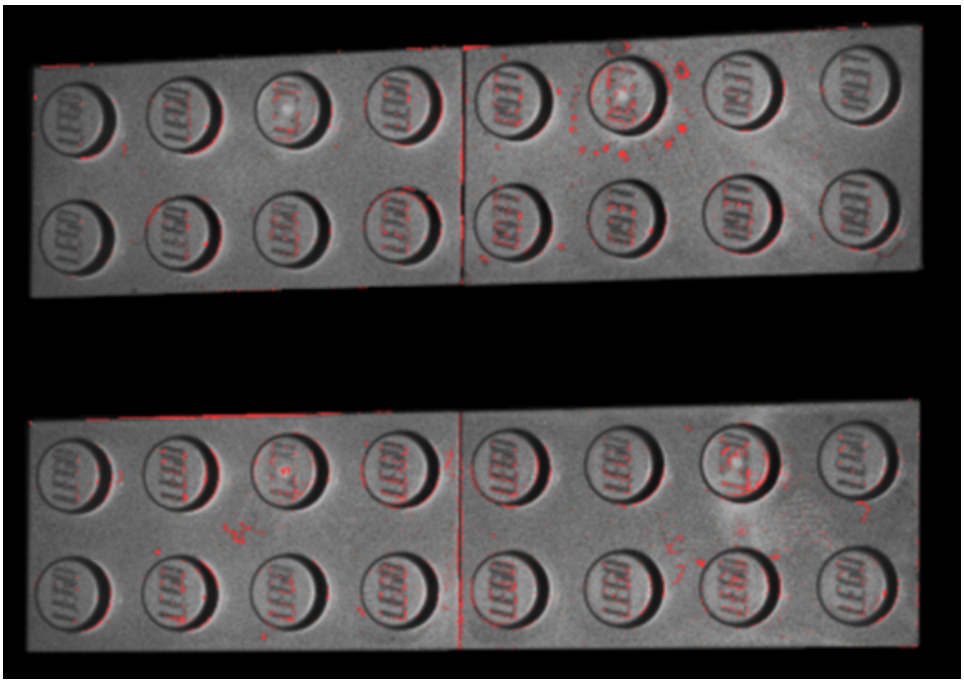
Utilizing synthetic data in defect detection in polymer surfaces



Master's Thesis

MTA211044

Aalborg University
Department of Architecture, Design and Media Technology
Rendsburggade 12-14
DK-9000 Aalborg

Copyright © Aalborg University 2021

AALBORG
UNIVERSITET

Department of Architecture, Design and Media Technology

Medialogy 10^{th} Semester
Aalborg University
http://www.aau.dk

Title:
Utilizing synthetic data in defect detection in polymer surfaces

Theme:
Master's Thesis

Project Period:
Spring Semester 2021

Project Group:
MTA211044

Members(s):
Anders S. Jensen
Lars Q. W. Nielsen
Marlene G. Lomborg
Mike L. H. Nguyen

Supervisor(s):
Mark Philip Philipsen

Page Numbers: 91

Date of Completion:
May 27, 2021

Abstract:

In the manufacturing industry, quality control is necessary to keep a high product standard. This process is typically manual labor, which can be subject to human error. A possible solution is an automated process using unsupervised deep learning, as defect data is uncommon in the manufacturing industry, making labeled data limited. This report investigates the generation of synthetic data and automatic segmentation to improve anomaly detection performance with different neural network architecture to find the one that yields the best results for detecting defects. U-net was implemented to segment the background and make the model more robust from background noise. A convolutional autoencoder was created for anomaly detection. In conclusion, U-net provided functional masks. The synthetic data could increase the dataset and improve a model; however, the similarity to the real dataset will need to be improved. The autoencoder created a distinction between defects and non-defects, but only for images from the same view. An adaptive anomaly threshold will need to be explored further.

Copyright©2016 This report and/or appended material may not be partly or completely published or copied without prior written approval from the authors. Neither may the contents be used for commercial purposes without this written approval.

Contents

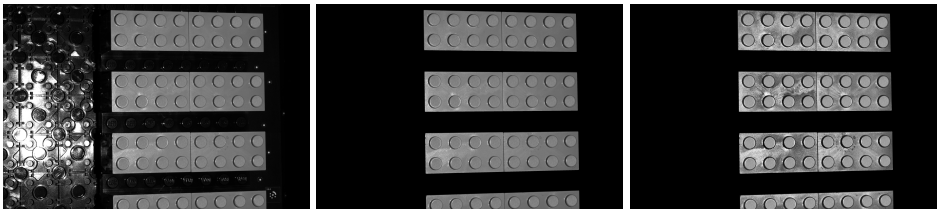
Chapter 1

Introduction

The manufacturing industry seeks to uphold a production standard for the products and is researching ways to improve the current methods. One of these areas is visual quality control. One company researching this is LEGO; they are currently using manual visual inspection to look for defects in their products. With manual visual inspection comes some different issues, such as training experts to identify the different defects. Since human decisions are subjective, there can be a difference in what each quality assessor marks as a defect, thereby creating individual standards. The current process for LEGO is taking samples from a batch and manually looking at them under a light to detect the defects in the bricks. This process can lead to issues such as the inspector getting eye strain which can cause the inspector to miss a defect. A possible solution to the time-consuming quality control is to have an automatic process or screening tool to assist the quality assessors by utilizing deep learning and computer vision. Deep learning has shown great promise for defect, and anomaly detection in the manufacturing context [1], as defects can be detected only with the use of data and with limited manual labor. Defects can occur in any shape and size, making unsupervised learning a viable option for detecting unknown/not previously seen defects. Deep learning, being very data-dependent, requires as much data as possible. However, in the manufacturing context, data acquisition might be in its starting stages, having a small amount of data available.

1.1 Previous work

This project is a continuation of an earlier project we had in collaboration with LEGO. The project focused on exploring techniques for quantifying and objectifying the quality control measure for polymer surfaces. This is done by using unsupervised deep learning strategies, where autoencoders were only trained on non-defect data to find outliers. Experiments were conducted with background segmentation, Contrast-Limited Adaptive Histogram Equalization (CLAHE), and data augmentation to show the effect of the addition of each processing step.



(a) Raw data (b) Segmented data (c) Preprocessed data

Figure 1.1: Showing the same image with no modification, masked and preprocessed

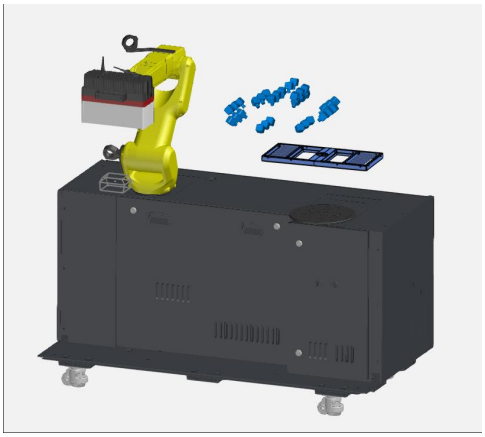
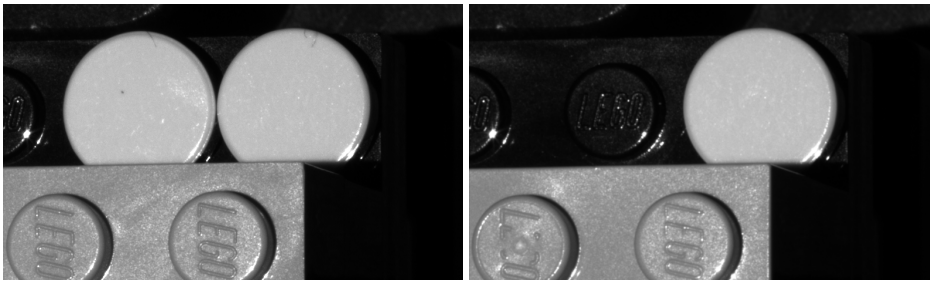


Figure 1.2: ATOS ScanBox with a mounted ATOS Capsule scanner (Image provided by collaborator)

The dataset consisted of 10 batches with 24 images in each batch. The 24 images were taken from different angles of the units. Figure 1.2 shows the ATOS ScanBox[2] with the ATOS Capsule scanner[3] attached. The scanner was rotating around the bricks, while taking gray scale images of the bricks. Examples of changes done to the original dataset be seen in Figure 1.1, which showcase the original images, a segmented dataset with the background removed. Furthermore, a segmented dataset with Contrast Limited Adaptive Histogram equalization (CLAHE) applied, enhancing the defects seen in the defective images visually. The three different versions were created to test the effect of removing the background and applying CLAHE. The original dataset was discarded because of a defect indicator next to the bricks see Figure 1.3. The indicator was removed during the background segmentation and was not present in the subsequent datasets.



(a) With defect (b) Without defect

Figure 1.3: Figure show indicator bricks that was used to show if the image is of a batch with or without defect

In the end, the experiments resulted in three test cases for the datasets:

- Background segmented dataset with no CLAHE or data augmentation
- Background segmented dataset with CLAHE applied but no data augmentation
- Background segmented dataset with CLAHE and with data augmentation added during training

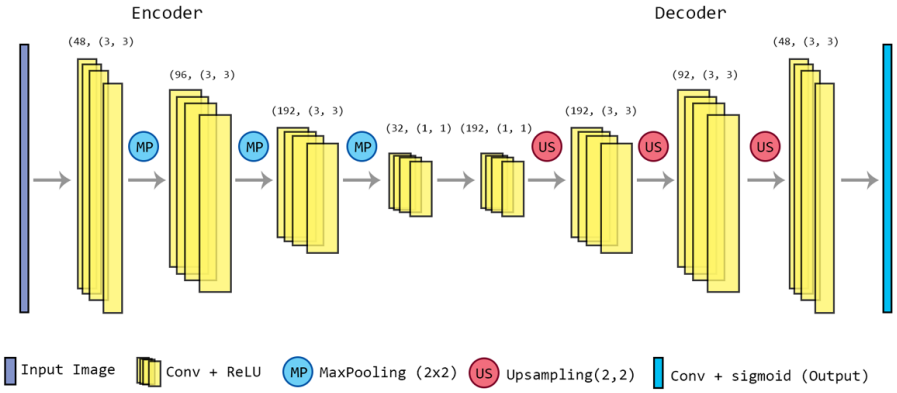


Figure 1.4: Old CAE model

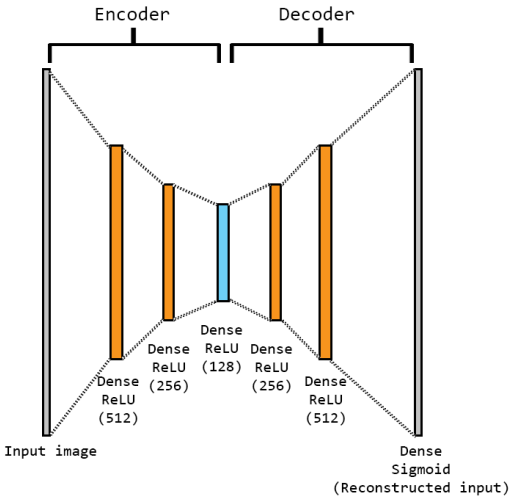
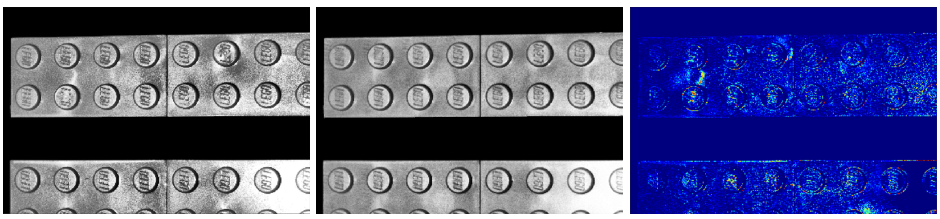


Figure 1.5: Old DAE model

A deep autoencoder(DAE) and a convolutional autoencoder(CAE), were used for anomaly detection can be seen in figure 1.4 and 1.5. The key differences between the two were the DAE being fully connected and the CAE using 2D convolutional kernels and more layers. Both models used Adam as optimizer and ReLU-activations for all convolutional and dense layers except the final layer with sigmoid activation. The datasets were split into train, validation, and test sets. Both autoencoders were trained only on good samples and would adjust the weights in the networks based on the reconstruction error between the input image and the reconstruction.

The finished trained models would be evaluated with the test set. If a test image had any defects, they would be removed in the reconstruction. The results were a patch-wise anomaly score and an average global anomaly score across the input image. An example of anomaly detection can be seen in figure



(a) Input image (b) Reconstructed image (c) Difference image

Figure 1.6: Image (with background segmentation and CLAHE applied) with no defects and reconstruction generated by the CAE with data augmentation enabled. The differences are squared and shown as a difference map. The model was trained for 130 epochs with binary cross-entropy loss.

(a) Input (b) Reconstructed image (c) Anomaly Output

Figure 1.7: Segmented and contrast enhanced data prediction and anomaly output using DAE

The datasets were trained separately and evaluated upon (see Figure 1.1). Before training, the full images of size 4248x2832 pixels were resized to 750x500 and sliced into 256x344 patches, with a 64-pixel overlap. Random rotation was applied to each slice in a -5 to 5 degree range. The area under the ROC Curve (AUC ROC) and f1-score were used as the evaluating measures, which can be seen in Figure 1.8. The experiments showed that when the images were both masked and preprocessed, the anomaly detection improved the most, with an AUCROC of 0.96 and f1-score of 0.70 (figure 1.1). The ROC curves for all experiments can be seen in Figure 1.1c.

Datasets	DAE		CAE	
	F1	AUCROC	F1	AUCROC
Segmented	0.29	0.63	0.55	0.82
Segmented and Contrast Enhanced	0.54	0.77	0.59	0.91
Segmented, Contrast Enhanced and Augmented	**0.68**	**0.92**	**0.70**	**0.96**

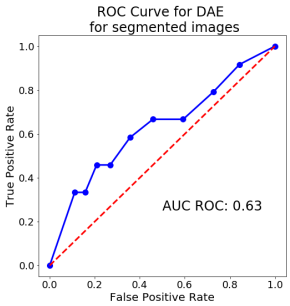
Table 1.1: F1- score and AUCROC for DAE and CAE.

Statistical Process Control (SPC) is a method of monitoring quality control in manufacturing. It was included as an exploratory measure and calculated with batch-based Quality Index (nQI) [4], given by the formula:

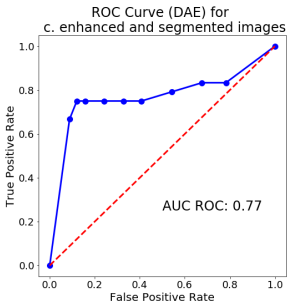
$$nQI = \frac{C - min_C(X_C)}{\sqrt{\frac{\sum_C (C-\mu_C)^2}{n_C}}} \tag{1.1}$$

Where C is the total sum of all anomaly pixels for a single image angle, min_C is the minimum value for the same angle across all batches, X_C is the standard deviation over a population across all batches for a single angle, μ_C is the mean of a single image angle across all batches and n_C is the sample size of all image angles and batches.

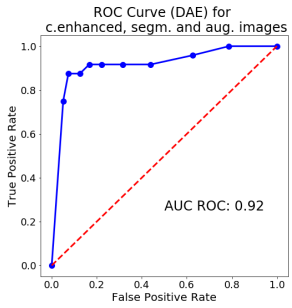
The best results for CAE and DAE can be seen in figure 1.9a and 1.9b.



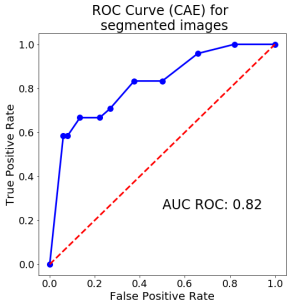
(a) DAE: Segmented images



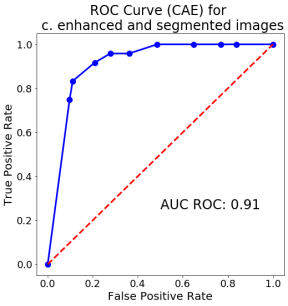
(b) DAE: Segmented and Contrast Enhanced images



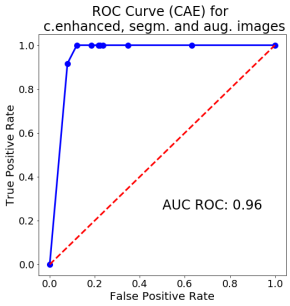
(c) DAE: Segmented, Contrast Enhanced and Augmented images



(d) CAE: Segmented images



(e) CAE: Segmented and Contrast Enhanced images



(f) CAE: Segmented, Contrast Enhanced and Augmented images

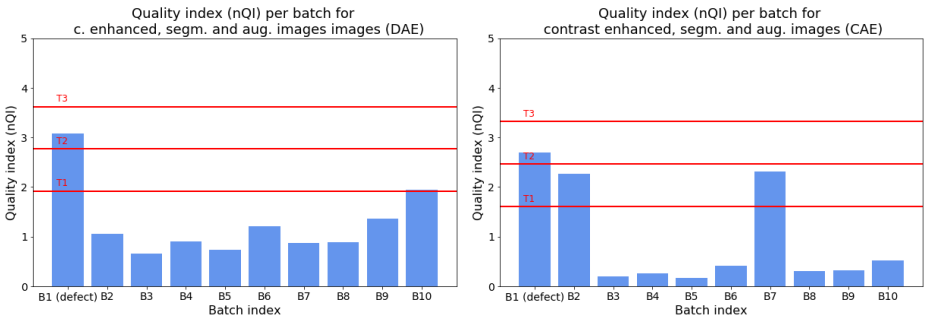
Figure 1.8: ROC curves for the different datasets. A true positive would be a defect classified correctly.

1.1.1 Takeaways from 8th semester project

This section describes what was attempted in the 8th semester project and could be improved this Semester.

Background Subtraction

During the 8th semester project, background subtraction was proven to be needed as the images of defects had an extra knob as shown in Figure 1.3. When the network trained on the images, it learned how many knobs there were present and did not take the polymer surface into account. Therefore, to gain any knowledge about the polymer surface, the background and unnecessary information need to be removed. In order to remove the background, a mask was needed to show what was the background and what was not. These masks were done by hand and proved to be a tedious and tiresome task, as mask was needed for each image and segmented. Making this process a lot smoother by automation could reduce time spent and make the mask more uniform.



(a) SPC for DAE

(b) SPC for CAE

Figure 1.9: SPC for each batch in the segmented, contrast-enhanced, and data augmented dataset for both DAE and CAE. The batch with defects are labeled as *B1*.

Agnostic and robust system

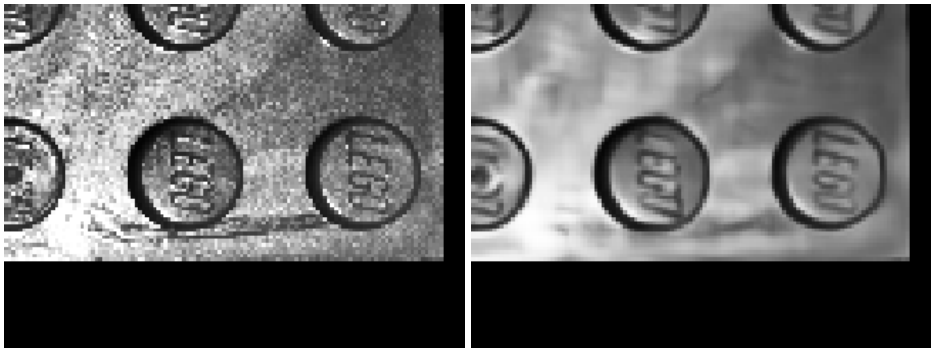
In the project on our 8th semester, we had a quick look at data augmentation. We looked specifically at rotation, yet there are various methods that we could apply when training our model or models. Using data augmentation can artificially increase the amount of data in a dataset. So when working with small datasets, its use can significantly impact the trained model's accuracy.

Improving the previous network

While the networks showed great promise and could distinguish between defect and non-defect batches, there were some problems to look at. The previous project used rotation as the data augmentation method and a fully connected model (DAE), and a convolutional model(CAE). Both models were under-fitted, so further developments would be to introduce other data augmentations and exploring other more complex neural network architectures to increase the performance and accuracy of defect predictions. The DAE network was too sensitive to data augmentation and had worse results than the CAE, so convolutional neural networks will be the starting point of further research. The CAE, while better than the DAE, could still be improved, as it did have issues with not being generalized enough and reconstructing defects instead of removing them and creating a non-defect image(example in Figure 1.10).

Furthermore, the networks used Central Processing Unit (CPU) to train on, which caused long training times, up to 6 hours, or even errors caused by lack of memory. This can be improved by using the Graphics Processing Unit(GPU) instead. Since the GPU has a higher memory bandwidth and has dedicated Video Random Access Memory(VRAM)[5] makes it easier for the GPU to handle big amount of data and make the training and predictions faster.

Using AUCROC was not the best measure, as the dataset was imbalanced, so



(a) Input image with defect (b) Defect reconstructed

Figure 1.10: Example of defect that was reconstructed instead of removed.

changing it to Area under the Precision-Recall curve would be better to compare any future models.

1.2 Problem Statement

Based on the takeaways from the previous project, this report will go through the process of creating a system that can automatically segment and detect defects across multiple different real-world datasets. The system needs to support batches with multiple image data from different angles and provide an objective anomaly score. These aspects can be combined in the problem statement:

How can a robust defect detection system be made for visual quality control, which utilizes deep learning and supports small and varying datasets as input?

To answer this, the following research questions were created:

1. How can we artificially generate more data using data augmentation or other methods?
2. How can the system be made robust towards lighting, perspective, and placement changes?
3. How can the background of a dataset be segmented automatically?
4. How can a network be made more generalized to avoid reconstruction of defects while keeping a good representation of the data?
5. Can one system be used on multiple datasets?

Chapter 2

Research

This chapter will aim to answer the research questions presented in the previous chapter. The research covers Data Augmentation methods, Robustness in Deep learning, Image segmentation, and Anomaly Detection.

2.1 Deep learning Fundamentals

Deep learning is a subfield of machine learning in artificial intelligence (AI), where neural networks are trained to carry out a task. Neural networks consist of multiple layers, which can be described in three types; The Input Layer, which sets the dimensions for the input data. The output layer, which is the results of the network, and the hidden layers, which are all the layers between the input and output layer [6]. Each layer consists of a unit (or hidden units [7]) which gets activated based on the input, weight, and bias. Figure 2.1 shows how a single neuron in a layer is calculated.

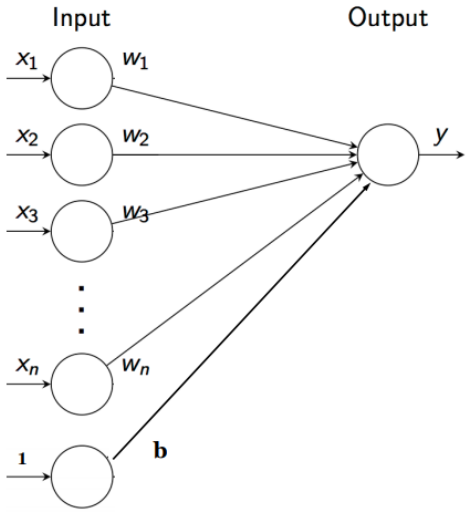


Figure 2.1: Image of An Individual Neuron[8]

A neuron or hidden unit can be described as:

$$Output = \sigma \sum_{i=1}^{n} x_i w_i + b, \tag{2.1}$$

Where the output of the neuron is equals, the sum of weights times input, added with a bias[6]. One single input is referred to as a sample, and if supervised training is done, classes are referred to as labels. Training a network is usually done in batches of a dataset. When all batches have been trained on, it is called an Epoch. The simplest neural networks are feed forward, meaning they receive an input, which passes through the hidden layers and returns an output without ever revisiting the other layers in a cycle.

Layers, Activation and Weights

Each unit has an activation function, which decides whether a neuron should be activated (returning an output) or not. Tanh, also known as a hyperbolic tangent, ranges from -1 to 1, as seen in Figure 2.2a. It is used for classification mainly[6].

The sigmoid function is similar to the Tanh function, as they are both S-shaped. The sigmoid function is useful for the prediction of probabilities, as they only can exist in ranges 0 to 1. The function is expressed as[6]:

$$f(x) = \frac{1}{1 + exp^{-x}} \tag{2.2}$$

and its curve can be seen in Figure 2.2b.

