DEVELOPMENT OF IMAGE RECOGNITION SYSTEM FOR STEEL DEFECTS DETECTION

CHEN WAI YANG

UNIVERSITI TUNKU ABDUL RAHMAN

DEVELOPMENT OF IMAGE RECOGNITION SYSTEM FOR STEEL DEFECTS DETECTION

CHEN WAI YANG

A project report submitted in partial fulfilment of the requirements for the award of Bachelor of Engineering (Honours) Electrical and Electronic Engineering

Lee Kong Chian Faculty of Engineering and Science
Universiti Tunku Abdul Rahman

May 2022

DECLARATION

I hereby declare that this project report is based on my original work except for citations and quotations which have been duly acknowledged. I also declare that it has not been previously and concurrently submitted for any other degree or award at UTAR or other institutions.

Signature : 

Name : CHEN WAI YANG

ID No. : 18UEB03052

Date : 8/1/2022

APPROVAL FOR SUBMISSION

I certify that this project report entitled **"DEVELOPMENT OF IMAGE RECOGNITION SYSTEM FOR STEEL DEFECTS DETECTION"** was prepared by **CHEN WAI YANG** has met the required standard for submission in partial fulfilment of the requirements for the award of Bachelor of Engineering (Honours) Electrical and Electronic Engineering at Universiti Tunku Abdul Rahman.

Approved by,

Signature : Chua Kein Huat

Supervisor : Dr Chua Kein Huat

Date : 24/4/2022

Signature :

Co-Supervisor :

Date :

ACKNOWLEDGEMENTS

I would like to thank everyone who had contributed to the successful completion of this project. I would like to express my gratitude to my research supervisor, Dr. Chua Kein Huat for his invaluable advice, guidance and his enormous patience throughout the development of the research.

In addition, I would also like to express my gratitude to my loving parents and friends who had helped and given me technical support and encouragement throughout the whole project.

ABSTRACT

Hot rolled steels are among the highest-demand steels in the construction and manufacturing industry. The manufactured steel inevitably comes with some defects from the production line. Hence, it is essential to conduct a quality control process to ensure the produced hot rolled steels meet the customer's requirements. Currently, most industries rely on human visual inspection systems for quality control. However, this inspection is not efficient and time-consuming. Furthermore, the quality of inspection may differ because different inspectors may have their own judgement on the quality. An image recognition system can improve the quality of hot roll steels and work efficiency. In this project, an image recognition system for steel defects detection has been developed to detect three types of hot rolled steel defects: rusting, edge, and loose wrap. For the rusting detection algorithm, a deep learning model, Single Shot Detector (SSD), was trained to detect and crop the hot rolled steel from the input image for colour detection. The colour detection was implemented to determine the rusting area on the hot rolled steel based on the orange-brown colour that appeared on the hot rolled steel. The system can decide whether to release or hold the hot rolled steel based on the percentage of the rusting area on the hot rolled steel. Meanwhile, the system carries out the model inference by utilizing the trained SSD model to find and crop the Region of Interest (ROI) from the input image regarding edge defects and loose wrap detection. Then, the system conducts Canny Edge Detection to find out the irregular edge lines caused by the defects. The system can determine whether to release or hold the hot rolled steel based on the generated edge lines that indicate its severity. Based on the experimental result, the rusting detection has more than 90% accuracy with less than 50 ms processing time. Besides, the edge defects detection has an average of 69% accuracy with an average 63 ms processing time. Last but not least, the loose wrap detection achieved an average of 84.9 % accuracy with 51.3ms inference time. The detection errors are due to the variety of input images in terms of angle and brightness.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS / ABBREVIATIONS

mAP	mean average precision
FPS	frames per second
IoU	intersection over union

LIST OF APPENDICES

CHAPTER 1

INTRODUCTION

1.1 General Introduction

Steel production is an important industry that has a significant impact on the global economy. Steel is available in various categories, standards, and forms. Based on the World Steel Association lists, there are over 3,500 distinct classes of steel, each with its own set of attributes. Steel's numerous kinds allow it to be widely utilized in construction, products, automobiles, power plants, and other uses (Reliance Foundry Co. Ltd, 2021). Hot rolled steel is one of the categories of steel that is rolled in shape. It has been roll-pressed at a temperature above 1,700 degrees Fahrenheit, well above the recrystallization temperature of most steels. It ensures the steel is simpler to shape, resulting in easier-to-work-with items.

Hot rolled steels always come with some common defects from the manufacturing process. These defects can be classified into plate shape defects, surface defects, appearance defects of the entire coil, geometric dimensions, and composition properties (Metallic Steel, 2020). Defect detection is necessary for maintaining high-quality products for the steel manufacturing industry. However, most factories are still relying on manual defect inspection and data recording. This kind of process is inefficient in terms of time and cost.

Machine vision has become one of the most famous applications of AI for the manufacturing industries. Machine vision systems are a set of interconnected components that are designed to autonomously direct manufacturing and production procedures such as go/no testing and quality control utilising data produced from digital images (Edwards, n.d.). In this project, image recognition is proposed as the solution for hot rolled steel defects detection.

1.2 Importance of the Study

As machine vision is a trend among the manufacturing industries, image processing has become the core technique for this technology. Image processing allows the computer or machine to detect and recognize pre-defined defects without human interference. In IR 4.0, the technology of deep learning enhanced image processing performance by implementing the Convolutional Neural Network (CNN)

architecture. Besides, all the data can be recorded and uploaded to the cloud database automatically. Eventually, it can boost the efficiency of quality-checking during the manufacturing process.

1.3 Problem Statement

The typical defects of hot rolled steel identified during the quality inspection process are rusty, edge dented, edge folded, edge crack, telescoping, loose wrap, etc. As mentioned in the previous section, most defect inspections are done by the workers based on the standard guideline. For example, the industry needs to hold the hot rolled with the telescoping exceed 50mm. Hence, it is necessary to measure the dimension of every hot rolled steel. In addition, there are much more criteria that need to be examined. Therefore, it is a time-consuming process to carry out the quality inspection manually. Moreover, all the data recording process is done manually before transferring it to digital format. Eventually, it requires another extra effort to digitalize the data.

1.4 Aims and Objectives

The project aims to improve the efficiency of hot-rolled steel quality inspection with the image processing technique. The objectives of this project are:

- To investigate the feasibility of the image recognition system for steel defects detection
- To develop an image recognition system for steel defects detection
- To evaluate the performance of the image recognition system

1.5 Scope and Limitation of the Study

This project focuses on the feasible image processing approach to carry out the hot rolled steel defects detection. A prototype of the image recognition system is hoped to be developed at the end of the project.

One of the limitations of the project is the insufficient amount of collected image datasets for the deep learning training purpose. Besides, the processing speed of the image processing system may not achieve the real-time inspection requirement due to the lack of computation resources.

1.6 Contribution of the Study

This project aims to develop an automated steel defects detection system by applying the image processing technique and the deep learning approach. This system can help the steel manufacturing industries to save cost and time on the steel defects identification process because the inspection system generates many outcomes within a few seconds. Besides, this study presented the feasibility of the developed system by discussing the experimental result of the system performance. It can help the future research to improve the system based on information shared by this study.

1.7 Outline of the Report

This report includes five chapters. Chapter 1 has briefed about the hot rolled production and its defects. Chapter 2 provides details about the background of image processing and deep learning approaches. Chapter 3 describes the methodology and work plan for developing the steel defects detection system. Chapter 4 discusses the experimental result for the performance of the developed system. Chapter 5 concludes and provides recommendations for the whole project.

CHAPTER 2

LITERATURE REVIEW

2.1 Introduction to Image Processing

In the late 1960s, NASA's Jet Propulsion Laboratory pioneered digital image processing by converting analogue signals from the Ranger spacecraft to digital pictures enhanced by computers. It is widely used in various applications, including Computed Aided Tomography (CAT) scanning and ultrasounds (Sagar, 2020). Image processing is the implementation of several procedures to an image and enhancing it or extracting useful information from it. Video clips or images act as the input for the image processing technique. The output is the part of the image that matches the information that the user wants to extract (Yolcu, 2020).

Image Processing is primarily concerned with applying and using mathematical functions and transformations to pictures, regardless of whether any intelligent inference is performed on the image. This implies that an algorithm does picture modifications such as sharpening, smoothing, stretching, and contrasting. Figure 2.1 illustrates the operation of sharpening an image. The computing system treats the image as a matrix, and it performs the matrix addition operation between the input image and a predefined detail filter to enhance its edges.

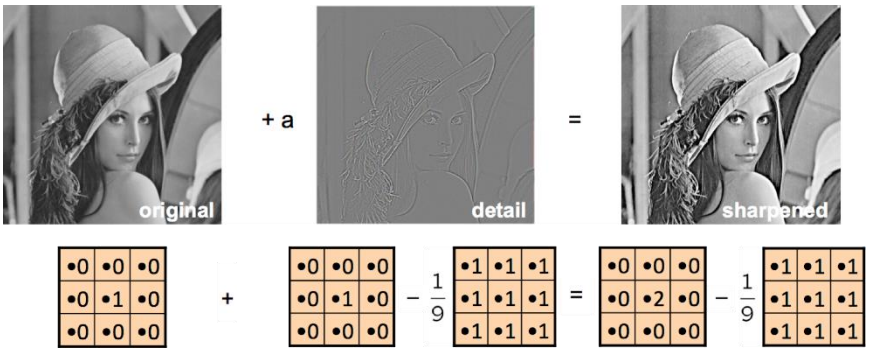


Figure 2.1 Matrix Addition for Image Sharpening (Yeung, n.d.)

These matrices-based modifications are very common in machine learning techniques such as convolutional neural networks (CNN). The CNN convolutes a filter across a picture (another matrix of pixel values) in order to identify edges or colour intensities. Computer vision is a popular topic under image processing in terms of machine learning techniques.

2.1.1 Colour Detection

In image processing, colour detection is detecting and identifying the name of any colour based on the colour model values of the image pixels (Bansal, 2021). A colour model is a mathematical abstraction illustrating how colours might be represented as a collection of integers (Dynamsoft, 2019). Colour models are defined using a coordinate system, with a single point in the coordinate space to represent each colour. There are different colour models, such as the RGB (Red Green Blue) model and the HSV (hue, saturation, value) model.

RGB colour model is the most commonly used colour model that stores values for red, green, and blue layers. These three primary colours can create completely black to white colours by adding their value. For instance, Figure 2.2 shows a hot rolled steel image and RGB model values of part of the pixels. There are three channels which are red, green, and blue that added up together and resulted in a group of grey colour pixels as the grey colour has the RGB values (128,128,128).

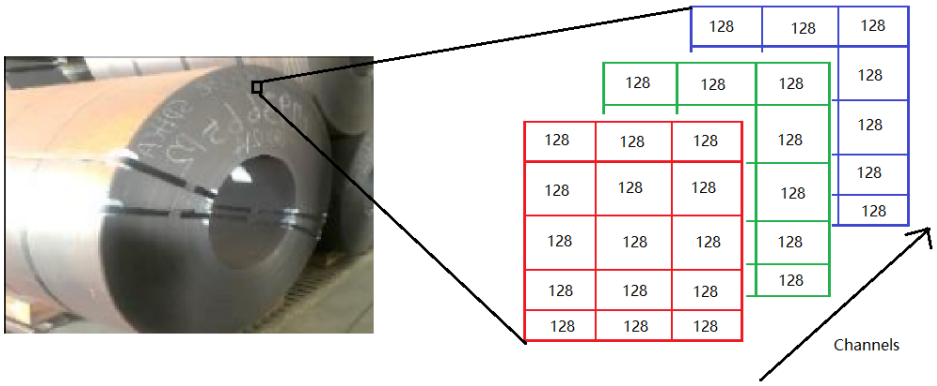


Figure 2.2: RGB model example

HSV, sometimes known as HSB (hue, saturation, brightness), is frequently used by artists because thinking about a colour in terms of hue and saturation is more natural than thinking about additive or subtractive colour components. The system is more similar to people's colour perception and experience than RGB. Colour, shading, and toning are used to communicate hue, saturation, and values in painting. Figure 2.3 shows an example of an image in the HSV colour model.

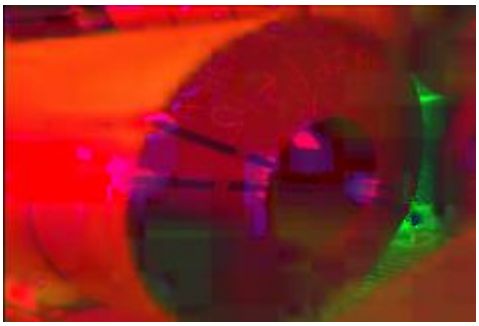


Figure 2.3 : HSV colour model image

The algorithm of colour detection is finding and determining the pixels in an image, whether that is the same colour or colour range as a specified colour. For example, the system needs to detect orange pixels in an image. Hence, it needs to find all the pixels with HSV values around (27,5,96). The HSV colour model is the most commonly used for colour detection as it is more equivalent to human colour perception.

2.1.2 Canny Edge Detection

Edge detection is one of the most fundamental images processing and recognition techniques. An image is an information system, and the edge of its contour provides much information (Zhang, 2010). In the field of computer vision, edge detection is crucial. Edges help in segmentation and object recognition by defining the boundaries between sections in a picture. Multiple operators have been introduced to conduct edge detection in various fields of images. However, not every operator performs well; it depends on image quality factors like lighting, similar-intensity objects, the density of edges in the scene, and noise (Suwanmanee et al., 2013).

There are different types of edge detector and Canny operator, Gaussian Laplacian, Kirsch operator and so on. Canny edge detector is a multi-stage operator for detecting a wide range of edges in images. The Canny edge detection algorithm includes five steps which are noise reduction, gradient calculation, non-maximum suppression, and hysteresis thresholding. Firstly, noise reduction is minimizing the noise on the image. It helps to enhance the performance of edge detection by removing unnecessary information. The Gaussian blur method is the most common noise reduction technique for edge detection. The equation of the Gaussian filter kernel can be written as following where the kernel size is $(2k+1) \times (2k+1)$.

$$H_{ij} = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{(i-(k+1))^2+(j-(k+1))^2}{2\sigma^2}\right); 1 \leq i,j \leq (2k+1) \quad (2.1)$$

Figure 2.4 below shows an example output from the Gaussian blur process. The structure of the output image looks blurry compared to its original input, but the significant edges line has remained.

Figure 2.4: Example of Gaussian Blur image

Furthermore, the gradient calculation phase uses edge detection operators to identify the direction and edge intensity by calculating the image's gradient. The edges in a picture are formed by the changes in the intensity of pixels. The most straightforward technique to detect it is to use filters that highlight the intensity shift in both horizontal (x) and vertical (y) directions. The derivatives Ix and Iy w.r.t. x and y can be calculated to yield the gradient magnitude along the dimensions. It can be realized by using the formula below:

$$\nabla S = \nabla(g * I) = (\nabla \mathrm{g}) * I \quad (2.2)$$

$$\nabla S = \begin{bmatrix} \mathrm{g}_x \\ \mathrm{g}_y \end{bmatrix} * I = \begin{bmatrix} \mathrm{g}_x * I \\ \mathrm{g}_y * I \end{bmatrix} \quad (2.3)$$

$$\nabla g = \begin{bmatrix} \frac{\partial \mathrm{g}}{\partial \mathrm{x}} \\ \frac{\partial \mathrm{g}}{\partial \mathrm{y}} \end{bmatrix} = \begin{bmatrix} \mathrm{g}_x \\ \mathrm{g}_y \end{bmatrix} \quad (2.4)$$

g or g(x,y) = Gaussian filter or kernel

I = Image

Figure 2.5 illustrates an example of the Gaussian derivative, which is applied on the Gaussian blurred image from the previous step for X-derivative, Y-derivative, and gradient magnitude.

