached to a game object representing the vessel. The code for rendering snapshots of the simulator's Camera view is written in Unity's *Update*-function, which means that a snapshot is taken on every frame. The rendered images are saved as PNG-files in a specified folder.

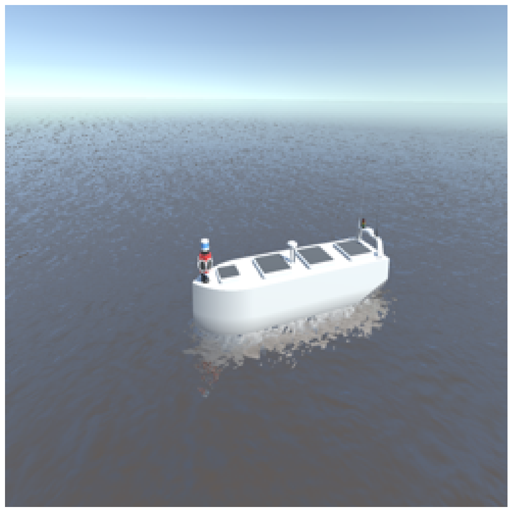


Figure 4.3: Rendered image from the Unity simulator.

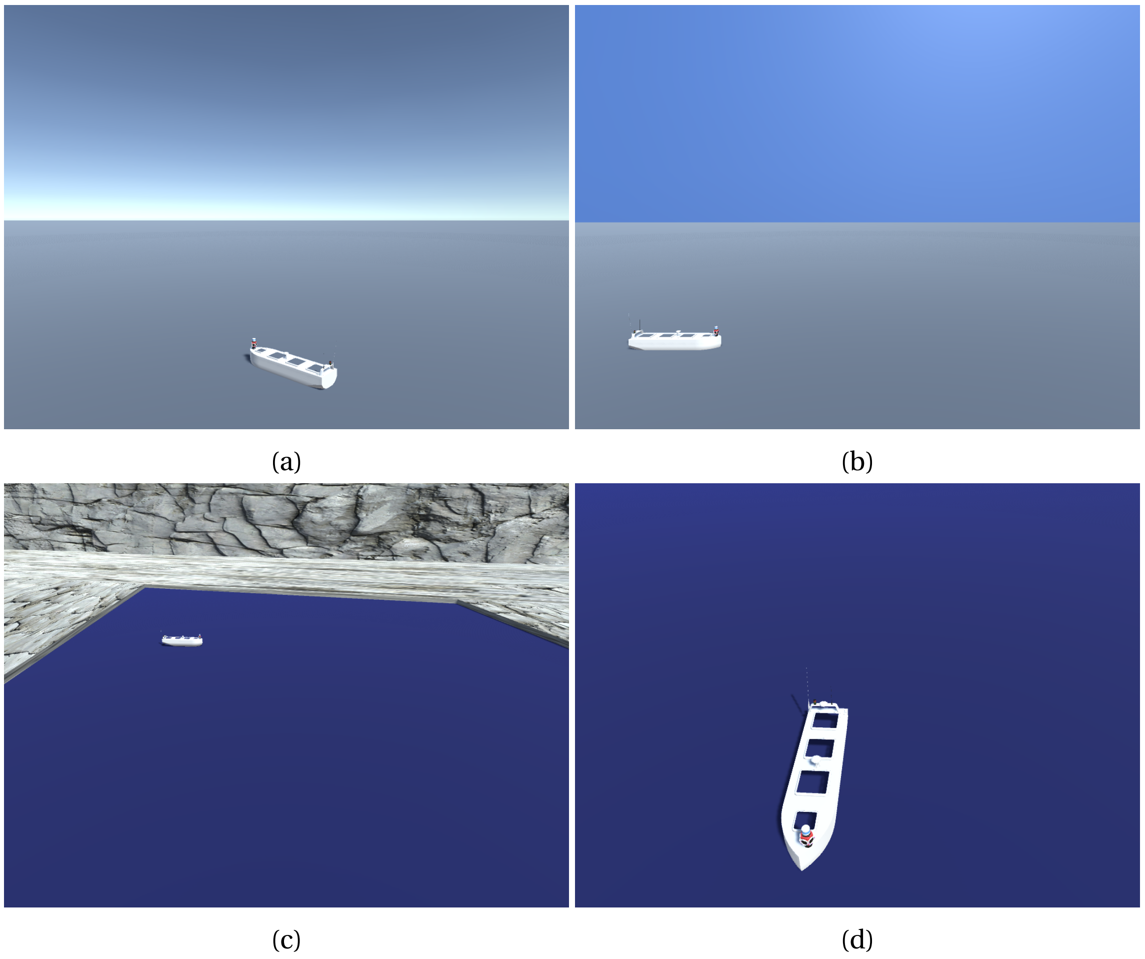

(a) (b) (c) (d)

Figure 4.4: Different Unity simulator scenes.

The game object needs to be able to move to render a vast amount of distinctive images. The game object's movement is enabled by implementing a graphical user interface (GUI) between the arrow keys on the hosting computer's keyboard and the Unity Simulator, referred to as the *controller*.

Controller

The first controller implemented moved the vessel in a circle with a fixed radius for a certain amount of seconds before it was translated a fixed distance and continued moving in circles. This process was repeated while the images were rendered to get distinctive images of the vessel. This approach does not introduce any randomization, and the network can easily detect a pattern.

Due to this, a manually operated forward- and sideways controller (sway and surge) is implemented to introduce randomization in the vessels position. This way, the dataset will not have a specific pattern. The controller is also implemented in Unity's Update-function. The vessel can be controlled manually through the key arrows on Unity's hosting computer. The implementation is done in a C-sharp script and is attached to the game object representing the vessel.

Bounding Box Labels

Bounding box labels for each rendered simulator image is needed to train the object detection algorithm. An area defined by two longitudes and two laterals is referred to as a bounding box. Rendering bounding box coordinates is done through a fitted Unity Collider attached to the vessel game object in the Unity simulator, and the Collider is rendered via a C-sharp script. A Collider is a class in the Unity engine that inherits from Component, which is a base class for everything attached to the game object.

The first bounding box labels were extracted by calculating the eight vertices of the Collider's bounds plus extents. These vertices were transformed from world space to viewport space, that is, from 3D coordinates to the Camera's 2D coordinates. This transformation was performed with the *Camera.WorldToViewportPoint(GameObject)*-function in the same C-sharp script. To enable for the transformed coordinates working in GUI-layout, the y-coordinates were also changed as followed,

$$y = Screen.height - y. \tag{4.1}$$

These transformed vertices are now in 2D in the GUI-layout. The maximum x and maximum y were calculated by looping through the eight transformed vertices. A specified margin was added to the minimum and maximum coordinates to make the bounding box embrace the entire vessel. A rectangle was created using the *MinMaxRect(xMin, yMin, xMax, yMax)*-function, and displayed in the Game view via the command *GUI.label(rectangle, textureToDisplay)* in the *OnGUI()*-function. These bounding box labels did not fit the vessel well; thus, the object detection predictions were not satisfactory.

Consequently, new bounding box labels were calculated by directly extracting the Collider's minimum and maximum bounds coordinates in x,y, and z. These coordinates were also transformed from world space to viewport space through the Camera.WorldToViewportPoint(GameObject)-function in the same C-sharp script. To make the coordinates work in GUI-layout the operation in Equation 4.1 were performed. Those transformed minimum and maximum coordinates are in the 2D GUI-layout, visualized in Figure 4.5, and can be used to create a rectangle around the vessel.

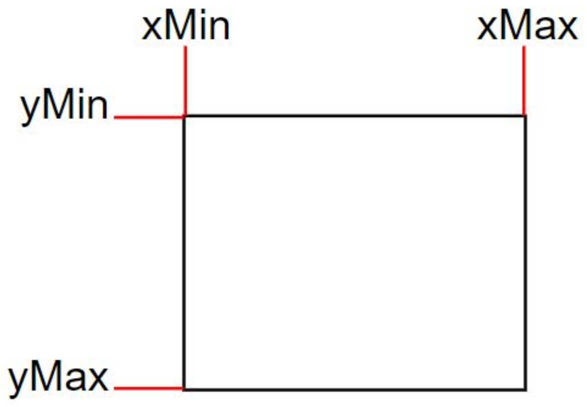


Figure 4.5: The min-max coordinates rendered from the Collider [23].

The rectangle, also referred to as the bounding box, is created by using Unity's *Rect(xMin, yMin, width, height)*-function. The xMin and yMin coordinates are already rendered, and the width and height can be calculated by taking $xMax - xMin$ and $yMax - yMin$ respectively, which can also be interpret from figure 4.5. The created bounding box is displayed in the Game view via the command *GUI.Box(rectangle, '.')* in the OnGUI()-function.

Figure 4.6 illustrates how the created rectangle is fitting the vessel. Since the coordinates for the rectangle is rendered from the game object's Collider, the bounding-box will follow as it moves around in the 3D world space via the controller. Due to lag when changing the vessel's position with the controller and rendering the images, the bounding-box will not always be fitted perfectly around the vessel, as shown in Figure 4.7. As mentioned in Section 2, the Unity engine utilizes the hosting computer's CPU. The Unity simulator needs much processing power, and lag can, therefore, be minimized by not running any other cumbersome processes simultaneously.

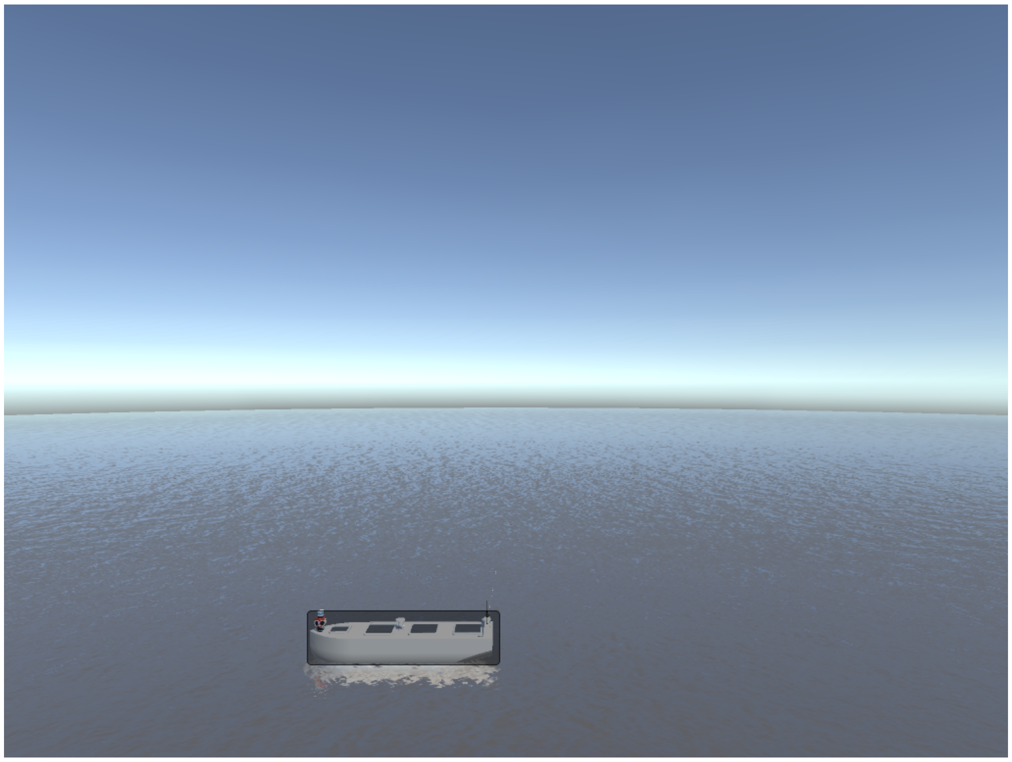


Figure 4.6: A visualization of the bounding-box.

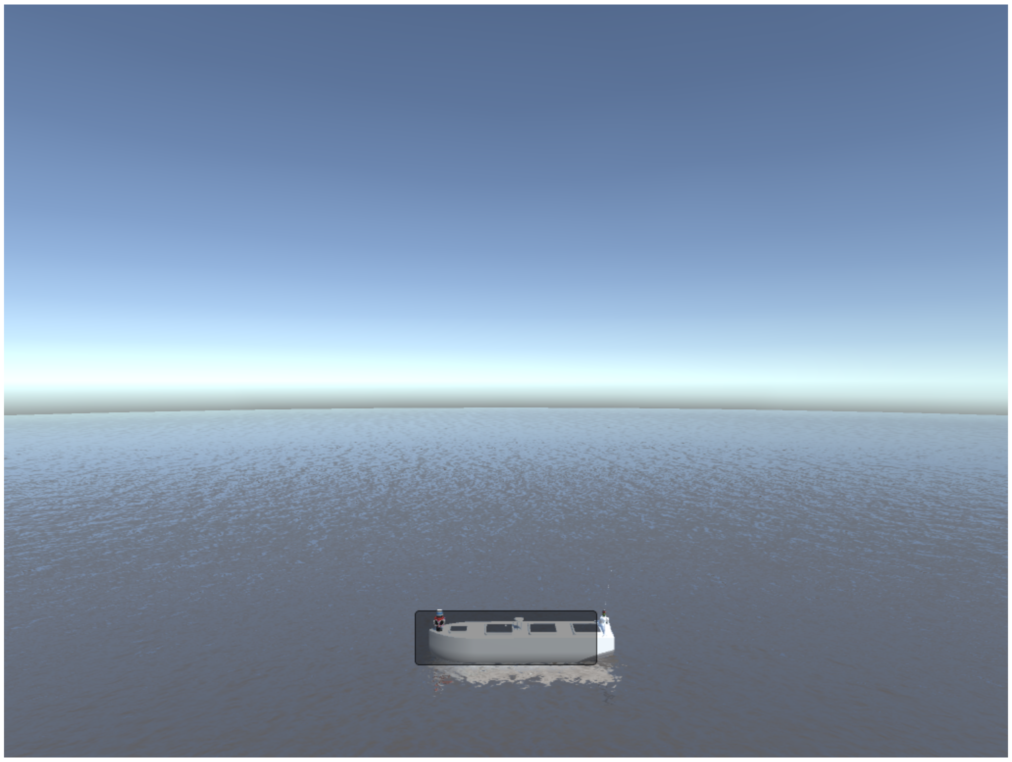
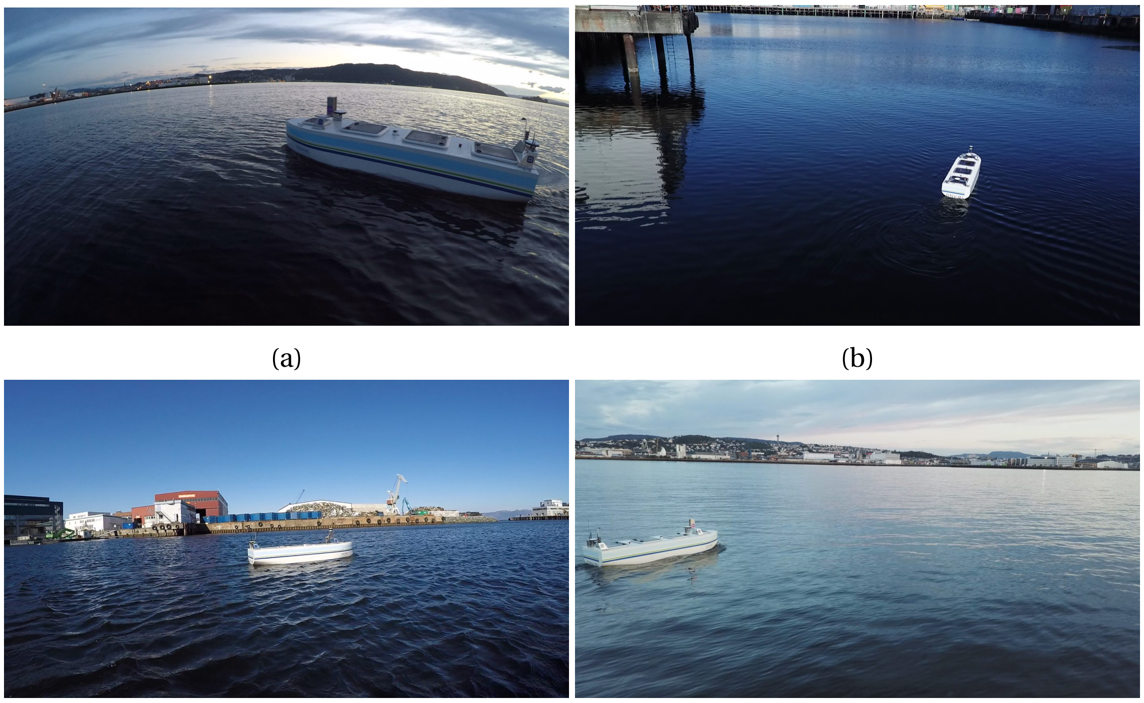


Figure 4.7: A visualization of the bounding-box with lag.

When acquiring the images for the dataset, the bounding box in Figure 4.6 and 4.7 is invisible, and only the transformed minimum and maximum coordinates of x and y are saved together with the corresponding snapshot of the Game view.

4.1.2 ReVolt Real-World Data

The real images of the DNV GL ReVolt vessel are retrieved from drone footage taken in Trondheimsfjorden. As for the rendered images from the OBJ file mentioned above, the real-world images from the drone footage are also acquired by utilizing the VLC Media Player. Drone footage with different factors of variation is utilized to acquire more distinctive real-world images.



(a) (b) (c) (d)

Figure 4.8: Real-world images from drone footage with different factors of variation.

The simulator- and real-world images are utilized as input data to train a cycleGAN network.

4.2 CycleGAN

For the implementation of cycleGAN [2] is utilized. The code is adapted to the purpose of this thesis, and run in Google Colaboratory, described in Section 2, to speed up the training process by running on a virtual GPU. To explain the cycleGAN utilized in this thesis, the overall network architecture is described first, followed by a short description of the dataset's structure . The last subsections will go through the chosen loss function and the implementation of the cycleGAN.

4.2.1 Architecture

Illustration of the architecture for the simulator- and real-world images, is done in two different figures. Simulator images are referred to as Actual A, and the real-world images are referred to as Actual B in the visualization. The first illustration, shown in Figure 4.9, takes a rendered image, Actual A, as input and passes it to Discriminator A and Generator A2B. Discriminator A is labelling Actual A as "real" or "fake," and for Discriminator A, it is desirable to label the real Actual A image as "real." Generator A2B is generating the Generated B image, and it aims to generate B so that it looks like it comes from the actual B data distribution. The Generated B image is labelled as "real" or "fake" by Discriminator B, which aims to label it as "fake" since it is a generated image. Generated B is also passed to the Generator B2A, aiming to generate the cyclic A identical to the Actual A image. The difference between the Actual A- and Cyclic A image is the cycle consistency loss, which is being minimized in cycleGAN's objective function. This objective function is written in Section 3 as Equation (3.13).