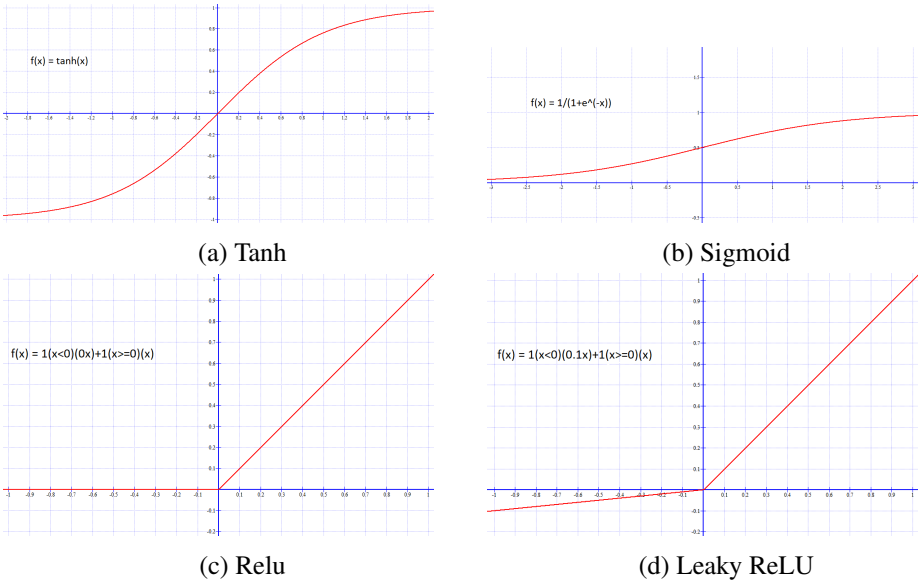
Both the Tanh (Figure 2.2a) and Sigmoid (Figure 2.2b) activations suffer from the vanishing gradient problem [9]. The vanishing gradient problem is a problem that can occur under each iteration in the early layers of training when using gradient-based learning methods to update the weights. The problem happens

when the gradient is so tiny that it prevents the weights from update their values during backward propagation. If the network can not change the values for the weight, it can, in the worst case, cause the network not to learn the data's features. Another method that does not suffer from the vanishing gradient problem is the activation function Rectified Linear Unit (ReLU)2.2c[10]. The ReLU-activation outputs any negative input to be 0, making the network lighter, and it's linearity makes it avoid any expensive exponential computations effectively making it faster to train than sigmoid and tanh. The ReLU-activation function is expressed as:

$$f(x) = max(0, x) \tag{2.3}$$

where x is the input[6].

The ReLU activation function can suffer from dying ReLU, a similar problem as the vanishing gradient problem. The neurons in the network end up in a state where they are inactive or *dead*, meaning the neurons' output zero. If too many neurons are inactive, the network can die. In practice this means that it can't predict anything. It usually happens when the learning rate is too high, but it can be lessened using leaky ReLU instead.[10](Figure 2.2d).



(a) Tanh

(b) Sigmoid

(c) Relu

(d) Leaky ReLU

Figure 2.2: Different types of activation functions

Convolution 2D

A kernel slides over a 2D matrix in a convolution operation could be an image and converts the matrix into another smaller matrix depending on if padding is used or not. The output is the weighted sums of the features where the weight is the kernel. It can be described by the following equation[6]:

$$G(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n), \tag{2.4}$$

where G is the resulting image I is the input image and K is the Kernel. If padding

is used, the values of the matrix will fall roughly the exact location as the original matrix. If no padding is used, the output matrix will be smaller than the original matrix decided by the size of the kernel. The reduction of the matrix can sometimes be called dimensionality reduction.

Dimensionality reduction can be accomplished by using max-pooling but can also be done with the use of stride. Stride works by skipping some of the slide locations. Depending on the stride, the convolutional operation is not done on each pixel. Some are skipped depending on the size of the stride. By doing so, the dimensionality of the matrix will be reduced like with max-pooling.

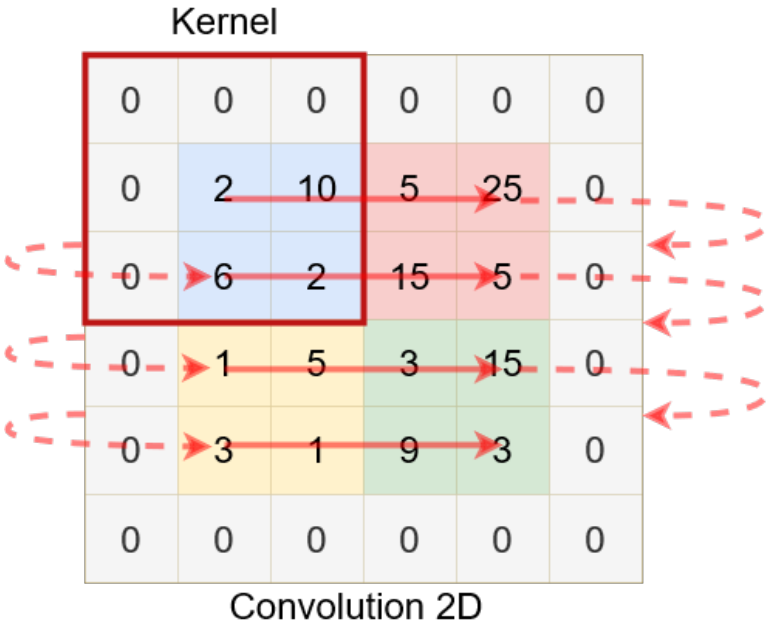


Figure 2.3: convolution 2D with a 3x3 kernel

Conv2DTranspose

Conv2DTranspose, a convolution layer that goes in the opposite direction compared to a normal convolution, performs an inverse convolution operation to upsample the input. Conv2DTranspose layers are often used in segmentation challenges or to increase the resolution of an image[11]. An example of how Conv2DTranspose works can be seen in Figure2.4 where a 2x2 input image is upscaled to 4x4.

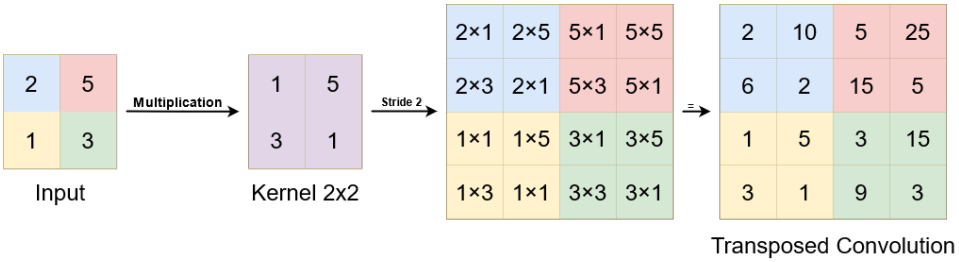


Figure 2.4: Conv2DTransposed

Not all layers in a neural network contains activation functions. Other operations such as max are commonly used[12].

Maxpooling is used to reduce the dimensionality of the output matrixes. Like with dropout, this also one way to avoid overfitting. The U-Net code Listing A.1 a MaxPooling of 2x2 is used, meaning that the dimensionality is halved, and an Example of using Maxpooling on a 4x4 matrix with the kernel size of 2x2 can be seen in Figure 2.5.

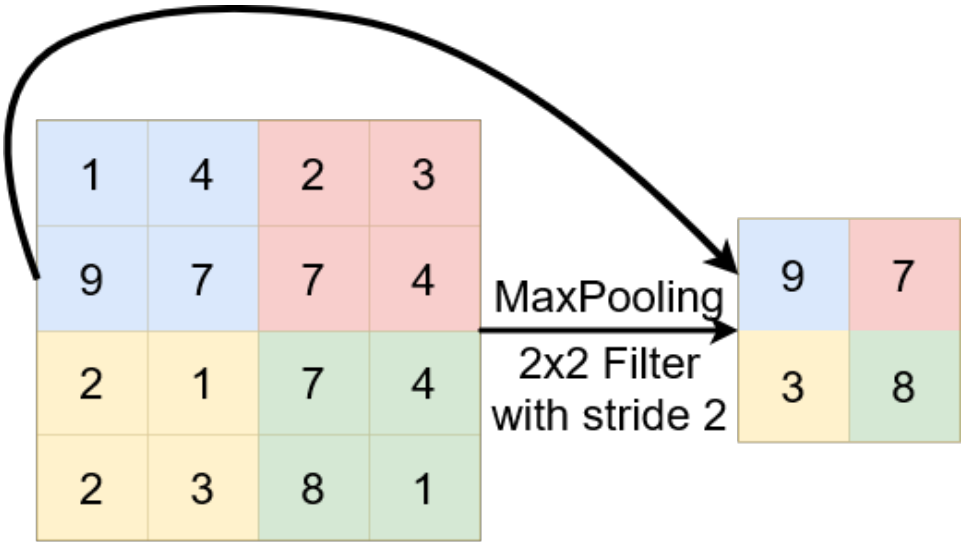


Figure 2.5: MaxPooling

Loss Functions and Optimization

The loss function is also referred to as the cost function or error function[6]. The aim is to maximize or minimize the error the loss function returns.

An example of minimizing the loss is with the loss function Mean Squared Error (MSE), which is expressed as[6]:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2, \tag{2.5}$$

where y is the input, $\hat{y}$ is the prediction based on the input.

Another loss function is Binary Cross-Entropy, Which is sometimes used in segmentation networks such as U-Net[13]. It returns a value between 0 and 1, and is used in binary classification cases where the probability of a class is needed. For instance, if a classifier is to determine whether the input is a dog or a cat, the loss function would return a value closer to 1, where 1 is a dog and 0 is a cat, then the input is predicted to being a 'dog' and vice versa.

Gradient Descent

An way of training a neural is by using Gradient Descent algorithm to minimize the loss and optimize the network accordingly through backward propagation (updating the weights).

The algorithm for gradient descent starts with finding the derivative for each parameter(weight and bias) of a chosen loss function (e.g., MSE), also known as taking the gradient of a loss function.

Random values are then initialized and inserted in the gradient. Step size is calculated with the formula: $StepSize = slope \cdot LearningRate$ With the step size, new parameters can be calculated with the formula: $Newparameter = OldParameter - StepSize$, and be inserted into the gradient. This cycle continues until the step size is below a specified minimum (e.g., 0.001). At that point, the gradient descent should have reached the minimum of the loss function, though this is not always the case see Figure2.6. Suppose the loss function has multiple 'valleys' or minimums. In that case, the gradient descent could reach a valley that would only be a local minimum or a saddle point (a plateau between a positive and negative slope) and not the global minimum. When all data is used to compute the gradient, it is referred to as Batch Gradient Descent[15]. Unfortunately, this variant isn't suitable for large datasets and doesn't allow for adding new sample without recalculating the gradient. Instead, Stochastic Gradient Descent (SGD) adds a randomness to the gradient descent by taking one sample at random and minimizing it. New samples can be added later. Another standard method is Mini-Batch Gradient Descent, where a random selection of samples is used to optimize (e.g., 50 samples instead of a single sample)[15, 6].

Another commonly used optimizer is Adaptive Moment Estimation (Adam)[16]. Where SGD uses a fixed learning rate, Adam uses an adaptive learning rate during

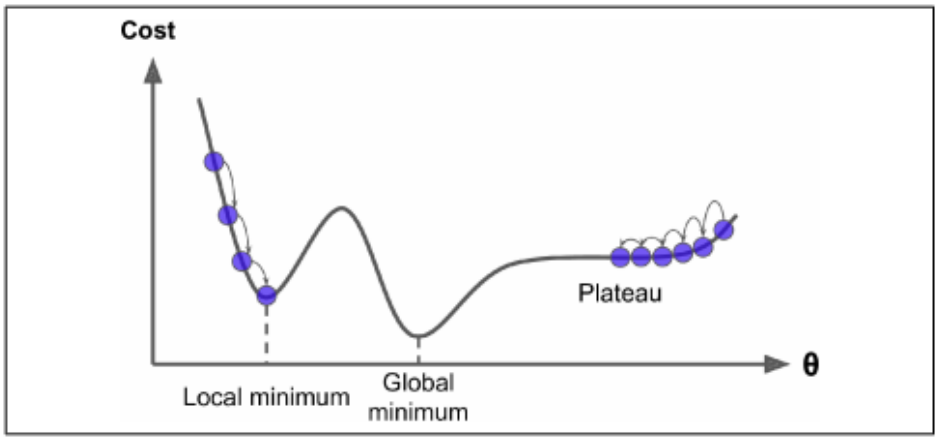


Figure 2.6: Image show gradient descent and the local minimum problem[14]

training. It is known to achieve good results and is faster than other optimizers. Adam uses estimations of the first and second moment, respectively the mean and the uncentered variance of the gradients, to adapt the learning rate [15].

Adam can be described as a "Heavy Ball with Friction since it averages over past gradients"[17]. The averaging acts as a velocity, which makes it avoid settling in local minimums. Adam is commonly used for anomaly detection[18, 12, 19].

Regularization

Regularization can be applied to avoid overfitting, making a model more robust and general in anomaly detection. When using regularization, the model loses some of its ability to fit the training data well, but it will be better to generalize on data it has not seen yet. It does it by adding a term to the loss function that penalizes for large weights.

Different types of regularization are L1 and L2 norms, and data augmentation [7].

The L1 norm regularization is the sum of the absolute value of weights, also known as the Manhattan distance. L1 is described as the equation [6]:

$$||x||_1 = \sum i|x_i| \tag{2.6}$$

L2 norm regularization is the squared value of weights, also known as the Euclidean Distance. The L2 norm is expressed as the equation[6]:

$$||x||_2 = \sum i|x_i|^2 \tag{2.7}$$

Dropout

Dropout is also a regularization method[6], that is implemented on a layer-basis. It avoids the network from overfitting the data by ignoring some of the neurons. The

neurons that are ignored are chosen randomly[11].

Data augmentation is also a regularization form. Data augmentation modifies the data to avoid biases by making small random changes to the data each epoch, based on the chosen data augmentation.

2.1.1 Network types

Deep learning can be categorized into two general types; Supervised and Unsupervised.

Supervised

Supervised learning is when the model is trained on labeled data, meaning that when the model is given data, it is also given a corresponding class or label. An example of some data could be an image of a dog or cat, and the corresponding label would then be "dog" or "cat." When the model is then given new data, it will try to classify it into one of the classes given by the labels. Common networks using supervised learning is Convolutional Neural Network(CNN) for classification.

Unsupervised

Unsupervised learning is a machine learning technique used when the data is not labeled, meaning that the model is not told what classes are in the data. The model will attempt to find structure from the data and extract the most valuable features that can be used to map the data. Common unsupervised networks are autoencoders and Generative Adversarial Networks (GAN).

2.2 Image Enhancement

With image enhancement, you can clean up data to be easier to use for further analysis. Image enhancement such as morphological operations, histogram equalization, or median filters can be used to enhance the contrast in an image or remove noise. It has before been used to pre-process data for the user to train a neural network. Toufique Ahmed Soomro et al. [20] show the effect of Contrast limited Histogram equalization on CNNModel for Retinal Blood Vessels Segmentation and found that they got better or comparable result compared to existing methods.

Histogram Equalization is used to enhance images. It does this by redistributing the image histogram so that all intensity values represent the whole image, thereby enhancing the contrast of the image.

Histogram equalization considers the global contrast of an image. Therefore, it is not a good choice when applied on an image in which histogram covers a more extensive region on the intensity spectrum and is better suited to be used on an

image where the intensity values of the image are located in a smaller region of the histogram.

Other variations of histogram equalization exist, such as adaptive histogram equalization (AHE), which splits an image into smaller tiles and applies histogram equalization on each tile separately. A problem with AHE is that it overamplifies the contrast in areas where the histogram is almost constant. Therefore, it may cause noise in the image to be amplified in these near-constant regions. Contrast limited adaptive histogram equalization (CLAHE) is a variation of AHE where the amplification of the contrast is limited(sometimes referred to as a clip-limit) to reduce the problem of amplifying the noise in the image. CLAHE has been used before to increase the contrast of images to make it easier for a CNN to extract features from dental x-rays [21].

2.3 Data Augmentation

Data augmentation is an easy and commonly used method to increase the size of a dataset used in neural networks and reduce overfitting. Data augmentation is done by making different label preserving augmentations to the training images in the dataset [22, 23, 24]. There is no scientific method mentioned of what data augmentations are needed [23, 22, 25, 24]. It is necessary to know what kind of data is being dealt with, how the data is captured, and its use to find the best data augmentations.

Label-preserving transformation

Since data augmentation is a good way to increase the dataset and to make a model more robust, it is necessary to consider what augmentation to use and how these would impact the data. In most cases, the augmentations have parameters to tweak and tune. These parameters need to be tuned in a way so that the image still represents the same concept and therefore keeping its label. Figure 2.7 shows an example from the MNIST dataset[26], where the rotation is taken too far, and the new image does not resemble the original image. Here the ”6” is turned into a ”9” by rotating the image 180 degrees, and it is therefore not a label-preserving transformation.

2.3.1 Data Augmentations based on basic image manipulations

This section will describe some of the most commonly used data augmentations and in which situations these can be useful. It is necessary to keep in mind that in most cases when applying data augmentation, the label must be preserved.

Translation is moving the image a certain amount in any direction[27]. This augmentation is very useful to avoid positional bias in the data. An example could be that tennis ball is at the center of the image in all the images. The model would

(a) Original image (b) Rotated 180 degrees

Figure 2.7: Example of a non label-preserving transformation

therefore need to be tested on centered images as well. Figure 2.8 shows a tennis ball that has been translated.

Figure 2.8: Example of translation data augmentation [28]

Cropping can be an efficient augmentation for images with mixed height and width by cropping an important region of an image. Cropping gives an effect that is a lot like translation due to the object of interest is moved because of the height and width being changed. However, ”Depending on the reduction threshold chosen for cropping, this might not be a label-preserving transformation.”[27], meaning if an image is cropped too much, the label can’t be placed on the object.

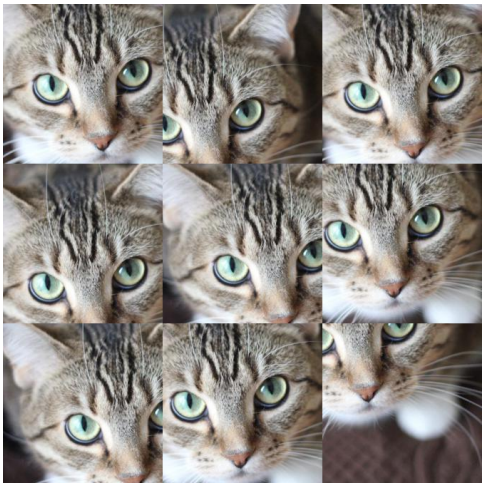


Figure 2.9: Example of cropping data augmentation [29].

Rotation augmentation is done by rotating the image around a point. However, as the rotation degree increases, the label for the data can be lost[27]. An example could be with digit recognition, and here a ”6” can be the same as a ”9” with a 90-degree rotation. This augmentation is useful if the object of interest is not always at the same angle and can, like translation, help to avoid positional bias.

Figure 2.10: Example of rotation data augmentation [29].

Flipping can be done either horizontally or vertically, with horizontally being more common than vertically[27]. Figure 2.11 shows an example of the flipping augmentation. On a dataset involving text recognition, flipping is not a label preserving transformation [27].

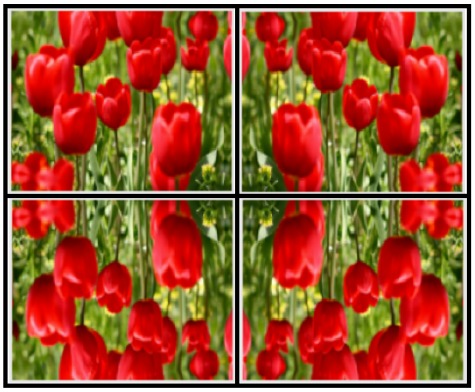


Figure 2.11: Flipping Augmentation. Original image is in the top left. In the top right, the image has been flipped horizontally. The bottom left has been flipped vertically, and the bottom right has been flipped vertically and horizontally [29].

Color space transformation is augmentations to the color space. The color space is how the colors in the image is encoded. One color space is the RGB color space, here the colors are divided into Red, Green and Blue. According to Shorten et al.[27] "Lighting biases are amongst the most frequently occurring challenges to image recognition problems.". It is therefore ideal to know what to do in order to avoid this problem.

An easy way to use the color space transformation would be to make the image brighter or darker. This can be done by adding or subtracting a constant from all the pixel values in the image. Addition and subtraction can also be made on a single color channel adding more value to that channel as seen in Figure 2.12.

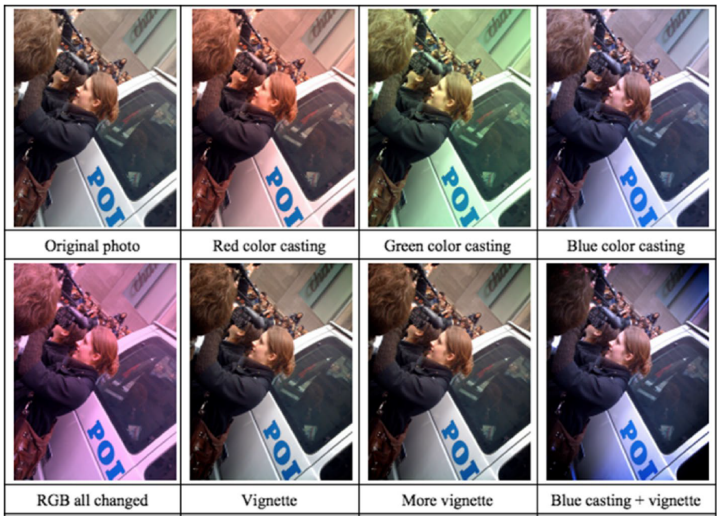


Figure 2.12: Examples of color augmentations [30].

Color space transformation can also be used to convert RGB images into grayscale. Combining the three channels into one, resulting in faster computation [27]. However Chatifled et al. [31] found that this would significantly decrease the classification accuracy. However there are also disadvantages to color space transformations done on images with multiple color channels. Each image takes up more memory, transformations are more computational heavy, increased training time, and it may discard important color information. In the case of discarding color information the transformation may not be label-preserving.

Perspective correction is correcting the view angle of an image so that the image appears to be taken straight in front of the object. Figure 2.13 shows how perspective correction can make license plates from vehicles easier to read, and thereby improving text recognition.

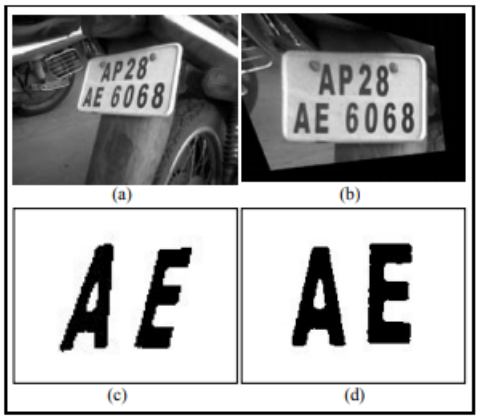


Figure 2.13: (a) Original image of a license plate. (b) Its perspective corrected version. (c) Two selected characters shown before correction and (d) The characters shown after correction. [25]

When working with images taken from cameras, these images often suffer from projective distortion[25]. In order to make up for this, the neural network would have to take in more parameters, thereby making it more complex. So by eliminating these distortions with perspective correction, it would be possible to reduce the complexity of the neural network. Perspective distortion of a plane can be interpreted as a generalized linear transformation of a plane. In order to correct the distortion, different clues can be gathered in the image, such as object boundaries and textual structure[25].

2.4 Segmentation

Segmentation is one of the larger areas in computer vision. To segment is to divide an image into two significant parts or more[32, 33]. There are two forms of segmentation; semantic and instance segmentation.

The goal of semantic and instance segmentation is to label individual pixels in an image into different classes. In recent years, progress in segmentation challenges is mainly done by CNNs.

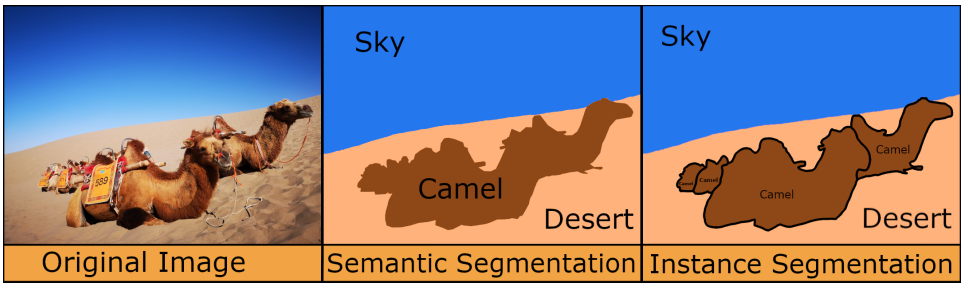


Figure 2.14: Example of semantic segmentation and instance segmentation. Even though there are more than one camel, they are all classified with the same label in Semantic segmentation but multiple camels in instance segmentation[34].

