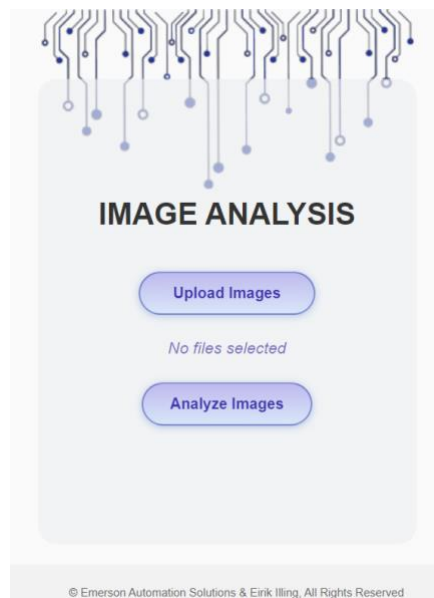


FMH606 Master's Thesis 2023
Industrial IT and Automation

Object detection, information extraction and analysis of operator interface images using computer vision and machine learning.



Eirik Illing

Faculty of Technology, Natural sciences and Maritime Sciences
Campus Porsgrunn

Course: FMH606 Master's Thesis, 2023

Title: Object detection, information extraction and analysis of operator interface images using computer vision and machine learning.

Number of pages: 140

Keywords: deep learning, object detection, OCR, image analysis, industry, operator

Student: Eirik Illing

Supervisors: Ole Magnus Brastein and Nils-Olav Skeie

External partner: Emerson Automation Solutions

Summary:

Operator interface display images, often referred to as HMI, contains large amounts of information that can be valuable to obtain. If access to the source code or design files are limited, modern frameworks for object detection and text extraction can be used to obtain this information directly from images. However, obtaining data and training such modern solutions is time consuming, and require a lot of manual work to get started. In this project, traditional computer vision methods have been used to extract objects from images, separated the objects into training data and transferred learned a ResNet model to do multi-label image classification of individual objects. This model, in combination with methods such as sliding window, pyramid scaling and NMS gave the foundation for creating a semi-automated annotation tool that generates training data for more optimized object detection methods, in this case YOLO object detector. The semi-automated annotation tool provides a starting point for engineers to do manual touchup on the training data, and finally export state of the art training images for YOLO. The YOLO model is transfer learned on the annotated data, achieving a satisfying mAP⁵⁰ score of 95.5%. A third-party library for OCR is used to obtain text information from preprocessed images, postprocessing the text by filtering tag data only, and an algorithm is used to link objects and tags together. The final solution is hosted in a software developed to focus on optimized user interaction, resulting in a excel formatted analysis document available for export to the end user.

The University of South-Eastern Norway takes no responsibility for the results and conclusions in this student report.

Preface

This thesis is written as a final examination and concludes the last semester of the Industrial IT and Automation (IIA) Industry Master's (IM) program, at University of South-Eastern Norway (USN), spring 2023. During my three years at the IM program, I have had the pleasure of being employed at Emerson Automation Solutions as a systems and software engineer. I would therefore like to start by giving a big thanks to my employer and external partner Emerson Automation Solutions and supervisor Geir Falkevik for supporting this project. I also want to give a big thanks to my supervisors at USN Ole Magnus Brastein and Nils-Olav Skeie for valuable discussions and support regarding project structuring and delivery. Lastly, I want to thank designer Kjersti B. Kjeldby for providing support on the UI design for the final application. This project provided an opportunity to deep dive into the field of machine learning for image analysis and made a foundation for object detection in industrial applications. There are many new ideas derived from this project that have potential to provide great value to the field, and I would like to encourage the reader to build on these ideas if desirable.

Porsgrunn, 11/05/2023

Eirik Illing

Contents

1	Introduction	8
1.1	Project background	8
1.2	Objectives	9
1.3	Methods for Development	9
1.4	Assumptions	10
1.5	Outline of report.....	11
2	Literature review	12
3	System description	14
3.1	Operator interface image systems	14
3.1.1	<i>Process control, interface, and component's structure</i>	14
3.1.2	<i>Existing design and analysis tools</i>	15
3.1.3	<i>Advantages and challenges with existing analysis tools</i>	15
3.2	Project scope	16
3.2.1	<i>Technology</i>	17
3.2.2	<i>The general idea</i>	17
3.2.3	<i>The general goal</i>	18
3.3	Semi-automated annotation tool (Program 1).....	18
3.4	Industrial Component Extraction tool – ICE (Program 2)	19
3.5	Development environment	20
3.5.1	<i>Hardware environment</i>	21
3.5.2	<i>Software environment</i>	22
3.5.3	<i>Web solutions</i>	22
3.5.4	<i>Software used</i>	22
3.5.5	<i>Python frameworks and libraries</i>	22
3.5.6	<i>Datasets</i>	24
3.6	Data collection and preparations.....	24
3.6.1	<i>Obtaining training data</i>	24
3.6.2	<i>Data preparations</i>	30
4	Methods.....	33
4.1	ANN - Artificial Neural Networks and Convolutions	33
4.1.1	<i>Using ANN for image classification</i>	33
4.1.2	<i>CNN – Convolutional Neural Networks</i>	34
4.2	Image classification.....	39
4.2.1	<i>ResNet – Residual Network</i>	39
4.2.2	<i>What is single-label classification?</i>	44
4.2.3	<i>What is multi-label classification?</i>	44
4.2.4	<i>Data preparations</i>	44
4.2.5	<i>Training and validating</i>	49
4.3	Object detection.....	51
4.3.1	<i>DLNN Detection</i>	52
4.3.2	<i>YOLOv8</i>	56
4.3.3	<i>Data preparations for YOLOv8 object detection</i>	62
4.3.4	<i>Training and validating YOLOv8 object detection</i>	63
4.4	Non-Maximum Suppression	64
4.4.1	<i>IoU – Intersection over Union</i>	64
4.5	Software analysis and design	65
4.5.1	<i>Semi-automated annotation software (Program 1)</i>	65
4.5.2	<i>Industrial Component Extraction tool – ICE (Program 2)</i>	68

4.6 Quick recap73

5 Result.....75

5.1 Image classification.....75

 5.1.1 *Single-label classification*75

 5.1.2 *Multi-label classification*83

5.2 Object detection.....97

 5.2.1 *Sliding window, image pyramid scaling and NMS*97

 5.2.2 *Semi-Automated annotation tool (Program 1)*..... 105

 5.2.3 *YOLO one-stage detector*..... 114

 5.2.4 *Tag extraction and object linking*..... 124

 5.2.5 *Industrial Component Extraction – ICE (Program 2)* 126

6 Discussion 130

6.1 Comparing sliding window classification and YOLO algorithm for object detection. 130

6.2 Improvements and future work 130

6.3 Thinking outside the box 131

7 Conclusion 133

Nomenclature

ANN – Artificial Neural Networks
API – Application Programming Interface
BCE – Binary Cross-entropy
CE - Cross-entropy
CMD – Command Prompt
CPU – Central Processing Unit
CSS – Cascading Style Sheet
CSV – Comma Separated Values
CTPN – Connectionist Text Proposal Networks
DCNN – Deep Convolutional Neural Networks
DCS – Distributed Control System
DLNN – Deep Learning Neural Network
DPM – Deformable Part-based Model
FANN – Feedforward Artificial Neural Networks
FCN – Fully Convolutional Networks
GPU – Graphics Processing Unit
GUI – Graphical User Interface
HMI – Human Machine Interface
HOG – Histogram of Oriented Gradients
HTML – Hypertext Markup Language
HTTP – Hypertext Transfer Protocol
IDE – Integrated Development Environment
IIA – Industrial IT and Automation
IM – Industry Master
IoU – Intersection over Unions
LAN – Local Area Network
MA – Mosaic Augmentation
mAP – Mean Average Precision
ML – Machine Learning
NMS – Non-Maximum Suppresjon
OCR – Optical Character Recognition

OpenCV – Open-Source Computer Vision
OOADP – Object-Oriented Analysis, Design and Programming
PC – Personal Computer
PCS – Process Control System
PS – Power Supply
RAM – Random Access Memory
R-CNN – Region-based Convolutional Neural Network
RoI – Region of Interest
RPi – Raspberry Pi
SCD – System Control Diagram
SSD – Single Shot Detector
SSD – Solid State Drive (Hard Drive)
SSD – System Sequence Diagram
SSR – Sum of Squared Residuals
SVG – Scalable Vector Graphics
SVM – Support Vector Machine
UCD – Use Case Diagram
UI – User Interface
USN – University of South-Eastern Norway
VBA – Visual Basic for Application
WSL – Windows Subsystem for Linux
YOLO – You Only Look Once

1 Introduction

This chapter will give an introduction to the project, explaining the background, objectives, methods for development, assumptions and outline of the report.

1.1 Project background

Image classification, object detection and Computer Vision (CV) technology are all hot topics in a modernizing society. Complex image recognition models that can detect large quantity of different objects in pictures and real-time video are developed and commercially available for most day-to-day applications. A lot of industrial applications also uses this technology for i.e., analyzing fertilizers [1], material fibers, object structure [2] and even P&ID document analysis (discussed in more detail in chapter 2). However, there are still applications that remain untouched or unthought of, and a lot of fields where these technologies would have huge benefits.

Analyzing operator interface images using CV and Machine Learning (ML) is one of these fields that have been shown little interest by the research community. Creating an application for object detection in operator interface images is quite interesting because of the degree of complexity these images contain, and the number of applications this analysis may be beneficial for. Complexity variations such as color differences, small symbols, limited symbol features, information text, tags, status, values, and other information makes object detection challenging and make them well suited for research within the field of machine learning analysis. If the gap in research and application development within this field is due to lack of interest, no uses cases for it or because no one have thought about it yet, is not known. However, there is now an interest and use case for operator interface images object detection and information extraction software.

The background for this project is the time consumption and complexity of migration projects related to Process Control System (PCS). Migrating from one PCS to another will normally require hardware change, logic configuration and operator graphics design. To get a clear estimate on these type of project costs, engineers analyze the hardware build, existing and additional logic, and complexity of operator graphics. Tools for analyzing hardware, logic and graphics exist and is easy to perform if access to the old system documentation and source code is available. Doing analysis within these areas is unquestionably harder if the only available documentation is paper formatted, image snippets, or pdf's of the existing configurations and graphics. If such a case should occur, an engineer with skills matching the current field would sit down and manually do this analysis and estimate cost. This is where machine learning and object detection comes into play for improving the efficiency and accuracy of these estimates. The rest of this project will revolve around operator graphics analysis, not logic configuration. This is because operator graphics contain a larger set of undefined variables, while logic configuration still is standardized in some way.

Emerson's Distributed Control System (DCS) DeltaV, introduces a new modernized fully integrated operator graphics interface framework called DeltaV Live in version 14. Even though the old fully integrated version DeltaV Operate is still supported, DeltaV Live [3] is the new state-of-the-art modern framework to use and has imposed a high demand marked of migrating old systems to the new system. When doing a migration project from an old system

or external vendor system, thorough analysis of existing graphics is a crucial first step. This is important for identify objects and information within each graphic image, calculating the complexity of the graphics and further calculate cost of such a project. The purpose of this master thesis is therefore to investigate development of tools, utilizing machine learning and CV technology to detect objects and do complexity analysis from images. This project thesis could also serve as a foundation for digitally migrating systems directly without the need for any source code of old systems and provide information that can be used to analyze and recommend improvements to existing design.

The new and exciting challenge in this project compared to previous work is the foundation of creating image object detection models for industrial purposes. Not only looking at standardized objects such as P&ID symbols, but objects with color variations and various degrees of complexity. How to handle noise such as process lines and different text data, read tags and correlate them to objects. Also looking into how good a transfer learned pretrained model can perform on totally new data, and how to use these models in object detection on large noisy operator interface images.

1.2 Objectives

There are 7 main objectives of this project listed as follows:

1. How to obtain valuable data for training, validation, and testing.
2. Trying a simple image classifier to test accuracy of pretrained networks transfer learned with training data for operator interface graphic components.
3. Creating a multi-label classification model for detecting multiple objects within an image.
4. Using the multi-label classification model in combination with traditional CV methods such as sliding window, pyramid image scaling and Non-maximum Suppression (NMS). Resulting in a multi-class object detector.
5. Developing a tool utilizing the multi-class object detector to annotate training data for a more sophisticated object detector.
6. Creating a one-stage detection method for object detection and compare it to objective 4.
7. Creating a hosting application for the one-stage detector, tag extraction with optical character recognition (OCR), linking objects and tags, and exporting the analysis in a user readable format.

These objectives are more detailed and specific than the “interim goals” from the task description in Appendix A which was the basis for the thesis and is fulfilled in depth during this project. The most important step for all these objectives is the collection of valuable data for training, validation, and testing. This will be the basis for training models and to annotate images for the one-stage detector approach.

1.3 Methods for Development

One of the most attractive fields within Artificial Intelligence (AI) and ML these days is Deep Learning (DL). The art of utilizing DL Neural Networks (DLNN) to recognize patterns and obtain features of objects to learn about real world applications and make predictions. DL

will be the main approach and therefore the main machine learning method used in this project for developing object detection software. DL is an ideal approach for solving this kind of problem, as it has the ability to learn for itself what is the key elements and features that defines different objects.

Each of the above objectives will be handled separately in different development environments. For obtaining data, a CV method for snipping objects from existing operator interface images will be used in combination with an Emerson developed tool. For image classification, some of the obtained training data will be used to test how good classification models can get based on the limited data available for retraining. For multi-label classification, a large quantity of data obtained in the first objective in combination with different data preparation methods will be used to train existing models using transfer learning. Then, a separate program for sliding window pyramid scale NMS will use the trained multi-label classification model to detect objects in a full-scale operator interface image. This will hopefully result in a good multi-class object detector, detecting objects in a cluttered image containing a lot of information. This approach can hopefully also be used as bases for a one-stage or two-stage detector training data annotator. A new program will be developed for adjusting the previously detected object annotations and adding more objects (if applicable) to the annotation document. This annotation will be used to train the one-stage detection method, that hopefully will perform even better and detect more objects with better precision. The final result should be a software where a user can upload operator graphic images in picture format and get an analysis back in report format.

1.4 Assumptions

It is expected that the reader has general knowledge about the following subjects:

- AI, CV, ML, and some of its subfields.
- General structure of Artificial Neural Networks (ANN) including but not limited to neurons, weights, biases, activation functions and back-propagation. The foundational knowledge before starting this project is according to the IIA1420 curriculum, Machine Learning and Sensor Technology course held autumn 2021.
- General knowledge about computers and computer systems.
- Familiar with industry related objects such as pumps, motors, different valves, so on and so forth.
- What an operator interface image is, how it differs from human machine interface (HMI), and that these two terms “operator interface” and HMI is used interchangeably in the industry. In this report the “more correct” term operator interface image or operator interface graphics is used as they often are more complex than HMI and the term is more generalized than HMI.
- Some general knowledge about software development and terminology is also expected.

Note: There is no clear professional language or decisions when to use the different terms operator interface image, operator graphics, HMI etc. This can make things confusing. The important thing to remember is that they are all graphical user interfaces (GUI’s) that the operator uses in their daily work for controlling and monitoring a process plant or factory. The tools developed in this project will be able to detect objects in images. Whether it is

images from HMI or operator interface graphics does not matter if the models are trained to detect objects for those images.

To the best of my knowledge, there has not been done any similar research with regards to object detection in operator interface images. This is a new approach where the usage of deep learning neural networks will be trained for object detection to further extract viable information from pictures. However, as mentioned, there has been extensive research within the field of digitizing P&ID and other industrial related documentation, but the application researched in this project have a huge difference in complexity challenges. See chapter 2 Literature review for more information about previous work and research.

1.5 Outline of report

The report is structured in a way that is best suited for the reader to get a complete understanding of the project, and a good flow while reading. However, it is a large project, making it a large report. It is structured based on the IMRaD model (Introduction, Method, Result and Discussion), with additional chapters such as Literature Review, System Description and a Conclusion. The Literature Review is necessary to get familiar with previous work in this field. The System Description provides a description of existing systems, developed system during this project and development system used to execute the project and derive results. The Conclusion chapter gives a good short overview of all the results. Short overview of each chapter and what they contain:

1. Introduction – Intro to the project, background, and report structure.
2. Literature Review – Overview of related research within similar fields.
3. System Description – Existing system and system to be developed.
4. Methods – Deep dive into theory and analysis of approaches used to derive Results.
5. Result – Step by step execution and project results.
6. Discussion – Discussing results, reasoning, and improvements to be made.
7. Conclusion – Final conclusion of the project.

2 Literature review

Limited to no research was found within the field of object detection in operator interface images. The literature study will therefore revolve around object detection and digitization of P&ID's, documentation, and tiny object detection in satellite or aerial photos, also called Earth Vision research. After extensive literature research within the field of object detection in large images and documentation within industry, a few interesting papers were selected as inspiration to this project.

Shubham Paliwal, Monika Sharma and Lovekesh Vig wrote a paper in 2021 explaining recognition of line-drawn symbols in P&ID's using only one typical for each symbol for training. The method used in their research uses sampled pixels sequentially along different contour boundaries in the image. The sampled points are used to construct a graph that captures the structure of the contours. This graph is then fed into a Dynamic Graph Convolutional Neural Network (DGCNN) that is trained to classify symbol classes. To make the classification network more robust, they append embeddings from the ResNet-34 network. Instead of using the standard cross-entropy loss combined with a softmax layer used in most classification tasks, they use an Arcface loss function that has a higher discrimination power on the classifier. This is used to prevent similar looking objects to be misclassified. This Arcface loss function is interesting and can be applicable in this project thesis. This is truly a genius approach as it only needs one sample of each symbol for training. Thus, preventing the large quantity of annotation work prior to training, as would be the case for fully supervised techniques and other deep learning approaches. The result of this research is that it is comparable to previous work done with fully supervised techniques, but each unique new symbol introduced will require model retraining [4].

Two years prior to the OSSR-PID paper, the same group of people plus an additional person called Rohit Rahul wrote a paper on the same topic using fully convolutional networks for object detection. This means that instead of using a single sample of each symbol for training, they had to annotate multiple training images with segmented pixels that identified the different symbol classes available. This is a tiresome job, but often results in good prediction models. Their goal was to create the first (to their knowledge) end-to-end data extraction system for P&ID diagrams by wrapping a bunch of computer vision and machine learning methods into a single pipeline. By separating the information extraction into two parts: 1. text containing pipeline codes, and 2. graphic objects like pipelines and symbols, they manage to extract a large amount of information from P&ID's. After detecting text and inlet/outlet tags, they remove them using probabilistic Hough transform to reduce noise in the image. This then made it easier to perform step 2. When the pipeline intersection and symbols were detected, pipeline text and inlet/outlet tags are related to the symbols and pipelines using minimum Euclidean distance from center of text to center of object. This methodology of performing information extraction in multiple steps/iterations can be investigated for this master thesis project as well. The minimum Euclidean distance calculation is interesting for linking together tags and objects and will be adapted in this project thesis. The result of the Automatic Information Extraction from P&ID's was a proposed end-to-end pipeline, using CTPN [5] and FCN for pipeline code and symbol detection. A low-level image process technique to detect inlet, outlet, and pipelines to capture flow was used. And finally displaying the information in a tree like structure to describe the P&ID flow [6].

Another interesting paper from the Korea University, School of Mechanical Engineering is the Deep Learning-Based Method to Recognize Line Objects and Flow Arrows from Image-Format Piping and Instrumentation Diagrams for Digitization by Moon et al. [7]. This research proposes a three-step method where the first step is to remove outer border and title box in the diagram, second one detects continuous lines, line signs and arrows that indicate flow direction. The third step uses the result of the second step to determine line type and adjust them accordingly, then merge belonging lines and arrows [7]. The result of this research is a novel method for recognizing various types of lines in images. A preprocessing step for removing noise. A detection step for detecting continuous lines, line signs and arrows. And a postprocessing step for adjusting and combining lines and arrows. In the detection step, line thinning, and pixel processing techniques were applied to horizontal and vertical lines, and Hough transform was used to detect diagonal lines. A RetinaNet model is trained on data consisting of line signs and flow arrows. This paper's primary focus is line detection and classification, this is valuable research for all types of drawn documentation, also for the master thesis project presented in this report.

As mentioned, objects in operator interface images can be considered as tiny objects compared to the size of the actual image. It is therefore natural to look for papers with research on similar topics. Object detection in satellite or aerial images is quite interesting because everything on earth looks small from the sky or space. These types of Earth Vision object detection papers can therefore be considered similar topics.

Jinwang Wang, Wen Yang, Haowen Guo, Ruixiang Zhang and Gui-Song Xia wrote a paper on Tiny Object Detection in Aerial Images in 2021, where they used anchor-free object detection methods on a 700,621 annotated object dataset called AI-TOD for training [8]. They proposed a new method called M-CenterNet utilizing multiple possible center points instead of just one. They further analyzed the metrics and compared this method to more popular SSD, R-CNN and YOLOv3 anchor-based detectors. This method turned out to outperform all the other state-of-the-art detection methods on this particular AI-TOD dataset. It is worth noting that this AI-TOD dataset has a mean annotation object size of just 12.8 pixels, which is much smaller than traditional datasets in both aerial images and natural image detection datasets. These tiny objects also make the prediction very sensitive to Intersection over Unions (IoU) that can cause a bounding box to be misclassified by just missing the center point by one pixel. That's why, in this paper, they proposed the M-CenterNet learning network to improve the localization performance of tiny object detections. This paper is interesting because it gives an insight in another approach for optimizing object detection in large images and might serve as a basis for testing other approaches for Region of Interest (RoI) localization before classification.

There is also a paper on "Interactive Multi-Class Tiny-Object Detection" by Chunggi Le et al. [9] explaining an experimental way of utilizing multi-class interactive annotations to improve annotation and increase efficiency. As per my understanding it uses object detection methods in combination with user interaction to annotate image. When a user clicks an object, it will be bounded and predicted by a multi-class detection model. This paper was discovered at the end of the project thesis when there was limited time left to test new annotation approaches, but this is something worth looking into as it is quite new research.

3 System description

This chapter gives an overview of operator interface image systems, existing tools for design and analysis of such systems, and challenges using those tools. It also includes project scope with regards to what technology is being applied during project execution, what the general idea and goal for this project is, and some challenges when developing such systems. Throughout this project there has been developed two tools, one as an engineering tool and the other as a user tool, where both tools use cases are explained in the following subchapters. There is also added a subchapter explaining the set-up of development environments, package handling, and sharing of environments between development stations to optimize efficiency. The final subchapter explains how to efficiently obtain data from existing images to prepare for training image classification and object detection models that are going to be used in the engineering and user tool.

3.1 Operator interface image systems

What operator interface images (also known as HMI), from systems such as DCS, Supervisory Control and Data Acquisition (SCADA), or PCS, are, is not trivial knowledge for all. A short explanation is therefore provided in this subchapter with additional information on advantages and challenges with existing analysis tools.

3.1.1 Process control, interface, and component's structure

A short simplified explanation on DCS, SCADA or PCS structure and how operator interface images are introduced as a way for the operator to interact with the physical process [10]:

- There are multiple physical components (field devices) in the field controlling or measuring variables in the production, process flows, or assembly lines. The physical device is connected to a PCS using a communication protocol of some sort. This connection provides communication between the field devices and the PCS by exchanging some information, or control signals.
- On top of the PCS logic there is a graphical user interface (GUI) which is in this term is known as operator interface images, HMI, or SCADA image. In most cases, these images look like the snippet shown in Figure 1, where some example objects are marked and named in with blue color. These marked objects represent the physical components in the field, and the tanks and lines represent physical tanks and pipelines where it normally flows some sort of process substance. These physical components have tags, and values with regards to their control signal or some status or measurement value from the field (temperature, pressure, flow etc.), that are displayed in the image.
- From this GUI, the operator can monitor and control the physical process, and take critical actions during certain situations.