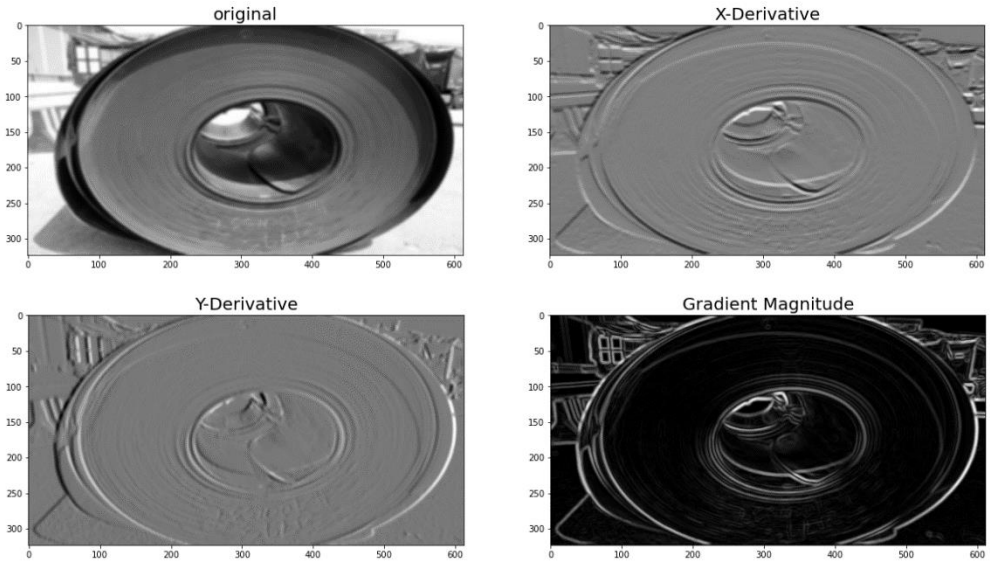


Figure 2.5 : Example of Gaussian Derivative

Generally, few spots along an edge improve the visibility of the edge. As a result, the edge points that do not significantly contribute to feature visibility can be discarded. The Non-Maximum Suppression (NMS) approach is used to achieve the same goal. The technique traverses the gradient intensity matrix in all directions and finds the pixels with the highest value in the edge directions. For example, Figure 2.6 shows the edge with three edge points. Assume that point (x,y) has the most significant edge gradient. Examine the edge points perpendicular to the edge and determine whether their gradient is less than (x,y). If the values are less than the (x,y) gradient, the non-maxima locations along the curve can be suppressed.

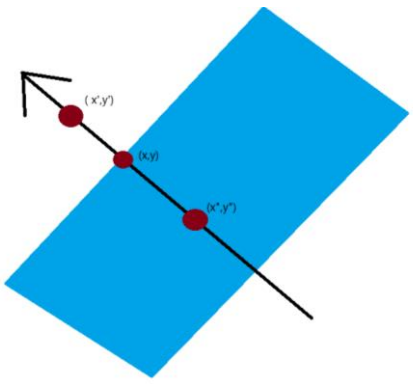


Figure 2.6 : Illustration of NMS

Lastly, hysteresis thresholding is the process of identifying three types of pixels which includes robust, weak, and non-relevant, then carrying out the threshold process. Strong pixels are those with an intensity high enough to be sure they contribute to the final edge. Weak pixels have an intensity value that is not high enough to be called strong, but it still needs to be considered. Other pixels are ignored since they are irrelevant to the edge. There are two thresholds which are the high threshold and the low threshold. The strong pixels are identified using a high threshold. The non-relevant pixels are specified using a low threshold. Basically, any pixel with an intensity that falls between the two thresholds is classified as weak.. Then, the hysteresis mechanism assists in determining which pixels are potentially strong and which are considered irrelevant.

After the NMS and thresholding process, the output of the Canny edge detection is shown in Figure 2.7 below which is the right-side image. The edge lines are more precise and significant than the gaussian derivative output.

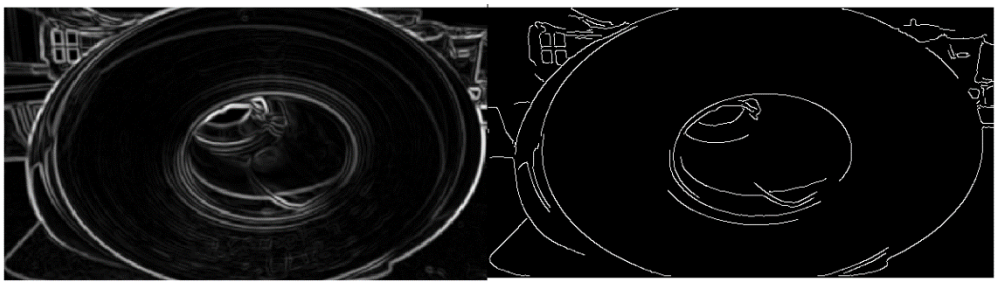


Figure 2.7 : Final Output of Canny Edge Detection

2.2 Computer Vision

Computer vision is derived from the modelling of image processing through machine learning methods. Computer vision makes use of machine learning to recognize patterns in pictures in order to understand them. Similar to the visual reasoning process in human eyesight, human can differentiate between things, categorize them, and arrange them based on their size. Also, computer vision is similar to the image processing as it accepts pictures as input and outputs information about their size, colour intensity, and other characteristics (Sagar, 2020).

One of the most popular applications regarding to the computer vision is self-driving vehicles. Automobile-mounted cameras capture footage from a range of viewpoints from around the vehicle, which is then fed into the image processing

application, which analyses the image data to determine road boundaries and traffic signs, as well as identify other vehicles and objects in the vicinity and, as well as human beings. It can be feasible for the self-driving car to navigate streets and highways on its own, avoiding hazards and transporting its passengers to their destination in a safe and efficient manner (Dickson, 2019).

Additionally, computer vision is vital for facial recognition applications, which utilize technology to recognize the identities of the people's faces in the images. By using computer vision algorithms, algorithms for identifying and comparing facial features in images are identified and compared to databases of facial profiles. Face recognition technology is used in consumer gadgets to authenticate the identity of its owners, such as smartphones, smart locks etc. Also, social networking programs employ face recognition technique to recognize and tag persons in their feeds. Additionally, police enforcement organizations employ facial recognition technology to identify offenders in real-time video broadcasts of their operations (Dickson, 2019).

As a result of augmented and mixed reality, which allows computing devices such as smartphones, and smart glasses to overlay and embed virtual things on real-world pictures, computer vision is essential in many of these mentioned applications. Augmented reality gear identifies objects in the real world and uses computer vision to determine where a virtual object should be displayed on a device's display screen. When it comes to identifying planes such as tabletops, walls, and floors, computer vision algorithms may help augmented reality apps immensely. This is important because it allows them to build depth and dimension while also placing virtual objects in their actual surroundings (Dickson, 2019).

2.2.1 Object Detection

Object detection is one of the techniques under computer vision technology. It includes the process of detecting and identifying different types of objects in images as well as videos. The main operations of object detection are object classification and object localization. Object classification is a technique in which a computer system attempts to predict and determine an object in an image. Figure 2.8 shows a process of classifying cat-dog images. The input images, which is only consisting of a cat or dog, are randomly sent to a trained system in order to go through the process of feature extraction. The trained system sends out the outputs with a label for each

input image based on the extracted data. However, the output of the object classification is solely the class of object. It does not contain any information related to the location of the object. Hence, object detection needs to carry out another task which is object localization.

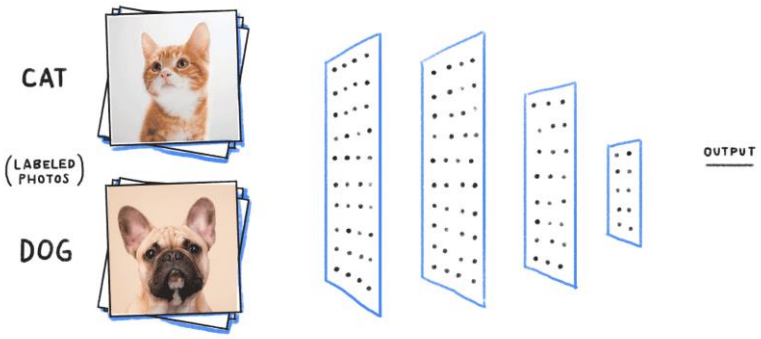


Figure 2.8 : Cat-Dog Classification Process (S. Sharma, 2019)

Object localization is the process of locating the detected object in the image with the bounding box as the indication. For instance, an image with a car is inputted into the system, which has been trained for car object detection tasks. Other than object classification, the system performs object localization to predict the height and width of the detected car in the image. Hence, the output of the object localization has four values which are pixel x-coordinate, pixel y-coordinate, height, and width. With these values, a bounding box can be drawn around the car in the image as shown in Figure 2.9.

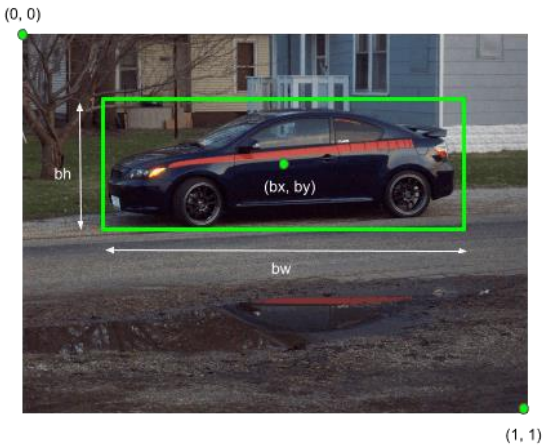


Figure 2.9 : Object Bounding Box (Halbe, 2021)

2.3 Deep Learning

Deep learning is a field of machine learning that focuses on algorithms that are inspired by the structure and function of the brain. Deep learning is becoming increasingly popular. Artificial neural networks are the algorithms that are used to create these networks (Brownlee, 2020b). Deep learning is a technique that mimics the way the human brain processes data and generates patterns for use in decision-making. Deep learning has been adopted in different fields of applications such as Natural Language Processing (NLP), Computer Vision (CV), voice recognition, predictive model, and etc. Deep learning models are sometimes referred to as deep neural networks due to the fact that most of the deep learning methodologies make use of topologies of neural networks. When describing the number of hidden layers in a neural network, the term "deep" is frequently employed. Unlike standard neural networks, which contain only a few hidden layers, deep neural networks can have up to 150 layers buried within them. In order to teach deep learning models, large volumes of labelled data are used in conjunction with the neural network, which automatically extracts information or features from the input data.

In the computer vision field, convolutional neural networks (CNN) are often used in deep neural networks. A CNN concatenates the studied features with input data and employs two-dimensional convolutional layers, making this architecture well-suited for two-dimensional processing data, such as pictures. In general, the CNN consists of three hidden layers, including Convolutional layers, Pooling layers, and fully connected layers (Gurucharan, 2021). Figure 2.10 shows a general architecture of the Convolutional Neural Network.

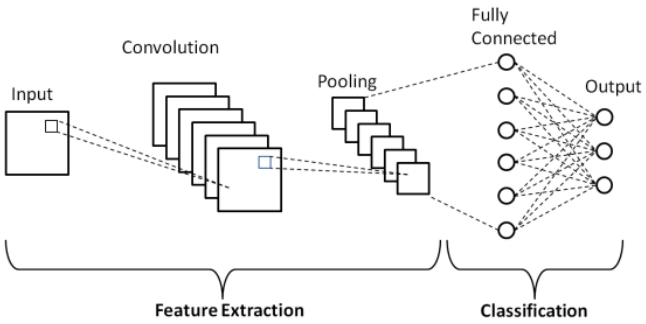


Figure 2.10 : General Structure for CNN (D. J. Sharma et al., 2020)

Convolutional layers use two input layers which are a portion of the original picture, and a filter of equal size called the kernel. This layer produces the dot product of its two inputs. Pooling is a technique for down sampling data. The Pooling Layer takes an input (a picture) and lowers its pixel count. This may be accomplished in two ways: via max pooling or through min pooling. Max pooling selects the highest value within the chosen area, while min pooling selects the lowest value within the selected region. As the name implies, fully connected layers connect all of one layer's outputs to the inputs of another layer. Eventually, these layers combined together to form a model which aids in data classification (Bansari, 2021).

While CNN works well when evaluating a single picture, it lacks one critical property. It considers just spatial and visual data, disregarding temporal, and time characteristics, such as how a frame is linked to the preceding frame. This is where Recurrent Neural Networks, abbreviated RNN, enter the picture. The term 'recurrent' implies that the neural network performs the same tasks on a sequence-by-sequence basis. Additionally, RNNs may be utilized in Natural Language Processing.

2.3.1 Loss Function

Loss function can be treated as one of the main cores of deep learning algorithms. It is an approach to evaluate how good is the deep learning model has been trained. A high loss function output value is meaning that the deep learning model has low accuracy in terms of predictions. Hence, the loss function is the way to tell that how much more improvement is needed for the algorithm. When it comes to deep learning, one of the most important stages is the design of loss functions to solve the specific job. In fact, there is various type of loss function to study. However, for object detection, the loss function can be classified into two categories in general which are classification loss and regression loss.

2.3.1.1 Classification Loss

Classification loss is beneficial for any job that needs categorization. When given k categories, it must make certain that the model performs well in categorizing x number of samples over k categories, which is the task. In the ImageNet competition, for example, there are 3300 pictures divided into 500 distinct categories, and the goal is to categorize each picture as one of the distinct classes.

Cross-Entropy loss is a measure of the amount of information that has been lost. When referring to cross-entropy loss, the terms "logistic loss", "log loss", and "cross-entropy" are often used interchangeably. Cross-entropy may be seen from two different perspectives. One is based on information theory, while the other is based on probabilistic view (Dudeperf3ct, 2019)

Using the cross-entropy function, it measures the similarity between the prediction of the model with the real label, which represents the actual probability distribution. If the projected probability score for the real category is near 0, the cross-entropy will increase significantly. However, as the accuracy of the forecast increases, the value of the cross-entropy decreases. In the case of perfect prediction, i.e., in the case the projected distribution is identical to the actual distribution, this value becomes 0.

2.3.1.2 Binary Classification

Binary classification is a kind of classification that is binary in nature. In accordance with the name, there are only two classes of categories (Dudeperf3ct, 2019). If there are two classes that are needed to categorize the input pictures, it is suitable to utilize binary cross-entropy to do it. Those predictions that are confident yet incorrect are severely penalized by cross-entropy loss. Let y^* as the anticipated output of the model, and y is the real value. For K example, the formula of the binary cross-entropy is expressed in the form of,

$$L_{BCE} = -\frac{1}{K}\sum_{i=1}^{K}(y_i log y_i{}^* + (1 - y_i)\log(1 - y_i{}^*)) \quad (2.5)$$

2.3.1.3 Multi-class Classification

For Multi-class classification, if the categories are above two classes which are needed to be recognized on the pictures, hence it should implement the multi-class classification cross-entropy function. The Softmax activation in the output layer of neural networks is utilized as a loss function in these networks. Softmax is a mathematical function that converts a vector of integers to a vector of probabilities. The value of the possibilities is proportional to the vector's relative scale. It can convert a vector of numbers into a vector of probabilities (Brownlee, 2020a). The

probability that an example belongs to each class is determined by the model. The multi class classification cross entropy can be written as following where C > 2.

$$L_{MCE} = -\sum_{C=1}^{C}(y_C ln y^*_c) \quad (2.6)$$

2.3.1.4 Regression Loss

In regression, the model produces a numerical value. In order to get a measure of error, it must first compare the output number to the anticipated value. For example, the investors are interested in predicting the values of a factory in the surrounding area. As a result, they can provide the model with various characteristics (such as the number of production areas, the number of toilets, the area, and so on) and apply the model to estimate the cost of the factory.