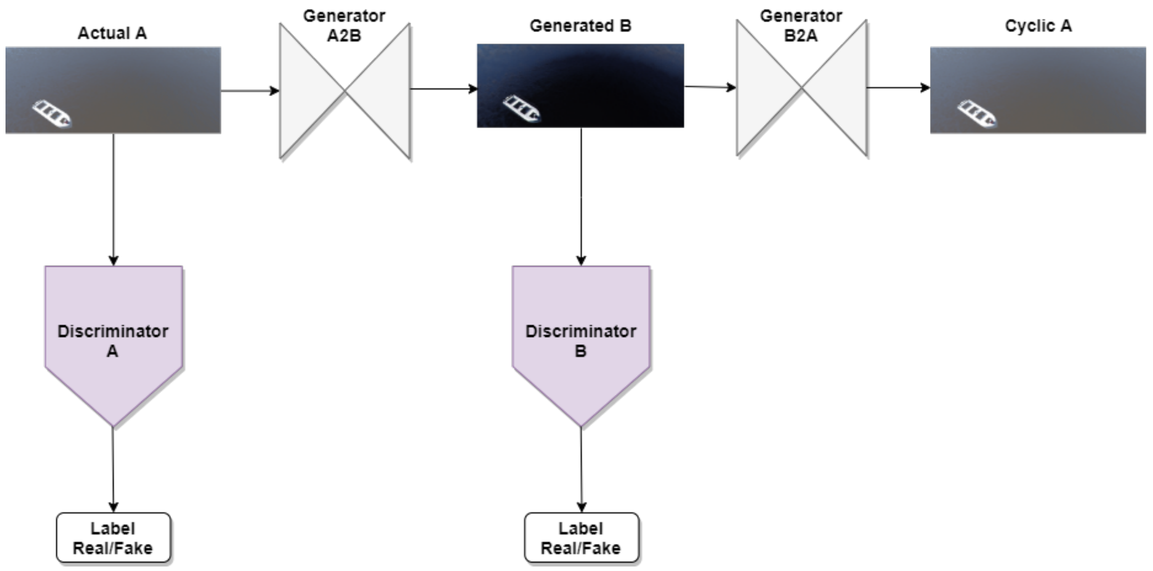


Figure 4.9: First part of the cycleGAN's architecture.

The illustration in Figure 4.10 starts with a real image of the ReVolt vessel, Actual B, which is passed to Discriminator B and labelled as "real" or "fake." Discriminator B aims to recognize Actual B as a real image and label it accordingly. Actual B is also passed to the Generator B2A aiming to generate the image Generated A in a way that it looks like it comes from the same data distribution as Actual A. Generated A is passed to Discriminator A, which aims to recognize this image as generated and therefore label it as "fake." Generator A2B takes the Generated A image as input and outputs the Cyclic B image, which is desired to be identical to the Actual B image. The difference between the Actual B- and Cyclic B image is also a part of the cycle consistency loss, which is minimized in the cycleGAN's objective function.

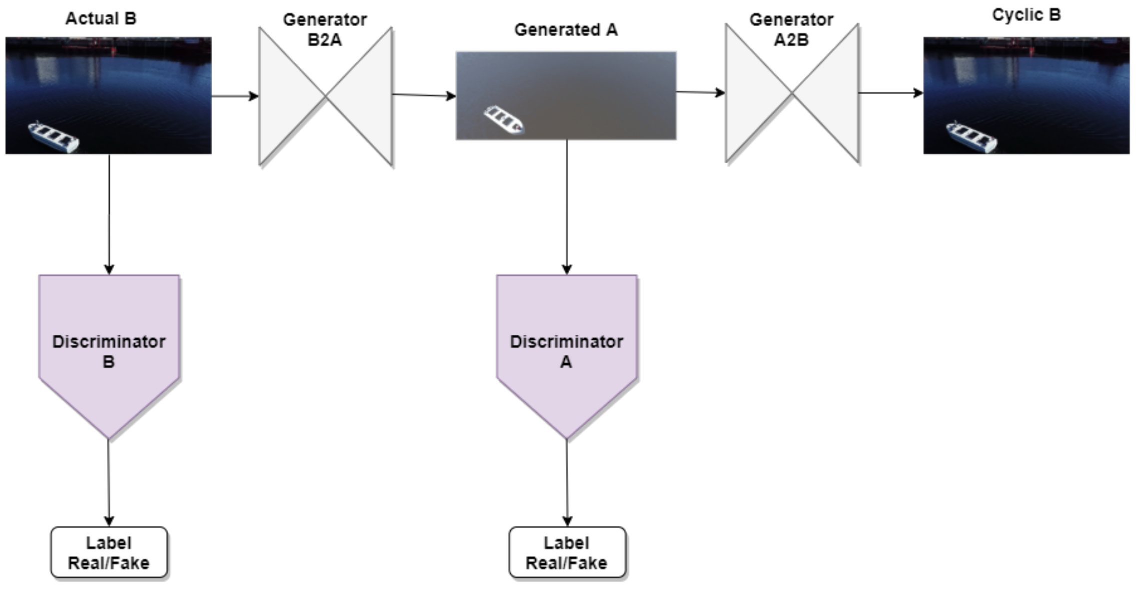


Figure 4.10: Second part of the cycleGAN's architecture.

The process illustrated in Figure 4.9 and 4.10, is repeated for a great amount of images to achieve trained generator networks. Since the aim of this thesis is to create a simulator that is closer to the real-world environment, the trained Generator A2B is the one needed.

4.2.2 Dataset

As mentioned in the theory Section 3, cycleGAN works with an unpaired dataset, that is, a dataset where the same data is not available in both domains. The retrieval process of the images for the cycleGAN's dataset is described above in 4.1. Each dataset contains four folders as follows,

- **trainA**: Containing simulator images in PNG format.
- **trainB**: Containing real-world images in PNG format.
- **testA**: Containing simulator images in PNG format.
- **testB**: Containing real-world images in PNG format.

The first two folders, trainA and trainB, is referred to as the train directory. The last two folders, testA and testB, is referred to as the test directory. The images in the train directory will be different from the images in the test directory. trainA and testA will contain distinctive images from the simulated environment, while trainB and testB will contain distinctive images from the real-world environment.

The first trainB directory contained a mix of real-world images with different factors of variation, which confused the network because it is made for mapping from one domain to another. For instance, cycleGAN can be used to map from a street view by day to the same street view by night. Hence, it does not make sense to try making the network learn to map to different scenes. The trainB directory, therefore, contains images with the same factors of variation.
The dataset's path is specified in the command starting the training and testing.

4.2.3 Loss Function

The loss function utilized in the implementation is the least-squares loss function. The negative log-likelihood objective in Equation (3.10) and (3.11) is replaced by the least-squares loss. This replacement is done to achieve a more stable training and generate images with higher quality. The cycle consistency loss in Equation (3.12) remains unchanged. The aim for a GAN loss,

$L_{GAN}(G, D, X, Y)$, is therefore as follows:

$$\min_D V_{GAN}(D) = \mathbb{E}_{y\sim p_{data}(y)}\left[(D(y) - a)^2\right] + \mathbb{E}_{x\sim p_{data}(x)}\left[(D(G(x)) - b)^2\right], \tag{4.2}$$

and

$$\min_G V_{GAN}(G) = \mathbb{E}_{x\sim p_{data}(x)}\left[(D(G(x)) - c)^2\right]. \tag{4.3}$$

In Equation (4.2), a and b represents the labels for real and fake images respectively. In the implementation the real and fake labels will be represented by 1 and 0 respectively, which means that $a = 1$ and $b = 0$. The value that the generator G wants the discriminator D to believe for fake images is represented by c in the above equation, which is 1. The goal of the training is hence for the generator to minimize $\mathbb{E}_{x\sim p_{data}(x)}\left[(D(G(x))-1)^2\right]$, and for the discriminators to minimize $\mathbb{E}_{y\sim p_{data}(y)}\left[(D(y) - 1)^2\right] + \mathbb{E}_{x\sim p_{data}(x)}\left[(D(G(x)))^2\right]$. This counts for both the sets of discriminator and generator in the cycleGAN.
The architecture in Figure 3.16 (b) results in the following objective function for $G_{A2B} : A \rightarrow B$ and its discriminator D_B,

$$L_{LSGAN}(G_{A2B}, D_B, A, B) = \mathbb{E}_{B\sim p_{data(B)}}\left[(D_B(B))^2\right] + \mathbb{E}_{A\sim p_{data(A)}}\left[(1 - D_B(G_{A2B}(A))^2\right]. \tag{4.4}$$

The objective function for $G_{B2A} : B \rightarrow A$ shown in 3.16 (c) and its discriminator D_A is,

$$L_{LSGAN}(G_{B2A}, D_A, B, A) = \mathbb{E}_{A\sim p_{data(A)}}\left[(D_A(A))^2\right] + \mathbb{E}_{B\sim p_{data(B)}}\left[(1 - D_A(G_{B2A}(B))^2\right]. \tag{4.5}$$

This choice of loss function means that the GAN mode of the cycleGAN model is set to least-square GAN (LSGAN) [51] in the implementation.

4.2.4 Implementation

The cycleGAN is implemented in Python utilizing PyTorch, described in Section 2. Running the code in Google Colaboratory, an environment is needed to be set up where different Python libraries with specific versions are installed and imported. In Google Colab, GPU's are prioritized for interactive users rather than long-running computations. This constrain is a problem when training cycleGAN since its training process is time-consuming. If the training stops after a few hours for some reason, it is essential to be able to continue where it stopped. Checkpoints are therefore saved every 1000 iteration to Google Disk. Images generated during training are also saved to Google Disk every second epoch to be able to see if the network is mapping the most important features.

The code from [2] is a general-purpose implementation and needs to be adapted to the purpose of this thesis. Flags are implemented to enable changing model-specific parameters and adapting the code to a specific purpose. Parameters for training and testing can also be changed via implemented flags. The network uses a default learning rate of 0.0002 as well as the λ in Equation 3.13 set equal to 10. All the models use a batch size of 1, which means that one image from the training dataset is utilized on every iteration. In the training script, the type of model needs to be specified, which in this case is cycleGAN. The specified model's objective functions, optimizations, and network architecture are then loaded during training.

The discriminators utilize PatchGANs, which are looking at patches of the images to determine if they are real or fake. In this thesis, the network looks at 70×70 overlapping patches of the image at a time. PatchGANs architecture enables the network to work on any image size. The discriminator is also updated based on a history of 50 generated images instead of only the images produced by the latest generators, which reduce the oscillations in the model.
A residual network of 9 blocks is utilized for the generators. The generator networks also contain two stride-2 convolutions and two fractionally strided convolutions with stride 0.5. The model's objective function is the least square GAN's objective described in the above subsection. The dataset described above is unpaired, and the model's dataset mode is therefore set to *unaligned*.

There are no random noise vector z_{latent} in this implementation. When adding the random noise, the output is not varying significantly as a function of z_{latent} and is therefore not included [2]. Since the generative model is conditioned on an input image, the random noise vector is not needed as long as the input is adequately complex to play the role of noise. The mapping will be deterministic if the input image is not complex enough, which can be subtle, depending on the purpose of the generator.

Controlling the autonomous ReVolt vessel in a photo-realistic environment created by a deterministic generator, the same scene will be seen when going forward, turning, and going back to the starting point. If the vessel is controlled in a probabilistic generated photo-realistic simulator, the scene will be different going back to the starting point. Although the surrounding shore will most likely not change in the real-world environment, the autonomous ReVolt vessel needs to be prepared for meeting other marine crafts, but the introduced artifacts by the probabilistic generator are random. Thus, it is better to improve the simulator by adding the desired objects like other vessels and use a deterministic generator for making it more realistic.

The implementation also includes a prepossessing of the images. CycleGAN is memory-intensive due to its need for two generators and two discriminators. The images are, therefore resized and cropped. No matter what the resolution of the training images is, the load size is set to 286×286. This way, all the images are loaded into the network with equal size, and then they are cropped to a crop size set to 256×256.

After training the implemented cycleGAN, the Generator A2B will be utilized to generate photo-realistic images based on the images from the simulated environment. Further, the generated images, together with the simulated images, are utilized for training the Mask R-CNN algorithm, described below.

4.3 Mask R-CNN

After generating photo-realistic images based on the simulated environment by utilizing cycle-GAN, an object detection algorithm is trained in the simulated- and generated environment. The two models are then tested in a real-world environment. The acquisition of the different images are described in Section 4.1. The generated images might look photo-realistic to the naked eye, but performing object detection provides a more accurate indication of the difference between the simulated- and generated images. The method used for object detection is the state-of-the-art Mask R-CNN described in Section 3. The performance of the models trained in the simulated- and generated environment is measured by a segmentation mask as well as percent wise object detection predictions on the real-world vessel.
This section will start by describing the architecture of the Mask R-CNN in connection with this thesis, followed by a description of the dataset needed. The last subsection will present the implementation of the Mask R-CNN.

4.3.1 Architecture

The architecture of the Mask R-CNN part of this thesis can be visualized by dividing it into two figures. The first figure, 4.11, illustrates a model trained in the generated environment and tested in the real-world environment. The simulator image, Actual A, is taken as input in the pre-trained Generator A2B, where its training is visualized in Figure 4.9, which outputs the Generated B image. The generated image, together with its bounding box label extracted for the Actual A image in the simulator described in Section 4.1, is first taken as input in the Deep CNN, which extracts feature maps from the image. The Deep CNN block represents the *backbone* in the network and cosists of FPN and a ResNet101. The RPN extracts region proposals. The feature map and RoI are processed by the RoIAlign and then taken as input in the Region CNN features box, where convolutional layers further process it. The output is the Box offset regressor, Softmax classifier, and the Mask FCN predictor. The model is learning by comparing the images corresponding bounding box labels with the predictions done by the network. This process is repeated for many Actual A images. When the model is done training, it takes a real-world environment image as input, Actual B, and makes a prediction on it, which results in the Predicted

B image. The resulting image, Predicted B, shows a red segmentation mask fitting the vessel, as well as a percent wise prediction in the upper left corner of the segmentation mask which is quite small and might be difficult to see in the figure.

If the Generator A2B manages to generate the Generated B in a way that it looks like it comes from the real-world data distribution, the Mask R-CNN model pre-trained on generated images will perform well when tested in the real-world environment.

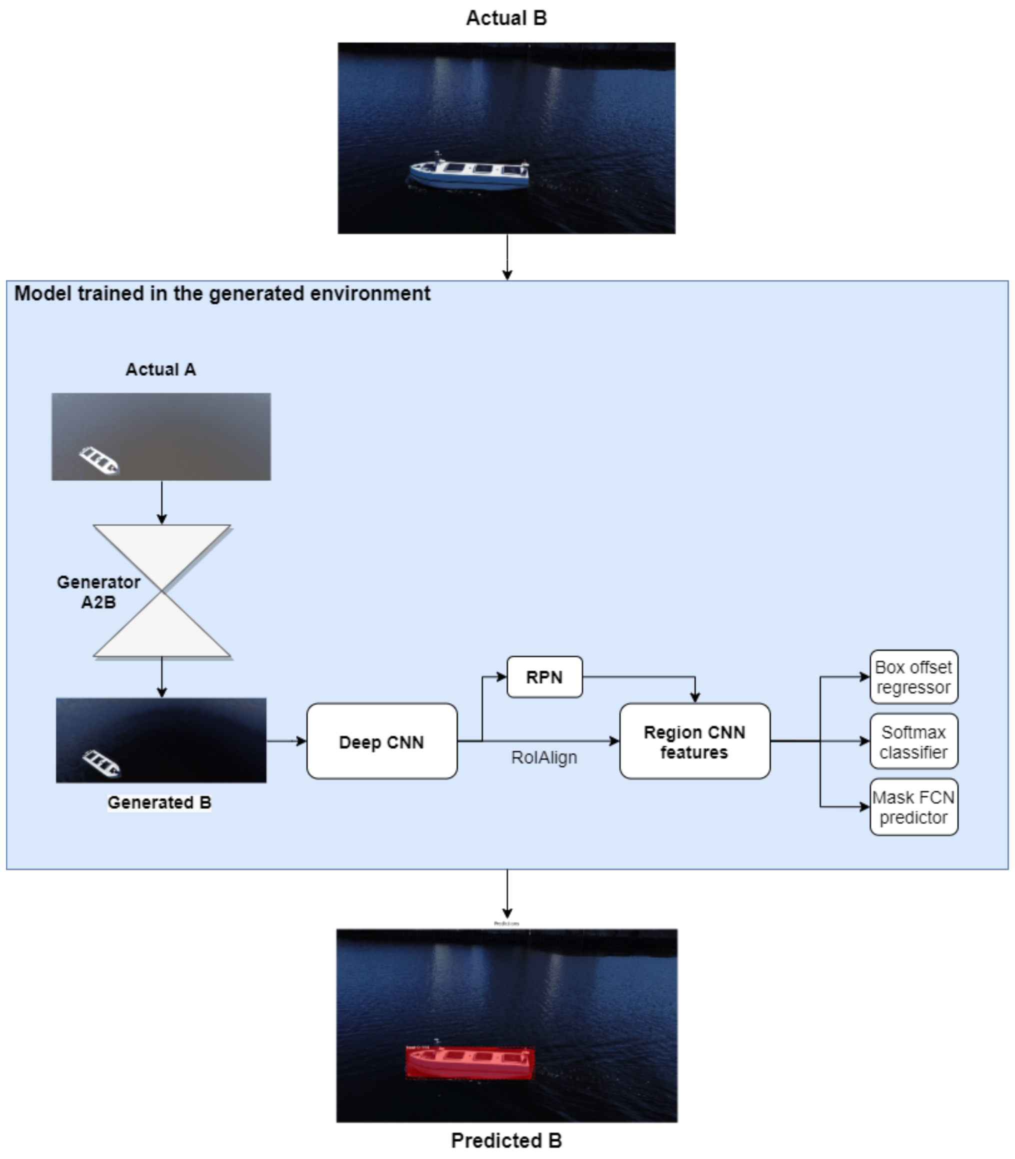


Figure 4.11: Mask R-CNN trained in the generated environment tested in the real environment.

Figure 4.12 shows the model trained in the simulated environment followed by testing in the real-world environment. The process is the same as described above for Figure 4.11, expcept the Actual A is not processed by the generator, but rather taken directly as input by the Deep CNN together with its corresponding bounding box label.

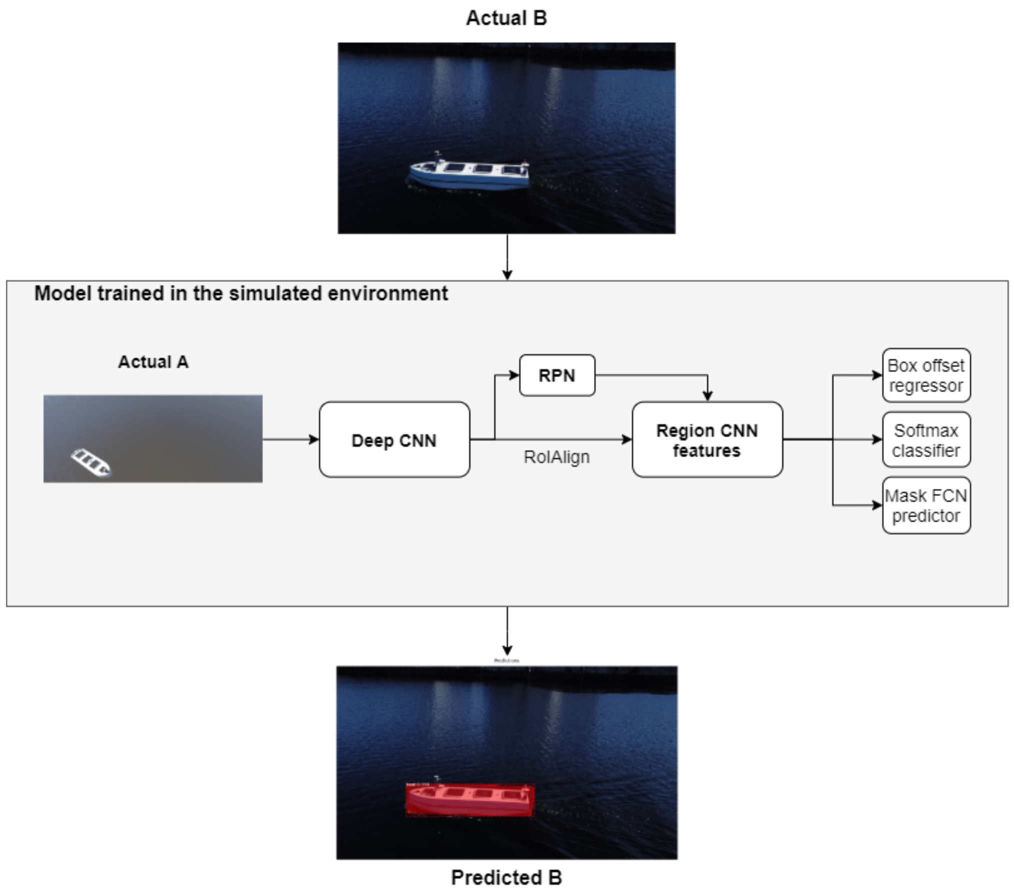


Figure 4.12: Mask R-CNN trained in the simulated environment tested in the real environment.