For a conversion project, migrating from old control images to new, it is important to be aware of all components in all process images, with all related tags and information. Different methods and tools for analysis are used to get this overview of all information that needs to be converted.

Note: The graphical components/objects for DeltaV Operate is referred to as “Dynamos”.

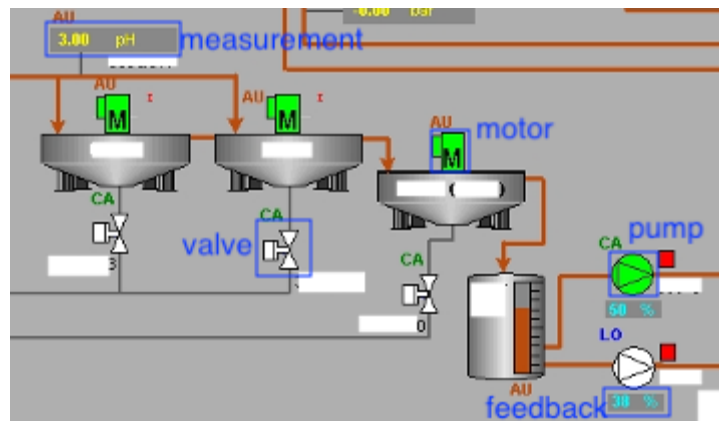


Figure 1: Snippet of operator interface image for a random process.

3.1.2 Existing design and analysis tools

These process control images are often constructed by graphical components such as object images, pixelated or scalable vector graphics (SVG) components, with configuration fields for values and tags that is retrieved from the control system. The graphical components are often path linked directly to the PCS logic, where the PCS logic acts as a “backend”, providing information to the GUI, or receiving operations from the GUI to adjust PCS logic, thus controlling the field physical component. The component animation logic (color animations, movements, etc.) is often standardized to visually represent some state of the “backend” logic and physical field device. For Emerson DeltaV as an example, the graphical components animation logic is standardized by libraries such as PBL and PCSD. Even if the animation logic is standardized, the look and feel of the objects can vary a great deal from project to project as the design is customizable based on the customers preference. The design changes can be custom color pallets, custom sizes, or symbols for objects, rearranging of values and tags connected to the objects etc. The design is customized, and objects linked from programs such as DeltaV Operate (old GUI design tool), and DeltaV Live (new GUI design tool).

When converting from old to new, or from one type of system to another, the engineer will use tools interacting with the old/existing source-code of the operator interface images (if available) to perform analysis for making sure the new design matches the old ones. For DeltaV Operate, this operator interface image source-code is Visual Basic for Application (VBA), controlling the graphical component animations and structuring the image layouts. The tool will typically iterate over all the source-code for each image and print a detailed analysis typically providing snippets of object and relevant information.

3.1.3 Advantages and challenges with existing analysis tools

Existing analysis tools have a huge benefit of being able to capture custom logic made in the image source-code, and therefore flag things that is not possible to capture by just looking at the images. However, using these existing tools require skill and system knowledge. It takes

time to perform such an analysis, and in lot of cases, using these tools are kind of an overkill. In simple terms, “These tools are not for all. And not always required.”.

The idea of this project is therefore to investigate if a more user-friendly tool can be made, that captures the basic structure, components, and information from images of the PCS GUI’s. The goal is NOT to make a conversion tool, as this requires huge amount of manual source-code mapping. This is discussed further in coming chapters.

3.2 Project scope

The project scope in short terms is to research the field of image classification and object detection to extract information from complex operator interface images displaying industrial applications such as production, process flows or assembly lines. More specific, the project will be performed in two iterations, where the first iteration involves training image classification models to test different network architectures and create an annotation tool. The second iteration will use annotated data from the annotation tool to train models for object detection, perform text extraction and linking, finally hosting these solutions in a user-friendly software. The final product from this thesis will be a tool that the user can access through a web solution, upload some operator interface images, and get a document for download in return. A flowchart of how the new analysis process will compare to a typical analysis with source-code tools or worst-case manual analysis, is provided in Figure 2.

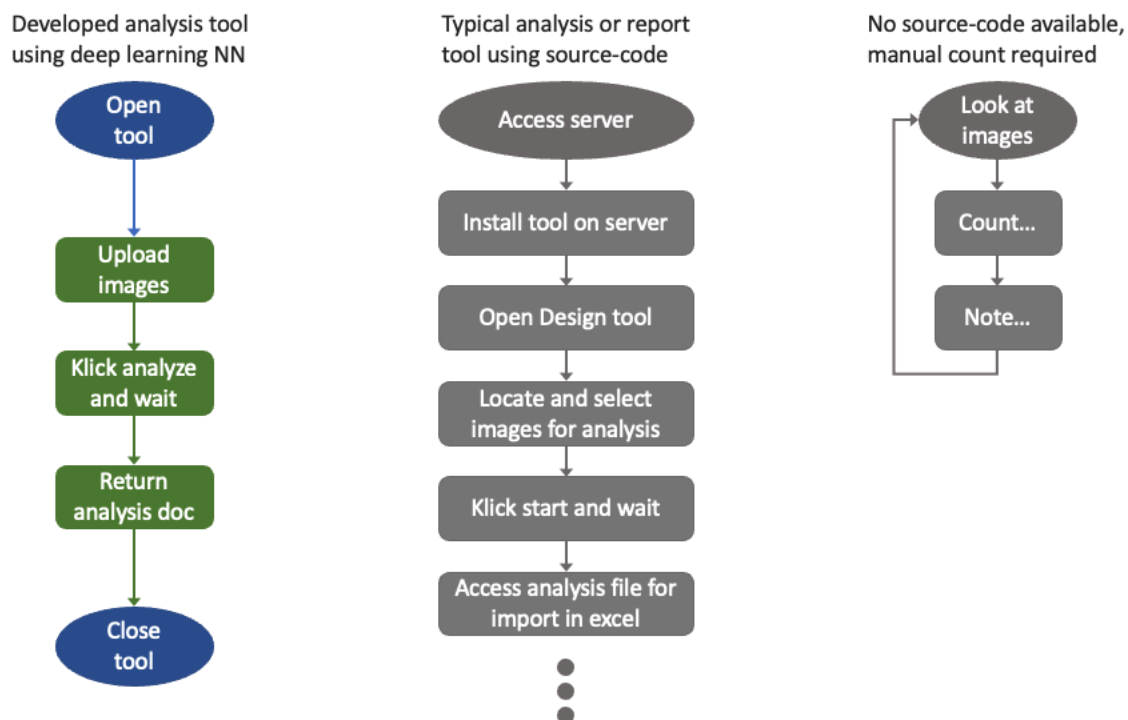


Figure 2: Flowchart of how the new user software (also called Program 2 and ICE) developed in this thesis will compare to other traditional ways of analyzing operator interface images. Note that the typical tool using source-code will differ from vendor to vendor in complexity, and the data retrieved from such a tool will always be more accurate than an image analysis tool using deep learning. But for simpler cases, as explained in chapter 3.1.3, the new tool will be much more efficient. And for cases where source-code from SCADA or PCS is not available, the new software will outperform the manual counting method on the right.

3.2.1 Technology

Machine learning is a subfield of AI that involves the design and development of algorithms and models that can predict future outcome and make predictions based on data. The models or algorithms are designed in a way that makes them learn similar to the way humans learn, and by repetitive training, gradually improving its accuracy [11]. There are multiple subcategories within machine learning, but for this project, the focus will be on supervised learning. Supervised learning is a way of training models based on data, providing it with both the input and output data pair. This means to provide a model with an input object, for example a picture of a valve and giving it the output label “valve”. The machine learning model will then learn, after repetitive training, that this object is a valve. The supervised machine learning model itself can be algorithms such as SVM, Decision Tree, Random Forest, ANN, etc. As mentioned in the introduction, this project will utilize the powers of deep learning. Deep learning is a type of ANN, but with a more complex and deeper structure [12]. This way of configuring an ANN makes deep learning neural networks particularly suited for complex tasks such as image classification and object detection.

3.2.2 The general idea

The idea is to create deep learning models for classifying multiple objects within operator interface images. The objects should be marked with a bounding box and labeled. The information about the object locations and label should be extracted into a separate document. The extracted data can further be used for complexity analysis, rebuilding new displays, and other applications as discussed in the introduction. Data obtained for training the machine learning models will also give a reference for identifying objects that are common within industrial solutions such as pumps, valves, measurements etc.

Two project approaches are taken into consideration for this project:

1. Training a multi-label classification model to recognize different objects based on their features. This multi-label classification model will be used in combination with a sliding window algorithm to detect different objects within the sliding window reference frame. This is also referred to as multi-class object detection. The objects that are identified will be marked with a bounding box, label, and number. The label, number and position of the object will be noted in a separate document for further analysis. The training data will be extracted from standard dynamo-sets from the DeltaV Operate library and existing customer graphics. The dynamo-sets and graphics comes as full images with a lot of components. Each component will be extracted using OpenCV canny edge detection and snipping tools. This will simplify the training process and reduce manual labor. This first method serves as a good starting point for the project as it limits large amounts of manual annotation preparations.
2. Training a one-stage detection method for object detection within large images. The objects detected will be marked with a bounding box, label, and number as for previous approach. The downside to this method is that it requires large amounts of manually annotated operator interface images to train a You Only Look Once (YOLO) or Single Shot Detector (SSD) algorithm. However, this will result in a more efficient and better object detection solution than approach 1.

If approach 1 yields satisfying results, it could be used as an annotation method for approach 2. Approach 1 will learn to classify different objects, then using sliding window over full scale images to annotate objects. Small corrections, removals or additions will be required before giving these annotated images to the one-stage detector model. An engineering tool for doing these adjustments will be developed and part of the project.

3.2.2.1 What is annotation?

It is important to clarify exactly what annotation is. Annotation is the process of marking an object in an image, giving it a bounding box and a label. The coordinates x, y, width, and heights of the bounding box is stored in a document with the given object label. The object is now annotated.

3.2.2.2 Challenges of object detection in operator interface images

Doing object detection in images that are generated in 2-dimensional space such as drawings and documentations have huge advantages compared to real world images or video. The more traditional challenges such as light conditions, angle of objects, line of sight, dirt and other real-life factors are none existing. However, these types of 2-dimensional drawn or designed images have other challenges that needs accounting for. Operator interface images contains a large number of objects, lines and text that represent various information. This introduces the challenge of noisiness and limited features because of object similarities, causing higher probability of misclassification. e.g., the pump object that can look a lot like an iso standard motor object, and an analog numeric value that look similar to the status object and other text information. List of objects and similarities are provided in Table 15.

The standard dynamo-sets that are initially thought of as input for training the models is offline mode objects, and therefore look different from the online operator interface image objects. This might cause challenges, and more training data will have to be collected from existing graphics manually to achieve a model with good accuracy and generalization. An example of a full-scale operator interface display image and how crowded they can get is available in Figure 9.

Note: All images are 2-dimensional, but real-life photographs and video is taken in a 3-dimensional space.

3.2.3 The general goal

Identifying objects within a frame of reference, marking the object and extract information about its location, classification type and tag related to object. Creating a foundation for operator interface display image complexity analysis.

3.3 Semi-automated annotation tool (Program 1)

A custom tool is developed as a part of this project, to optimize the annotation process for generating training and validation data for modern object detector models. It uses the models trained in the first approach described in chapter 3.2.2. See full use case diagram (UCD) of this engineering tool software in Figure 3. The use cases describe the software functionality, often started by a verb as a thing that is executed. It is an important part of the unified

process, and is used for visualizing the functional requirements [13]. The goal is to focus on how the system will fulfill the requirements from and to the actors, focusing on the user. For this application, it is required that the hardware has a GPU to do the classification. The GPU is therefore listed as a non-functional requirement. The user will upload an image, crop it or leave it in original size, upload a multi-label classification model, perform a pre-analysis, then do annotation if required, and finally export the training data. Analysis and design of the software is provided in chapter 4.5.1. The final resulting software test is shown in chapter 5.2.2.

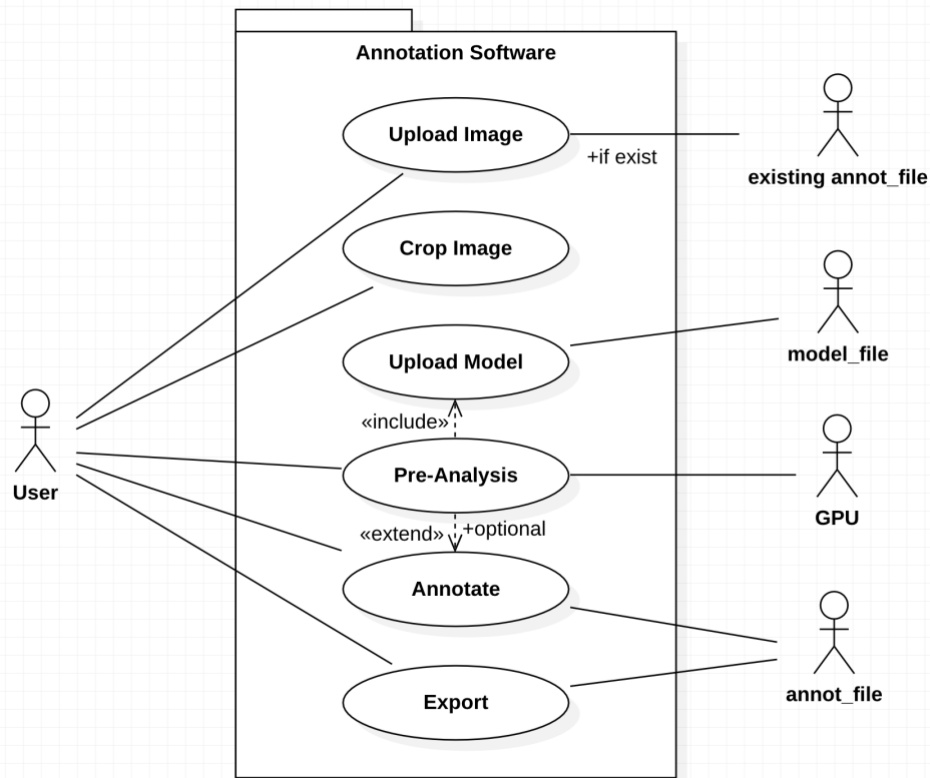


Figure 3: Use Case Diagram annotation software.

3.4 Industrial Component Extraction tool – ICE (Program 2)

As a final solution for this project, a user-friendly software will be developed where the user can upload one or more images and get an analysis report in return. The objects in the images should be correctly labeled and listed with correlated information.

A UCD is created to describe the functional requirements of the system, seen in Figure 4. Users will initially interact with the UI to upload images and start analysis. In the backend, all inclusions and extensions are processed, to finally return an analysis report for the user to download. The user interacts with the UI to view progress on the display and download the final analysis document. Remember that “include” is relationships between use cases that is necessary to achieve the end goal of a use case. That means that the “Perform Analysis” use case relies on the “include” linked use cases, and the “Generate Excel” is just an extension of this parent use case. The “Generate Excel” is however necessary for the user to be able to download the analysis, and therefore included as an “inclusion” of the “Download Analysis”

use case. User is the primary actor; upload folder and display are supporting actors providing services to the use cases.

This final software will be an alternative to existing analysis tools interacting with source-code as explained earlier. It can be used by none-technical personnel to easily retrieve a basic analysis of process control (operator interface) images. The analysis and design of the UCD with additional requirements is elaborated in chapter 4.5.2. The software testing and result is explained in Results chapter 5.2.5.

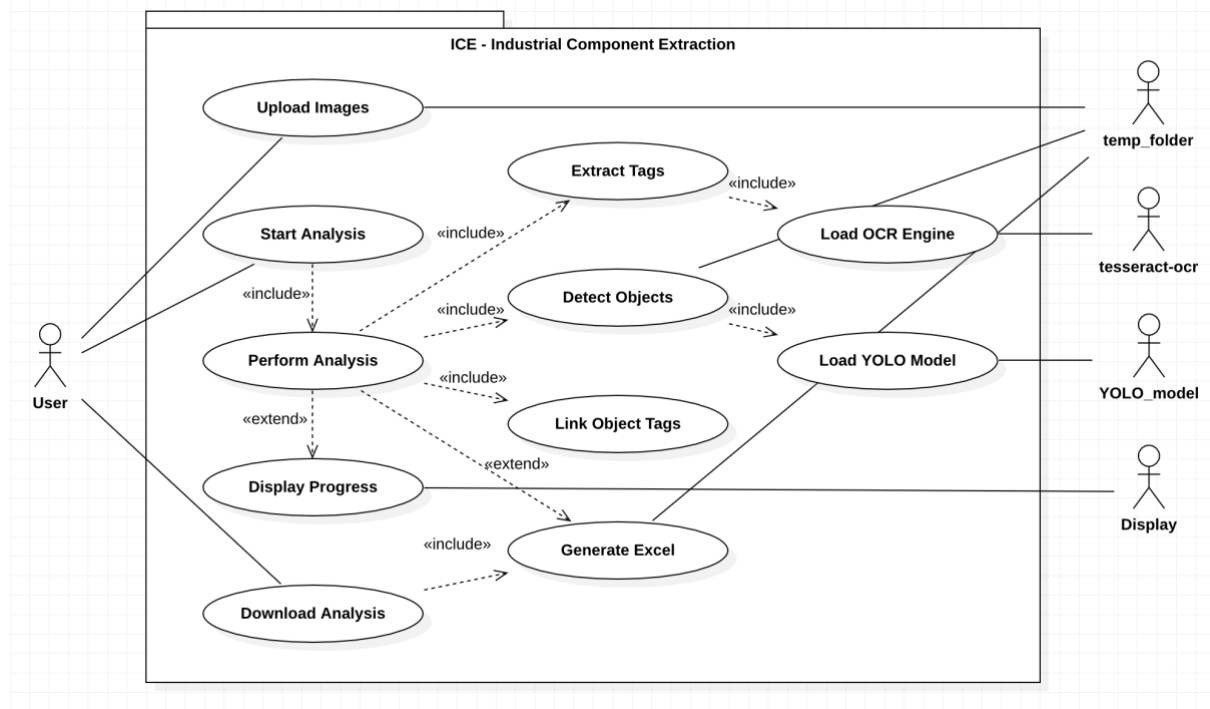


Figure 4: Use Case Diagram for the final analysis software called ICE – Industrial Component Extraction.

3.5 Development environment

Deep learning tasks can be computationally heavy to perform, especially during development, while training and testing models. A decent hardware and software environment is key for efficiency and performance. The development station and environment will be hosted on a local computer in the office, with remote access via TeamViewer. The computer will also be connected to a Raspberry Pi4 (RPI) that is configured to reboot if/after power loss. This RPI can also be reached with TeamViewer, where a wake on LAN magic package can be sent from the RPI to the development station, thus turning it on. Communication and device setup is illustrated in Figure 5. The RPI is an alternative solution instead of configuring the office router for remote turning on and off the development computer. The development station is configured with Wake On LAN in bios and on the Ethernet Controller.

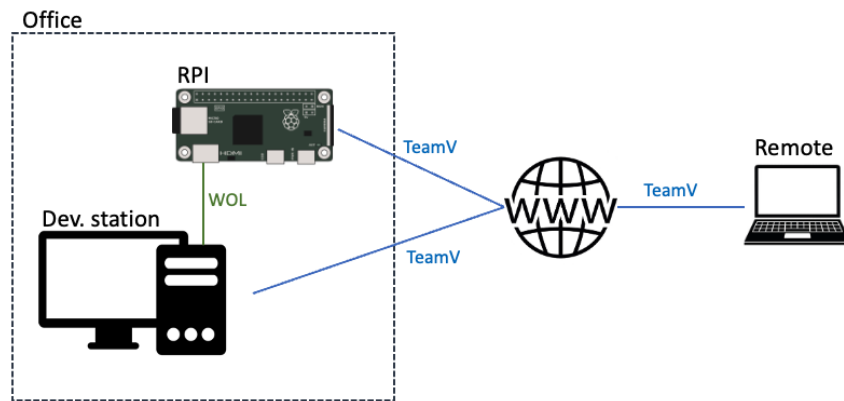


Figure 5: Illustrating device and communication setup.

3.5.1 Hardware environment

When deciding upon hardware components for machine learning development, GPU and cooling will be the most crucial components. More about why GPU is a key component for machine learning, and how it differs from CPU computations in Appendix D. This information is separated into an appendix as it is not key information for the project, but important knowledge when working with machine learning in general.

For this project, an old gaming computer is reinstalled and set up with machine learning environments utilizing GPU support. The computer has a GTX1080 overclocked GPU, an Intel Core i5-8400 processor, 16gib of DDR4 RAM, 250gib M.2 SSD. Table 1 gives an overview of components and part numbers used in the development machine.

Table 1: List of development environment hardware

Part name	Part number	Description
MSI B360I Gaming Pro AC, Socket-1151	B360I GAMING PRO AC	Motherboard
Intel Core i5-8400 Processor	BX80684I58400	CPU
Asus GeForce GTX 1080 Rog Strix	ROG STRIX-GTX1080-A8G-GAMING	GPU
Corsair Vengeance LPX DDR4 2400MHz 16gb	CMK16GX4M2A2400C14	RAM
WD Black SSD 250GB M.2 PCIe	WDS250G2X0C	SSD
Cooler Master MasterWATT 650	MPX-6501-AMAAB-EU	PS

3.5.2 Software environment

The pc is reinstalled with Windows 10, student edition. Windows is often beneficial due to good commercial software and drivers support, and well suited for interacting with the GPU hardware. However, for development, Ubuntu gives more flexibility in package installation and environment customization. Windows Subsystem for Linux (WSL2) is installed and configured on the developing machine. WSL is Microsoft's solution to running a native Linux OS on Windows, capable of installing different Linux distros. Native folder structure support using explorer.exe is integrated in WSL and Windows. Visual Studio Code is set up as code editor, installed with Python support packages to turn this into an Integrated Development Environment (IDE). The WSL is installed with git and Docker support, connected to GitHub and installed with MiniConda environment for Python development. MiniConda is a smaller version of Anaconda which is a full fledged data scientist Python environment for development. Jupyter Notebook is also installed for testing and developing different models and methods. Conda virtual environments is used for each task in this project and is important to separate package support to prevent conflicts. It also eases the work of taking backups and installing dependences. Virtual Python environments is especially important when doing something that require a different package version or contain conflicting packages to previous programs. A more detailed explanation on why WSL is chosen, how it compares to other virtual machine or dual boot solutions as well as how it is configured is provided in Appendix D. This information is separated into an appendix as it is not key information for the project, but interesting if the reader wants to replicate this project at a later stage or learn more about recommended solutions to work with machine learning in Python.

3.5.3 Web solutions

Kaggle and Google Colab is used for minor testing of package dependencies and when getting to know the different libraries. FastAI provides all its teaching materials on these two platforms, so it is easy to get started by running code directly within their books in Colab.

3.5.4 Software used

- Chrome
- WSL
- Visual Studio Code
- Jupyter Notebook
- Zotero
- StarUML
- OneNote
- Word
- Excel
- PowerPoint

3.5.5 Python frameworks and libraries

A short description of the packages used is provided in the following subchapters.

3.5.5.1 FastAI and Fastbook

The FastAI and fastbook libraries developed by the engineers at fast.ai are libraries built on top of PyTorch as high-level API for quickly training and deploying deep learning models. FastAI's goal is to make deep learning a low effort field to get started with, and at the same time provide state-of-the-art results in deep learning domains [14].