2.3.1.5 Mean Squared Error

The mean squared error is a measure of how accurate a measurement is. It is easy to construct these error functions. For this error function, it will be using the square of error; then it gets the mean of these squared error functions to achieve the result (Grover, 2021). It is solely focused on the average volume of mistakes, regardless of the direction in which they occur. In contrast, predictions that are far off the mark in relation to actual values are severely punished as a result of the squaring procedure. This kind of mistake is referred to as L1 loss. Assume y^* is the anticipated output from the model and that y is the actual value. For the K number of training, the formula of MSE loss can be written as follows:

$$L_{MSE} = \frac{1}{K}\sum_{i=0}^{K}(y_i - y^*)^2 \quad (2.7)$$

2.3.1.6 Mean Absolute Error

Mean Absolute Error (MAE), which is similar to the one before, takes into account the relative error difference between the goal and projected output. In the same way as MSE does, this quantifies the number of mistakes without taking account of their direction. The distinction is that MAE is more resistant to outliers than square since it does not rely on the square function. This kind of mistake is referred to as L1 loss.

$$L_{MAE} = \frac{1}{K}\sum_{i=0}^{K}|y_i - y^*| \quad (2.8)$$

2.3.1.7 Root Mean Square Error

The root mean square error can be calculated by using the square root of the LMSE. The MSE penalises big mistakes more severely than small errors, and as a result, it is very sensitive to outliers. To prevent this, it is often to utilise the squared root version of the formula.

$$L_{RMSE} = \sqrt{\frac{1}{K}\sum_{i=0}^{K}(y_i - y^*)^2} \tag{2.9}$$

2.3.2 Faster r-CNN

Faster Region-Based Convolutional Neural Networks, also known as Faster r-CNN, is one of the popular object detection architectures that applies convolution neural networks (Khazri, 2021). It was introduced in 2015 by Ross Girshick, Shaoqing Ren, Kaiming He, and Jian Sun. The Faster r-CNN comprises three parts which are convolution layers, Region Proposal Network (RPN), and Classes and Bounding Boxes prediction. The convolution layers train the filters for feature extraction purposes by feeding the network a sufficient amount of image dataset. Besides, the Region Proposal Network (RPN) is a tiny neural network that slides over the final feature map of the convolution layers and predicts if an item exists or not, as well as its bounding box. Lastly, the Classes and Bounding Boxes prediction uses another Fully connected neural network to carry out classification of objects and regression by taking regions proposed by the RPN as inputs.

Qirui Ren et. al (2019) had proposed a real-time detection of steel strip surface defects with Faster r-CNN architecture as the CNN model backbone. The proposed network is named Slighter Faster R-CNN because it can achieve 0.05s average processing time for one image with 98.32 % accuracy. Based on their experiment results, it is slightly faster than the original Faster r-CNN network, which has an average speed of 0.2s per image as shown in Table 2.1. The Slighter Faster R-CNN was constructed by adding the depth wise separable convolution. It is a technique for significantly reducing computation time and model size by factorizing a conventional convolution into depth wise and pointwise convolutions. Depth wise convolution applies a single convolutional filter to each input channel, whereas pointwise convolution creates a linear combination of the depth wise convolution

output. As a result, the proposed network inference time is 0.15s or 75% faster than the original Faster R-CNN. However, the drawback of the constructed network is its mean Average Precision (mAP) is 1.19% lower than Faster R-CNN.

Table 2.1 : Results from the Qirui Ren team research

	mAP	Recall	Inference time per image
Faster R-CNN	99.02 %	97.55 %	0.2 s
Proposed Network	97.83 %	96.67 %	0.05 s

2.3.3 YOLO

You Only Look Once (YOLO) is a famous algorithm used for real-time object detection by processing images or videos (Aggarwal, 2021). This algorithm is applying the convolutional neural networks (CNN) for real-time object detection. According to the term "You Only Look Once," when it comes to object identification, the approach only takes one forward propagation through a neural network. A single algorithm run is sufficient to predict the characteristics of an entire picture, indicating that the method is efficient. With the help of a CNN algorithm, it is possible to forecast several class probabilities and bounding boxes simultaneously. The YOLO algorithm comprises several different versions, such as Yolo tiny, Yolov3, Yolov4, and YOLOv5.

The algorithm of the YOLO can be described into three parts: Residual blocks, Bounding box regression, and Intersection Over Union (IoU). Firstly, the YOLO algorithm begins by dividing an image or frame into a grid of squares with dimensions. Figure 2.11 illustrates how is the grid cells apply to an image. Objects that occur within a grid cell can be detected by every grid cell.

Figure 2.11 : Example of image with grid (Karimi, 2021)

Suppose that an object centre occurs within a certain grid cell, and that cell is responsible for detecting the object. Then, some numbers of bounding boxes are predicted with its corresponding score of confidence. All the bounding boxes have attributes such as width, height, class (for example, car, cat, dog, etc.), and the centre of the box. Figure 2.12 shows the example of bounding box around a car which including the information for width, height, and centre point coordination.

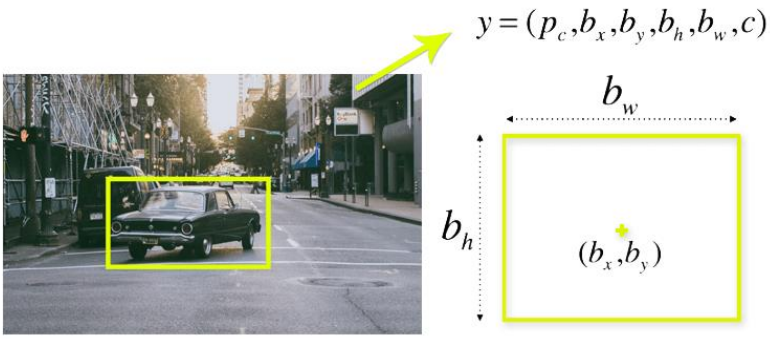


Figure 2.12 : Example of image with bounding box (Maj, 2018)

Moreover, the phenomena in object detection describe how boxes overlap when they are intersected over union (IoU). Figure 2.13 shows an example of IoU condition on a cat detection image. YOLO takes advantage of IoU to create an output box that surrounds the items in the scene. As every grid cell can predict the bounding boxes and the confidence ratings associated with them, it allows YOLO to eliminate bounding boxes that are not identical in size to the actual bounding box based on the confidence score. There are two bounding boxes, one in blue and the other one in red.

The predicted box is represented in blue colour, while the red box is the actual box. YOLO need to make sure that both boxes are match.

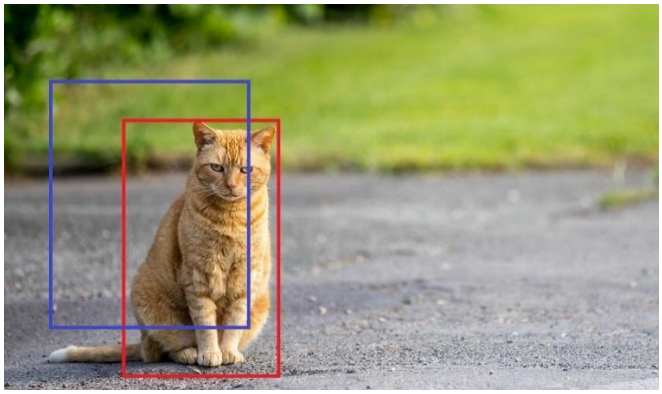


Figure 2.13 : Example of image with IoU (Rakshit, 2021)

There is a proposed algorithm by Yu Zhang et. Al (2020) for using the YOLO-tiny network to build the hot rolled steel strip defect detection system. YOLO-tiny network is a compact or optimized version under the YOLO family, which has a shallow network layer compared to the other members of the series. The convolutional self-encoder, also known as CAE in short, is implemented in their solution to work as the compression pre-processing framework. It replaces undifferentiable quantization with smooth approximation, uses Gaussian scale mixture (GSM) to estimate entropy, and allows for rate control by changing the number of channels in the encoder's final convolution layer. The model is developed and trained using the defect data set of the hot rolled steel strip surface from Northeastern University (NEU), which comprises 1800 pictures categorized into six labels, each class including 300 images, divided into six categories. Their surface has crazing patches, pitted surfaces, rolled-in scale, and scratched surfaces. Figure 2.14 shows the comparison result between the YOLO-tiny and the proposed model. The proposed model or RYOLO-tiny has lower fps than the YOLO-tiny, but its mAP is higher than YOLO-tiny mAP when the pixel depth is equal and more than 0.235.

Pixel depth	Network	FPS	mAP	Cr AP	In AP	Pa AP	Ps AP	Rs AP	Sc AP
0.122	YOLO-tiny	166	91.6	89.8	91.2	94.9	90.5	89.1	94.1
	RYOLO-tiny	47	90.8	88.6	90.1	94.5	89.5	88.0	94.1
	IYOLO-tiny	125	89.5	87.1	89.2	92.8	88.7	85.4	93.6
0.235	YOLO-tiny	166	91.6	89.8	91.2	94.9	90.5	89.1	94.1
	RYOLO-tiny	43	92.7	90.9	92.4	95.6	92.7	89.9	94.8
	IYOLO-tiny	125	92.0	90.6	91.4	95.2	91.7	89.0	94.3
0.481	YOLO-tiny	166	91.6	89.8	91.2	94.9	90.5	89.1	94.1
	RYOLO-tiny	40	93.4	91.9	92.9	95.6	92.7	91.9	95.5
	IYOLO-tiny	111	92.5	90.4	92.3	95.4	92.1	89.6	95.1
0.983	YOLO-tiny	166	91.6	89.8	91.2	94.9	90.5	89.1	94.1
	RYOLO-tiny	35	94.5	93.6	93.8	96.9	93.4	92.6	96.8
	IYOLO-tiny	111	93.3	90.8	92.7	96.4	92.3	91.5	96.3

Figure 2.14 : Research result from Yu Zhang research team on Yolo approach Source: Zhang et al., 2020

2.3.4 Single Shot MultiBox Detector

Single Shot MultiBox Detector (SSD) is an approach of the object detection that detects several objects in a single shot. It was released at the end of November 2016 and set new benchmarks in terms of performance and precision. Its object detection tasks were performed at a high level of precision and performance, with more than 74% of mean Average Precision (mAP) achieved at speeds of 59 frames per second (FPS) by processing the datasets such as PascalVOC and COCO (Dash, 2019). Table 2.2 shows that SSD has higher speed and mAP than the Faster R-CNN because SSD accelerates the process by removing the requirement for a region proposal network, which is required in the Faster R-CNN algorithm (Hui, 2020). SSD implements several enhancements to make up for the loss inaccuracy, including multi-scale features and default boxes. It allows the SSD to match the accuracy of the Faster R-CNN while working with lower quality pictures, significantly increasing the system's speed.

Table 2.2 : Comparison on different CNN

CNN architecture	Mean Average Precision (mAP)	FPS	Number of Boxes	Dimension of Input
Faster R-CNN (VGG16)	73.20	7.0	~6000	~1000 x 600
YOLO (customized)	63.40	45.0	98	448 x 448

SSD300* (VGG16)	77.20	46.0	8732	300 x 300
SSD512* (VGG16)	79.80	19.0	24564	512 x 512

SSD object detection can be separated into two parts which are feature extraction and convolution filters. Figure 2.15 illustrates the simplified SSD Network Architecture which consists of the input image and also the convolutional neural network. In terms of extracting feature maps, SSD implements VGG16 which was a convolutional neural network model suggested by K. Simonyan and A. Zisserman from the University of Oxford in the publication "Very Deep Convolutional Networks for Large-Scale Image Recognition." Next, it uses the 38 × 38 Conv4_3 layer to detect and predict objects.

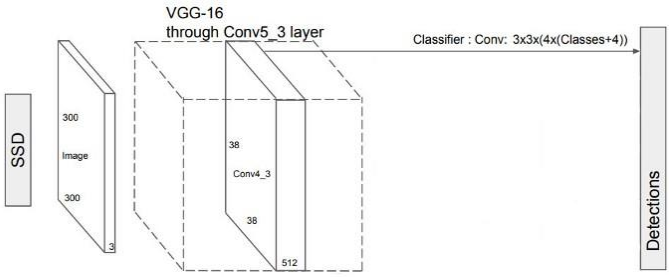


Figure 2.15 : Simplified SSD Network Architecture (Hui, 2020)

Each prediction comprises a boundary box and 21 scores for each class, with the highest score determining which class the bounded item belongs to in each case. SSD reserves the class "0" to signify that it does not have any objects. As mentioned, the SSD does not apply to the region proposal network; instead, small convolution filters are used to compute the scores for both the location and the class. SSD makes predictions for each cell after extracting the feature maps and applying three-way convolution filters to each cell.

SSD predictions are divided into two categories: positive matches and negative matches. When evaluating the cost of localization, SSD only considers positive matches. The match is positive whenever the default boundary box (as opposed to a projected boundary box) has an IoU greater than 0.5 with the ground truth. In all other cases, it is negative. It is crucial to highlight that the intersection

over the union (IoU), also known as the intersection over the intersection, is the ratio between intersected and linked areas between two regions.

For the loss function, the mismatch between the ground truth box and the projected boundary box is called localization loss (L_{loc}). SSD only penalizes predictions that result in a successful match. Ideally, it would want the forecasts from the positive matches to get closer to the actual results. Negative matches may be disregarded if they are not significant. It can be assumed that the l is the predicted box, g as the ground truth box. Meanwhile, the cx,cy as the offset to the default bounding box d of width w and height h. Eventually, the equation for localization loss can be expressed as below:

$$L_{loc}(x,l,g) = \sum_{i\in Pos}^{N} \sum_{m\in\{cx,cy,w,h\}} x_{ij}^{k}\, smooth_{L1}\,(l_i^m - \hat{g}_j^m) \qquad (2.10)$$

$$\hat{g}_j^{cx} = (g_j^{cx} - d_i^{cx})/d_i^w \qquad \hat{g}_j^{cy} = (g_j^{cy} - d_i^{cy})/d_i^h$$

$$\hat{g}_j^w = \log\left(\frac{g_j^w}{d_i^w}\right) \qquad \hat{g}_j^h = \log\left(\frac{g_j^h}{d_i^w}\right)$$

$$x_{ij}^p = \begin{cases} 1\ if\ IoU > 0.5\ between\ default\ box\ i\ and\ ground\ true\ box\ j\ on\ class\ p \\ 0\ other\ wise \end{cases}$$

Besides, the loss of confidence (L_{conf}) is the inability to make a class forecast correctly. For every good match prediction, it penalizes the loss based on the confidence score of the relevant class in the forecast. If there is no item identified by the confidence score of class "0," it punishes the loss based on the confidence score of class "0." Class "0" identifies no object as being detected. Let assume c as the class score for multiple classes confidences. The formula for the loss of confidence can be written as below:

$$L_{conf}(x,c) = -\sum_{i\in Pos}^{N} x_{ij}^p \log\left(\hat{c}_i^p\right) - \sum_{i\in Neg} \log(\hat{c}_i^0)\ \ where\ \ \hat{c}_i^p = \frac{\exp\,(c_i^p)}{\sum_p \exp\,(c_i^p)} \qquad (2.11)$$

Hence, the loss function after combined both L_{conf} and L_{loc} can be written as below:

$$L(x,c,l,g) = \frac{1}{n}(L_{conf}(x,c) + \propto L_{loc}(x,l,g)) \qquad (2.12)$$

The number of positive matches is denoted by n, while the weight for the localization loss is denoted by $\propto$.

2.4 Summary

Image processing and deep learning convolutional neural network is the core of developing the automated defects detection system. Image processing can be used to extract important information from pictures. Hence, it can be utilized to detect some defects which are not complicated, such as the rusting defects, as they can be recognized based on colour. However, the other defects, such as the edge crack and folded, may not be detected easily with ordinary image processing techniques. So, deep learning can be the way to solve this issue. Based on the literature review, there are types of CNN architecture that can be implemented. For the defects detection system, optimum accuracy and processing speed is important to achieve a feasible real-time application. SSD is chosen as the main CNN model as it has high FPS and optimum mean average precision.

