FHV
Vorarlberg University
of Applied Sciences

Object Detection of PLC Modules Using CAD-Based Synthetic Data Improved With GANs

Master Thesis
to obtain the academic degree

Master of Science in Engineering (MSc)

Vorarlberg University of Applied Sciences
Computer Science

Supervised by
DI Dr. techn. Sebastian Hegenbart

Submitted by
Lukas Lins, BSc

Dornbirn, July 2023

Abstract

Programmable Logic Controller (PLC) modules are used in industrial settings to control and monitor various manufacturing processes. Detecting these modules can be helpful during installation and maintenance. However, the limited availability of real annotated images to train an object detector poses a challenge. This thesis aims to research object detection of these modules on real images by using synthetic data during training. The synthetic images are generated from CAD models and improved with Generative Adversarial Networks (GANs).

The CAD models are rendered in different scenes, and perfectly annotated images are automatically saved. A technique called domain randomization is applied during rendering. It renders the modules in different poses with constantly changing backgrounds. As the CAD models do not visually resemble the real modules, it is necessary to improve the synthetic images. This project researches StarGAN and CycleGAN for the task of image-to-image translation. A GAN is trained with real and synthetic images and can then translate between these domains.

YOLOv8 and Faster R-CNN are tested for object detection. The best mean Average Precision (mAP) is achieved when training with a synthetic dataset where 50% of the images were improved with StarGAN. When trained with YOLOv8 and evaluated on a real dataset, it achieves a mAP of 84.4%. Overall, the accuracy depends on the quality of the CAD models. Using a GAN improves the detection rate for all modules, but especially for unrealistic CAD models.

Kurzreferat

PLC Module werden in der Industrie zur Steuerung und Überwachung verschiedener Maschinen und Anlagen eingesetzt. Während der Installation und Wartung kann eine Objekterkennung dieser Module sehr hilfreich sein. Dies wird jedoch erschwert, da für deren Training kaum reale Bilder zur Verfügung stehen. Diese Masterarbeit untersucht daher das Trainieren einer Objekterkennung anhand von synthetischen Daten. Mit CAD Modellen werden synthetische Bilder generiert, die dann anschließend mit GANs (Generative Adversarial Networks) verbessert werden.

Die CAD Modelle werden in verschiedenen Szenen gerendert und als gelabelte Bilder gespeichert. Während dem Rendering wird Domain Randomization angewendet. Mit dieser Methode werden die Modelle in verschiedenen Posen und unterschiedlichen Hintergründen gerendert. Weil die gerenderten Bilder aber weiterhin nicht realistisch sind, werden sie anschließend mit GANs verbessert. Dazu werden StarGAN und CycleGAN verwendet. Ein GAN wird mit realen und synthetischen Bildern trainiert und kann anschließend Bilder zwischen diesen Domänen umwandeln.

Für die Objekterkennung werden YOLOv8 und Faster R-CNN untersucht. Die beste Genauigkeit wird beim Training mit einem synthetischen Datensatz erreicht, der zu 50% mit StarGAN verbessert wurde. YOLOv8 erreicht dort bei der Evaluierung mit realen Bildern einen mAP (mean Average Precision) von 84.4%. Generell ist die Genauigkeit bei der Objekterkennung von der Qualität des CAD Modells abhängig. Die Verwendung eines GANs verbessert die Genauigkeit bei allen Modulen, besonders aber bei denen mit unrealistischen CAD Modellen.

Contents

1 Introduction **6**
1.1 Problem Statement . . . 6
1.2 Aim of the Work . . . 7
1.3 State of the Art . . . 8
1.3.1 GANs . . . 8
1.3.2 Object Detectors . . . 10

2 Experiments **13**
2.1 Initial Data . . . 13
2.2 Synthetic Image Generation . . . 16
2.2.1 Rendering Engine . . . 16
2.2.2 CAD Model Conversion . . . 18
2.2.3 Image Generation . . . 19
2.3 Image Enhancement With GANs . . . 24
2.3.1 Experiments . . . 25
2.3.2 Implementation . . . 31
2.4 Object Detection . . . 37
2.4.1 YOLO . . . 37
2.4.2 Faster R-CNN . . . 38
2.4.3 Real Validation Dataset . . . 39

3 Results **41**
3.1 Overview of Results . . . 41
3.2 GANs . . . 44
3.3 Object Detectors . . . 46

4 Summary and Outlook **48**
4.1 Summary . . . 48
4.2 Outlook . . . 50

Bibliography **51**

List of Acronyms **55**

List of Figures **56**

List of Tables **57**

1 Introduction

Programmable Logic Controller (PLC) modules are part of a PLC system specifically designed to control and monitor machinery and processes in industrial settings. They consist of different modules put onto the wall in a rack or chassis. Object detection can be very useful for detecting these modules during installation and troubleshooting.

However, this task is difficult as the modules are used in many application areas. For example, depending on the application, the modules are mounted onto different surfaces with different cables plugged into them. These cables also cover part of the module, making identifying them difficult.

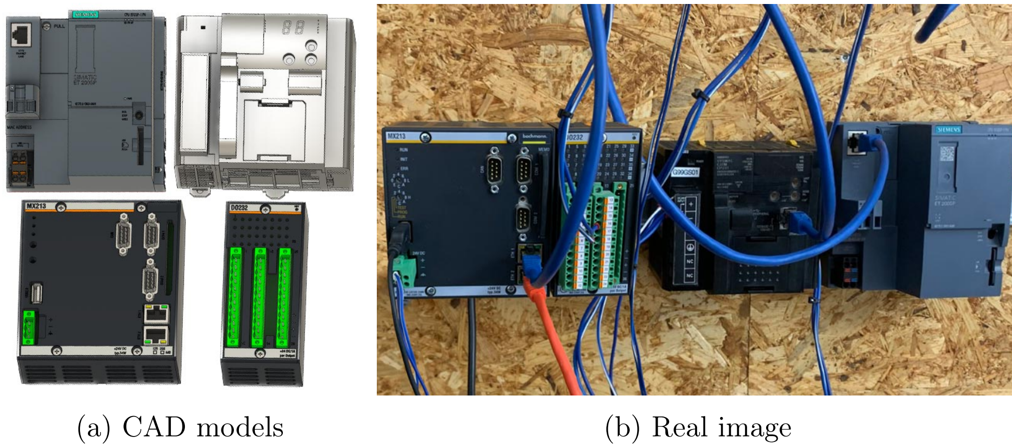
Object detectors require thousands of images during training. Creating and labeling real images for all different scenarios is time-consuming. But for PLC modules, Computer-Aided Design (CAD) models are always available as they are used during development and assembly. This thesis aims to research object detection based on these CAD models without the need for real labeled images.

1.1 Problem Statement

Computer vision projects require vast amounts of training data. Creating these datasets is an expensive and time-consuming task. Images must be collected and labeled by human annotators. It also brings various challenges related to a possible bias in the dataset. For example, creating variations that only rarely occur in the real world can be very difficult. In addition, datasets are often created for a specific domain and require new data when transferred to another domain.

An alternative to a real dataset is training based on synthetic data. A virtual environment is used to create images in different scenarios. These scenarios can be created in a controllable and customizable manner. As an advantage, thousands of images can be created within minutes and are labeled automatically.

Synthetic image data is mostly based on realistic 3D models. However, in the area of mechanical engineering, realistic 3D models are often not available. Instead, CAD models are commonly used during development and assembly. These are technical models with the correct dimensions but do not always look like the real object as visualized in Figure 1.1.



(a) CAD models (b) Real image

Figure 1.1: Example PLC modules

For example, CAD models often have different levels of detail, and text printed on the surface is not drawn on the CAD model. Some models have the same surface color as the actual module but do not include any surface textures.

Additionally, PLC modules have different ports and connectors. For example, they can have connectors for power, USB, Ethernet, or various analog outputs. In the real world, different cables are plugged into them depending on the application. These cables or plugs are not drawn in the CAD model and may covert parts of the module on a real image.

1.2 Aim of the Work

This thesis aims to research 2D object detection of PLC modules with deep learning methods based on synthetic data. Initially, CAD files of different PLC modules are used to create a synthetic dataset. The CAD models have different detail levels and are not photo-realistic. Therefore, generative approaches are researched to improve the realism of the dataset. These methods require a set of real data during training to improve the quality and make the synthetic images more realistic.

Additionally, the real images have PLC modules with plugs and cables plugged into them. During development, it is unknown which connectors have cables plugged in. As they will occlude part of the module on an actual image, it should be considered that parts of the module may not be visible.

In the end, an object detector is trained using the generated dataset. The goal is to annotate real images with bounding boxes and labels.

1.3 State of the Art

Overall, object detection algorithms are primarily trained on real data. However, there is some research in the field of synthetic data. Most authors use realistic 3D models, which is not available for this project: [1], [2], [3]. Some papers use CAD models, but the used objects mostly have no textures and monochrome surfaces in CAD and reality: [4], [5], [6].

No direct state-of-the-art methods exist for the entire use case of this project. However, the project will use GANs to make synthetic images more realistic and object detectors to detect the modules in an image. The algorithms and models used in this project are described in the following paragraphs.

1.3.1 GANs

Generative Adversarial Networks (GANs) are deep learning models capable of learning representations and patterns to generate new data instances. A GAN typically consists of two components: a generator and a discriminator. The generator aims to create images that resemble a given target domain. The discriminator differentiates between a real and generated image. Based on the feedback of the discriminator, the generator learns how well it translated the image. Repeating this process, the generator constantly learns to generate better images.

Various papers have conducted research in GANs to convert images between two domains. Pix2Pix [7] is the best-known paper for paired image-to-image translation. It requires exactly the same images from two different domains. Others allow for an unpaired image-to-image translation. The most known are CycleGAN [8] and StarGAN [9].

CycleGAN and StarGAN are used for this project. Their architecture and functionality are described in the following paragraphs.

CycleGAN

CycleGAN was introduced in 2017 as the first model to perform image-to-image translation without paired data [8]. It supports only two domains and translates an image from an input to an output domain.

The CycleGAN model [8] has two generators G and F and two discriminators D_X and D_Y for the domains X and Y. Illustrated in Figure 1.2a are the two mapping functions: $G : X-> Y$ and $F : Y-> X$. Discriminator D_X learns to distinguish between image x and translated image $\hat{x}$, discriminator D_Y between y and $\hat{y}$.

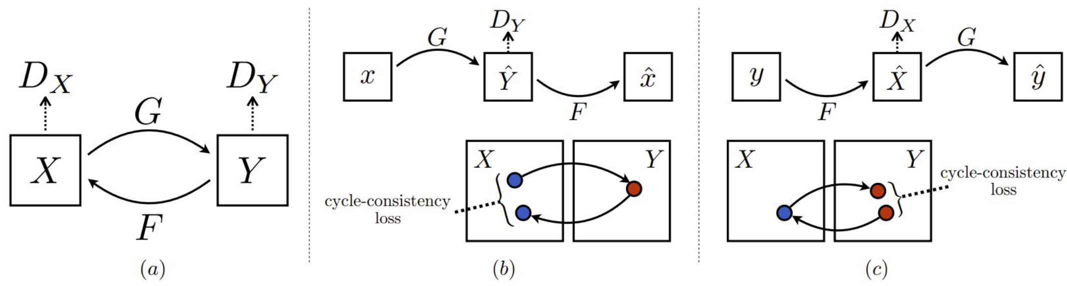


Figure 1.2: Overview of the CycleGAN model (source: [8])

The model uses cycle-consistency to ensure the resulting image looks similar to the original image. Figure 1.2b displays the forward cycle-consistency loss. Image x is passed to the generator, which generates image $\hat{y}$. $\hat{y}$ is then passed to generator F which results in a cycled image $\hat{x}$. The cycle-consistency loss is then calculated as the mean absolute error between X and $\hat{X}$. The same process is calculated vice-versa for the backward cycle-consistency loss in Figure 1.2c

StarGAN

StarGAN is a type of GAN designed for multi-domain image-to-image translation tasks. Unlike CycleGAN, it allows for translation between multiple domains using a single unified model. The first version of StarGAN [9] was published in 2017. The successor StarGANv2 [10] was published in 2020 and is used for this project. It improved the architecture by adding two new modules: A mapping network and a domain-specific style encoder. Besides minor improvements, this leads to better image quality and fewer artifacts [10].

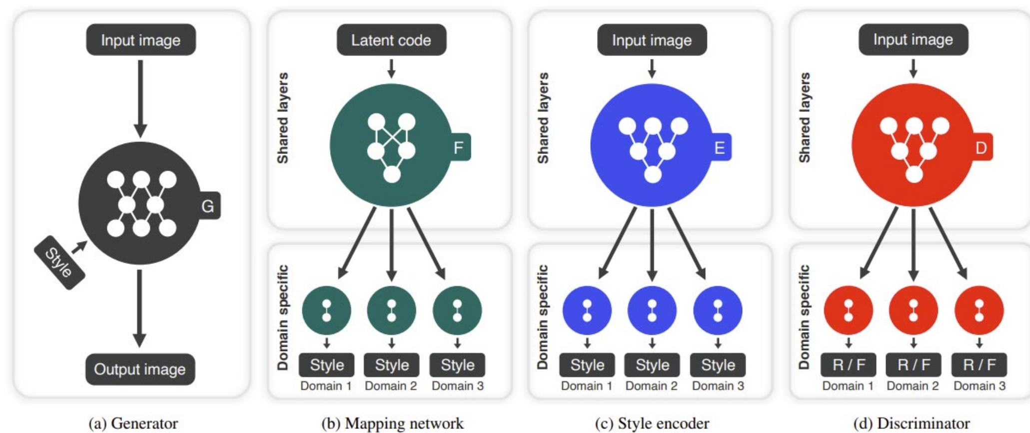


Figure 1.3: Overview of the StarGANv2 modules (source: [10])

The StarGANv2 architecture consists of four main modules, visualized in Figure 1.3. The goal is to train one generator G to generate images for different domains, visualized in Figure 1.3a. It translates an input image to an output image which includes the domain-specific style code.