The overall goal is that the Generator A2B manages to generate Generated B images that are preserving the annotation of the boat in the image, that is, it is not changing the position of the boat, and generating images looking like it comes from the Actual B data distribution. This way, the model trained in the generated environment could potentially perform better when tested in the real-world environment than the model trained in the simulated environment. If this is achieved, the overall goal of this master thesis is reached.
The dataset for the training of Mask R-CNN is described in the following section.

4.3.2 Dataset

The dataset needed for Mask R-CNN is first of all simulated images with bounding box labels. Section 4.1 describes how the simulated images and the corresponding labels are acquired. The simulated images, together with the bounding-box labels, are utilized to get a Mask R-CNN model trained in a simulated environment.

The generated environment for training Mask R-CNN is rendered by giving the simulator images as input in the pre-trained generator described above in Section 4.2. The output images, which we call the generated images, together with their corresponding labels which are the ones extracted for the simulator images, is utilized for training a Mask R-CNN model in a generated environment.

The real-world images acquired from the drone footage of the ReVolt vessel, also described in Section 4.1, is utilized to test the pre-trained models. This test will reveal how well the models generalize to the real-world environment.

4.3.3 Implementation

The implementation is done in Python utilizing Keras and TensorFlow, described in Section 2. It is based on [20, 3], and adapted and tuned to fit the dataset of this thesis. As for the cycleGAN, the object detection also runs in Google Colaboratory, described in Section 2, to speed up the training process by running on a virtual GPU. Mask R-CNN is designed for accuracy rather than memory efficiency; thus, it is not a light-weight model. To make sure the GPU does not run out of memory during training, it is recommended to have a GPU with 12GB or more [3]. As mentioned in Section 2, Google Colaboratory provides a GPU of 12 GB which is adequate for the algorithm. Running the code in Google Colaboratory, an environment is needed to be set up where different Python libraries with specific versions are installed and imported. Moreover, Google Colab only provides one GPU; thus, the *workers* in the model-file needs to be changed to 1. Otherwise, an error will occur, *Using a generator with 'use multiprocessing=True*, which prevents the code from running.

To import the dataset of images and labels for training, a class called *BoatDataset* is implemented. A train set and a test set is created utilizing this class. A class function loading the dataset is called for both instances, where the input is the path to the data folder as well as a boolean variable *isTrain* set to true and false for creating the train- and test set respectively. The loading process will split the data located in the specified folder into a train- and test set where the split is specified depending on the amount of data. The split is also experimentally changed to achieve the best possible model. The function checks if the isTrain is set to true or false and creates the set accordingly. The next step in the process of loading the data is to extract the bounding boxes and the resolution of the images. Masks are created based on the bounding boxes extracted and are associated with the image by an image id. This id will be given to the image itself and its belonging metadata like resolution, masks, along with others. The information of the images is saved in a data structure and can easily be accessed by the image id. This class will also add the class "boat" for the predictions and associate the created masks with their corresponding class.

A configuration class is also created to enable for easily changing the configuration parameters to get the desired trained model. Some of these parameters will not be changed while training the models, and they are mentioned in this paragraph. The number of GPUs utilized and the number of images per GPU are both set to 1. The amount of classes is set to $1 + 1$, which represents the background- and the boat class. Setting a high learning rate speeds up the network, but if it is too high the loss could "explode," thus the learning rate is set to 0.006. The learning momentum is set to 0.9, and the weight decay regularization is 0.0001. The number of RoI is set to 200, and the maximum number of instances is kept at 100, although this could be lower to save memory.

The implemented model can be loaded with the pre-trained weights from training on the COCO dataset [52], pre-trained weights from training on the ImageNet [53], or without any pre-trained weights. The code has to be changed accordingly to the desired initial weights. Ideally, it was wanted to train the Mask R-CNN from scratch, but due to constraints in time and resources, this was not possible. The other two options have, therefore, been used for weights initialization. Using pre-trained weights means that the early layers are already trained to extract low-level features. Which layers to train can also be chosen, but for this thesis, all the layers are trained.

Training only the heads takes the least memory, and training all layers takes the most memory.

In the below section, LIME is utilized to assess credibility in the Mask R-CNN models' predictions by checking which features the predictions were based on.

4.4 LIME

For the overall system to achieve reliability, LIME is used to validate the predictions performed by Mask R-CNN. The theory of LIME is described in Section 3.5.1. This section presents the architecture of LIME, the needed data, and its implementation.

4.4.1 Architecture

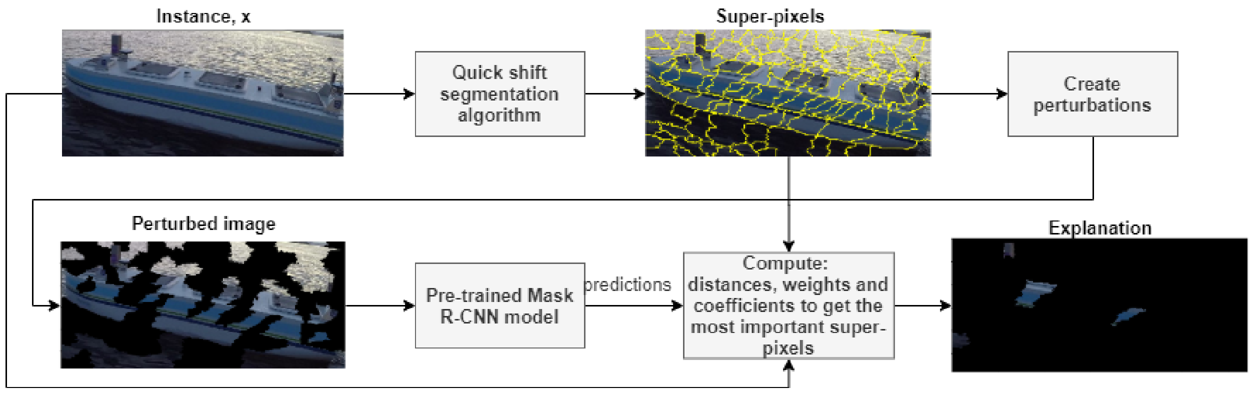

Instance, x
Quick shift segmentation algorithm
Super-pixels
Create perturbations
Perturbed image
Pre-trained Mask R-CNN model
predictions
Compute: distances, weights and coefficients to get the most important super-pixels
Explanation

Figure 4.13: An overview of the LIME architecture.

Figure 4.13 illustrates the architecture of the LIME algorithm in connection with the overall system. An instance, x, is first given to a quick shift algorithm calculating the super-pixels for this image. The super-pixels are needed to create a specified amount of perturbed images, that are input to the pre-trained Mask R-CNN model, which makes predictions on each one of them. The predictions, along with the super-pixels and x, are utilized to compute the distances between the instance and the perturbed images and the weights. Coefficients are retrieved from fitting a linear regression model to the computed data. The essential super-pixels for the prediction of the "boat" class are calculated and turned on in the resulting explanation image in Figure 4.13 while the other super-pixels are turned off, that is, black.

4.4.2 Implementation

The implementation of LIME is based on [22] and [4], and adapted to integrate well with the Mask R-CNN classifier. The images are loaded and processed with Python's Scikit-image[1] library. For the images to work with the Mask R-CNN classifier and the Scikit-image library, the pixels need to be represented in unsigned 8-bit integer.

The first step in the implementation is to compute the instance, x's, super-pixels using Scikit-image's quick shift segmentation algorithm, which segments the image using quick shift clustering in Color-(x,y) space. The algorithm uses a Gaussian kernel with a width of four to smooth the sample density, where larger kernel size corresponds to fewer super-pixels. The maximum distance, which represents the cut-off point for data distances, is set to 200. Higher distance also means fewer super-pixels. The algorithm's ratio, a number between zero and one, balances color-space- and image-space proximity. For this implementation, the ratio is set to 0.2.

After retrieving the super-pixels, 150 random perturbations are created, which are the original image with super-pixels randomly turned on and off. The perturbations are randomly drawn from a binomial distribution of the super-pixels using Python's NumPy [2] library.

Next, a prediction for each perturbed image is computed with the pre-trained Mask R-CNN model. The LIME algorithm can only explain one class at a time; thus, a for-loop goes through the predictions for each perturbed image and adds the prediction score for the class "boat" in an array. Mask R-CNN can predict more than one boat in an image, but only the predicted boat with the highest accuracy is added to the predictions array for each perturbed image. This step is the most computationally expensive one and depends on the classifier and the number of perturbed images.

The next step in the LIME implementation is to compute the distances between the original image and each of the perturbed images. The distance is computed using the cosine distance, which measures the cosine of the angle between two vectors and determines whether or not they are pointing in the same direction. Further, the distances are mapped using a kernel function to a value between zero and one, which corresponds to the weights. The following function

[1]Scikit-image is a collection of algorithms for image processing.

[2]NumPy adds support for large, arrays and matrices, along with mathematical operations for the arrays.

is utilized for calculating the weights,

$$\pi_x = \sqrt{\exp\left(-D_L^2/\sigma^2\right)}, \tag{4.6}$$

where the distance is represented by D_L, the kernel width by $\sigma = 0.25$, and the weight is π_x. The perturbations, predictions and weights calculated above, is utilized to fit a weighted linear regression model. Each coefficient in this linear model corresponds to a super-pixel in the perturbated image, and represents how important this super-pixel is for the prediction of the "boat" class. The coefficients are then sorted to find the super-pixels with highest coefficients. Finally, the resulting explanation is presented to the user by turning on the most important super-pixels in the prediction of the "boat" class in the original image, and turn all the others off.

Chapter 5

Results

As mentioned at the beginning of the thesis, the main objective of this work is to evaluate the possibility of utilizing GAN to improve the data quality acquired by a marine simulator and make it more realistic to achieve better detection algorithms in the marine environment. The below-presented results will thus start by showing and explaining the different cycleGAN models obtained during training and their corresponding generated images. The title of this thesis is *Synthesizing Photo-Realistic images from a Marine Simulator via Generative Adversarial Networks.* Thus, evaluating the achieved degree of photo-realism of the generated images is performed by a 'Visual Turing Test' in Section 5.2. Fifty participants have performed the test of classifying nine generated- and real-world images as either real or fake. However, a more accurate measurement can be achieved by utilizing Mask R-CNN as done in Section 5.3. Finally, the quantitative results achieved by Mask R-CNN are validated with the LIME algorithm, and its results are displayed in Section 5.4.

5.1 CycleGAN

Training cycleGAN is time-consuming, and achieving a good model is complex. Loss curves do not reveal much information in training GANs, and cycleGAN is not an exception [9, 2]. The generator is only graded against the current discriminator, and the discriminator is constantly improving. Thus, the loss function evaluated at different points in the training process cannot be compared. Periodically generating images and analyzing them is, therefore necessary, and has given rise to the models presented in this section.

The cycleGAN results are achieved by testing the pre-trained Generator A2B networks on a test directory of images different from the images used for training. In the testing phase, the generator maps the rendered simulator images to the domain of the real-world images. A perfect generator maps a rendered simulator image to an image that looks like it comes from the same data distribution as the frames from the drone video of the ReVolt vessel. In the process of getting satisfactory results, several different datasets are tested over a different amount of epochs resulting in different models. However, only the main ones are presented below. All the below datasets include the same real-world images, shown in Figure 5.3. An overall observation from the training process is when training over more than 50 epochs, new artifacts are introduced in the generated images. Thus, the presented models are trained for less than 50 epochs.

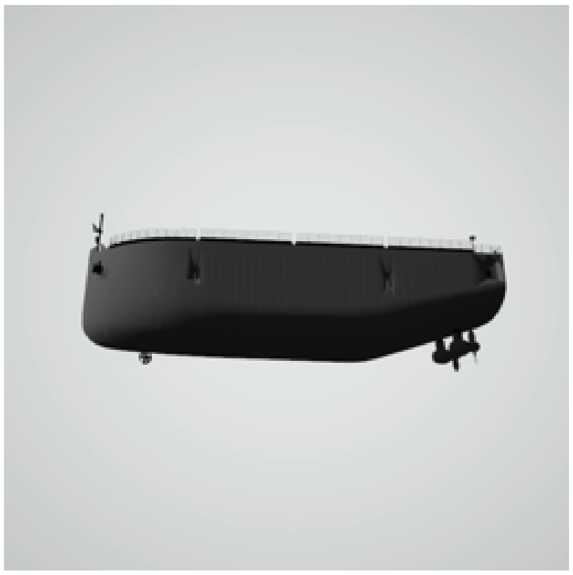
Below four different models are first presented, followed by a discussion at the end.

5.1.1 Model 1

Training over 30 epochs on Dataset1 shown below, resulted in Model 1. The model was first trained for five epochs, then stopped, and continued training for another 25 epochs. The test results generated from 5.1a can be seen in Figure 5.1b.

Dataset 1

Dataset 1 contains OBJ file images similar to the one in Figure 5.2, for both trainA and testA; however, these two folders contain distinctive images. The real-world images in trainB and testB for Dataset 1 comes from the same drone footage as the images in 5.3. During training, trainA contains 904 OBJ file images, and trainB contains 300 real-world images.


(a) Snapshot from OBJ file of the vessel.

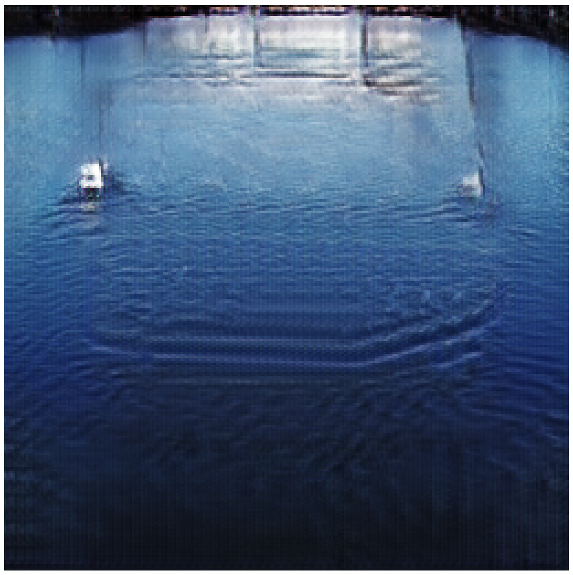
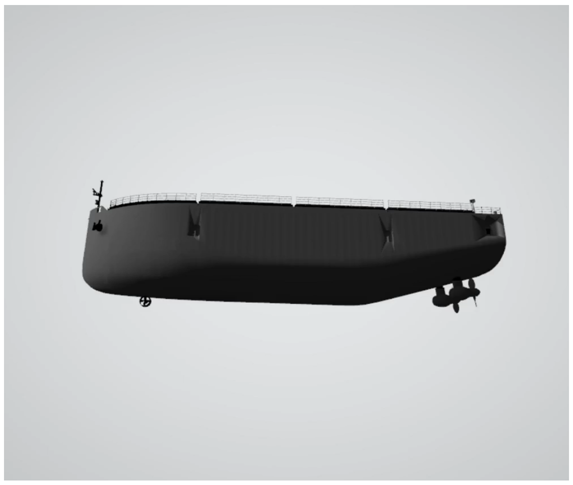
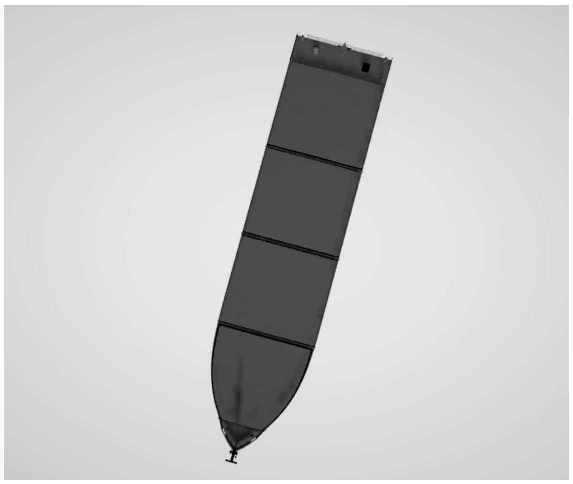
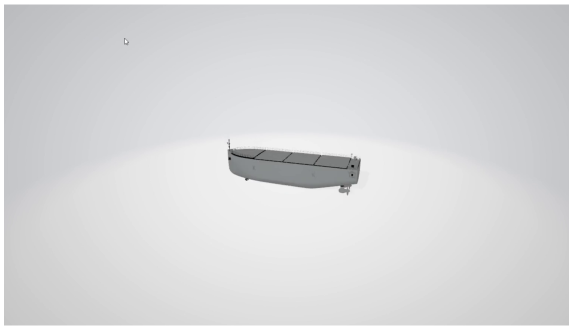

(b) Generated image based on 5.1a

Figure 5.1: Model 1's test results from training over 30 epochs.


(a)


(b)


(c)

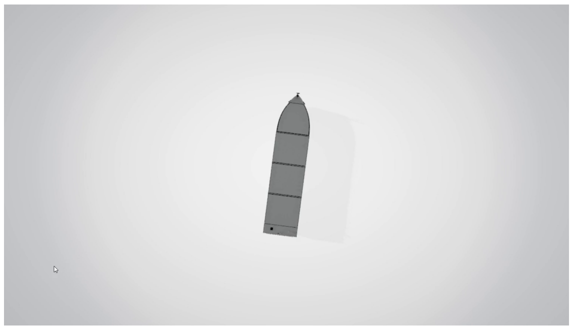
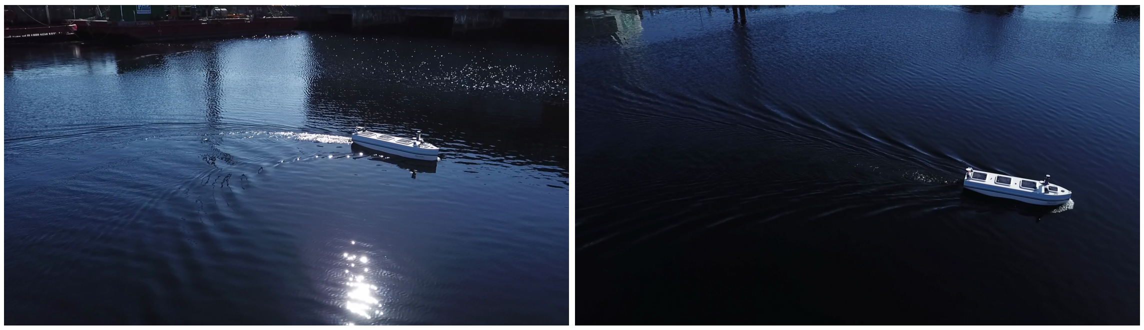

(d)

Figure 5.2: The OBJ file images for trainA and testA in Dataset 1.