The main benefit of using this high-level API compared to using PyTorch directly is that the simplified solution removes the need for writing custom training loops, defining data blocks, data loaders and handling GPU acceleration. The FastAI API provides easy support for data augmentation, pretrained models and interpretability with a range of tools for visualizing results.

3.5.5.2 Ultralytics

Ultralytics is a team of skilled people determined to make AI easy [15]. They have an open-source Python library called Ultralytics which is available under the GNU General Public License. The library contain the source code for the YOLOv8 network/model, and all functions for training, validating, testing and deploying models [16].

3.5.5.3 PyTorch and Torchvision

PyTorch is a popular open-source machine learning framework in Python. It provides tensor computation framework for building and training deep learning models. PyTorch has a more complicated APIs than FastAI and is well suited for more skilled personnel wanting more flexibility. Torchvision is a library part of the PyTorch framework and is designed to be used on machine learning applications regarding vision tasks such as image classification and object detection [17].

3.5.5.4 Pytesseract OCR

Pytesseract is an open-source Optical Character Recognition (OCR) Python library, based on the OCR platform originally developed by Hewlett-Packard and later taken over by Google [18]. Pytesseract serves as a wrapper for the Google Tesseract-OCR engine and can run as a stand-alone script directly in Python. It is used in combination with OpenCV to extract text from images or documents where copying is not possible [19]. OpenCV often serve as the input image preprocessing step before utilizing Pytesseract OCR. Pytesseract OCR is often referred to as image-to-text Python OCR.

3.5.5.5 OpenCV

Open-Source Computer Vision Library (OpenCV) is a popular computer vision and machine learning Python library used for tasks such as object detection, image processing, analysis, text recognition etc. It is less deep learning focused than FastAI and PyTorch, but it has a large variety of functions that perform various image and video processing tasks. Example of processing task would be edge detection methods, object tracking, feature detection, filtering and blurring etc. [20]. In this project, the OpenCV library is used for extracting training and validation data by using its methods for edge detecting, filtering, then bounding box the area around objects and finally snip each object into a separate folder.

3.5.5.6 NumPy and Pandas

Libraries for numerical and tabular computing and data handling in Python. It is one of the most used Python libraries in data science, scientific computing, and finance [21]. In this project it is used to preprocess and prepare data for machine learning. They are both excellent library for viewing, rearranging and preprocessing data to get the right format for a task. The FastAI DataBlock used for training deep learning models requires that the input data is structured in a certain way. This differs from application to application, but in most cases the DataBlock require a data frame containing the training and validation input and label. NumPy and Pandas has a range of functions to help with data preparation, converting the data into data frames that the FastAI DataBlock can read.

3.5.6 Datasets

No datasets are included in the report as they are confidential to the customer of Emerson Automation Solutions. During this project, there is obtained large datasets for training, validating, and testing machine learning models for image classification. This dataset does not reveal any classified information and can be requested by emailing the author.

The datasets for training, validating, and testing object detection in operator interface graphics are not available for anyone outside the Emerson organization.

3.6 Data collection and preparations

Collecting and preparing data is the foundation upon which deep learning models result lay. This chapter explains how data for training and validation was collected for this project, and how to prepare and structure the data for deep learning tasks.

3.6.1 Obtaining training data

For a machine to learn how the world works it needs data. Data is the key element in all machine learning algorithms or models. This is also true for the application of creating a deep learning neural network to detect and identifying objects in operator interface display images. For this specific project, the training and validation data is images of the objects that the machine learning models are going to learn to detect. This data can be extracted from standard dynamo-sets or existing graphics, by obtaining the full-scale images and snipping out individual objects. For extracting custom graphics and objects that exist in DeltaV Operate, Emerson has developed a tool called URD collection tool. The tool is installed on the engineering stations and used to extract all images and objects from the DeltaV Operate systems VBA code. Further extraction of individual objects from these full-scale images can be done either manually by snipping or by writing a script for object extraction using OpenCV or similar libraries in Python.

3.6.1.1 Emerson URD collection tool

The URD collection tool interacts directly with the VBA source-code of DeltaV Operate graphics, collecting objects from those images. This tool is a preferred way to extract objects if the old system is DeltaV Operate and the source-code is available. However, the data

extracted from this tool will be offline state objects and might not represent the objects in this application a good way.

3.6.1.2 Object extraction tool using OpenCV

If the source code is unavailable, or live objects is required, it may be necessary to extract training data objects directly from raw image files. To streamline this process, a program is developed for object extraction, which eliminates the need for manual snipping tool usage. OpenCV has plenty of methods that, in combination with each other can help simplifying the training, validation, and test set object extraction from raw operator interface images. A script has been created to convert input images to grayscale, apply a threshold and dilation, and identify contours using the "find contours" method. These contours are bounding boxed using a separate method from the OpenCV library. Each of these bounding box objects is then snipped from the full-scale image into a separate folder. Some of the objects are applicable for use in training, validation, and test sets, and some are just "half pictures" containing a lot of noise. This method is not perfect, but it gives a bases for the next step of manually sorting objects into class folders for labeling.

3.6.1.2.1 Step-by-step guide

An example of a raw input image for extracting objects that are going to be added to the training, validation, and test set is shown in Figure 6. The image has been somewhat retouched for anonymization purposes.

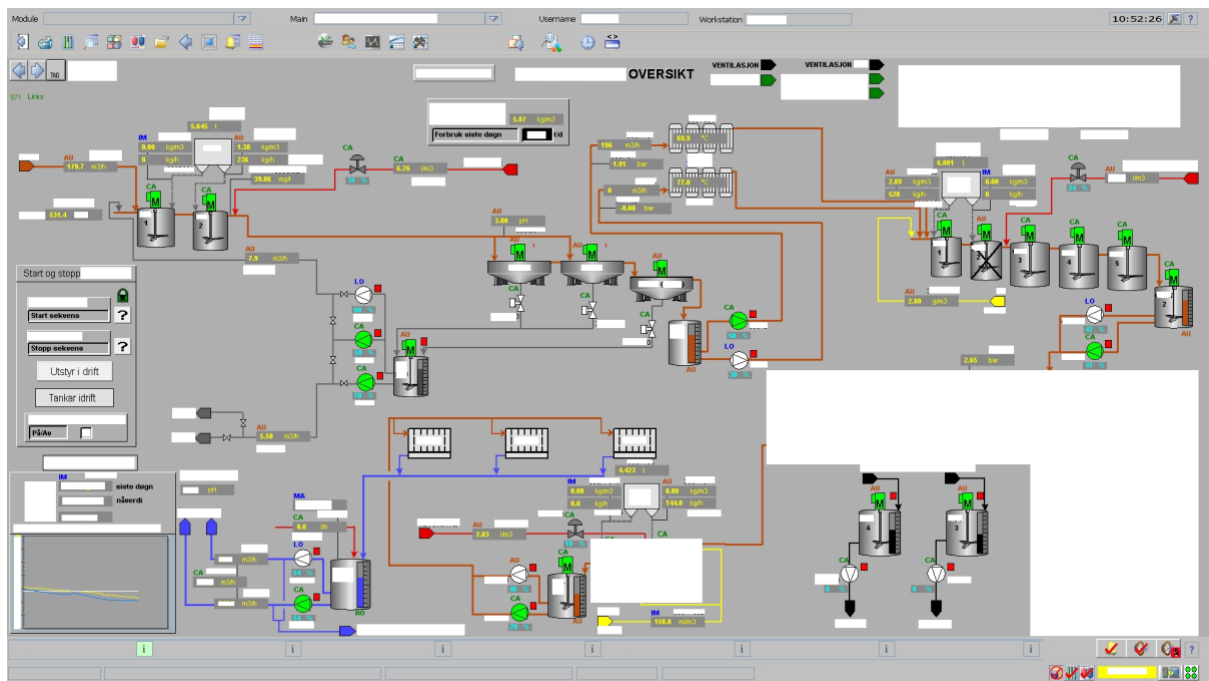


Figure 6: Example of operator interface graphics image for a process.

5-10 images are putt into a folder named "Displays" as shown in Figure 7.

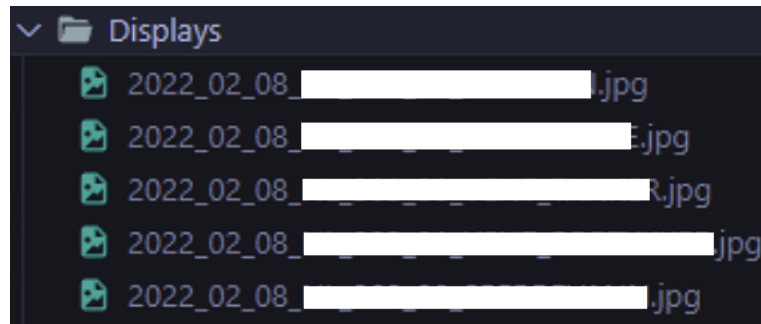


Figure 7: Raw files in Displays folder.

The top navigation bar and the bottom status bar causes noise in these images, so a simple crop of these images is preferred before object extraction. A script for cropping multiple images in the “Display” folder is created as shown in Table 2. The new cropped images are put into a separate folder marked “cropped” as shown in Figure 8.

Table 2: Script for cropping multiple images in a Displays folder.

```

folder_path = Path("/home/user/git/ext_obj/Displays/")
cropped_path = Path("/home/user/git/ext_obj/cropped/")

for images in folder_path.ls():

    # Open the image file
    image = Image.open(images)

    # Define the crop box (left, upper, right, lower)
    box = (10, 120, 1920, 1000)

    # Crop the image
    cropped_image = image.crop(box)

    # Save the cropped image
    cropped_image.save(str(cropped_path)+str(images).
removeprefix(str(folder_path)))

```

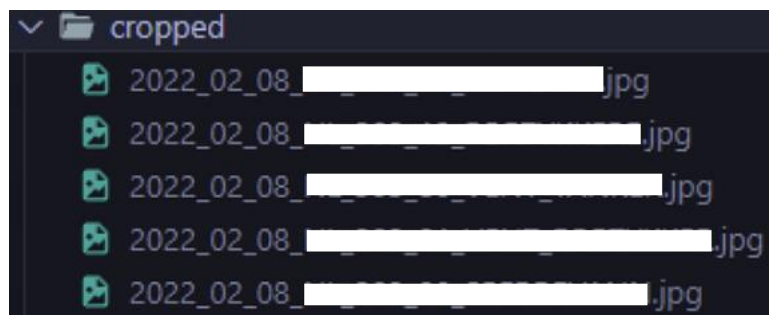


Figure 8: Cropped files in cropped folder.

The result is a cropped images only containing the process image as shown in Figure 9.

3 System description

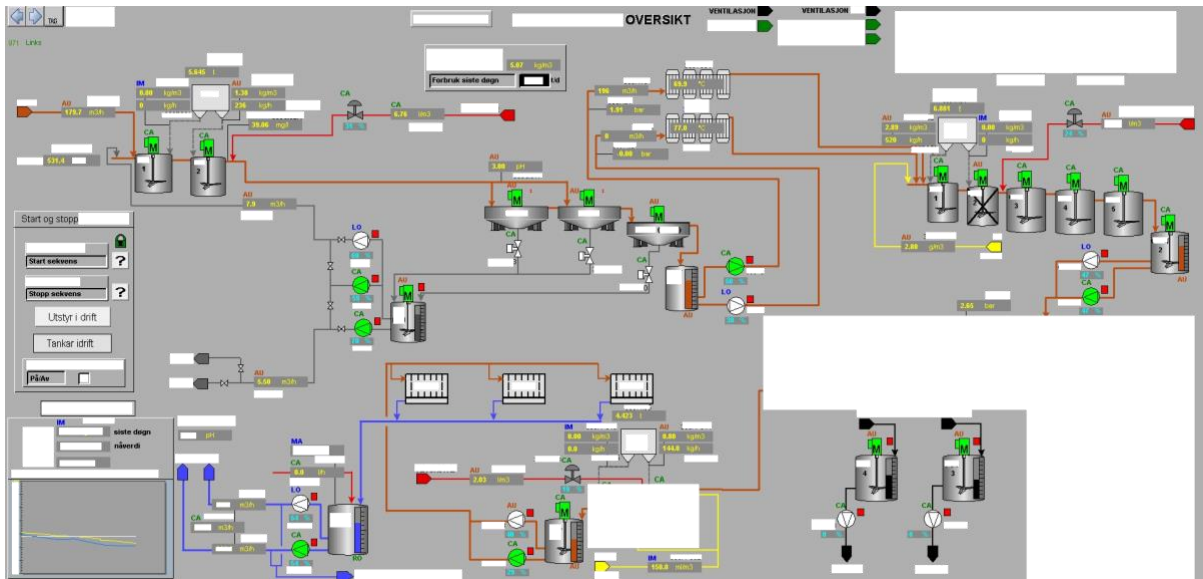


Figure 9: Cropped example of operator interface graphics image for a process.

A script utilizing the functions of the OpenCV library is created to extract components from the image, that can further be used for training deep learning models. Start by adding a variable to the image folder path marked for extraction, shown in Table 3.

Table 3: Variable path declaration.

```
path = Path('/home/user/git/ext_obj/cropped/')  
ROI_path = ('/home/user/git/ext_obj/ROI_ext/')
```

Then, for simplicity, put all the image preprocessing into a function. This function has some options for canny edge thresholding or binary inverse threshold. There is also an option to select between different types of canny edge threshold types, see Table 4. All these parameters are experimental and must be tested. Highly dependent on the background color of the image. The best result for these images is the canny edge threshold with triangle threshold type. An example of how this image preprocessing looks like is shown in Figure 10.

Table 4: Image preprocessing function.

```

def param(gray, background="gray", canny=True, threshtype="triangle"):

    if canny == False:
        if background == "white":
            blur = cv2.GaussianBlur(gray, (5,5), 0)
            kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (7,2))
        elif background == "gray":
            blur = cv2.GaussianBlur(gray, (1,1), 9)
            kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (1,1))
        thresh = cv2.threshold(blur, 0, 255, cv2.THRESH_BINARY_INV +
            cv2.THRESH_OTSU)[1]
    else:
        if background == "white":
            blur = cv2.GaussianBlur(gray, (5,5), 0)
        elif background == "gray":
            blur = cv2.GaussianBlur(gray, (5,5), 0)

        if threshtype == "otsu":
            otsu_thresh, _ = cv2.threshold(blur, 0, 255, cv2.THRESH_OTSU)
            otsu_thresh = get_range(otsu_thresh)
            edge_otsu = cv2.Canny(blur, *otsu_thresh)
            thresh = edge_otsu
        elif threshtype == "triangle":
            triangle_thresh, _ = cv2.threshold(blur, 0, 255,
                cv2.THRESH_TRIANGLE)
            triangle_thresh = get_range(triangle_thresh)
            edge_triangle = cv2.Canny(blur, *triangle_thresh)
            thresh = edge_triangle
        elif threshtype == "manual":
            manual_thresh = np.median(blur)
            manual_thresh = get_range(manual_thresh)
            edge_manual = cv2.Canny(blur, *manual_thresh)
            thresh = edge_manual
        kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (1,1))

    return [thresh, kernel]

def get_range(threshold, sigma=0.33):
    return (1-sigma) * threshold, (1+sigma) * threshold

```

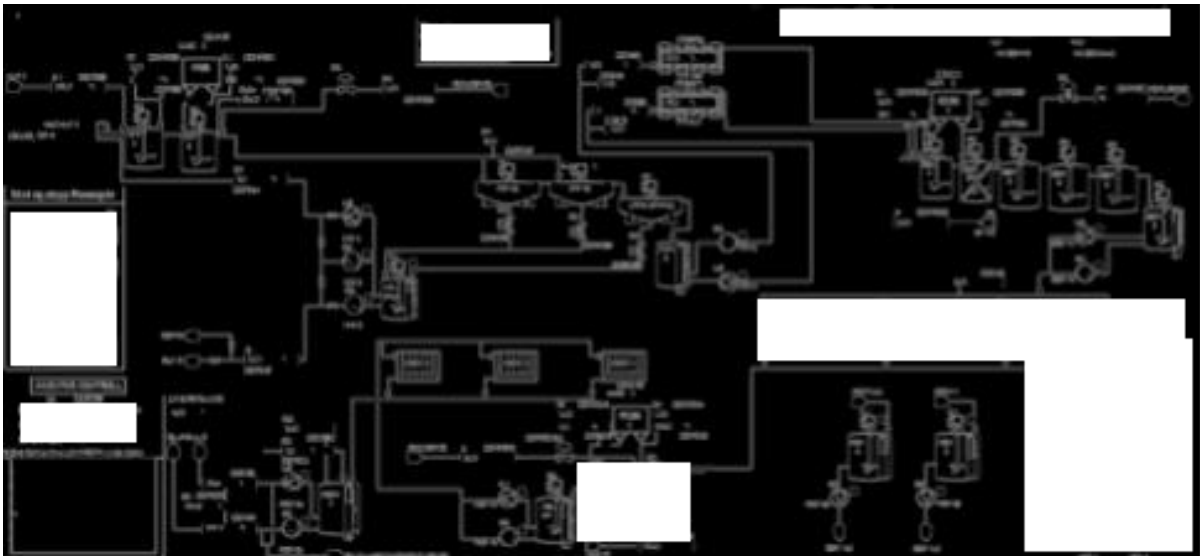


Figure 10: Example image with applied blur, edge detection and dilation.

The for-loop code unpacking, applying filters, contours and snip all rectangle bounded objects from all images in the “cropped” folder is shown in Table 5.

Table 5: Looping through all images, applying filters and contours and snipping objects.

```

for i in path.iterdir():
    image = cv2.imread(str(i))
    original = image.copy()
    name = "lib_" +
        str(i).removeprefix("/home/user/git/ext_obj/cropped/").removesuffix(
            ".PNG")

    #grayscale, Gaussian blur, Otsus treshold, dilate
    gray = cv2.cvtColor(image,cv2.COLOR_BGR2GRAY)
    thresh, kernel = param(gray)
    dilate = cv2.dilate(thresh, kernel, iterations=1)

    # Find contours, obtain bounding box coordinates, and extract ROI
    cnts = cv2.findContours(dilate, cv2.RETR_EXTERNAL,
        cv2.CHAIN_APPROX_SIMPLE)
    cnts = cnts[0] if len(cnts) == 2 else cnts[1]
    image_number = 0
    for c in cnts:
        x,y,w,h = cv2.boundingRect(c)
        cv2.rectangle(image, (x, y), (x + w, y + h), (36,255,12), 2)
        if w>15 and h>15:
            ROI = original[y:y+h, x:x+w]
            cv2.imwrite(os.path.join(ROI_path,
                name+"_ROI_{}.png".format(image_number)), ROI)
            image_number += 1

```

These objects are stored in the “ROI_ext” folder shown in “ROI_path” variable. As seen in Figure 11, this can get messy, but based on experience it is better to get too many objects and manually filter out the bad once than getting too few objects.

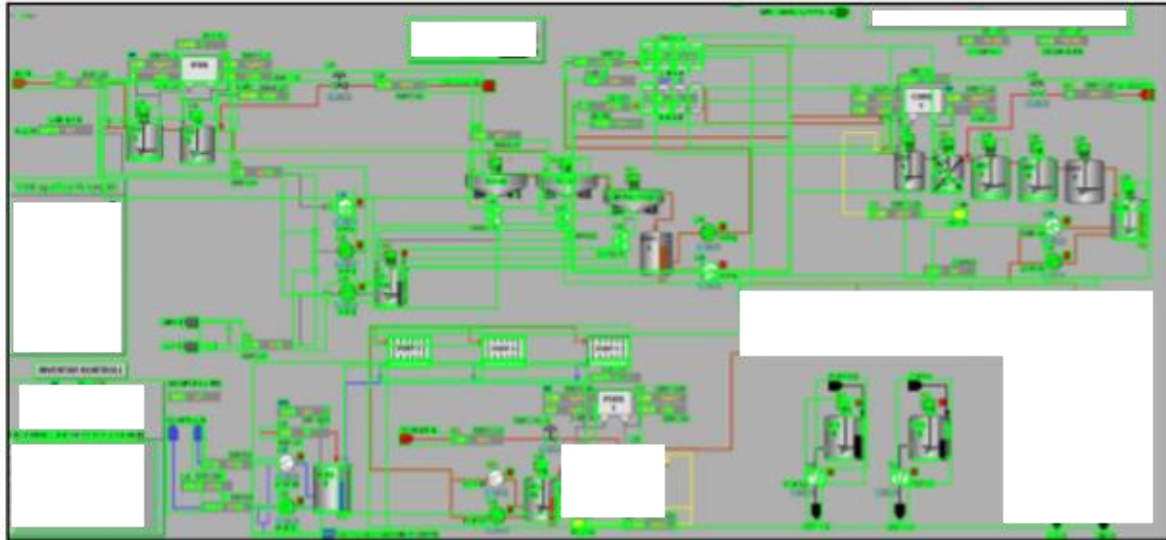


Figure 11: Bounding boxes on objects in example image.

Full source code can be obtained by combining Table 3, Table 4 and Table 5. Just remember to alter the file paths to match directory.

3.6.2 Data preparations

After extracting individual objects from the full-scale images using the python script, the extracted objects need to be manually moved into separate folders. Each folder represents different object classes. This is also one way of labeling the data. Separating the objects into different folders makes it easier to keep track of what classes are defined, and each object within a class folder is now labeled. This folder structure will be sufficient for the first task of single-label classification using the FastAI library shown in chapter 5.1.1. The folder structure also simplifies the next step of creating a specification file for multi-label classification. This specification file contains image name (object) and class type (label) and a decision parameter if the object should be used for training or validation. Table 9 gives an example of how such a file will look like. This specification file is important for the multi-label classification shown in chapter 5.1.2.

3.6.2.1 Validating extracted data

While going through the extracted data it is important to check that the objects put into class folders represents the data that the model needs to learn. If there is a snippet containing a valve, the valve should be “whole” and not containing a lot of noise i.e., Figure 12. A snippet containing a single object should be put into a single class folder as shown in Figure 13. If multiple objects are visible in the snippet, the snippet should be put into a multi-object folder.



Figure 12: Snippet of control valve object, label name “valve_p”, for valve pneumatic.

3.6.2.2 Object classes and folder structure

Object classes and labels are per this report the same by definition. An object class is a common denominator for multiple objects with the same class as label. Different folders are created with different object class names and these folder names will be the contained images labels. Meaning, each image will be labeled with the respective parent folder name. For the first iteration, folders containing names such as “valve”, “pump”, “motor”, “value”, “status” and “static” will be created as shown in Figure 13. Multi-object folder should have all classes listed on the folder name separated by a space as shown on the “bargraph tag” and “bargraph value” folder in Figure 14. Note that the single-label classification method only requires images in individual class folders to start training (no object specification file needed).

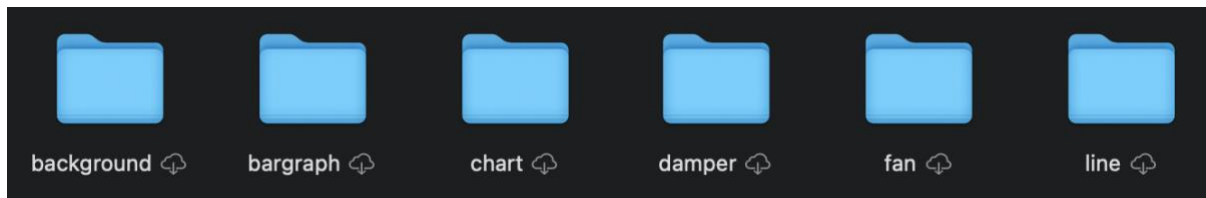


Figure 13: Single class object folders.