In Figure 2.14, it is shown what semantic and instance segmentation outputs. In the image, four camels are present, and some of them are obscuring each other. There are three classes: Sky, Camel, and Desert. Semantic segmentation labels each pixel as one of these classes. Despite there being multiple camels, they are all labeled as the same. This method has been used to help autonomous cars drive on the current road infrastructure[35], or in the medical field to help experts process medical images faster [36].

While instance segmentation assigns classes to pixels in the same way as semantic segmentation, it additionally identifies how many instances of an object of a specific class occur in an image. Instance segmentation is a two-part problem; Object detection and segmentation. In Figure 2.14, all the camels have their own separate contour, indicating that they are multiple instances of the class *camel*.

2.4.1 Segmentation Methods

This section describes several semantic segmentation techniques.

Threshold Method

One of the simplest methods for segmenting pictures is thresholding see Figure 2.15, where gray-scale image values are split into two; foreground and background. This is done by setting a threshold on the pixel intensity 0-255. If a threshold of 75 was set, it would imply that all pixel values underneath would be a background, and all pixel values above would be the foreground.

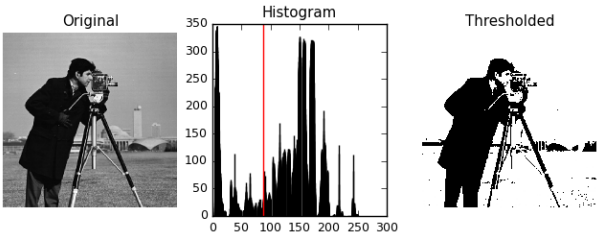


Figure 2.15: shows a example of thresholding using the Otsu method[37]

There exist two methods of thresholding; local and global. Local thresholding separates the picture into smaller fragments, and multiple thresholds split the fragments into foreground and background. Global thresholding has a single threshold for the entire picture. There are various methods for picking the threshold. One regularly utilized is the Otsu method[38]. The greatest advantage of thresholding for segmenting is that it is quick, no labels needed and good at segmenting if the foreground and the background have a significant pixel intensity difference[39].

Clustering Based Segmentation

Another segmentation method is Clustering-based methods (see Figure 2.16), for example, k-Means. K-Means is an unsupervised machine learning algorithm that segments images based on various classes. For example, in background and foreground segmentation, there would be two classes defined by the k-parameter (k = 2). The algorithm is unsupervised, meaning that labeling isn't necessary as K-means will attempt to cluster the data into classes. Having a k-parameter of two will make the K-Means algorithm pick 2 points as starting points. It then looks at which points are closest and move towards them. It will continue to do this until the best means of the two clusters are found, and a threshold can be chosen based on the two clusters' means [39].

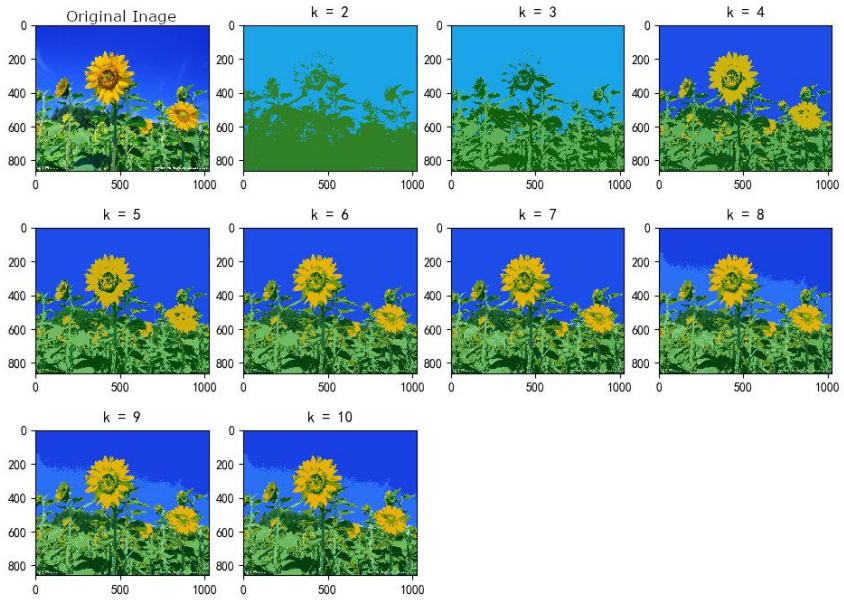


Figure 2.16: Shows what happen to an image if you chose a different k[40]

Region-Based Segmentation

The watershed algorithm is a region-based segmentation method (see Figure 2.17). Watershed sees the image as a topographic landscape and treats all intensity values in an image as a height; the higher the intensity, the higher the peak is, and the lower the intensity, the more profound the valley is likewise called the basin [41]. The watershed algorithm mimics the idea of water, filling up the valleys with the basins' absolute minimums as its origin point. The point of contact where two water sources meet is known as the ridgelines, which are utilized to isolate two objects and would be the edge of an object. Each body of water is then marked as different objects [41].

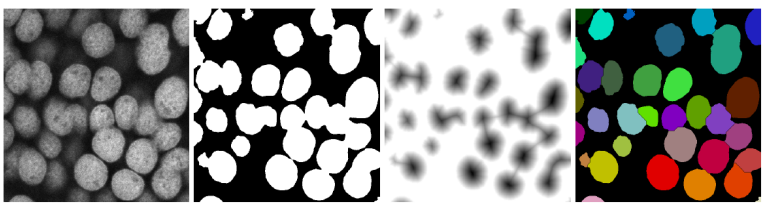


Figure 2.17: Show am example of watershed being used to segment nuclei from a confocal laser scanning[42]

Artificial Neural Network Based Segmentation

Newer approaches when segmenting images are utilizing CNNs, for example, U-Net, which was first utilized in 2015 for the division of biomedical images [13]. CNNs work by detecting patterns in an image with the assistance of its convolutional layers. Convolutional layers are comprised of various filters that detect patterns, for example, edge detectors and corner detectors, and some layers detect circles and squares. The deeper, the more sophisticated the layers become, for example, layers that can detect eyes or full shapes like animals [43].

2.4.2 U-Net

U-net is an encoder-decoder network that comprises of two parts 2.18; a contracting path where dimensionality is reduced (Here, the image resolution gets reduced), and this is the encoder part of the network. It uses a 3x3 kernel in convolution layers in this path with an activation function of ReLU and Maxpooling of 2x2. The second part, the decoder, is an expansive path that comprises upsampling with 2x2 kernel convolution layers. In this path, the images get upscaled, and the image from the contracting path gets concatenated. This is followed by two 3x3 kernel convolution layers with ReLU. The network can train on a few images compared to other networks and is fast to train on GPUs.

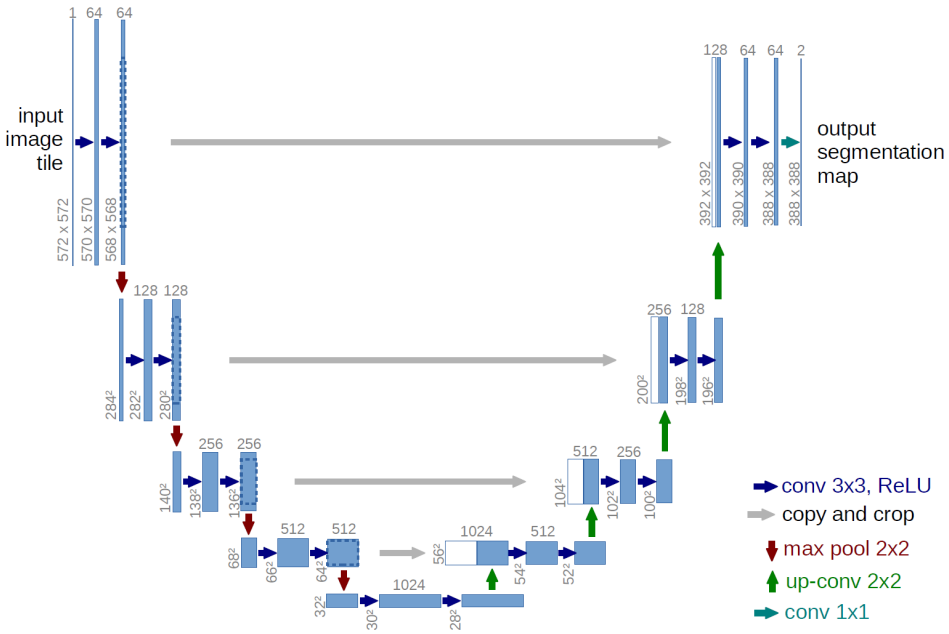


Figure 2.18: U-Net Architecture [13].

2.5 Anomaly Detection

Anomaly detection, also known as outlier detection or defect detection, is an unsupervised method of finding outliers from a distribution. Anomaly detection is unsupervised because the network is not passed any labeled data. It is usually trained on only data without anomalies.

2.5.1 Autoencoders

The autoencoder is a bottleneck architecture, as seen in Figure 2.19. The purpose of the encoder is to learn the representation of the input data in a smaller compressed state known as the latent space. This forces the encoder to only learn the most valuable features from that input data to later decode it as a reconstructed image.

Autoencoders can be expressed as

$$\bar{x} = D(E(x)), \tag{2.8}$$

where x is the input, E is the encoder part of the model, D is the decoder, and $\bar{x}$ is the reconstructed version of the input x [44].

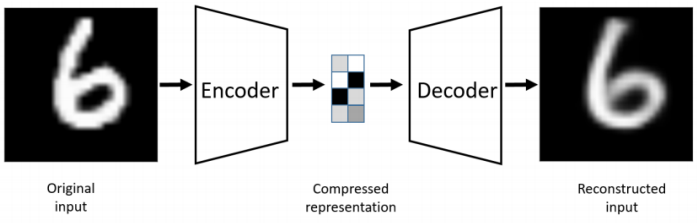


Figure 2.19: Autoencoder bottleneck architecture [45]

For calculating the reconstruction error, Mean-Square-Error (MSE)[12] have regularly been used. MSE is known as loss function or cost function. Commonly used optimizers have been Stochastic Gradient Descent (SGD)[46] and Adaptive Moment Estimation (Adam)[19, 12, 18].

Autoencoders with more than one hidden layer are usually referred to as deep autoencoders [44].

The chosen optimizer and loss function depends on the specific architecture and the activation used in the layers.

Autoencoders can be used for denoising[47], dimensionality reduction and anomaly detection[48, 49, 12].

For image data, convolutional autoencoders (CAE) uses a kernel to extract 2D feature data from the input images [50, 12]. When using images with large dimensions, it is impossible to use the whole image as input due to the limited GPU memory. Solutions have been to use overlap-tile strategies with extrapolation to stitch the images later seamlessly together [12]. Chow et al. [12] used a sliding

window of 256x256 and extrapolated samples to 320x320 using mirroring before normalizing (range -1 to 1) and inputting them to a CAE. Tsai and Jen (2021) [18] implemented a 13-layer CAE for anomaly detection for surface defect inspection, which can be seen in Figure 2.20. The CAE used MSE as loss function and the Adam optimizer. Their results showed that applying regularization to their model increased defect detection.

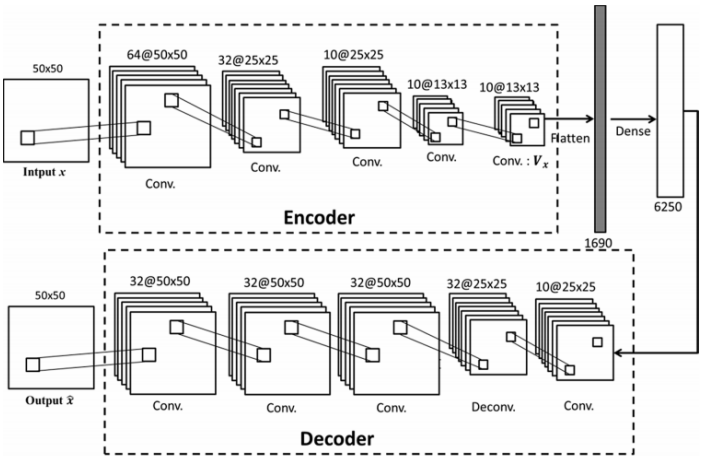


Figure 2.20: CAE for anomaly detection[18].

2.5.2 Generative Adversarial Networks

GANs are another network used for anomaly detection[46]. The network consists of two models; a discriminator and a generative model (see Figure 2.21). The network is referred to as 'adversarial' as the two models are opposing each other. The generator is trained on real data and receives randomly distributed noise as the input. The result is a generated image that is passed to the discriminator, trying to mimic real data. The discriminator model acts as a classifier, labeling the data to be real or fake. The model is successful if the discriminator cannot tell the generated fake image from the real image. The results are used to fine-tune the generator[51].

While GANs can create good results, they can be difficult to train in practice due to the high dimensional spaces, while AE and CAE are more straight forward [18].

2.5.3 Evaluating Anomaly Detection

For evaluating the performance of anomaly detection models, the most commonly used methods are Area under the ROC curve (AUROC) and area under the precision-recall curve (AUPRC) [19].

Dealing with skewed and imbalanced datasets, the AUPRC is preferred over AUROC [52]. In other cases, simply Precision, Recall, and F1 and F2-score are

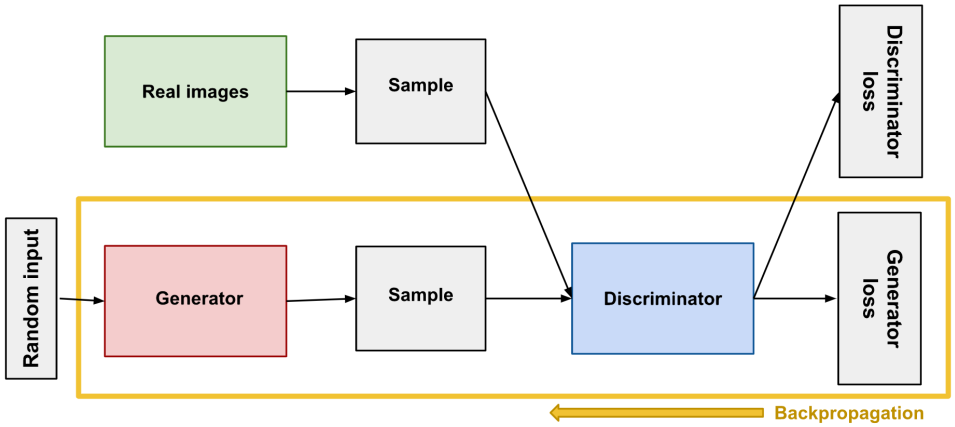


Figure 2.21: Generative Adversarial Network Architecture [51]

used as evaluating measure[12].

F1 and F2-scores are calculating with the formula:

$$F_\beta = (1+\beta^2)\frac{precision \times recall}{(\beta^2 \times precision)recall}, \tag{2.9}$$

where β is a positive real number that is a weight allocated to precision and recall. The weight in F2-score is added more to recall over precision, which puts more importance of determining real defects, by avoiding false negatives[12].

To display defects, anomaly maps have been used to inform inspectors of any possible defects. Chow et al.[12] used the reconstruction error (e):

$$e = \sum_{c=1}^{3} (p(x,y) - \hat{p}(x,y))^2, \tag{2.10}$$

where c is the channel, p is the input pixel, $\hat{p}$ is the reconstructed pixel. As they were working with RGB image and not grayscale images, each channel had to be summed to reach an anomaly score.

2.6 Synthetic Data

Industrial anomaly detection is a challenging task, not only with the lack of defects but also for evaluating defects. Evaluating what constitutes a defect can result in false negatives or false positives depending on the priority. In addition, the dataset can contain multiple categories of defects which makes it even more difficult for the model to train on with the already minimal dataset. Creating synthetic datasets can provide more information to the model for detecting defects and strengthening them. Synthetic Data is artificial data created to mirror the statistical properties of original data without revealing any sensitive information.[53][54] Synthetic data

could be utilized for instance-segmentation. Segmenting objects from an image can be troublesome, and one method for achieving good segmentation is by creating the mask by hand. This, of course, takes a lot of time and human resources. Therefore, a network trained on synthetic masks can predict the masks for real data and be utilized to eliminate the background and other unnecessary information in the images.

2.6.1 Methods for creating synthetic data

There have been different ways to create a synthetic dataset. One method could be to train a Generative Adversarial Networks(GAN). Jain et al. [55] trained different versions of GAN to generate new surface defect images in steel strips from random noise filters.The GANs was trained on augmented data so the networked learned to generated synthetic augmentation images for then to be tested on a Convolutional Neural Networks(CNN), to see which GAN generated synthetic data improved the network the most. Three different GAN models were made to create new synthetic data, first is the Deep Convolutional GAN (DCGAN). The benefit for this network is in the discriminator, which excels in feature extraction. This can improve the training stability and the quality of generated images compared with a normal GAN. A result of generated synthetic data from the DCGAN can be seen in figure2.22. Another GAN used to generate data was the Auxiliary Classifier GAN (ACGAN), which allows the model for conditioning on information obtained from a provided class label. This makes the network able to incorporate the provided class labels from its training to produce labeled image samples for supervised data, which improved the image sample quality. The last GAN used is the Information-theoretic GAN (InfoGAN). The benefit of InfoGAN is that the generator input is split into two parts: the noise vector and the latent code vector (also called the bottleneck). This makes the InfoGAN capable of separating the generated defects from background and therefore changing the amount of defect should appear in the synthetic data generated images. The three networks was then evaluated on the accuracy of the CNN to predict the right classes of defects, against the classic data augmentation such as flipping, rotation, and scaling. The result for the classic augmentation had a accuracy of 90.28% then compared with the three networks where DCGAN had a accuracy of 95.78%, ACGAN had a accuracy of 92.78% and InfoGAN with a accuracy of 94.86%. even tho there is an improvement in accuracy the findings of this study suggest that a GAN-based data augmentation can offer an advantage, in not have the opportunity to collect new data but also reducing the time it takes for collection it.

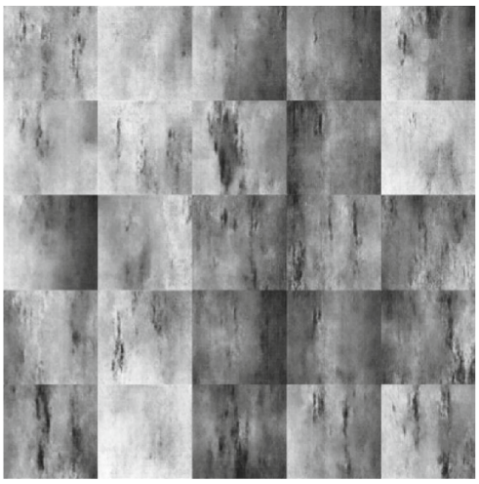


Figure 2.22: Synthetic data generated from DCGAN from the study[55]

Ekbatani et al.[54] trained a Deep Convolutional Neural Networks(DCNN) to count people in a crowd. Synthetic data was then introduced to increase the dataset and at the same time protect privacy and confidentiality compared to using real data. The dataset contains clips of groups of people walking towards and away from the camera, consisting of 34 training video samples and 36 testing video samples. All video samples were used to generate the synthetic dataset. The method was split into five parts. First, background extraction, where they extracted the background in the videos, then extracted the pedestrians morphological labeling methods. With the extracted backgrounds, they then generate their own backgrounds, and to make them more realistic. They apply image augmentation such as changing the global illumination of the images randomly and adding some random Gaussian noise to the backgrounds. They then applies a filter to show only the region of interest(ROI) in the images. Then adds pedestrians to the synthetic images which got normalized between 0 and 255 and resized to match the network. These five steps can be seen in Figure2.23. the study concludes that a DCNN trained on synthetic a highly realistic synthetic dataset of pedestrians in a walkway, synthetic data generated by Ekbatani et al. had a lower mean absolute error(MAE) 0.707 and mean squared error (MSE) 0.942 compared to synthetic dataset generated by another study (Segu´ı et al., 2015)[56] where the MAE 0.74 and MSE of 1.12.

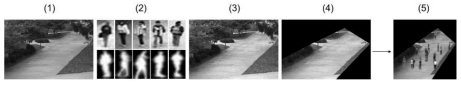


Figure 2.23: Steps that was made to generated the synthetic data of pedestrians in [54]

Another approach is to make the synthetic from 3D models. Ward et al. [57] implemented an automated segmentation to find phenotypic traits in plant leaves.

They tested if there is an improvement between real-world data and synthetic data to train and segment plant leaves. The synthetic leaf is made in a free 3D software tool called Blender[58]. The approach for the synthetic data was to make a *inspiration leaf* which was made in Blender with manipulation points. Each generated leaf is then scaled along its axis individually and applied random leaf texture this can be seen in Figure 2.24. Before rendering the synthetic data image, there is applied a random background behind the leaves. As for the camera and lighting, the camera angle was from a top-down view and a single light source, there was a variation in the position of both camera and light. With the synthetic data generated, the evaluation then consisted of training a Mask-Region Based Convolutional Neural Networks(RCNN) on different versions of synthetic data and real. The best result for segmenting phenotypic was by training on a combination of synthetic data and real data.

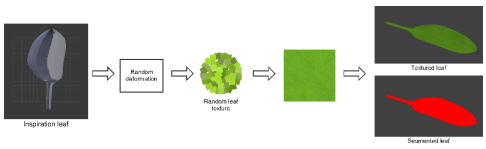


Figure 2.24: Stages for rendering each leaf

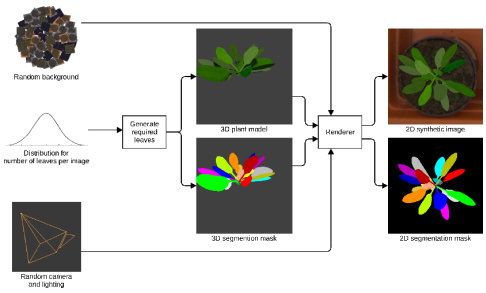


Figure 2.25: Pipeline for generating synthetic 2D images of the leafs

2.7 Summary of findings

- It is possible to increasing the size of a dataset with the use of data augmentation or with synthetic data by using either a neural network or a graphical program. Thereby answering the first research question: *"How can we artificially generate more data using data augmentation or other methods?"*
- In data augmentation, it is important that the augmentation is label preserving, and knowing about you data to chose the correct augmentations in order to make the data invariant for certain biases. Thereby answering the second research question: *"How can the system be made robust towards lighting, perspective, and placement changes?"*

- Image segmentation is done mainly by CNN in state of the art, but other methods such as thresholding can be used if the data allows it. This answers the third research question: *"How can the background of a dataset be segmented automatically?"*
- Extrapolation should be used to avoid border-effects.
- MSE and Adam are the most common loss function and optimizer.
- L1 norm, L2 norm, dropout, and data augmentation are used to avoid overfitting and can make the model more generalizing.
- F2-measure puts more weight on recall to avoid false negatives and defects.
- GANs can give good results, but very hard to train in practice, where Convolutional Autoencoders are easy and faster to train.

Chapter 3

Design and Implementation

This chapter covers the implementation process and design choices that has been made throughout the implementation of this project with explanation on the tools used and a description of the setup. Throughout the process of implementing this project, we have made use of internal testing.

3.1 Overview of the whole system

The process of anomaly detection in this project works as a pipeline. The pipeline will take in the raw data given to us by our collaborators and afterward segment the background out from the images, thereby eliminating most of the unnecessary information in the images. The data with no background will be fed into the anomaly detection model and checked for anomalies. The output of the anomaly detection would a reconstructed image and a picture which shows where in the image the anomalies are found, along with an anomaly score which can be utilized to get an idea to which batch has the most anomalies in it. Figure 3.1 shows the proposed system. Two squares show the networks for training the models utilized in the primary system to segment and detect anomalies. Example images of input and output are likewise shown.

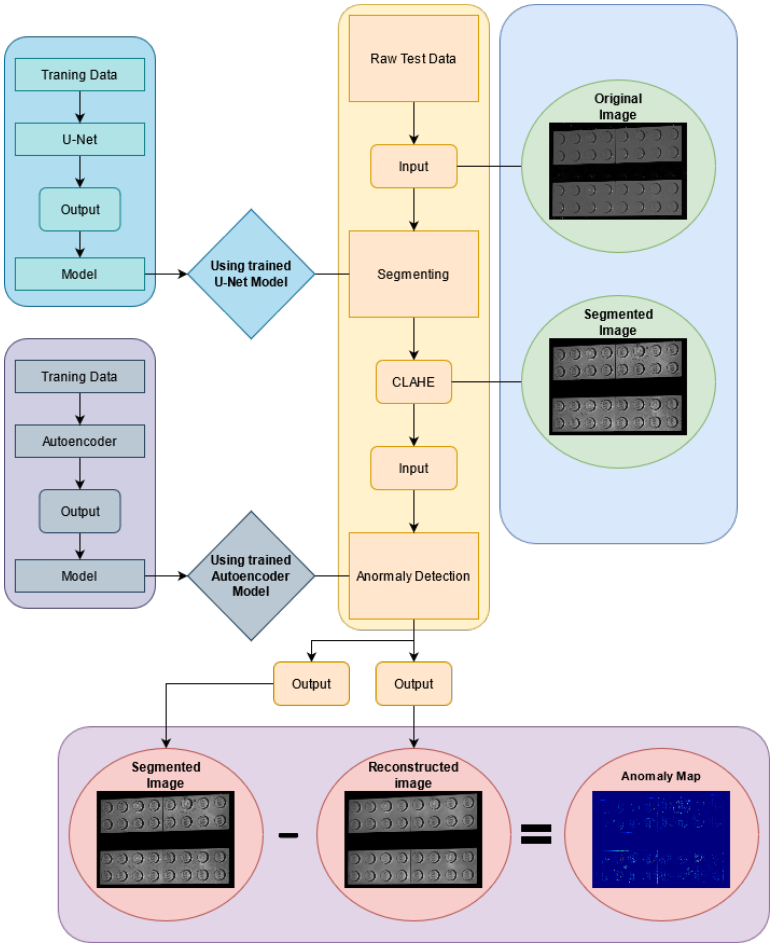


Figure 3.1: Overview of the process

3.2 Tools

During this project, different tools and software has been utilized. The tools and software used will be described in this section.