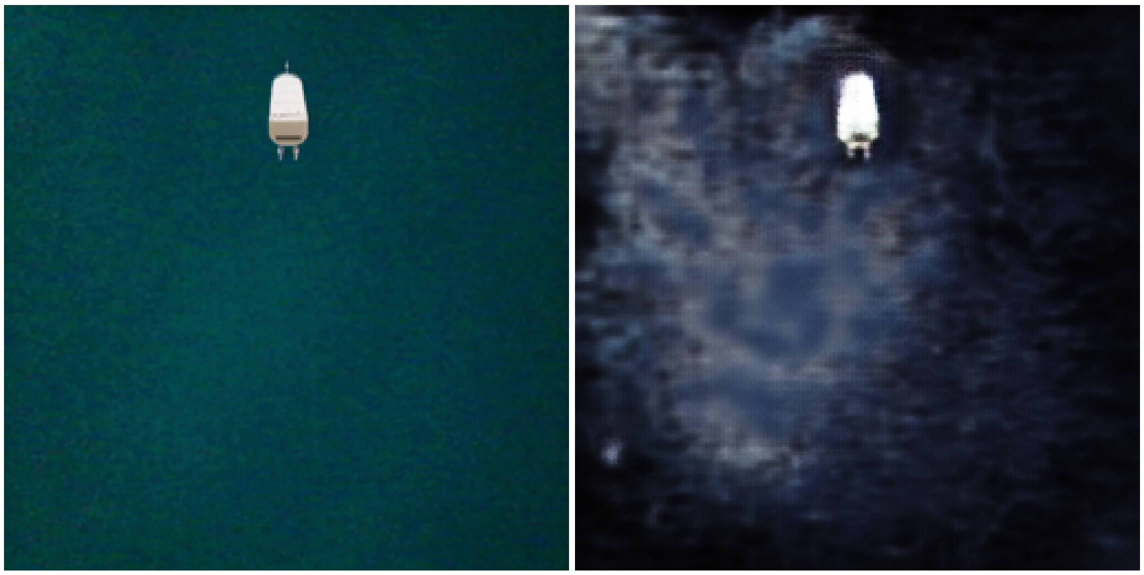


(a) (b)

Figure 5.3: The real-world images for all four datasets.

5.1.2 Model 2

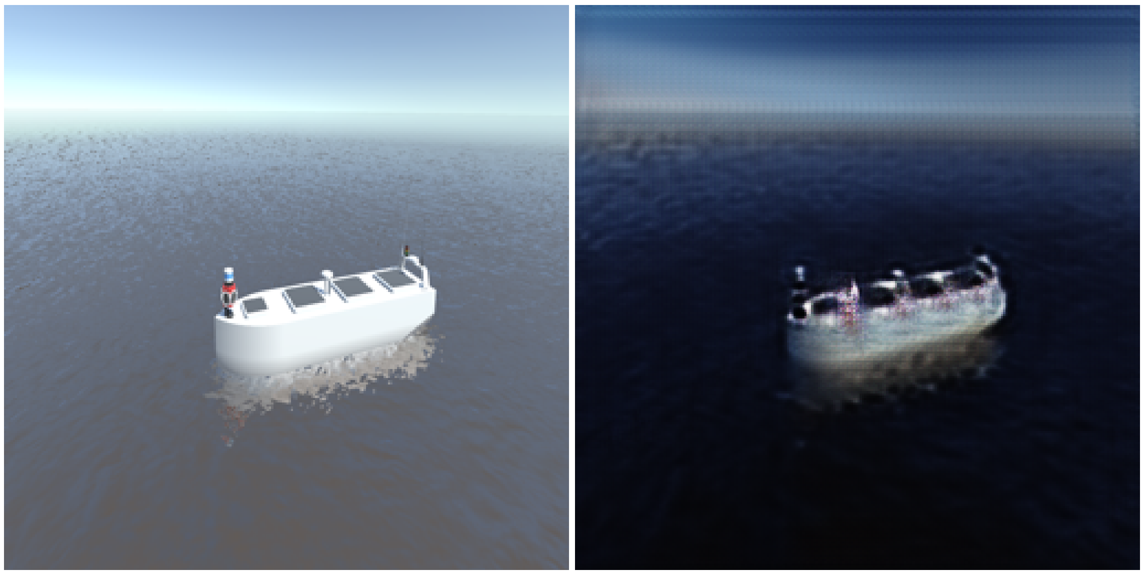
Model 2 was achieved by training for 38 epochs on Dataset 2, described below. The model was first trained for 24 epochs, then stopped, and trained for another 14 epochs. The test results generated after 38 epochs were run, based on the simulator image shown in 5.4a, is illustrated in 5.4b.



(a) Snapshot from OBJ file of the vessel. (b) Generated image based on 5.4a.

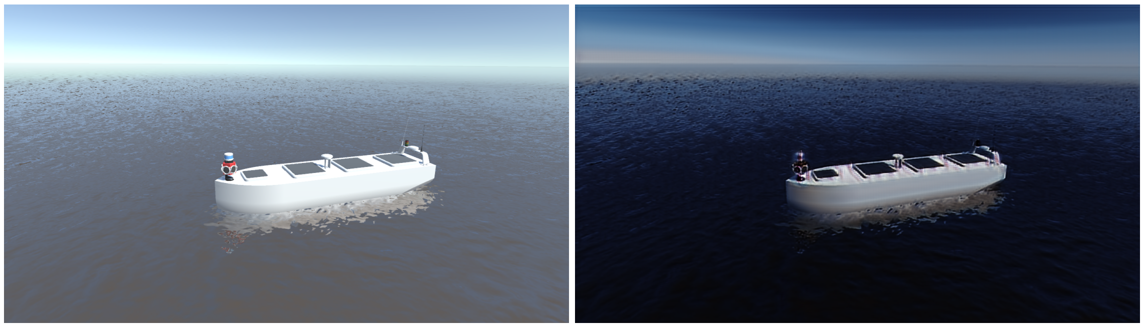
Figure 5.4: Model 2's test results from training for 38 epochs on Dataset 2.

Model 2 was also tested on a Unity simulator image, shown in Figure 5.5a and the generated result is displayed in 5.5b.



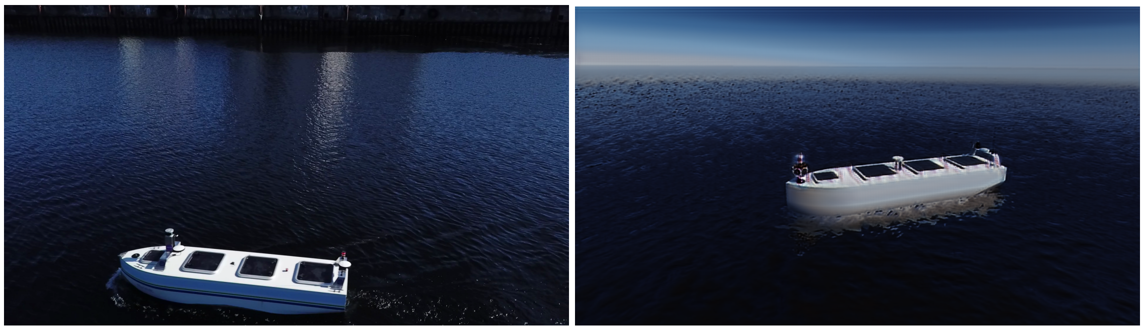
(a) Snapshot from the Unity-file (b) Generated image based 5.5a.

Figure 5.5: Model 2's result from testing on a Unity simulator image.



(a) Snapshot from the Unity-file (b) Generated image based on 5.6a

Figure 5.6: Shows the test image and generated result from Model 2 without any preprocessing

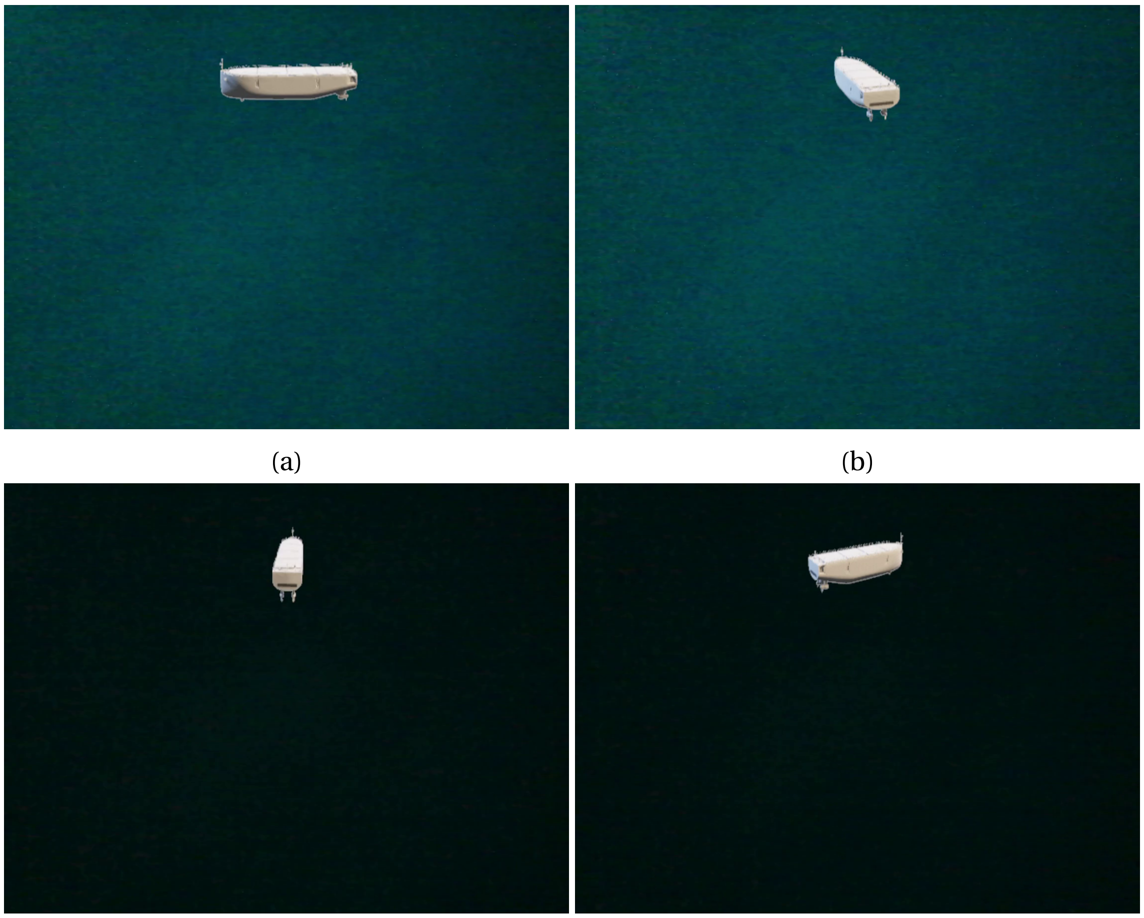


(a) Real image of the ReVolt vessel (b) Generated image based on 5.6a

Figure 5.7: Comparing a real and a generate image of the ReVolt vessel

Dataset 2

Dataset 2 contains images from the edited OBJ file similar to the one in Figure 5.8, for both trainA and testA; however, these two folders contain distinctive images. The real-world images in trainB and testB for Dataset 2 comes from the same drone footage as the images in 5.3. During training, trainA contains 845 edited OBJ file images, and trainB contains 300 real-world images.



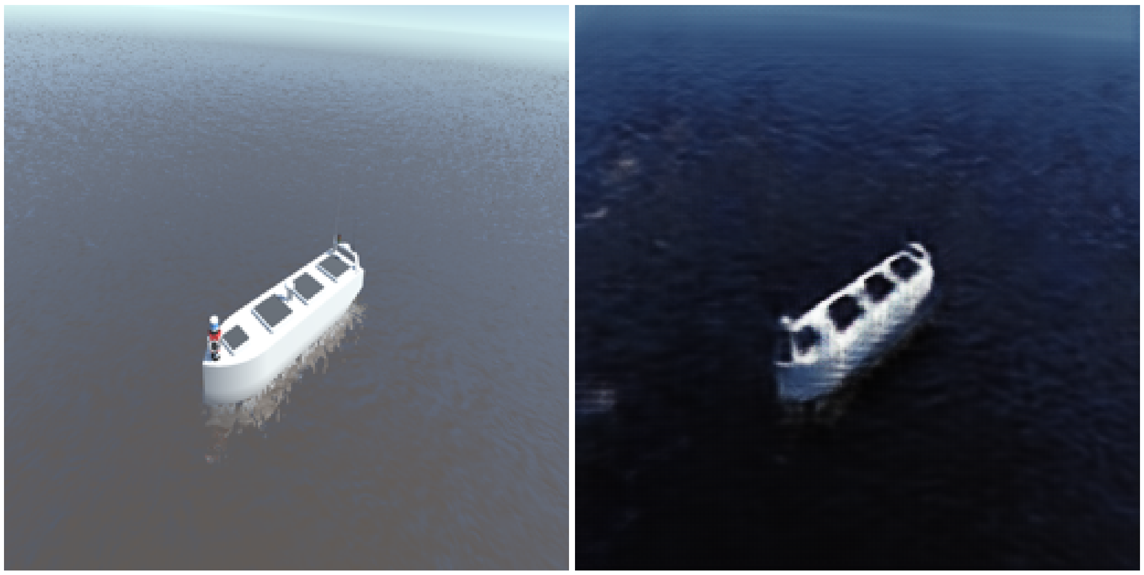
(a) (b)

(c) (d)

Figure 5.8: The edited OBJ file images for trainA and testA in Dataset 2.

5.1.3 Model 3

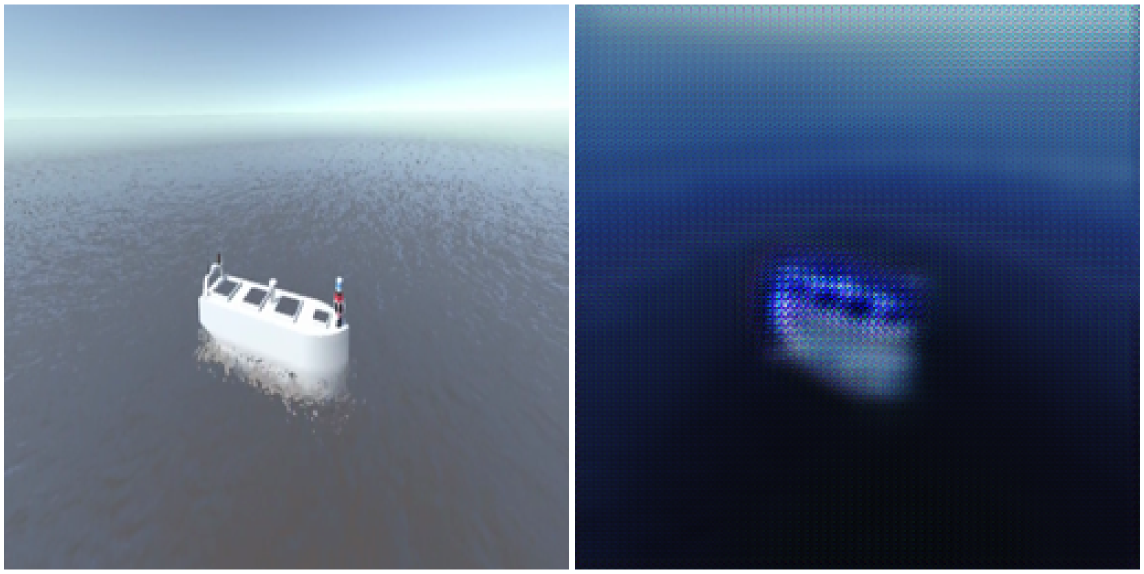
Training the cycleGAN for 25 epochs straight, referred to as Model 3, results in the generated image represented in 5.9b when testing on a simulator snapshot, shown in Figure 5.9a.



(a) Snapshot from the Unity simulator. (b) Generated image based on 5.9a

Figure 5.9: Model 3's test results from training for 25 epochs.

A generated test image and its corresponding simulator image saved during training after epoch two is displayed in Figure 5.10.

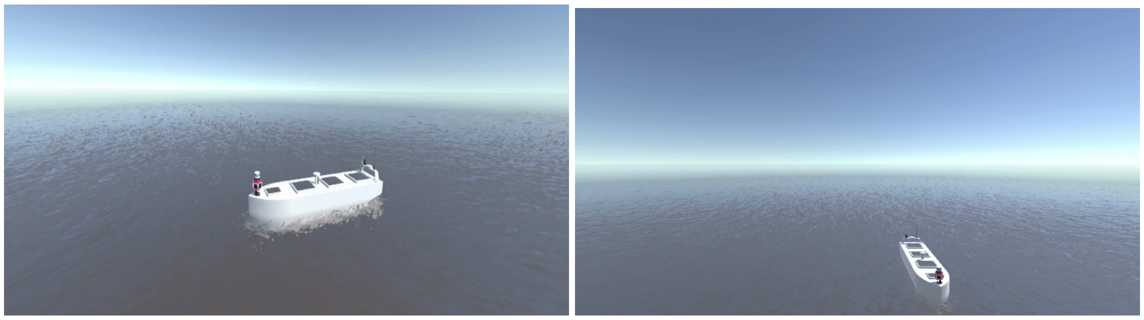


(a) Snapshot from the Unity simulator. (b) Generated image based on 5.10a

Figure 5.10: Model 3's test results saved during training after two epochs.

Dataset 3

Dataset 3 consists of Unity simulator images similar to the one in Figure 5.11, for both trainA and testA; however, these two folders contain distinctive simulator images. The real-world images in trainB and testB for Dataset 4 comes from the same drone footage as represented in Figure 5.3. During training, trainA contains 500 simulator images, and trainB contains 300 real-world images.

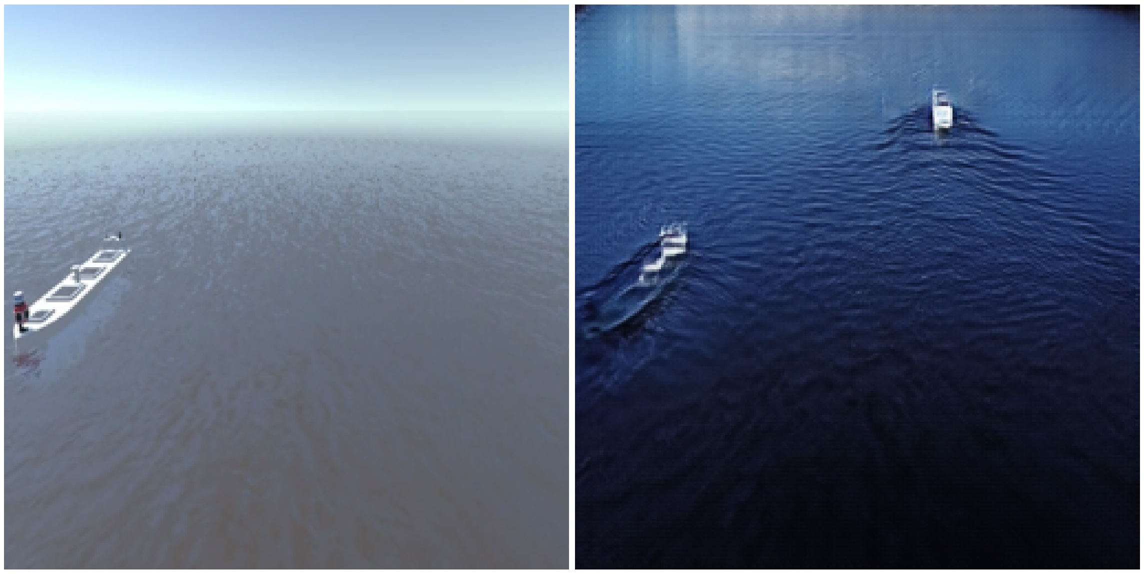


(a) (b)

Figure 5.11: The simulator images for trainA and testA in Dataset 3.