CHAPTER 3

METHODOLOGY AND WORK PLAN

3.1 Introduction

In this project, three main types of defects have been addressed, namely rusting, edge defects, and loose wrap. The methodology for detecting the mentioned defects can be separated into two main parts, namely deep learning modelling and the development of the image processing system for defects detection. Deep learning modelling is a phase of data collection, training and evaluating the trained model. Single Shot Detector (SSD) is chosen as the framework for training the deep learning model. The development of detecting a defect image processing system consists of three approaches for different types of defects. Hence, the input images have to go through three sub-function blocks in evaluating each defects condition. The flow of the overall defect detection system is shown in Figure 3.1.

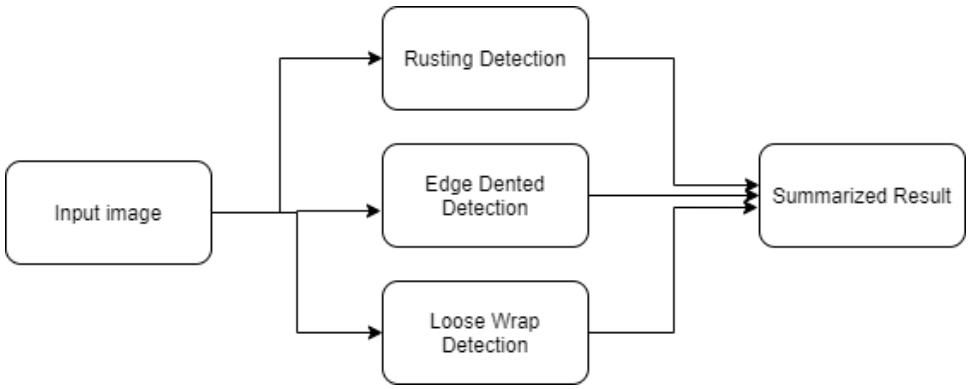
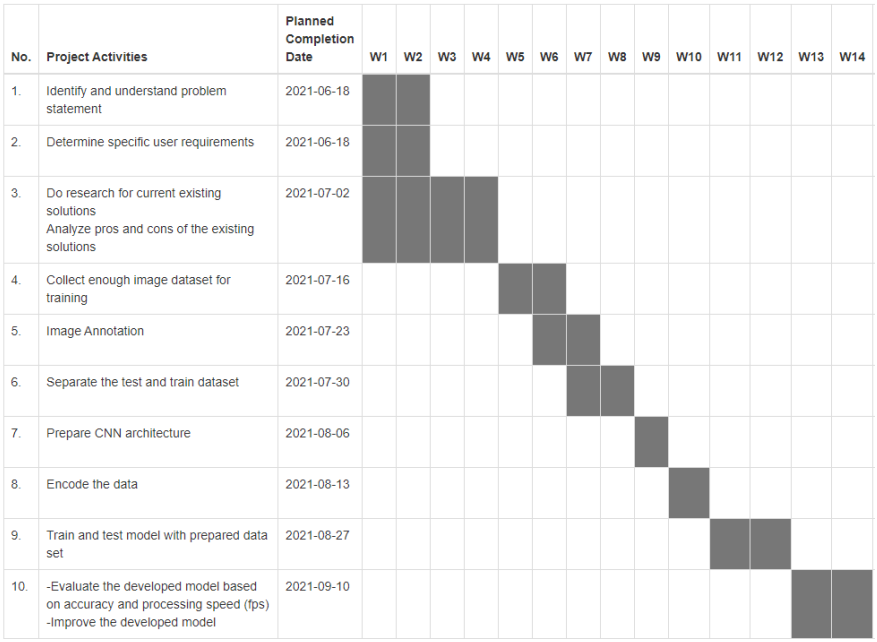


Figure 3.1 : Overall defect detection system

Figure 3.2 and Figure 3.3 show the project Gantt Chart for the first and second trimesters. Identifying the problem and understanding the user requirements have been done at the beginning phase of the project. Then, the in-depth literature review on the existing solutions was carried on for the first month. After that, it is required to collect and annotate a sufficient amount of dataset, which is needed for the deep learning model training in the further stage. After pre-processing the image data, the CNN architectures were prepared and trained with the image data. Lastly,

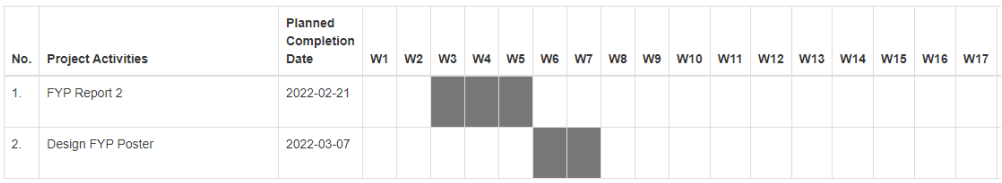
the trained model must be evaluated based on its performance, such as accuracy and processing speed.



No.	Project Activities	Planned Completion Date	W1	W2	W3	W4	W5	W6	W7	W8	W9	W10	W11	W12	W13	W14
1.	Identify and understand problem statement	2021-06-18														
2.	Determine specific user requirements	2021-06-18														
3.	Do research for current existing solutions Analyze pros and cons of the existing solutions	2021-07-02														
4.	Collect enough image dataset for training	2021-07-16														
5.	Image Annotation	2021-07-23														
6.	Separate the test and train dataset	2021-07-30														
7.	Prepare CNN architecture	2021-08-06														
8.	Encode the data	2021-08-13														
9.	Train and test model with prepared data set	2021-08-27														
10.	-Evaluate the developed model based on accuracy and processing speed (fps) -Improve the developed model	2021-09-10														

Figure 3.2 : Gantt Chart for first semester

Since most of the technical tasks for the system development have been done during the first trimester, the second trimester will only focus on the report writing and the FYP poster design work.



No.	Project Activities	Planned Completion Date	W1	W2	W3	W4	W5	W6	W7	W8	W9	W10	W11	W12	W13	W14	W15	W16	W17
1.	FYP Report 2	2022-02-21																	
2.	Design FYP Poster	2022-03-07																	

Figure 3.3 : Gantt Chart for second semester

3.2 Requirements

The primary programming language chosen for application development is Python for this project. Python is a general-purpose programming language interpreted at a high degree of abstraction. It is required to set up a software environment for developing the defect detection algorithm. The software requirements are the Spyder IDE, Google Colab, and the essential tools libraries.

The Spyder IDE is a free integrated development environment (IDE) that allows the users to develop Python algorithms such as data collection, data analysis, data visualization, image processing, etc. It offers a one-of-a-kind mix of a complete programming tool's sophisticated editing, research, debugging, and profiling skills with data exploration, interactive execution, and attractive visualization capabilities.

Google Colab is a web-based IDE for Python that allows anybody to create and run unlimited python code using the browser. It is suitable for data analysis, machine learning, and deep learning. It is because Google Colab offers free hardware resources such as central processing unit (CPU), graphics processing unit (GPU) as well as Tensor Processing Unit (TPU) to execute the python code online. It benefits the deep learning developer to train the CNN network model, which is a process that requires high intensive usage of GPU or TPU.

The essential libraries include Tensorflow, OpenCV, and labelImg, which is used during the development process. Tensorflow is a free and open-source artificial intelligence library that constructs models using data flow graphs. Using this technique, programmers may create large-scale neural networks with many layers. Classes, perception, understanding, discovering, prediction, and creation are among the most common applications for TensorFlow. Besides, OpenCV is a library for computer vision and image processing applications. OpenCV can be downloaded for free from GitHub. It can be used to process pictures and videos for detecting items, faces, and even handwriting. In addition, LabelImg is an open-source image annotation tool. Image annotation is a vital process for data collection that needs to be done before training the deep learning object detection model.

3.3 Deep Learning Modelling

The basic workflow for developing the deep learning model is shown in Figure 3.4. Two deep learning models are required for different purposes in the defects detection system: hot rolled steel detection and edge defects detection.

The hot rolled steel detection model only consists of 1 class of the object, "hot rolled steel." This model is implemented in the rusting detection and loose wrap detection. The purpose of doing this is to allow the system to find the Region of interest (ROI) from the image for further steps of image processing such as colour detection and Canny Edge detection.

Besides, the edge defects detection model is used in locating and classifying the types of edge defects. The included classes are edge crack, edge dented, and edge folded. Like hot rolled steel detection, this model can help the system determine the ROI of edge defects detection, which can be used for Canny Edge processing to evaluate its severity. It is explained more in the other sections.

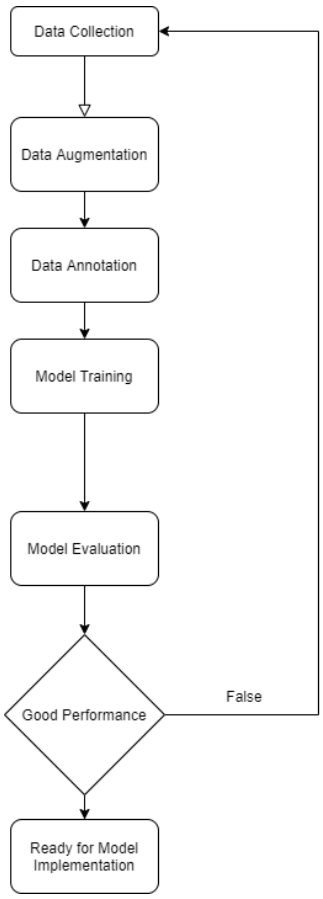


Figure 3.4 : Flow Chart of Deep Learning Modeling Process

Firstly, a sufficient amount of image data has to be collected. The recommended amount of image data is about 1000 to 2000 images for each class. However, it is a challenge to collect such an amount of data from the industry in addition to the pandemic condition. Hence, data augmentation is necessary for increasing the current collected data amount. It is possible to expand the quantity of data by slightly modifying the original data or by creating new synthetic data from current data using some techniques. For example, data augmentation can be done by slightly adjusting its brightness and making another copy. Also, noises can be added to the images and create replicas. The purpose of doing these is to mimic that the

images are collected under different conditions. The techniques used for this project are adjusting the brightness, rotation, and adding noises to the pictures. A comparison example between the original image and its augmented image with noises is shown in Figure 3.5.

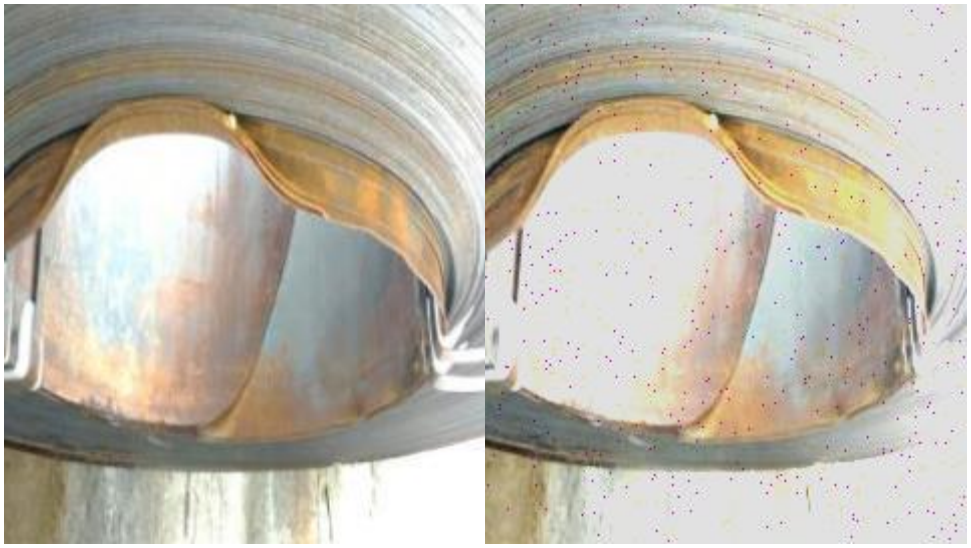


Figure 3.5 : Comparison of original image(left) and augmented image (right)

After getting enough data, the data annotation is the next step before training the model. Data annotation is the classification and labelling of data for artificial intelligence applications. Training data must be correctly classified and annotated to be useful in a particular use case. It is a way to prepare the learning material for the machine to learn how to recognize and detect objects. In this project, the data annotation tool which has been used is called labelImg. It is an open-source python GUI application capable of exporting the annotation files in COCO, XML, and YOLO format. A data annotation example by using labelImg is shown in Figure 3.6. It labels the edge dented defects on one of the collected image data.

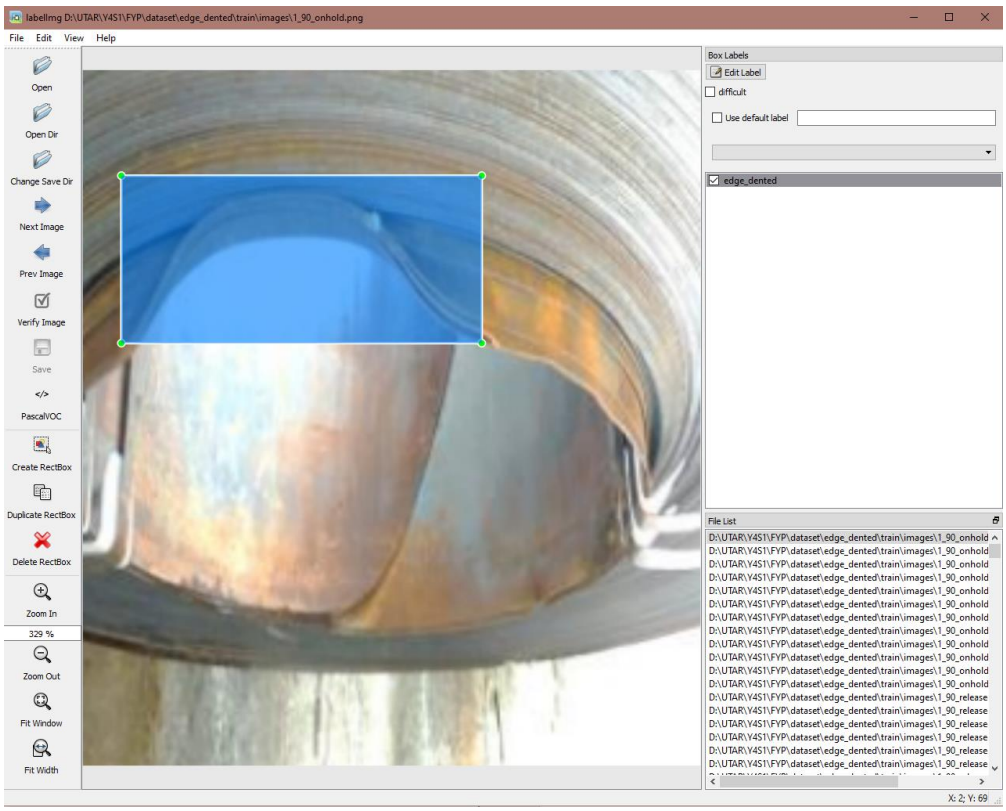


Figure 3.6 : LabelImg GUI Annotation Example

Next, it is required to prepare the software environment and tool to train the deep learning model with all the annotated images. Google Colab is chosen as the model training platform because it is accessible in terms of software and hardware. The GPU offered by the Google Colab platform is Tesla V4. It is good enough for the deep learning model training task. Besides, the model training library uses Tensorflow with SSD Mobilenet as the model framework. The trained model can be evaluated based on its mean average precision and loss function value. If the model's performance is not achieving the expected result, it may need to retrain by inserting more data.

3.4 Rusting Detection

The designed workflow for rusting detection is shown in Figure 3.7. The mechanism can be separated into the three sections which are the hot rolled steel detection, rusting surface detection and hold/release decision making.