The mapping network F in Figure 1.3b transforms latent code into style code for multiple domains. During training, one style code is randomly selected.

The style encoder E in Figure 1.3c extracts the style code of an image. The generator randomly selects one style code to perform reference-guided image synthesis. In Figure 1.3d, the discriminator D distinguishes between real and fake images.

1.3.2 Object Detectors

The state-of-the-art object detectors can be categorized into two main methods: one-stage and two-stage. One-stage methods directly predict bounding boxes and labels in a single pass over the image. They are designed to prioritize speed and enable real-time object detection. The most popular implementation of this method is YOLO.

Two-stage methods are also known as region-based detectors and consist of two stages. The first stage generates a set of potential object regions. The second stage then classifies the proposed object regions. The most popular implementation of a two-stage method is Faster R-CNN.

YOLO and Faster R-CNN are used for this project and will be described in more detail in the following paragraphs.

YOLO

The first You Only Look Once (YOLO) model was introduced in 2015 as a real-time object detection system [11]. YOLO is a one-stage algorithm predicting bounding boxes and class probabilities with a single neural network, making it extremely fast.

Over the years, the YOLO family was constantly improved with new versions: YOLOv2 (2016) [12], YOLOv3 (2018) [13], YOLOv4 (2020) [14], YOLOv5 (2020) [15], YOLOv6 (2022) [16], YOLOv7 (2022) [17], and YOLOv8 (2023) [18]. Most improvements are incremental and do not change the fundamental ideal of the algorithm. For a detailed history and review of different YOLO versions, see [19]. All versions focus on balancing speed and accuracy, constantly improving them and minimizing the tradeoff between these two factors.

This project uses YOLOv8 [18], the newest version of YOLO. It is developed by Ultralytics, the same company that developed YOLOv5. It was released in January 2023, but no detailed paper has been released yet. YOLOv8 is

still under active development as of July 2023, but mainly for the semantic segmentation model, which is not used in this project.

As no official paper has been released, it is difficult to mention the differences compared to previous versions. The authors of [20] explain the new architecture in detail. The improvements result in higher accuracy, with a slightly slower CPU inference speed but a higher GPU speed. The authors of [21] compare YOLOv5, YOLOv7, and YOLOv8 and confirm the improved accuracy.

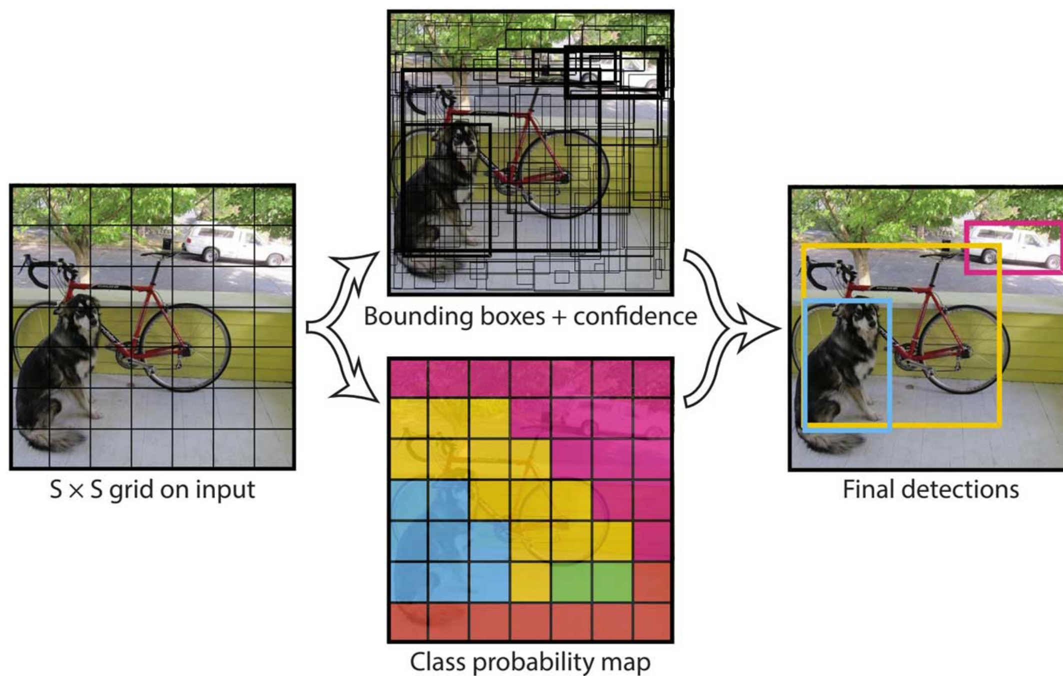


Figure 1.4: YOLO model (source: [11])

Figure 1.4 provides an overview of the YOLO model and its fundamental architecture [11]. An input image is divided into an $S \times S$ grid of cells. Each cell predicts bounding boxes with a confidence score, as shown in the top middle image in Figure 1.4. This score indicates if the grid cell contains an object and the accuracy of the predicted box.

Additionally, each cell predicts conditional probabilities $Pr(Class_i|Object)$ for all classes. These predict the probability of an object belonging to each class, displayed in the middle bottom image in Figure 1.4.

The conditional class probabilities and individual bounding box confidence scores are multiplied, resulting in class-specific confidence scores for each box. These scores are the final detections, displayed in the right image in Figure 1.4. They indicate the presence of a class within the bounding box and the predicted accuracy of how well the predicted box fits the actual object.

Faster R-CNN

Faster R-CNN [22] was developed in 2016 as a real-time object detection network. It is a two-stage algorithm and uses a Region Proposal Network (RPN).

The first predecessor of the Region-based Convolutional Neural Network (R-CNN) family was released in 2014 as R-CNN [23], which invented the region proposal method. Fast R-CNN [24] followed in 2015, improving the performance and simplifying the algorithm. Faster R-CNN was published in 2016 and significantly improved the object detection speed.

Since then, newer versions have been released for semantic segmentation, for example, Mask R-CNN [25], and Cascade R-CNN [26]. These are built on Faster R-CNN but added and improved the semantic segmentation. This project does not require semantic segmentation, so the original Faster R-CNN model is used.

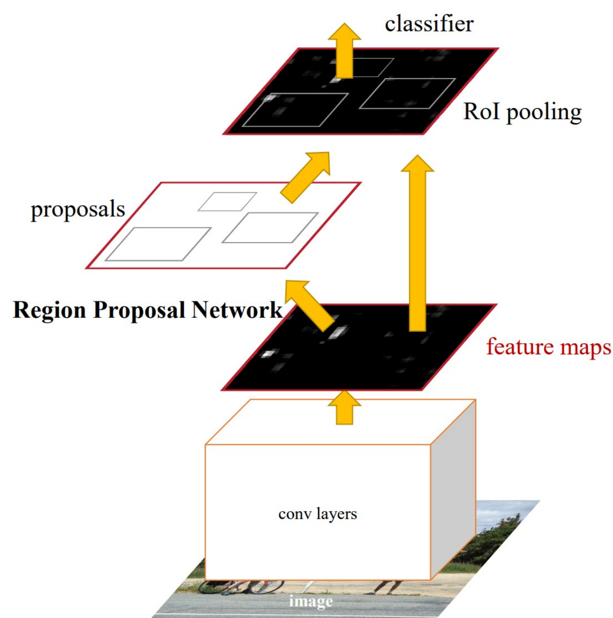


Figure 1.5: Faster R-CNN model (source: [22])

The Faster R-CNN model is composed of two stages, displayed in Figure 1.5. The first stage consists of an Convolutional Neural Network (CNN) that extracts feature maps from an input image. These features are then used by the RPN to generate object proposals. The proposals include an objectiveness score used to measure the likelihood of the region containing an object.

The second stage is then responsible for object classification and refinement. It takes the region proposals, classifies the objects within each region, and predicts the final bounding boxes.

2 Experiments

This chapter describes the experiments conducted for the various aspects of this project. The first section starts by having a look at the used PLC modules. It compares the visual differences between the CAD and real models. The next section then describes the process of generating pure synthetic images. This includes converting a CAD to a 3D model and rendering the objects in different scenarios.

As the CAD models do not always visually reflect the real module, improving the quality and realism of synthetic images is necessary. Therefore, the next section discusses the enhancement of images with GANs. The idea is to learn the style of real images with a GAN and transfer the synthetic images to make them more realistic. In the end, the object detection task is described in detail using two real-time object detectors: YOLO and Faster R-CNN.

2.1 Initial Data

The PLC modules are available as CAD files with different levels of detail. Computer-Aided Design (CAD) is used during manufacturing to construct mechanical objects with precise measurements. They are technical drawings where the dimensions and measurements have the highest priority.

However, this means they are not photo-realistic due to missing details, as the visual appearance is not that important. For example, textures are often missing entirely, or the color of the CAD model does not correspond with the color of the real module.

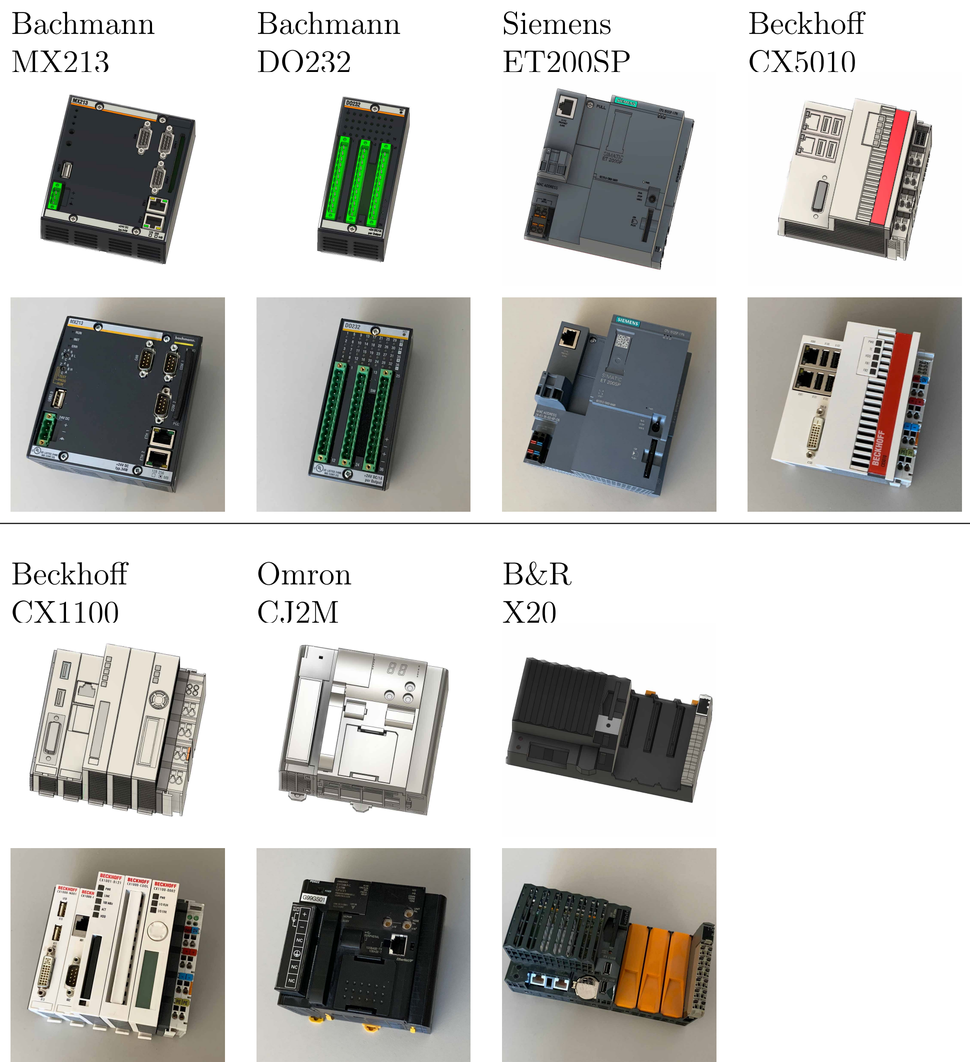


Figure 2.1: CAD and real images of used PLC modules

For this thesis, seven PLC modules with different levels of detail were chosen, displayed in Figure 2.1. All modules have slightly different surface textures on the actual module, as CAD models do not contain any rendering information. For example, the surface of a CAD model may have the same color, but it does not contain any information on how the surface reflects the light and if it should be shiny, reflective, or matt.

The Bachmann MX213, Bachmann DO232, and Siemens ET200SP CAD models have the highest detail level, only missing some text and imprints on the surface.

The Beckhoff CX5010 and Beckhoff CX1100 modules have less detailed CAD

models. The CAD objects miss text printed on the surface, and connectors are not fully modeled. The colors are slightly off. The actual module has a white surface, whereas the CAD model is slightly gray.

Omron CJ2M and B&R X20 have the worst detailed CAD models, with missing text on the surface. The Omron module has the correct dimensions, but the CAD model lacks the ethernet connector. Additionally, the color of the CAD model is gray, whereas the actual object has different black levels with various reflections. The B&R module has the worst detail level. The orange coverings are missing entirely on the CAD model. In addition, the surface of the real module is partly see-through, whereas the CAD model has a solid surface.

The real images will also contain various connectors and cables plugged into the modules. These may occlude parts of the modules and are not available as CAD models. To still detect the module, other techniques will be used when generating the images, as described in the next section.

2.2 Synthetic Image Generation

The PLC modules are used to generate large annotated image datasets. Using CAD files instead of realistic 3D models brings some challenges. The models must be rendered in a scene, saved, and annotated with labels. These steps are impossible with CAD programs and require a separate rendering engine, as described in the following sections. An additional plugin will be required to convert CAD models to 3D models to be used in a rendering engine.

During rendering, a technique called domain randomization will be applied [3]. It randomizes different parameters to simulate varying environments. The idea is that the object detector detects the objects on a real image and sees it as another variation of an environment it has been trained on.