5.1.4 Model 4

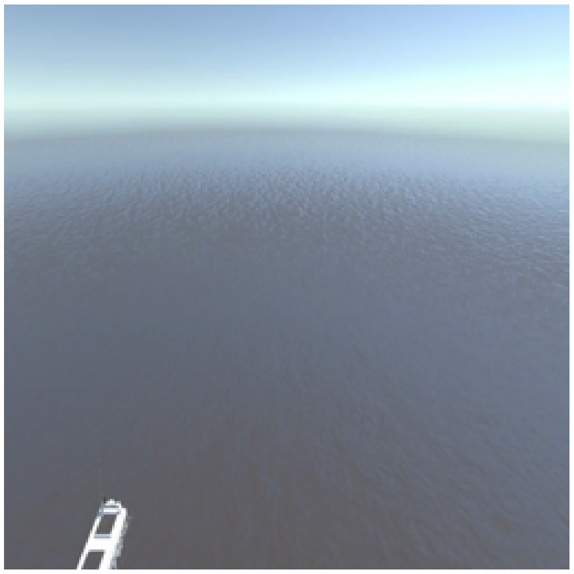
Model 4 was achieved by training for 39 epochs straight, on Dataset 4, described below. The test results generated during training after 27 of the 39 epochs were run, based on the simulator image shown in 5.12a, is shown in 5.12b.



(a) Snapshot from the Unity simulator. (b) Generated image based on 5.12a

Figure 5.12: Model 4's test results from training over 39 epochs on Dataset 4.

A generated test image and its corresponding simulator image saved during training after epoch one is displayed in Figure 5.13. The test results generated during training after 29 of the 39 epochs were run, based on the simulator image shown in 5.14a, is shown in 5.14b.
The test results generated after 39 epochs were run, based on the simulator image shown in 5.15a, is shown in 5.15b.


(a) Snapshot from the Unity simulator.

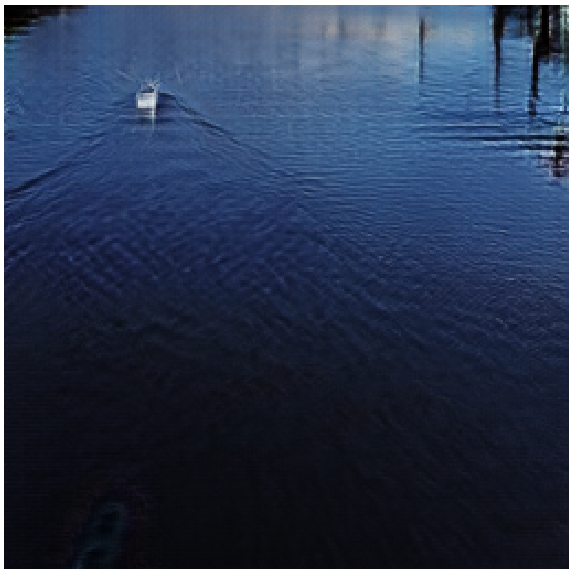
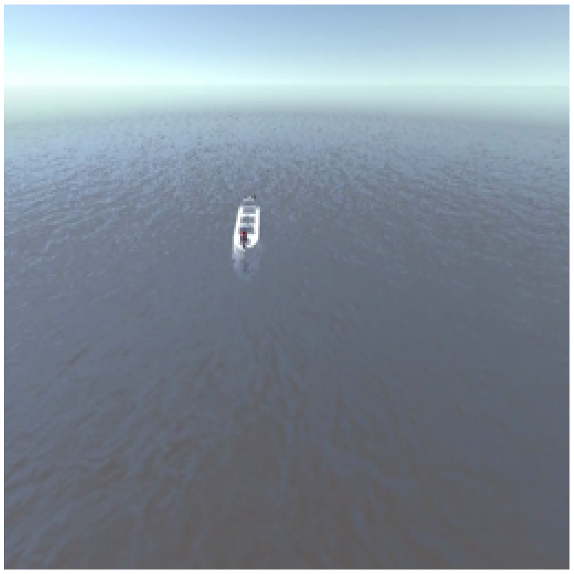

(b) Generated image based on 5.13a

Figure 5.13: Model 4's test results saved during training after one epoch.


(a) Snapshot from the Unity simulator.

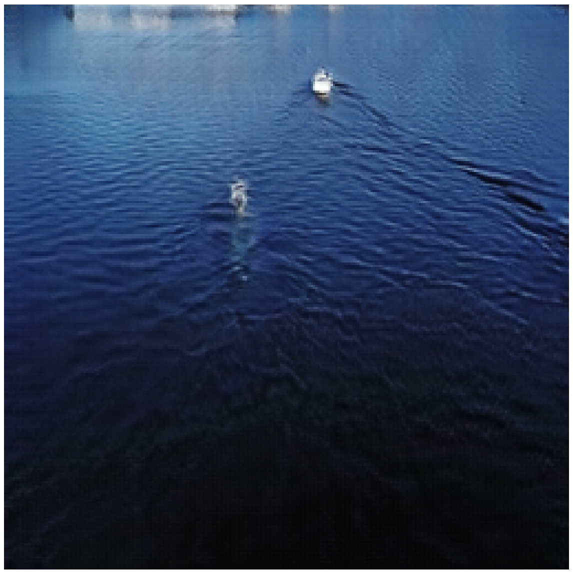
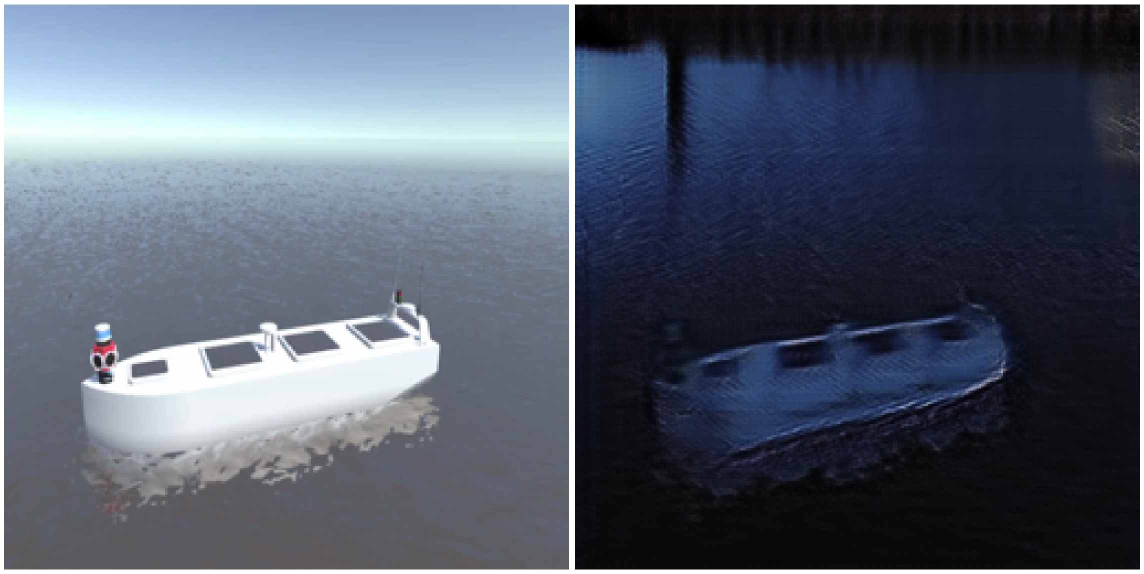

(b) Generated image based on 5.14a

Figure 5.14: Model 4's test results saved during training after 29 epochs.



(a) Snapshot from the Unity simulator. (b) Generated image based on 5.15a

Figure 5.15: Model 4's test results from training over 39 epochs on Dataset 4.

Dataset 4

Dataset 4 consists of Unity simulator images similar to the one in Figure 5.11, for both trainA and testA; however, these two folders contain distinctive simulator images. The real-world images in trainB and testB for Dataset 4 comes from the same drone footage as represented in Figure 5.3. During training, trainA contains 500 simulator images, and trainB contains 300 real-world images. Thus, Dataset 4 is similar to Dataset 3.

5.1.5 Discussion

After training on the same dataset in different amounts of epochs, it was noticed that training, for instance, 40 epochs all at once could generate less satisfactory results than training for 20 epochs, and then continue training for another 20 epochs. By looking into this issue, it seems like the network is sensitive to initialization. The training should be restarted if high contrast colors between the input- and generated image are observed in the images saved during training. Thus, some of the above models are trained for a certain number of epochs, stopped and continued training.

Model 1 generates the texture of the sea quite well; it looks real, indicated by Figure 5.1. It has even managed to get the reflection from the sky. On the other hand, the model has problems generating the actual vessel based on the snapshot from the OBJ file. Model 1 does not manage to differentiate between the ocean- and the white vessel's texture since it seems like the network has given the vessel in the generated image the texture of the ocean.

In an attempt to solve the problems with Model 1, the OBJ file was edited to be more representative of the real-world environment, as shown for Dataset 2. Model 2 manages to differentiate the textures between the vessel and the ocean, displayed in Figure 5.4. Another problem that arises is that the ocean looks less realistic in the generated image by Model 2 than it did for Model 1. However, by testing Model 2 on a Unity simulator image, the generated image improved significantly, as illustrated in Figure 5.5, although it is suffering from low resolution.
Different techniques were tested to get higher resolution on Model 2's generated images. One of them was to train the network over 80 epochs on the new simulator images, but this resulted in artifacts being introduced in the generated images, as mentioned in this section's introduc-

tion. More images were added to the training dataset to check if the network was overtrained, but the results did not improve. Finally, testing revealed that the low resolution was not due to the dataset or the number of epochs the network was trained, but because of the pre-processing described in Section 4.2. Information got lost because the output images were larger than the input images, which resulted in poor resolution. Testing Model 2 again on a Unity simulator image without doing any pre-processing resulted in the generated image with high resolution illustrated in Figure 5.6. The quality of the generated image has increased drastically.
Although Model 2 is trained on a dataset containing snapshots of the OBJ file with relatively poor quality and few details compared to the Unity simulator images, this model is the best one achieved.
These results could indicate that the details in the simulator images for training is not as important as it is for the images for testing. As mentioned in the introduction of this section, a perfect generator will map a rendered simulator image to an image that looks like it comes from the same data distribution as the frames from the drone footage of the ReVolt vessel. Figure 5.7 illustrates that Model 2's generated image looks more realistic compared to a real image of the ReVolt vessel than the Unity simulator image in Figure 5.6a.

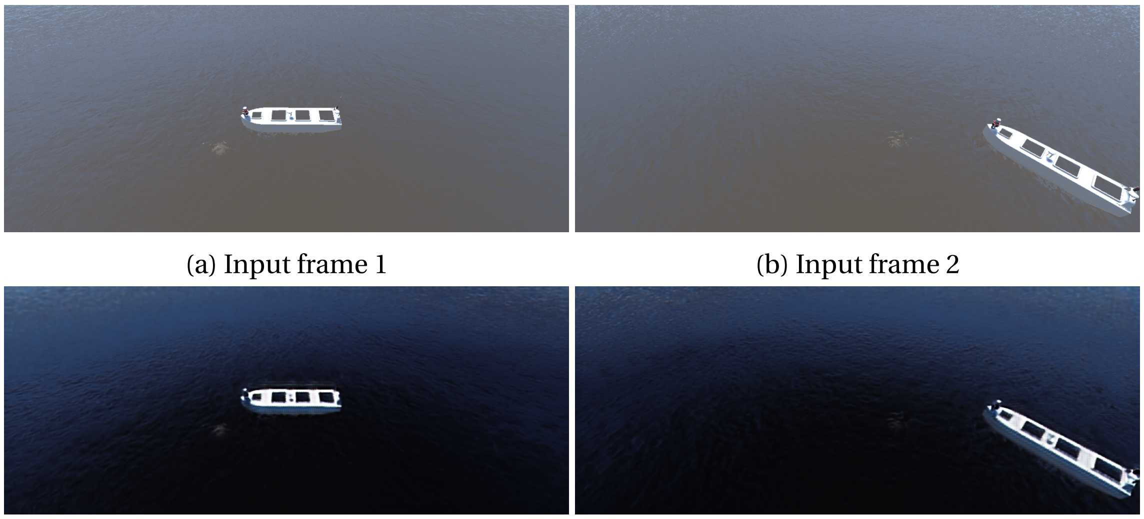
The OBJ file images in Dataset 2 have a slightly textured, one-colored background, a bit similar to the ocean. The vessel is white, but it is not submerged. In an attempt to achieve an even better model, cycleGAN was trained on Unity simulator images edited to have a plane, non-textured background in a dark blue color where the vessel is white and submerged, illustrated in Figure 4.4d. However, the resulting generated images are not nearly as good as the ones for Model 2. Thus, the texture in the OBJ file images for Dataset 2 might play a central role in achieving a good generator model.

Since training cycleGAN on the poor quality and few detailed OBJ file images resulted in the satisfying Model 2, training cycleGAN on the detailed, high-quality Unity simulator images, 5.11, should improve the model. However, different models, including Model 3 and 4, have been trained on datasets with the high-quality Unity simulator images, but none of the models became better than Model 2. Although Model 3 and Model 4's results gives an interesting insight into the process of training cycleGAN, and is thus still presented in this section.

Model 3 gives an example of the consequences of not initializing the weights when high contrast colors between the input- and generated image is observed. The generated images of the final trained Model 3 have poor resolution, which is not due to pre-processing. However, by looking at the generated image saved during training after epoch 2, displayed in Figure 5.13b, the vessel has a high contrast colour compared to the vessel in the input image, displayed in Figure 5.10a. Thus, the training should have been stopped and restarted already at epoch 2.

Model 4's generated images, already from epoch one, is not mapping the correct features, and by looking closely at Figure 5.13a, it can be seen that there is high contrast in the vessel's color between the input and output image. However, the generated images look like they come from the same data distribution as the real-world images from the drone footage. Model 4's generated images are also the ones getting the best score in the 'Visual Turing Test' in the section below. Nevertheless, these generated images will not contribute in achieving the main objective, that is, improve the data quality acquired by a simulator and make it more realistic to achieve better detection algorithms, since it is not mapping the correct features. The results from Model 4 indicate that photo-realism comes at the expense of mapping the correct features.

A video based on the simulator is generated by Model 2, and the result from two different frames in the video is shown in Figure 5.16. Subfigure 5.16a and 5.16b displays snapshots from the video of the simulated environment, and 5.16c and 5.16d shows snapshots from the resulting video of the generated environment.
As mentioned in Section 4.2.4, the implementation of this cycleGAN does not include the random noise vector z_{latent} as input in the generators. The randomness is introduced through the input image of the generator if it is sufficiently complex. The input simulator images in the above video are not adequately complex, which results in a deterministic generator. Although, when utilizing cycleGAN to create a real-time photo-realistic simulator, it is fine to have a deterministic generator, justified in Section 4.2.4.
Evaluating the achieved degree of photo-realism of the generated images is first performed by a 'Visual Turing Test' in the below section.


(a) Input frame 1 (b) Input frame 2

(c) Output frame 1 (d) Output frame 2

Figure 5.16: Model 2: simulator ⟶ real-world

5.2 'Visual Turing Test'

A 'Visual Turing Test' was created to evaluate the visual quality of the generated images quantitatively. The test consisted of nine questions where the participants were asked to classify images as real or fake. The images were of the generated- and real-world environment, and they were randomly ordered in the test.

	% selected as real	% selected as fake
Real-world image	48	52
Generated image	30	70

Table 5.1: The average results in % from a "Visual Turing test" user study with 50 participants for classifying generated vs real-world images.

The participants in the test consisted of 50 fellow students and professors at NTNU, family, and others. The generated images utilized and the percentage of participants classifying each image as real is displayed in Figure 5.17. The real-world images utilized and the percentage of participants classifying each image as fake is displayed in Figure 5.18. The average results in percentage from the test is shown in Table 5.1. This table indicates that 48 percent of the real-world images was in average classified as real, while 30 percent of the generated images were classified as real. Although the generated images in Figure 5.17a and 5.17b achieved better accuracy of being a real-world image, the generated images in Figure 5.17c and 5.17d are the ones mapping the correct features. The two last-mentioned subfigures are thus more qualified for improving the performance of detection algorithms even though they appear less realistic.

The generated image's quality can be measured by the naked eye as performed in this 'Visual Turing Test'. However, a more accurate measurement can be achieved by utilizing Mask R-CNN as done in the section below.

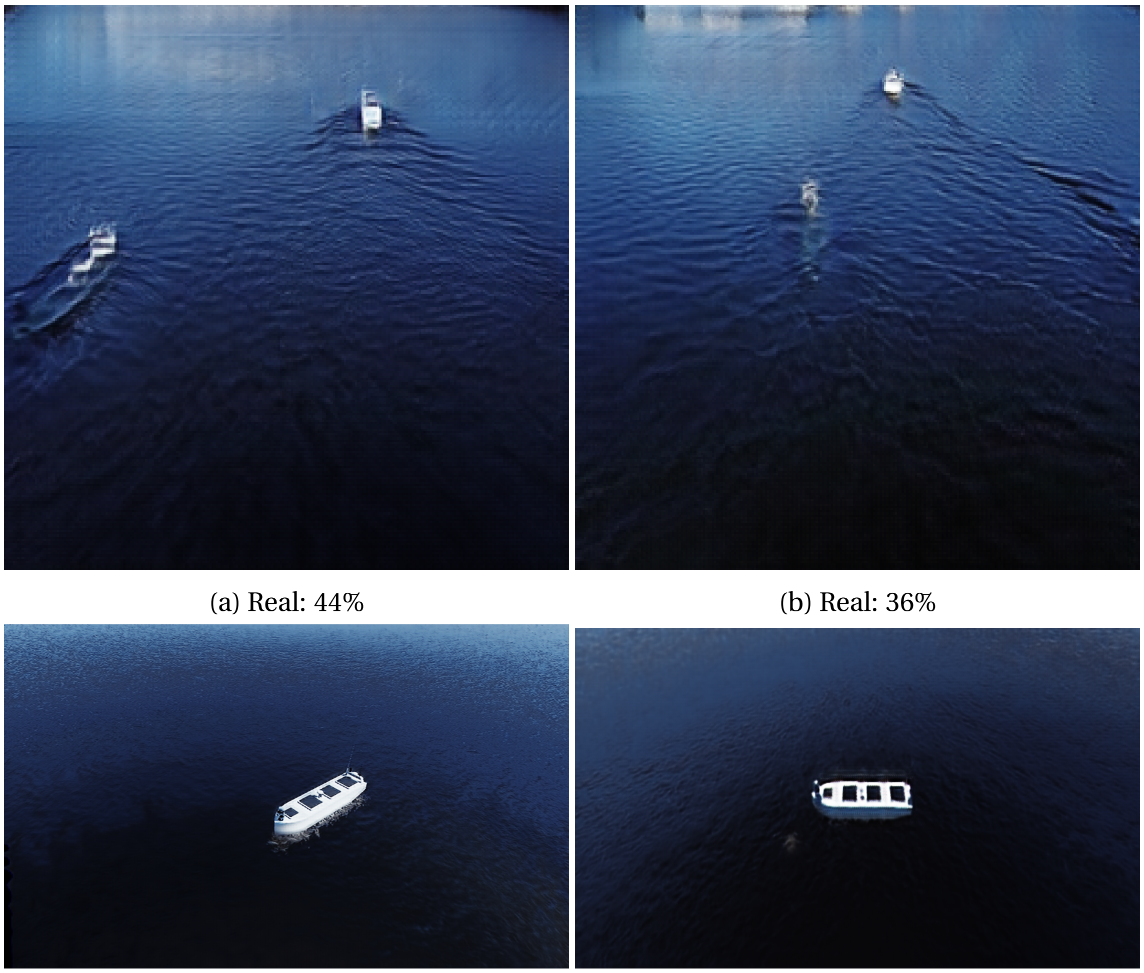
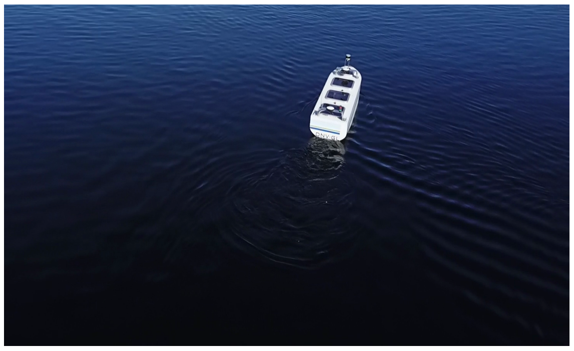
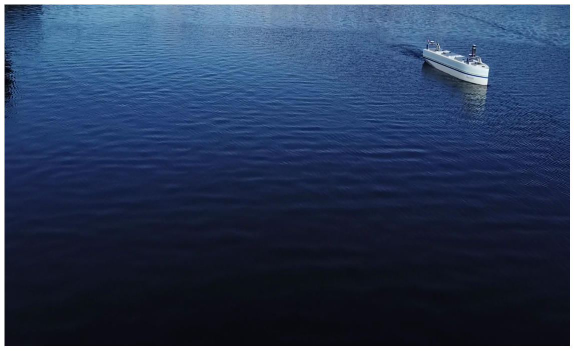

(a) Real: 44%
(b) Real: 36%
(c) Real: 26%
(d) Real: 14%

Figure 5.17: The generated images utilized in the 'Visual Turing Test' in Table 5.1 labelled with the percentage of 50 participants classifying the image as real. Model 4 generated (a) and (b), while Model 2 generated (c) and (d).