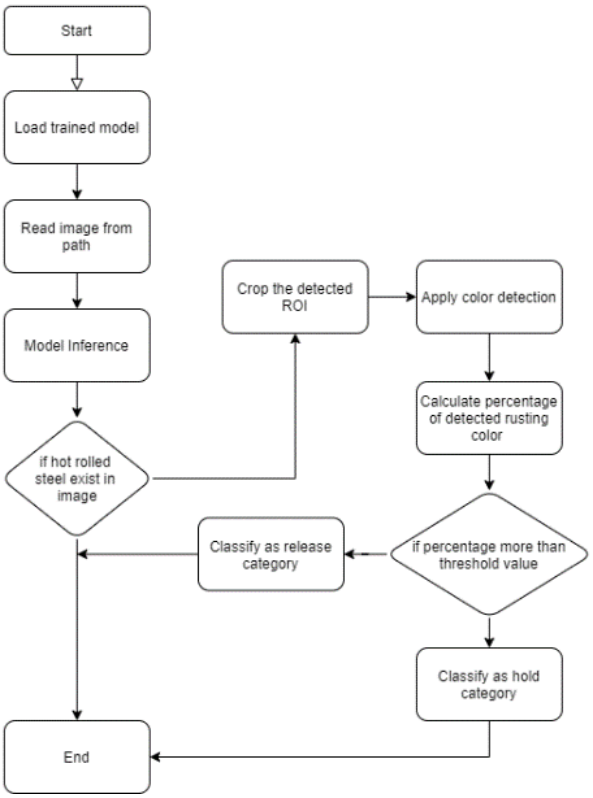


Figure 3.7 : Flow Chart of Rusting Detection

3.4.1 Hot rolled Steel Detection

The object detection technique is applied for the hot rolled steel detection to remove the unrelated image data from the image, such as the background, before carrying out the colour detection and detected rust percentage calculation. It can improve the accuracy of the classification process for the hold and release classes. The trained SSD model is implemented in this step.

Firstly, the program loads the trained model before the model inference. The model inference can be carried out after an input image is read from its directory path and converted into an appropriate data format. At this stage of the operation, an input image is fed into the detection algorithm using the OpenCV image read function. The input image is converted digitally into a width × height pixels of NumPy array data type with 3-channels representing the BGR (blue green red). This NumPy data is

used to carry out the further steps of image processing. Figure 3.8 illustrates one of the input images examples.

Figure 3.8 : Sample Input Image of Rusty Hot Rolled Steel

From the output of the model inference, information such as the number of instances, classes of detected instances and coordinates of the bounding box around the instances can be retrieved. This information allows the program to capture the detected hot rolled steel as the region of interest (ROI) for colour detection. Figure 3.9 shows an example image where the hot rolled steel is detected by the trained SSD model. The detected hot rolled steel is cropped and serve as the input for the next step.

Figure 3.9 : Detected Hot Rolled Steel

3.4.2 Rust Surface Detection

As rust is a type of corrosion that results in the orange-brown colour coats on the surface of the metal, it is more effective and less computational efforts by using the colour detection methodology. In the program, a range of colour spectrum for rusting

(reddish-brown colour) is pre-set. Wherever the pixel of the input image is within the range, it is considered as a part of the rust region.

In this part, the cropped hot rolled steel is the input data or ROI to carry out the colour detection. The program then carry out the colour conversion to convert the BGR-channel of the image data into HSV (Hue, Saturation, Value) by applying the colour conversion function from the OpenCV. In contrast to RGB or BGR, which utilizes primary colours, HSV is more closely related to how people see colour.

Hue is representing the colour portion of the image in degrees from 0 to 360. For example, the hue value of the red colour is between 0 to 60 degrees, and the green colour is between 121 to 180 degrees. Besides, the saturation is about how much is the grey pixels introduced into the images. For instance, the lower the saturation, the more grey pixels are introduced, resulting in a faded image. Moreover, the Value is about the brightness of the colour, and its value is ranged from 0 to 100 percent. The reason for using the HSV colour space is due to the fact that the R, G, and B components of an item's colour in a digital picture are all linked with the quantity of light that hits on the object, and therefore with each other, making image descriptions in terms of those components difficult to understand. Descriptions in terms of hue or saturation are often more appropriate than descriptions in terms of colour. Figure 3.10, it shows one of the example images after being converted to HSV colour space.

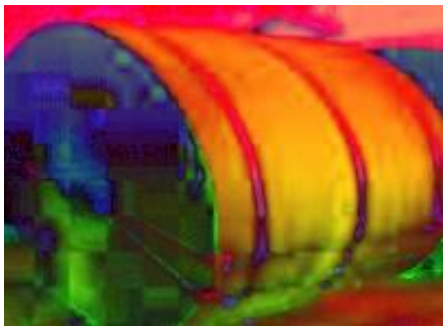


Figure 3.10 : Converted HSV Image

The converted HSV image data is then utilized to proceed with the colour detection by carrying out the thresholding operations for determining wherever the image pixel is within the range of HSV colour space that would like to be detected. For the orange-brown rust colour, the predefined range of HSV colour space is from

(0,36,23) to (33,255,255). This operation results in binary image data where the rusty part of the steel is white in colour while the rest are black. A sample output of the operation is shown in Figure 3.11.

Figure 3.11 : Sample of Threshold Image

3.4.3 Hold/Release Decision Making for Rusting Condition

The hold/release decision-making operation determines whether the hot rolled steel can be released or need to be held. The hot rolled steel can be released if the rusty condition is not severe. This classification can be done by the calculation of the rust percentage on the hot rolled steel surface. If the calculated percentage value is above the preset threshold value, the hot rolled steel will be classified as the on hold category. Else, the hot rolled steel with less than 30% of the rusty surface will be released.

Based on the outputted binary image data from the previous step, the percentage of the rust on a hot rolled steel surface can be calculated based on the number of white pixels. The percentage can be calculated with the formula:

$$Rust\ Percentage = \frac{amount\ of\ white\ pixels}{total\ amount\ of\ pixels}$$

$$= \frac{amount\ of\ white\ pixels}{height\ of\ image\ pixels \times width\ of\ image\ pixels} \qquad (3.1)$$

It is important to know how severe the rusting corrosion on the steel surface is because it acts as the gauge to evaluate whether the rusty steel can be released for the production phase. According to the inspection operation of the hot rolled steel company, the inspectors are obligated to find out all the rusty steel and classify which steels should be on hold or release based on the rusting condition. Hence, by calculating the percentage of rust, it can automate this kind of inspection process.

3.5 Edge Defects Detection

Edge defects on the hot rolled steel are shape defects where the edge sides of the hot rolled steel are deformed due to damages. It includes edge crack, edge dented, and edge folded. The flow chart for edge defects detection is shown in Figure 3.12. Similar as the rusting detection, the approach can be separated into the three sections which are the defects detection, Canny edge detection and hold/release decision making.

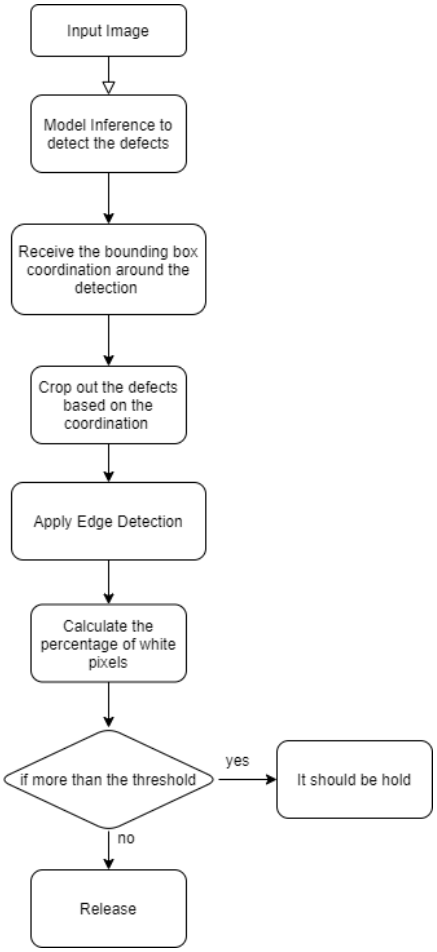


Figure 3.12 : Edge Defects Detection Flow Chart

3.5.1 Edge Defects Localization and Classification

At the initial process, an input image with hot rolled steel is read and converted into NumPy array data format for image processing. Then, the program needs to load the trained SSD model capable of edge defects detection. The model is used to carry out the localization and classification process. For example, Figure 3.13 has shown a result in which the model locates the detected defect on the hot rolled steel with a

bounding box and classify it as the edge folded defect. This process is similar to hot rolled steel detection for rusting detection. It locates and categorises the classes of edge defects from the image and crop the ROI for Canny edge detection.

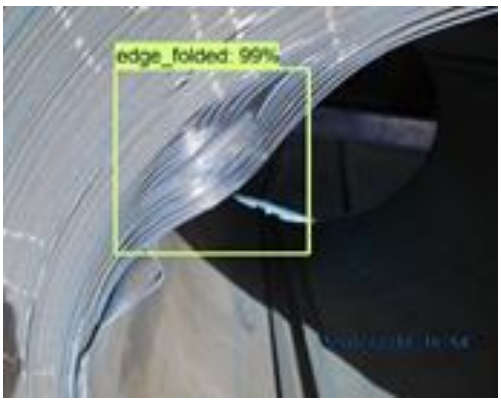


Figure 3.13 : Sample of the Localization and Classification Result

3.5.2 Canny Edge Detection for Edge Defects

As the edge defects will form irregular edge lines on the hot rolled steel, the Canny edge detection can be applied to detect the edge lines from the image. Canny edge detection is one of the edge detection operator in OpenCV library. Figure 3.14 shows the output of the Canny edge detection after processed the cropped sample image from Figure 3.13. Based on the output, it is noticeable that the white pixels lines are appearing at the region where the edge folded is occurred. Meanwhile, the region without edge folded is showing black pixels in the output. Hence, the severity of the edge defects can be inspected by referring to the amount of the output white pixels from Canny edge detection.

Figure 3.14 : Output of Canny Edge Detection for Edge Defect

3.5.3 Hold/Release Decision Making for Edge Defects Condition

In the previous section, it is known that the the severity of the edge defects can be inspected by referring to the amount of the output white pixels from Canny edge detection. So, the program can decide wheher to hold or release the hot rolled steel with edge defects based on the percentage of the white pixels within the output image from the Canny edge detection:

$$Edge\ Defect\ Percentage = \frac{amount\ of\ white\ pixels}{total\ amount\ of\ pixels} \tag{3.2}$$

3.6 Loose Wrap Detection

Loose wrap of the hot rolled steel is when the layers of steel sheets become loosely between each other, caused by the broken packaging belt. An example of the hot rolled steel with loose wrap issue is shown in Figure 3.15. It can be seen that the packing belt, which was initially used to tighten the hot rolled steel, is broken. Hence, the steel sheets are further from each other, and the hot rolled steel looks larger.

Figure 3.15 : Hot rolled steel with Loose Wrap issue

As the loose wrap issue will also produce irregular edge lines on the hot rolled steel, the loose wrap detection algorithm is similar to the edge defects detection. The only distinction between them is the method to extract the ROI image. In this loose wrap detection, the hot rolled steel detection is implemented. The flow chart for loose wrap detection is shown in Figure 3.16. The approach can be separated into three sections: hot rolled steel detection, Canny edge detection, and hold/release decision making.

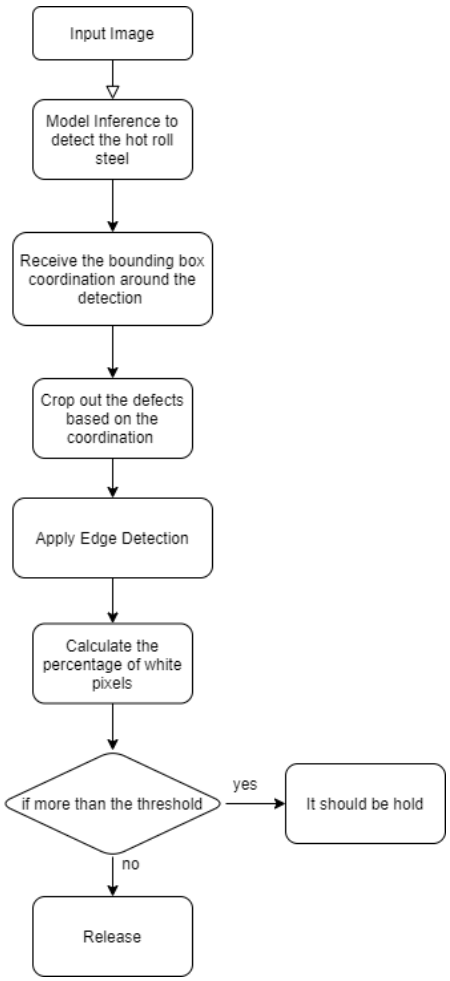


Figure 3.16 : Loose Wrap Detection Flow Chart

3.6.1 Hot rolled Steel Detection

This is the same process in the rusting detection to extract the ROI from the image for further image processing step. In this loose wrap detection, the cropped ROI is used for Canny edge detection to find out the severity of the loose wrap issue of the hot rolled steel. Figure 3.17 shows the example of hot rolled steel detection.

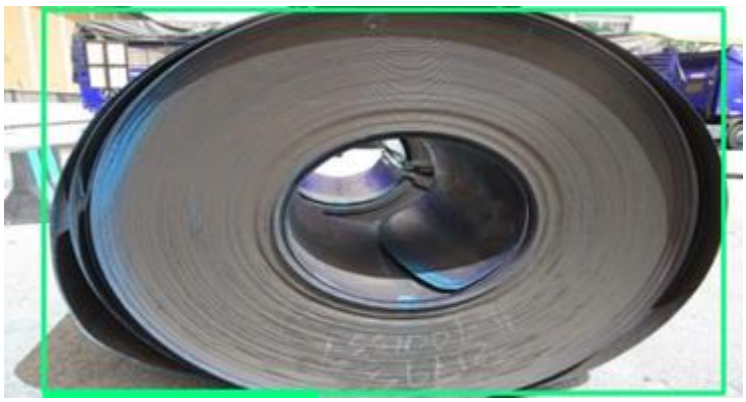


Figure 3.17 : Output of Canny Edge Detection for Loose Wrap

3.6.2 Canny Edge Detection for Loose Wrap

Loose wrap issue will also form irregular edge lines on the hot rolled steel, so the Canny edge detection can be implemented. Figure 3.18 shows the output of the Canny edge detection.

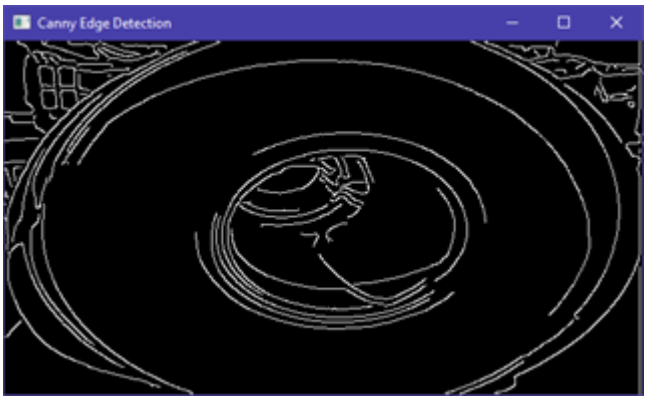


Figure 3.18 : Output of Canny Edge Detection for Loose Wrap

3.6.3 Hold/Release Decision Making for Loose Wrap Condition

Similar as the edge defect detection, the hold/release classification can based on the percentage of the white pixels :

$$Loose\ Wrap\ Percentage = \frac{amount\ of\ white\ pixels}{total\ amount\ of\ pixels} \quad (3.3)$$

3.7 Summary

The workflow of the rusting defects of the hot rolled steel can be easily carried out since it does not require any training process. It can save a lot of computational requirements compared to the deep learning approach. However, the hot rolled steel does not only consist of the rusting issue, so the deep learning approach is required to tackle the rest of the defects, such as classifying the types of edge defects. Furthermore, the Canny edge detection is implemented in the detection process for the edge defects and loose wrap. It helps to evaluate the severity of the defects based on the extracted irregular edges created by the defects themselves. Hence, with this methodology, the system can decide to hold or release the hot rolled steel with defects.