2.2.1 Rendering Engine

The purpose of CAD software is to create and edit CAD models and not to render images. A separate rendering engine is required to render the modules in different scenarios. The engine renders the objects with different positions, rotations, background objects, and other factors. It has knowledge of all objects in a scene and can generate images with perfect annotations. Different rendering engines could be chosen for this project. The most popular ones are Blender, Unreal Engine, and Unity.

Blender

Blender[1] is made for 3D modeling and rendering and is used in [5] and [27]. Its main purpose is to render large scenes and animations. However, it lacks support for CAD files, and they would have to be converted using standalone software. Additionally, there are no existing plugins for image generation and labeling. Creating different scenes would require many custom scripts, and every functionality must be manually implemented. An advantage of Blender is rendering photo-realistic scenes with high resolutions. As this project works with CAD files that are not realistic in the first place, this is not necessary.

Unreal Engine

Unreal Engine[2] is mainly a game engine and could be chosen for the work of this thesis. It has plugins to import CAD files [28] and various computer vision tasks, such as automatic annotations [29]. Its primary purpose is to

[1] https://www.blender.org/
[2] https://www.unrealengine.com/

render games with high frame rates, so it performs faster than Blender. This is especially noticeable as this work requires rendering thousands of different scenes.

Unity

Unity[3] is a game engine and the main competitor to Unreal Engine. Unity has support for plugins to import CAD files and computer vision tasks. For example, the Unity computer vision team developed the Perception Package[4], a framework for domain randomization to generate large-scale synthetic datasets. In Unity, the scene and the behavior of individual components are controlled using scripts. They can be written in different languages, such as C#, C++, and JavaScript.

Rendering Engine Selection

The ideal rendering engine was selected based on two main factors: Support for CAD files and the work required to generate synthetic datasets.

Blender was no option due to the missing support for CAD files and the lack of plugins for computer vision tasks. As mentioned above, some papers use Blender, but it requires many custom scripts to be implemented manually. And with the time frame of this thesis, it is easier to choose a rendering engine with an existing toolset to build on.

Unity and Unreal Engine have plugins to import CAD files with the same features. However, in the end, Unity was chosen as it has a clear advantage when considering the plugin to generate synthetic datasets.

The UnrealCV[5] plugin is the most popular computer vision plugin for Unreal Engine. But the last update for the plugin was in July 2019. Based on issues on the GitHub project page, there are many bugs, and most importantly, it lacks support for newer versions of the Unreal Engine [30]. Using an older version is possible, but it lacks many features and is not future-proof.

Unity is therefore used for this project in combination with the Perception Package. Its last update was in November 2022, with the first release of a preview version. The plugin has some bugs and is not perfect either, but it provides a framework to build on. It provides a toolset to generate and export annotated images. The scene layout to generate the images must be manually implemented using scripts. The plugin is written in C#, and the same language

[3] https://www.unity.com/
[4] https://github.com/Unity-Technologies/com.unity.perception
[5] https://github.com/unrealcv/unrealcv

is used for the scripts of this project, which makes it easier to adapt and extend the plugin.

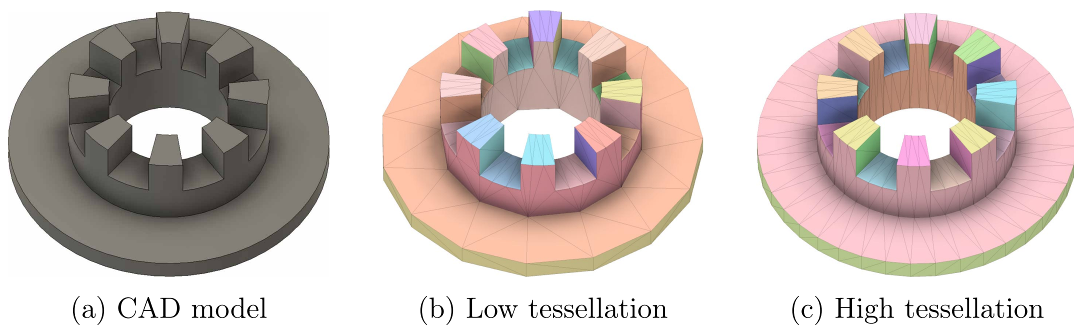
2.2.2 CAD Model Conversion

Game engines are made for 3D models and support different types of 3D files, but they have no built-in support for CAD files. CAD models are technical drawings with the correct dimensions and measurements of the real object. In contrast, in 3D computer graphics, 3D models are created with a process called polygonal modeling. The models are commonly only used in digital environments like animations or video games. Compared to CAD models, they mostly have detailed and, in some cases, even photo-realistic textures.

Because they are used in different environments, they use different technologies. CAD formats use mathematical functions to define the structure of surfaces. This allows for perfect accuracy of the surface, independently of the scale but requires more computation for rendering. In comparison, polygonal models consist of a mesh that consists of many connected polygons.

Tessellation

To convert the CAD models into polygonal meshes, a process called tessellation is used. It converts the CAD surface into a tesselated mesh composed of many small polygons [31].



(a) CAD model (b) Low tessellation (c) High tessellation

Figure 2.2: Different tessellation detail levels

The detail of the 3D model varies depending on the number of polygons, illustrated in Figure 2.2. It has to be chosen manually according to the object size and resolution of the rendered images. A lower tessellation level can lead to rough curves. In combination with light and reflections, an object can look very differently compared to using a higher detail level. However, this leads to worse performance and needs more time to render.

Pixyz Plugin

To import the CAD models into Unity, the Pixyz Plugin[6] is used. Different tools could be used, but none have the same level of integration into Unity. Pixyz supports all commonly used CAD file formats and a variety of import settings. For example, different quality levels for the tessellation process.

Due to the integration in Unity, it supports the import of surface colors. It imports the colors and converts them into Unity texture materials. However, they still consist of only one color, about the same as the one used in the CAD model. They could be edited manually to further reflect the actual object by changing the color or light reflection settings. But during the work of this thesis, the materials were not edited manually after import.

2.2.3 Image Generation

When using synthetic data to train an object detection algorithm, a so-called reality gap exists between the real and synthetic domains. This gap describes the visual differences between real and synthetic images. If this gap is too big, the object detector learns from synthetic data during training but then fails to detect objects on real images.

The authors in [3] explore a solution called domain randomization. It addresses the reality gap by exposing the object detector to different environments during training. The idea is by simulating various parameters, the variability of the generated data is so large that it includes the real world. If the object detector then sees a real image, it may think that it is just another variation.

The Perception Package for Unity is used as a framework to apply domain randomization. It is used to generate a synthetic dataset by randomizing various parameters.

Perception Package

The Perception Package[7] is a plugin developed by the Unity computer vision team to generate large-scale synthetic datasets. It provides a framework for domain randomization. At runtime, it uses a set of randomizers that are called for each frame. These randomizers are scripts that interact with objects in the scene. After calling all scripts, an image of the scene is taken and saved. The objects of all PLC modules are identified with a tag. The package uses this tag to identify and label them automatically. Thus, in the end, an image with an associated annotation file is saved for each frame.

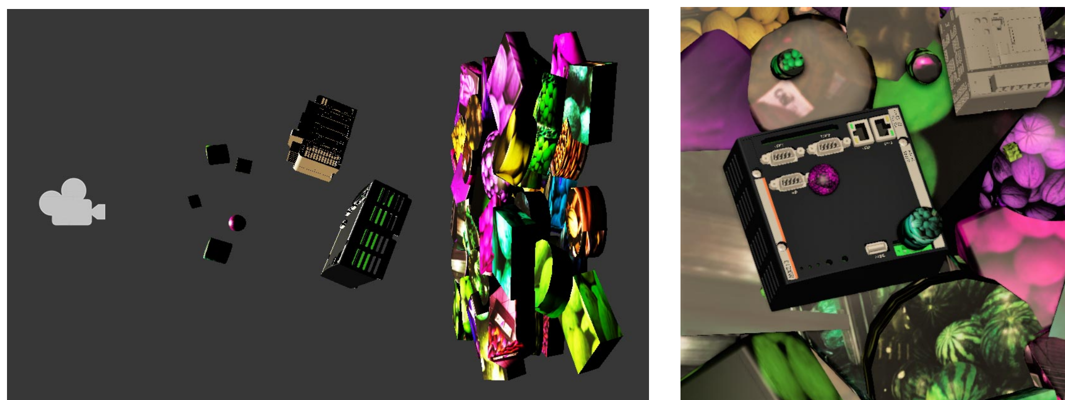
[6] https://www.pixyz-software.com/plugin/

[7] https://github.com/Unity-Technologies/com.unity.perception

In Unity, parameters used in a script can be configured in the Unity Editor without opening an external IDE to edit the script. This makes it easier to test and research different randomizer configurations.

The parameters can have different types depending on the use case, even complex ones. For example, some randomizable parameters are defined using a parametric definition. This allows the selection of a random value from a set of normal, uniform, or binomial distributions.

The package provides some basic randomizers[8] that were used and modified. More complex ones were implemented for this project.



(a) Side view of the Unity scene (b) Output image

Figure 2.3: Example frame and side view of the Unity scene

A scene is composed of different layers, shown in Figure 2.3a from right to left: Background Layer, Foreground Layer, and Occlusion Layer. The camera icon in the left part of the figure indicates the position of the camera. This camera takes and stores an image of the scene. Besides these layers, some postprocessing is applied to the entire scene.

The final image, displayed in Figure 2.3b, is saved with a separate metadata file. The metadata file includes the annotations for the individual image. This includes the class, position, and size of a bounding box for each module. Additionally, each layer is saved as an individual image with a transparent background. The individual layers will be used later when improving a synthetic image with a GAN.

[8] `https://docs.unity3d.com/Packages/com.unity.perception@1.0/manual/Randomization/RandomizerLibrary.html`

Background Layer

The background of an actual image is unknown and has to be simulated. The idea is to generate a unique random background for each image. The object detector should not learn any correlations between the modules and their background [2].

For this project, a set of 10 different shapes with 529 textures are used. They were imported from a Unity object detection sample project: SynthDet[9]. The shapes are different variations of cubes, cylinders, and spheres. Combined with the textures, they can create more than 5000 different objects.

The authors in [2] place background objects until every pixel is covered. For this project, their idea was adapted. First, the size of a background object is set relative to the size of the foreground objects (the PLC modules). Then, a random position is selected that is not within a minimum separation distance from other objects. This process is repeated until the surface is covered and no more objects can be added. Due to the minimum separation distance, it is possible to get empty spots between two objects. To prevent this, multiple layers of background objects are stacked in front of one another.

The size of each background object is relative to the median size of all foreground objects with an additional scaling factor. The authors of [2] researched the effect of the background object size on the final object detection results in more detail. They achieve the best results when scaling the background objects with a factor between 0.9 and 1.5 relative to the size of the foreground objects. These parameters are also used for this project, and each background object is scaled relatively with a randomized factor within the given range.

Foreground Layer

The foreground layer places the PLC modules that will be labeled. It selects several objects and places them in a random position. A minimum of 0 and a maximum of 5 objects can be shown simultaneously. As in the background layer, a minimum separation distance parameter can be set. Objects can overlap slightly but not be directly in front of each other.

After placing the objects, their scale and rotation are randomized. The rotation is uniformly randomized between -40 and 40 degrees for the x and y-axis and 0 to 360 degrees for the z-axis. The PLC modules will always be visible from the front as they are mounted with the back side onto a wall. In addition, many modules can not be identified from the side. Therefore, this range of degrees was selected for the viewing angle. It should cover all viewing

[9] `https://github.com/Unity-Technologies/SynthDet/tree/master/SynthDet/Assets`

angles that occur in an actual image. Figure 2.4 displays some example poses of a PLC module.

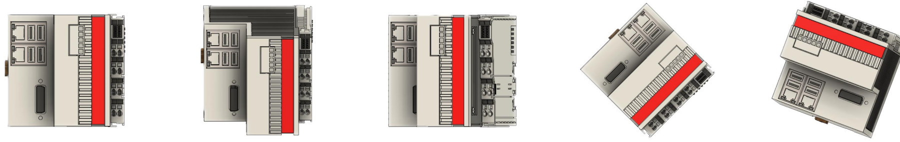
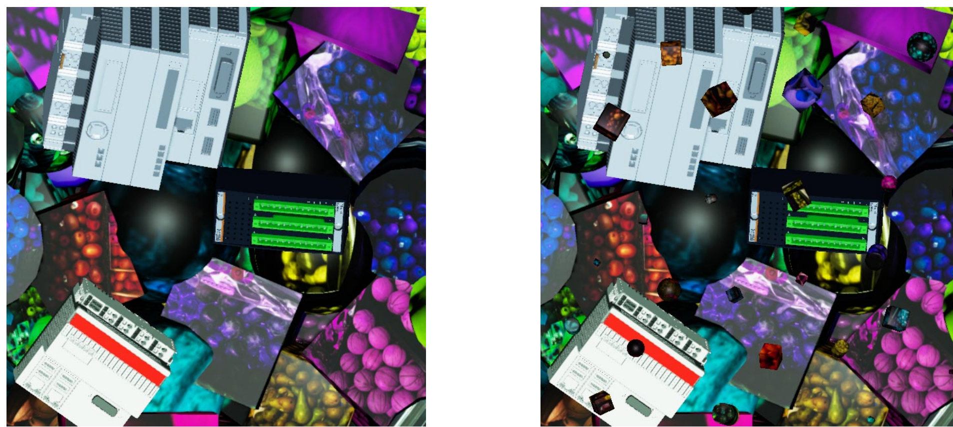


Figure 2.4: Example poses of a PLC module

Occlusion Layer

The purpose of the occlusion layer is to hide part of the modules during training. In an actual image, the occlusion layer is represented by cables plugged into a module or minor differences where the real module may look different. This layer uses the same shapes and textures as the background layer. They are placed at a random position with a random rotation.

The scale of the individual occluder objects is slightly randomized but much smaller than the foreground objects. Figure 2.5 displays example images with and without the occlusion layer.