(a) Fake: 6%

(b) Fake: 56%


(c) Fake: 30%


(d) Fake: 18%

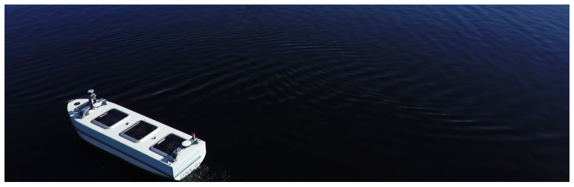

(e) Fake: 62%

Figure 5.18: The real-world images utilized in the 'Visual Turing Test' in Table 5.1 labelled with the percentage of 50 participants classifying the image as fake.

5.3 Mask R-CNN

Model 2 is utilized to generate images for the rest of this section.
Ideally, it was desirable to train the Mask R-CNN from scratch, but due to constraints in time and resources, this was not possible. COCO- and ImageNet weights have, therefore, been used for weights initialization. Using pre-trained weights means that the early layers are already trained to extract low-level features. Which layers to train can also be chosen, and for this thesis all the layers are trained. The trained model also depends on the number of epochs it is trained, how well the bounding box coordinates extracted from the simulator fits the boat, if the generator preserves the vessel's position, the input image resolution, and the number of input images. These factors are varied to get the desired models. The results from testing the models are dependent on the real-world input image. Testing on images with different factors of variation than the ones used for training the generator also affects how well the model performs.

Applying a Mask R-CNN model with the pre-trained COCO weights for object detection and instance segmentation, without training on the dataset described in Section 4.3.2, on a ReVolt vessel simulator image provides the result revealed in Figure 5.19.

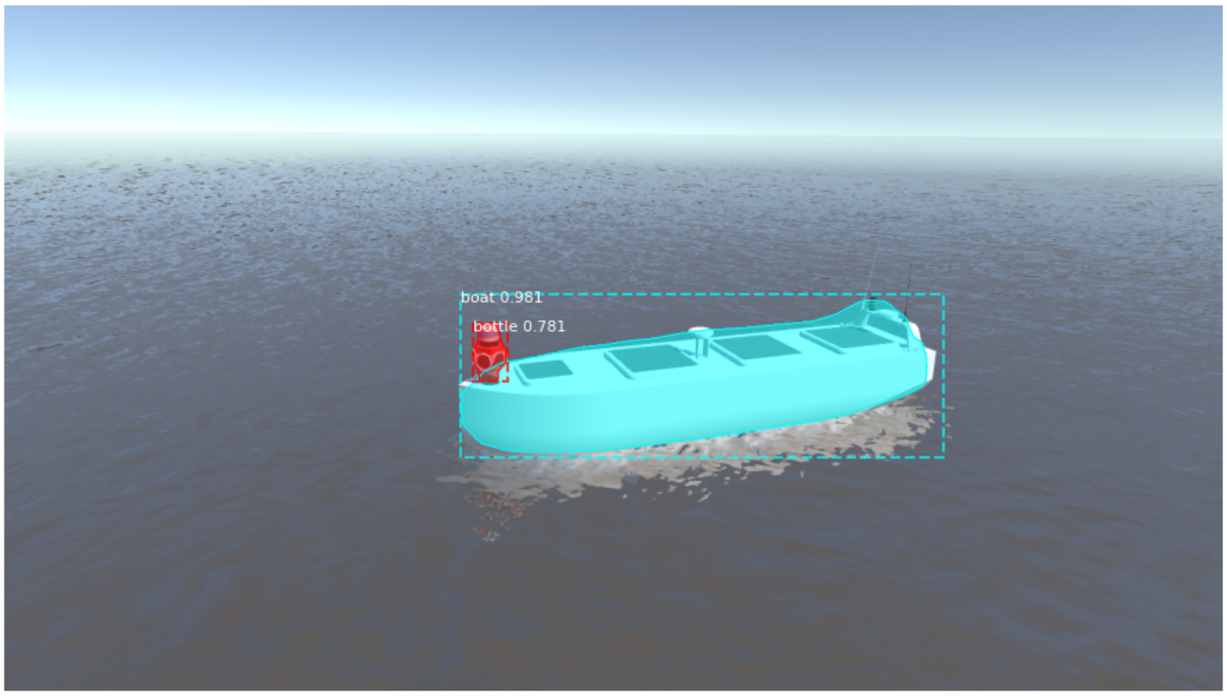


Figure 5.19: Mask R-CNN trained on the COCO-dataset applied on a simulator image.

Figure 5.19 shows that the Mask R-CNN model detected a boat in the simulator image with an accuracy of 0.981, which is quite high. When the detection for the simulator is already quite high, it is not easy to increase the accuracy. Figure 5.20 displays the results of applying the same Mask R-CNN model pre-trained on the COCO dataset on an image generated based on the simulator image in Figure 5.19. Figure 5.20 shows that the model detected a boat in the

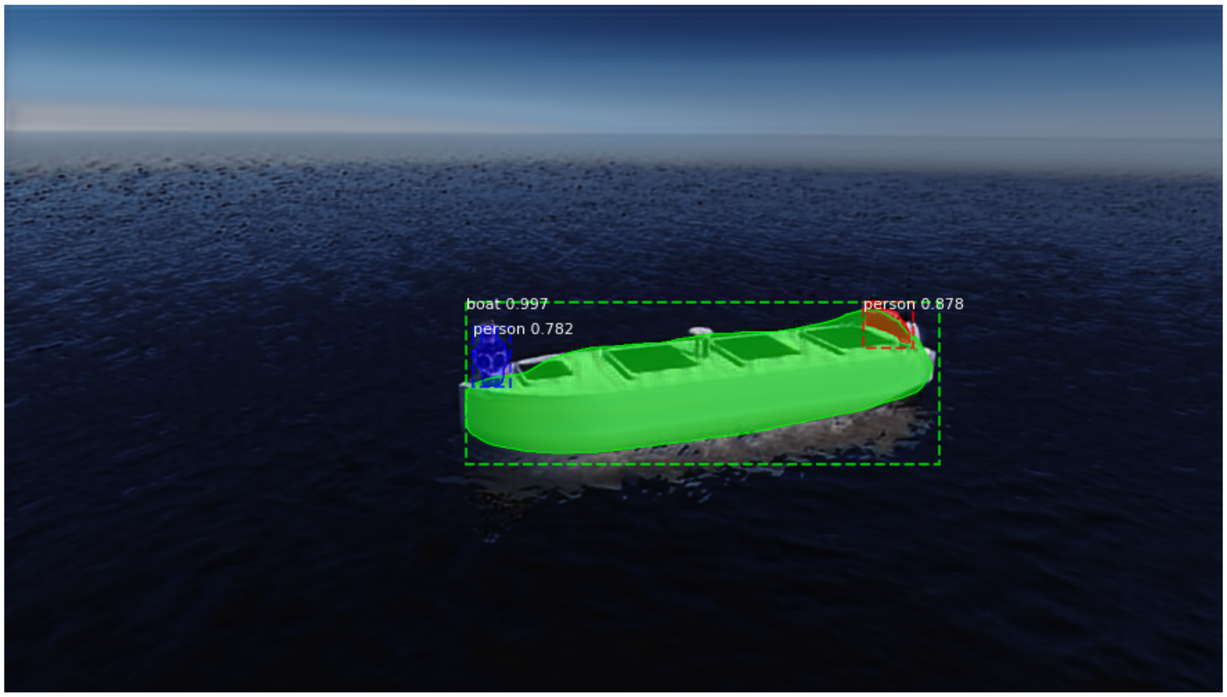


Figure 5.20: Mask R-CNN trained on the COCO-dataset applied on an image generated based on 5.19.

image with an accuracy of 0.997. Meaning that the Model 2 presented above has increased the detection accuracy by 1.16 percent! An increase of 1.16 percent when the accuracy is already 0.981 is satisfactory.

When applying the same pre-trained Mask R-CNN model on an actual image of the ReVolt vessel from the drone footage, the accuracy is 0.998 as illustrated in Figure 5.21. The object detection accuracy is 0.1 percent higher for the real image 5.21, than for the generated image 5.20. These two images are not representing the same scene and this comparison is therefore not valid. Another reason for this comparison not being valid is that the COCO dataset could be biased.

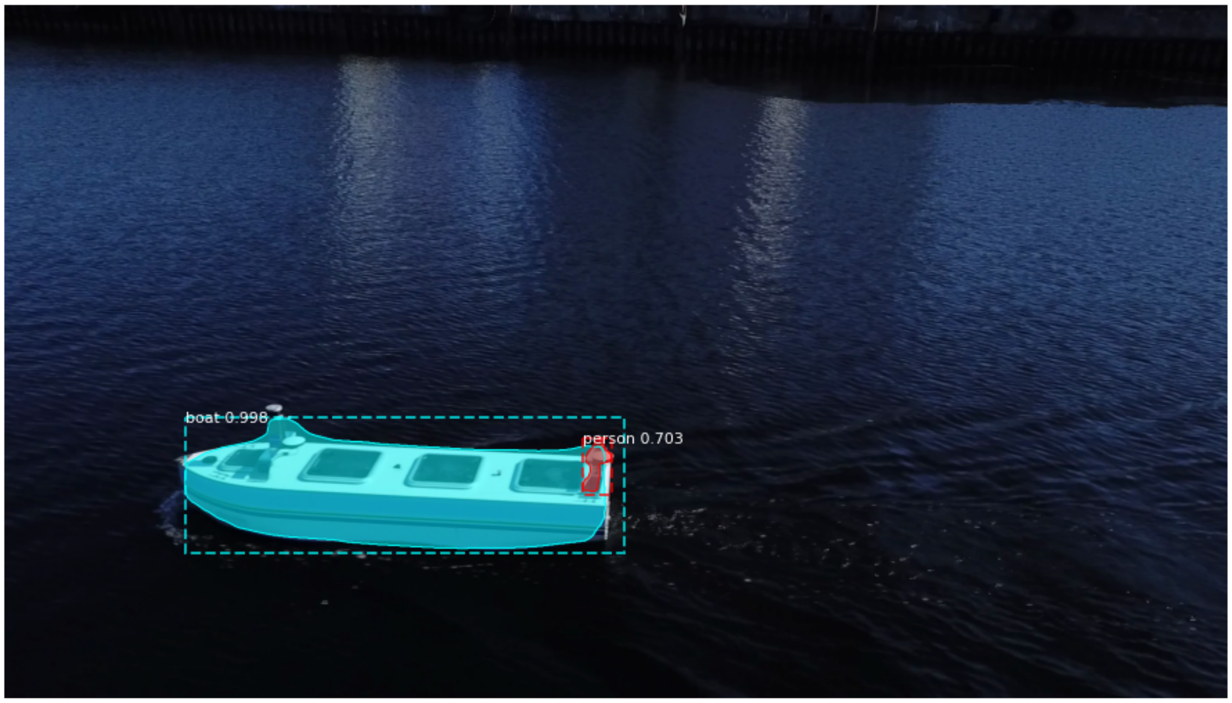


Figure 5.21: Mask R-CNN trained on the COCO-dataset applied on a real image of the ReVolt vessel.

Training a Mask R-CNN model on simulator images, and another on the generated images, followed by testing both the models on real-world images of the ReVolt vessel is a more accurate way of checking if the generated images have added value.

5.3.1 Trained in Simulated- and Generated Environment

The two Mask R-CNN models were achieved by training all the layers for ten epochs on 607 simulator- and generator images with a resolution of 1024×768. The simulator images are from the Unity simulator, similar to the ones in Dataset 3 and 4 shown in Figure 5.11, and the generated images are generated from these simulator images by Model 2 from above. The Mask R-CNN model's weights were initialized with the weights pre-trained on the Imagenet dataset. It is important to note that the real-world images used for testing the models are different from the images used to train the generative model. The drone footage that the Mask R-CNN test images are taken from is shot on different days with different weather conditions than the real-world images used for training Model 2, shown in Dataset 2 earlier in this section. The same two Mask R-CNN models are utilized for the rest of this section, and tested on three different real-world images, referred to as Image 1, Image 2, and Image 3, shown below.

Image 1

The results from testing the models on the first real-world image of the ReVolt vessel referred to as Image 1, are shown in Figure 5.22 and 5.23.

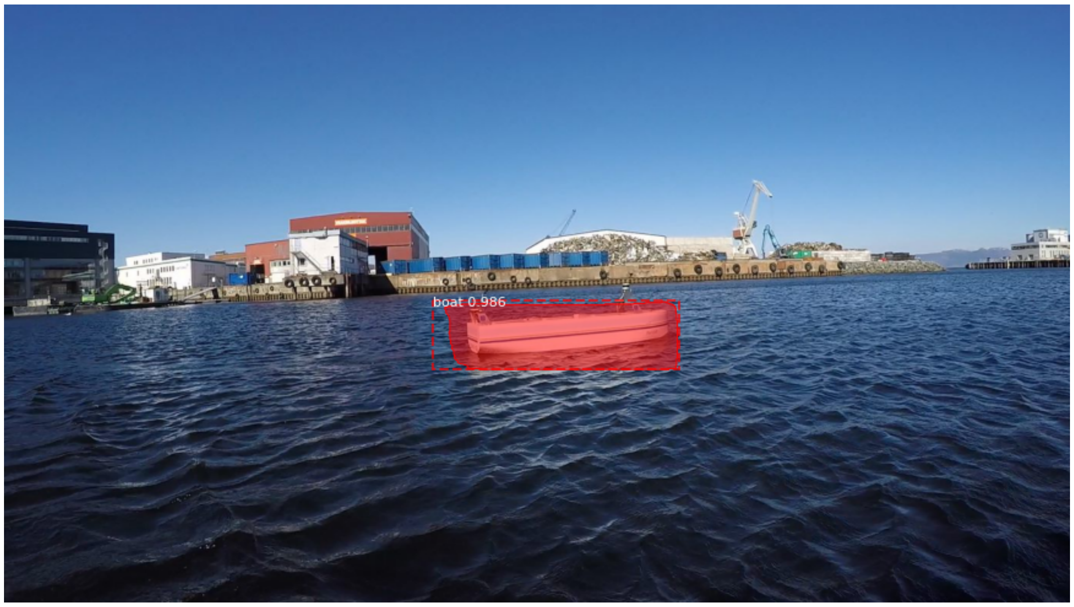


Figure 5.22: Mask R-CNN model trained in the simulated environment tested on Image 1 with the boat class prediction of 0.986.

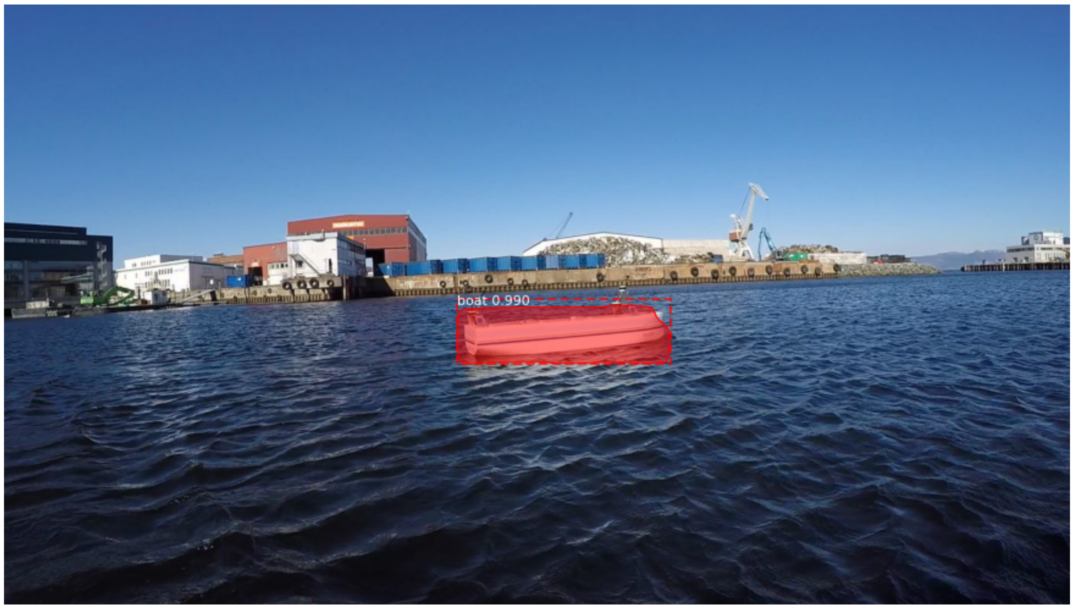


Figure 5.23: Mask R-CNN model trained in the generated environment tested on Image 1 with the boat class prediction of 0.990.

Image 2

The results from testing the models on the second real-world image of the ReVolt vessel referred to as Image 2, are shown in Figure 5.24 and 5.25.