CHAPTER 4

RESULT AND DISCUSSION

4.1 Introduction

In this chapter, the result of the performance for three types of defects detections which are rusting detection, edge defects detection and the loose wrap detection is presented and discussed. For all types of detections, the final output of a process is to classify an image into the “hold” or “release” category. The performance is studied in terms of accuracy and average processing time. In the experiment, the accuracy is the quantity of the correctly classified images over the total input images in percentage. Besides, the average processing time is the average duration from inputting an image to the system until the system successfully classifies the image. The cause of the faulty detection is investigated and discussed in this chapter as well.

4.2 Rusting Detection Performance

To investigate the performance of the rusting detection algorithm, the collected image data and its augmented image data are used to conduct a test on classifying the “release” and “hold” categories. The “release” category means that the rusting condition of the hot rolled steel is acceptable and ready to be released for usage. Meanwhile, the “hold” category indicates that the rusting condition with higher severity, and the hot rolled steel shall behold. In the rusting detection algorithm, the rusting severity is based on the percentage of the total detected rusty area on the hot rolled steel.

In the experiment, the size of the input images are ranged from 119×139 pixels to 651×408 pixels.The input images were fed into the written Python script that carry out the process as shown in the Figure 3.7. There were also output images with green colour layer on the detected rusting region exported by the script. It can help to identify the location as well as the area of the detected rusting region by the system. The system will classify a hot rolled steel with “hold” class if the rusty area exceeds 60%. The performance of this system is studied in terms of accuracy and processing time.

4.2.1 Rusting Detection Test Result

In the test, 372 images have been fed into the system and generate the results in an excel file. Table 4.1 shows the result of the rust detection and the decision making for hold and release conditions. Based on the generated result, 228 images belong to the on-hold condition, while 144 images are good to be released. The accuracy for the hold and release are 96.05% and 97.92%, respectively. The average accuracy is calculated by using the formula below:

$$Average\ accuracy = \frac{Total\ Corrected\ Classified\ Images}{Total\ Images} \times 100 \qquad (4.1)$$

Table 4.1: Performance of Rust Detection

	Classification		**Accuracy**	**Average Inference Time (ms)**
	Success	Failure		
Hold	219	9	96.05%	49.8
Release	141	3	97.92%	

Figure 4.1 and Figure 4.2 show the examples of input and output classified as the hold and release categories, respectively. The green layer on the output images acts as the indicator for the detected rusting region by the system.

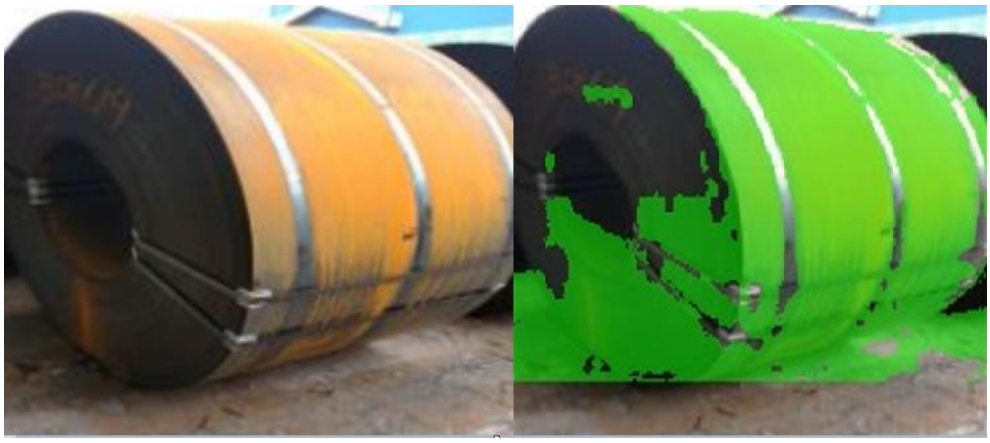


Figure 4.1: An example input (left) and its output (right) classified as hold category

Figure 4.2 : An example input (left) and its output (right) classified as release category

4.2.2 Discussion for Rusting Detection Performance

Based on the performance of rusting detection in Table 4.1, the detection accuracy is more than 90%, which can be considered highly accurate. However, there are still some wrongly classified images during the detection process. The remaining errors are due to the input images being taken from different angles, environment and brightness conditions. It will affect the accuracy of an image in presenting the colours. If the brightness value is too high in an image, some colours might not be noticeable by the system. In addition, if the background or the environment contains a similar colour as the rusting colour, the system might wrongly detect the background as one of the rusting regions on the hot rolled steel. Eventually, this can affect the accuracy of the colour detection to identify and calculate the percentage of rusting region. For example, Figure 4.3 shows one of the input images, which is wrongly classified as “hold” category.

Figure 4.3 : Example of wrongly classified input image

Figure 4.4 presents the output image with a green layer on the detected rusting region for Figure 4.3. It can be observed that the background of the hot rolled steel (metal deck roof and wall) contains some green layer regions. This is because the HSV values of the background in the image is within the range of the predefined rusting HSV range from (0,36,23) to (33,255,255). In other words, the colour of the background is nearly close to the rusting colour. Hence, it was incorrectly identified as the rusting region of the hot rolled steel.

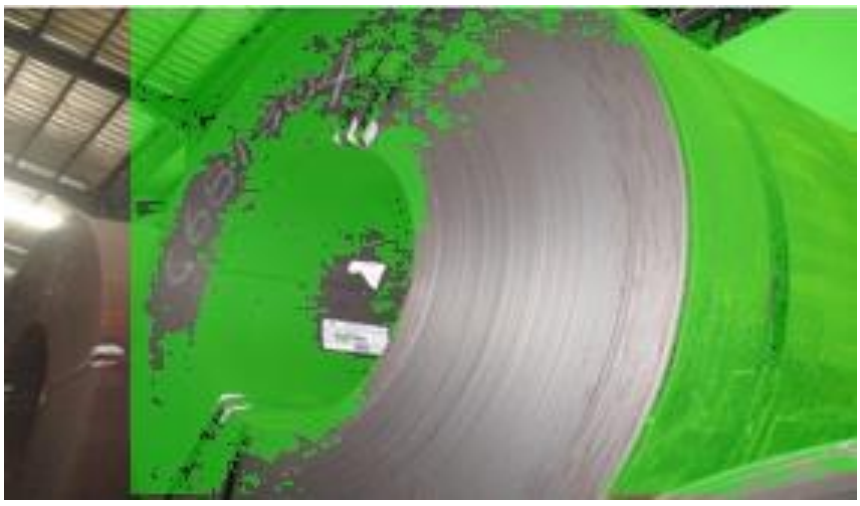


Figure 4.4 : Example of wrongly classified output image

Furthermore, the result for the average inference time or average image processing time shows 49.8 ms. It also indicates that the system can process around

$\frac{1}{(49.8/1000)sec} = 20$ images per second. Hence, the developed rusting detection system can help the steel factory inspect over 72000 hot rolled steel in one working hour for rusting defect.

4.3 Edge Defects Detection

The performance of the edge defects detection can be studied by carrying out the similar testing process in Section 4.2. In this edge defects detection algorithm, there are three types of edge defects that are needed to be tested which are edge dented, edge crack and edge folded. The collected image data for each type of the edge defects is tested on classifying "hold" and "release" categories. The classification is also based on the severity of the defects by evaluating the percentage of generated edge line from the Canny edge detection. Also, its average inference time is also recorded for each defect.

4.3.1 Edge Defects Detection Test Results

For this edge defects detection performance test, there are 885 images including edge crack, edge dented, and edge folded were used as the testing dataset. Table 4.2 shows the result of the edge defects detection and the decision making for hold and release conditions. The calculation of the accuracy is also using the formula (4.1) in the rusting detection test. According to the result, the accuracy for the edge crack is 100 %. Meanwhile, the edge dented is having the 94.61% and 42.44% accuracy for hold and release category respectively. In terms of the edge folded, its accuracy for hold and release are 52.98% and 55.95% respectively.

Table 4.2 : Performance of Edge Crack

Edge Defects		**Classification**		**Accuracy**	**Average Inference Time (ms)**
		Success	Failure		
Edge Crack	Hold	184	0	100%	74.4
Edge Dented	Hold	158	9	94.61%	50.0

	Release	59	139	42.44%	
Edge Folded	Hold	89	79	52.98%	64.5
	Release	94	74	55.95%	

4.3.2 Discussion for Edge Defects Detection Performance

According to Table 4.2, all the edge crack images were successfully classified without error. Based on the inspection manual from the steel factory, the edge crack has only the “hold” category, which means the hot rolled steel is needed to behold if any edge crack defects are detected from the input image. The trained model has successfully detected all the input images with edge crack defects. For example, Figure 4.5 shows an output image in which the hot rolled steel consists of an edge crack defect. The output image contains a yellow bounding box that localizes the detected defect's position.

Figure 4.5 : Example of an output image with edge crack defect

In addition, the release classification for edge dented defects is less than 50% in terms of accuracy. Also, the accuracies of hold and release classification for the edge folded is near to 50% as well. The errors are mainly due to the collected images for edge dented and edge folded cases being taken under different angles, brightness conditions, and distance. The Canny edge detection output depends on the detail and structure presented in an image. The variety of distance and angle of taking the photos can significantly impact the sharpness of the visible edge lines. For example,

Figure 4.6 presents the images of the edge folded defects taken from different distances and angles. It is noticeable that the edge lines of the image on the right side are much more visible than the left-side image. The right-side picture is taken nearer from the hot rolled steel; hence, it can capture more details than the left-side image. Eventually, the hold/release classification accuracies for the edge dented and edge folded were reduced.

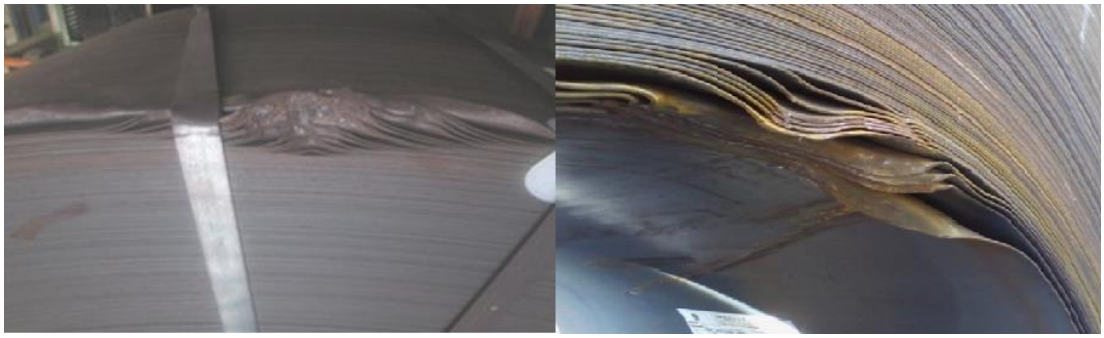


Figure 4.6 : Images of edge folded defects taken from different distances

Moreover, the average inference time for the edge defects detection is ranged from 50 ms to 74.4 ms for different types of defects. The inference time of an image is depending on the image size. The higher the image size, the larger the amount of the data pixels in an image. Hence, it requires more time to process an image. This developed edge defects detection system has the capability to process minimum $\frac{1}{(74.4/1000)sec} = 13$ images per second with image size around 290×136 pixels.

4.4 Loose Wrap Detection

A similar performance test from the previous section is also conducted for the loose wrap detection. The input images were fed into the developed system for hot rolled steel detection using the trained SSD model. Then, the severity of the loose wrap detection is also evaluated by the percentage of generated edge line from the Canny edge detection of the detected hot rolled steel within the images.

4.4.1 Loose Wrap Detection Test Result

In this performance test, the amount of the testing dataset images is 73. The range of the image size is from 280×151 to 612×323 pixels. Based on Table 4.3, it shows the result of the accuracy for the hold category is 69.81 % while the release category is 100%.

Table 4.3 : Performance of Loose Wrap Detection

	Classification		**Accuracy**	**Average Inference Time (ms)**
	Success	Failure		
Hold	37	16	69.81%	51.3
Release	20	0	100%	

4.4.2 Discussion for Loose Wrap Detection Performance

According to the result, the “hold” category contains 16 failed categorized images which caused the 69.81% accuracy. It is also because the input images have different angles, brightness, and background, which affect the accuracy of the Canny edge detection. The previous section shows that the distance and angle can affect the visibility of the edge lines in an image. For this loose wrap detection, the background also impacts the accuracy of the classification process. For example, Figure 4.7 below shows the Canny edge detection output. It is noticed that some of the white pixels or detected edge lines from the background (red bounding boxes) do not belong to the hot rolled steel. However, the system mistakenly includes these edge lines into the severity evaluation for the loose wrap defect. Hence, it can affect the accuracy of the loose wrap severity calculation.

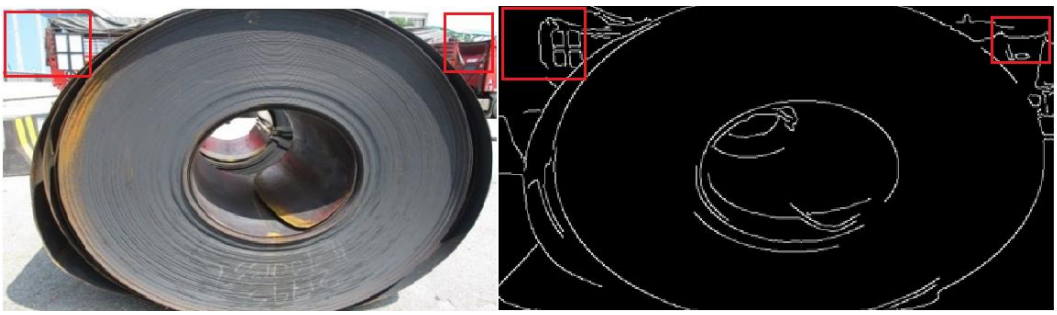


Figure 4.7 : Example of Canny Edge Detection with edge lines from background

For the average inference time, the system takes about 51.3 ms to process an image. It proved the system has the ability to carry out the inspection for about $\frac{1}{(51.3/1000)sec} = 19$ images or hot rolled steels per second.

4.5 Summary

This chapter discussed about the performance of proposed solution for inspecting the three types of defects on the hot rolled steel. It is noticed that the rusting detection has the greatest performance in terms of the accuracy. It is because the rusting defects has a very significant appearance in terms of colour. Although the pictures are taken under various angle and brightness condition, it does not have a great impact on the accuracy of colour detection compared to the Canny edge detection.

Meanwhile, the edge defects detection and the loose wrap detection have lower accuracy compared to the rusting defects detection. This is because the Canny edge detection is more sensitive to the distance and angle of the taken input images. All these variations can affect the edge structure in the images and the output of Canny edge detection may not be consistent for each image. Thus, it is important to improve the way in collecting and inspecting the input images.

CHAPTER 5

CONCLUSIONS AND RECOMMENDATIONS

5.1 Conclusions

In conclusion, this project has studied and presented the feasible hot rolled steel defects detection system with image recognition technology. In this project, the rusting defect detection has the highest accuracy, which is more than 90% accurate. Meanwhile, the overall accuracy for edge defects detection and loose wrap detection is less than 70%. The issues that caused the errors were identified in the performance test.

The problem encountered for rusting defects detection is that the collected images are taken at different brightness conditions and angles. It can affect the performance of the colour detection algorithm as the quality of the photos varies. For example, the image taken under high-intensity sunlight may cause false detection as the colours of the picture may not be precise.