(a) Background and foreground layer (b) All layers

Figure 2.5: Synthetic image with and without the occlusion layer

Postprocessing

After assembling the scene and its layers, some randomized postprocessing is applied to the entire scene: [2]

- A hue offset to simulate different lighting environments
- The red, green, blue, and alpha values of all colors are changed slightly. This simulates different surface colors, as the color of the CAD model may not correspond with the one on a real module.
- Changing the brightness to simulate darker and lighter environments
- Adding a blur effect

Figure 2.6 displays some generated example images with their bounding box annotations. The annotation is stored in a separate file and not drawn in the saved image.

Unity exports the dataset in the Synthetic Optimized Labeled Objects (SOLO) format [32]. This format is Unity specific and can not be used to train an object detection system. Therefore, a converter imports the SOLO format and exports the images and annotations in the Common Objects in Context (COCO) format [33]. Microsoft developed COCO, which is used in this project as an intermediate format. It will later be converted into the specific format required by the object detector.

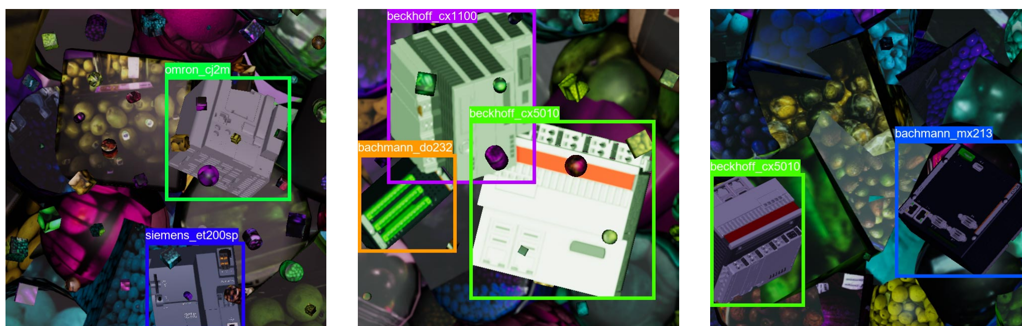


Figure 2.6: Example synthetic images

2.3 Image Enhancement With GANs

GANs are used for this project for the task of image-to-image translation. The idea is to train a GAN with real and synthetic images of the individual PLC modules. After training, it can translate images between these two domains. This process is especially useful in cases where the CAD model does not resemble the real PLC module. For example, a synthetic image of a model with an incorrect surface color can be translated to reflect a more realistic image.

For image translation tasks, there are two main methods:

- Paired image-to-image translation (e.g., Pix2Pix [7]): This process requires corresponding image pairs from both domains during training. However, pairing synthetic images with perfectly matching real images is challenging and almost impossible for this project. Real and synthetic images would have to be created where the PLC module is visible with precisely the same position, rotation, and size.

- Unpaired image-to-image translation: This process does not require corresponding image pairs. It only needs a set of images from different domains.

For this project, unpaired image-to-image translation is chosen using CycleGAN and StarGANv2. CycleGAN allows translation between two domains, and StarGANv2 between multiple domains. Experiments are conducted for both GANs, as described in the following sections. In the end, the trained GANs convert synthetic images in an image processing pipeline. This pipeline takes the synthetic images as input, translates the PLC modules with a GAN, and outputs a more realistic image.

For CycleGAN, the official implementation[10] from the CycleGAN paper is used, for StarGAN the official tensorflow implementation[11].

[10] `https://github.com/junyanz/pytorch-CycleGAN-and-pix2pix`
[11] `https://github.com/clovaai/stargan-v2-tensorflow`

2.3.1 Experiments

The following experiments aim to test the capabilities and behavior of the GANs without the full complexity of a generated synthetic image. Initial experiments were conducted with multiple different PLC modules visible in one image. Both GANs generated output images without any clearly recognizable objects. Sometimes they generate contours of an object, but the texture of the generated object is not identifiable with any PLC module. Therefore, the GANs can not be used to translate a synthetic image directly, and the images are simplified so that only one module is visible per image.

The following paragraphs describe the findings related to the training data. These apply to StarGAN and CycleGAN, and both GANs use the same images during training. In the end, the specific experiments for the training of both GANs are mentioned.

Training Data

To begin with, initial tests showed that synthetic images could not contain the randomized background and occlusion layer. Otherwise, both GANs generate unusable images with objects either not visible at all or without any clear edges. The reason is most likely that the GANs get distracted by the randomized background that differs for each image. The same applies to the occlusion layer, which prevents them from learning the texture of an object, as occluder objects randomly cover some parts.

To prevent this issue, synthetic images are generated without the background and occlusion layer. The same applies to real images, and they are taken on a single-colored background (e.g., a table) without cables or other distracting objects. Figure 2.7 displays example training images for some modules.

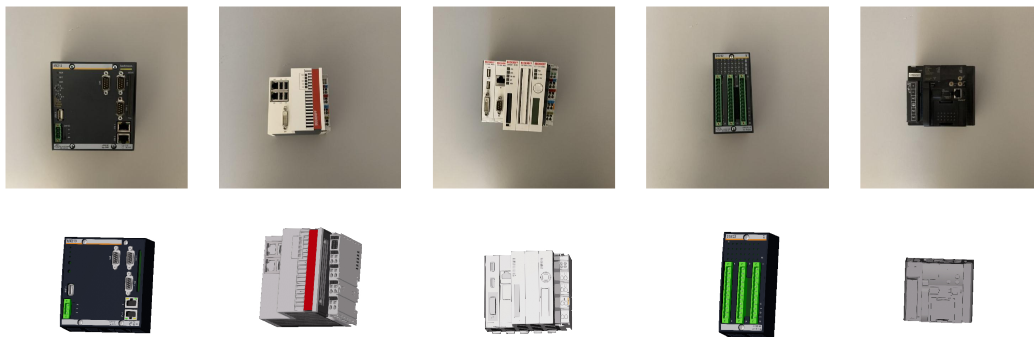


Figure 2.7: Example GAN training images

Further experiments showed that the PLC modules displayed in the images must have approximately the same pose in both datasets. Otherwise, the GAN learns a bias towards a specific characteristic more present in one dataset. To control these parameters, the dataset must be created considering the following rules for the position, rotation, and size of an object in an image.

Position: For each image, exactly one object must be visible, and it has to be positioned roughly in the center of the image. This can easily be controlled when capturing real images and generating synthetic images.

Rotation: The PLC modules must be horizontal in the image, with the bottom of the object pointing to the bottom of the image. Images should be taken from slightly different angles, but the modules can not rotate around the z-axis. For the synthetic dataset, this is controlled by reducing the possible ranges for the z-axis to [-5; 5] degrees. The rotations around the other axes are still randomized with the same ranges as during the generation of the synthetic dataset. This allows for the same object poses as during the full synthetic data generation, except for the z-axis.

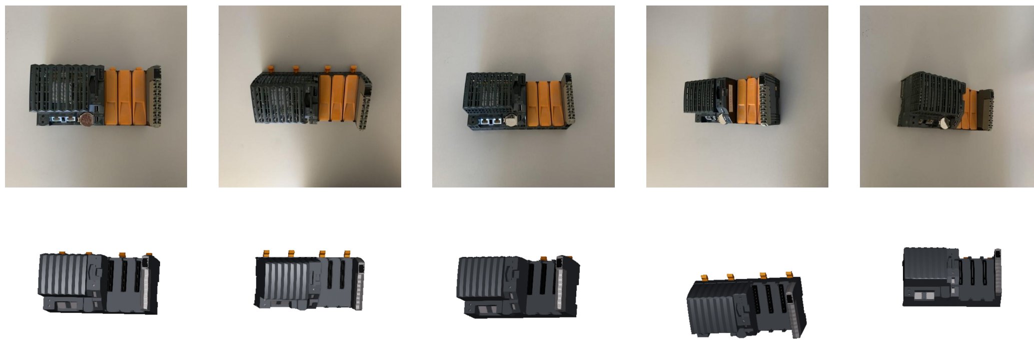


Figure 2.8: Example GAN training images for B&R X20

Figure 2.8 displays example images for one module. The object is placed in the center, but the images are shot from different viewing angles. Both GANs generate images with objects roughly in the same pose as on the input image. In an ideal scenario, both datasets display the module overall in the same set of poses. Due to the small dataset size of 73 real and 73 synthetic images per PLC module, this is not perfectly achieved in this project. Using a larger dataset could improve the correspondence between the datasets and, consequently, the translation. It may generate better images but most likely will not affect the overall object detection performance. And as the idea behind this project is to use as few real images as possible, the dataset size was not increased.

Size: Controlling the size is the most challenging task as the objects are easily a bit larger in one dataset. The GANs then start to generate slightly larger objects in one domain. This leads to issues when replacing the object on the synthetic image. When the scaling is slightly off, the original bounding boxes are no longer correct, leading to problems during object detection.

However, having the exact same object size is nearly impossible to control for both training datasets. This will be addressed in the image processing pipeline before transferring the object back onto the original foreground image.

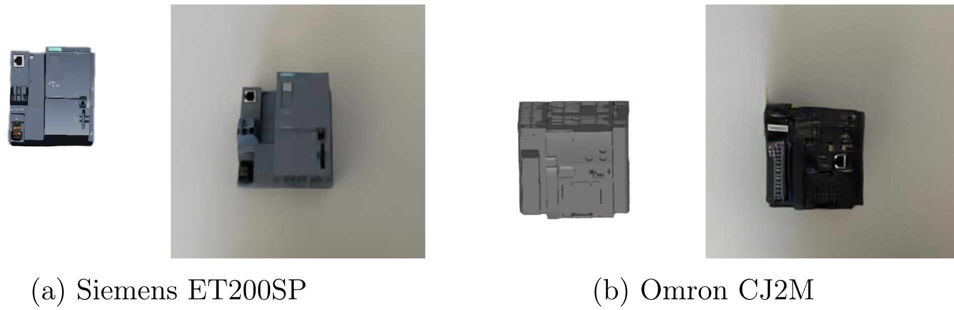
Two datasets are created for each PLC module: One with real images and another with synthetic images. A dataset contains 73 images, split into 60 training and 13 test images. The default resolution of both GANs is used for the image size, which is 512x512 pixels. The following experiments for CycleGAN and StarGAN are performed on an NVIDIA RTX A4500 with 20GB of GPU memory.

CycleGAN Experiments

Besides the experiments described above, an additional experiment was conducted for CycleGAN. The idea is to test whether one CycleGAN can be trained for all PLC modules using one domain for all synthetic images and the second domain for real images. The theory behind this experiment is that all modules have in common that they are either real images or synthetic images generated from CAD models.

However, this experiment failed as CycleGAN generated images with objects that are visually a combination of all modules. It is not able to learn a mapping between the individual modules. The underlying problem is that the difference between the CAD and real objects varies for each module, especially between manufacturers. For example, the surface color of the real and CAD objects matches on some modules but is entirely different on others.

The final solution for CycleGAN is to train a separate GAN for each PLC module with one synthetic and one real domain of the specific module. Figure 2.9 displays example translations for the Siemens ET200SP and Omron CJ2M modules. Each figure displays the input image on the left and the output image from the GAN on the right.

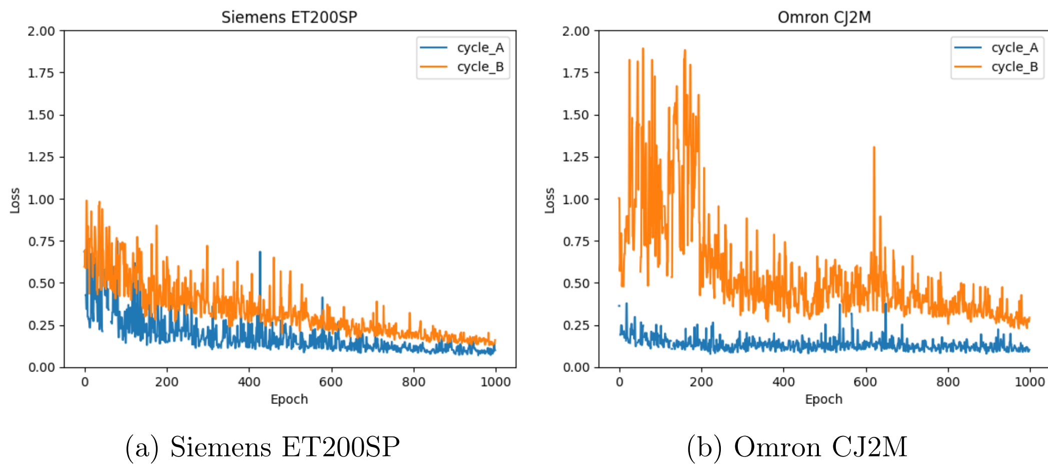


(a) Siemens ET200SP (b) Omron CJ2M

Figure 2.9: CycleGAN example translations

Figure 2.10 displays the graphs of the cycle-consistency losses for the two PLC modules. *Cycle_a* is the loss calculated for the translation between synthetic-real-synthetic, and *cycle_b* is the loss calculated for real-synthetic-real. Both *cycle_a* losses indicate that CycleGAN overall easier translates real images to synthetic ones, which is unfortunately not required for this project.

When looking at *cycle_b* for both graphs, the model manages to translate the Siemens ET200SP module faster to a better quality. The Siemens module has a realistic CAD model, whereas the CAD of the Omron module is less detailed.



(a) Siemens ET200SP (b) Omron CJ2M

Figure 2.10: Comparison between cycle-consistency losses of CycleGAN

As an individual GAN has to be trained for each module, the GANs were trained to different epochs. Modules with a lower CAD quality were trained longer than others. Table 2.1 displays the final training time and cycle-consistency loss for all GANs. Only the *cycle_b* loss is displayed as the *cycle_a* does not change much and is not that relevant for this project.

There is no clear indicator of when CycleGAN training is finished. For this project, the training was stopped when the *cycle_b* loss reached about 0.2. With the visual inspection of the results, this seemed to be a suitable metric. Especially as the improvements for the cycle-consistency loss slowed down, and minimal improvements would require significantly more training.

Overall, the more realistic CAD modules took about 1000 epochs, whereas others were trained for 2000 epochs. Combining the training times for all GANs requires about 40 hours of training.