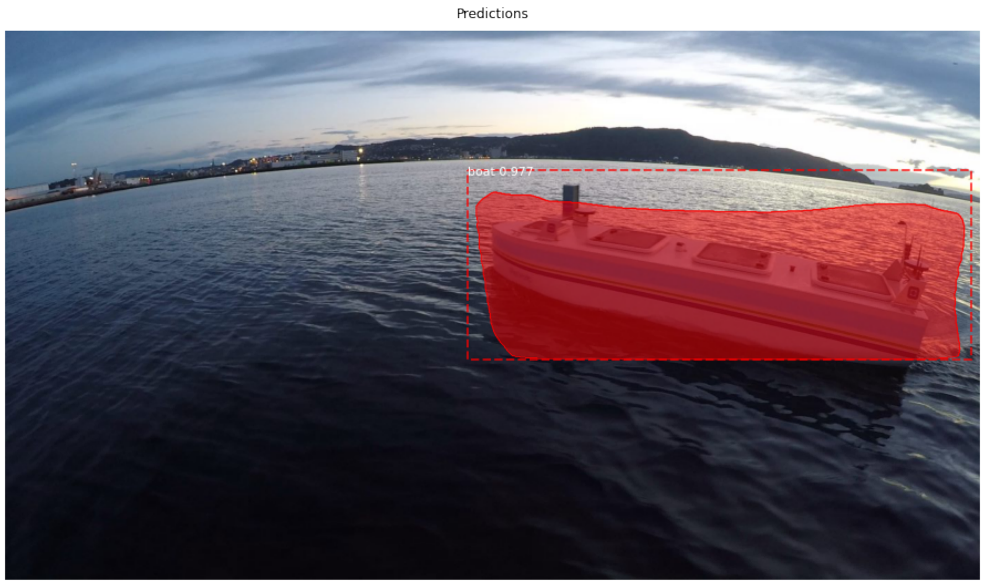


Figure 5.24: Mask R-CNN model trained in the simulated environment tested on Image 2 with the boat class prediction of 0.977.

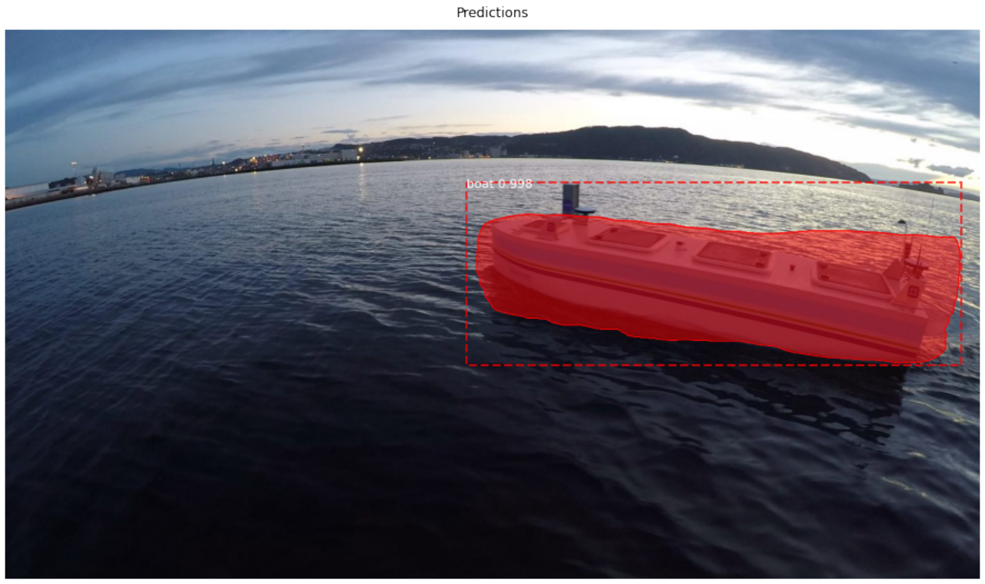


Figure 5.25: Mask R-CNN model trained in the generated environment tested on Image 2 with the boat class prediction of 0.998.

Image 3

The results from testing the models on the third real-world image of the ReVolt vessel referred to as Image 3, are shown in Figure 5.26 and 5.27.

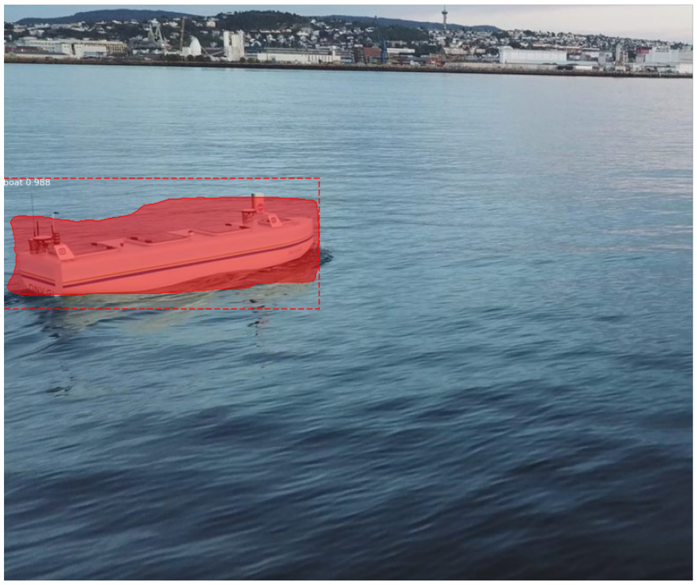


Figure 5.26: Mask R-CNN model trained in the simulated environment tested on Image 3 with the boat class prediction of 0.988.

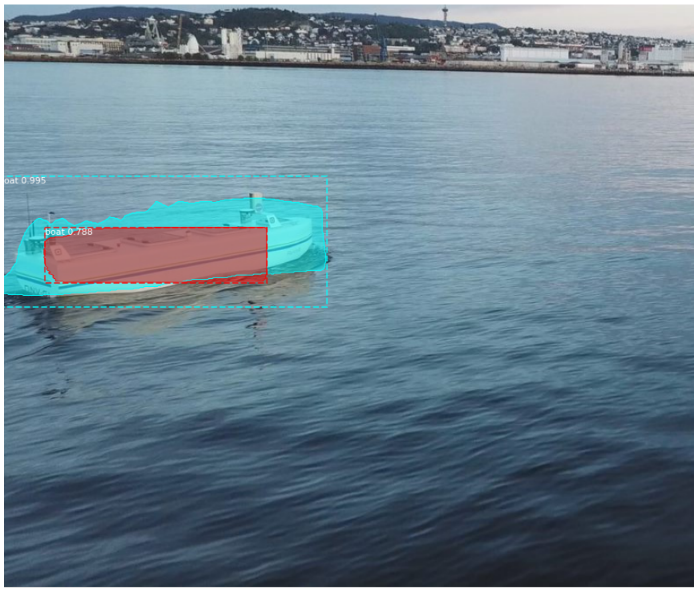


Figure 5.27: Mask R-CNN model trained in the generated environment tested on Image 3 with the boat class predictions of 0.995 and 0.788.

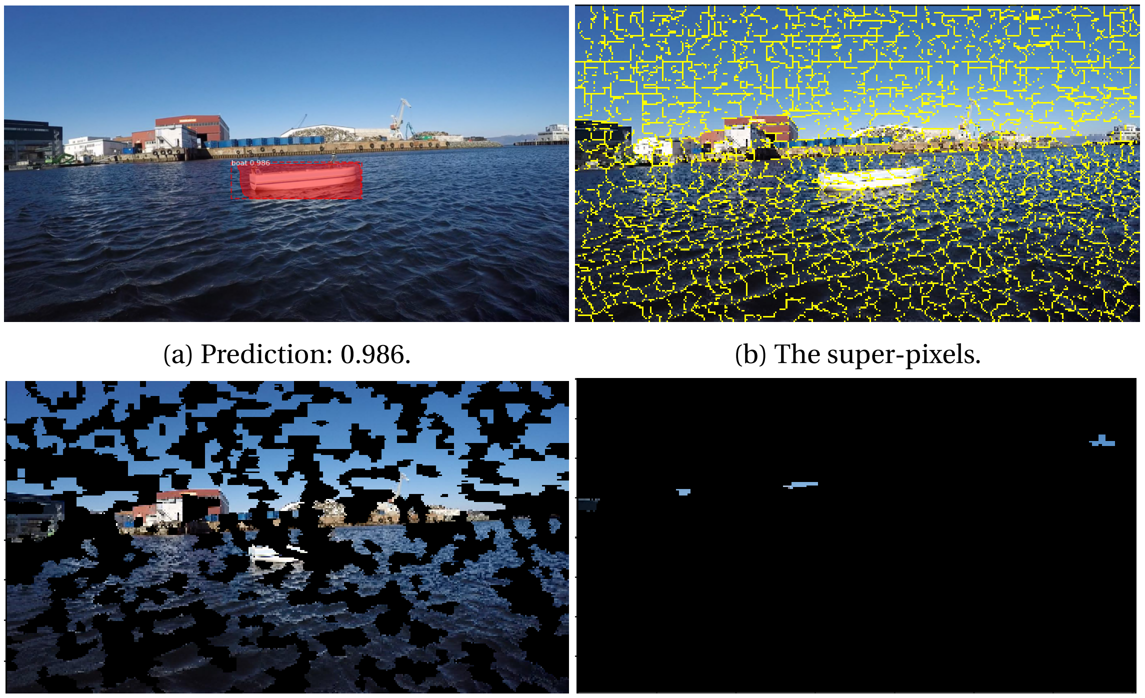
Real-world test images	**Mask R-CNN models' prediction of the boat class**	
	Trained in simulated environment	Trained in generated environment
Image 1	0.986	0.990
Image 2	0.977	0.998
Image 3	0.988	0.995 and 0.778

Table 5.2: Overview of the Mask R-CNN models' predictions on the three real-world test images.

Table 5.2 gives an overview of the predictions performed by the two models, trained in the generated- and simulated environment, on the three real-world images. The results indicates that the model trained in the generated environment generalizes better to the real-world environment than the model trained in the simulated environment. To validate the quantitative results achieved by the Mask R-CNN models, the XAI algorithm, LIME, is utilized in the below section.

5.4 Validation with LIME

This section gives an explanation for the Mask R-CNN models' predictions on Image 1, Image 2 and Image 3 by utilizing the LIME algorithm. The LIME explanation for the simulator- and

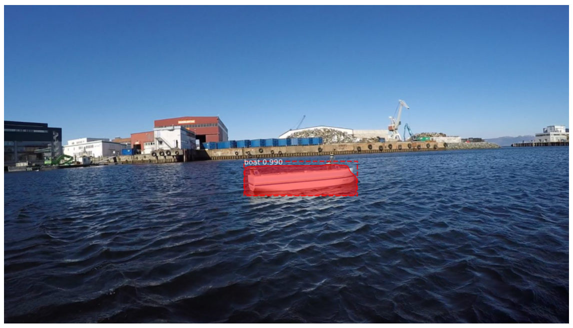


(a) Prediction: 0.986. (b) The super-pixels.

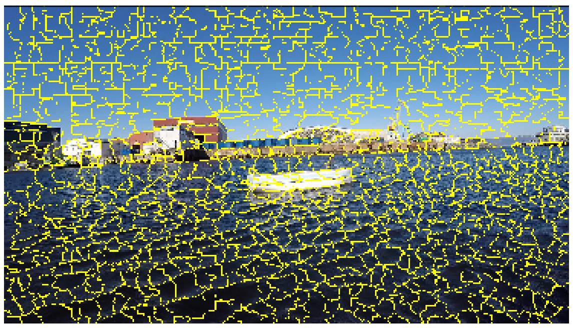
(c) A preturbed image. (d) The explanation.

Figure 5.28: LIME explanation of the Mask R-CNN simulator model's prediction on Image 1.

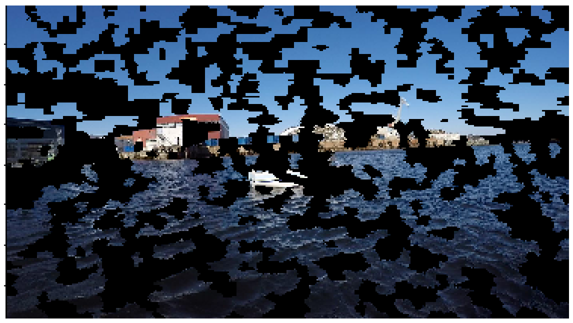
generator trained Mask R-CNN model's prediction on image 1, is illustrated in Figure 5.28 and 5.29. The simulator trained model appears to base its prediction of the boat on super-pixels of the sky, as illustrated in Figure 5.28d. The generator trained model, on the other hand, bases its prediction on a larger super-pixel of the vessel as well as some super-pixels of the sky, shown in Figure 5.29d.



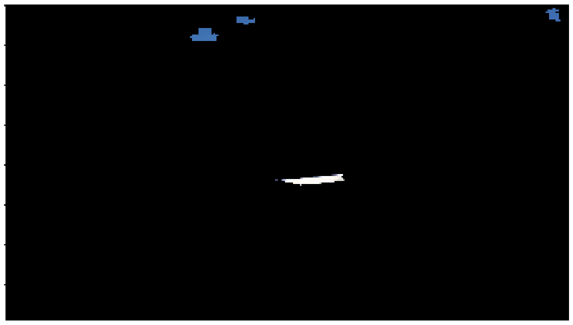
(a) Prediction: boat 0.990.



(b) The super-pixels.

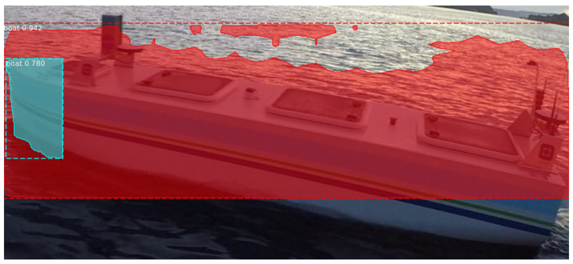


(c) A preturbed image.

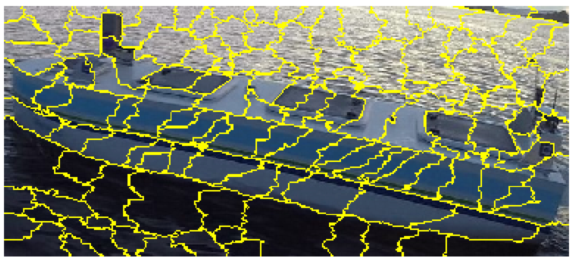


(d) The explanation.

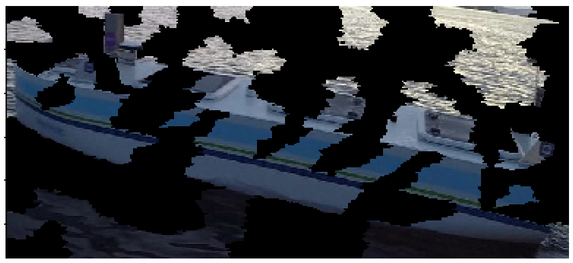
Figure 5.29: LIME explanation of the Mask R-CNN generator model's prediction on Image 1.



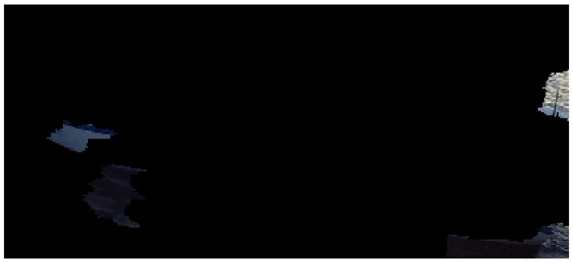
(a) Prediction: boat 0.942, boat: 0.780.



(b) The super-pixels.

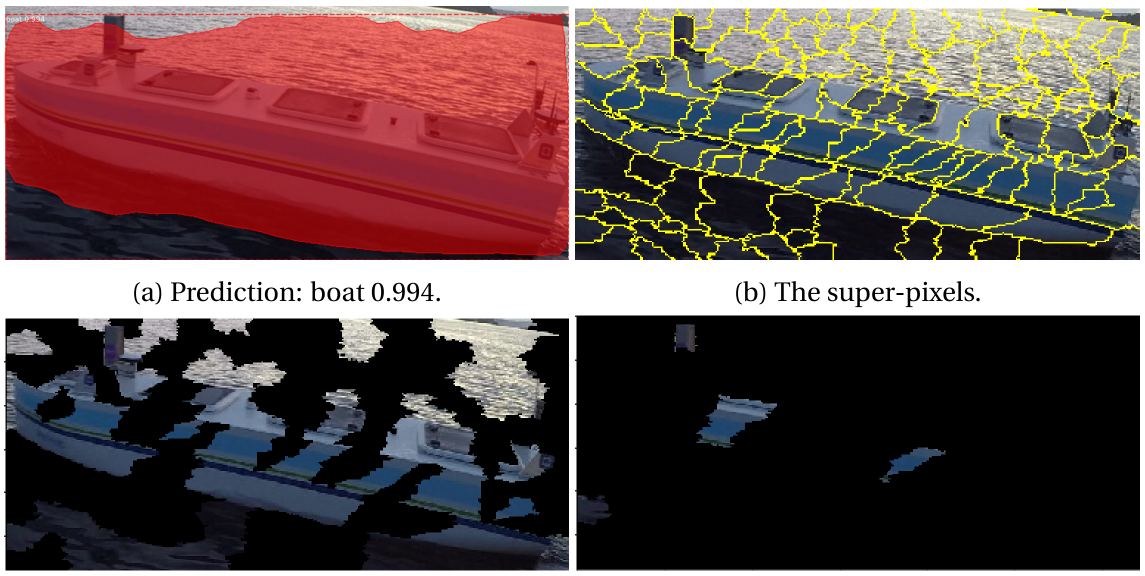


(c) A preturbed image.



(d) The explanation.

Figure 5.30: LIME explanation of the Mask R-CNN simulator model's prediction on a cropped version of Image 2.



(a) Prediction: boat 0.994.

(b) The super-pixels.

(c) A preturbed image.

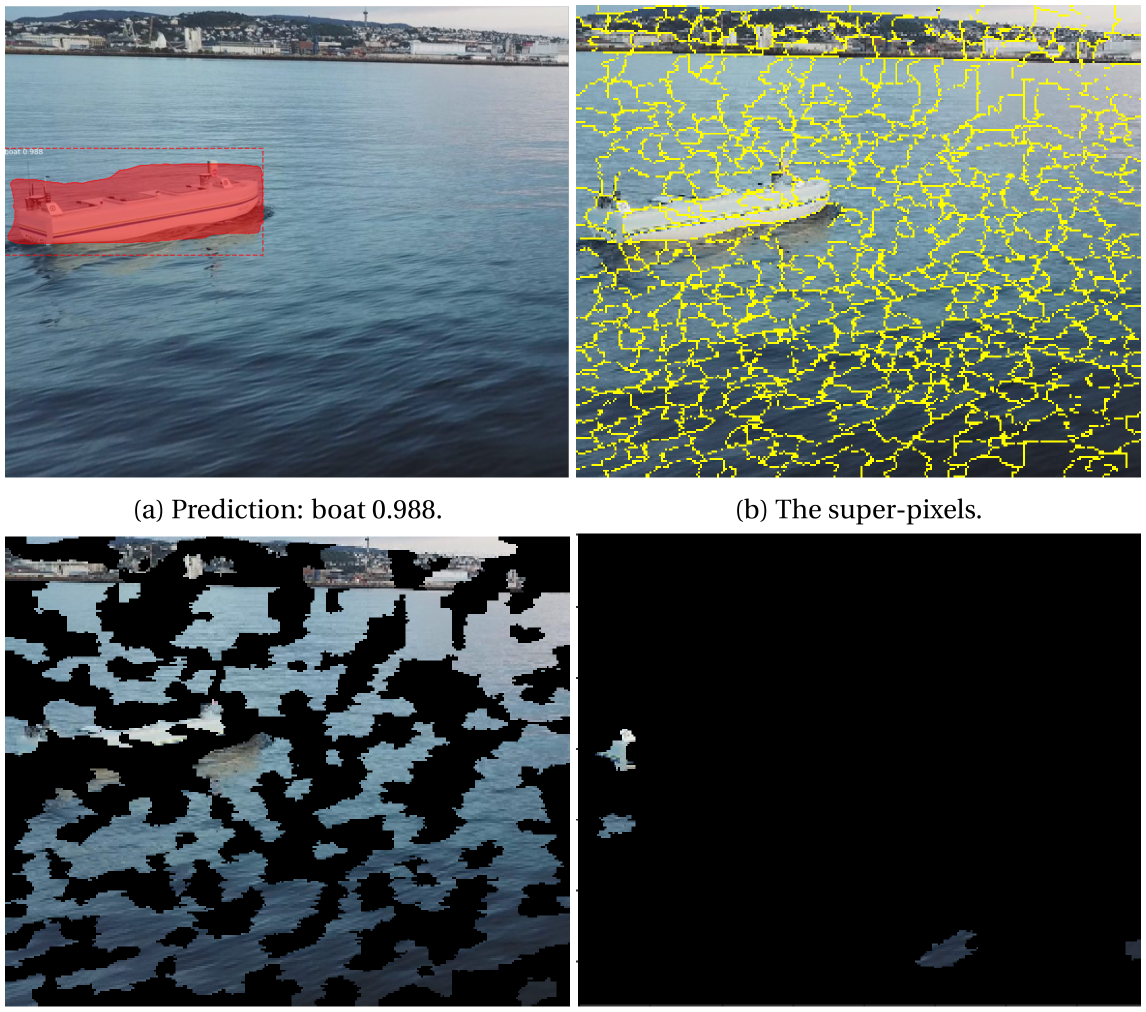
(d) The explanation.