Furthermore, the edge defects detection and loose wrap detection system is also met a similar issue which is the variation of image quality in terms of the distance and angle. These factors will affect the output of the extracted edge lines from Canny edge detection. Eventually, the result of the defect severity evaluation might not be accurate.

Besides, the amount of the collected image dataset is not sufficient to justify the performance of the detection system. Due to the pandemic, collecting the data is a challenge for this project. Also, the deep learning model does not have enough data for the training purpose due to this problem.

5.2 Recommendations for future work

The recommended solution to tackle the image quality issue for hot rolled steel defects detection is to standardize the input image method. It can be done by setting an indoor defects inspection environment where the camera is set at the fixed position. Then, the hot rolled steel can be sent into the inspection room for the detection process. It ensures that the light intensity, distance, angle and background are always constant when taking the images.

One of the fastest ways to solve the insufficient data problem is data augmentation, which is a process to create a different version of the existing data. For example, a new image can be made by rotating a current picture by 90 degrees. Also, another new image can be generated by adding some noise to an existing image. Eventually, the number of data can be increased to a sufficient amount for training the deep learning model.

Although data augmentation can quickly increase the dataset's size, it might cause the trained model to overfit. The model becomes overfit when the model is used to the features of the training dataset and unable to work well with the new data. Hence, it would be better to collect more new data for training the model so that it can generalize well with all types of conditions.

REFERENCES

Aggarwal, A. (2021). *YOLO Explained - Analytics Vidhya*. https://medium.com/analytics-vidhya/yolo-explained-5b6f4564f31

Bansal, I. (2021). *Color Detection using Python - Beginner's Reference*. https://www.askpython.com/python/examples/color-detection

Bansari, S. (2021). *Introduction to how CNNs Work - DataDrivenInvestor*. https://medium.datadriveninvestor.com/introduction-to-how-cnns-work-77e0e4cde99b

Brownlee, J. (2020a). *Softmax Activation Function with Python*. https://machinelearningmastery.com/softmax-activation-function-with-python/

Brownlee, J. (2020b). *What is Deep Learning?* https://machinelearningmastery.com/what-is-deep-learning/

Dash, A. K. (2019). *Single Shot Detection (SSD) Algorithm*. https://iq.opengenus.org/single-shot-detection-ssd-algorithm/

Dickson, B. (2019). *What is computer vision?* https://bdtechtalks.com/2019/01/14/what-is-computer-vision/

Dudeperf3ct. (2019). *Mystery of Object Detection*. https://dudeperf3ct.github.io/object/detection/2019/01/07/Mystery-of-Object-Detection/#loss-functions

Dynamsoft. (2019). *Image Processing 101 Chapter 1.3: Color Space Conversion*. https://www.dynamsoft.com/blog/insights/image-processing/image-processing-101-color-space-conversion/

Edwards, E. (n.d.). *An Introduction to Machine Vision and the Machine Vision System*. https://www.thomasnet.com/articles/automation-electronics/machine-vision-systems/

Grover, P. (2021). *5 Regression Loss Functions All Machine Learners Should Know*. https://heartbeat.comet.ml/5-regression-loss-functions-all-machine-learners-should-know-4fb140e9d4b0

Gurucharan, M. (2021). *Basic CNN Architecture: Explaining 5 Layers of Convolutional Neural Network.* https://www.upgrad.com/blog/basic-cnn-architecture/#:%7E:text=There%20are%20three%20types%20of,CNN%20architecture%20will%20be%20formed.

Halbe, S. (2021). *Object Detection and Instance Segmentation: A detailed overview*. https://medium.com/swlh/object-detection-and-instance-segmentation-a-detailed-overview-94ca109274f2

Hui, J. (2020). *SSD object detection: Single Shot MultiBox Detector for real-time processing*. https://jonathan-hui.medium.com/ssd-object-detection-single-shot-multibox-detector-for-real-time-processing-9bd8deac0e06

Karimi, G. (2021). *Introduction to YOLO Algorithm for Object Detection*. https://www.section.io/engineering-education/introduction-to-yolo-algorithm-for-object-detection/

Khazri, A. (2021). *Faster RCNN Object detection - Towards Data Science*. https://towardsdatascience.com/faster-rcnn-object-detection-f865e5ed7fc4

Maj, 152. (2018). *Object Detection and Image Classification with YOLO*. https://www.kdnuggets.com/2018/09/object-detection-image-classification-yolo.html

Metallic Steel. (2020). *Analysis of common surface defects of hot rolled steel sheet*. https://www.metallicsteel.com/analysis-of-common-surface-defects-of-hot-rolled-steel-sheet.html

Rakshit, S. (2021). *Intersection Over Union - Koderunners*. https://medium.com/koderunners/intersection-over-union-516a3950269c

Reliance Foundry Co. Ltd. (2021). *Hot Rolled vs Cold Rolled Steel*. https://www.reliance-foundry.com/blog/hot-vs-cold-rolled-steel

Sagar, R. (2020). *What Is The Difference Between Computer Vision And Image Processing*. https://analyticsindiamag.com/what-is-the-difference-between-computer-vision-and-image-processing/

Sharma, D. J., Dutta, S., & Bora, D. J. (2020, January 20). *REGA: Real-Time Emotion, Gender, Age Detection Using CNN—A Review*. https://doi.org/10.15439/2020KM18

Sharma, S. (2019). *What the Hell is Perceptron? - Towards Data Science*. https://towardsdatascience.com/what-the-hell-is-perceptron-626217814f53

Suwanmanee, S., Chatpun, S., & Cabrales, P. (2013, October). Comparison of video image edge detection operators on red blood cells in microvasculature. *The 6th 2013 Biomedical Engineering International Conference*. https://doi.org/10.1109/BMEiCon.2013.6687686

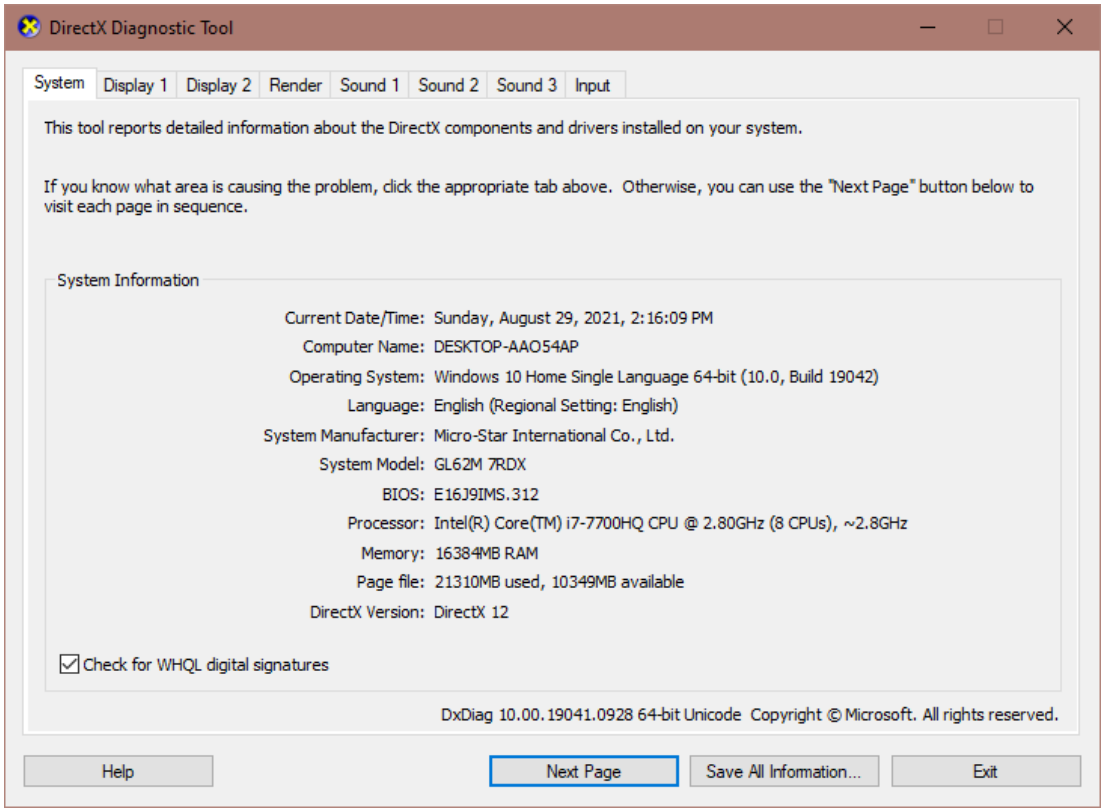
Yeung, S. (n.d.). *Tutorial 1: Image Filtering. Stanford Artificial Intelligence*. https://ai.stanford.edu/~syyeung/cvweb/tutorial1.html

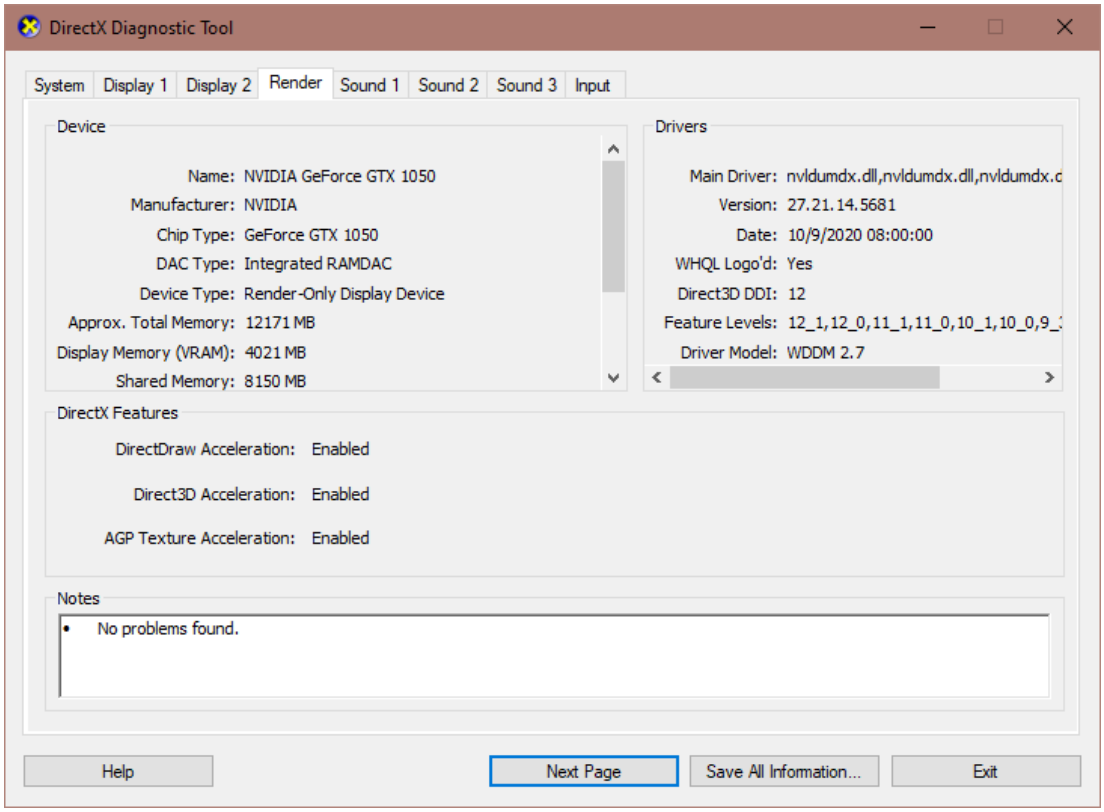
Yolcu, Ş. A. E. (2020). *Introduction to Image Processing*. https://www.udentify.co/Blog/12/2019/introduction-to-image-processing/

Zhang, J. (2010). Edge Detection in Glass Fragmentation Images Based on One Order Differential Operator. *2010 Second International Conference on Computer Engineering and Applications*. https://doi.org/10.1109/ICCEA.2010.278

APPENDICES

APPENDIX A: Computer Specification





APPENDIX B: Python Code for Rusting Detection

```
import tensorflow as tf
from object_detection.utils import label_map_util

import os
import cv2
import time
import numpy as np
from PIL import Image
import datetime
import pandas as pd
files = os.listdir("rust_data")

PATH_TO_LABELS=r"E:\utar\Y4S2\FYP\project_folder\dataset\dataset2\hot_label
map.pbtxt"
threshold=60

category_index =
label_map_util.create_category_index_from_labelmap(PATH_TO_LABELS,
                                                   use_display_name=True)

data_frame = pd.DataFrame(columns=['image', 'release', 'hold'])
gpus = tf.config.experimental.list_physical_devices('GPU')
for gpu in gpus:
    tf.config.experimental.set_memory_growth(gpu, True)



PATH_TO_SAVED_MODEL =
r"E:\utar\Y4S2\FYP\project_folder\ssd_hot_model\exported_model\saved_model"



print('Loading model...', end='')
start_time = time.time()
```

```
# Load saved model and build the detection function
detect_fn = tf.saved_model.load(PATH_TO_SAVED_MODEL)

end_time = time.time()
elapsed_time = end_time - start_time
print('Done! Took {} seconds'.format(elapsed_time))

def load_image_into_numpy_array(path):
    """Load an image from file into a numpy array.

    Puts image into numpy array to feed into tensorflow graph.
    Note that by convention we put it into a numpy array with shape
    (height, width, channels), where channels=3 for RGB.

    Args:
      path: the file path to the image

    Returns:
      uint8 numpy array with shape (img_height, img_width, 3)
    """
    return np.array(Image.open(path))

def rust_detection(img):
    lower_rust=(0,31,23)
    higher_rust=(33,255,255)

    hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)

    mask = cv2.inRange(hsv, lower_rust , higher_rust)
    #cv2.imwrite("mask.jpg",mask)
    percentage = (mask==255).mean() * 100

    mask_overlay=img.copy()
    mask_overlay[np.where((mask==[255]))] = (0,255,0)
```

```
    img = ((0.5 * img) + (0.5 * mask_overlay)).astype("uint8")


    return percentage,img


#('Running inference for {}... '.format(image_path), end='')
for i in range(len( files)):
    current_time=datetime.datetime.now()

    image_np=cv2.imread("rust_data/"+files[i])
    img=cv2.imread("rust_data/"+files[i])

    #image_np = load_image_into_numpy_array("data/"+files[i])
    basename = os.path.basename("rust_data/"+files[i])

    ori_image = cv2.imread("rust_data/"+files[i])



    # The input needs to be a tensor, convert it using `tf.convert_to_tensor`.
    input_tensor = tf.convert_to_tensor(image_np)
    # The model expects a batch of images, so add an axis with `tf.newaxis`.
    input_tensor = input_tensor[tf.newaxis, ...]

    # input_tensor = np.expand_dims(image_np, 0)

    detections = detect_fn(input_tensor)

    # All outputs are batches tensors.
    # Convert to numpy arrays, and take index [0] to remove the batch dimension.
    # We're only interested in the first num_detections.
    num_detections = int(detections.pop('num_detections'))
    detections = {key: value[0, :num_detections].numpy()
```

```
                for key, value in detections.items()}
    detections['num_detections'] = num_detections

    # detection_classes should be ints.
    detections['detection_classes'] = detections['detection_classes'].astype(np.int64)

    image_np_with_detections = image_np.copy()

    filtered_detection=[i for i in range(len(detections['detection_scores'])) if detections['detection_scores'][i] >=0.3]


    for item in filtered_detection:


minX,minY=int(detections['detection_boxes'][item][1]*image_np.shape[1]),int(detections['detection_boxes'][item][0]*image_np.shape[0])

maxX,maxY=int(detections['detection_boxes'][item][3]*image_np.shape[1]),int(detections['detection_boxes'][item][2]*image_np.shape[0])

        defect_type=category_index[detections['detection_classes'][item]]["name"]


percentage,ori_image[minY:maxY,minX:maxX]=rust_detection(ori_image[minY:maxY,minX:maxX])
        processed_time=(datetime.datetime.now()-current_time).microseconds / 1000
        if percentage >threshold:

            true = 1
            if "release" in files[i]:
                true = 0
```

```
        new_data = pd.DataFrame([{'image': files[i], 'release': "","hold":"Yes","True":true,"Process Duration":processed_time,"Image Resolution":(img.shape[1],img.shape[0]),"Number of Pixels":img.shape[1]*img.shape[0],"Rust Percentage":percentage}],
                                columns =['image', 'release', 'hold',"True","Process Duration","Image Resolution","Number of Pixels","Rust Percentage"])

        data_frame=data_frame.append(new_data, ignore_index = True)

        cv2.imwrite("rust_hold/"+str(files[i]).split(".")[0]+".jpg",ori_image)
    else:
        true = 1
        if "hold" in files[i]:
            true = 0
        cv2.imwrite("rust_release/"+str(files[i]).split(".")[0]+".jpg",ori_image)
        new_data = pd.DataFrame([{'image': files[i], 'release': "Yes","hold":"","True":true,"Process Duration":processed_time,"Image Resolution":(img.shape[1],img.shape[0]),"Number of Pixels":img.shape[1]*img.shape[0],"Rust Percentage":percentage}],
                                columns =['image', 'release', 'hold',"True","Process Duration","Image Resolution","Number of Pixels","Rust Percentage"])

        data_frame=data_frame.append(new_data, ignore_index = True)

    break

percentage_accuracy = len(data_frame[(data_frame['True']==1)])/len(data_frame) * 100

print("Accuracy:"+str(percentage_accuracy))
data_frame.to_excel("output_rust.xlsx")
```