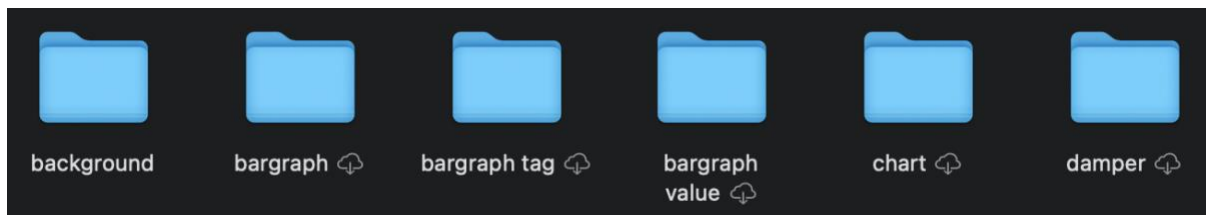


Figure 14: Multi class object folders.

3.6.2.3 Annotating data for object detection

Object detection using more sophisticated algorithms such as YOLO, SSD or R-CNN requires training data in a different format than image classification. As mentioned earlier, a deep learning neural network requires training data that represents its application in a good way, this is common for both object detection and image classification. Object detection deep learning algorithms therefore require training images representing the application with annotated objects to learn how to detect these objects in cluttered environment. The creation of such training data is tiresome, and often a job that is outsourced. For this project, since it is a new application of object detection, all the training data needs to be annotated from scratch. Starting with some operator interface images, then an annotation software that can export the image annotation files in the required format for the selected object detection framework. Numerous annotation tools are available, but due to time constraints for this project and the aversion to repetitive work, the multi-class object detector mentioned previously as approach 1 in chapter 3.2.2 will be utilized to develop the semi-automated annotation tool introduced previously in chapter 3.3. The annotated image will be in raw format without any visual

3 System description

bounding boxes on it, but each annotated image will have an annotation file that specifies the object class and localization (bounding box). Example of an annotated image using the annotation tool is provided in Figure 87. The annotation process will be explained in more detail in chapter 5.2.3.

4 Methods

This chapter provides information about the technology used to achieve the project goals, and a deep dive into important techniques and methods used to do so. It will introduce and explain concepts that might not be general knowledge to the reader and is beneficial to know before moving on to the Results chapter. The analysis and design of the two software programs (Program 1 and Program 2) developed, is also included as a subchapter. As mentioned in chapter 3.2, two approaches are considered during this project. The first approach require knowledge about image classification, the second approach require knowledge about object detection. But first, some general knowledge of what deep learning artificial neural networks is, and how convolutions are used to improve them in the application of working with images.

4.1 ANN - Artificial Neural Networks and Convolutions

In general, ANN are used for extracting information from data without the need for feature engineering, as more traditional ML approaches would require. In this chapter the focus will be on usage of traditional Feedforward ANN (FANN) for image classification, why this approach is sub-optimal, and how the FANN architecture for image feature extraction can be improved by the help of Convolutions, using a different ANN architecture called Convolutional Neural Networks (CNN).

4.1.1 Using ANN for image classification

Deep learning neural networks uses layers of neurons with simple (linear) mathematical functions in combination with activation functions (creating un-linearity) to extract features from images. Each feature in a previous layer will be feed into the next layer, extracting more and more features. This is called abstraction. An example is provided in Figure 15 where a series of simple mapping functions extract different features of an image in each layer, resulting in a complex pixel mapping prediction in the output layer [22]. Layer 1 is called the input layer because it contains the variables visible to the human eye, in this case, each image pixel is an input. If an image has a size of 28x28 pixels (black and white) this gives an input layer of 784 nodes. Layers between input and output layer is called the hidden layers because the values extracted to obtain features is not given in the input data but determined by the model. The number of hidden layers (network depth) and number of neurons in each layer are trivially chosen while designing the network. The last layer is called the output layer, as it gives a final prediction based on the previous layers feature extraction of the input. The size of the output layer is dependent of number of classes defined for classification. When the first hidden layer has detected edges using i.e., pixel intensity, the feature is fed forward to the second hidden layer where it detects corners and contours. Based on the corners and contours, the third hidden layer can identify object parts, and then start to understand what prediction should be prioritized (weighted highest). This is a simplified theoretical explanation of a feed forward deep learning neural network used for image classification, and therefore a theoretical assumption of abstraction within this network. In practice, the abstraction in the receptive fields would appear a lot more random than detecting edges, corners and contours, and objects. This is due to the architecture of the feed forward neural network. Using such a

network for image classification would be sub optimal, and often replaced by a combination of convolutional layers, max pooling layers and a dense layer, resulting in a fully connected convolutional neural network.

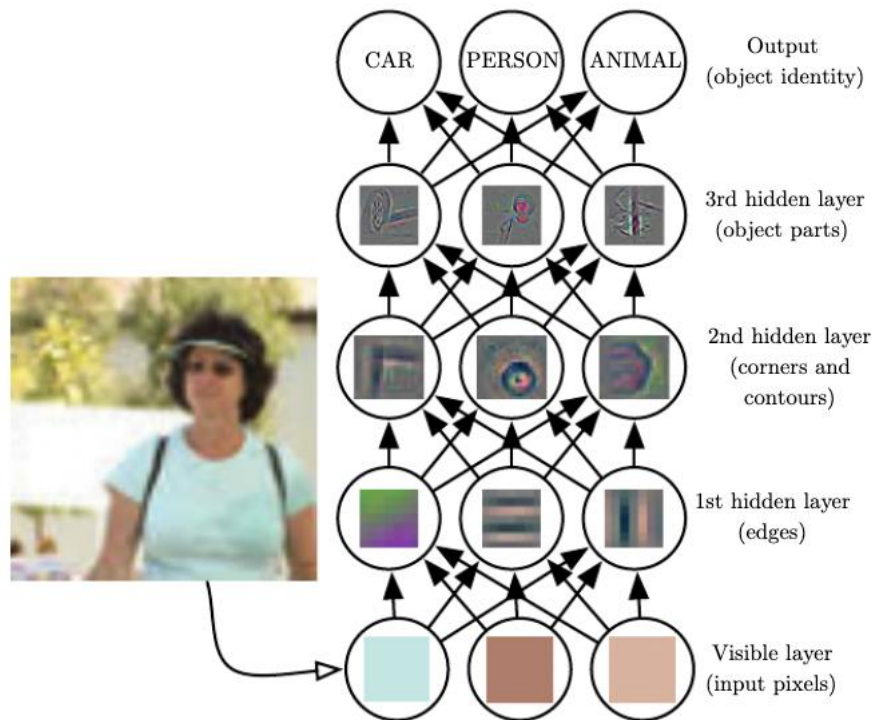


Figure 15: Illustration of a deep learning model, figure copied from [22].

4.1.2 CNN – Convolutional Neural Networks

Images are stored digitally as matrices of pixel values. These pixel values are separated into three channels for – red, green, and blue (RGB). The RGB channels are stored on top of each other, forming three two dimensional matrices to form a complete image. Now, if assuming the usage of feed forward neural network for classifying such an image of scale 28x28 pixels, it would require an input layer of 2352 neurons. If the image was of size 1920x1080 it would require more than 6 million neurons just in the first layer. Adding some hidden layers and output layers to this network, and suddenly there is 100s of millions of weights in the network to compute. As expected, this is way to computational heavy for a computer. Also, a traditional ANN is location sensitive, meaning that it might not be able to detect something that has been moved with reference to the training data. For example, if the network is trained to detect faces in an image, but the training data only contains faces located in the center of those images. That might result in a network not able to detect faces located left or right in that image. Because of these limitations, there has been developed neural network architectures specifically designed for image processing.

CNN uses the mathematical function of convolutions to iterate over an image and detect features. Convolution is a mathematical operation taken on two functions to generate a third function. In this case, a window of size $S \times S$ moves across an image, calculating values to

create a new third matrix. The sliding window is called a filter or kernel. A visual representation of a kernel, sliding window and the mathematical operation of a “top edge” filter is provided by deeplizard.com [23], an interactive tool for visualizing how convolutions work on the MNIST handwritten digit dataset in Figure 16. The step size of the sliding window is called stride. The math behind convolutions is not that interesting for this project, just note that convolution operation on an image is sort of taking the dot product of the kernel matrix on the pixels in the sliding window on the original image. Although this is not entirely correct as each stride takes an elementwise multiplication of each element in the two matrices and sums them. As an example, the filter provided in Figure 16 would output strong positive numbers when the top row of the filter is filled with zeros, and the middle one is filled with ones, and last one is not important. This indicates that the filter is a top edge filter. That means that a strong negative number indicates a bottom edge. The top edges are illustrated using deep red colors, and the bottom edges are illustrated using deep blue colors. If the filter is transposed it would become a left edge detector filter. See example shown in equation (1).

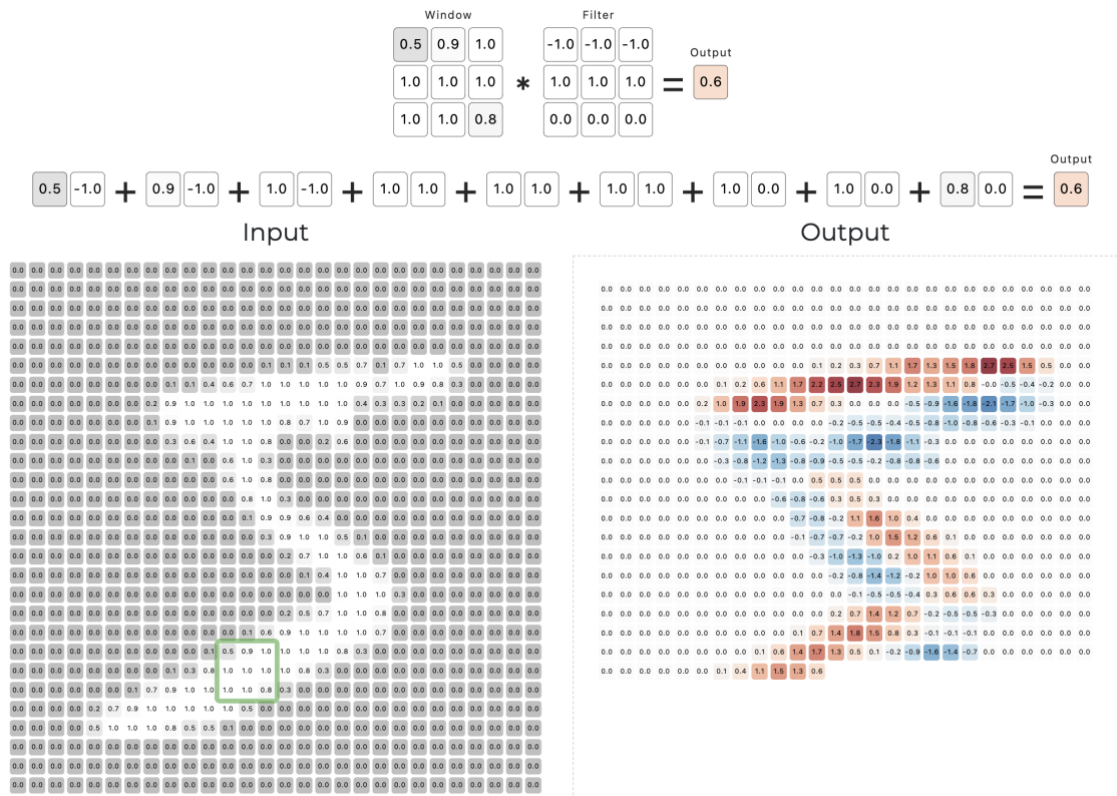


Figure 16: Deeplizard.com visual representation of convolutions on the MNIST handwritten digits dataset. Performing a convolutional operation using a top edge detection filter/kernel on a handwritten number 5. A green box indicating the 3x3 filter is shown on the input. The positive top edge is visualized in deep red color, and the corresponding negative bottom edge is shown in deep blue color on the output image/matrix. There is only one channel of illumination normalized from 0 (dark) to 1 (bright) pixels in this input image/matrix [23].

$$\begin{bmatrix} 0 & 0 & 0 \\ 1.0 & 1.0 & 1.0 \\ 0.2 & 0.3 & 0.2 \end{bmatrix} \cdot \begin{bmatrix} -1 & -1 & -1 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

$$= 0 * (-1) + 0 * (-1) + 0 * (-1) + 1 * 1 + 1 * 1 + 1 * 1 + 0 * 0.2 + 0 * 0.3 + 0 * 0.2$$

$$= 3$$

(1)

As expected, running a convolution operation would result in some down sampling of the image, as a 3x3 kernel would calculate the elementwise sum of the matrix in a 9 sized grid, thus removing one pixel on the output edges when striding on the edge of the input image, see Figure 17. A 28x28 pixel image with a kernel of 3x3 would result in a 26x26 pixel feature map. This down sampling could cause loss of information if edge pixels are important, or the kernel size is large relative to the image. Padding the image with black pixels would solve this problem.



Figure 17: Example of how a 3x3 stride would down sample an image by removing the edge and top pixels in top left corner of the image.

Since all kernels/filters are being applied to all parts of the image, the features are not tied to a specific location, thus making CNN a position invariant feature detector. Both the stride and the kernel size are hyperparameters that needs to be specified. It is worth mentioning that there are many different kernels designed for various detection of features that will not be discussed further in this project. The main idea is to get an overview of what is CNN's and how it differs from FANN.

A fully connected CNN (FCCNN) often consist of many parts and layers. An example of a traditionally structured CNN is taken from the original paper by Keiron O'Shea and Ryan Nash, published in 2015 [24], shown in Figure 18. The network architecture consists of multiple convolution (conv) layers, activation functions and pooling layers before the data is fed to a dense fully connected neural network for prediction. The conv and pooling layers are known as feature extraction layers. Each conv layer gives a feature map output which is fed to the next layer. More on activation functions and pooling layers in chapter 4.1.2.1 and 4.1.2.2.

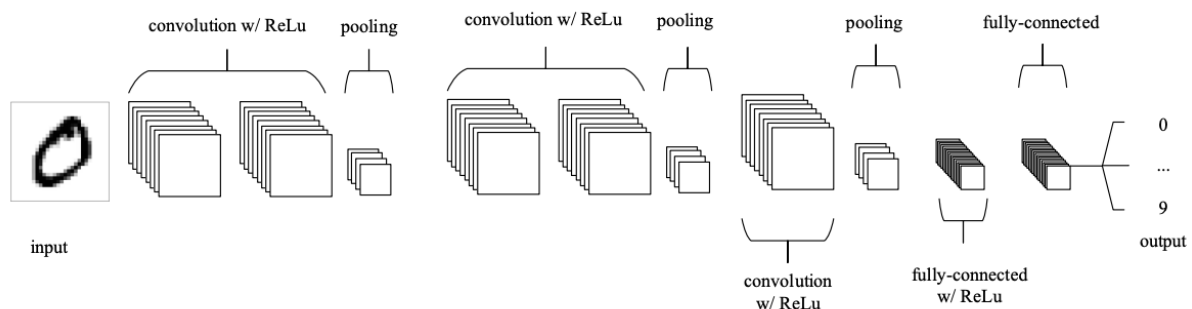


Figure 18: Common CNN architecture with conv layers stacked between ReLu activation functions and pooling layers before going to multiple layers of fully-connected feed forward structured neural networks with activation functions. Taken an input of handwritten digit 0 from the MNIST dataset and providing a prediction on the output layer of 0 to 9.

The first layer of convolution often detects basic features such as edges and corners, and as the data progresses through the network, it detects more and more complex features by

leveraging previously obtained feature maps to build more complex object features. This is known as abstraction and is visually presented in Figure 15. The end goal of the convolutional and pooling layers is to obtain low- to high-level features with as low spatial resolution as possible to reduce computational stress. The last fully connected layers will do the actual classification based on the high resolution down sampled feature maps from the conv layers. It is the mathematical operations of convolutions that makes these types of networks highly depended on parallel computation power provided by hardware such as GPUs, but also makes this network outperform traditional ANN.

It is important to remember that this is a highly generalized and simplified explanation of CNN, not taking details such as hyperparameters, tuning and the fully connected layers into account. The usage of filters in CNN makes CNN position invariant, but it is worth noting that the CNN does not take care of object scaling and rotation by itself. This can be achieved through training data with different scale or rotation, or by data augmentation during training.

A simple illustration showing the entire convolutional process provided by CodeBasics from his Deep Learning Tutorial series on YouTube [25] is shown in Figure 19. This shows an illustration of how pixel values of the number 9 are calculated with a “loopy pattern” filter kernel, generating a feature map. An activation function called ReLU is applied to the feature map, only outputting values that are above 0. Then the max pooling layer of size 2x2 only extracts the max values down samples the feature map while keeping the identifying information. When the image is shifted as shown in Figure 20, it is still able to identify the crucial information of this feature.

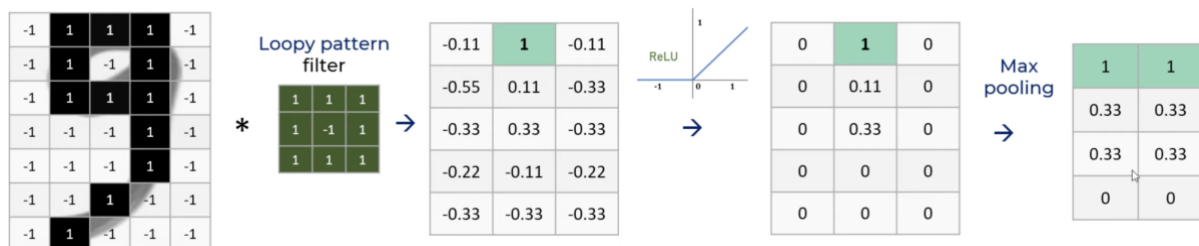


Figure 19: Original illustration from CodeBasics, Deep Learning Tutorial. Input image, filter/kernel, activation function and pooling layer [25].

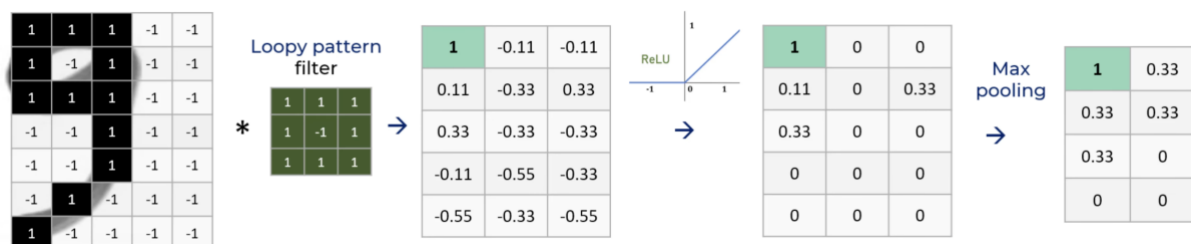


Figure 20: Shifted illustration from CodeBasics, Deep Learning Tutorial. Input image, filter/kernel, activation function and pooling layer [25].

4.1.2.1 Pooling layer

Pooling layers are used to reduce the spatial dimension of the feature map while still obtaining information that uniquely identify features. It is basically a down-sampled matrix containing only the key data of a feature map that the computer needs to uniquely identify a feature. There are different types of pooling such as average pooling and max pooling. The

average pooling calculates the average of the $S \times S$ sized filter and outputs this as one value in a new down sampled feature map as shown in example Figure 21. Max pooling is often more used, and it works the same way as average pooling except that it only takes the max value from the $S \times S$ filter grid and returns as output as shown in example Figure 22. These examples both have a stride of 1. A stride of 2 would result in an output feature map matrix of 2×2 .

0	1	1	0
0	0.3	0.2	0
0	0.1	0.1	0
0	0	0	0

0.325	0.625	0.3
0.1	0.175	0.075

Figure 21: Average pooling

0	1	1	0
0	0.3	0.2	0
0	0.1	0.1	0
0	0	0	0

1	1	1
0.3	0.3	0.2

Figure 22: Max pooling on 4×4 resulting in down sampled 3×3 .

4.1.2.2 ReLU Activation Function

One of the most commonly used activation functions in deep learning [26]. ReLU or Rectified Linear Unit was first mentioned as an analog threshold element in a feature extractor network in a paper published in 1969 by Fukushima Kunihiko [27] and later by the same person in 1975 [28]. Though he did not call the function ReLU, this was the first mentioned of such a function. 35 years later, a paper published by Vinod Nair and Geoffrey E. Hinton suggested ReLU to improve RBM (Restricted Boltzman Machine) [29]. This is the citation that is most often referenced when talking about the beginning of ReLU and the first usage of ReLU for optimization. RBM is, in short terms:

“A ANN with two layers (visible and hidden layer), an algorithm useful for reduction, classification, regression, collaborative filtering, feature learning and topic modelling” – Chris V. Nicholson [30].

ReLU was quickly adapted into the deep learning domain as it is highly computational effective, only activating neurons that are above zero while still introducing non-linearity to

the network. A graphical representation of the ReLU function is shown in Figure 23. In addition to being computational effective and simple, it encourages sparsity in the network by only activating a portion of the neurons at any given time [31]. This can help with reducing overfitting and improve generalization. Sparsity means to only have a small fraction of the neurons and weights active at a given time.

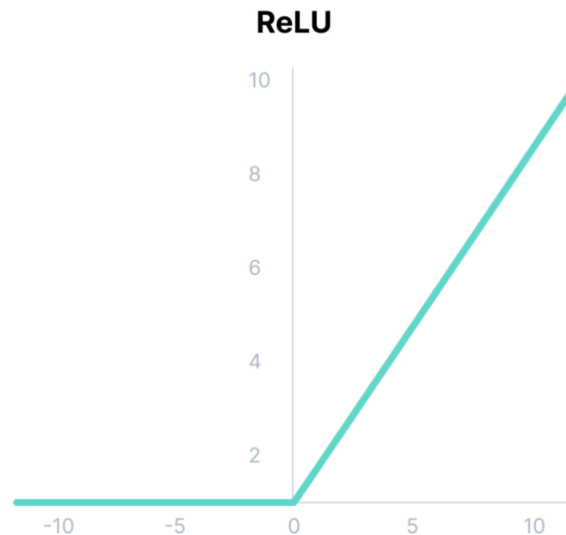


Figure 23: ReLU function [32].

It is worth noting that even though ReLU was commonly known as the most used activation function per 2021, better hardware might give the possibility for more complex activation functions. And more complex problem domains might need more complex activation functions. As an example, the YOLOv7 network swapped the ReLU activation function for a SiLU activation function, which might indicate that a more complex activation function gives better / other results. Of course, this is highly dependent on the whole network architecture and all its components in relations to each other. Anyways, this is just the authors opinion.

4.2 Image classification

Image classification is the task of predicting a class of an object within an image to label that image with the objects class [33], e.g., a picture of a cat, would be classified as a cat. Image classification does not give any information about the object's location in the image. There are effectively two types of image classification subcategories: Single-label classification and multi-label classification. These two will be discussed in more details in the following subchapters along with some theory around the ResNet architecture and how it was a game changer for the deep learning community is a necessity. This theory will build upon the CNN theory, as well as mentioned some key building blocks within DCNN that has been adopted by many and is used in most networks today.

4.2.1 ResNet – Residual Network

ResNet is a deep neural network architecture that was developed by researchers at Microsoft Research Asia, He Kaiming et al. [34] in 2015. The main idea for this research was to solve problems that occurred with previously developed CNN models when they started to exceed

a certain depth (number of convolution layers). In 2012, Alex Krizhevsky et al. [35] laid the foundation for Deep Convolutional Neural Networks (DCNN) with their invention of the AlexNet, trained on the LSVRC-2010 ImageNet training set. This was the first time that a DCNN performed better on the ImageNet dataset than traditional feature engineered ML methods. AlexNet only consisted of 8 neural network layers: 5 convolutional layers and 3 fully connected layers. The general idea was that more layers would be able to learn more features thus perform better on data. Kaiming et al. [34] proved that this was not actually the case and at some point the training result and test results would actually get worse, as illustrated in Figure 24. Since the training error increases with increased number of layers, it is a clear indication that this problem is not due to overfitting and must be caused by another issue.