PLC Module	**Epochs**	**Training Time**	**Cycle_B Loss**
Bachmann MX213	1000	4h	0.20
Bachmann DO232	1000	4h	0.20
Siemens ET200SP	1000	4h	0.16
Beckhoff CX5010	1000	4h	0.19
Beckhoff CX1100	2000	8h	0.19
Omron CJ2M	2000	8h	0.18
B&R X20	2000	8h	0.22

Table 2.1: CycleGAN training times

StarGAN Experiments

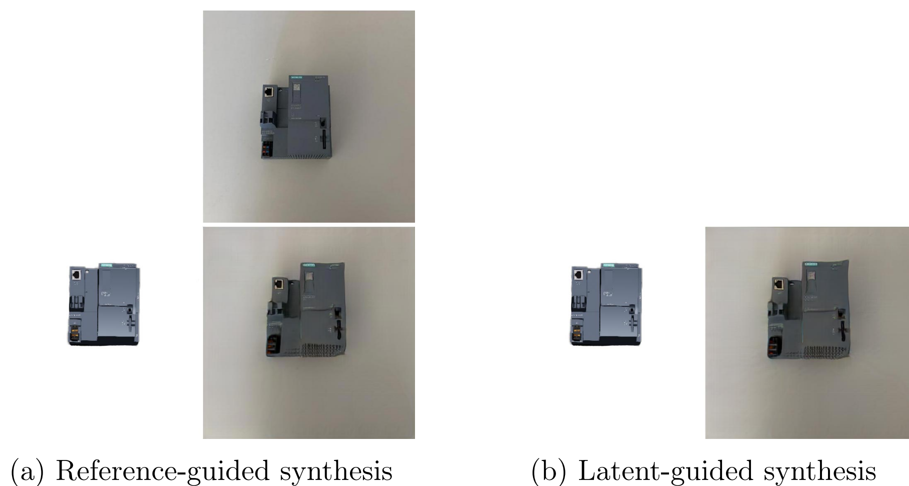
StarGAN uses the same training data as CycleGAN. But it supports multiple domains, so only one GAN is trained for all modules. Each module gets an individual real and synthetic domain. For the 7 modules, this results in 14 domains which is quite a large and complex GAN to be trained. This is no issue for this project but could lead to issues when using many more modules.

StarGANv2 supports two modes for the task of image translation: latent-guided synthesis and reference-guided synthesis [10]. Latent-guided synthesis translates source images with randomly generated sample latent codes to a target domain. Reference-guided synthesis requires at least one additional reference image. It then translates the source image to the target domain and reflects the style of the given reference image.

The authors of StarGANv2 evaluated both methods, and reference-guided synthesis scores a lower Frechet Inception Distance (FID) value [10]. FID is an indicator that calculates the difference between the real and generated images [34]. Lower scores mostly correlate with a higher-quality image translation, although this metric does not always correspond to the visual perception of the human eye.

Both methods are tested for this project, and no visual difference is visible to the human eye, although no exact FID values are calculated. Both methods translate images where the object on the output image has roughly the same pose but a more realistic surface texture.

In the end, reference-guided synthesis is used due to the better performance observed by the authors of StarGANv2. A real image from the training dataset is used during the translation for the reference image.



(a) Reference-guided synthesis (b) Latent-guided synthesis

Figure 2.11: StarGAN image synthesis results

Figure 2.11 displays an example of reference-guided and latent-guided synthesis. Figure 2.11a displays the reference-guided synthesis with a reference image from the training dataset on the top, the input image on the left, and the output (translated) image on the bottom right. Figure 2.11b displays the latent-guided synthesis with the input image on the left and the output on the right.

StarGAN allows the translation from one domain to all other domains. Theoretically, a synthetic image of one PLC module could be translated to a real image of another PLC module. For example, a synthetic image of a Bachmann DO232 can be translated to a real image of a Siemens ET200SP. This was tested, and the GAN produced good images for the other domains. However,

the images are visually better when translating from the synthetic to the real domain of the same module. Therefore, the GAN is only used to translate between domains of the same module for this project.

The used StarGANv2 implementation automatically exports images for all domains. An additional option was implemented to specify the target domain and only export the image for that domain. This significantly improved the performance of the GAN as thousands of images were converted for this project.

StarGAN does have training losses, but they are not comparable with CycleGAN due to the different architectures. The training was stopped after 3 days at 75000 epochs when visually acceptable results were achieved for all modules, and the improvements during each epoch slowed down.

2.3.2 Implementation

This section covers the implementation of the image processing pipeline that translates the synthetic images with a GAN. CycleGAN and StarGAN are implemented for comparison, and one is selected for the individual tests. For the GANs, the trained models from the previous section are used.

Image Processing Pipeline

In the generated synthetic dataset, most images contain multiple PLC modules of different types. However, a GAN can only convert entire images from one domain to another.

To solve this issue, a pipeline was developed to convert images with multiple different modules. In the beginning, it extracts the objects from the synthetic image. They are then individually translated with a GAN, and, in the end, pasted back onto the synthetic image. This process is the same for CycleGAN and StarGAN, except for the GAN used to translate the individual images.

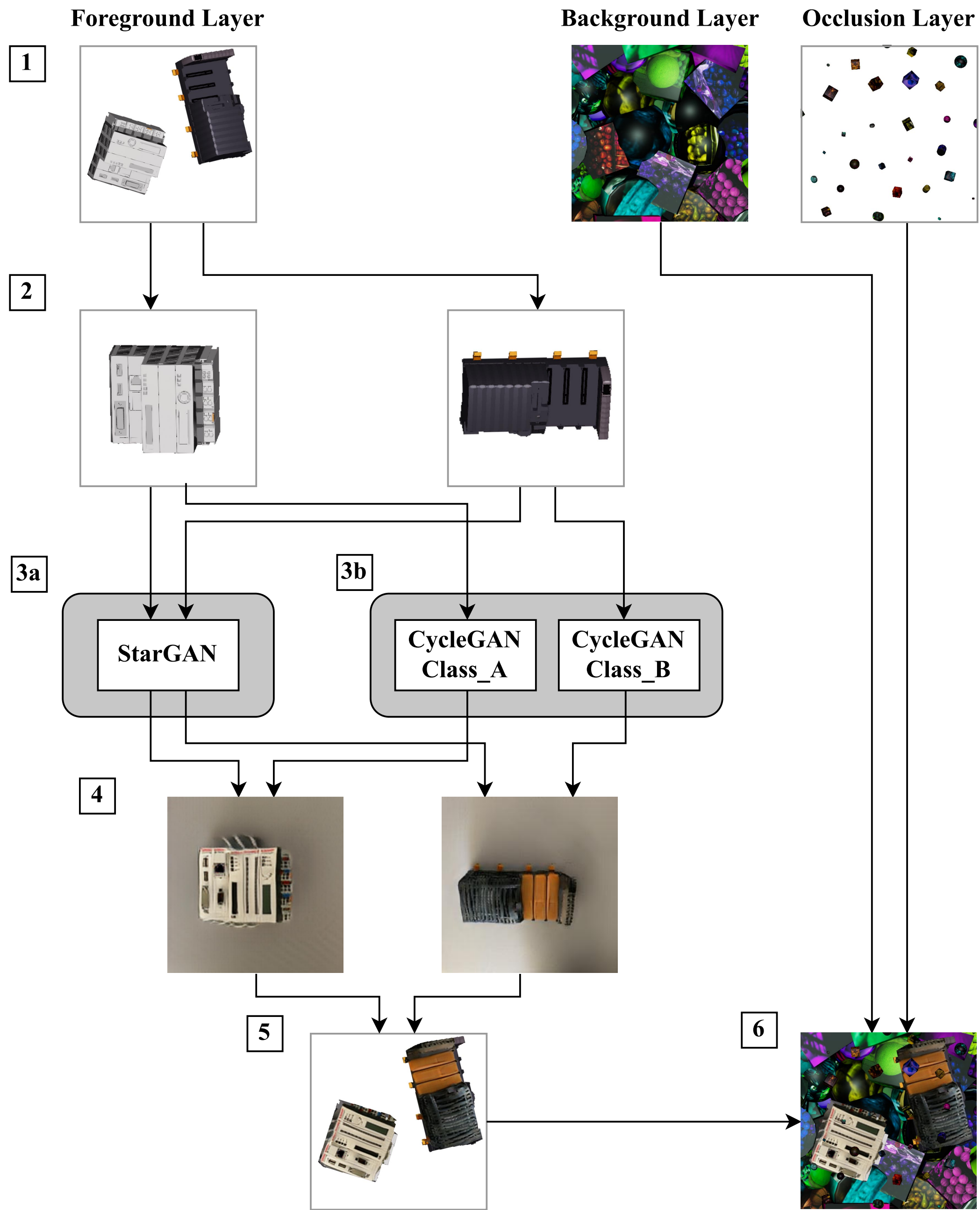


Figure 2.12: GAN image processing pipeline

Figure 2.12 illustrates the process of translating the entire foreground image. Depending on the use case, this process is repeated for every image in the generated synthetic dataset or only for a part of the dataset.

1. The process starts with the image of the foreground layer. It contains the PLC modules without any background and occluders.

2. Images of individual modules are extracted from the foreground image.

   During the generation of the synthetic dataset, a separate metadata file is stored for each image. It includes the bounding box, class, and rotation around the z-axis for each object. The bounding box and rotation data are used to cut out the object and revert the rotation during this step. Only the rotation around the z-axis is reverted as the GAN was trained with the object being horizontal in the image but with slightly different viewing angles. In the end, a separate image is exported for each module. This includes the class of the module.

3. The images are translated individually with StarGAN (3a) or CycleGAN (3b). The class of the specific module defines the current and target domains.

   a) For StarGAN, one GAN is used to translate images for all domains. The real domain of the current module is chosen for the target domain.

   b) For CycleGAN, an individual GAN was trained for each PLC module. The correct GAN is selected based on the type of the module and then used to translate the image.

4. The GANs output the fake real images. These images display the objects in slightly different positions, rotations, and sizes, as already found during previous experiments.

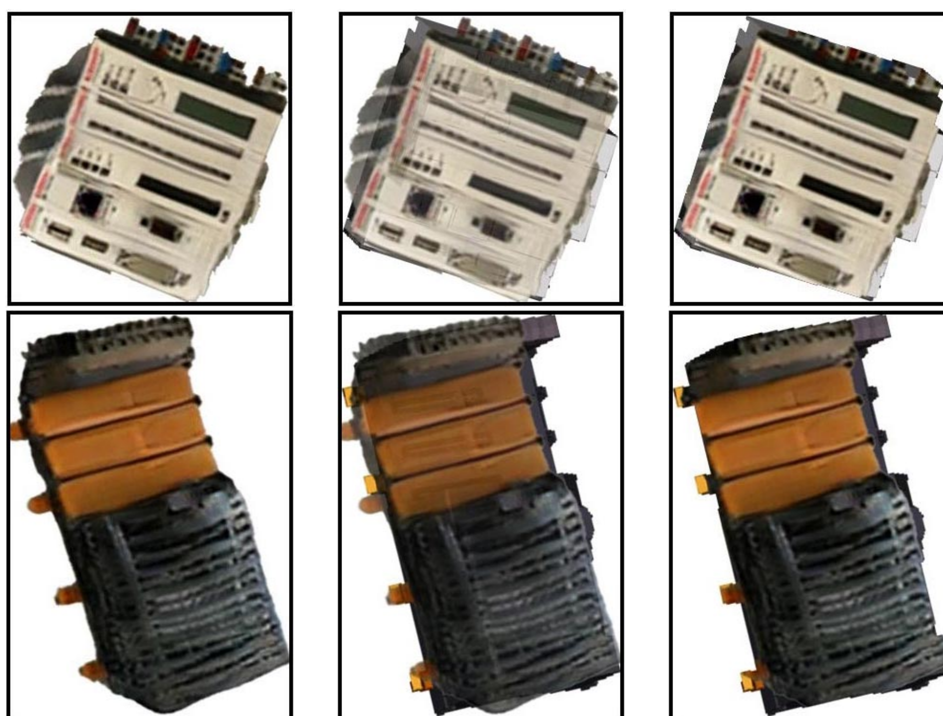
5. The objects are then pasted back onto the original image of the foreground layer. The background is removed during this process, and the objects are rotated and resized to their original position. A complex process is required to ensure that the bounding boxes are still perfect, described in detail later.

6. The last step combines the individual layers into one image. The foreground layer is layered onto the background layer with the occlusion layer on top of it. The resulting image is then used to train the object detector.

Transferring the GAN Output Image

The experiments for both GANs showed that it is impossible to translate an image while keeping the exact position, rotation, and size of an object. The objects on a translated image always have slightly different dimensions. As the GAN cannot output any bounding boxes, it is difficult to determine the exact position and scale of an object in the image. Additionally, the object is often distorted and does not have clear edges or contours.

This process aims to extract the module from the GAN output image, copy it onto the synthetic image of the foreground layer, and use the existing annotations and bounding boxes. Three options are researched to implement this process and ensure the original bounding boxes are still correct for each pixel. Figure 2.13 displays examples of these options, and they are described in detail in the following paragraphs. Removing the background and cutting out the module from the GAN output image is necessary for all options. For this process, the Rembg tool[12] is used.



(a) Option 1: GAN objects only (b) Option 2: GAN and synthetic objects (c) Option 3: GAN objects as texture

Figure 2.13: Options to transfer the objects from the GAN output image

[12] `https://github.com/danielgatis/rembg`

Option 1: Replace the objects on the synthetic image with those from the GAN output image.

First of all, the background is removed. Each object is then rotated to the same rotation as in the synthetic image and scaled to fit the original bounding boxes. The new object is placed in the same position and replaces the object in the original synthetic image.

This option is not ideal, as the GAN sometimes outputs distorted objects without clear edges. An example is displayed in Figure 2.13a, where the object is distorted, and the tool to remove the background fails to clearly distinguish the object from the background. If the tool cuts away too little of the object, the bounding box includes some background pixels. If it cuts away too much, the object may get a different shape.