3.2.1 Blender

The process for creating our synthetic data was decided to be made in Blender since we received a Blender project with a 3D Computer-aided design (CAD) model of the bricks from our collaborator.
Blender is a free, open-source 3D creation program that supports the possibility of modeling, rigging, animation, simulation, rendering, compositing, and motion tracking. The rendering feature is the process of making a 3D scene into a 2D image, and Blender includes three different rendering engines: Eevee, Cycles, and workbench. Blender also gives the user the possibility to download other third-party rendering engines through the add-on plugin. Eevee is a physically-based real-time rendering engine. It focuses on speed and interactivity while achieving the goal of rendering physically based renderer(PBR) materials. This makes it great to use for previewing the materials in real-time. Eevee uses a process called rasterization that turns vector graphics shapes into raster images. So through numerous algorithms, it estimates how light interacts with objects and materials, but Eevee has many limitations compared to Cycles because of this.

Cycles is a physically-based path tracing. It casts rays from each pixel of the camera into the scene to collect information for objects it reflects, refract, or gets absorbed by until they either hit a light source or reach their bounce limit. Cycles also fire additional randomized rays from the same pixels called samples and average the results over time. Cycles, compared to Eevee, use ray-tracing in its render engine for this task. Cycles makes the rendering a lot more photo-realistic. Lastly, the workbench rendering method is not meant for final rendering but for displaying a scene in the 3D viewport as it is optimized for fast rendering. The rendering engine has mostly the same parameters that define the outcome of the rendering, which are cameras, lights, and material. These parameters are also shared between Eevee and Cycles. [59, 60]

Figure 3.2: Blender UI and functionality

3.2.2 TensorFlow

TensorFlow is an open-source software library used for machine learning, such as training different deep neural networks. The Google Brain team developed it for internal use but was released under the Free Apache License 2.0[61] in 2015. TensorFlow can run on multiple CPUs and GPUs. GPUs are better for training a deep neural network model because they are good at handling simple calculations but can calculate many of them in parallel. A CPU is best at handling single but more complex calculations one after the other. Because GPU is so much faster when training, segmentation, and anomaly detection models will be using GPUs to train, an example of the difference between the performance of CPUs And GPUs can be seen in Figure 3.3.

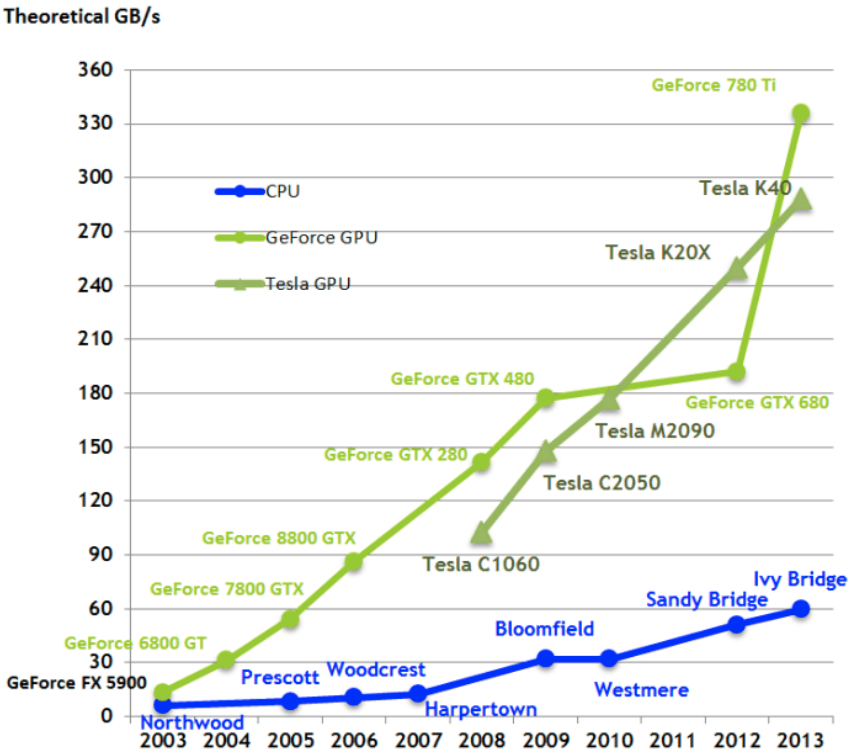


Figure 3.3: Transfer speeds for CPUs and GPUs[5]

3.2.3 Google Colab

Since we only had access to our personal computers to train our models, we investigated online browser solutions such as google Colab, a free service provided by Google. Google Colab can run python code in the form of Jupiter Notebook directly from the browser. Google Colab runs python 3.7.10 at the time of this report. Resources at Google Colab might not always be available for users. Google Colab gives access to one of the graphical processors available at the time. It can

be Nvidia K80s, T4s, P4s, or and P100s. There is no controlling which of the processing units get connected to the session at any given time when using their services[62].

The code seen in Listing 3.2.3 was used to see the amount of GPU utilized at the time and to print the device information. An example of the device information can be seen in Figure 3.4.

```
!ln -sf /opt/bin/nvidia-smi /usr/bin/nvidia-smi
!pip install gputil
!pip install psutil
!pip install humanize
import psutil
import humanize
import os
import GPUtil as GPU
from tensorflow.python.client import device_lib
# Prints devices that are avalible to train on GPU/CPU
print(device_lib.list_local_devices())
GPUs = GPU.getGPUs()
#Check for free GPU
gpu = GPUs[0]
def printm():
 process = psutil.Process(os.getpid())
 print("Gen RAM Free: " + humanize.naturalsize( psutil.
    virtual_memory().available ), " | Proc size: " + humanize.
    naturalsize( process.memory_info().rss))
 print("GPU RAM Free: {0:.0f}MB | Used: {1:.0f}MB | Util {2:3.0f}%
     | Total {3:.0f}MB".format(gpu.memoryFree, gpu.memoryUsed, gpu
    .memoryUtil*100, gpu.memoryTotal))
printm()
```

```
[name: "/device:CPU:0"
device_type: "CPU"
memory_limit: 268435456
locality {
}
incarnation: 7119588184595433954
, name: "/device:GPU:0"
device_type: "GPU"
memory_limit: 15703311680
locality {
  bus_id: 1
  links {
  }
}
incarnation: 953731231542289705
physical_device_desc: "device: 0, name: Tesla P100-PCIE-16GB, pci bus id: 0000:00:04.0, compute capability: 6.0"
]
Gen RAM Free: 12.3 GB  | Proc size: 921.2 MB
GPU RAM Free: 15931MB | Used: 349MB | Util   2% | Total 16280MB
```

Figure 3.4: Show the information printed form the code in Listing 3.2.3

3.3 Data Description

The dataset given to us by our collaborator consists of 10 batches, each with 24 gray-scale images from different viewpoints. One of the batches contains defects on all the bricks, while the rest are non defect. An example of a non-defect batch can be seen in Figure 3.5.

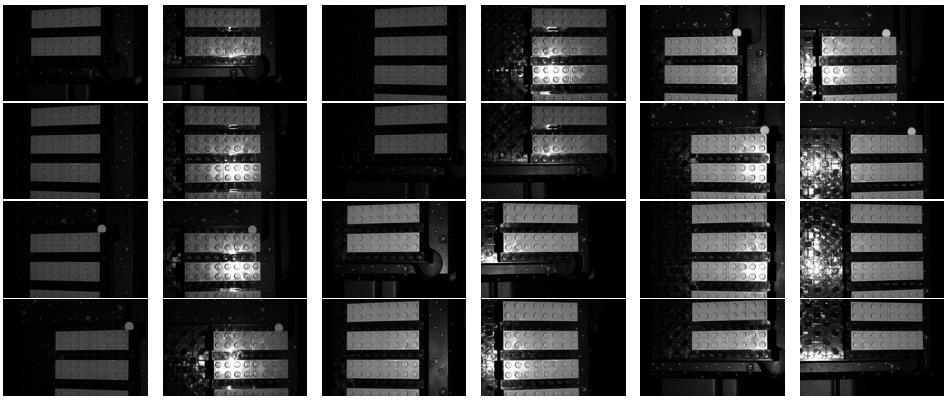


Figure 3.5: An example of a non-defect batch with the 24 images from different view angles.

Image acquisition

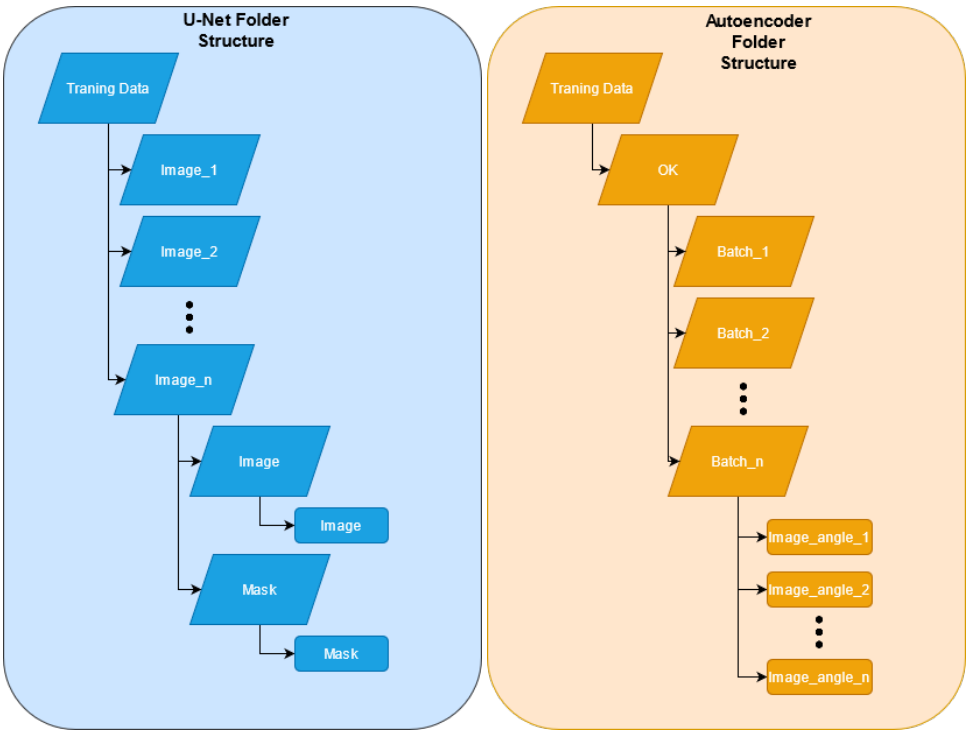
Our collaboration partner collected and provided the data in this project. The data was recorded by placing polymer surface elements on a plate by hand and placing the plate in an ATOS ScanBox series 4[2], where an ATOS capsule scanner[3] rotated around the polymer surfaces taking gray-scale images. The scanner has a 120 mm measurement volume. The images are taken from 12 different positions. The two images are taken from two cameras being roughly 5 cm apart as on the ATOS Capsule for each position. However, the exact angles the images are taken from are unknown.

3.4 Data Preprocessing

Before the data can be put into the network it is necessary to prepare the data. This is done by making sure the folder structure is correct for the network to take in and with the use of Contrast Limited Histogram Equalization. Before the data can be put into the U-Net, true masks also had to be created for the data, these are done in hand using GIMP [63].

3.4.1 Folder Structures

For this project, two separate neural networks were made. One for U-Net and one for the autoencoder. As the use of these networks are different and need different input, the folder structure for the two are therefore different. Figure 3.6a shows the folder structure for the U-Net and Figure 3.6b shows the folder structure for the autoencoder.



(a) Folder Structure used when training U-Net

(b) Folder structure used when training Autoencoder

Figure 3.6: Folder structures for neural networks used in this project

3.4.2 Contrast Limited Histogram Equalization

In the past project, we saw improvements in the performers of the defect detection network's ability to find defects in the images when we applied Contrast Limited Histogram Equalization (CLAHE) to the images. A code example using OpenCV[64] can be found in listing 3.1. This code was also used to get the histograms, as shown in Figure 3.8 and Figure 3.9. They show the change of distribution of the pixels for an image with a specular highlight and a dark image.

When also counting the background pixels in the image, we would get histograms that did not represent the distribution of the pixels properly. That is why we chose not to count the black pixels in the image when computing the histogram of different images. An example of the change can be seen in Figure 3.7.

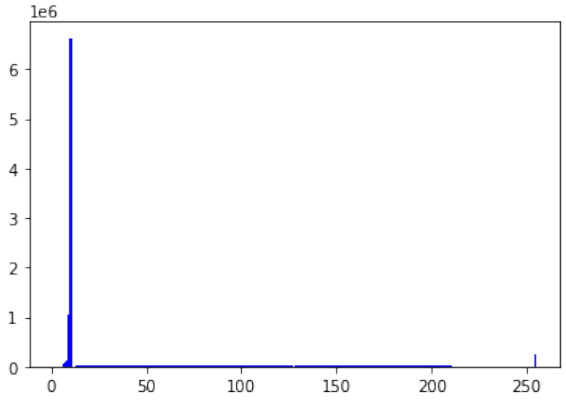
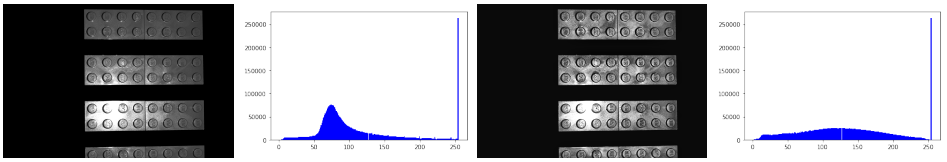


Figure 3.7: Example of a histogram where we count the zero-value background pixels in the image. Notice that it is hard to see the non-zero values because of the large number of background pixels.

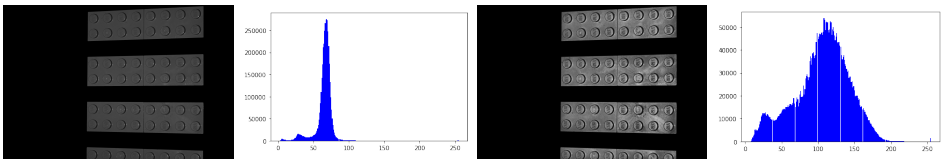
```
import cv2
import numpy as np
import matplotlib.pyplot as plt
from google.colab.patches import cv2_imshow

ImagePath = '/content/Masked_Light_Grey_Brick_Angle_19_Batch_1.png
    '
MaskPath = '/content/Light_Grey_Brick_Angle_19_Mask.png'


img = cv2.imread(ImagePath,0)
finalMask = cv2.imread(MaskPath,0)

CLAHE = cv2.createCLAHE(clipLimit = 50, tileGridSize = (16,16))
CLAHEimg = CLAHE.apply(img)

MaskedImage = cv2.bitwise_and(CLAHEimg, CLAHEimg, mask=finalMask)
cv2_imshow(MaskedImage)

# plt.hist(MaskedImage[MaskedImage != 257], color="blue", bins
    =256)

plt.hist(MaskedImage[MaskedImage != 0], color="blue", bins=256)

plt.figure
plt.show()
```

Listing 3.1: Contrast Limited Histogram Equalization Code



(a) Input Image (b) Histogram for Input Image (c) CLAHE Image (d) Histogram for CLAHE Image

Figure 3.8: When applying CLAHE to an image with an area of high-intensity light, the pixel distribution changes to be flatter which gives the image a higher uniform contrast.



(a) Input Image (b) Histogram for Input Image (c) CLAHE Image (d) Histogram for CLAHE Image

Figure 3.9: When applying CLAHE to a low light image, the pixel distribution changes to be flatter which gives the image a higher uniform contrast

Contrast limited histogram equalization was applied to improve the result for the anomaly detection as a preprocessing step. It was done when the masks generated by the U-Net model were applied to the image data. When first applying CLAHE, it was noticed that it affected the background of the images. The background would become lightly more grey of about 10-pixel intensity value. The masks were applied again to remove the change in the background caused by applying CLAHE, and it can be seen in Listing 3.2 on line six to nine.

```
1       CLAHE = cv2.createCLAHE(clipLimit = 9, tileGridSize = (9,9))
2
3       # APPLY MASK OF THE WHOLE DATA AND APPLY (CLAHE)
4
5       MaskedImage = cv2.bitwise_and(OriginalImage, OriginalImage,
    mask=Mask)
6       MaskedImage = cv2.cvtColor(MaskedImage, cv2.COLOR_BGR2GRAY)
7       CLAHEimg = CLAHE.apply(MaskedImage)
8       MaskedImage = cv2.bitwise_and(CLAHEimg, CLAHEimg, mask=Mask)
9
10      # USED IF WE WANT TO SPLIT THE BRICKS
```

Listing 3.2: CLAHE

3.5 Creating synthetic data

As mentioned the synthetic data was created in Blender, using the CAD model received from our collaboration partner. To make the synthetic data similar to the real data, we added a spotlight that pointed at the bricks in the scene, which simulated the spotlight source from the real data. We got the information from the collaborators that an ATOS GOM scanner[3] was used to take the images. The ATOS GOM scanner contains two cameras, and therefore two cameras were added to the scene that both looked at the same spot to simulated the ATOS GOM scanner (see Figure 3.10). Bezier curves were placed, which the cameras would follow to get different angles. The Bezier curves were made to simulate the movement of the scanner looking at the bricks from different angles, and an example can be seen in Figure 3.11.

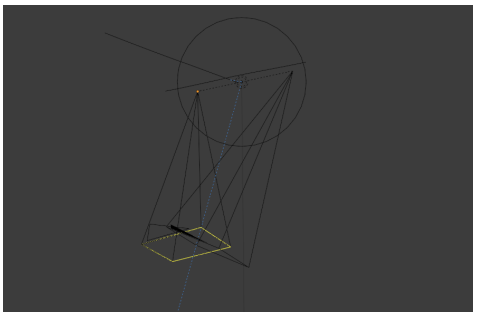


Figure 3.10: The Setup of the ATOS GOM Scanner in Blender

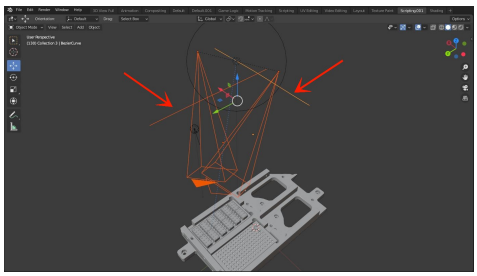


Figure 3.11: The two bezier curves where the cameras are attached to

With the cameras attached to the bezier curves, the animation properties "follow path" and "look at" were applied to the cameras. The "follow path" property moves the cameras according to animations within the keyframes specified (See Figure 3.12), and the "look at" property makes sure the cameras always focus an point in the scene, in our case the brick model. With the cameras attached, they will follow along the Bezier curve in the animation.

Figure 3.12: The animation window with keyframes

The Blender project contained a lot of polygon information, which caused it to be slow or not respond, since it was run on personal computers. To combat Blender not responding, the number of faces was reduced by decimating all objects in the scene. Decimate is a modifier that reduces the vertex/face count of a model with as minimal changes to the model as possible. The total number of faces was reduced

from 36.168.875 to 74.083 faces by keeping only ten bricks and massively reducing the number of faces on the background brick holder, as shown in Figure 3.13. Each brick got a slight reduction in faces as we still wanted the quality to be as high as possible for the bricks.

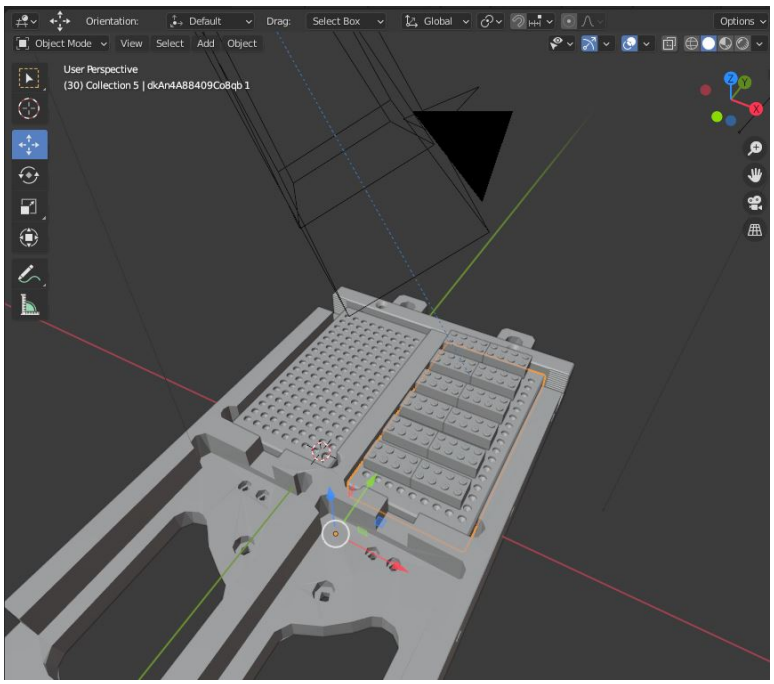


Figure 3.13: Showcasing the models that is in the scene with both the LEGO bricks and the holder for them from the real scanner

The chosen rendering engine was Cycles, as it gives the most photo-realistic renders compared to the other engines. The images were rendered in 4248x2832 resolution matching the size of the data.

To control the rendering process, a python script was created to automatically render the images across the Bezier curve, as seen in Listing 3.3. Additionally, masks were rendered per image, where the bricks were added to a layer, and its alpha channel was rendered as a binary image. All masks and original images were saved to separate folders to be later used to train a segmentation network.

```
import bpy
import os
import time

counter = bpy.context.scene.frame_start
framenumber = 20 #every x frame will there be generated an images

start = time.time()
path_dir = bpy.context.scene.render.filepath #save for restore
print('Taking pictures')
```

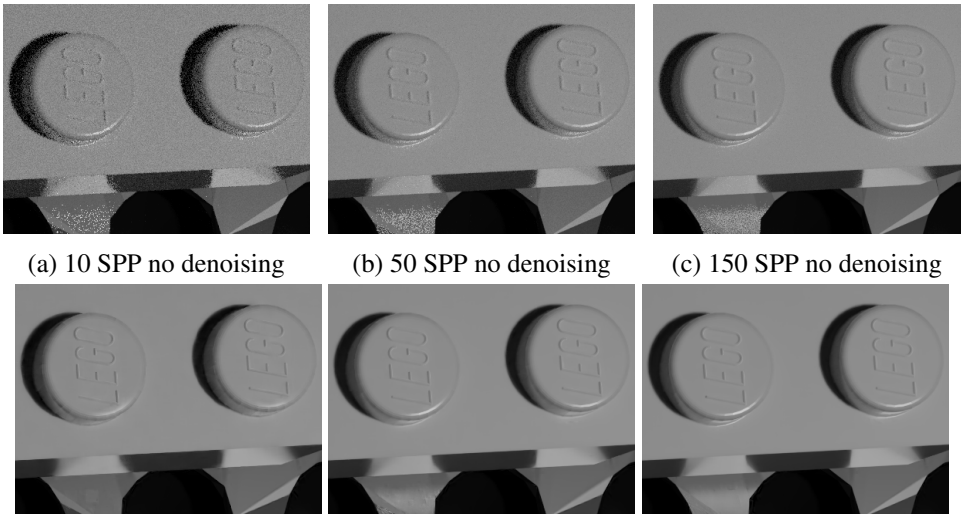
```
for f in range(bpy.context.scene.frame_start, bpy.context.scene.frame_end+1):
    if counter >= framenumber:
            print(f, bpy.context.scene.frame_end)
            bpy.context.scene.frame_set(f)
            for cam in [obj for obj in bpy.data.objects if obj.type == 'CAMERA']:
                print(cam.name)
                bpy.context.scene.camera = cam
                bpy.context.scene.render.filepath = os.path.join(path_dir, cam.name + str(f))
                bpy.ops.render.render(write_still=True)
                bpy.context.scene.render.filepath = path_dir
                counter = 0
                print("done did it")


    counter +=1

print( "Elapsed time",  time.time()-start)
```

Listing 3.3: Blender Rendering script

Different samples per pixel(SPP) renders were compared to find the optimal render time without sacrificing the quality. The chosen SPP were 10, 20, 50, 100, and 150 SPP, and their renders can be seen in Figures 3.14a, 3.14b and 3.14c. It was decided on using the 50 samples because it kept most of the highlights compared to the lower SPP images. Increasing the SPP over 50 samples did not add enough visual fidelity to the images to be worth the additional render time, as 50 SPP took 4 minutes and 18 seconds per render while 150 took 9 minutes and 20 seconds per render, rendered on a NVIDIA GeForce 1060 6GB graphics card. All the times can be seen in Table 3.1. A Non-Local Means(NLM) denoising filter was applied to remove some of the noise on the rendered images. This effect can be seen in Figures 3.14d, 3.14e and 3.14f.