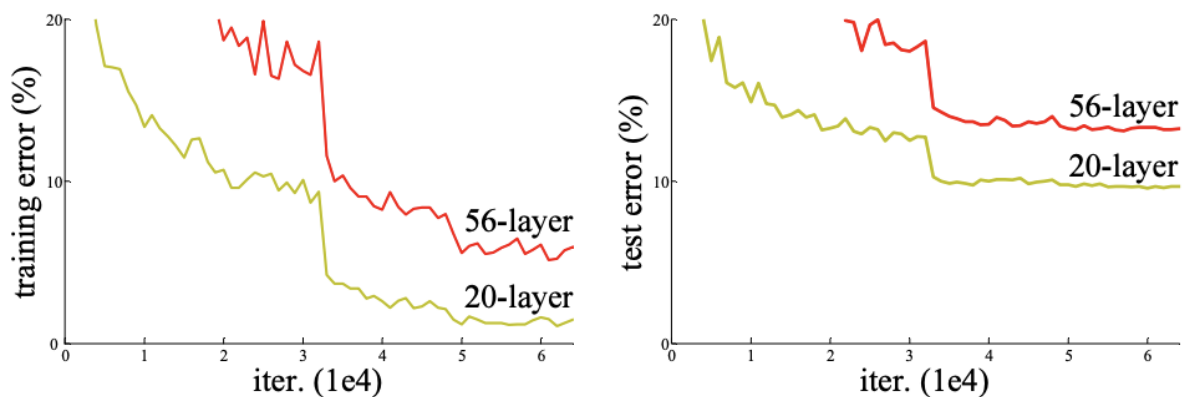


Figure 24: Training error (left), Test error (right) for stacking more layers in a “traditional” CNN [34].

Kaiming et al. discusses many possible reasons why this problem occurs but the suggested solution ended up being a method that is used in most neural network architectures since called Residual block and skip connections. To understand what the Residual building block does and how it works, some theory around convolution neural networks architecture is needed.

The philosophy behind DCNN is that the convolutional layers initially maintain a high resolution of the image while increasing the channel size. As more filters are added, the image resolution is downscaled. The reasoning behind this architecture is that low-level features, such as edges, are essential for image classification and their precise location is significant. However, as the network learns more abstract features at higher layers, the exact localization of these features becomes less important. A visual representation of the downscaling and added filters are shown in Figure 25. The high-level features are responsible for connecting the visual representations features and the low-level features, thus giving localization to objects.

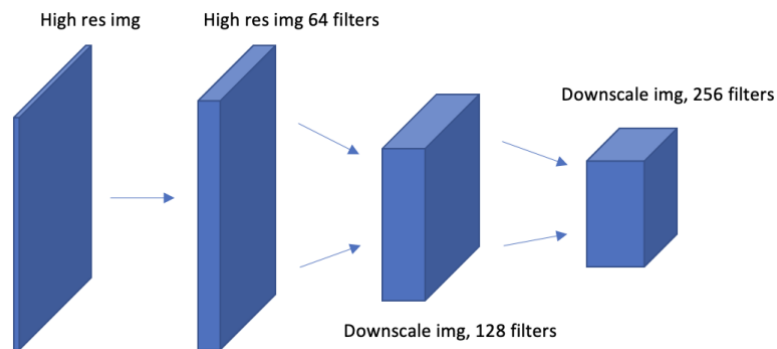


Figure 25: Visual presentation of CNN architecture.

Ok, so if this simple shallow architecture that is presented is trained to classify images with some accuracy, it is natural to believe that a deeper architecture consisting of the shallow plus added additional layers would at least be able to get the same accuracy. The deeper architecture only has to copy the initially trained shallow layers and learn the identity functions of the deeper layers. A visual simplified presentation of deeper architecture is shown in Figure 26 and Figure 27. Now, Kaiming et al. argues that the reason the deeper layers do not learn these identity functions is due to the initialization of weights, which normally happens towards or around zero. What they propose is therefore the Residual building block which is a residual connection that helps initializing this identity function.

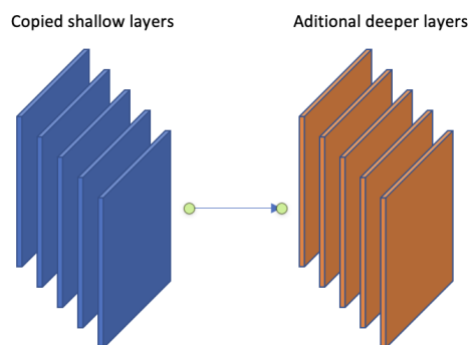


Figure 26: Additional deeper layers in orange, stacked as an additional set of layers.

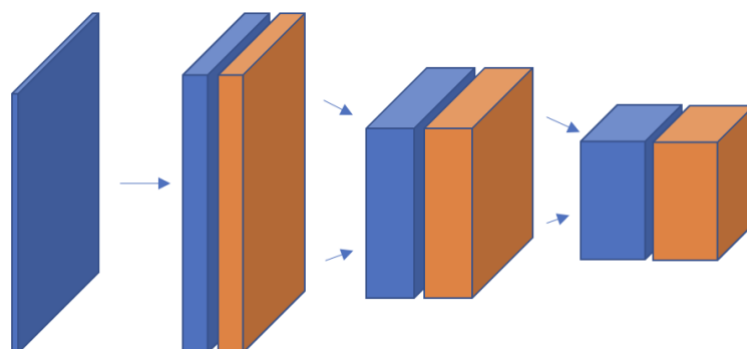


Figure 27: Additional layers in orange stacked as they would in the architecture.

Simply put, the Residual block or residual connection initializes the weights so that instead of new additional layers having to learn transforming x into x (which is the identity function)

from zero, it will transfer x directly from previous layer and learn what needs to be adjusted. A visualization of this is presented Figure 28 and the actual Residual block representation from the original paper is shown in Figure 29.

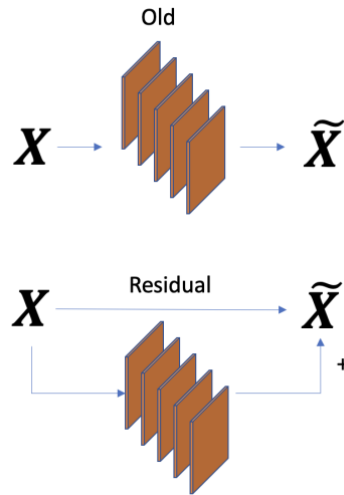


Figure 28: Visual representation of how the Residual layer uses skip connections to transfer the identity function to the next layer, thus initializing the weights closer to the function that is going to be learned. The additional layers just have to add the additional corrections to the \tilde{x} .

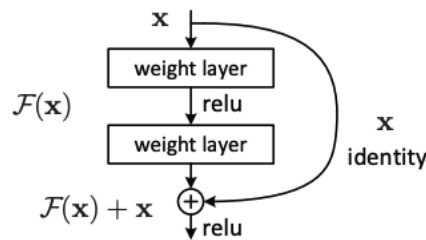


Figure 29: The Residual learning building block [34].

The mathematical function of the Residual learning block and skip connection is shown in equation (2), where for the above example in Figure 29, $F(x, \{W_i\})$ would be $W_2\sigma(W_1x)$ where σ denotes the ReLU function [34].

$$y = F(x, \{W_i\}) + x \tag{2}$$

A problem now would be the rescaling and increasing of dimensions between layers. This is solved by 1) adding zero padding, or 2) using 1×1 convolutions in the shortcut connection to upscale the 64 filters to 128 filters shown in dotted lines in Figure 30. Method 2) is the one that is adopted and used today.

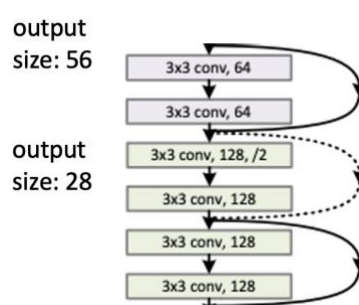


Figure 30: A snippet of a ResNet 34-layer architecture from the original paper [34]. Solid lines indicate skip connections between layers of same dimensions, dotted lines indicate shortcuts with increasing dimension.

The shortcut connections between layers of different dimensions have to be denoted in a different way with a linear projection W_s of x shown in equation (3) [34].

$$y = F(x, \{W_i\}) + W_s x \quad (3)$$

The result of this Residual block was a network that was able to scale to any depth without worsening the accuracy. And in the original paper they show this by training different size networks until the network starts to overfit to the training data. This is why the ResNet architecture has become highly popular, and there exist a lot of networks utilizing this as backbone architecture network. A comparison of a plain architecture DCNN and ResNet DCNN was presented in the same paper and is included here in Figure 31, clearly indicating that deeper network structures need the Residual block to function as intended.

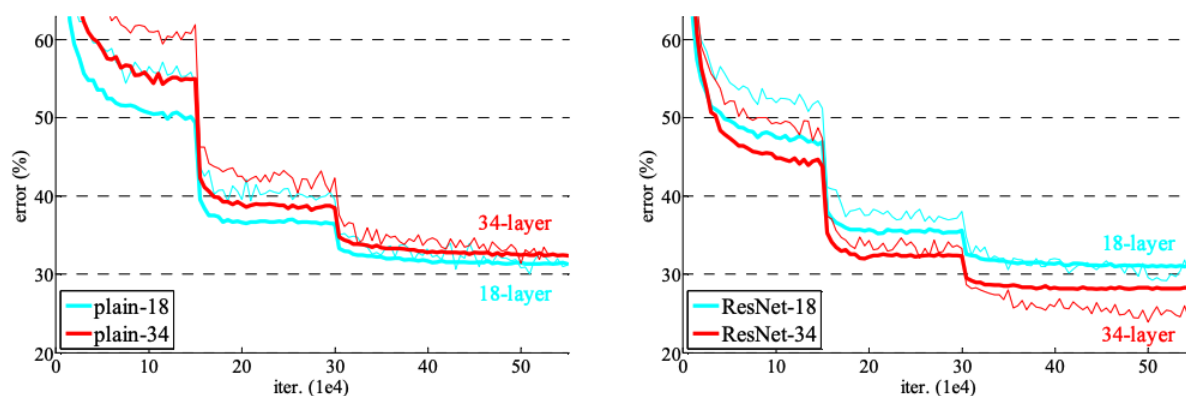


Figure 31: Comparison of a plain (before ResNet) DCNN architecture and a ResNet DCNN architecture, both of size 18 and 34. Thick lines are validation error, thin line is training error. As shown, the Residual block only shows real performance increase when the depth of the network increases. The 18-layer architecture shows almost the same error rate, but there is a massive difference in the 34-layered architecture.

It is recommended to read the original paper or watch a YouTube video called “Deep Residual Learning for Image Recognition (Paper Explained)” by Yannic Kilcher [36] which the illustrations shown in Figure 26 and Figure 28 in this report is highly inspired by. The original paper also includes a Bottleneck block to help with computations of large dimensional layers, shown in Figure 32. This Bottleneck block downscales a high dimensional layer using a 1x1 convolution, then do the computational feature extraction on the downscaled layer and then upscale to the original dimension. This method is also adapted in most modern network architectures.

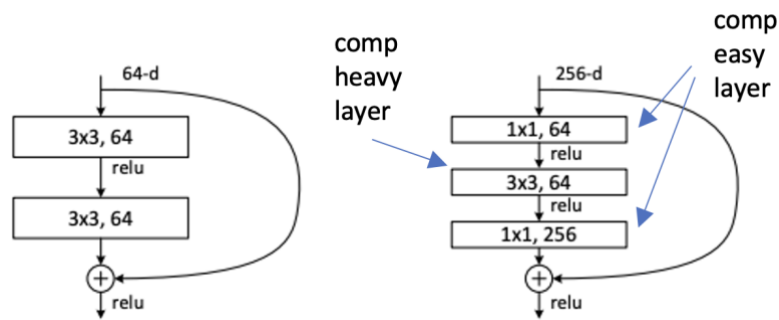


Figure 32: Left shows two 64 dimensions traditional layers. Right shows a 256 dimensions Bottleneck block layer, projected down to 64 dimensions by a less computational heavy 1x1 conv, then feature extracted, and projected back to the original 256 dimensions. The 1x1 conv is 9 times less computationally heavy than the 3x3 layer. So, one of these Bottleneck is a little more computationally heavy than one of the left 3x3 64 dimensions layers.

The reasons for going into details on the ResNet architecture is not only to show why this was a gamechanger for the deep learning community, but also because the first part of this project will utilize the ResNet50 architecture for image classification.

4.2.2 What is single-label classification?

More commonly known as normal image classification where the goal is to give one label (also called prediction) to the whole image. This could be an image containing more than one object, but the goal is to identify and predict a main object that dominates that image. An example could be the picture of a car which normally also contains a road, maybe some signs and buildings. These other objects can be denoted as noise in this context. The main object in this picture would be a car, and the image would be labeled as such. The more noise an image contains the harder the image is to classify. The classification model itself can have multiple classes such as car, bus, train, pedestrian. But the goal is to give one image, one prediction.

4.2.3 What is multi-label classification?

Is the task of classifying multiple object categories within one image. Take the example of the car again, it would be natural in this context to classify both the car, road, signs, and buildings. However, it is not an object detection or segmentation task, so the location of each object is not predicted and not of interest. This method is also specifically handy because it gives the opportunity to not classify any object if the model cannot identify any objects in the image. The previously mentioned single-label classification would try to classify the image into some category regardless of whether the object is present in the image or not. This could be solved by adding a separate category for training “unknown” objects but would require even more training data. Multi-label classification is therefore a good alternative, and a method that will be implemented as the first part of this project.

4.2.4 Data preparations

Collecting and preparing data for image classification is relatively simple in both single-label and multi-label classification. The most important thing is to have enough training and validation data available, and that the training data represents the real-world data in a good

way. How to collect data for this project, and scripts to optimize the process is explained in chapter 3.6.

4.2.4.1 For single-label classification

When gathering data for traditional image classification, it is generally preferable to have data that clearly depicts the main object with minimal noise. The presence of additional variations and complexities in the image can make it difficult for the model to learn the underlying features of the labeled objects. However, if the real-world application contains a lot of background noise, it is important to introduce such examples in the training and validation data. This will increase the networks robustness and generalization. More training data is therefore always better.

There are different ways of structuring this training and validation data for different types of models. The FastAI library, used in this project needs the data to be separated into folders which contain the class name as shown in Figure 13. Full path to the folder containing the class subfolders needs to be provided to the FastAI functions for construction of data blocks as shown in Table 16.

4.2.4.2 For multi-label classification

If the applications are the same, the training data gathered for single-label classification can serve as an excellent starting point for multi-label classification. However, in addition to classes with one “main” object presented, multi-label classification also requires classes with a combination of objects with multiple class labels as shown in Figure 14. The rule for noise (complexity and variations) also applies here, so a large dataset is important. Based on experience, if a multi class labeled image contains i.e., a “valve” and a “tag”, it is important that there also exist single class labeled images that only contain a “valve” and “tag”. This is important for the model to better learn what separates these objects. Examples of this is shown in Table 20 where all multi-label classes also have a single-label class representing each of the individual objects.

Now, when structuring this data, it is no longer possible to only have the data separated into class folders as one image could contain multiple classes. Instead, all files need to be combined into a common folder, and a specification file needs to specify image name, class labels and if it is training or validation data. Luckily there are ways to optimize this process and is demonstrated in chapter 4.2.4.2.1.

4.2.4.2.1 Creating object specification file for use in multi-label classification.

When a sufficient amount of training data is collected and put into separate class folders for labeling, a Python program is created for renaming all files in all folders with the respective parent folder name and an iterative number, see code snippet in Table 6.

Table 6: Renaming all image files in class folders to respective class name (labeling images).

```

source_folder =
    Path("/home/user/git/classify_multi_obj/Classification_small_
        multiobject")

# loop through each file in the source folder
i = 0
j = 0
for folders in source_folder.ls():
    i = 0
    pref = str(folders).removeprefix(str(source_folder)+"/")
    for filename in os.listdir(folders):
        extension = os.path.splitext(filename)
        if extension[1] != ".Identifier":
            j +=1
            newname = pref+" " +str(i)+extension[1]
            source_file = os.path.join(folders, filename)
            my_dest = os.path.join(folders, newname)
            os.rename(source_file, my_dest)
            i += 1
        elif extension[1] == ".Identifier":
            source_file = os.path.join(folders, filename)
            os.remove(source_file)
print(j)

```

Next, a Python program for generating the specification CSV file required for the multi-label classification is created. The CSV file will contain three columns separated by comma. The columns will contain the image name called “fname” the respective “label” of that image (which is the folder name where the image is located), and a randomized “is_valid” column. The “is_valid” column will have a 20% true and 80% false for each object class period. The source code for creating the CSV file with given specifications is shown in Table 7. An example specification CSV file in table view can be seen in Table 9.

Table 7: Script for creating the items.csv file with 20% validation and 80% training data.

```

folder_path =
    Path("/home/user/git/classify_multi_obj/Classification_small_
        multiobject")

csv_file = 'items.csv'

# create a new csv file
with open(csv_file, 'w', newline='') as f:
    writer = csv.writer(f)
    writer.writerow(['fname', 'labels', 'is_valid'])

    for folders in folder_path.ls():
        # loop through each item in the folder
        for item in os.listdir(folders):
            # get the item's name and folder name
            fname = item
            labels = str(folders).removeprefix(str(folder_path)+"/")
            if random.random() < 0.2:
                is_valid = True
            else:
                is_valid = False
            writer.writerow([fname, labels, is_valid])

```

When the specification CSV file is created, all files need to be grouped together into a training folder, i.e., the class folders are not useful anymore (only applicable for the multi-label classification). This is achieved by creating another Python script shown in Table 8 that copies all files within class folders into one training folder.

Table 8: Script for copying files from multiple folders into one folder.

```

source_folder =
    Path("/home/user/git/classify_multi_obj/Classification_small_
        multiobject")
dest_folder = '/home/user/git/classify_multi_obj/train'

# loop through each file in the source folder
for folders in source_folder.ls():
    for filename in os.listdir(folders):
        source_file = os.path.join(folders, filename)
        dest_file = os.path.join(dest_folder, filename)
        shutil.copy(source_file, dest_file)

```

The specification file is now the master file, giving instructions on what class each object image belongs to and if it should be used in training or in the validation of the models. This means that the initial folder structure was only useful for the single-label classification

approach, and further works as a simpler way to structure all the snipped data and verify each object. When training a multi-label image classification model, all data needs to be specified in this object specification file. This is also the case for object detection where each image file will have a specification file (annotation file) giving localization and sizes of each object.

Table 9: Example of specification file for multi-label classification in csv format, each column separated by comma.

fname	label	is_valid
object1.png	pump	false
object2.png	valve	false
object3.png	valve	true

4.2.4.3 Augmentation

If the available training data is limited, augmenting the data would serve as a technique to distort data, creating more variety in the dataset. The data is not actually duplicated, distorted, and stored as part of the training set, but rather each batch of training and validation gets a manipulation added to it so that the model sees the same data differently for each batch. This helps in improving the model to be more robust. The image manipulations could be flipping, rotating, angling/phasing, blurring, zooming, or cropping. There are many ways of manipulating an image to look slightly different while still keeping its main structure, and people come up with new methods all the time. What type and amount of augmentation can be set by the system designer.

4.2.4.4 Resizing or pre-sizing

The network input image size is decided by the network architecture. ResNet50 needs all input images in the scale of 224x224, but there are options on how to perform this resizing. The FastAI library provides all data preparations steps in the data loader pipeline, with extra parameters for resizing and data augmentation. There are effectively five different resize techniques often used in image classification:

1. Resizing to a fixed size: All images are resized to a fixed size, such as 224x224. This is a common approach used in many pre-trained models like VGG and ResNet. Pros: easy to implement. Cons: may result in distortion or loss of information.
2. Center cropping: Input image is cropped to a square in the center and then resized to the desired size. This approach is useful when the object of interest is centered in the image. Use with care, as it may result in loss of information.
3. Random cropping: Multiple random crops of the input image are taken, and each crop is resized to the desired size. This approach helps to capture different views of the object and reduces overfitting. This technique is used in many pre-trained models like ResNet.

4. Aspect ratio preserving resizing: The input image is resized while preserving its aspect ratio. This approach prevent distortion but may result in padding or loss of information. To prevent loss of information, the padding alternative is a good option.
5. Scale augmentation: In this technique, the input image is randomly scaled up or down before resizing. This approach helps to capture different scales of the object and increases the model's robustness to scale variation.

For this project, the input data will be of varying size, and all information in the input data is important. Technique number 4 with zero padding will most likely be used to prevent unnatural distortions to images. Zero padding is adding black pixels to the edges to keep the images aspect ratio. This zero padding adds extra computational load, but this is insignificant compared to the loss of information that could occur if not used.

4.2.5 Training and validating

Training and validating a model are easy and will generally give good result if the prementioned steps are done with care. The functions to perform these code snippets are already built and available from libraries such as PyTorch, FastAI, TensorFlow, Ultralytics and so on. How to use these will be shown in the chapter 5 Results. Although training and validation code can be executed relatively easily, the underlying theory of these processes is not straightforward. In this subchapter, the theory behind the ResNet50 network's training, validation, and adjustment using loss functions and backpropagation will be explained.

4.2.5.1 Loss functions

In general, loss functions, often referred to as cost functions, objective functions or error functions calculates how well a model can predict a desired output for a given input. It is worth noting that even though these four names are used interchangeably, the loss and error function is more common to use when talking difference between predicted and true output. Loss function being mentioned in the context of optimization, and error function in terms of evaluating performance. Cost and objective function is more correct to mention in context of whatever function a model is trying to optimize. The two loss functions worth mentioning for this project is Cross-Entropy (CE) loss and Binary Cross-Entropy (BCE) loss.

4.2.5.1.1 Cross-entropy loss (CE)

The Cross-Entropy (CE) loss is particularly valuable in applications where the model's prediction must be classified into N different classes, but an image can only belong to a single class, as is the case in the single-label image classification part of this project [37] [38]. Simply put, the CE loss computes the summation of the true probability ($P^*(i)$) multiplied with the log predicted probability ($\log P(i)$) over all classes in the distribution, equation (4).

$$H(P^*|P) = - \sum_i^N P^*(i) * \log P(i) \tag{4}$$

The predicted probability outputted from the last layer in the neural net is normalized by the SoftMax function to become a prediction between 0 and 1. SoftMax require the predicted output layer to sum up to a total of 1, so if one class has predicted probability of 0.9, all the

other N-1 classes would sum up to the final 0.1. Then take the negative log predicted probability of the actual inputted class and multiplied with the true probability (1 on the inputted class, 0 on the other). Do this for each N number of classes in the batch and sum it to get the total error of the network. Backpropagation is further used to adjust weights (FNN) or kernels (CNN) and hopefully improve the network accuracy. The reason for using CE instead of other loss functions such as Sum of Square residuals (SSR) is because CE exponentially increases the loss as the prediction gets worse, due to the log part of the equation. So, if a model predicts a hard wrong, the loss gets exponentially higher, resulting in a large incentive in the backpropagation to step towards a better prediction. This also means that a small prediction error, results in smaller incentive in the backpropagation of correcting the prediction. This has to do with the derivative (slope) of the tangent line of the CE loss used in gradient descent calculation of the step size in backpropagations.