The different shapes will lead to issues later during object detection. The object detector does not exactly know where to place the bounding box as it varies depending on the image.

Option 2: Layer the GAN output objects on top of the original objects with adjustable opacity.

The idea is to use Option 1 but keep the synthetic object in the background and layer the new image on top of the synthetic object with a customizable opacity. Theoretically, the opacity could be randomized, and the object detector could then learn different variations of the surface colors.

The issue with this option is that the object on the output image does not perfectly align with the object on the input image, as shown in Figure 2.13b. Additionally, it has the same issues as Option 1 regarding the different sizes of the cut foreground object. This idea is still mentioned as it theoretically could be a solution if the GAN outputs an image with exactly the same object pose as on the input and if it is clearly distinguishable from the background.

Option 3: Using the GAN output objects as a texture for the original objects.

The idea for the third option is to use the output image from the GAN as a texture that is put on top of the synthetic objects. Compared to Option 2, the difference is that the contour and surface area of the synthetic objects is kept the same.

For this process, each object is cut out, rotated, and resized to fit the original bounding box on the synthetic image. The original synthetic object is kept, and all non-transparent pixels from the synthetic object are then replaced with corresponding pixels from the GAN object. An example is displayed in Figure 2.13c. This option works in favor of domain randomization. The texture of

the 3D model is changed partially, but the contours of the object are always the same. As this process is repeated for every image, the surface is constantly generated slightly differently. Over the entire dataset, the whole surface of an object is covered at least once. This favors domain randomization as the object detector detects the real object in an image as it may think the surface of the specific object is just another variation.

Option 3 is selected for this project as it should achieve the best results. It is the only option with always correct contours and bounding boxes. The surface texture changes slightly for every object, which favors domain randomization as this process is repeated for thousands of images.

2.4 Object Detection

The object detection task aims to predict the position and type of one or more objects in an image. It consists of two main objectives: localization and classification. The localization predicts the position and size of an object in the image, typically by drawing a bounding box around the object. Classification predicts the type of object in the form of a class label.

For this project, YOLO and Faster R-CNN are implemented for comparison, as described in the following sections. Both use the same training datasets, consisting of 15000 images with different variations of pure synthetic images and images refined by a GAN. Of these 15000 images, 80% of images are used for training, 10% for testing, and 10% for validation. The synthetic images are generated with a resolution of 640x640 pixels. The same resolution is used to train, test, and validate both object detectors.

A validation dataset of real images is created to evaluate the object detectors, as described in the last section.

2.4.1 YOLO

For YOLO, the newest version YOLOv8 is used from the Ultralytics GitHub repository[13].

A pre-trained YOLOv8 model[14] is selected for training. Ultralytics provides five different pre-trained models with different sizes: YOLOv8n (nano), YOLOv8s (small), YOLOv8m (medium), YOLOv8l (large), and YOLOv8x (extra large). All models are pre-trained on the COCO val2017 dataset[15]. YOLOv8n is the smallest model with the fewest parameters (weights and biases), and according to Ultralytics, it is the fastest but least accurate. YOLOv8x is the largest and most accurate but, at the same time, the slowest model. The YOLOv8n, YOLOv8m, and YOLOv8l models are used for this project.

The following hyperparameters were changed during training:

- Disabled image scaling, rotation, and mirroring in all directions. During the generation of synthetic images, the objects are already scaled and rotated in all desired directions. It is easier to generate images differently in the first place than during object detection.

[13] `https://github.com/ultralytics/ultralytics`
[14] `https://github.com/ultralytics/ultralytics/#Models`
[15] `https://cocodataset.org/`

- Disabled mosaic augmentation. This new feature in YOLOv8 combines multiple images into a single mosaic image. It forces the model to learn objects with different neighboring pixels in different locations and helps it see some variations of the training images. Again, these parameters are already randomized during the data generation and are therefore deactivated here.

Disabling the data augmentation features ensures that YOLO and Faster R-CNN are trained with the same images for a fair comparison. For example, Faster R-CNN has no mosaic augmentation feature. Tests also showed that changing these parameters slightly improved the detection performance for YOLO. For the other arguments and hyperparameters, the values are left default[16]. During training, the batch size is set to 16 for the optimal utilization of the GPU.

2.4.2 Faster R-CNN

For Faster R-CNN, the implementation from the Detectron2 library [35] is used. This implementation supports different backbones and pre-trained models. These are listed in the Model Zoo collection[17]. For this project, the FPN and C4 backbones are tested. FPN uses a ResNET+ FPN backbone which should obtain the best tradeoff between speed and accuracy. C4 is the original backbone described in the Faster R-CNN paper and uses a ResNet conv4 backbone with a conv5 head.

The parameters during training and testing are mostly left default[18]. As for YOLO, the hyperparameters to flip and crop the image randomly are deactivated. The images are already flipped, cropped, and rotated during the dataset generation with domain randomization.

[16] `https://docs.ultralytics.com/modes/train/#arguments`
[17] `https://github.com/facebookresearch/detectron2/blob/main/MODEL_ZOO.md`
[18] `https://github.com/facebookresearch/detectron2/blob/main/detectron2/config/defaults.py`

2.4.3 Real Validation Dataset

A real validation dataset was created to evaluate the performance of YOLO and Faster R-CNN. This dataset is not used during the training of the object detectors and only for validation after training is finished.

The goal of this dataset is to represent different real-world scenarios. Therefore, the following parameters are constantly changed for each image. Figure 2.14 displays some example images from this dataset.

- Combinations of different modules with different positions in the image
- Images are shot from different angles, closer and further away from the modules
- Backgrounds that are entirely different from the ones used during the GAN training
- Lighting conditions, e.g., well illuminated but also with shadows from the side
- Cables plugged into the modules and in front of the modules

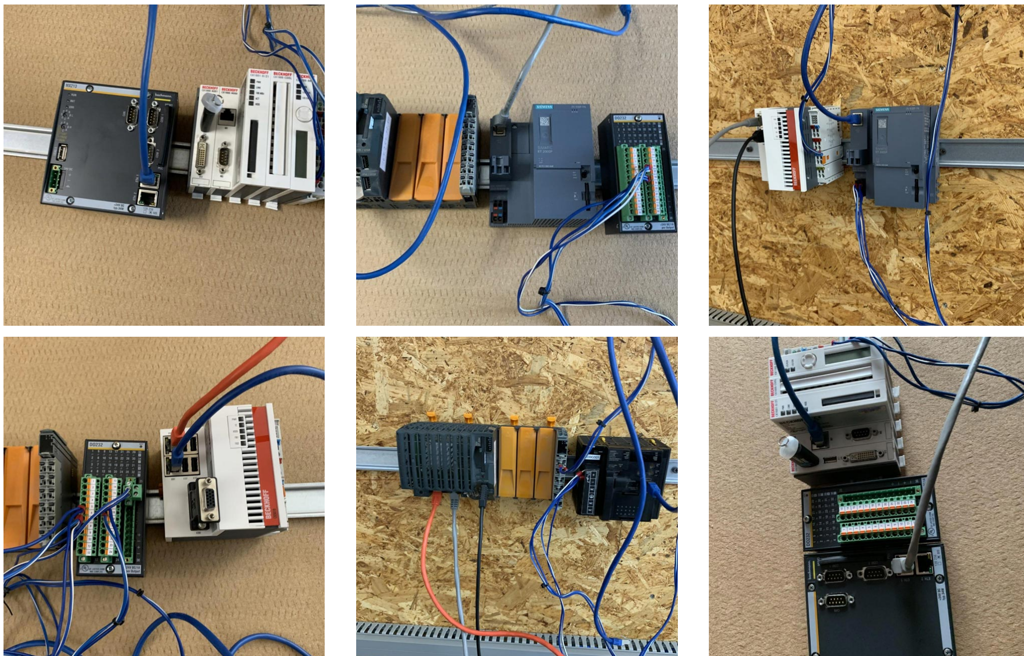


Figure 2.14: Example images from the real validation dataset

The dataset consists of 1345 manually labeled images. Table 2.2 lists the overall instances of the individual modules. A minimum of 0 and a maximum of 5 modules are displayed per image.

PLC Module	**Instances**
Bachmann MX213	351
Bachmann DO232	296
Siemens ET200SP	303
Beckhoff CX5010	383
Beckhoff CX1100	277
Omron CJ2M	324
B&R X20	304

Table 2.2: Distribution of instances in the real validation dataset

3 Results

Different combinations of GANs and object detectors are tested to evaluate the performance. The next section provides an overview of all results. The following sections explain the results for the GANs and object detectors individually in more detail.

The models for all tests are trained on the same synthetic dataset of 15000 images, split into sets of 80% training, 10% testing, and 10% validation. Generating these 15000 synthetic images and converting them to the specific YOLO or Faster R-CNN formats takes about 3 hours.

If a GAN is used, a percentage of this dataset is translated with the corresponding GAN. Translating all 15000 images takes about 6 hours, the same for CycleGAN and StarGAN. However, this implementation is not performance optimized and could be further improved.

The trained models are then evaluated on the real dataset from the previous chapter. All tests are performed on an NVIDIA RTX A4500 with 20GB of GPU memory.

3.1 Overview of Results

Figure 3.1 displays example images with the predicted bounding boxes using a YOLOv8m model, trained for 100 epochs with 50% of the synthetic images translated with StarGAN.

The runs are evaluated with the mean Average Precision (mAP). It is calculated as the mean average precision over all classes. The Average Precision (AP) is calculated using precision values across different recall levels for one class. It computes the area under the precision-recall curve, representing precision across various recall levels. The recall measures the true positive rate, the rate of correctly identified objects.

To define if a predicted region is considered true or false positive, the Intersection over Union (IoU) is used. It is calculated by taking the overlap between the ground truth and the predicted bounding box and dividing them by the union of these boxes.

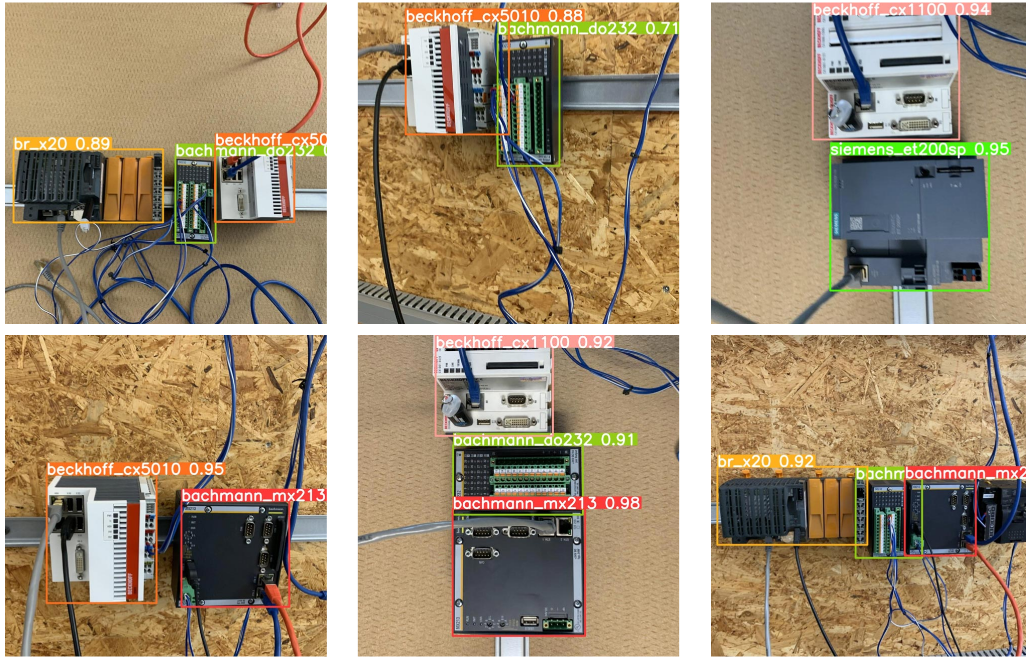


Figure 3.1: Example YOLO object detection images

The mAP^{50} and mAP^{50-95} are used for evaluation. mAP^{50} is the mean average precision with an IoU of 0.5. This is the most important metric for this project, as the goal is to detect the modules, but the bounding boxes do not have to match exactly. The mAP^{50-95} is displayed as an additional metric. It is defined by the average mAP over different IoUs thresholds between 0.5 and 0.95 with steps of 0.05.

	YOLOv8m		**Faster R-CNN**	
Training Data	**mAP^{50}**	**mAP^{50-95}**	**mAP^{50}**	**mAP^{50-95}**
synthetic	55.7	45.1	43.2	18.4
25% StarGAN	83.8	69.5	64.6	31.0
50% StarGAN	84.4	68.1	77.6	40.3
75% StarGAN	83.1	66.6	72.3	32.6
100% StarGAN	64.8	47.7	80.1	40.0
25% CycleGAN	67.3	51.6	54.2	24.7
50% CycleGAN	66.0	52.2	60.0	28.4
75% CycleGAN	62.9	47.6	56.9	25.8
100% CycleGAN	36.8	26.6	48.8	22.0

Table 3.1: Overview of the YOLO and Faster R-CNN results

Table 3.1 lists an overview of the object detection results evaluated on the real evaluation dataset. StarGAN performs overall better than CycleGAN, but CycleGAN is better than using a pure synthetic dataset.

Synthetic data is still essential as translating a higher percentage of a dataset with a GAN decreased the performance. StarGAN performed roughly the same at 25%, 50%, and 75%, but the mAP gets worse when translating all images. This is also reflected in CycleGAN, where the best results are at 25% and 50%. The mAP decreases when translating more than 75%.

The R50-FPN-3x backbone is used for Faster R-CNN, and the YOLOv8m model for YOLO as they achieve the best results. When comparing YOLO with Faster R-CNN, YOLO achieves higher mAP values in almost all categories. This will be described later in more detail.