SPP	Time
10	2.14 minutes
20	2.39 minutes
50	4.18 minutes
100	6.19 minutes
150	9.20 minutes

Table 3.1: Time it takes to render one angel for different samples per pixel



(a) 10 SPP no denoising (b) 50 SPP no denoising (c) 150 SPP no denoising

(d) 10 SPP with denoising (e) 50 SPP with denoising (f) 150 SPP with denoising

Figure 3.14: Comparison for different samples per pixels with and without denoising

3.5.1 Data Augmentation

To choose the most fitting data augmentation, we need to know more about our data. Other than knowing that the images are gray-scale and taken from different angles, we know from our collaborator, who collected the data, that the polymer bricks were placed by hand. This means that there are slight differences in position and rotation. In Figure 3.15 it can be seen how the differences are highlighted around the edges due to the slight positional difference.

Figure 3.15: Image of absolute difference between two images

Based on the knowledge from the data, we have to make networks positional and rotational invariant, as we do not know where the position and rotation of the bricks will be.

The augmentation we are using is therefore:

- **Translation** to get it positional invariant
- **Rotation** to get it rotational invariant

Implementation of Data augmentation

The data augmentation is implemented using the image data generator function from Tensorflow-library. The image data generator *"Generate batches of tensor image data with real-time data augmentation"*[65]. The selected data augmentations are specified as arguments. For rotation and translation, the arguments are *rotation_range*, *width_shift_range*, and *height_shift_range*. The *rotation_range* takes an integer value representing the maximum amount of rotation that can be applied. The *width_shift_range* and the *height_shift_range* determine how much random translation is added to the horizontal translation and the vertical translation. This is done with a integer value representing the max amount of pixels that the image can be translated.

3.6 Segmentation

Since we are only interested in finding defects in the bricks, we see the background as unwanted noise in the images. Therefore our segmentation aims to eliminate the background of the images and consequently remove noise. We are not interested in the individual objects or their location in the images, and we, thus, focus on semantic segmentation as a segmentation method. This section implements various methods to see what results we get utilizing two images with a low and high specular highlight.

3.6.1 Testing Segmentation Methods

This section covers the test of different segmentation methods **Thresholding** A quick test was performed utilizing ImageJ[66] and the Plugin Auto Thresholding[67] to see what sort of masks global and local thresholding would create, which can be found in Figures 3.16 3.17 3.18 3.19

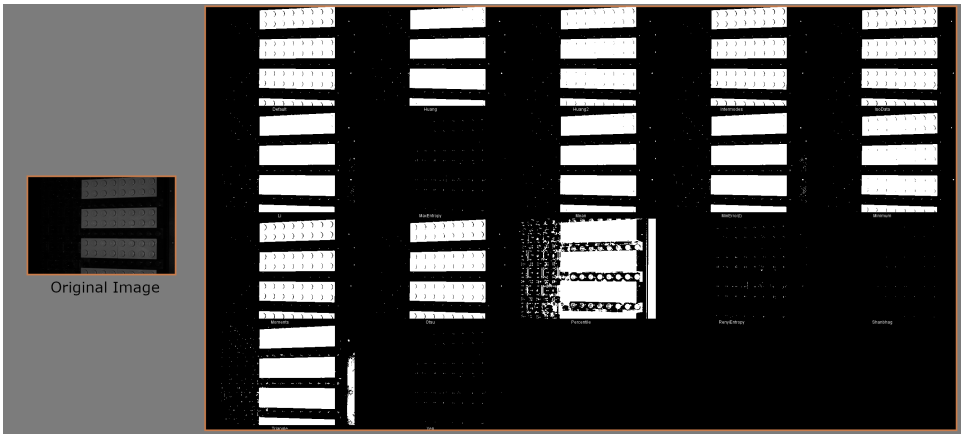


Figure 3.16: The original image is an example taken from our data with low light. Different Global thresholding methods were applied using ImageJ[66] and the plugin Auto Threshold[67]. The method used starting from the left is Default, Huang, Huang2, Intermodes, IsoData, Li, MaxEntropy, Mean, MinError(i), Minimum, Moments, Otsu, Percentile, RenyiEntropy, Shanbhag, Triangle, and Yen.

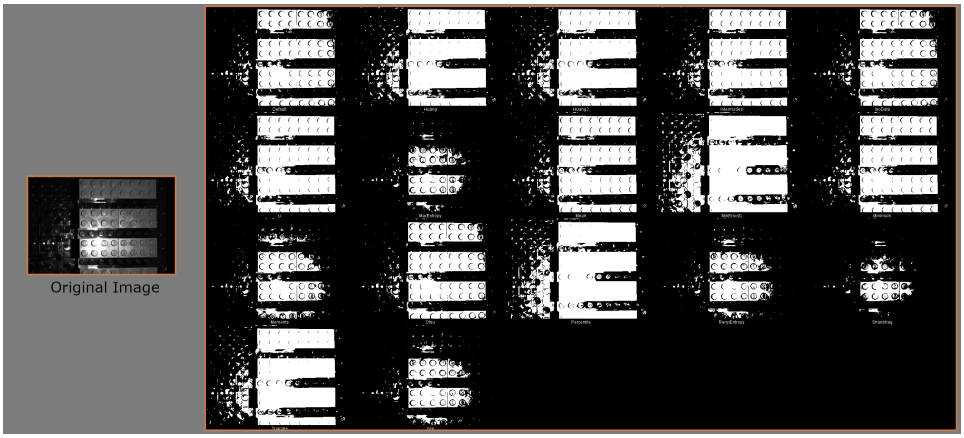


Figure 3.17: The original image is an example taken from our data with intense specular highlights. Different Global thresholding methods were applied using ImageJ[66] and the plugin Auto Threshold[67]. The method used starting from the left is Default, Huang, Huang2, Intermodes, IsoData, Li, MaxEntropy, Mean, MinError(i), Minimum, Moments, Otsu, Percentile, RenyiEntropy, Shanbhag, Triangle, and Yen.

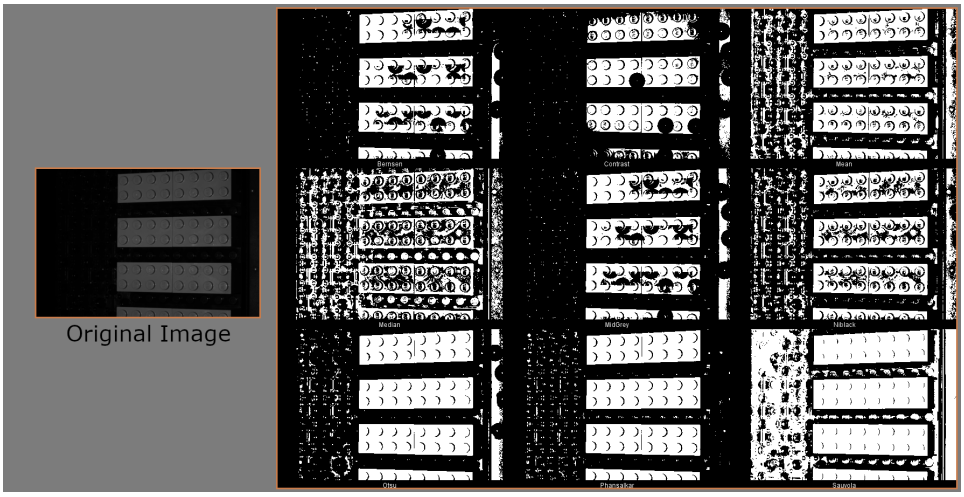


Figure 3.18: The original image is an example taken from our data with low light. Different local thresholding methods were applied using ImageJ[66] and the plugin Auto Threshold[67]. The method used starting from the left is. Bernsen, Contrast, Mean, Median, MidGrey, Nibblack, Otsu, Phansalkar, and Sauvola.

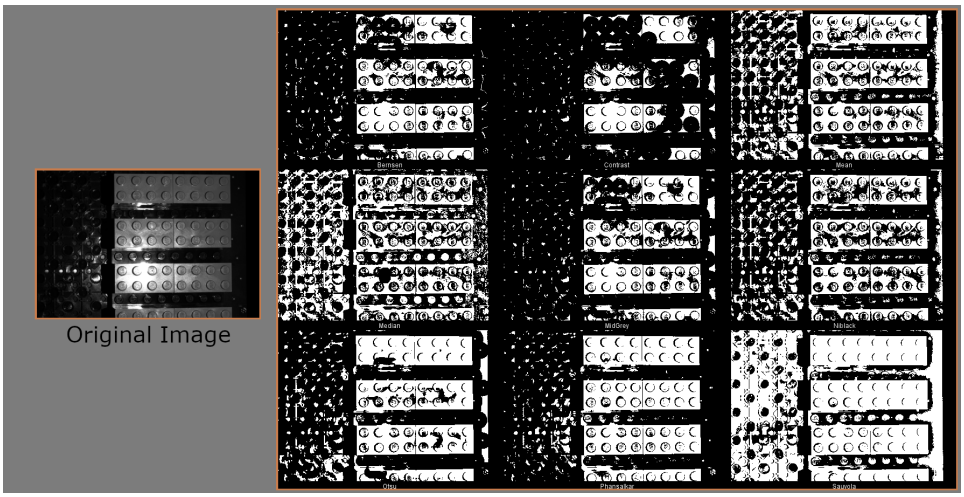


Figure 3.19: The original image is an example taken from our data with intense specular highlights. Different local thresholding methods were applied using ImageJ[66] and the plugin Auto Threshold[67]. The method used starting from the left is. Bernsen, Contrast, Mean, Median, MidGrey, Nibblack, Otsu, Phansalkar, and Sauvola.

Both global and local thresholding perform well on the image with low light, and a decent mask could be made with the assistance of some morphology operation like closing. However, it is not easy to get a proper mask from doing the same on the picture with a specular highlight.

K-means A small test of K-means was conducted utilizing OpenCV[64] with k=2, and random initial centers were chosen for each iteration. The test was done to see what kind of result we would get on the various images in the dataset. The result of the test can be found in Figures 3.20 and 3.21.

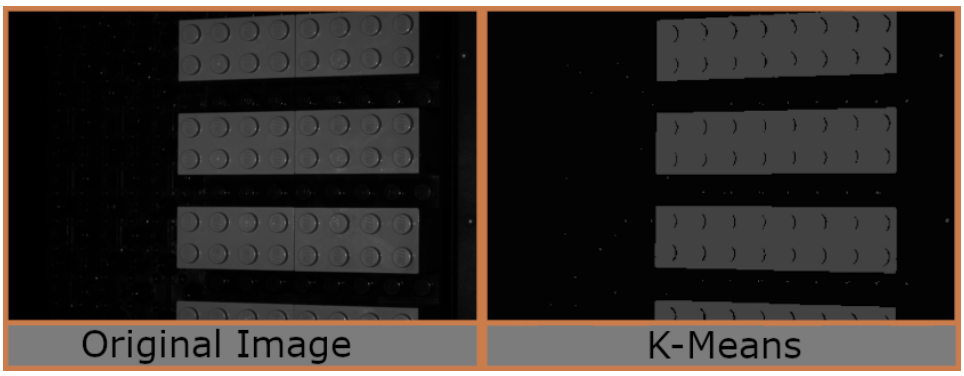


Figure 3.20: The original image is an example of a low-light image from the dataset. To the left is the effect of utilizing OpenCV[64] K-Means where k=2.

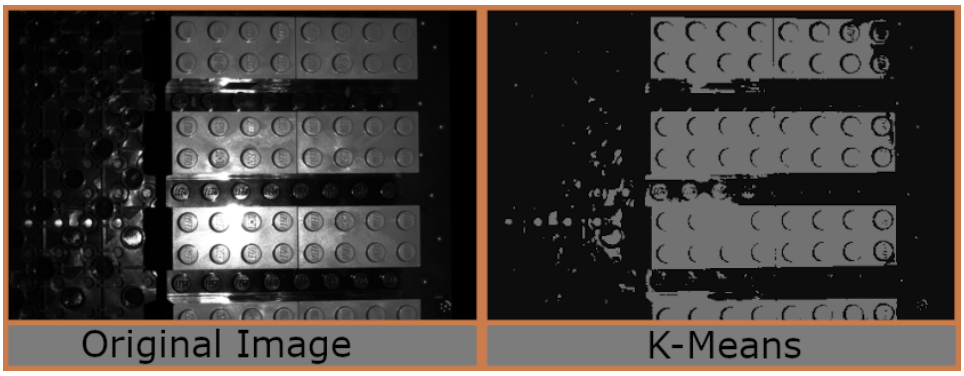


Figure 3.21: The original image is an example of an intense specular highlight image from the dataset. To the left is the effect of utilizing OpenCV[64] K-Means where k=2.

As seen with thresholding, k-means seems to have a problem getting good results on the image seen in Figure3.21. Nevertheless, it has no problem with the image with low light seen in Figure3.20.

Watershed Algorithm A test was done using ImageJ to see how the watershed algorithm would perform when applied to the same images used for testing the other methods. The results can be seen in Figures3.22 and 3.23.

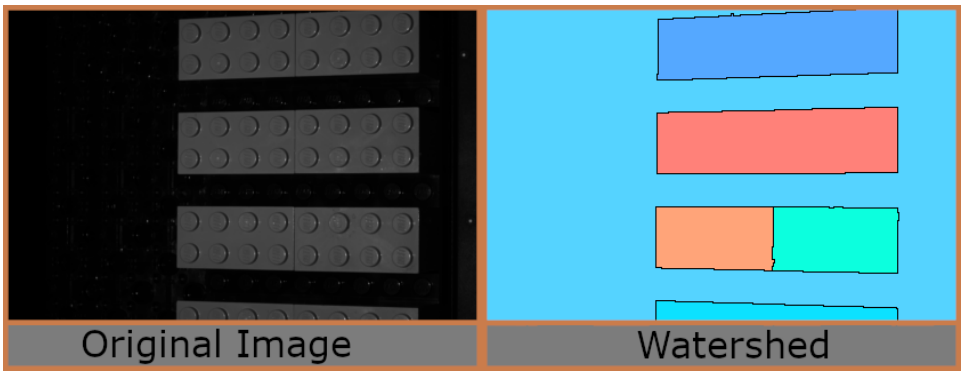


Figure 3.22: The left images show a low light image from the dataset, and the right is the same image with watershed applied using a plugin [68] in ImageJ[66].

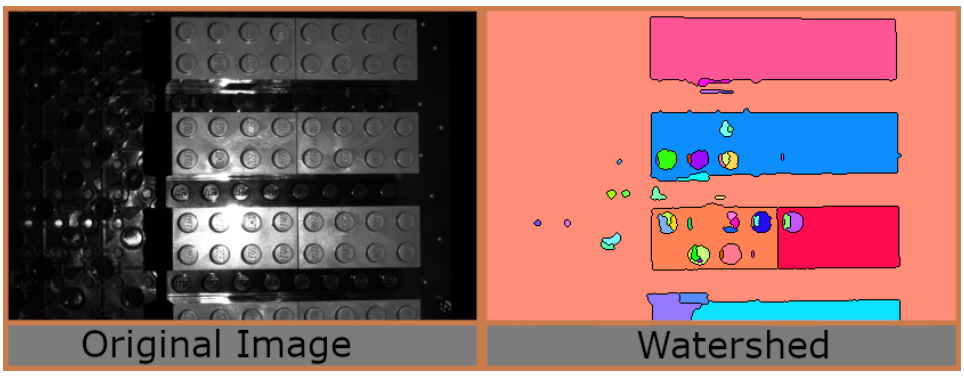


Figure 3.23: The left image shows an intense specular highlight image from the dataset, and to the right is the same image with watershed applied using a plugin [68] in ImageJ[66].

Although the watershed algorithm still has issues with getting a good segmentation of the image with specular highlights, but it is performing better than the previous methods that have been tested. Figures 3.22 and 3.23 show the results of the watershed algorithm used on the two images.

U-Net While traditional segmentation algorithms can give good segmentation on low-light images, they still not performing well for images with high specular highlights. Therefore, we have also chosen to explore the use of neural networks for segmentation. An test was done utilizing an implementation of U-Net[13] on the data given to us with masks made manually in GIMP[63]. The result of the test can be found in Figures 3.24 and 3.25.

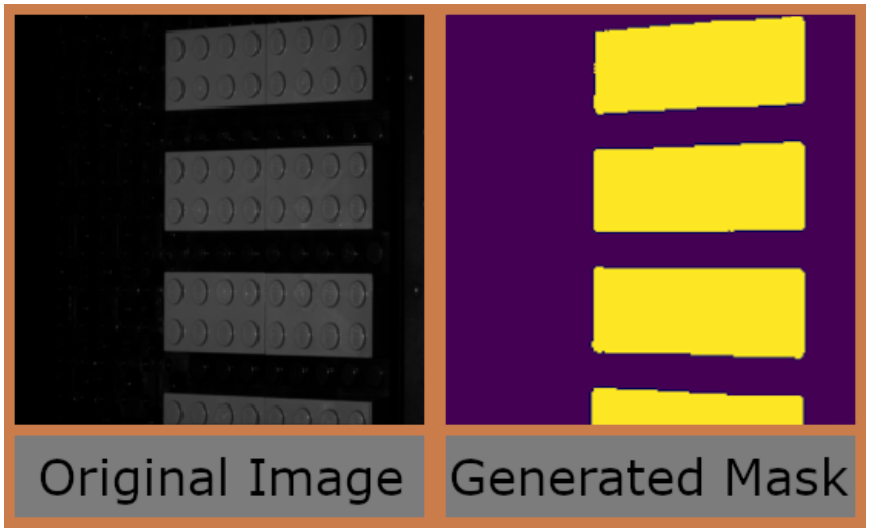


Figure 3.24: The mask generated by U-net along with the original low light image from the dataset. The optimizer used was Adam with the loss function Binary Cross-Entropy. Early stopping was used with a patience of 9 for a maximum of 100 epochs with a batch size of 16.

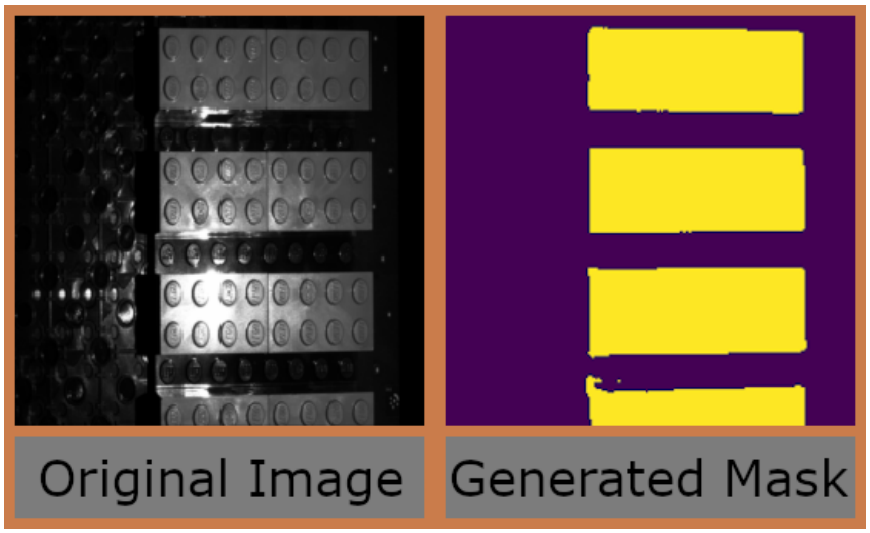


Figure 3.25: The masks generated by U-net along with the original image high specular highlight from the dataset. The optimizer used was Adam with the loss function Binary Cross-Entropy. Early stopping was used with a patience of 9 for a maximum of 100 epochs with a batch size of 16.

Looking at the generated masks in the tests conducted, U-net is the approach that gives the best resulting masks. Subsequently, we decided to go with U-net to segment the data for later use in anomaly detection.

3.6.2 Test Summary

When looking at the different results, U-Net is the one that gives the best mask, but it requires predefined masks to train the network for generating them. It is, therefore, our hope that we can use synthetic data to train a model to make a mask for the real data.

Even if the model trained on synthetic data performs poorly on real data, it might still be worth using a u-net model trained on real data only. The generated mask from the real data can still be accurate on the images with high specular highlights, granted that the future obtained data does not change significantly. When the model is trained, it will continue to generate a mask for future data collection efforts.

3.6.3 U-Net implementation

This section goes through the implementation of the U-Net. The images first get resized down to 512x512 so that U-Net can process them. The U-Net model uses ReLU and Sigmoid activation functions, Adam as Optimizer, and Binary Cross-Entropy as its loss function. Early stopping was used to stop the model from training when it did not improve its validation loss. The model was saved every time it improved, so if the training was stopped early for any reason, we would be able to continue training from that point.

Loading and Resizing Image Data

The code seen in Listing 3.4 was used to load in, resize and store all the images and masks in NumPy arrays. The code for resizing images consists of two loops. First is a double for-loop that goes through all the images and masks in the train folder and resizes them to fit the input for the model. If multiple masks exist for one image, they are combined into a single mask. The second also resizes the images but for the test images, where the are no masks. The images and masks are stored in X_train and Y_train for the train image and X_test for the test images.

```
1
2  #resize train images
3  print('resizing traning images and masks')
4  for n, id_ in tqdm(enumerate(train_ids), total=len(train_ids)):
5      path = TRAIN_PATH + id_
6      img = imread(path + '/images/' + id_ + '.png')[:,:,:
       IMG_CHANNELS]
7
8      img = cv2.resize(img, (IMG_HIGHT, IMG_WIDTH))
9      X_train[n] = img #fill empty X_train with values from img
10     mask = np.zeros((IMG_WIDTH, IMG_HIGHT, 1), dtype=np.bool)
11     for mask_file in next(os.walk(path + '/masks/'))[2]:
12         mask_ = imread(path + '/masks/' + mask_file)
13         mask_ = np.expand_dims(resize(mask_, (IMG_WIDTH, IMG_HIGHT
       ), mode='constant', preserve_range=True), axis = -1)
```

```
        mask = np.maximum(mask,mask_)
    Y_train[n] = mask

#resize test images
X_test = np.zeros((len(test_ids), IMG_WIDTH, IMG_HIGHT,
    IMG_CHANNELS), dtype=np.uint8)
sizes_test = []
print('resizing Test images and masks')
for n, id_ in tqdm(enumerate(test_ids), total=len(test_ids)):
    path = TEST_PATH + id_
    img = imread(path + '/images/' + id_ + '.png')[:,:,:
    IMG_CHANNELS]
    sizes_test.append([img.shape[0], img.shape[1]])
    img = cv2.resize(img, (IMG_HIGHT, IMG_WIDTH))
    X_test[n] = img

print('Done Resizing')
```

Listing 3.4: Code snippet used to resize images for U-Net

U-net

The code in Listing A.1 a lot of the inspiration for the code is from a GitHub repository by Bhattiprolu[69]. U-Net consists of two parts; the encoder (contracting path) and the decoder (expansive path). It uses different layers such as Conv2D, Dropout, Maxpooling, Conv2Dtranspose, And concatenate layers. In the Keras layer Conv2D, five inputs are given: the dimensionality space, kernel size, activation function, kernel initializer, and padding mode.

- **Dimensionality filters** are the number of kernels that the convolutional layer applies. For the first layer (c1) in U-net, this is 16. The number given here is equal to the number of feature maps we get for the layer[11].
- **Kernel size** is the size of the kernels which will be applied. In the first layer (c1), the height is 3, and the width is 3. These 3x3 matrices are the kernels that are used in the convolutional layer as the filters[11].
- **Activation function** is ReLU for the first layer (c1) and decides if the neuron should be active or not by calculating the weighted sum and adding the bias[11].
- **Kernel Initializer** decides the initial weight for the layer. For the first layer (c1), He_normal is used, which picks a random number based on the truncated normal distribution centered around zero and looking at the size of the previous layer. The weights are random, but the range of the numbers to pick from changes based on the size of the previous layer.[70, 11]
- **Padding** is used to keep the input image size by adding the number of pixels lost in all directions. Because of how convolutional operations work, a 3x3

kernel will remove a one-pixel wide border in all directions of the image when applied. The amount lost is more significant on larger kernel sizes. The 'same' parameter for padding means that we want to keep the size of the input image by adding zero-padding [11].