4.2.5.1.2 Binary cross-entropy loss (BCE)

BCE is typically used in models performing binary classification problem i.e., when there are only two different classes. However, it can be used in the multi-label classification by utilizing elementwise BCE operation on each of the output nodes to predict whether a class is present or not in the input [39]. To use BCE in multi-label classification, the output layer needs one node for each of the label classes in the training data, and the SoftMax needs to be replaced by a Sigmoid function. During training, the BCE loss function calculates the difference between the predicted probabilities and the true labels in a probability range from 0 to 1 using the Sigmoid function. As the input of true probabilities is now a vector that could consist of multiple true labels, the output predicted probabilities is also a vector. The total loss is given by the sum of all output nodes calculated BCE loss. The mathematical expression is shown in (5), where M is number of rows in the probability vector, and N is number of classes [40].

$$H(P^*|P) = -\frac{1}{M} \sum_j^M \sum_i^N P^*(i,j) * \log P(i,j) \quad (5)$$

4.2.5.2 Backpropagation

When the loss is calculated and the gradient of the loss obtained, it is time to backpropagate through the network updating weights in the fully connected network and kernel values in the CNN. This is a mathematical operation of calculating the local gradient of each layer and calculating the updated weights of the kernel with gradient loss from previous layer multiplied with learning rate. In fully connected networks, the weights are updated using the gradient of the loss with respect to the weights. In CNNs, the kernel values are updated using the gradient of the loss with respect to the kernel values. Multiplying by learning rate is used to control the size of the weight updates. Backpropagation is an iterative process, and it is repeated for each mini batch of training data until the network converges to a set of weights that minimizes the loss function. This is how a deep learning neural network learns. A visual representation of a single layer backpropagation is shown in Figure 33 with additional

mathematical expression of how the updated filter/kernel values $x_{updated}$ is calculated in equation (6), α denotes the learning rate.

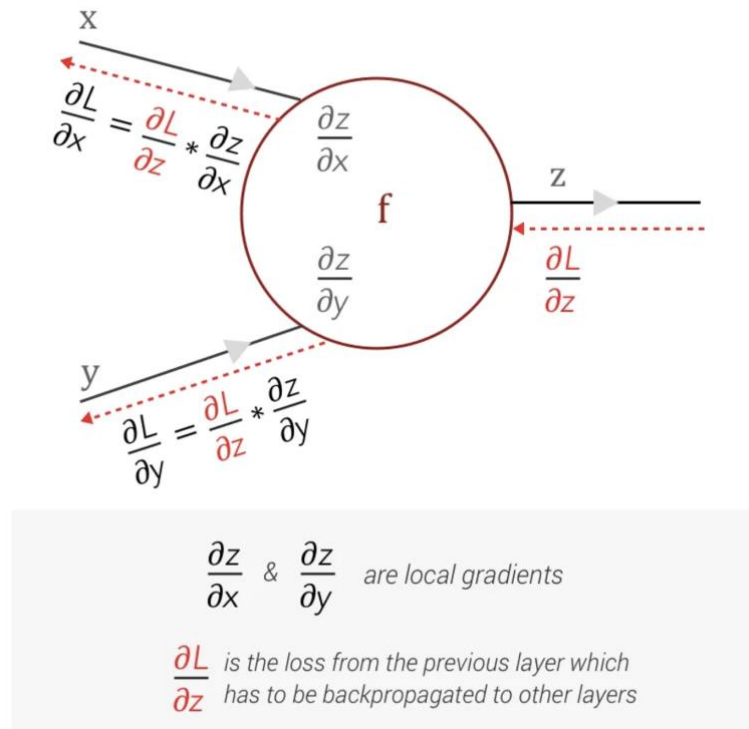


Figure 33: Single layer backpropagation example. The function f must be thought of as a convolution for CNN. This image is borrowed from a blog post by Pavithra Solai on medium, please see citation [41].

$$x_{updated} = x - \alpha \frac{\partial L}{\partial x} \quad (6)$$

4.3 Object detection

Object detection differs from image classification because of its ability to perform both classification and localization of an object within an image or video [42]. The result from an object detection method would be bounded box regions of objects and a classified label of set objects. The resulted object classes and locations are also available in some sort of text format, thus indicating that the computer reads an image and interprets it.

The first object detection methods can be dated all the way back to the 1970s when Optical Character Recognition (OCR) was introduced. These methods were based on traditional computer vision methods such as edge detection and corner detection. However, the first real object detection algorithm called Viola-Jones was introduced in 2001, and is formally dated as the first real-time object detection algorithm [43]. The Viola-Jones algorithm used a technique called Haar-like features which essentially detect rectangular features in an image. This technique was used in combination with a sliding window algorithm to find multiple features at different positions and at different scales in an image. If a particular set of Haar-like rectangular features matched the characteristic patterns of a human face, it classified a face or multiple faces in an image [44]. In 2005 came the Histogram of Oriented Gradients

(HOG) feature descriptor that focused on the shape of an object [43]. It worked by extracting the gradient and orientation of edges, and was mostly used to detect humans in an image [45]. In 2008 came an extension of the HOG detector called Deformable Part-based Model (DPM). In short terms DPM decomposes an object into separate parts for classification, and by combining them to form the full object [43] [46]. These three methods are noted as key points in object detection history and has been the foundation for and influenced many modern object detector solutions. They are all based on traditional computer vision techniques.

The first Convolutional Neural Network (CNN) were originally introduced in late 1980s as a technique for character/handwritten characters recognition. The first paper on the work that had been performed during the timespan of many years since the late 1980s was published 1998 [47]. In 2012 CNN was re-introduced for neural network based image classification and quickly adopted into the object detection field in 2014 with a method called Region-based CNN (R-CNN). This was the beginning of using deep learning neural networks (DLNN) for classification and object detection, both improving speed, flexibility, and accuracy of object detection. Later on, the R-CNN method was vastly optimized and improved by small but important tweaks, giving the methods of Fast R-CNN and Faster R-CNN [43]. Today, the latter is mostly used in detection method that requires high accuracy.

In 2015, both the You Only Look Once (YOLO) [48] and Single Shot Detection (SSD) [49] methods were proposed. Both these methods differ a lot from the previously mentioned methods in architecture, as it applies only a single neural network to the whole image, thus massively increasing the detection speed compared to R-CNN [43]. These methods are still relevant and have been massively improved since 2015, YOLO being the most publicly known open source used.

4.3.1 DLNN Detection

A deep neural network can learn and extract robust high-level features in images. This introduced an opportunity and a foundation for learning specific object features and classifying images. Then, by using a pretrained CNN on multiple proposed object regions in an image, a deep learning neural network would be able to classify parts of the image and get the location of the classified object, resulting in object detection. This idea gave birth to the Region-based CNN. However, the initial R-CNN had a major drawback with being slow, as it required a lot of computing power to classify a lot of proposed regions in an image. This drawback was improved by the Fast and Faster R-CNN methods. The R-CNN and its enhanced variations, Fast R-CNN and Faster R-CNN are two-stage detection methods that are discussed in greater depth in the following chapter 4.3.1.1.

In order to further enhance speed and reduce computational load, the YOLO algorithm partitions the entire image into multiple grids instead of suggesting a set of region candidates. Each cell in the grid is responsible for detecting the object within its boundaries and provide a confidence score. All predictions are made simultaneously using a single CNN. To further ensure that the predicted bounding boxes matches the real object boxes, an Intersection over Union (IoU) calculation is performed. The algorithm keeps the predictions closest to the real object annotation (also called ground truth box). Both the YOLO and the SSD method is one-stage detection methods. YOLO will be further analyzed and used throughout this project due to its latest release of the algorithm (version 8), released January 2023. The YOLO architecture and YOLOv8 model is further discussed in chapter 4.3.1.2 and 4.3.2.

4.3.1.1 Two-stage detection method R-CNN

R-CNN is a so-called two-stage detection method. It is called that because it uses two primary steps to perform the object detection. The first step is to propose regions of interest, and second step is to extract features from each proposed region using a pre-trained CNN.

The initial R-CNN detection method was proposed by Ross Girshick in 2014 [50]. It used a method called selective search for finding 2000 object region candidates. These region candidates were refined by warping each region to a predefined size and further fed to a pretrained CNN for feature extraction. The feature vector for each region proposal was then passed into Support Vector Machines (SVM) model for classifying object categories and bounding box regressor around the object. The downside of this approach is that it was computationally intensive and time-consuming, as it required the classification of 2000 region candidates. It also used a method called selective search for finding object region candidates. This is a “brute force” or “exhaustive search” method using a sliding window algorithm for grouping correlating pixels and founding regions as shown in Figure 35. As it is a fixed algorithm (not learning) sliding over a window with predefined size and scale, it could lead to inaccurate region proposals. A graphical representation of the R-CNN architecture is shown in Figure 34.

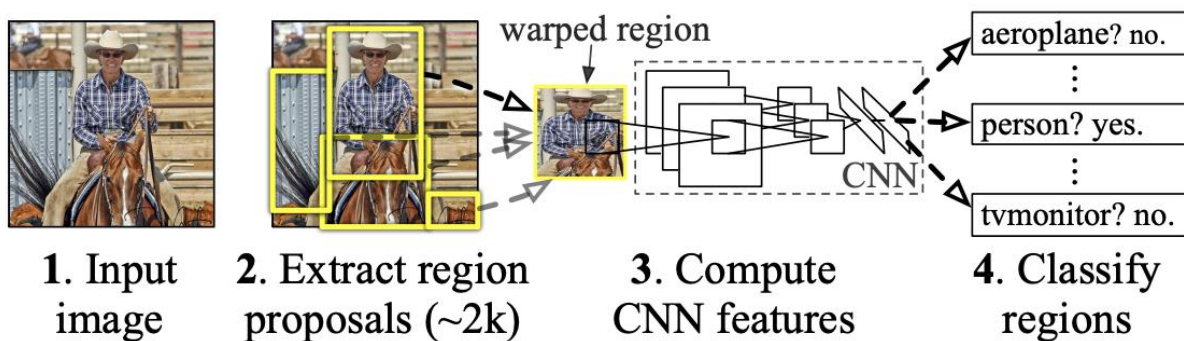


Figure 34: Initially proposed R-CNN architecture, 2014 [50].



Figure 35: Selective search algorithm [51].

Fast R-CNN improved the “classify 2000 region candidates” drawback by instead feeding the entire input image and a set of region proposals to the CNN for generating a convolutional feature map. Fast R-CNN further uses a combination of RoI pooling layer to warp the RoI’s to a fixed size in one single layer. This RoI feature vector is then fed into a fully connected layer where a softmax probability layer is used (instead of SVM) as classifier to predict class and bounding box regression offset for the proposed region [52]. This improves the object detection using Fast R-CNN by ten folds compared to R-CNN. There is still a bottleneck to this approach, and that is the usage of selective search in the region proposal generation algorithm, same as initially proposed R-CNN [53]. This could result in inaccurate region proposals and is a time inefficient method. Fast R-CNN was proposed by the same person as initially proposed the R-CNN method in 2015 [54]. A graphical representation of the Fast R-CNN architecture is shown in Figure 36.

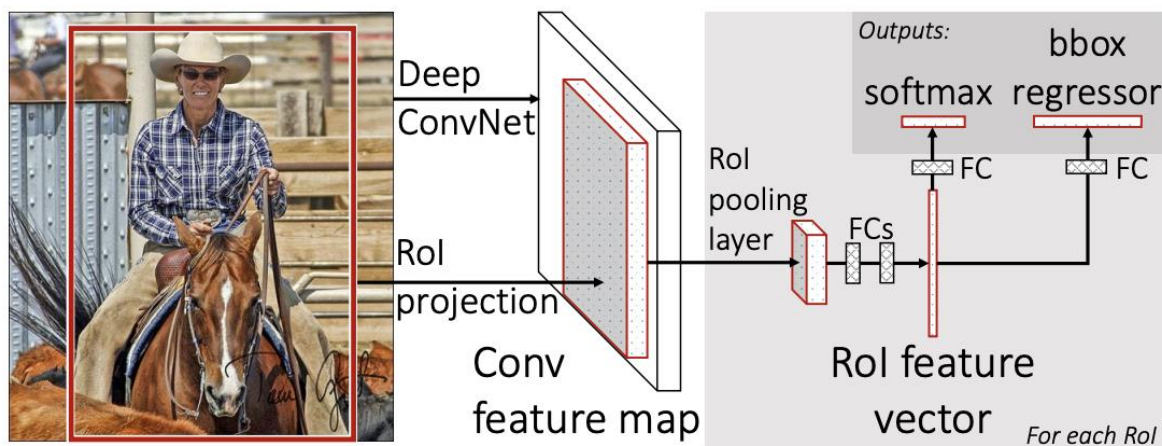


Figure 36: Fast R-CNN architecture [54].

Faster R-CNN was introduced in 2015 by Shaoqing Ren, in cooperation with Kaiming He, Jian Sun and Ross Girshick who proposed both the R-CNN and Fast R-CNN [55]. Faster R-CNN removes the bottleneck of selective search, by instead adding a separate network for predicting region proposals. The separate network is called Region Proposal Network (RPN). RPN is a fully CNN that takes feature map from the first step as input and generates region proposals by sliding an anchor window over it. It then predicts the probability of an object being present in that anchor window and the parameters (x, y, width, and height) of its bounding box. This gives a lot of candidate proposals that are further refined using regression and NMS to obtain an optimal set of candidate objects (graphical representation similarity to the 4th, 5th and 6th image in Figure 35). This network is trained to detect anchors box proposals in an image, thus learning, and drastically increasing network efficiency. It is recommended to read more about this in the original paper [55], if it is of interest. The rest of Faster R-CNN is similar to the previous Fast R-CNN approach using a RoI pooling layer for reshaping the candidate objects and further classify object and find localization. The changes from previous methods, where the RPN is used on the feature map to extract RoI’s is graphically represented in Figure 37.

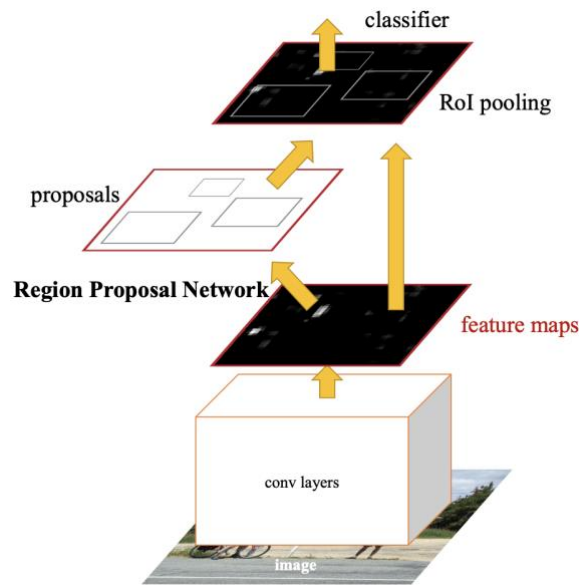


Figure 37: Faster R-CNN architecture [55].

4.3.1.2 One-stage detection method YOLO

A one-stage detection method differs from the previously mentioned two-stage method because it directly predicts the bounding boxes and class probability for all objects in an image in one single forward pass of CNN. The two-stage method such as Faster R-CNN had a separate RPN, YOLO does not. The YOLO algorithm's ability to simultaneously predict the class probability and bounding boxes for each object allows it to be faster than two-stage methods whilst still achieving high accuracy.

YOLO was the first one-stage detector in the deep learning era [43], first time proposed by Redmon Joseph in 2015 [56]. As mentioned, the YOLO algorithm splits an image into an $S \times S$ grid. Each grid cell is responsible for prediction within its own boundaries, so the cell predicts B bounding boxes related confidence scores as well as one C class probability per cell. Each of the bounding boxes consist of 5 predictions - x , y , width, height, and confidence. These 5 predictions are important to remember as they are key in training a custom network at a later step in the project. The confidence score indicates the IoU between the predicted box and the ground truth box. After applying NMS on the grid of predictions, the result should be the bounding boxes with the highest score. An illustration of the four steps taken from the original paper [56] is shown in Figure 38.

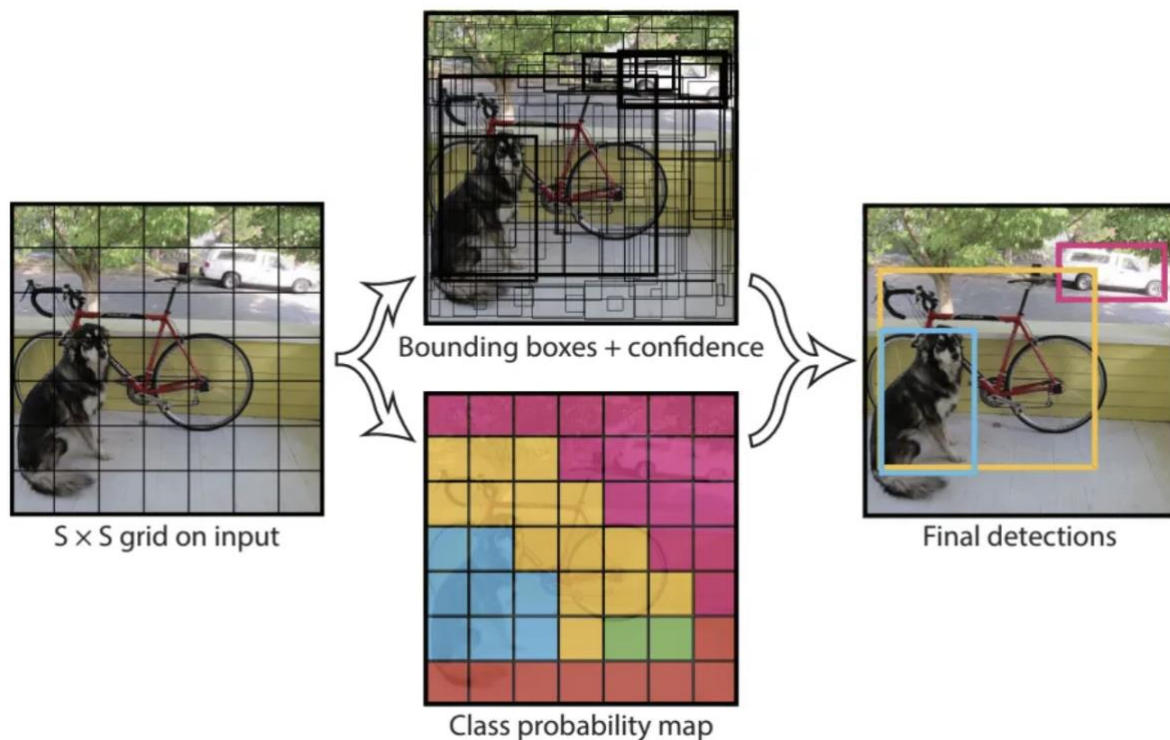


Figure 38: Four steps of grid, B bounding boxes, confidence score and final prediction [56].

4.3.2 YOLOv8

The YOLO object detection system has evolved a lot the last 7-8 years with an exponential speed the last couple of years. The first system YOLOv1 introduced in 2016 only consisted of a single fully CNN. The first version was fast and accurate but suffered from low recall (failing to identify a large portion of positive instances) and localization errors. Since then, seven new version have been introduced each with new and improved techniques for improving accuracy, speed and reducing localization error [57]. Originally the YOLO system was not well suited for detecting tiny objects in images, but that has also drastically changed with each version. In 2020, YOLOv4 was released and introduced a new anchor-free detection head. This means that instead of using the traditional anchor box approach, the method instead directly predicts object location and sizes, which can simplify the model architecture and improve performance. YOLOv8 is built from this technique, adding a lot of features since then.

The latest version YOLOv8, developed by Ultralytics, was released January 10th, 2023 [58]. This method scores significantly better than the 3 previous versions in both speed and mAP^{50-95} when trained and tested on the COCO val2017 dataset, shown in Figure 39. This is still not in the top range between 60-65 mAP^{50-95} where the large scale models perform [59], but YOLO aims to be compact simple models designed to be fast, accurate and easy to use with a large community and support [60].

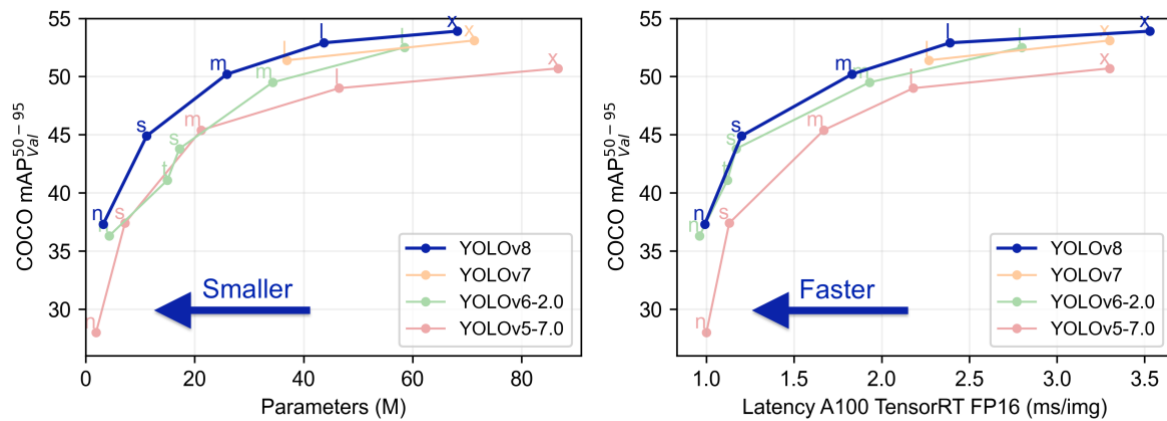


Figure 39: YOLO version comparison [16].

Ultralytics YOLOv8 library also introduces backwards flexibility with previous versions, making it easy to test and switch between different versions. It includes a new backbone network, a new anchor-free detection head, and a new loss function. YOLOv8 is chosen for this project because it is new, state of the art, flexible, easy to use and multi-platform compatible. Performs well on both CPUs and GPUs, which is ideal if the application developed should run on multiple devices. YOLOv8 does not have a publish paper by the time of writing, so the technical explanation presented in this report is based on some easy readings from Roboflow Blog [58] and LearnOpenCV [61] where they have analyzed the available information and GitHub repo. In addition to new features, a huge part of the networks success is the Mosaic Augmentation (MA) in model training that was implemented in the YOLOv5 network. MA is a method of stitching four images together (four $\frac{1}{4}$ of training images) into one image, forcing the model to learn objects in new locations, partially hidden or overlapped, and against different surrounding pixels [58]. This was first time introduced by Zhiwei Wei in 2020 to improve scale variations, object sparsity and class imbalance [62], tested on aerial images. This is one of the methods that has improved accuracy and tiny object detection in the YOLO architecture. An example of MA is shown in Figure 40.



Figure 40: Mosaic augmentation example. Training and validation images of a chessboard with chess piece detection is randomly snipped and combined in different combinations to increase variety during training [58].

As mentioned, there is still limited available theoretical information about what is new in the new backbone network, anchor-free detection head and the new loss function. Luckily, this information is not that relevant to get started with YOLOv8. There exist 5 different sizes of the network, as shown in Table 10. Note: The benchmarks mAP and Speed values are for the COCO val2017 dataset.

Table 10: List of available YOLOv8 detection models.

Model	size (pixels)	mAP ^{val} 50-95	Speed CPU ONNX (ms)	Speed A100 TensorRT (ms)	params (M)	FLOPs (B)
YOLOv8n	640	37.3	80.4	0.99	3.2	8.7
YOLOv8s	640	44.9	128.4	1.20	11.2	28.6
YOLOv8m	640	50.2	234.7	1.83	25.9	78.9
YOLOv8l	640	52.9	375.2	2.39	43.7	165.2
YOLOv8x	640	53.9	479.1	3.53	68.2	257.8

For the specially interested, GitHub user RangeKing made a visual representation of the entire YOLOv8 architecture that can be seen in Figure 41.