	YOLOv8m - mAP^{50}							
Training Data	**All Classes**	**Bachmann MX213**	**Bachmann DO232**	**Siemens ET200SP**	**Beckhoff CX5010**	**Beckhoff CX1100**	**Omron CJ2M**	**B&R X20**
synthetic	55.7	67.1	54.3	70.8	94.2	54.9	40.1	8.2
25% StarGAN	83.8	88.9	87.1	90.5	96.6	86.7	84.5	52.4
50% StarGAN	84.4	91.3	87.0	92.9	96.6	82.3	83.5	57.1
25% CycleGAN	67.3	84.9	83.6	74.6	91.9	57.6	37.0	41.2
50% CycleGAN	66.0	74.9	76.9	72.7	93.9	85.9	12.3	45.4

Table 3.2: Detailed mAP^{50} results for all PLC modules

Table 3.2 lists the detailed class metrics for all modules. The YOLOv8m model was trained for 100 epochs with different datasets and then evaluated on the real dataset.

All modules have a higher mAP when adding a GAN, but the best improvement is seen with low-quality CAD models. The low-quality ones are listed as the columns towards the right in Table 3.2. The B&R X20 module has a grey surface on the CAD model but is black and orange in reality. Adding a GAN improved the detection rate from a mAP^{50} of 8.2% with pure synthetic data to 57.1% when translating 50% with StarGAN. Another module with poor CAD quality is the Omron CJ2M. Using only synthetic data, the mAP^{50} is 40.1% but improved to 84.5% with 25% StarGAN.

An example of a minor improvement is the Beckhoff CX5010. It already

reaches a mAP^{50} of 94.2% using pure synthetic data. Adding 25% StarGAN improves the mAP^{50} to 96.6%. The reason is most likely due to the unique red stripe on the front surface of the module that is present in CAD and reality.

Additional tests were conducted regarding the size of the dataset. Using less than 15000 images decreased the object detection performance, whereas more images did not improve it either. However, this heavily depends on the number of objects and domain randomization parameters. For example, if the objects should be detected from all sides, including the rear side, more images will be necessary to show all different pose variations.

3.2 GANs

This section aims to evaluate the effect and purpose of StarGAN and CycleGAN. To research them, a comparison with a baseline of real images is required. Due to the limited amount of real data, the real dataset created for the evaluation is split for the following tests. It includes about 290 images where the modules are on the wall but without any cables in front of them. These images are extracted into a separate smaller real dataset, used for the training in Table 3.3. The rest of the dataset, about 1050 images, is used for evaluation.

All tests in Table 3.3 are conducted with a YOLOv8m model. The results are overall slightly worse compared to previous tests where the same tests were conducted but with the entire real evaluation dataset. This is due to YOLO better detecting objects with no cables in front of them. But as these images are used during training with the smaller real dataset, they are missing in the evaluation dataset, which results in slightly worse results.

	YOLOv8m			
Training Data	**Epochs**	**Training Time**	**mAP^{50}**	**mAP$^{50-95}$**
synthetic	100	12h	50.9	41.6
real	50	1.5h	50.2	34.1
synthetic + real	100 + 100	12h + 3.5h	72.0	54.5
50% StarGAN	100	12h	82.3	64.4
50% CycleGAN	100	12h	62.7	47.7

Table 3.3: Results evaluated on a smaller real dataset, trained with and without the use of GANs

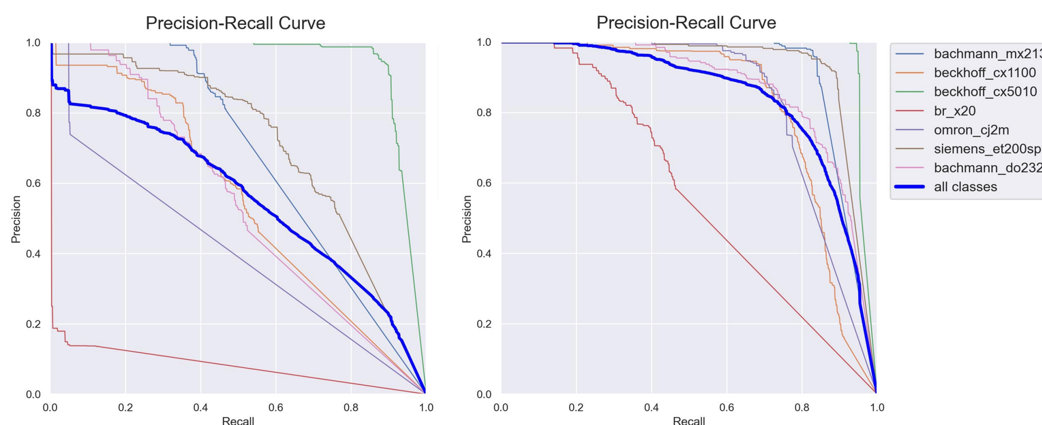
Training YOLOv8m directly with the small real dataset reaches a mAP^{50} of 50.2%. This is roughly the same as when using pure synthetic data. The training was stopped at 50 epochs because the model no longer improved.

For the second test, a model is trained on pure synthetic data and fine-tuned for 100 more epochs on the small real dataset (synthetic + real). It then reaches a mAP^{50} of 72%, indicating that the individual synthetic and real models detect partially different images.

Overall, translating 50% of the images with StarGAN still performs better than fine-tuning a synthetic model with real images. CycleGAN, on the other hand, performs worse. In a direct comparison between StarGAN and CycleGAN, StarGAN achieves higher mAP values overall.

The training times listed in Table 3.3 are only for YOLO and do not include the time to train the GANs. The training of StarGAN took about 3 days, and all CycleGAN models together took a bit less than 2 days.

CycleGAN performs worse but has an advantage when adding a domain after finishing the GAN training, as only one GAN needs to be trained and added. Additionally, each GAN can be trained individually to the point it translates images well. In contrast, the entire GAN must be retrained for StarGAN, which requires more training time.



(a) Pure synthetic dataset

(b) Synthetic dataset with 50% StarGAN

Figure 3.2: Precision-recall curves for YOLOv8m with and without GAN

Figure 3.2 displays the Precision-Recall (PR) curves to visualize the purpose of StarGAN compared to the pure synthetic dataset. Both examples were trained for 100 epochs on the YOLOv8m model and evaluated on the full real validation dataset.

A PR curve visualizes the relationship and the trade-off between precision and recall. The precision metric shows the accuracy and indicates the number of true positive predictions out of all positive predictions. The recall focuses on finding all positive examples and measures the proportion of true positive predictions out of all positive examples.

Generally, a model with a better, and therefore higher, PR curve is closer to the upper-right corner. Figure 3.2b clearly shows the improvement when using StarGAN compared to the pure synthetic dataset in Figure 3.2a.

The curve also shows the detection of the B&R X20 module, which is in both cases worse detected than the other modules. With 50% StarGAN, the recall reaches a maximum of about 0.45, suggesting that the model is missing many positive examples. This is confirmed after a visual inspection of the predictions. The module is either predicted with high confidence or not at all. This explains the line for the B&R X20 in Figure 3.2b, as there are no more precisions for a recall over 0.45.

3.3 Object Detectors

This section aims to compare different YOLO and Faster R-CNN models. All YOLO models were trained for 100 epochs, Faster R-CNN for 1000 epochs. Training them for some more epochs did not increase or decrease the accuracy.

					$\mathbf{mAP^{50}}$							
	Model	**Epochs**	**Training Time**	**Inference Time (ms per image)**	**All Classes**	**Bachmann MX213**	**Bachmann DO232**	**Siemens ET200SP**	**Beckhoff CX5010**	**Beckhoff CX1100**	**Omron CJ2M**	**B&R X20**
YOLO	YOLOv8n	100	6h	2	79.1	88.1	82.7	90.2	95.3	64.5	85.8	50.0
YOLO	YOLOv8m	100	12h	7	84.4	91.3	87.0	92.9	96.6	82.3	83.5	57.1
YOLO	YOLOv8l	100	16h	11	82.7	90.4	87.6	93.4	96.0	66.7	84.9	59.7
Faster R-CNN	R50-C4-1x	1000	45min	1040	69.4	55.8	68.6	87.1	92.9	61.7	66.3	53.7
Faster R-CNN	R50-C4-3x	1000	50min	1065	69.4	72.5	69.5	80.7	92.0	53.1	68.6	49.7
Faster R-CNN	R50-FPN-1x	1000	20min	438	73.1	74.5	75.9	85.3	93.9	71.4	75.9	36.2
Faster R-CNN	R50-FPN-3x	1000	20min	468	77.6	69.1	77.6	82.9	93.8	80.7	76.1	62.7

Table 3.4: Results of different YOLO and Faster R-CNN models, 50% of the training dataset translated with StarGAN

YOLO achieves better results than Faster R-CNN as listed in Table 3.4. YOLOv8m reaches an overall mAP^{50} of 84.4% for all classes, whereas Faster R-CNN with the R50-FPN-3x backbone only reaches a mAP^{50} of 77.6%. However, training Faster R-CNN is significantly faster with about 20 minutes, compared to the 12 hours of YOLOv8m.

When comparing the inference time, YOLO clearly has a faster detection performance than Faster R-CNN due to its one-stage algorithm predicting bounding boxes and labels in a single pass. The smallest model (YOLOv8n) detects objects with an interference time of 2ms per image, and the largest tested model (YOLOv8l) takes about 11ms. The Faster R-CNN R50-C4 backbone takes about 1050ms, and the R50-FPN backbone is faster but still takes about 450ms.

The YOLOv8m (medium) model achieves the best results when comparing different YOLO models. YOLOv8n (nano) achieves a lower mAP but took only half of the training time compared to YOLOv8m and has the fastest inference time. YOLOv8l (large) is the largest model, and the accuracy is lower than YOLOv8m. This may be because the model is too complex for this use case. When training the smallest YOLOv8n model for a longer duration, the mAP values did not increase or decrease.

For Faster R-CNN, the R50-FPN-3x backbone has the best performance. It has a 3x learning rate schedule, compared to R50-FPN-1x, which uses the same R50-FPN backbone but with a 1x learning rate schedule. The R50-C4 backbone is the original one from the Faster R-CNN and performs worse than the R50-FPN backbone.

4 Summary and Outlook

The results of this project demonstrate that training an object detector based on CAD files is possible without using real labeled images. Real images are required to improve the synthetic images with a GAN, but these do not have to be annotated with bounding boxes and are less time-consuming to be made.

The following section will provide a short summary of the conducted experiments and the results. The last section provides an outlook with possible improvements and some concluding remarks.

4.1 Summary

The conducted experiments started with the generation of synthetic images. The initial data is based on CAD files of the PLC modules, which have to be converted to usable 3D models. These can then be rendered in a rendering engine, for which Unity was chosen in combination with the Perception Package. This provides a toolset to generate perfectly annotated images.

Due to the data being initially available as CAD files, the 3D models do not contain any information on how the texture should be rendered. When directly rendering them in a scene, they do not look as photo-realistic as the real objects. To overcome this issue, images are generated with domain randomization. The idea is to generate images in different scenarios. The object detector should then learn different representations of the modules and detect the modules on a real image as it sees the real image as another variation it has been trained on.

In Unity, custom scripts were developed to control the task of domain randomization. A new scene is generated for each image, composed of three layers. A background layer is generated with constantly changing objects and textures. This prevents the object detector from learning any correlation between the background and the PLC module. A foreground layer displays the PLC modules in all possible positions with different rotations and sizes. An occlusion layer is put on top of the other layers with randomly generated small objects. This covers part of the PLC module to learn scenarios where not the entire module is visible. In a real image, this simulates the cables plugged into or occluding part of the module.

Training the object detector directly on the synthetic data only works when using high-quality and realistic CAD models. However, some modules of this project have a big visual difference between the digital and real objects. An example is the B&R X20 module, which has a CAD model with a gray surface with correct dimensions, but the real module is black and orange. The object detector fails to detect this module in a real image when trained on synthetic data only.

The next experiments researched GANs for the task of image-to-image translation. They are trained with real and synthetic images of individual modules and can then translate images between these two domains. CycleGAN was tested as a GAN that only supports two domains, and a separate GAN must be trained for each module. Additionally, StarGAN was implemented, which supports multiple domains and allows the training of one GAN for all PLC modules.

Using a GAN brought new challenges as the GAN outputs images without bounding boxes. Since the modules on the output image do not have precisely the same position and dimension as on the input image, they can not be directly pasted onto the synthetic image. To ensure this process works for every object, a pipeline was developed to paste objects back onto the synthetic image by keeping the exact dimensions and bounding boxes.

When directly comparing CycleGAN and StarGAN, StarGAN achieves better results and higher accuracy during object detection. CycleGAN struggles with translating unrealistic CAD models. Overall, translating all images of a dataset with a GAN results in worse detection accuracy than only translating a part of the dataset. Translating about 50% of the images of a synthetic dataset with StarGAN achieves the best results. This also demonstrates the importance of synthetic data as it is still required and has a positive effect.

Tests then compared the use of a GAN with fine-tuning the synthetic dataset on real labeled images, and StarGAN still achieves a higher mAP. Only a small dataset of real images was used for fine-tuning, and a larger dataset may improve the results. However, this comes with the significant overhead of needing real and labeled images. The GANs also require real images, but the images do not have to be annotated with bounding boxes or show the modules in completely different scenarios.

Different YOLO and Faster R-CNN models were tested and evaluated for the object detection task. YOLOv8, the newest version of YOLO, achieves the highest mAP. It detects almost all objects with a mAP^{50} of 84.4% when translating 50% of the dataset with StarGAN. It detects most modules even better than that except the B&R X20 module with a mAP^{50} of only 57.1%. This module has the worst CAD quality and is rarely detected without a GAN.

Objects with a higher-quality CAD model generally achieve better results than low-quality ones. Adding a GAN improves the detection rate for all modules, as all CAD models are missing photo-realistic textures.

When comparing YOLO with Faster R-CNN, YOLOv8 achieves better mAP values. In addition, YOLO is extremely fast, with an inference time of below 10 milliseconds per image, compared to the over 400 milliseconds with the fastest Faster R-CNN model. However, it requires more training time, about 12 hours compared to the 20 minutes for Faster R-CNN. Still, YOLOv8 is the preferred object detection algorithm for this project.