APPENDIX C: Python Code for Edge Defects Detection

```
import tensorflow as tf
from object_detection.utils import label_map_util
from object_detection.utils import visualization_utils as viz_utils
import glob
import os
import cv2
import pandas as pd
import datetime
data_frame = pd.DataFrame(columns=['image', 'release', 'hold'])

def getListOfFiles(dirName):
    # create a list of file and sub directories
    # names in the given directory
    listOfFile = os.listdir(dirName)
    allFiles = list()
    # Iterate over all the entries
    for entry in listOfFile:
        # Create full path
        fullPath = os.path.join(dirName, entry)
        # If entry is a directory then get the list of files in this directory
        if os.path.isdir(fullPath):
            allFiles = allFiles + getListOfFiles(fullPath)
        else:
            allFiles.append(fullPath)

    return allFiles

PATH_TO_LABELS=r"E:\utar\Y4S2\FYP\project_folder\dataset\dataset\defects_la
belmap.pbtxt"

category_index                                                    =
label_map_util.create_category_index_from_labelmap(PATH_TO_LABELS,
                                                    use_display_name=True)
```

```
gpus = tf.config.experimental.list_physical_devices('GPU')
for gpu in gpus:
    tf.config.experimental.set_memory_growth(gpu, True)

import time
from object_detection.utils import label_map_util
from object_detection.utils import visualization_utils as viz_utils

PATH_TO_SAVED_MODEL =
r"E:\utar\Y4S2\FYP\project_folder\ssd_defects_model\exported_model\saved_mode
l"

print('Loading model...', end='')
start_time = time.time()

# Load saved model and build the detection function
detect_fn = tf.saved_model.load(PATH_TO_SAVED_MODEL)

end_time = time.time()
elapsed_time = end_time - start_time
print('Done! Took {} seconds'.format(elapsed_time))



import numpy as np
from PIL import Image
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings('ignore')   # Suppress Matplotlib warnings

def load_image_into_numpy_array(path):
    """Load an image from file into a numpy array.

    Puts image into numpy array to feed into tensorflow graph.
```

```
    Note that by convention we put it into a numpy array with shape
    (height, width, channels), where channels=3 for RGB.

    Args:
      path: the file path to the image

    Returns:
      uint8 numpy array with shape (img_height, img_width, 3)
    """
    return np.array(Image.open(path))

IMAGE_PATHS=getListOfFiles(r"edge_data")

def edge_folded_check(img):

    # Convert to graycsale
    img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    # Blur the image for better edge detection
    img_blur = cv2.GaussianBlur(img_gray, (9,9), 0)



    # Canny Edge Detection
    edges = cv2.Canny(image=img_blur, threshold1=30, threshold2=100) # Canny
Edge Detection



    total_white=np.sum(edges == 255)
    total_pixels=edges.shape[0]*edges.shape[1]

    white_pixel_percentage=total_white/total_pixels*100

    if white_pixel_percentage>=6:
        return True
    else:
```

```
        return False

def edge_dented_check(img):

    # Convert to graycsale
    img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    # Blur the image for better edge detection
    img_blur = cv2.GaussianBlur(img_gray, (9,9), 0)



    # Canny Edge Detection
    edges = cv2.Canny(image=img_blur, threshold1=30, threshold2=100) # Canny Edge Detection




    total_white=np.sum(edges == 255)
    total_pixels=edges.shape[0]*edges.shape[1]

    white_pixel_percentage=total_white/total_pixels*100

    if white_pixel_percentage>=4:
        return True
    else:

        return False


for image_path in IMAGE_PATHS:
    current_time=datetime.datetime.now()
    print('Running inference for {}... '.format(image_path), end='')

    image_np = load_image_into_numpy_array(image_path)
```

```
basename = os.path.basename(image_path)

ori_image = cv2.imread(image_path)

# Things to try:
# Flip horizontally
# image_np = np.fliplr(image_np).copy()

# Convert image to grayscale
# image_np = np.tile(
#     np.mean(image_np, 2, keepdims=True), (1, 1, 3)).astype(np.uint8)

# The input needs to be a tensor, convert it using `tf.convert_to_tensor`.
input_tensor = tf.convert_to_tensor(image_np)
# The model expects a batch of images, so add an axis with `tf.newaxis`.
input_tensor = input_tensor[tf.newaxis, ...]

# input_tensor = np.expand_dims(image_np, 0)
try:
    detections = detect_fn(input_tensor)

    # All outputs are batches tensors.
    # Convert to numpy arrays, and take index [0] to remove the batch dimension.
    # We're only interested in the first num_detections.
    num_detections = int(detections.pop('num_detections'))
    detections = {key: value[0, :num_detections].numpy()
                  for key, value in detections.items()}
    detections['num_detections'] = num_detections

    # detection_classes should be ints.
    detections['detection_classes'] = detections['detection_classes'].astype(np.int64)

    image_np_with_detections = image_np.copy()
```

```
filtered_detection=[i for i in range(len(detections['detection_scores'])) if detections['detection_scores'][i] >=0.3]

for item in filtered_detection:

minX,minY=int(detections['detection_boxes'][item][1]*image_np.shape[1]),int(detections['detection_boxes'][item][0]*image_np.shape[0])

maxX,maxY=int(detections['detection_boxes'][item][3]*image_np.shape[1]),int(detections['detection_boxes'][item][2]*image_np.shape[0])

        defect_type=category_index[detections['detection_classes'][item]]["name"]

        release=0
        hold=0
        true=0


#cv2.imwrite("cropped/"+basename.split(".")[0]+str(item)+".jpg",ori_image[minY:maxY,minX:maxX])

        if defect_type=="edge_folded":
            hold_item=edge_folded_check(ori_image[minY:maxY,minX:maxX])
            if hold_item:
                defect_type=defect_type+"_hold"
                hold=1
            else:
                defect_type=defect_type+"_release"
                release=1

        elif defect_type=="edge_dented":
            hold_item=edge_dented_check(ori_image[minY:maxY,minX:maxX])
            if hold_item:
                defect_type=defect_type+"_hold"
```

```
                hold=1
            else:
                defect_type=defect_type+"_release"
                release=1

        elif defect_type=="edge_crack":
            hold=1

        if "release" in image_path and release==1:
            true=1

        if "hold" in image_path and hold==1:
            true=1


        processed_time=(datetime.datetime.now()-current_time).microseconds / 1000
        new_data = pd.DataFrame([{'image': image_path.split("\\")[-1], 'release': release,"hold":hold,"True":true,"Process Time":processed_time}],
                                columns =['image', 'release', 'hold',"True","Process Time"])

        data_frame=data_frame.append(new_data, ignore_index = True)


        cv2.rectangle(ori_image, (minX,minY), (maxX,maxY), (0,255,0), 2)
        cv2.putText(ori_image, str(defect_type), (minX,minY), cv2.FONT_HERSHEY_SIMPLEX,1, (0, 255, 255), 4)



    cv2.imwrite("edge_output/"+basename,ori_image)


    viz_utils.visualize_boxes_and_labels_on_image_array(
        image_np_with_detections,
```

```
            detections['detection_boxes'],
            detections['detection_classes'],
            detections['detection_scores'],
            category_index,
            use_normalized_coordinates=True,
            max_boxes_to_draw=200,
            min_score_thresh=.30,
            agnostic_mode=False)

        plt.figure()
        plt.imshow(image_np_with_detections)
        print('Done')
        #cv2.imwrite("output/"+basename,image_np_with_detections)

    except:
        None

plt.show()
data_frame.to_excel("edge_output.xlsx")
percentage_accuracy = len(data_frame[(data_frame['True']==1)])/len(data_frame) *
100

print("Accuracy:"+str(percentage_accuracy))
# sphinx_gallery_thumbnail_number = 2
```

APPENDIX D: Python Code for Loose Wrap Detection

```
import tensorflow as tf
from object_detection.utils import label_map_util
from object_detection.utils import visualization_utils as viz_utils
import glob
import os
import cv2
import pandas as pd
import datetime
data_frame = pd.DataFrame(columns=['image', 'release', 'hold'])
def getListOfFiles(dirName):
    # create a list of file and sub directories
    # names in the given directory
    listOfFile = os.listdir(dirName)
    allFiles = list()
    # Iterate over all the entries
    for entry in listOfFile:
        # Create full path
        fullPath = os.path.join(dirName, entry)
        # If entry is a directory then get the list of files in this directory
        if os.path.isdir(fullPath):
            allFiles = allFiles + getListOfFiles(fullPath)
        else:
            allFiles.append(fullPath)

    return allFiles

PATH_TO_LABELS=r"E:\utar\Y4S2\FYP\project_folder\dataset\dataset2\hot_label
map.pbtxt"

category_index                                                    =
label_map_util.create_category_index_from_labelmap(PATH_TO_LABELS,
                                                    use_display_name=True)
```

```
gpus = tf.config.experimental.list_physical_devices('GPU')
for gpu in gpus:
    tf.config.experimental.set_memory_growth(gpu, True)

import time
from object_detection.utils import label_map_util
from object_detection.utils import visualization_utils as viz_utils

PATH_TO_SAVED_MODEL                                                    =
r"E:\utar\Y4S2\FYP\project_folder\ssd_hot_model\exported_model\saved_model"

print('Loading model...', end='')
start_time = time.time()

# Load saved model and build the detection function
detect_fn = tf.saved_model.load(PATH_TO_SAVED_MODEL)

end_time = time.time()
elapsed_time = end_time - start_time
print('Done! Took {} seconds'.format(elapsed_time))



import numpy as np
from PIL import Image
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings('ignore')   # Suppress Matplotlib warnings

def load_image_into_numpy_array(path):
    """Load an image from file into a numpy array.

    Puts image into numpy array to feed into tensorflow graph.
    Note that by convention we put it into a numpy array with shape
```

```
    (height, width, channels), where channels=3 for RGB.

    Args:
      path: the file path to the image

    Returns:
      uint8 numpy array with shape (img_height, img_width, 3)
    """
    return np.array(Image.open(path))

IMAGE_PATHS=getListOfFiles(r"loose_data")



def loose_wrap_check(img):

    # Convert to graycsale
    img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    # Blur the image for better edge detection
    img_blur = cv2.GaussianBlur(img_gray, (9,9), 0)



    # Canny Edge Detection
    edges = cv2.Canny(image=img_blur, threshold1=100, threshold2=100) # Canny
Edge Detection



    total_white=np.sum(edges == 255)
    total_pixels=edges.shape[0]*edges.shape[1]

    white_pixel_percentage=total_white/total_pixels*100

    if white_pixel_percentage>=3:
        return True
```

```
    else:

        return False


for image_path in IMAGE_PATHS:
    current_time=datetime.datetime.now()
    print('Running inference for {}... '.format(image_path), end='')

    image_np = load_image_into_numpy_array(image_path)
    basename = os.path.basename(image_path)

    ori_image = cv2.imread(image_path)

    # Things to try:
    # Flip horizontally
    # image_np = np.fliplr(image_np).copy()

    # Convert image to grayscale
    # image_np = np.tile(
    #     np.mean(image_np, 2, keepdims=True), (1, 1, 3)).astype(np.uint8)

    # The input needs to be a tensor, convert it using `tf.convert_to_tensor`.
    input_tensor = tf.convert_to_tensor(image_np)
    # The model expects a batch of images, so add an axis with `tf.newaxis`.
    input_tensor = input_tensor[tf.newaxis, ...]

    # input_tensor = np.expand_dims(image_np, 0)
    try:
        detections = detect_fn(input_tensor)

        # All outputs are batches tensors.
        # Convert to numpy arrays, and take index [0] to remove the batch dimension.
        # We're only interested in the first num_detections.
```

```
num_detections = int(detections.pop('num_detections'))
detections = {key: value[0, :num_detections].numpy()
              for key, value in detections.items()}
detections['num_detections'] = num_detections

# detection_classes should be ints.
detections['detection_classes'] = detections['detection_classes'].astype(np.int64)

image_np_with_detections = image_np.copy()

filtered_detection=[i for i in range(len(detections['detection_scores'])) if detections['detection_scores'][i] >=0.3]

for item in filtered_detection:

minX,minY=int(detections['detection_boxes'][item][1]*image_np.shape[1]),int(detections['detection_boxes'][item][0]*image_np.shape[0])

maxX,maxY=int(detections['detection_boxes'][item][3]*image_np.shape[1]),int(detections['detection_boxes'][item][2]*image_np.shape[0])

    defect_type=category_index[detections['detection_classes'][item]]["name"]

    release=0
    hold=0
    true=0

    release=0
    hold=0
    true=0


    hold_item=loose_wrap_check(ori_image[minY:maxY,minX:maxX])
    if hold_item:
```

```
            defect_type=defect_type+"_hold"
            hold=1
        else:
            defect_type=defect_type+"_release"
            release=1

        if "release" in image_path and release==1:
            true=1

        if "hold" in image_path and hold==1:
            true=1
        processed_time=(datetime.datetime.now()-current_time).microseconds / 1000
        new_data = pd.DataFrame([{'image': image_path.split("\\")[-1], 'release': release,"hold":hold,"True":true,"Process Time":processed_time}],
                                columns =['image', 'release', 'hold',"True","Process Time"])

        data_frame=data_frame.append(new_data, ignore_index = True)

        cv2.rectangle(ori_image, (minX,minY), (maxX,maxY), (0,255,0), 2)
        cv2.putText(ori_image, str(defect_type), (minX,minY), cv2.FONT_HERSHEY_SIMPLEX,1, (0, 255, 255), 4)

    cv2.imwrite("loose_output/"+basename,ori_image)


    viz_utils.visualize_boxes_and_labels_on_image_array(
        image_np_with_detections,
        detections['detection_boxes'],
        detections['detection_classes'],
        detections['detection_scores'],
        category_index,
        use_normalized_coordinates=True,
        max_boxes_to_draw=200,
        min_score_thresh=.30,
```

```
            agnostic_mode=False)

        plt.figure()
        plt.imshow(image_np_with_detections)
        print('Done')
        #cv2.imwrite("output3/"+basename,image_np_with_detections)

    except:
        None

plt.show()
data_frame.to_excel("loose_output.xlsx")
percentage_accuracy = len(data_frame[(data_frame['True']==1)])/len(data_frame) *
100

print("Accuracy:"+str(percentage_accuracy))
# sphinx_gallery_thumbnail_number = 2
```