Figure 5.31: LIME explanation of the Mask R-CNN generator model's prediction on a cropped version of Image 2.

The LIME explanation for the simulator- and generator trained Mask R-CNN model's prediction on a cropped version of Image 2, is illustrated in Figure 5.30 and 5.31 respectively. The explanations reveal that the simulator trained model use more super-pixels of the ocean when predicting the vessel in the image than the generator trained model.

The explanations of the predictions performed on Image 3 for both the simulator- and generator model, shown in Figure 5.32 and 5.33 respectively, are quite similar. Both models base its predictions on a super-pixel of the vessel and some super-pixels of the ocean. However, the prediction accuracy is higher for the generator trained model, although it is detecting two boats.

The LIME explanations, together with the Mask R-CNN predictions on the three different images, reveals that the model trained in the generated environment generalizes better than the simulator model. Moreover, although the model trained in the simulated environment detects the boat with quite a high accuracy, it mainly bases the prediction on the sky and the ocean instead of the vessel. The model trained in the generated environment, on the other hand, use more of the correct features when predicting the vessel.



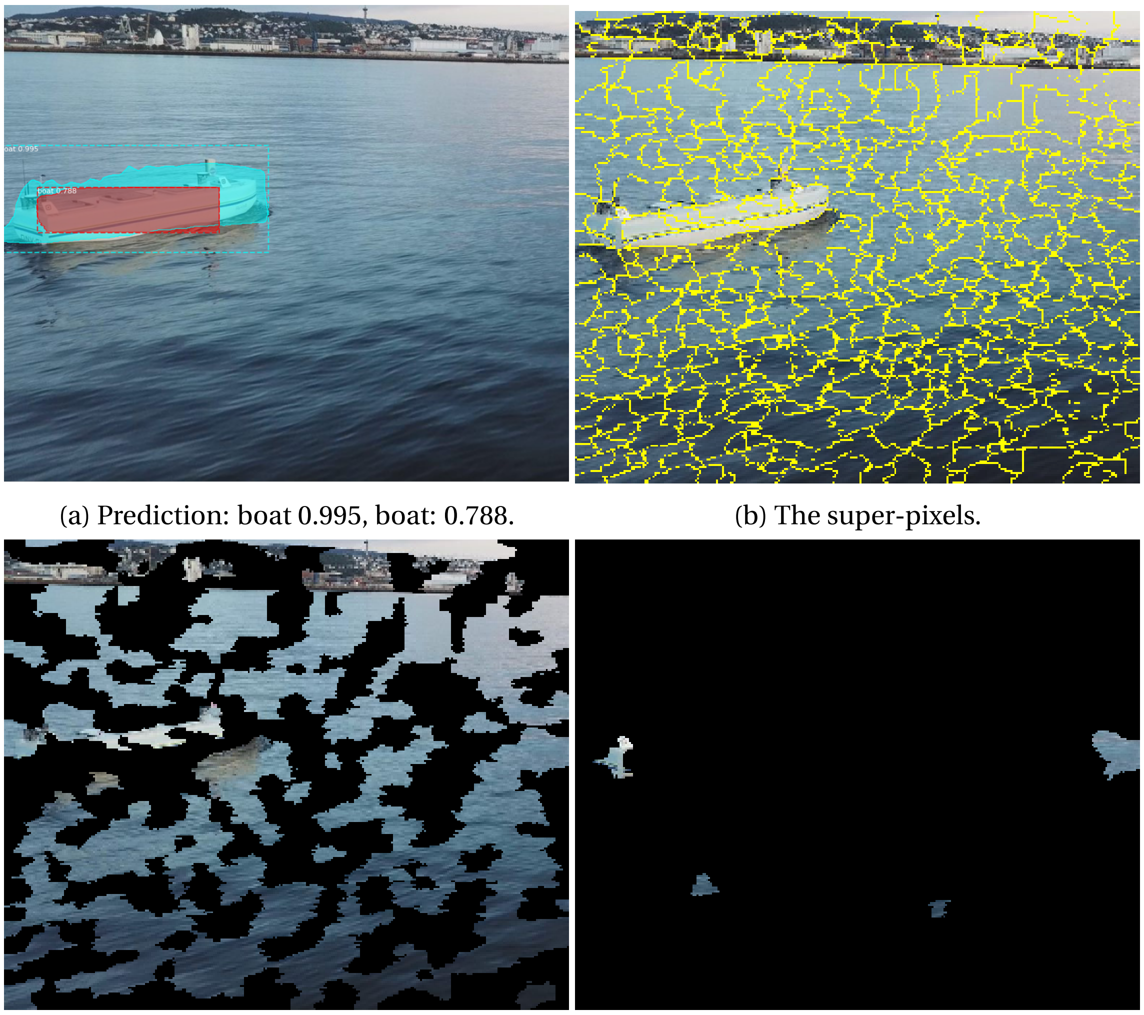
(a) Prediction: boat 0.988.

(b) The super-pixels.

(c) A preturbed image.

(d) The explanation.

Figure 5.32: LIME explanation of the Mask R-CNN simulator model's prediction on Image 3.



(a) Prediction: boat 0.995, boat: 0.788.

(b) The super-pixels.

(c) A preturbed image.

(d) The explanation.

Figure 5.33: LIME explanation of the Mask R-CNN generator model's prediction on Image 3.

Chapter 6

Conclusion and Further Work

6.1 Concluding remarks

The presented thesis has developed a complete system to synthesize photo-realistic images from a marine simulator via cycleGAN. The needed data is acquired from the ReVolt simulator as well as an OBJ file, and drone footage of the vessel in Trondheimsfjorden. Moreover, the system evaluates the generated image's added value with Mask R-CNN and data from the real-world environment. A 'Visual Turing Test' also evaluates the generated images. Furthermore, the thesis has presented several results of cycleGAN that helped in forming a self-critical discussion. The results showed that even though cycleGAN can create photo-realistic images at a high degree, photo-realism comes at the expense of mapping the correct features. Thus, the cycleGAN Model 2 managed to generate images more photo-realistic than the corresponding simulator images, but not as realistic as model 4's generated images that did not map the correct features. Consequently, Model 4's generated images got a higher accuracy of being real in the 'Visual Turing Test' than the Model 2 generated images. Since this thesis aims to minimize the reality gap to achieve better detection algorithms, mapping the correct features is essential. Thus, Model 2 was utilized to create the generated environment for training Mask R-CNN.

The results indicate that the Mask R-CNN model trained in the generated environment generalizes better to the real-world environment than the model trained in the simulated environment. Furthermore, although the model trained in the simulated environment detects the boat with quite a high accuracy, it bases the prediction on the incorrect features. The model trained in the generated environment, on the other hand, use more of the correct features when predicting the vessel. Thus, cycleGAN has improved the data quality acquired by the ReVolt vessel simulator and achieved a sufficient Mask R-CNN model. Hence, cycleGAN shows great promise in potentially improving the autonomy of the ReVolt project and contribute in the shift towards more autonomous systems in general. Nevertheless, improvements can be made to the system, which is stated in the subsection below.

6.2 Further Work

This master thesis was carried out over five months in the spring of 2020. Spending this much time on a problem brings several ideas of improving and extending the system.

Ideally, it was wanted to train the Mask R-CNN from scratch, but due to constraints in time and resources, this was not possible. Pre-trained weights had to be used, which resulted in an improvement that is not as great as desired. As mentioned in Section 3.3.4, Apple Inc. has suggested simGAN to address the problem of the reality gap, which implies that huge companies having great resources are trying to tackle these problems, and they are thus difficult to tackle with the scale of this thesis.

After training the network for more than 50 epochs, new artifacts were introduced, as mentioned in Section 5. The artifacts are random and not looking like any realistic known objects. Being able to choose what artifacts should be generated would enable more realistic generated images. Since it is not possible to choose generated artifacts, a more detailed simulator could be ideal for making the generated environment more realistic. For instance, introducing different marine crafts with their automated controller script, as well as enable for changing the weather in the simulator could be done in further work.

The usage of Google Colaboratory's GPU is time- and usage limited unless a premium account is purchased. Unfortunately, today, it is only possible to use the premium version of Google Colaboratory in the United States. Training and testing the model in this thesis was, therefore constrained. For further work, the GPU power needed should be gained from somewhere else where unlimited access is available.

In this thesis, cycleGAN was utilized to generate photo-realistic images that could be used to close the reality gap. As stated in 3.3.5, coGAN and simGAN were also potential candidates to reach the goal of this thesis. CycleGAN was chosen by comparing results from another paper testing different GANs, illustrated in Figure 3.22. The comparison is based on a mapping between different domains than the ones in this thesis. The results from utilizing the different GANs for the issue of this thesis could, therefore, be different. For further work on closing the reality gap for the simulator of the autonomous ReVolt vessel, coGAN and simGAN could improve the system.

Bibliography

[1] Bekkeheien L. *Supervised Learning and its Development to Generative Adversarial Networks.* 2019;78.

[2] Zhu JY. *junyanz/pytorch-CycleGAN-and-pix2pix*; 2020. Original-date: 2017-04-18. Available from: `https://bit.ly/2WPqbOq`.

[3] Abdulla W. *Mask R-CNN for object detection and instance segmentation on Keras and TensorFlow.* Github; 2017. Available from: `https://bit.ly/2LM7IMH`.

[4] Arteaga C. *Interpretable Machine Learning with LIME for Image Classification*;. Available from: `https://colab.research.google.com/github/arteagac/arteagac.github.io/blob/master/blog/lime_image.ipynb#scrollTo=qtG5BIcuU3w6`.

[5] Fossen T. *Handbook of Marine Craft Hydrodynamics and Motion Control.* John Wiley Sons, Ltd; 2011.

[6] Neves A, Gonzalez I, Leander J, Karoumi R. *A New Approach to Damage Detection in Bridges Using Machine Learning*; 2018. p. 73–84.

[7] Loiseau JCB. *Rosenblatt's perceptron, the very first neural network*; 2019. Available from: `https://towardsdatascience.com/rosenblatts-perceptron-the-very-first-neural-network-37a3ec09038a`.

[8] Giles M. *The GANfather: The man who's given machines the gift of imagination*;. Available from: `https://bit.ly/2zaVISe`.

[9] Foster D. *Generative Deep Learning: Teaching Machines to Paint, Write, Compose, and Play.* O'Reilly Media, Inc.; 2019.

[10] Valle R. *Hands-On Generative Adversarial Networks with Keras: Your guide to implementing next-generation generative adversarial networks.* Packt Publishing Ltd.; 2019.

[11] Sapunov G. *Welcome to the Simulation.* Intento; 2019. Available from: `https://bit.ly/2zim1pt`.

[12] Isola P, Zhu JY, Zhou T, Efros A. *Image-to-Image Translation with Conditional Adversarial Networks*; 2017. p. 5967–5976.

[13] Zhu JY, Park T, Isola P, Efros AA. *Unpaired Image-To-Image Translation Using Cycle-Consistent Adversarial Networks.* In: *The IEEE International Conference on Computer Vision (ICCV)*; 2017. .

[14] Liu MY, Tuzel O. *Coupled Generative Adversarial Networks.* In: *Proceedings of the 30th International Conference on Neural Information Processing Systems.* NIPS'16. Red Hook, NY, USA: Curran Associates Inc.; 2016. p. 469–477.

[15] Liu MY, Breuel T, Kautz J. *Unsupervised Image-to-Image Translation Networks.* In: *Proceedings of the 31st International Conference on Neural Information Processing Systems.* NIPS'17. Red Hook, NY, USA: Curran Associates Inc.; 2017. p. 700–708.

[16] Shrivastava A, Pfister T, Tuzel O, Susskind J, Wang W, Webb R. *Learning from Simulated and Unsupervised Images through Adversarial Training.* In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*; 2017. p. 2242–2251.

[17] Girshick R, Donahue J, Darrell T, Malik J. *Rich feature hierarchies for accurate object detection and semantic segmentation.* 2014;Available from: `http://arxiv.org/abs/1311.2524`.

[18] Girshick R. *Fast R-CNN.* 2015;Available from: `http://arxiv.org/abs/1504.08083`.

[19] Ren S, He K, Girshick R, Sun J. *Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks.* Curran Associates, Inc.; 2015. Available from: `https://bit.ly/3b0epYZ`.

[20] He K, Gkioxari G, Dollár P, Girshick R. *Mask R-CNN*. 2018;Available from: http://arxiv.org/abs/1703.06870.

[21] Weng L. *Object Detection for Dummies Part 3: R-CNN Family*. 2017;Available from: https://bit.ly/2LKTpYr.

[22] Ribeiro MT, Singh S, Guestrin C. *"Why Should I Trust You?" Explaining the Predictions of Any Classifier*. 2016;Available from: http://arxiv.org/abs/1602.04938.

[23] Technologies U. *Unity - Scripting API*;. Available from: https://docs.unity3d.com/ScriptReference/Rect.html.

[24] *ReVolt*; 2017. Available from: https://www.autonomousshipshq.com/revolt/.

[25] Goodfellow IJ, Pouget-Abadie J, Mirza M, Xu B, Warde-Farley D, Ozair S, et al. *Generative Adversarial Nets*. In: *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*; 2014. Available from: https://bit.ly/2WO1z8R.

[26] Richter SR, Vineet V, Roth S, Koltun V. *Playing for Data: Ground Truth from Computer Games*. 2016;Available from: http://arxiv.org/abs/1608.02192.

[27] Skaldebø M, Muntadas AS, Schjølberg I. *Transfer Learning in Underwater Operations*. In: *OCEANS 2019 - Marseille*; 2019. p. 1–8.

[28] Gupta A, Booher J. *CycleGAN for sim2real Domain Adaptation*;p. 8.

[29] Bousmalis K, Irpan A, Wohlhart P, Bai Y, Kelcey M, Kalakrishnan M, et al. *Using Simulation and Domain Adaptation to Improve Efficiency of Deep Robotic Grasping*. 2017;Available from: http://arxiv.org/abs/1709.07857.

[30] Sallab AE, Sobh I, Zahran M, Essam N. *LiDAR Sensor modeling and Data augmentation with GANs for Autonomous driving*. 2019;Available from: http://arxiv.org/abs/1905.07290.

[31] Galbusera F, Niemeyer F, Seyfried M, Bassani T, Casaroli G, Kienle A, et al. *Exploring the Potential of Generative Adversarial Networks for Synthesizing Radiological Images of the Spine*

to be Used in In Silico Trials. Frontiers in Bioengineering and Biotechnology. 2018;Available from: https://www.frontiersin.org/articles/10.3389/fbioe.2018.00053/full#B8.

[32] Lee KH, Ros G, Li J, Gaidon A. *SPIGAN: Privileged Adversarial Learning from Simulation.* 2019;Available from: http://arxiv.org/abs/1810.03756.

[33] Saseendran A. *Learning from Simulation to Improve Robotic Sensing.* Universität Bremen; 2019. Available from: https://elib.dlr.de/131943/.

[34] Maffione W. *Addressing Domain Shift between Real and Synthetic data for Semantic Segmentation with GANs.* 2020;p. 86.

[35] Ketkar N. *Introduction to PyTorch.* In: *Deep Learning with Python: A Hands-on Introduction.* Berkeley, CA: *Apress*; 2017. p. 195–208. Available from: https://doi.org/10.1007/978-1-4842-2766-4_12.

[36] Chollet F, et al.. *Home - Keras Documentation*; 2015. Available from: https://keras.io/.

[37] Abadi M, Agarwal A, Barham P, Brevdo E, Chen Z, Citro C, et al.. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*; 2015. Available from: https://www.tensorflow.org/.

[38] Haas JK. *A history of the unity game engine.* 2014;.

[39] Géron A. *Hands-on machine learning with Scikit-Learn and TensorFlow: concepts, tools, and techniques to build intelligent systems.* O'Reilly Media, Inc.; 2017.

[40] *Google Trends*; 2020. Available from: https://trends.google.com/trends/explore?date=2010-04-18%202020-04-18&q=machine%20learning.

[41] Delude CM. *Computing Intelligence. Brain Scan.* 2011;Available from: https://mcgovern.mit.edu/wp-content/uploads/2019/01/brainscan{_}issue19.pdf.

[42] Goodfellow I, Bengio Y, Courville A. *Deep learning.* MIT press Ltd; 2016.

[43] Rosenblatt FF. *The perceptron: a probabilistic model for information storage and organization in the brain. Psychological review.* 1958;65 6. Available from: `https://doi.org/10.1037/h0042519`.

[44] Minsky M, Papert S. *Perceptrons: An Introduction to Computational Geometry*; 1969.

[45] Rumelhart DE, Hinton GE, Williams RJ. *Learning representations by back-propagating errors. Nature.* 1986;323. Available from: `https://www.nature.com/articles/323533a0`.

[46] Isola P. *phillipi/pix2pix*; 2019. Available from: `https://github.com/phillipi/pix2pix`.

[47] Ngiam J, Khosla A, Kim M, Nam J, Lee H, Ng AY. *Multimodal Deep Learning.* In: *Proceedings of the 28th International Conference on International Conference on Machine Learning.* ICML'11. Madison, WI, USA: Omnipress; 2011. p. 689–696.

[48] Wang S, Zhang L, Liang Y, Pan Q. *Semi-Coupled Dictionary Learning with Applications to Image Super-Resolution.* In: *and Photo-Sketch Image Synthesis," Proc. IEEE Conf. Computer Vision and Pattern Recognition*; 2012. p. 2216–2223.

[49] Srivastava N, Salakhutdinov R. *Multimodal Learning with Deep Boltzmann Machines.* J Mach Learn Res. 2014 Jan;p. 2949–2980.

[50] Salimans T, Goodfellow I, Zaremba W, Cheung V, Radford A, Chen X. *Improved Techniques for Training GANs.* In: *Proceedings of the 30th International Conference on Neural Information Processing Systems.* Curran Associates Inc.; 2016. p. 2234–2242.

[51] Mao X, Li Q, Xie H, Lau RYK, Wang Z, Smolley SP. *Least Squares Generative Adversarial Networks.* arXiv:161104076 [cs]. 2017;Available from: `http://arxiv.org/abs/1611.04076`.

[52] Lin TY, Maire M, Belongie S, Bourdev L, Girshick R, Hays J, et al. *Microsoft COCO: Common Objects in Context.* 2015;Available from: `http://arxiv.org/abs/1405.0312`.

[53] *ImageNet*;. Available from: `http://www.image-net.org/`.