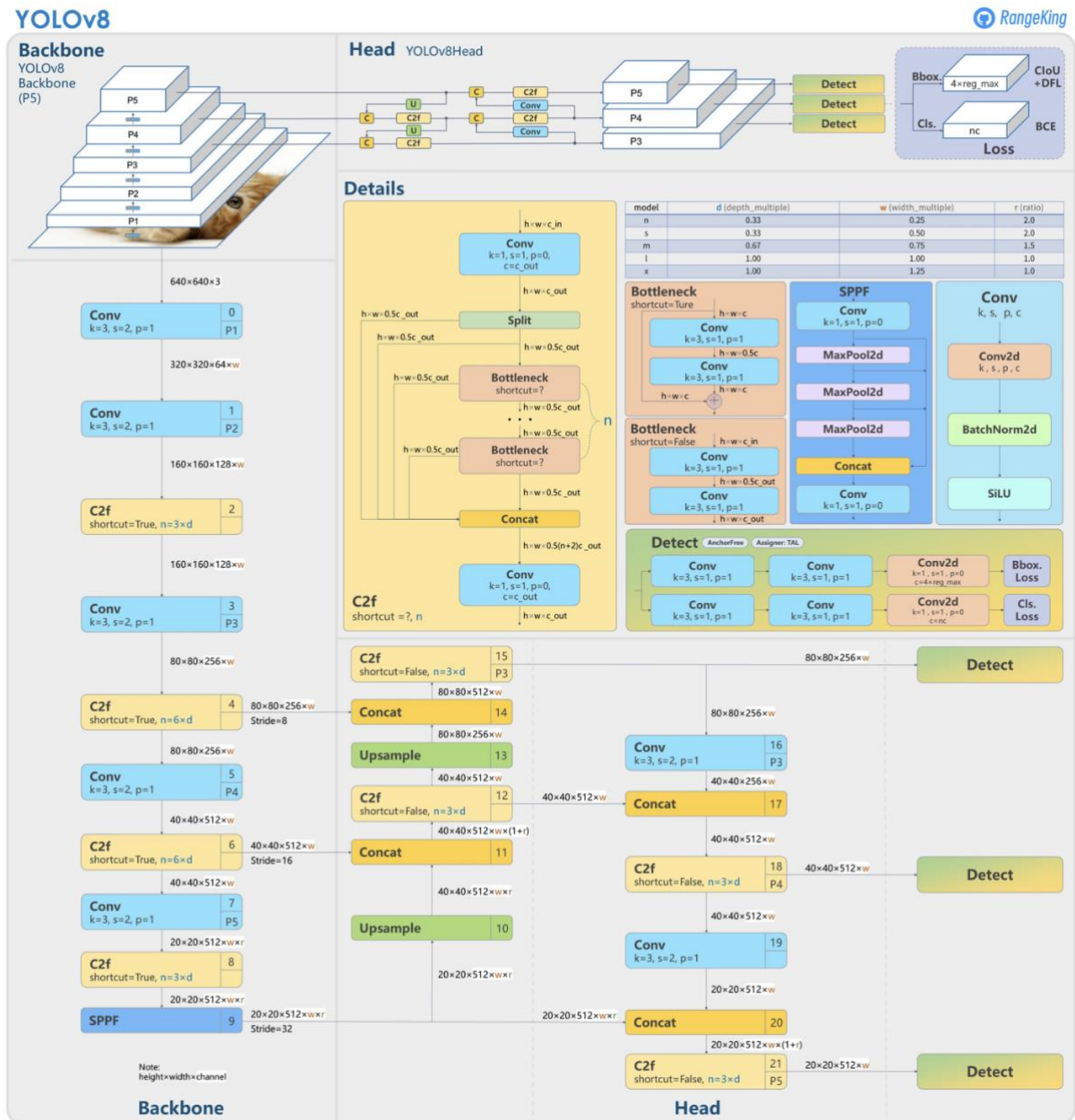


Figure 41: YOLOv8 Architecture, analyzed and visualized by GitHub user RangeKing [58].

Loosely spoken, the general network structure shown in Figure 41 can be interpreted as:

- A backbone network that often is referred to as a feature extraction network, designed to be generic for a wide range of feature extractions.
- Then some C2f (convolution to fully connected) flattening part that converts the 3D tensor to a 1D tensor for prediction. This is required as the fully connected prediction network needs a flattened tensor.
- A head network responsible for task specific feature extraction. Often the network that is being retrained for the specific task, modifying the network to produce the final desired result.
- Then multiple fully connected neural networks for predictions.
- And a final bounding box and class prediction loss calculation.

YOLOv8 has a pyramid multi-scale feature extraction architecture. So, all the conv layers, C2f, Concatenation and Up-sampling basically creates feature maps in different scales. Resizes and restructure the tensors and concatenated them before being fed to the fully connected network for classification. As shown in the last light blue Conv section, the activation function used is called SiLU (Sigmoid Linear Units).

4.3.2.1 SiLU Activation Function

SiLU was originally proposed by Elfwing Stefan [63] in 2017 as an activation function for reinforcement learning neural networks. The original paper called the function Sigmoid-Weighted Linear Unit but it is also commonly known as Sigmoid Linear Units or swish function [64]. A graphical representation of the SiLU activation function compared to the ReLU activation function is shown in Figure 42.

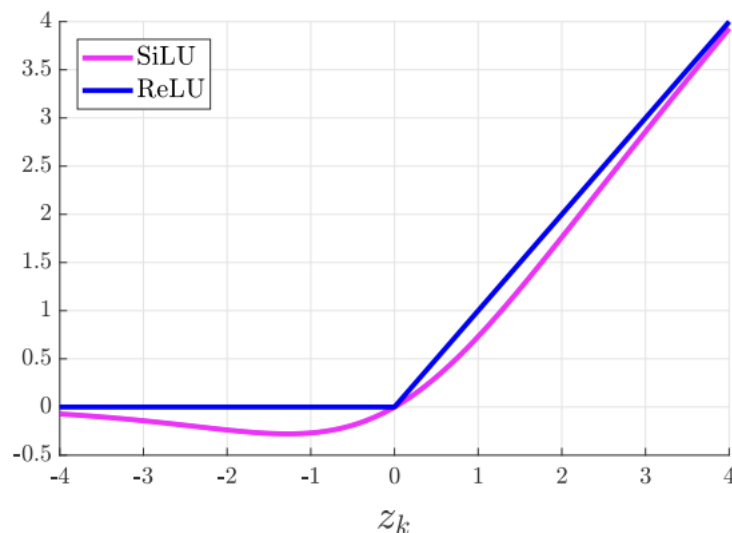


Figure 42: Graphical presentation of SiLU compared to ReLU activation function.

Advantages of SiLU compared to ReLU is that it is a smooth function. Small negative values are still accounted for instead of just zeroed out like for ReLU. This could be important as small negative values could retain information that is important for capturing patterns in the data [32]. Also, it is a non-monotonous function, meaning that it has both decreasing and increasing regions. This contributes to capture more complex interactions between the input and the weights (because it gets both positive and negative values) and can lead to improved

learning and model performance. SiLU is the Sigmoid / Logistic function multiplied by its input as indicated in equation (7) and (8).

$$\alpha_k(x_k) = x_k * \sigma(x_k) \quad (7)$$

Where Sigmoid / Logistic function is:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \quad (8)$$

4.3.2.2 YOLOv8 Loss functions

There are two loss calculations performed for the YOLOv8 object detection network. First one is the bounding box loss calculation of how good a predicted box matches the ground truth bounding box. The second one is the class prediction loss function with BCE. BCE loss function is explained in chapter 4.2.5.1.

The loss calculation for the bounding box prediction is performed using a method called DIoU (Distance Intersection over Union). It is somewhat similar to IoU but the difference is that IoU only works when the boxes are actually overlapping. IoU is explained in chapter 4.4. DIoU on the other hand, does not need the boxes to be overlapping because it calculates both the IoU and the distance (D) from the ground truth box. Thus, taking both the actual size and localization of both boxes into account. DIoU is responsible to measure the similarity between the two boxes.

“Distance-IoU (DIoU) loss incorporates the normalized distance between the predicted box and the target box, which converges much faster in training than IoU and GIoU losses.” – Zhaohui Zheng et al. [65].

See equation (9) where “d” is the Euclidean distance between center point of prediction and ground truth box, and “c” is the diagonal length of the smallest enclosing box that would cover the two boxes [66]. This is also visually presented in Figure 43.

$$DIoU = 1 - IoU + \frac{d^2}{c^2} \quad (9)$$

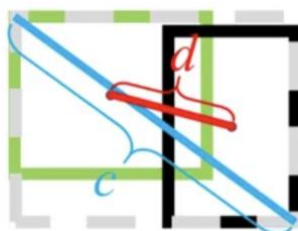


Figure 43: Visual presentation of prediction versus ground truth box and the parameters d and c in the calculation of DIoU. Green box is predicted, grey box is ground truth, dashed line box is enclosing box that would in theory cover both boxes.

The bounding box loss calculation DIoU is performed in combination with a method called DFL (Distance Focal Loss). DFL is a modified version of the more traditional Focal Loss

(FL) method that tries to handle the issue of class imbalance, assigning more weights to easily misclassified examples and less weight to easy examples [67]. DFL uses this theory of FL to incorporate the distance error between predicted and ground truth bounding box into the focal loss calculation, thus giving more weight to samples with larger localization error. DIOU + DFL is combined into one loss function, resulting in a faster loss to zero convergence and accuracy improvement of the bounding box prediction.

4.3.3 Data preparations for YOLOv8 object detection

Custom training data need to be in a certain format for the YOLOv8 model to interpret it. The formatting is simple, shown in Table 11.

Table 11: YOLOv8 annotation file.

class_id	x_center	y_center	object_width	object_height
3	0.267	0.509	0.033	0.023

It is important that the values are normalized between 0 and 1 with respect to full image width and height. As an example, the actual x center of an object id 3 in Table 11 in an image of width 1200px would be calculated as equation (10).

$$\begin{aligned}
 x_{center} &= 0.267 * 1200px \\
 x_{center} &= 320,4px
 \end{aligned}
 \tag{10}$$

The training and validation data can be bounding box annotated using an annotation tool that supports this format. Each class category should have class id, which means that there must be a label map export of class name and related class id as shown in Table 12.

Table 12: Class name – class id annotation mapping.

class_name	class_id
valve	3

When the training and validation data is ready, they need to be sorted into a folder structure as shown in Figure 44. The images could be any file format, and the labels should be .txt formatted.

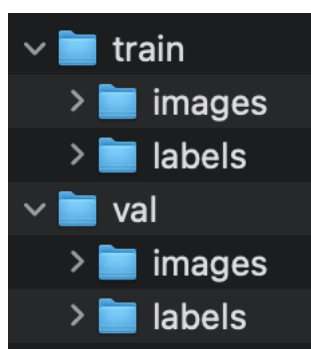


Figure 44: Folder structure, train and validation images and labels.

The final step is to create a custom config.yaml file that can be uploaded on initialization of training, telling the model where to find the data and number of class categories and a list of the class category names. This is shown in Figure 45.

```

1 train: /home/engineirik/git/yolov8_custom/train
2 val: /home/engineirik/git/yolov8_custom/val
3
4 nc: 27
5
6 names: ["value", "valve_m", "nav", "tag", "valve_p", "conveyor", "status", "bargraph", "motor", "valve", "mixer",

```

Figure 45: Configuration file callable on run, data_custom.yaml file. The list of objects needs to be in order from 0 to nc. In this case, “value” has class_id 0. This process is automated by creating a script for reading the label_map.txt file that was exported with the last annotation file.

This process of annotating data and creating the necessary arrangements for this method to run can be automated and optimized. This is a partial goal for this project and will require a custom annotation tool and some data preparation scripts.

4.3.4 Training and validating YOLOv8 object detection

Running a training session with a custom dataset is typically easy for all ML methods if the data is prepared correctly. For YOLOv8, it can either be run directly from the command line (CLI) or terminal, or using Python with only a few lines of code, as shown in Table 13. There are a lot more parameters that can be set for training and predicting, ref documentation.

Table 13: Example of running training and prediction on CLI or Python script.

CLI train	<code>yolo task=detect mode=train epochs=100 data=data_custom.yaml model=yolov8m.pt imgsz=640 batch=8</code>
CLI pred	<code>yolo task=detect mode=predict model=yolov8m_custom.pt show=True conf=0.5 source=1.png line_thickness=1 save=True save_txt=True</code>
Py train	<code>model = YOLO("yolov8x.pt") model.train(data="data_custom.yaml", batch=8, imgsz=640, epochs=500, workers=1, patience=100)</code>
Py pred	<code>model = YOLO(best_model) model.predict(conf=0.5, source="path/test", line_thickness=2, save=True, save_txt=True)</code>

Note: This is of course dependent on environment and package installation. Ultralytics must be installed on the system or virtual environment [68]. PyTorch with GPU support is also highly recommended.

4.4 Non-Maximum Suppression

Non-maximum suppression is in short terms a filter that is applied to filter out all bounding box proposals that does not meet a certain criterion [69]. It is an algorithm that takes a list of boxes, their corresponding confidence score and location as input and further:

1. Sort each box based on their confidence score and removes all boxes bellow a preset threshold. The confidence threshold is defined by the user.
2. Picks the bounding box with the highest score and remove all the other boxes that overlaps with it within a preset threshold. The overlap threshold is defined by the user. The overlap is calculated using the IoU, also called the Jaccard index formula shown in equation (11), graphically represented in Figure 46.
3. Step 2 is repeated until there are no more bounding boxes to process.

The result after this non-maximum suppression should be a single box classifying each individual object in the picture.

4.4.1 IoU – Intersection over Union

Intersection over Union (IoU) is a way to measure how much two things overlap. It is commonly used to compare the accuracy of object detection or image segmentation models. IoU is calculated by dividing the area where the two object detection boxes overlap by the total area that they cover. The resulting value is between 0 and 1, where 1 means the two things completely overlap and 0 means they don't overlap at all.

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{11}$$

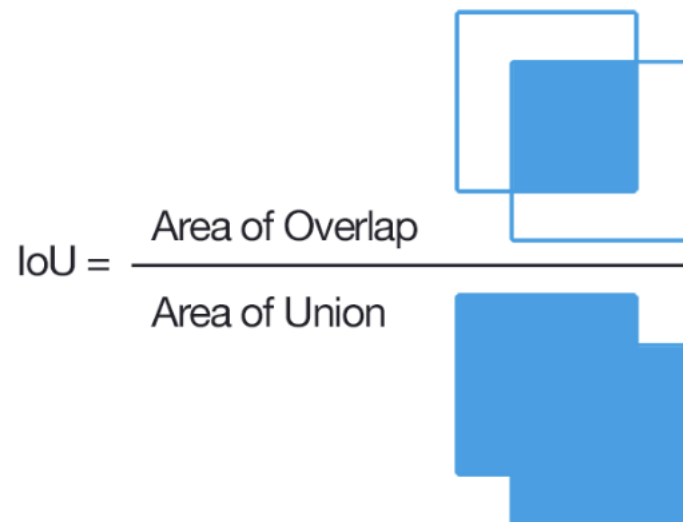


Figure 46: Graphical representation of the IoU formulation. IoU is calculated by union the divided overlap between proposed bounding box and ground truth [70]. The ground truth being the bounding box with the highest score.

4.5 Software analysis and design

Two pieces of software is being developed for this project. The first one is a semi-automated annotation software, and the second one is the final Industrial Component Extraction software (ICE). The software functionality is analyzed using concepts from Object-Oriented Analysis, Design and Programming (OOADP). The testing of each software is explained in the Results chapter 5.2.2 and 5.2.5.

4.5.1 Semi-automated annotation software (Program 1)

The semi-automated annotation software will utilize the multi-label classification with sliding window pyramid scale NMS method and serve as a custom software designed for engineers to annotate images for one-stage or two-stage detectors.

4.5.1.1 Application requirements

"A software for annotating images that will serve as training data for a one-stage or two-stage detection method. The annotation software should have the ability to pre-analyze the image using a multi-label classification sliding window algorithm. This will help with reducing manual labor when annotating training images."

Without pre-analysis:

- User should be able to upload an image.
- Preprocess the image by cropping it to a desired size.
- Directly annotate the image without any pre-analysis.
- Export the image file used for annotation and the annotation file in correct format according to detection method (xml, csv, or txt format).

With pre-analysis:

- User should be able to upload an image.
- Preprocess the image by cropping it to a desired size.
- Pre-analyze the image by selecting a .pkl file (model) used for sliding window classification.
- Watch the analysis progress.
- View the pre-analyzed image.
- Make changes and improve the annotation.
- Export the image file used for annotation and the annotation in correct format according to detection method (xml, csv, or txt format).

How to annotate:

- User should be able to drag new boxes on the screen and add a label to the box.
- The label and number of same label objects should be listed in a separate window.
- User should be able to resize, move, edit label, and delete box by using the mouse.
- Suggested mouse interactions:
 - o Mouse left-click-drag makes new box. Choose size, release mouse to put sized box on image. Label automatic prompted on mouse release, enter by typing.
 - o Mouse mid-click within existing box, delete the box.
 - o Mouse right-click within existing box opens label edit prompt.
 - o Mouse left-click-drag within existing box moves the box. Release mouse button on new location.
 - o Mouse left-click-drag on corner of existing box resizes the box. Release mouse button on new size.
- The annotation should automatically be updated in the list, csv file and export format file.
- Save on quit.

4.5.1.2 Domain Model and System Sequence Diagram

A domain model is helpful to visualize the conceptual classes in object-oriented programming. As the focus when developing this software was not object-oriented, this domain model shown in Figure 47 serves as an overview of all the required functions that the software needs to meet the requirements.

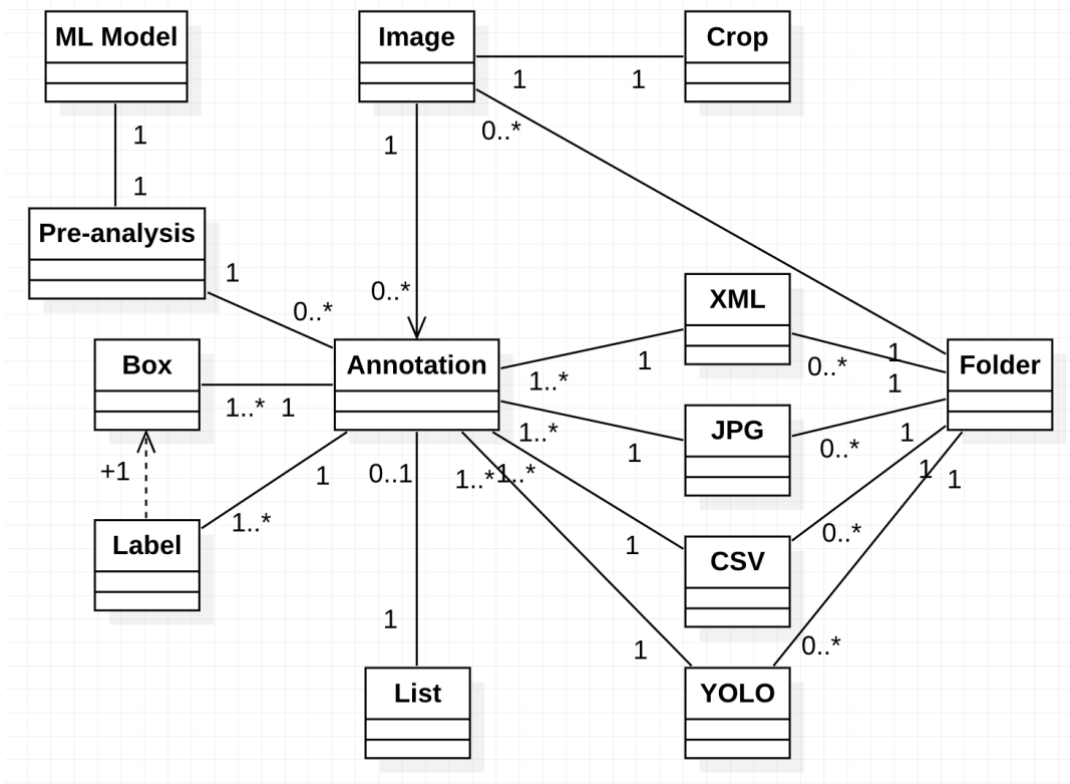


Figure 47: Domain Model annotation software.

A System Sequence Diagram (SSD) is also helpful to visualize how the process flow of the software will work from input to analysis to annotation to export. See Figure 48. Normally an SSD explains the informational flow of one or more use cases, but for this software it is simply used to visualizing how the interaction to the software triggers functions and how each function triggers sub functionality within the software.

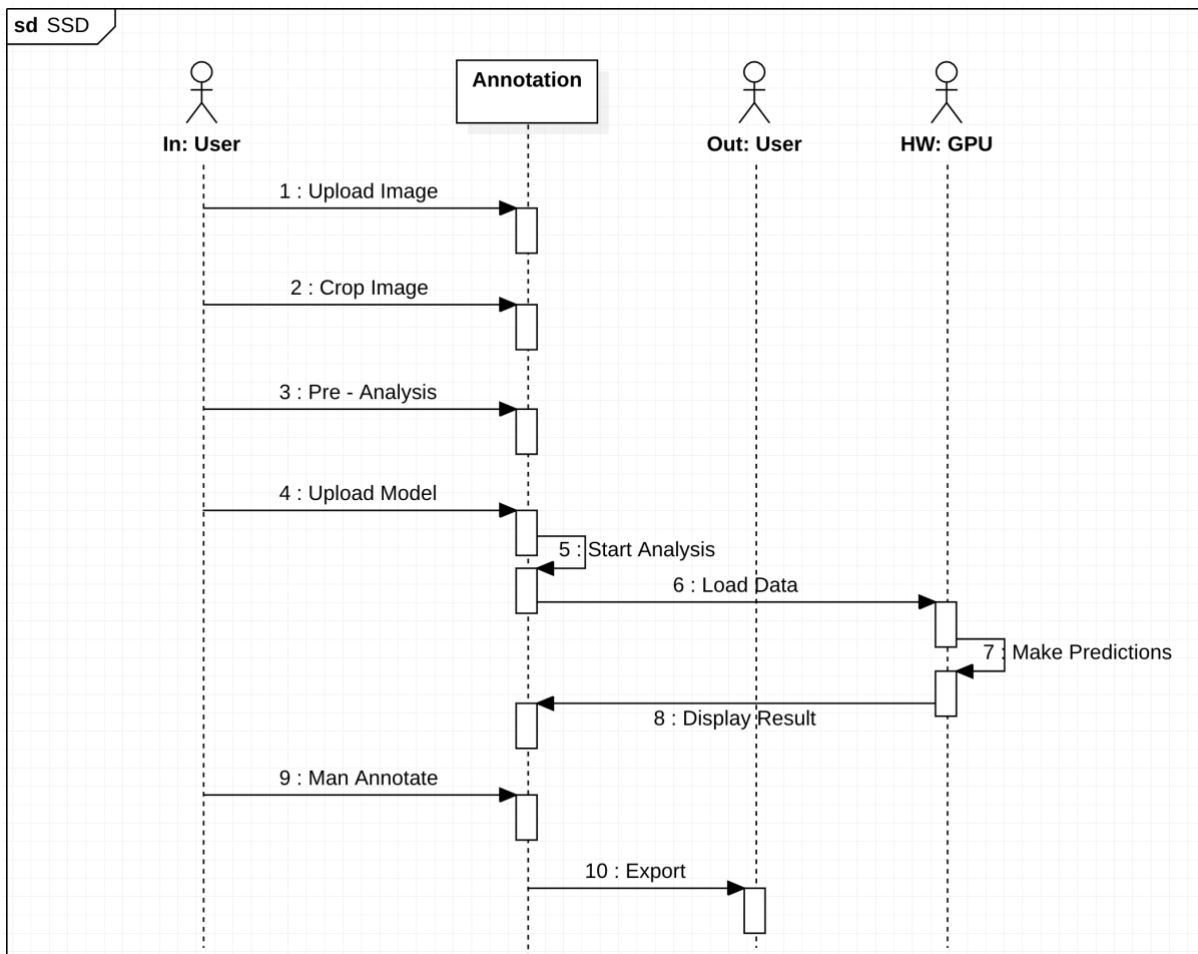


Figure 48: System sequence diagram annotation software.