Concatenate Layer

Concatenate layer links two layers together. The first time we see the concatenate layer in U-Net is in the decoder, where it links the Conv2DTranspose layer (u6) with the Conv2D layer in the encoder (c4) corresponding to its size. It is done to get back information the might have been lost during convolutional operations.

Output Layer

The output layr is the last layer in a neural network and is used to return the final result. In the instance of U-Net, the network returns a mask. The last layer uses the sigmoid activation function because it works well with binary classification tasks, which for u-net is classifying between background and brick.

Training and saving model

ModelCheckpoint is used to save the model each time the model improves after an epoch. The callbacks included early-stopping with a patience of 20, meaning if the model did not improve after 20 epochs, it would stop training based on the validation loss. A log is also saved using TensorBoard, informing about the model's training.

The Model.fit() function is used to start the training. It takes multiple parameters, such as the number of epochs it should train for, batch size (the number of images in the network at one time), the validation split (which is 10% seen in line 11), and settings for callbacks. It can all be seen in Listing 3.5.

```
1
2 checkpointer = tf.keras.callbacks.ModelCheckpoint('
    model_for_TestDataModel.h5', verbose=1, save_best_only=True)
3
4
5 callbacks =[
6     tf.keras.callbacks.EarlyStopping(patience=50, monitor='
    val_loss'),
7     tf.keras.callbacks.TensorBoard(log_dir='logs'),
8     checkpointer
9     ]
10
11 results = model.fit(X_train,Y_train, validation_split=0.1,
    batch_size=16, epochs=400, callbacks=callbacks)
```

Listing 3.5: code for saving model and starting training

3.7 Using the U-Net model

The model was trained on images with the size 512x512. Using the model to predict the mask for a new image, we needed to resize them to be the same size. The code used for that purpose can be seen in Listing 3.6, where the original image is also saved in a variable that is used later with the mask to subtract the background.

```
1
2       image = image.numpy()
3       #SAVE AN COPY OF IMAGE
4       OriginalImage = image
5
6       #RESIZE
7       image = cv2.resize(image, (IMG_HIGHT, IMG_WIDTH))
8
9       print("image shape: ", image.shape)
```

Listing 3.6: Resizing the images

The morphology operation 'Closing' was used to remove holes in the generated mask after the mask was predicted, seen in Listing 3.7. Afterwards, it was resized back to the original image size.

```
1       # PREDICT MASK BASED ON U-NET MODEL
2       preds_mask = model.predict(image, verbose=1)
3       preds_mask_t = (preds_mask > 0.5).astype(np.uint8)
4
5
6       # RUN MORPH (CLOSING) TO REMOVE GAPS IF ANY
7       Mask = np.squeeze(preds_mask_t*255)
8       Mask = cv2.morphologyEx(Mask, cv2.MORPH_CLOSE, cv2.
    getStructuringElement(cv2.MORPH_CROSS,(5,5)), iterations= 3)
9       Mask = cv2.resize(Mask, (OriginalImage.shape[1],
    OriginalImage.shape[0]))
10
11      ret,Mask = cv2.threshold(Mask,220,255,cv2.THRESH_BINARY)
12      Mask = Mask.astype(np.uint8)
```

Listing 3.7: Closing code used to fill in gaps

In Listing 3.8, the code for segmenting individual bricks by cutting masks into separate different masks can be seen, with their resulting masks being shown in Figure 3.26. We ended up not using the functionality, as it changed the dataset too much and was therefore not representative of the data collected.

```
1
2 ##----------- For Cutting in slice -----------##
3 #Cropout function
4 def get_segment_crop(img,tol=0, mask=None):
5     if mask is None:
6         mask = img > tol
7     return img[np.ix_(mask.any(1), mask.any(0))]
8
9 # SEPARATE MASK
```

```
10 def create_separate_mask(mask):
11     #get the masks
12     mask = mask
13     #List Containing the final masks
14     maskList = []
15     label_im, nb_labels = ndimage.label(mask)
16 
17     for i in range(nb_labels):
18 
19       # create an array which size is same as the mask but filled
       with
20           # values that we get from the label_im.
21           # If there are three masks, then the pixels are labeled
22           # as 1, 2 and 3.
23       mask_compare = np.full(np.shape(label_im), i+1)
24 
25           # check equality test and have the value 1 on the location
        of each mask
26       separate_mask = np.equal(label_im, mask_compare).astype(int)
27 
28           # replace 1 with 255 for visualization as rgb image
29       separate_mask[separate_mask == 1] = 255
30 
31           # Append mask to list
32       maskList.append(separate_mask)
33     return maskList
34 
35 ##----------- For Cutting in slice -----------##
```

Listing 3.8: Code used to separated bricks based on mask

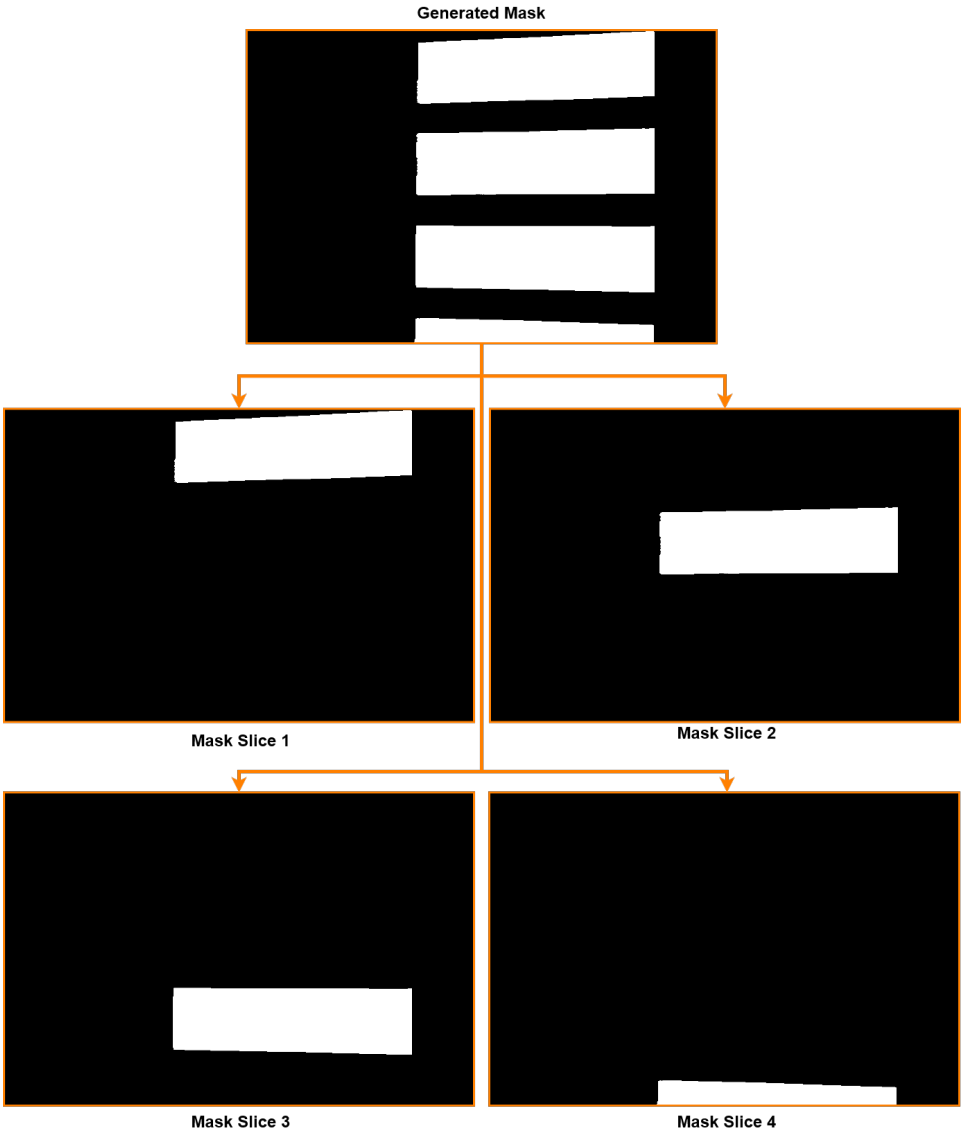


Figure 3.26: Sliced mask

3.8 Anomaly Detection

To determined the best possible autoencoder for our dataset, different smaller experiments were conducted. The input was resized from 4248x2832 to 1062x708 and then normalized to the 0 and 1 range.

Early stopping was set to a patience of 10, so if the model does not improve in a consistent 10 epochs, it will stop the training.

3.8.1 Data Preprocessing

Due to the original images having a large resolution size (4248x2832), the images were resized to 1536x1024 before slicing them into smaller patches. The resizing resolution was found by checking all available resolutions under 4248x2832, where 256x256 would fit each dimension, avoiding any leftover pixels. The patches were determined to be 256x256, slightly overlapping so they would be easier to stitch together after. To avoid any border effects on the patches, the patches were increased to 320x320. The slicing windows can be seen in Figure 3.27.

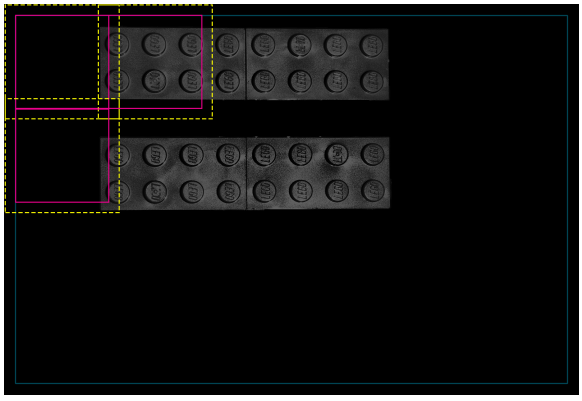


Figure 3.27: Slicing of the images. The yellow dashed squares show the 320x320 patches that are used as the input for the model, and the pink squares show the 256x256 resulting patches. These patches are repeated over the whole image.

The reconstructed patches of 320x320 would be cropped back to 256x256 and stitched together to a 1536x1024 image. The full slicing and stitching process can be seen in Figure 3.28.

The amount of slices would then end up being 24 images per image, which in total was 576 images per batch.

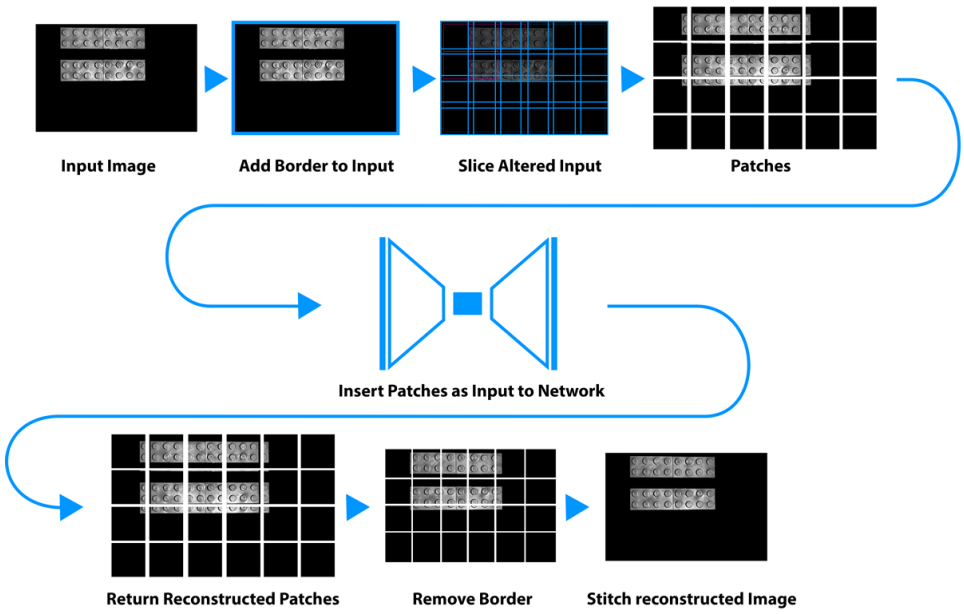


Figure 3.28: Input being processed for anomaly detection

3.8.2 Testing Architecture Parameters

In this section we will test different architecture parameters, such at architecture size, dropout, and latent space.

Architecture Size

A small experiment was made to test the effect of the amount of layers in a convolutional autoencoder.

A hidden layer block consisted of either a compressing or expanding layer followed by two convolutional layers.

The encoders hidden layer block consisted of a Max Pooling layer with 2x2 kernel and stride of 2, followed by two Convolutional layers. For the decoder, the hidden layer block consisted of a Transposed Convolutional layer, and two convolutional layers. All layers were using 3x3 kernels.

The results show that the more layers we added, the more blurred and generalized the surface of the brick got. This made any visible defect 'disappear' in its reconstruction, which consequently increased the defect on the difference map. The knobs on the brick became sharper and clearer with the extra layers.

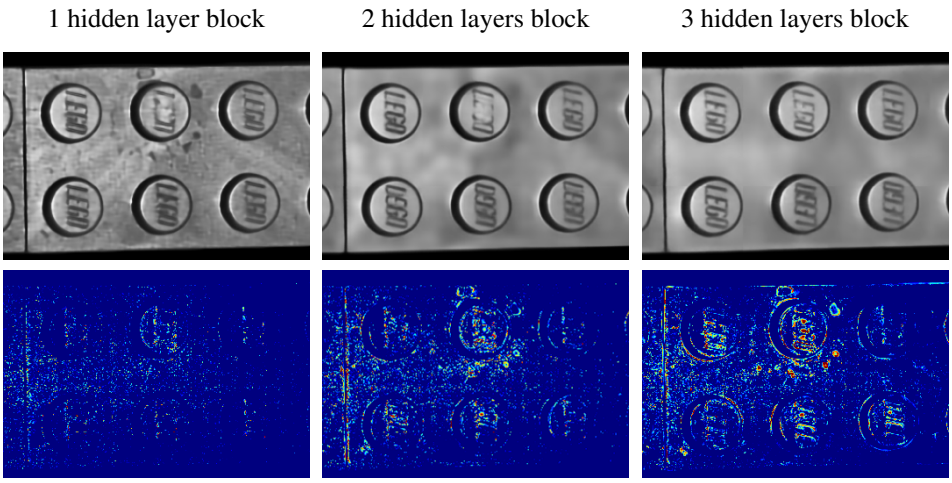


Figure 3.29: Comparison of layer sizes in a CAE.

Dropout

As we are interested in having a more generalized model, to avoid reconstruction of the defects, a dropout was experimented with, to add regularization to the network. A dropout layer was added in the encoder part of the network, and the addition shows increased anomaly scores for the defect images, which can be seen in Figure 3.30.

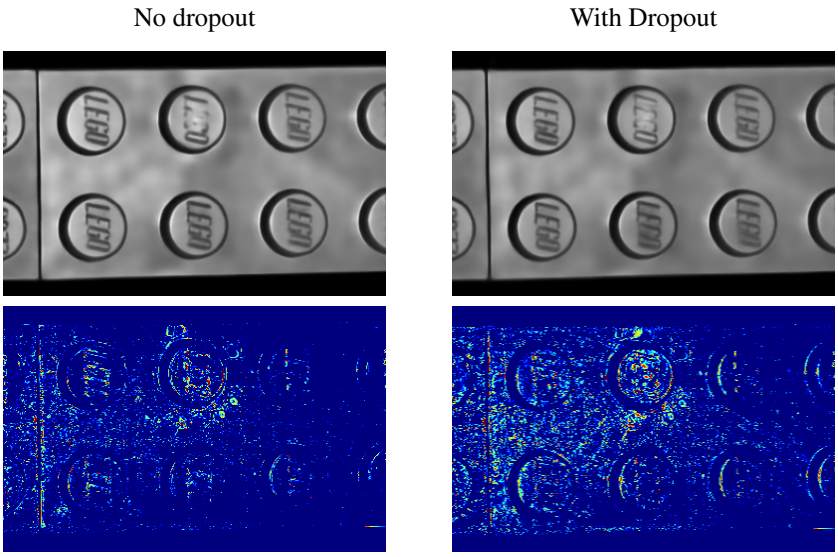


Figure 3.30: Comparison of the addition of dropout in the model.

Latent Space

As the latent space is the bottleneck layer of the network, which consists of the information the decoder will reconstruct the image from. Having a small latent space forces the network to extract the most meaningful features for the reconstruction. Two latent spaces were compared, showing that when the latent space is decreased from 50 to 25 in a dense layer, the defect stands out slightly more, and edges are more likely to be reconstructed right. The difference can be seen in Figure 3.31.

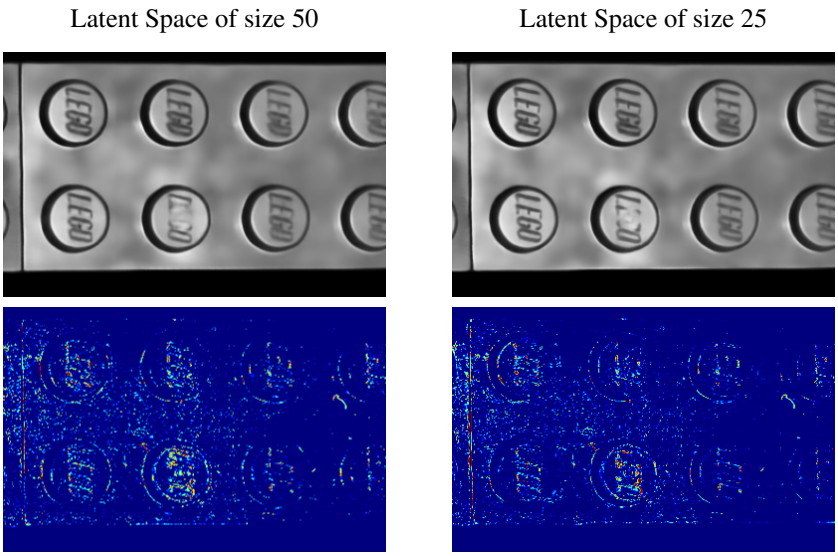


Figure 3.31: Comparison of the addition of geometric transformations during training.

3.8.3 Chosen Architecture

The chosen architecture for anomaly detection was inspired by the work by Chow et al. [12].

After the smaller experiments, the final CAE was a 31-layer convolutional autoencoder, consisting over Conv2D-layer with ReLU activation, dropout-layer and max-pooling layers in the encoder, while the decoder consisted of Transposed Conv to increase the dimensions back up, Conv2D layers with ReLU, except for the last layers with sigmoid activation, as we are interested in greyscale values between 0 and 1. The latent space consisted of 3 dense-layer to compress the data, with the smaller dense layer being 25 in size. The architecture can be seen in Figure 3.32, with a larger size of the overview found in Appendix A and the models code can be seen in Appendix A.

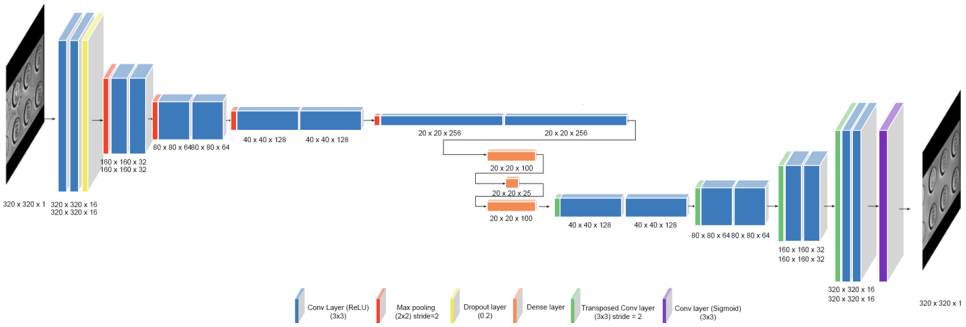


Figure 3.32: Final Convolutional Autoencoder

Chapter 4

Evaluation

This chapter will describe the experiments done in this project to evaluate the proposed system. The goal for this chapter is to answer the research question *"How can a robust defect detection system be made for visual quality control, which utilizes deep learning and supports small and varying datasets as input?"*. As this chapter will describe multiple experiments, each experiment will be presented with a procedure and results of the experiments.

4.1 Experimental setup

The setup for the experiments in this project is done using an experimental setup, where one process is tested at a time. The experiments were conducted in Google Colab with GPU hardware acceleration. The experiments will involve testing the Automatic segmentation and Anomaly Detection. The final experiment will be about how well the system detects defects in polymer surfaces.

4.1.1 Data

For testing the impact of synthetic data on automatic segmentation and on the anomaly detection network, three datasets were made; Real data, Synthetic data, and a mix of synthetic data and real data.

Real Data

The real data is the original light grey brick dataset, consisting of 10 batches with 24 images from different viewpoints, where one of the batches contains defects. Two of the batches were reserved as test data; one with defects and one without defects. In total, 240 images, 168 for training and 48 for testing.

Synthetic Data

The synthetic dataset consisted of 640 images generated in Blender. This dataset was used for training U-Net for segmentation and the autoencoder for anomaly detection. The test data was the same as the real data, with 48 real images.

Mixed Data

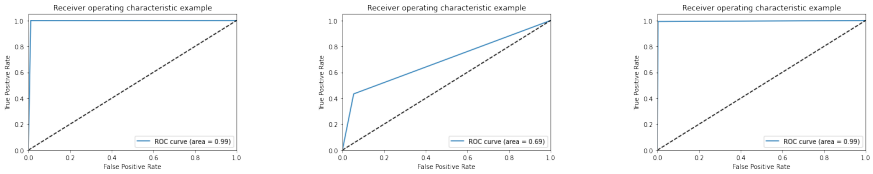
For training, the 168 train images from the real data were combined with the 640 synthetically generated images, in total 808 images. The test data was still the 48 real images as with the other datasets.

4.2 Automatic segmentation

For the automatic segmentation, the U-net was tested by comparing a U-net trained on synthetic data, real data, and mixed data. The purpose of this test was to investigate if the U-net could be trained on synthetic data only. Real images are used to test the model and its performance. The results can be seen in Table 4.1 and the Receiver operating characteristic(ROC) can be seen in Figure 4.1.

Training Data	**Test Data**	**Precision**	**Recall**	**F1**	**AUC**
All Synthetic Data	Real(Batch 1-2)	0.734	0.433	0.55	0.69
Real Data (Batch 3-10)	Real(Batch 1-2)	0.968	0.999	0.98	0.97
Mixed Data	Real(Batch 1-2)	0.991	0.991	0.99	0.99

Table 4.1: Results of U-Net



Real Synthetic Mixed

Figure 4.1: ROC curve for Real, Synthetic, and Mixed

4.2.1 Mask Results

In Figure 4.2 the resulting masks are generated using the models trained on the Real, Synthetic, and Mixed data set can be seen. It is easy to see that a model trained only on the synthetic data set does not perform well enough to segment the real data. The model trained on the mixed data set is the one performing the best out of the three.

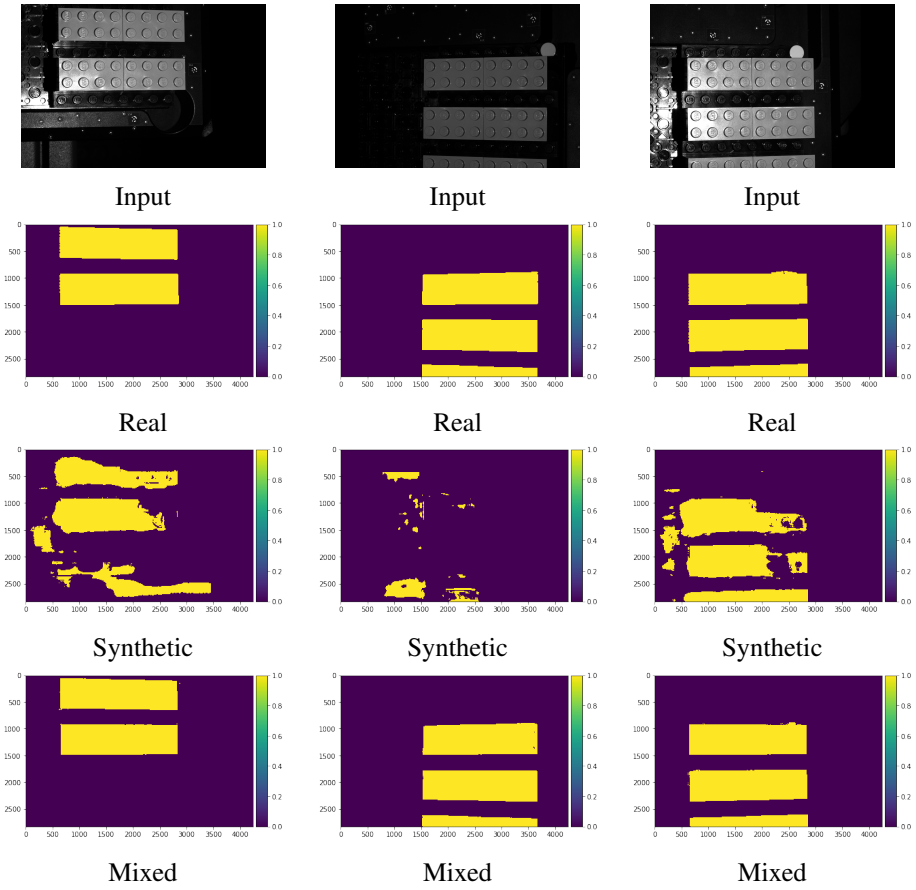


Figure 4.2: Random selected mask generated by models trained on Real, Synthetic and mixed data

4.3 Anomaly Detection

Three experiments were conducted with the three different train datasets. All experiments used the same test and validation dataset. The network used patience of 10, MSE-loss function, and Adam optimizer. The experiments were run in Google Colab.