4.2 Outlook

This thesis demonstrates that object detection with CAD-based synthetic data is possible. Combined with GANs and some real images, it is a clear alternative to using only real images during training, even if the CAD models are not photo-realistic. Generating thousands of synthetic images is significantly cheaper than creating and labeling real images.

Still, there may be some improvements with further research. The pipeline to translate synthetic images with a GAN could be further optimized for larger datasets. The current implementation translates all objects in one image simultaneously but not multiple synthetic images at the same time, which is not that performance optimized. This could be improved by translating larger batches with a GAN or introducing multithreading. This was no issue for this project, but it will lead to performance issues when using way more data.

The solution implemented in this project works for PLC modules but could also be used in other areas. The generation of synthetic images is independent of the type of object and can easily be adapted. Additionally, not only CAD models but almost any type of 3D model could be used.

Bibliography

[1] S. Borkman, A. Crespi, S. Dhakad, *et al.*, *Unity Perception: Generate Synthetic Data for Computer Vision*, arXiv:2107.04259 [cs], Jul. 2021. DOI: 10.48550/arXiv.2107.04259. [Online]. Available: http://arxiv.org/abs/2107.04259 (visited on 03/23/2023).

[2] S. Hinterstoisser, O. Pauly, H. Heibel, M. Marek, and M. Bokeloh, *An Annotation Saved is an Annotation Earned: Using Fully Synthetic Training for Object Instance Detection*, arXiv:1902.09967 [cs], Feb. 2019. DOI: 10.48550/arXiv.1902.09967. [Online]. Available: http://arxiv.org/abs/1902.09967 (visited on 03/23/2023).

[3] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel, *Domain Randomization for Transferring Deep Neural Networks from Simulation to the Real World*, arXiv:1703.06907 [cs], Mar. 2017. DOI: 10.48550/arXiv.1703.06907. [Online]. Available: http://arxiv.org/abs/1703.06907 (visited on 04/03/2023).

[4] P. Tang, Y. Guo, H. Li, Z. Wei, G. Zheng, and J. Pu, "Image dataset creation and networks improvement method based on CAD model and edge operator for object detection in the manufacturing industry," *Machine Vision and Applications*, vol. 32, Sep. 2021. DOI: 10.1007/s00138-021-01237-y.

[5] I. G. B. Sampaio, L. Machaca, J. Viterbo, and J. Guérin, *A novel method for object detection using deep learning and CAD models*, arXiv:2102.06729 [cs], Feb. 2021. DOI: 10.48550/arXiv.2102.06729. [Online]. Available: http://arxiv.org/abs/2102.06729 (visited on 03/17/2023).

[6] C. Manettas, N. Nikolakis, and K. Alexopoulos, "Synthetic datasets for Deep Learning in computer-vision assisted tasks in manufacturing," en, *Procedia CIRP*, 9th CIRP Global Web Conference – Sustainable, resilient, and agile manufacturing and service operations : Lessons from COVID-19, vol. 103, pp. 237–242, Jan. 2021, ISSN: 2212-8271. DOI: 10.1016/j.procir.2021.10.038. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2212827121008799 (visited on 04/03/2023).

[7] P. Isola, J.-Y. Zhu, T. Zhou, and A. A. Efros, *Image-to-Image Translation with Conditional Adversarial Networks*, arXiv:1611.07004 [cs], Nov. 2018. DOI: 10.48550/arXiv.1611.07004. [Online]. Available: http://arxiv.org/abs/1611.07004 (visited on 03/23/2023).

[8] J.-Y. Zhu, T. Park, P. Isola, and A. A. Efros, *Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks*, arXiv:1703.10593 [cs], May 2017. DOI: 10.48550/arXiv.1703.10593. [Online]. Available: http://arxiv.org/abs/1703.10593 (visited on 03/23/2023).

[9] Y. Choi, M. Choi, M. Kim, J.-W. Ha, S. Kim, and J. Choo, *StarGAN: Unified Generative Adversarial Networks for Multi-Domain Image-to-Image Translation*, arXiv:1711.09020 [cs], Sep. 2018. DOI: 10.48550/arXiv.1711.09020. [Online]. Available: http://arxiv.org/abs/1711.09020 (visited on 03/23/2023).

[10] Y. Choi, Y. Uh, J. Yoo, and J.-W. Ha, *StarGAN v2: Diverse Image Synthesis for Multiple Domains*, arXiv:1912.01865 [cs], Apr. 2020. DOI: 10.48550/arXiv.1912.01865. [Online]. Available: http://arxiv.org/abs/1912.01865 (visited on 05/19/2023).

[11] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, *You Only Look Once: Unified, Real-Time Object Detection*, arXiv:1506.02640 [cs], May 2016. DOI: 10.48550/arXiv.1506.02640. [Online]. Available: http://arxiv.org/abs/1506.02640 (visited on 05/17/2023).

[12] J. Redmon and A. Farhadi, *YOLO9000: Better, Faster, Stronger*, arXiv:1612.08242 [cs], Dec. 2016. DOI: 10.48550/arXiv.1612.08242. [Online]. Available: http://arxiv.org/abs/1612.08242 (visited on 05/29/2023).

[13] J. Redmon and A. Farhadi, *YOLOv3: An Incremental Improvement*, arXiv:1804.02767 [cs], Apr. 2018. DOI: 10.48550/arXiv.1804.02767. [Online]. Available: http://arxiv.org/abs/1804.02767 (visited on 05/29/2023).

[14] A. Bochkovskiy, C.-Y. Wang, and H.-Y. M. Liao, *YOLOv4: Optimal Speed and Accuracy of Object Detection*, arXiv:2004.10934 [cs, eess], Apr. 2020. [Online]. Available: http://arxiv.org/abs/2004.10934 (visited on 05/29/2023).

[15] Ultralytics, *YOLOv5*, May 2020. DOI: 10.5281/zenodo.3908559. [Online]. Available: https://github.com/ultralytics/yolov5 (visited on 05/29/2023).

[16] C. Li, L. Li, H. Jiang, *et al.*, *YOLOv6: A Single-Stage Object Detection Framework for Industrial Applications*, arXiv:2209.02976 [cs], Sep. 2022. DOI: `10.48550/arXiv.2209.02976`. [Online]. Available: `http://arxiv.org/abs/2209.02976` (visited on 05/29/2023).

[17] C.-Y. Wang, A. Bochkovskiy, and H.-Y. M. Liao, *YOLOv7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors*, arXiv:2207.02696 [cs], Jul. 2022. DOI: `10.48550/arXiv.2207.02696`. [Online]. Available: `http://arxiv.org/abs/2207.02696` (visited on 05/29/2023).

[18] Ultralytics, *YOLOv8*, May 2023. [Online]. Available: `https://github.com/ultralytics/ultralytics` (visited on 05/29/2023).

[19] J. Terven and D. Cordova-Esparza, *A Comprehensive Review of YOLO: From YOLOv1 and Beyond*, arXiv:2304.00501 [cs], May 2023. DOI: `10.48550/arXiv.2304.00501`. [Online]. Available: `http://arxiv.org/abs/2304.00501` (visited on 05/29/2023).

[20] J. Solawetz and Francesco, *What is YOLOv8? The Ultimate Guide.* en, Jan. 2023. [Online]. Available: `https://blog.roboflow.com/whats-new-in-yolov8/` (visited on 05/29/2023).

[21] A. Aboah, B. Wang, U. Bagci, and Y. Adu-Gyamfi, *Real-time Multi-Class Helmet Violation Detection Using Few-Shot Data Sampling Technique and YOLOv8*, arXiv:2304.08256 [cs], Apr. 2023. DOI: `10.48550/arXiv.2304.08256`. [Online]. Available: `http://arxiv.org/abs/2304.08256` (visited on 05/29/2023).

[22] S. Ren, K. He, R. Girshick, and J. Sun, *Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks*, arXiv:1506.01497 [cs], Jan. 2016. DOI: `10.48550/arXiv.1506.01497`. [Online]. Available: `http://arxiv.org/abs/1506.01497` (visited on 06/09/2023).

[23] R. Girshick, J. Donahue, T. Darrell, and J. Malik, *Rich feature hierarchies for accurate object detection and semantic segmentation*, arXiv:1311.2524 [cs], Oct. 2014. DOI: `10.48550/arXiv.1311.2524`. [Online]. Available: `http://arxiv.org/abs/1311.2524` (visited on 06/09/2023).

[24] R. Girshick, *Fast R-CNN*, arXiv:1504.08083 [cs], Sep. 2015. DOI: `10.48550/arXiv.1504.08083`. [Online]. Available: `http://arxiv.org/abs/1504.08083` (visited on 06/09/2023).

[25] K. He, G. Gkioxari, P. Dollár, and R. Girshick, *Mask R-CNN*, arXiv:1703.06870 [cs], Jan. 2018. DOI: `10.48550/arXiv.1703.06870`. [Online]. Available: `http://arxiv.org/abs/1703.06870` (visited on 06/09/2023).

[26] Z. Cai and N. Vasconcelos, *Cascade R-CNN: High Quality Object Detection and Instance Segmentation*, arXiv:1906.09756 [cs], Jun. 2019. DOI: 10.48550/arXiv.1906.09756. [Online]. Available: http://arxiv.org/abs/1906.09756 (visited on 06/09/2023).

[27] A. Jabbar, L. Farrawell, J. Fountain, and S. K. Chalup, "Training Deep Neural Networks for Detecting Drinking Glasses Using Synthetic Images," en, in *Neural Information Processing*, D. Liu, S. Xie, Y. Li, D. Zhao, and E.-S. M. El-Alfy, Eds., ser. Lecture Notes in Computer Science, Cham: Springer International Publishing, 2017, pp. 354–363, ISBN: 978-3-319-70096-0. DOI: 10.1007/978-3-319-70096-0_37.

[28] U. Engine, *Datasmith*, en-US, 2023. [Online]. Available: https://www.unrealengine.com/en-US/datasmith (visited on 05/01/2023).

[29] UnrealCV, *UnrealCV*, original-date: 2016-09-08T18:03:51Z, Jul. 2019. [Online]. Available: https://github.com/unrealcv/unrealcv (visited on 03/23/2023).

[30] UnrealCV, *UnrealCV - Issues*, original-date: 2016-09-08T18:03:51Z, Apr. 2023. [Online]. Available: https://github.com/unrealcv/unrealcv/issues (visited on 05/01/2023).

[31] U. Technologies, *About Tessellation*, en, 2022. [Online]. Available: https://www.pixyz-software.com/documentations/html/2021.1/studio/AboutTessellation.html (visited on 04/01/2023).

[32] U. Technologies, *Synthetic Optimized Labeled Objects (SOLO) Dataset Schema*, 2022. [Online]. Available: https://docs.unity3d.com/Packages/com.unity.perception@1.0/manual/Schema/SoloSchema.html (visited on 04/07/2023).

[33] T.-Y. Lin, M. Maire, S. Belongie, *et al.*, *Microsoft COCO: Common Objects in Context*, arXiv:1405.0312 [cs], Feb. 2015. [Online]. Available: http://arxiv.org/abs/1405.0312 (visited on 04/07/2023).

[34] M. Heusel, H. Ramsauer, T. Unterthiner, B. Nessler, and S. Hochreiter, "GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium," in *Advances in Neural Information Processing Systems*, vol. 30, Curran Associates, Inc., 2017. [Online]. Available: https://papers.nips.cc/paper_files/paper/2017/hash/8a1d694707eb0fefe65871369074926d-Abstract.html (visited on 05/31/2023).

[35] Y. Wu, A. Kirillov, F. Massa, W.-Y. Lo, and R. Girshick, *Detectron2*, 2019. [Online]. Available: https://github.com/facebookresearch/detectron2.

List of Acronyms

AP Average Precision

CAD Computer-Aided Design

CNN Convolutional Neural Network

COCO Common Objects in Context

FID Frechet Inception Distance

GAN Generative Adversarial Network

IoU Intersection over Union

YOLO You Only Look Once

mAP mean Average Precision

PLC Programmable Logic Controller

PR Precision-Recall

R-CNN Region-based Convolutional Neural Network

RPN Region Proposal Network

SOLO Synthetic Optimized Labeled Objects

List of Figures

1.1 Example PLC modules 7
1.2 Overview of the CycleGAN model (source: [8]) 9
1.3 Overview of the StarGANv2 modules (source: [10]) 9
1.4 YOLO model (source: [11]) 11
1.5 Faster R-CNN model (source: [22]) 12

2.1 CAD and real images of used PLC modules 14
2.2 Different tessellation detail levels 18
2.3 Example frame and side view of the Unity scene 20
2.4 Example poses of a PLC module 22
2.5 Synthetic image with and without the occlusion layer 22
2.6 Example synthetic images 23
2.7 Example GAN training images 25
2.8 Example GAN training images for B&R X20 26
2.9 CycleGAN example translations 28
2.10 Comparison between cycle-consistency losses of CycleGAN . . . 28
2.11 StarGAN image synthesis results 30
2.12 GAN image processing pipeline 32
2.13 Options to transfer the objects from the GAN output image . . 34
2.14 Example images from the real validation dataset 39

3.1 Example YOLO object detection images 42
3.2 Precision-recall curves for YOLOv8m with and without GAN . . 45

List of Tables

2.1 CycleGAN training times 29
2.2 Distribution of instances in the real validation dataset 40

3.1 Overview of the YOLO and Faster R-CNN results 42
3.2 Detailed mAP^{50} results for all PLC modules 43
3.3 Results evaluated on a smaller real dataset, trained with and without the use of GANs 44
3.4 Results of different YOLO and Faster R-CNN models, 50% of the training dataset translated with StarGAN 46

Statuatory Declaration

I declare that I have developed and written the enclosed work completely by myself, and have not used sources or means without declaration in the text. Any thoughts from others or literal quotations are clearly marked. This Master Thesis was not used in the same or in a similar version to achieve an academic degree nor has it been published elsewhere.

Dornbirn, July 08, 2023 Lukas Lins