The anomaly score was calculated by taking the squared pixel difference:

$$AnomalyScore = \sqrt{(x - \hat{x})^2}, \tag{4.1}$$

where x is the original image and $\hat{x}$ is the predicted image.

The anomaly threshold was found by taking the highest anomaly score from the training data per experiment. The anomaly score range greatly varied between each model and a static threshold.

4.3.1 Results

This section will cover the result of anomaly detection. For each dataset, an example of a non-defect image and defect image will be shown, together with the reconstructed image from the Autoencoder, a difference map of the input image and the reconstructed image. An anomaly map is placed on the input image to show where the detected defects on the polymer surface are located. The defect threshold for the anomaly maps are based on the median maximum anomaly scores from the training data, so the anomalies need to be above the training images maximum anomaly score.

Real Data

For the autoencoder train on real data, it can be seen that the reconstruction reminds a lot about the input image. It does however have a smoother surface, without the structure of the brick and without any defects, making the difference map and anomaly map highligt the defects.

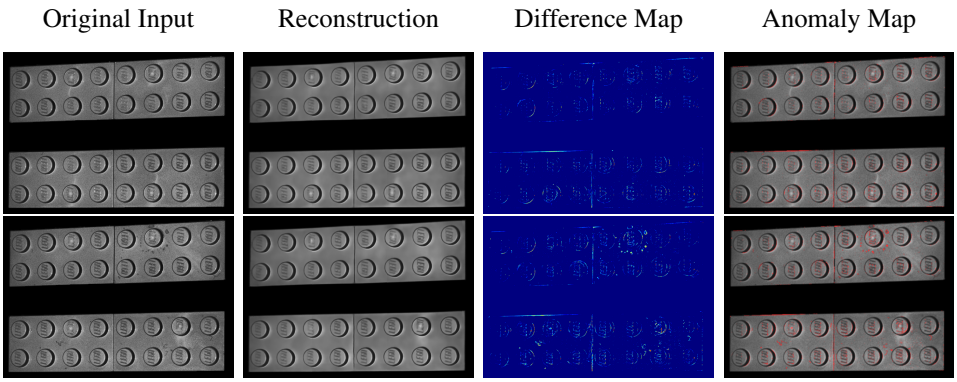


Figure 4.3: Defect and non-defect predictions and anomaly maps for the network trained on real data.

Synthetic Data

For the autoencoder trained on only synthetic data, it can be seen that the reconstruction looks similar in shape of the brick, there is a correct amount of knobs, and they are placed correctly. However, the reconstruction seems blurry and almost low poly. This makes the difference map light up, especially around the knobs of the brick. The anomaly map shows that the defects are located around the knobs, but the ”real” defect is not highlighted at all.

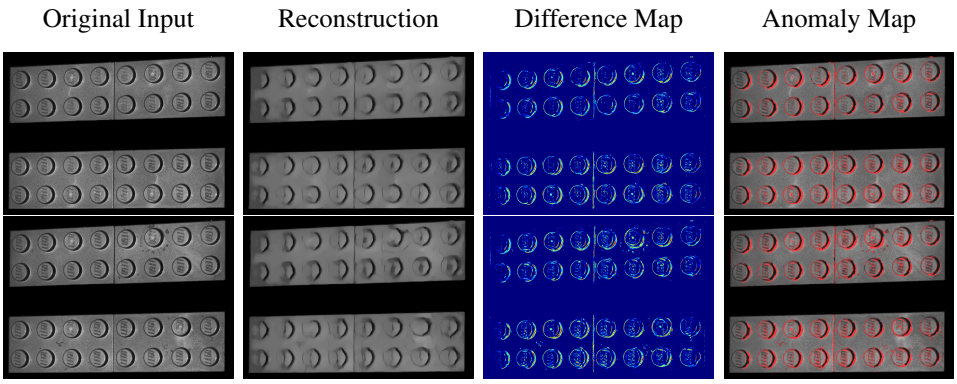


Figure 4.4: Defect and non-defect predictions and anomaly maps for the network trained on synthetic data.

Mixed Data

For the autoencoder trained on the mixed dataset, the reconstructions look a lot like the input image, with a lot of detail and a smooth surface. However, it also has a small hint of the defect in the reconstructed image. The difference maps show a lot of small differences, and when we look at the anomaly map, we can see that due to the low threshold, all the structure from the input image is now classified as a defect, making it hard to see the ”real” defect.

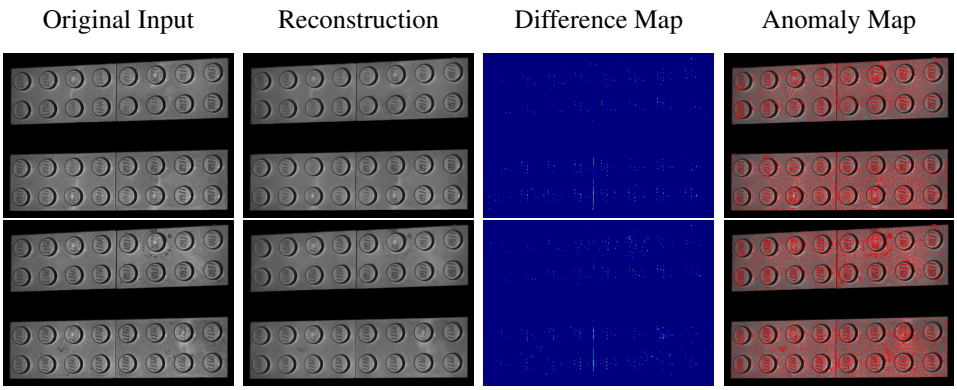


Figure 4.5: Defect and non-defect predictions and anomaly maps for the network trained on mixed data.

4.3.2 ROC Curve, F1

Receiver operating characteristic (ROC) curves are made for the three different datasets, showing that generally, for all three models that a simple straight threshold can not be determined for the data. The AUROC score for the three dataset are as follows: Real data = 0.53, Synthetic data = 0.54 and Mixed data = 0.51 as seen in Figure 4.6. These scores tell us that with the thresholds that we made, the network can not predict whether the images are a defect or non-defect. They are all equivalent to make a 50/50 guessing if the image is a defect or not.

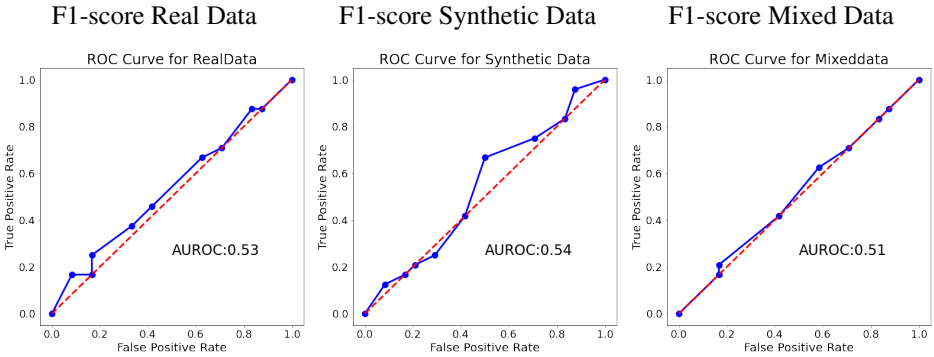


Figure 4.6: ROC curves

F1 score was then plotted with the precision and recall values for all the different thresholds and can be seen in figure4.7. F1-score, which is the weighted average of the precision and recall curve, tells us which of the selected threshold would give us the best result if used on the data.

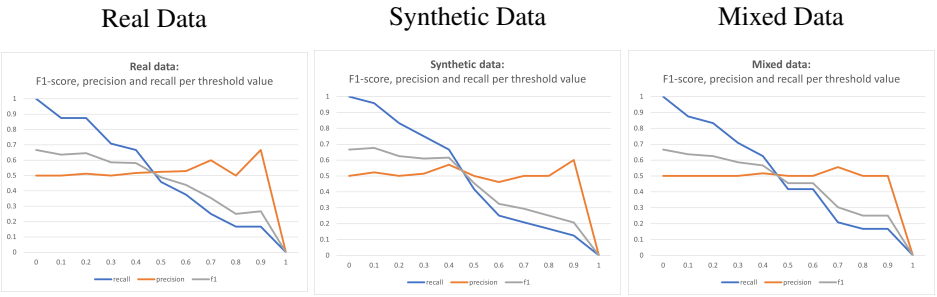


Figure 4.7: F1 score plotted with Precision and Recall

From the ROC curve and f1 score, we can tell that a simple threshold for the data is not possible to make. The anomaly score for each image is to cluster together per angle, so no matter the threshold value, the network can do nothing but guess which anomaly score is a defect and which one is a non-defect. This is the

case for all examples of real, synthetic, or mixed data. An example can be seen in figure 4.8, where the anomaly scores for defect and non-defect are almost on top of each other.

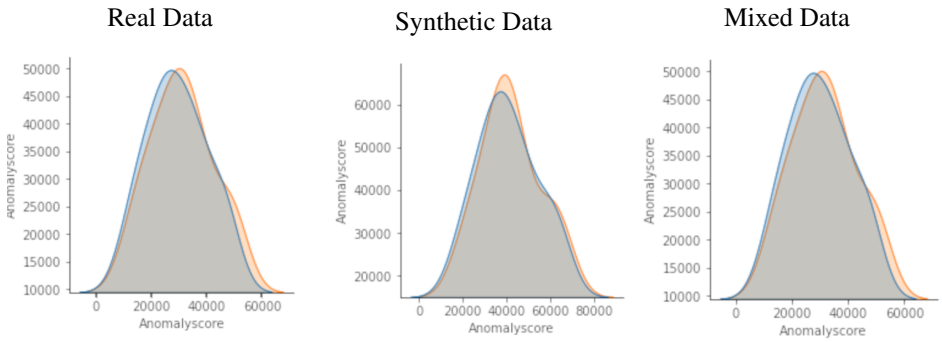


Figure 4.8: The anomaly scores

Chapter 5

Discussion

This chapter discusses different aspects of the project's process and will discuss possible solutions for future work.

5.1 U-Net

If the data was acquired in another way, U-Net could be redundant. Our reason for using U-net is that the data is "bad" for traditional segmentation methods such as thresholding or watershed due to highlights in the images and gray-scale color space instead of RGB or HSI. The data was collected in a controlled environment, which means additional steps can be taken, making segmenting the data more straightforward. One solution could be having a uniform background, so pixel values between target and background have no similarities.

5.1.1 Mean mask in U-Net

There were ten images taken from the same viewpoint in the dataset. Therefore, a mask generated from one image could be used for a corresponding image from the same view. Creating a mean mask that could be applied to all images from that viewpoint would be worth investigating further, as it might work better than a mask generated from a single image. Having a mean mask could have given a better result, but we ran into out-of-memory errors when attempted. To avoid this, other ways to generate mean masks or more computational power would be necessary.

5.2 Quality and Realism of Synthetic Data

The synthetic dataset was generated in Blender with a provided CAD model from our collaboration partner. A problem that occurred was that the Lego logos on the knobs were not very visible in the render compared to the real images, and the texture in the real images is gone in the synthetic data. This difference made it

hard for the network to learn these features. Additional adjustments with lighting and render settings are needed to get the renderings closer to the real images.

As seen in Figure 5.1 the synthetic data is too ”*clean*”. It does not have the texture that the real data has.

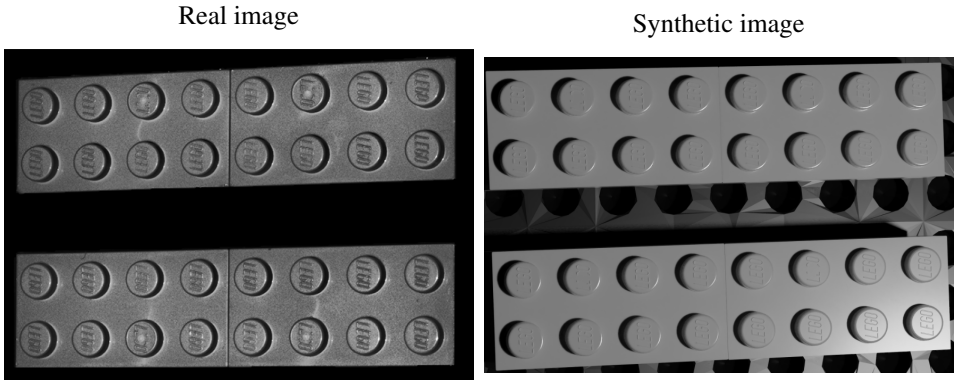


Figure 5.1: The difference in texture between real and synthetic data

The texture could have been added on a bump map or possibly with random noise on the UV map but will need further testing to see how it affects the network.

For the realism of the synthetic data, a qualitative test could have been conducted where participants could have looked at synthetic data examples with different rendering settings and compared them to real data to see which synthetic images represent the real data the most. Parameters such as samples per pixel, materials, and light options can be changed to make synthetic data look even more realistic.

5.3 The effect of CLAHE

Contrast limited histogram equalization was used to increase the contrast in the image. The change in contrast will visually increase the difference between what is a defect and what is not. By changing the contrast, the hope was that the network would have an easier time determining what is and is not a defect. Furthermore, when generating a reconstruction of a brick to subtract from the original image, the places where a defect occurs will give a higher anomaly score.

5.4 The effect of perspective correction

When applying Perspective Correction to our dataset, as seen in Figure 5.2, Perspective Correction appears to only have corrected some bricks. The upper half of the image shows how projection correction should work by making the objects appear as if the image was taken from a front parallel view. Due to this unwanted effect, we have decided not to test on projection corrected images as a new method

for projection correction would have to be found, which does not distort some of the images the same way.

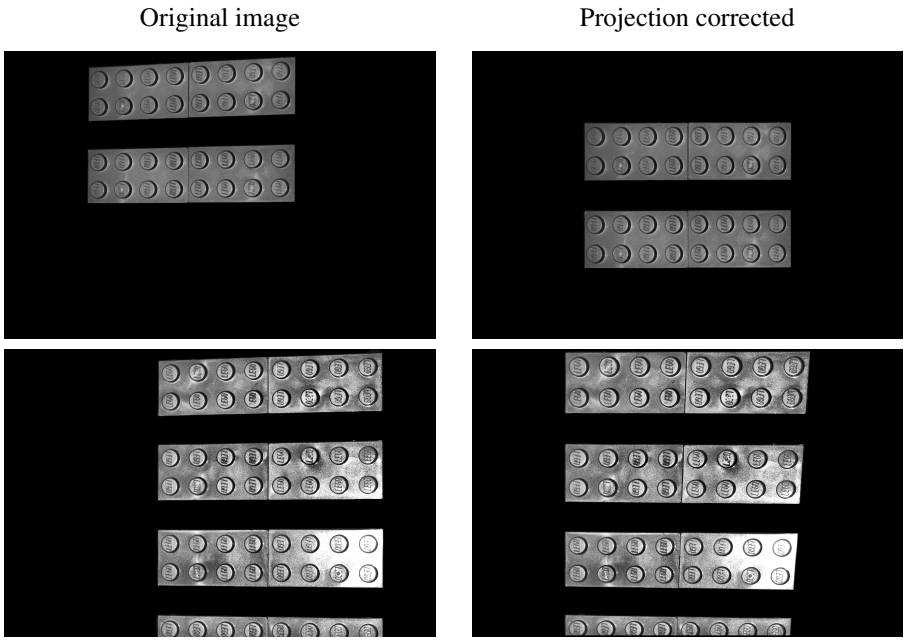


Figure 5.2: Example of good projection correction(top) and bad projection correction(bottom)

5.5 Brightness as Data Augmentation

When testing out U-Net with real, synthetic, and mixed data, there is a difference in how well they perform. The network using real data performs better than the one on synthetic data, and the mixed performs a little better than real data. As seen in Figure 4.2, the real data has quite a lot of difference in brightness compared to each other. It can also be seen that training on synthetic data performs better on brighter data, indicating that the synthetic data might be a lot brighter than the real data. One way to maybe get around this would be to test the effect of random brightness as a data augmentation when training on the synthetic data. However, this will need to be tested at a future point.

5.6 Experiments

Throughout this project, multiple experiments have been made. This section will discuss some of the choices made in these experiments.

5.6.1 Selecting of test and train batches

The experiments performed on U-Net and the anomaly detection both require data for training and testing. Batch one is the batch with defects and is put in the testing. However, we also need to ensure that the data can perform well on non-defect data. Therefore a batch of good samples (batch two) was added to the testing dataset. The rest of the batches (batch three to ten) would then be the training data.

However, there could be a potential bias because batch two is always in the testing set. Even though all batches, except batch one, are evaluated by our collaborator as good batches, there could be differences in texture and placement of the bricks, making batch two potentially better or worse than the other batches to test on.

5.6.2 Unequal amount of data (more synthetic data than original)

The amount of synthetic data generated was 664 images. These images were all generated looking from the same angels. The cameras always went from left to right or from the bottom to the top of the synthetic bricks. Since the data was not generating the same amount of images from both angles, there were more images from one angel than the other, meaning that the synthetic data set could be biased to one view.

Ways to improve the synthetic data could be to generate data from new angles. Generating new angles can be done in Blender by making the Bezier curves cover a wider range of angles. Having more angles could also help make the network more robust for new angles in the future.

5.6.3 Results

This section will discuss some of the results from the experiments. This will include results from the U-Net and results from the anomaly detection.

U-Net

The results of U-Net show that the mixed data performs the best. However, given that the mixed data have roughly 3.8 times the amount of data compared to the result of the real data and the time used to set up the synthetic data. The real data might be the better option for automatic segmentation.

Threshold of Defects

When only looking at the anomaly scores, both the defect and non-defect images are distributed across the whole anomaly score range, regardless of defects being present or not. This is due to the visually different images, where some images have four bricks, and others have six visible. This difference makes the image with

more bricks have a higher anomaly score, simply because there is more to calculate the difference from.

The anomaly score would be more accurate when calculating the anomaly threshold based only on the corresponding image angle in the training data. It would no longer be a global threshold being calculated across all image angles. Examples of anomaly thresholds being calculated per view can be seen in Figure 5.3.

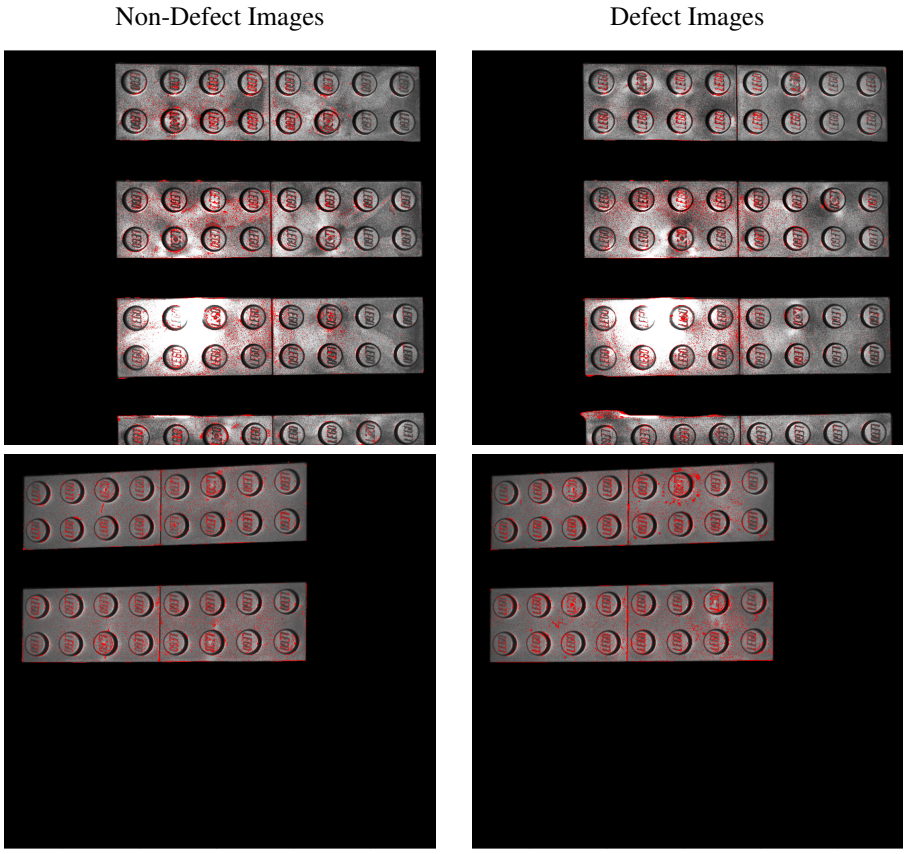


Figure 5.3: Image from two different angles with their appropriate thresholds. The thresholds are based on their train image angle counterparts, where the median was calculated based on the train images anomaly scores.

In Figure 5.4, the images are displayed as data points when the anomaly scores are compared against the number of pixels above zero. The difference between left and right camera angles is displayed as triangle and circle points. The anomaly scores are higher for the defect images for each corresponding view angle.

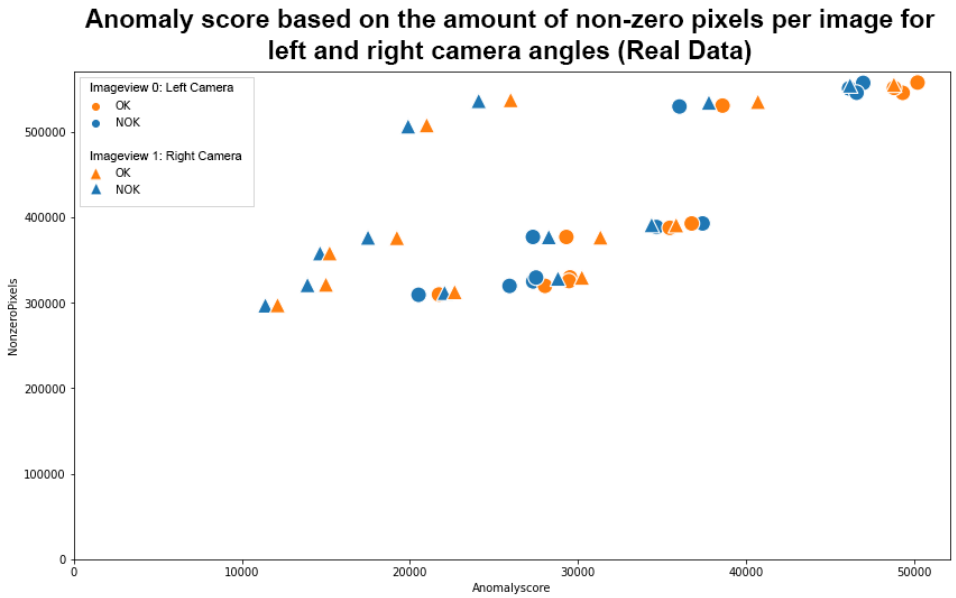


Figure 5.4: Distribution of the defect and non-defect batches based on anomaly score and amount of non-zero pixels in the images. Triangle points are images taking from the right camera, where circles are taken from the left.

A solution to the anomaly scores being influences by their view angles could be having an adaptive threshold created based on the view angle. Another solution to be explored is using a traditional classifier, such as SVM, to create the threshold boundary of the view angles, and clustering with a KNN could make the model more dynamic. Both SVM and clustering would need relevant features to be utilized.

The simpler solution could also be to eliminate some angles as not all angles showed enough distinction between the defect and non-defect images.

5.7 Future work

In this project, many different experiments have been done together with much internal testing. Based on these, some possible solutions have been found to make this process of defect detection smoother and how the process could be improved or needs further testing to determine how well the process works.

5.7.1 Future Unknown Angles

As U-Net and the CAE have been training on batches with all angles present, it is unclear whether the networks would perform the same way with future unknown angles. The use of synthetic data could make the network robust towards this, but future testing is required.

5.7.2 Data Collection

When new data is collected, we suggest that the element is placed on a uniform background and the images are in color. A uniform background would help make sure that the network does not learn from the background. Furthermore, a color image would give more data to train on because the image would have three channels. Some color mistakes would not show up as much on a grey-scale image and might be easier to catch if the images were in color. It is also essential to think about how the light hits the element since the surface can reflect the light in the camera, making an area of intense light. An area of intense light on the element would be impossible to get any data from because there will be no change in pixel intensity, making it hard for a network to learn anything. The placement of the element in the image is also something to consider when collecting data.

What to be aware of when collecting data:

- Background
- Light
- Color Space
- Placement

Masks

If the data were collected with the purpose of segmentation, watershed would be a reasonable choice, as seen in Figure 3.22. When creating a mask for the images with no intense light, the watershed algorithm visually gives a good mask. If the images were in color, other methods such as color thresholding would also be something that might give a good mask, but this would need to be tested on a dataset with colors.

Brick separation

By cutting the data up and into separate images, we might get better results in anomaly detection. Together with perspective correction, it could improve even further. It would remove some of the differences the images have from being taken from different viewpoints. Implementation of brick segmentation was made, and a dataset was generated, but it was never thoroughly tested but is something to consider in the future.

Chapter 6

Conclusion

In this project, a process for automatic segmentation and anomaly detection has been set up. The process was designed around the problem statement:

"How can a robust defect detection system be made for visual quality control, which utilizes deep learning and supports small and varying datasets as input?"

A system consisting of U-net and a Convolutional Autoencoder was created, which utilized data augmentation and synthetic data to increase the dataset.

6.1 Synthetic data

The synthetic data did not work well enough for segmentation and defect detection. There is still a need to improve lighting, surface detail, and random background in the synthetic data to be more helpful.

6.2 Automatic segmentation

U-Net was chosen for the automatic segmentation of the datasets. When U-Net is trained on real or mixed data, it can segment the background out. If a different method was used to capture the data, a segmentation method such as watershed could be used, removing the need to train a U-Net for segmentation.

6.3 Anomaly detection

The selected network could support a small dataset as it could predict images reasonably close to non-defect data. While the condition with real and mixed data showed the best looking reconstructions, the synthetic data performed the best with a global threshold. Subsets and adaptive thresholds would be needed to determine the best viewpoints and best thresholds in the future. Further work is needed in more fitting synthetic data, which is closer to the original real images.

Bibliography

[1] J. Liu, J. Guo, P. Orlik, M. Shibata, D. Nakahara, S. Mii, and M. Takáč. Anomaly detection in manufacturing systems using structured neural networks. In *2018 13th World Congress on Intelligent Control and Automation (WCICA)*, pages 175–180, 2018.

[2] GOM Gmbh. Atos scanbox series 4, 2021-04-21. https://www.zebicon.com/maaleudstyr/automatiseret-3d-scanning/atos-scanbox-serie-4/.

[3] GOM Gmbh. Precise industrial 3d metrology, atos capsule, 2021-03-19. https://www.gom.com/en/products/high-precision-3d-metrology/atos-capsule.

[4] Anne Juhler Hansen, Hendrik Knoche, and Thomas B. Moeslund. Fantastic plastic? an image-based test method to detect aesthetic defects in batches based on reference samples. *Polymer Testing*, 89:106585, 2020.

[5] Shachi Shah. Do we really need gpu for deep learning? - cpu vs gpu, 2018.

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

[7] Ismoilov Nusrat and Sung-Bong Jang. A comparison of regularization techniques in deep neural networks. *Symmetry*, 10(11):648, Nov 2018.

[8] Rob Fergus. Computer vision - csci-ga.2271-001.

[9] Chigozie Nwankpa, W. Ijomah, A. Gachagan, and S. Marshall. Activation functions: Comparison of trends in practice and research for deep learning. *ArXiv*, abs/1811.03378, 2018.

[10] Janusz Kolbusz, Pawel Rozycki, and Bogdan M Wilamowski. The study of architecture mlp with linear neurons in order to eliminate the "vanishing gradient" problem. In *Artificial Intelligence and Soft Computing*, Lecture Notes in Computer Science, pages 97–106, Cham, 2017. Springer International Publishing.

[11] Keras Team. Keras documentation: Conv2d layer.

[12] J.K. Chow, Z. Su, J. Wu, P.S. Tan, X. Mao, and Y.H. Wang. Anomaly detection of defects on concrete structures with the convolutional autoencoder. *Advanced Engineering Informatics*, 45:101105, 2020.

[13] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In Nassir Navab, Joachim Hornegger, William M. Wells, and Alejandro F. Frangi, editors, *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*, pages 234–241, Cham, 2015. Springer International Publishing.

[14] NVS Yashwanth. Understanding gradient descent, Sep 2020.

[15] Sebastian Ruder. An overview of gradient descent optimization algorithms, Jan 2016.

[16] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *International Conference on Learning Representations*, 12 2014.

[17] Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, Günter Klambauer, and Sepp Hochreiter. Gans trained by a two time-scale update rule converge to a nash equilibrium. *CoRR*, abs/1706.08500, 2017.

[18] Du-Ming Tsai and Po-Hao Jen. Autoencoder-based anomaly detection for surface defect inspection. *Advanced Engineering Informatics*, 48:101272, 2021.

[19] Raghavendra Chalapathy, Aditya Krishna Menon, and Sanjay Chawla. Robust, deep and inductive anomaly detection, 2017.

[20] Toufique Ahmed Soomro, Ahmed J. Afifi, Ahmed Ali Shah, Shafiullah Soomro, Gulsher Ali Baloch, Lihong Zheng, Ming Yin, and Junbin Gao. Impact of image enhancement technique on cnn model for retinal blood vessels segmentation. *IEEE Access*, 7:158183–158197, 2019.

[21] Stephen M. Pizer, E. Philip Amburn, John D. Austin, Robert Cromartie, Ari Geselowitz, Trey Greer, Bart ter Haar Romeny, John B. Zimmerman, and Karel Zuiderveld. Adaptive histogram equalization and its variations. *Computer Vision, Graphics, and Image Processing*, 39(3):355–368, 1987.

[22] Ekin D Cubuk, Barret Zoph, Dandelion Mane, Vijay Vasudevan, and Quoc V Le. Autoaugment: Learning augmentation strategies from data. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 113–123, 2019.

[23] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, May 2017.

[24] Giuseppe Russo. Data augmentation with automatic label preserving transformation. Master's thesis, ETH Zurich, 2020.

[25] L Jagannathan and CV Jawahar. Perspective correction methods for camera based document analysis. In *Proc. First Int. Workshop on Camera-based Document Analysis and Recognition*, pages 148–154, 2005.

[26] Li Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[27] Connor Shorten and Taghi M Khoshgoftaar. A survey on image data augmentation for deep learning. *Journal of Big Data*, 6(1):1–48, 2019.

[28] Arun Gandhi. Data augmentation — how to use deep learning when you have limited data - part 2.

[29] Tensorflow. Tensorflow data augmentation.

[30] Ren Wu, Shengen Yan, Yi Shan, Qingqing Dang, and Gang Sun. Deep image: Scaling up image recognition, 2015.

[31] Ken Chatfield, Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Return of the devil in the details: Delving deep into convolutional nets, 2014.

[32] Swarnendu Ghosh, N. Das, I. Das, and U. Maulik. Understanding deep learning techniques for image segmentation. *ACM Computing Surveys (CSUR)*, 52:1 – 35, 2019.

[33] Alberto Garcia-Garcia, Sergio Orts-Escolano, Sergiu Oprea, Victor Villena-Martinez, and José García Rodríguez. A review on deep learning techniques applied to semantic segmentation. *CoRR*, abs/1704.06857, 2017.

[34] Unsplash. Beautiful free images pictures.

[35] M. Siam, S. Elkerdawy, M. Jagersand, and S. Yogamani. Deep semantic segmentation for automated driving: Taxonomy, roadmap and challenges. In *2017 IEEE 20th International Conference on Intelligent Transportation Systems (ITSC)*, pages 1–8, 2017.

[36] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In Nassir Navab, Joachim Hornegger, William M. Wells, and Alejandro F. Frangi, editors, *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*, pages 234–241, Cham, 2015. Springer International Publishing.

[37] Scikit-Image Development Team. Thresholding.

[38] Nobuyuki Otsu. A threshold selection method from gray-level histograms. *IEEE Transactions on Systems, Man, and Cybernetics*, 9(1):62–66, 1979.

[39] Song Yuheng and Yan Hao. Image segmentation algorithms overview. *CoRR*, abs/1707.02051, 2017.

[40] Programmer Sought. Hsv color space clustering based on k-means - programmer sought, 2019.

[41] Bernhard Preim and Charl Botha. *Visual computing for medicine: theory, algorithms and applications*. Elsevier, 2014.

[42] Ignacio Arganda Carreras and David Legland. Morpholibj-distance-transform-watershed-basics.png, Aug 16AD.

[43] Matthew Stewart. Simple introduction to convolutional neural networks, Jul 2020.

[44] Chong Zhou and Randy C. Paffenroth. Anomaly detection with robust deep autoencoders. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '17, page 665–674, New York, NY, USA, 2017. Association for Computing Machinery.

[45] Dor Bank, Noam Koenigstein, and Raja Giryes. Autoencoders, 2020.

[46] Nicolas Camargo-Torres and Jens Brandenburger. Unsupervised deep learning for detection of non-uniform surface defect distributions in flat steel production. In Valentina Colla and Costanzo Pietrosanti, editors, *Impact and Opportunities of Artificial Intelligence Techniques in the Steel Industry*, pages 92–103, Cham, 2021. Springer International Publishing.

[47] L. Gondara. Medical image denoising using convolutional denoising autoencoders. In *2016 IEEE 16th International Conference on Data Mining Workshops (ICDMW)*, pages 241–246, 2016.

[48] Z. Chen, C. K. Yeo, B. S. Lee, and C. T. Lau. Autoencoder-based network anomaly detection. In *2018 Wireless Telecommunications Symposium (WTS)*, pages 1–5, 2018.

[49] S. Kim, Y. Jo, J. Cho, J. Song, Y. Lee, and M. Lee. Spatially variant convolutional autoencoder based on patch division for pill defect detection. *IEEE Access*, 8:216781–216792, 2020.

[50] Muyuan Ke, Chunyi Lin, and Qinghua Huang. Anomaly detection of logo images in the mobile phone using convolutional autoencoder. In *2017 4th International Conference on Systems and Informatics (ICSAI)*, pages 1163–1168, 2017.

[51] Google. The generator - generative adversarial networks — google developers, Apr 2019.

[52] Jesse Davis and Mark Goadrich. The relationship between precision-recall and roc curves. In *Proceedings of the 23rd International Conference on Machine Learning*, ICML '06, page 233–240, New York, NY, USA, 2006. Association for Computing Machinery.

[53] H. Surendra and S. MohanH. A review of synthetic data generation methods for privacy preserving data publishing. *International Journal of Scientific & Technology Research*, 6:95–101, 2017.

[54] Hadi Keivan Ekbatani, Oriol Pujol, and Santi Segui. Synthetic data generation for deep learning in counting pedestrians. *Proceedings of the 6th International Conference on Pattern Recognition Applications and Methods*, 2017.

[55] Saksham Jain, Gautam Seth, Arpit Paruthi, Umang Soni, and Girish Kumar. Synthetic data augmentation for surface defect detection and classification using deep learning. *Journal of Intelligent Manufacturing*, 2020.

[56] Santi Seguí, Oriol Pujol, and Jordi Vitrià. Learning to count with deep object features. *CoRR*, abs/1505.08082, 2015.

[57] Daniel Ward, Peyman Moghadam, and Nicolas Hudson. Deep leaf segmentation using synthetic data. *CoRR*, abs/1807.10931, 2018.

[58] Blender Online Community. Blender - a 3d modelling and rendering package, 2018.

[59] Jonathan LampelinCulture. Cycles vs. eevee - 15 limitations of real time rendering in blender 2.8, 2019.

[60] Blender Online Community. *Blender - a 3D modelling and rendering package*. Blender Foundation, Stichting Blender Foundation, Amsterdam, 2018.

[61] The Apache Software Foundation. Apache license - version 2.0, Jan 2004.

[62] Google Colaboratory, 2018.

[63] The GIMP Development Team. Gimp.

[64] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[65] Keras Team. Tensorflow image data generator.

[66] Curtis Rueden. Fiji imagej.

[67] Johannes Schindelin, Mark Hiner, and Stefan Helfrich. Auto threshold, Apr 2017.

[68] David Legland and Ignacio Arganda-Carreras. Morpholibj, Jul 2014.

[69] Sreenivas Bhattiprolu. bnsreenu/python_for_microscopists, 2020.

[70] Vishnu Kakaraparthi. Xavier and he normal (he-et-al) initialization, Sep 2018.

Appendix A

Neural networks models

U-Net model

```
"""## U-Net Model """

#U-Net
inputs = tf.keras.layers.Input((IMG_WIDTH, IMG_HIGHT, IMG_CHANNELS))

#convert input to float by dividing with 255
s = tf.keras.layers.Lambda(lambda x: x /255)(inputs)

#Contracting path

#first layer in Contracting path and its dropout percentage 1.?? 2.(3,3) is the kernel size 3. the activation function 4. the start weights .5 the padding is the change in image size after kernel
c1 = tf.keras.layers.Conv2D(16, (3,3), activation="relu", kernel_initializer='he_normal', padding='same')(s)
c1 = tf.keras.layers.Dropout(0.1)(c1)
c1 = tf.keras.layers.Conv2D(16, (3,3), activation="relu", kernel_initializer='he_normal', padding='same')(c1)
p1 = tf.keras.layers.MaxPooling2D((2,2))(c1)

#second layer
c2 = tf.keras.layers.Conv2D(32, (3,3), activation="relu", kernel_initializer='he_normal', padding='same')(p1)
c2 = tf.keras.layers.Dropout(0.1)(c2)
c2 = tf.keras.layers.Conv2D(32, (3,3), activation="relu", kernel_initializer='he_normal', padding='same')(c2)
p2 = tf.keras.layers.MaxPooling2D((2,2))(c2)

#third layer
c3 = tf.keras.layers.Conv2D(64, (3,3), activation="relu", kernel_initializer='he_normal', padding='same')(p2)
c3 = tf.keras.layers.Dropout(0.1)(c3)
c3 = tf.keras.layers.Conv2D(64, (3,3), activation="relu",
```

```
    kernel_initializer='he_normal', padding='same')(c3)
p3 = tf.keras.layers.MaxPooling2D((2,2))(c3)

#fourth layer
c4 = tf.keras.layers.Conv2D(128, (3,3), activation="relu",
    kernel_initializer='he_normal', padding='same')(p3)
c4 = tf.keras.layers.Dropout(0.2)(c4)
c4 = tf.keras.layers.Conv2D(128, (3,3), activation="relu",
    kernel_initializer='he_normal', padding='same')(c4)
p4 = tf.keras.layers.MaxPooling2D((2,2))(c4)

#fifth layer
c5 = tf.keras.layers.Conv2D(256, (3,3), activation="relu",
    kernel_initializer='he_normal', padding='same')(p4)
c5 = tf.keras.layers.Dropout(0.3)(c5)
c5 = tf.keras.layers.Conv2D(256, (3,3), activation="relu",
    kernel_initializer='he_normal', padding='same')(c5)

#Expansive path

#layer
u6 = tf.keras.layers.Conv2DTranspose(128, (2,2), strides=(2,2),
    padding='same')(c5)
u6 = tf.keras.layers.concatenate([u6, c4])
c6 = tf.keras.layers.Conv2D(128, (3,3), activation="relu",
    kernel_initializer='he_normal', padding='same')(u6)
c6 = tf.keras.layers.Dropout(0.2)(c6)
c6 = tf.keras.layers.Conv2D(128, (3,3), activation="relu",
    kernel_initializer='he_normal', padding='same')(c6)

#layer
u7 = tf.keras.layers.Conv2DTranspose(64, (2,2), strides=(2,2),
    padding='same')(c6)
u7 = tf.keras.layers.concatenate([u7, c3])
c7 = tf.keras.layers.Conv2D(64, (3,3), activation="relu",
    kernel_initializer='he_normal', padding='same')(u7)
c7 = tf.keras.layers.Dropout(0.2)(c7)
c7 = tf.keras.layers.Conv2D(64, (3,3), activation="relu",
    kernel_initializer='he_normal', padding='same')(c7)

#layer
u8 = tf.keras.layers.Conv2DTranspose(32, (2,2), strides=(2,2),
    padding='same')(c7)
u8 = tf.keras.layers.concatenate([u8, c2])
c8 = tf.keras.layers.Conv2D(32, (3,3), activation="relu",
    kernel_initializer='he_normal', padding='same')(u8)
c8 = tf.keras.layers.Dropout(0.1)(c8)
c8 = tf.keras.layers.Conv2D(32, (3,3), activation="relu",
    kernel_initializer='he_normal', padding='same')(c8)

#layer
u9 = tf.keras.layers.Conv2DTranspose(16, (2,2), strides=(2,2),
    padding='same')(c8)
u9 = tf.keras.layers.concatenate([u9, c1], axis=3)
```

```
c9 = tf.keras.layers.Conv2D(16, (3,3), activation="relu",
    kernel_initializer='he_normal', padding='same')(u9)
c9 = tf.keras.layers.Dropout(0.1)(c9)
c9 = tf.keras.layers.Conv2D(16, (3,3), activation="relu",
    kernel_initializer='he_normal', padding='same')(c9)

#Output
outputs = tf.keras.layers.Conv2D(1, (1, 1), activation='sigmoid')(
    c9)


model = tf.keras.Model(inputs=[inputs], outputs=[outputs])
model.compile(optimizer='adam', loss='binary_crossentropy',
    metrics=['accuracy'])
model.summary()
```

Listing A.1: U-Net models in code

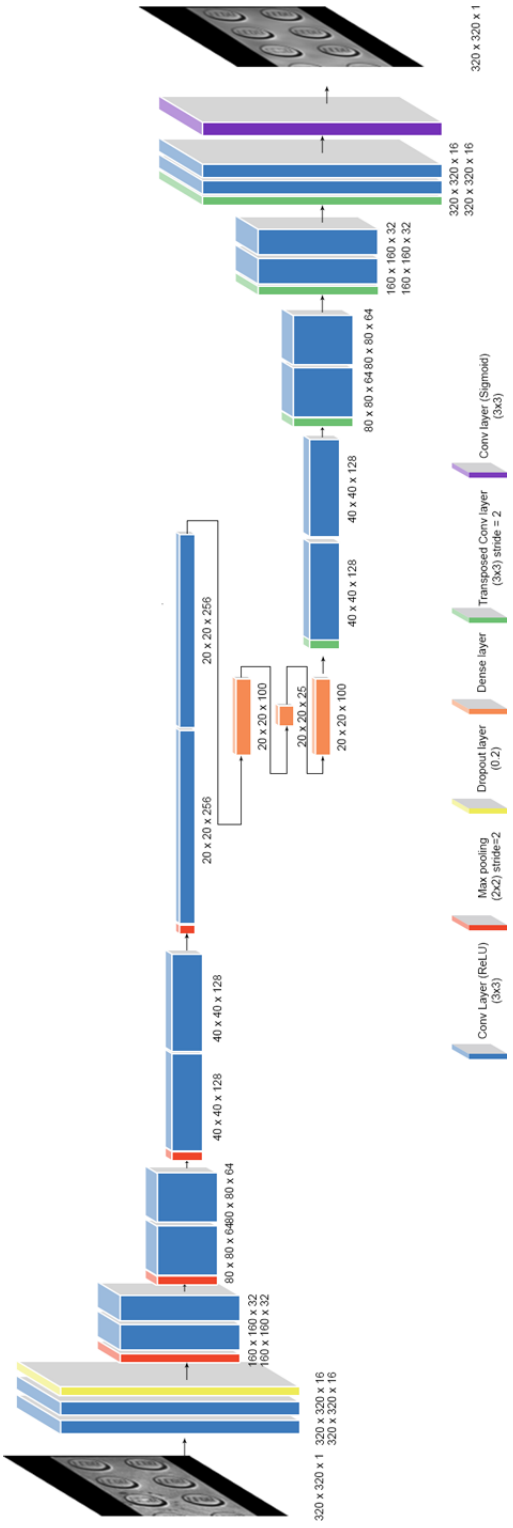
Anomaly Detection model

```
AE_name = 'FinalAE'

input_img = Input(shape=(WINDOW_HEIGHT_PADDED, WINDOW_WIDTH_PADDED
    , 1))


x = Conv2D(16, (3, 3), activation='relu', padding='same')(
    input_img)
x = Conv2D(16, (3, 3), activation='relu', padding='same')(x)
x = Dropout(0.2)(x)
x = MaxPooling2D((2, 2), strides=2, padding='same')(x)
x = Conv2D(32, (3, 3), activation='relu', padding='same')(x)
x = Conv2D(32, (3, 3), activation='relu', padding='same')(x)

x = MaxPooling2D((2, 2), strides=2, padding='same')(x)
x = Conv2D(64, (3, 3), activation='relu', padding='same')(x)
x = Conv2D(64, (3, 3), activation='relu', padding='same')(x)

x = MaxPooling2D((2, 2), strides=2, padding='same')(x)
x = Conv2D(128, (3, 3), activation='relu', padding='same')(x)
x = Conv2D(128, (3, 3), activation='relu', padding='same')(x)

x = MaxPooling2D((2, 2), strides=2, padding='same')(x)
x = Conv2D(256, (3, 3), activation='relu', padding='same')(x)
x = Conv2D(256, (3, 3), activation='relu', padding='same')(x)

x = Dense(100)(x)
x = Dense(25)(x)
x = Dense(100)(x)


x = Conv2DTranspose(128,(3,3), strides=2, padding='same')(x)
x = Conv2D(128, (3, 3), activation='relu', padding='same')(x)
x = Conv2D(128, (3, 3), activation='relu', padding='same')(x)
```

```
33
34 x = Conv2DTranspose(64,(3,3), strides=2, padding='same')(x)
35 x = Conv2D(64, (3, 3), activation='relu', padding='same')(x)
36 x = Conv2D(64, (3, 3), activation='relu', padding='same')(x)
37
38 x = Conv2DTranspose(32,(3,3), strides=2, padding='same')(x)
39 x = Conv2D(32, (3, 3), activation='relu', padding='same')(x)
40 x = Conv2D(32, (3, 3), activation='relu', padding='same')(x)
41
42 x = Conv2DTranspose(16,(3,3), strides=2, padding='same')(x)
43 x = Conv2D(16, (3, 3), activation='relu', padding='same')(x)
44 x = Conv2D(16, (3, 3), activation='relu', padding='same')(x)
45 r = Conv2D(1, (3, 3), activation='sigmoid', padding='same')(x)
46
47
48 autoencoder = Model(input_img, r)
49 autoencoder.compile(optimizer='adam', loss='mse')
50 autoencoder.summary()
```

Listing A.2: Anomaly detection model

CAE overview



Appendix B

Digital Folder

This appendix will be available as a digital download together with the project and will include a short film about the project (AV production), the code used in the project and the gray-scale images received from our collaboration partner. The CAD model received from our collaboration partner used for synthetic data is not included in the appendix, as this data is sensitive information.
Below is an overview of the folder structure for the digital appendix.

- AV production
  - This folder contains the .MP4 file, which is the audio visual production about the project.
- Code
  - Anomaly detection
    - This folder contains the code used in the anomaly detection as a python file and as Jupyter Notebook files
  - U-Net
    - This folder contains the code used in the U-Net, as a Jupyter Notebook file.
  - Evaluation
    - This folder contains the code used for calculating the different elements in the evaluation

- Data
  - Anomaly Detection design
    - This folder contains internal test made for Anomaly detection
  - CSV Files
    - This folder contains comma separated files, which includes anomaly scores for the different images and some statistical calculations
  - OutputImagesMixedData
    - Here you will find the Anomaly Maps, Difference Maps, Reconstructed images and the images used for reconstructing for the dataset of mixed images.
  - OutputImagesRealData
    - Here you will find the Anomaly Maps, Difference Maps, Reconstructed images and the images used for reconstructing for the dataset of real images.
  - OutputImagesSyntheticImages
    - Here you will find the Anomaly Maps, Difference Maps, Reconstructed images and the images used for reconstructing for the dataset of synthetic images.
  - Real data
    - Here you will find 240 folders, each containing an image taken by the collaborator and a mask made in hand by us.
  - Synthetic data
    - Here you will find 664 folders, each containing and image and a mask for the synthetic data
- Evaluation plots
  - This folder contains plots used for the evaluation of the anomaly detection.
- Models
  - Anomaly
    - This folder contains Trained TensorFlow Models for the anomaly detection
  - U-Net
    - This folder contains Trained TensorFlow Models for U-Net