4.5.1.3 Prototype design mockup

Creating a simple prototype mockup is an important step to visualize the idea, give a sense of interaction flow and functionality. This tool will only be used by technical personnel and engineers, so focus on UI design for consumer market is not of any concern. The theme for the application is folder structure, grey colors, and button interaction. The cropping of images will be performed using trackbars. The annotation of images should be performed as explained in suggested mouse interaction section of the requirements list. A mockup can be drawn using tools or using pen or paper. A paper drawn mockup is available in Appendix M.

4.5.2 Industrial Component Extraction tool – ICE (Program 2)

The ICE software will be the final solution of this project, provided as a user-friendly tool for performing operator interface image analysis.

4.5.2.1 Requirements

“A software where the user can upload one or more operator interface graphics images, click a button to analyze, view progress, and get a downloadable document in return. The software should have a nice design and be user-friendly. In the backend, the software will

perform object detection, tag extraction, linking of objects and tags, and generating an analysis document for export.”

User interaction requirements:

- Correct design based on UI design principles.
- Maximum of 3 clicks to achieve the main goal of the application.
- Home screen:
 - o A button for uploading images. Prompt user with folder structure where the user can select documents.
 - o A text field that shows how many documents that are selected for the analysis.
 - o A button to start the analysis.
- Loading screen:
 - o A progress or waiting indication.
- Finished screen:
 - o A text showing status analysis.
 - o A button for downloading the analysis document.

Backend requirements:

- Main function containing the object detection and calling of all subclasses: text extraction, object tag linking, generate excel document.
- Separate class for OCR text extraction.
- Separate class for minimum Euclidean distance calculation and object tag linking.
- Separate class for excel document generation.

4.5.2.2 System Sequence and Class diagram

A simplified system sequence diagram is created to show the flow of execution from the user interacting with the UI and methods executing in the background in Figure 49.

- The user upload images, and the images are loaded into a uploads folder.
- The N number of uploaded images are displayed to the user.
- The user then clicks the start analysis button, and the analysis is initialized.
- In the backend, the software starts by feeding the uploaded images to the OCR for extracting tags, then detect objects using the YOLO prediction model. These two functions create separate annotation files for objects and tags.
- The annotation files are fed to a link objects function that links the objects and tags that are close to each other as explained in minimum Euclidean distance chapter 5.2.4.2.
- Now the final analysis is then fed to a method for generating a excel sheet that will be available for download.
- While the backend is working, the user will see a progress bar, indicating that the software is working.
- When the excel doc is generated, the user will be displayed with a success screen with an option to download the analysis excel document or perform a new analysis.
- User can click to prompt file explorer and download document.

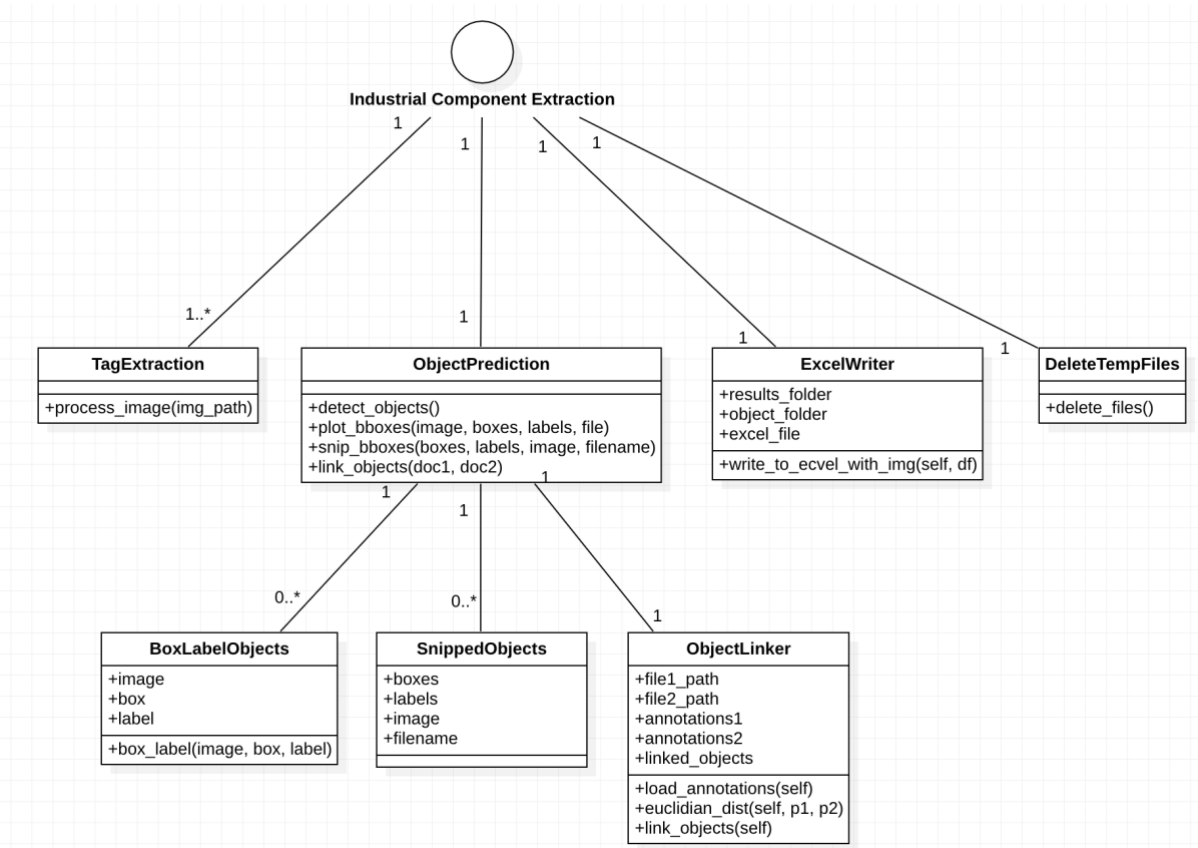





Figure 50: Class diagram for ICE software.

4.5.2.3 UI prototyping and design mockup

Figma is used to prototype the design for this application. Two designs is tested for both desktop/web and mobile layout. The mobile design can be viewed Figure 51, and the desktop/web designs can be viewed in Appendix N. The desktop/web design are similar to the mobile design, just rescaled to fit the browser window width and height. The design is kept simple, choosing colors of blue and purple which represents trust, safety, peace, and calm. These colors are also good for getting good color contrasts on white background [71]. All colors used in this design is listed in Table 14, and are within the recommended specifications of at least 7:1 on color contrasts for normal text, and 4.5:1 on large text as specified by the WCAG AAA (Web Content Accessibility Guidelines 2.0) [72]. All contrasts are checked with a contrast checker from WebAIM [73].

Table 14: Color table components in design. All contrasts are checked towards a background of #E3F3FE which is a nuance of blue and a representative of where the text is located on the gradient blue to white background.

Component	Color	Contrast to background
Text	#3126A5	9.4:1
Border	#4B4EDC	5.4:1

Header text	#333333 grey 	11.1:1
White background	#FAFAFA mild nuance of regular white 	
Blue background gradient	#3A50B0 	

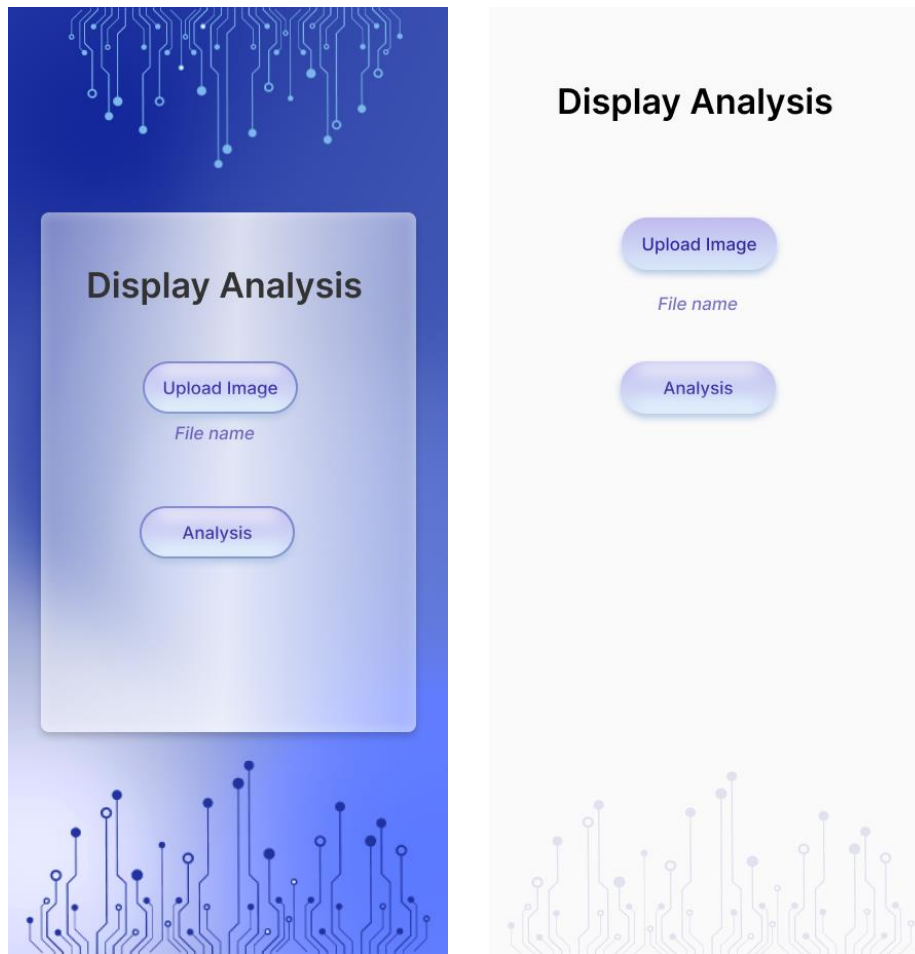


Figure 51: Design prototyping mobile layout. The desktop layout looks the same, it just scales depending on browser window size.

The loading and results screen will inherit the same design for colors on the progress bar, buttons, and text. No design mockup is provided for these.

4.5.2.4 Development

The software is developed using a Python framework called Flask. Flask Python is a lightweight web framework only providing the most essential components for building web applications, frontend, backend, and APIs [74]. Since all the software developed in this project is in Python, it was only natural to develop the final software with a Python framework. Flask, like most web frameworks are based on the Hypertext Transfer Protocol

(HTTP) protocol, using “GET”, “POST”, “PUT” and so on. It is therefore easy to host such and application on a server at a later stage. During this project’s development and testing, the application is hosted locally.

Since this is a web application with requirements of three screens, three routes are created in the applications main document “app.py” as shown in Figure 52. These three request routes call the applicable functions to achieve the use case functionality. Each route has its own user interface, coded in Hypertext Markup Language (HTML), and styled using Cascading Style Sheets (CSS).

```

33 @app.route('/')
34 def index():
35     return render_template('index.html')
36
37 @app.route('/', methods=['POST'])
38 def upload_files():
39     if 'file' not in request.files:
40         flash('No file part')
41         return redirect(request.url)
42
43     files = request.files.getlist('file')
44
45     for file in files:
46         if file.filename == '':
47             flash('No selected file')
48             return redirect(request.url)
49
50     # Save file to uploads folder
51     filename = file.filename
52     file.save(os.path.join(app.config['UPLOAD_FOLDER'], filename))
53
54     return render_template('loading.html')
55
56 @app.route('/start_analysis', methods=['POST'])
57 def start_analysis():
58     print("Starting analysis...")
59     # Perform analysis in a separate thread
60     extract_tag()
61     detect_objects()
62     response = {"status": "success"}
63     # If the analysis is complete, return the response
64     return response
65
66 @app.route('/results')
67 def show_results():
68     return render_template('results.html')
69

```

Figure 52: Index route “/”, upload_files() button click “POST” request, start_analysis() button click “POST” request and final results “/results” route defined in app.py. Note: the comment “#Perform analysis in separate thread”, is not correct as there is no multi-threading in this application.

4.6 Quick recap

The detailed explanations of ResNet architecture, CNN, activation functions, pooling layers, loss calculations, backpropagation, NMS, IoU, and YOLOv8 architecture are included in this report because they are crucial concepts that can help the reader understand the upcoming project results. The remaining sections of the report will present results on single-label

classification and testing of models, as well as multi-label classification models, multi-class object detection, and YOLOv8 object detection models, without further elaboration on technicalities. The theory and analysis provided in this Methods chapter lays the theoretical foundation for further reading.

5 Result

The Results chapter presents a detailed explanation of all the tasks undertaken during the project, including the approach and techniques used, scripts developed, and the final outcomes. It describes how each task was executed and explains the reasoning behind the selected approach. In addition, this chapter will go through additional tricks and code that were developed to simplify repetitive tasks and the final product that was delivered. The chapter contains a lot of tables and figures to try and simplify explanations.

5.1 Image classification

This chapter will go through single-label and multi-label image classification. Why they are used, how to train a custom model, and results related to each approach. Starting with data assembly listing class labels, and some important steps in creating the code for training, validating, and testing.

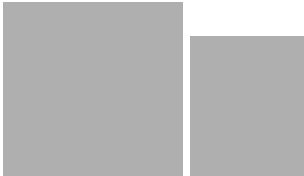
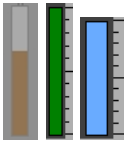
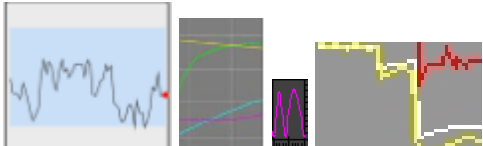
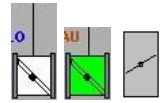


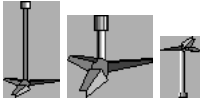
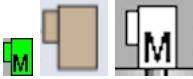



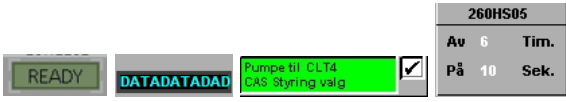
5.1.1 Single-label classification


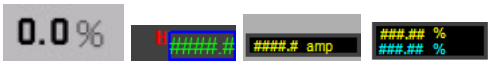




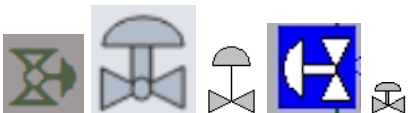

The single-label classification method serves as a bases in this project, to check how well a prebuilt deep learning neural network model performs after training it on the specific project data. An untrained ResNet50 model and a pretrained ResNet50 model will be retrained using the project data. The two models will be compared to see if it is beneficial to use transfer learning on a pretrained model, thus keeping some of the learned features, or if it is best to start from an untrained model. The ResNet neural network architecture is chosen because the FastAI API provides out of the box modules pretrained on the ImageNet dataset. ResNet50 is chosen as it is the arguably best balance between complexity and size. In general, a model with deeper architecture will be able to model data more accurately but will also be more prone to overfitting [75]. There are many warnings and arguments to this generalization, but a deeper architecture with more layers and parameters will capture the data in a better way. As the training data in this project has a lot of similar features, it is important to select a model that has enough capacity to capture the specific feature that differentiate two “look-alike” object classes. Remember, deeper architecture will require more GPU RAM, and could compromise hardware performance, thus resulting in out of memory failure. If this issue occurs, it can be solved by reducing the training and validation batch size.

5.1.1.1 Assemble data

Data for this specific task is collected from the dataset obtained in the beginning of the project. Only pictures that contain one and only one object are separated into different class folders. 20 class folders are created, containing pictures that fits the class description and therefore labeling the pictures. See Table 15 for list of classes and example dataset objects.

Table 15: Single-label classifier classes and dataset examples.

Class name / folder name	Example from dataset
background	
bargraph	
chart	
damper	
fan	
line	
mixer	
motor	
nav	
pump_isa	
pump_iso	
status	

tag	
value	
valve	
valve_h (hand)	
valve_m (motor)	
valve_m_3w (motor 3way)	
valve_p (pneumatic)	
valve_pr (pressure release)	

A keen observer will quickly see that due to limited features in each class, the model might get overconfident and overfit easily. This is important to keep an eye out for when training. There are also some classes that are quite similar such as “status”, “tag” and “value”, therefore might end up misclassifying a lot.

5.1.1.2 Setting up DataBlock and preparing learner

Next step is to write a Jupyter Notebook Python program, separate the data into training and validation set and prepare a DataBlock for training. As mentioned in the system description chapter 3.5.5, the FastAI toolbox have predefined classes and methods that makes it easier to quickly set up DataBlocks, train models and deploy applications. Start by creating a path variable to the datasets, shown in Table 16.

Table 16: Define path variable to training and validation dataset.

```
path =
Path("/home/user/git/classify_singl_obj/Classification_small_singleobject")
Path.BASE_PATH = path
path.ls() #lists objects in path
```

Then define the DataBlock and load the dataset into it using dataloaders, see Table 17. A batch from the training set can be viewed by calling the “show_batch” method on the data object, as shown in Figure 53. From the DataBlock code, it is important to specify what type

of deep learning task that is going to be performed, in this case a ImageBlock for image classification. The items are collected in the “get_items” parameters using a function “get_image_files” that will retrieve all image files in subfolders from path, and label them with the folder name (see chapter 3.6.2.2). The data is split 20% into validation set and 80% training set, with a seed of 42 ensuring the same split every time this DataBlock is called. The “get_y” parameter gets the folder name of each object to label the data. The “item_tfms” transforms each image according to specifications such as resizing, cropping, padding etc.

Table 17: Define the DataBlock and load the data using dataloaders.

```
data = DataBlock(
    blocks=(ImageBlock, CategoryBlock),
    get_items=get_image_files,
    splitter=RandomSplitter(valid_pct=0.2, seed=42),
    get_y=parent_label,
    item_tfms=Resize(224, ResizeMethod.Pad, pad_mode='zeros')
)
dls = data.dataloaders(path)

dls.valid.show_batch(max_n=4, nrows=1)
```

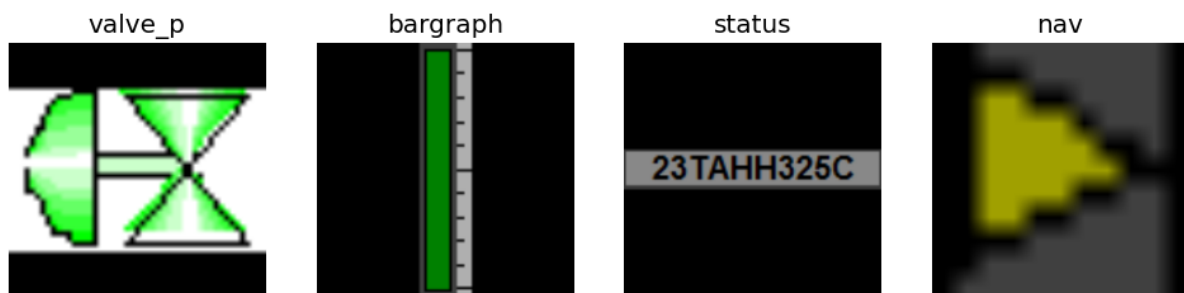


Figure 53: Batch from dataloader training set.

5.1.1.3 Training non-pretrained model

Next, train the non-pretrained ResNet50 model called xresnet. The xresnet50 is downloaded from the FastAI API and loaded into a model parameter, specifying number of outputs from the dls (dataloaders). Based on the theory provided in chapter 4.2.1, it was shown the importance of specifying number of outputs in the dense layer (number of classes). A learn parameter is called with the “model”, dataloaders (dls), and “loss_function” is set to CrossEntropyFlat with metrics set to “accuracy”, see Table 18. CrossEntropyFlat also known as Flat Cross Entropy is a variation of the CE loss function designed to handle class imbalance, which is important in this project as some classes have way more sample than others. Since this is a non-pretrained network, there is no need to freeze any epochs when training so the “fit_one_cycle” method is called on the learn object. The accuracy will gradually improve during training, as seen in Figure 54.

Table 18: Defining model, learner, metrics and start learning cycle of single-label non-pretrained xresnet50 classification model.

```

model = xresnet50(n_out=dls.c)
learn = Learner(dls, model, loss_func=CrossEntropyLossFlat(),
metrics=accuracy)
learn.fit_one_cycle(12)

```

epoch	train_loss	valid_loss	accuracy	time
0	2.268960	2.210411	0.241758	00:20
1	1.729840	1.700031	0.494505	00:19
2	1.421203	1.306692	0.600733	00:19
3	1.192690	2.140647	0.560440	00:19
4	1.037965	1.185952	0.666667	00:19
5	0.923955	0.903173	0.761905	00:19
6	0.799859	0.688817	0.776557	00:19
7	0.684407	0.709529	0.761905	00:19
8	0.591151	0.585039	0.798535	00:19
9	0.507836	0.564602	0.831502	00:19
10	0.440936	0.527033	0.846154	00:19
11	0.387851	0.522242	0.846154	00:19

Figure 54: Non-pretrained xresnet50 training result.

After training for 12 epochs, a decent base result with an accuracy at 84.6% is achieved. The training could have been limited to 11 epochs, as the accuracy don't approve at all after that. Looking at the training and validation loss plot in Figure 55, the validation flats out at the end indicating that the model will only start overfitting at this point.

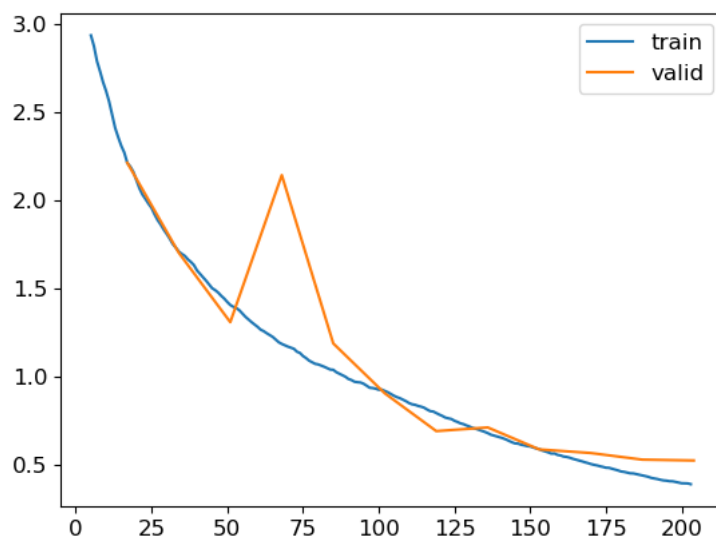


Figure 55: Training and validation loss plot non-pretrained xresnet50.

It is possible to improve this model by using additional techniques such as normalization, progressive resizing, test time augmentation, mixup and label smoothing. These techniques are really important when working with models that are being trained from scratch [76]. However, there are many pretrained versions of this xresnet model that is available from the FastAI library, where many of these techniques mentioned above are automatically applied.

5.1.1.4 Training a pretrained model

A pretrained deep learning model has been trained on a large dataset for a specific task such as image classification. During the training it learned about general features such as edge detection, corners and contours and other features as illustrated in Figure 15 in chapter 4.1.1. This pretrained model is later used as a starting point for fine-tuning on a new, smaller dataset. Typically, this will require some modifications to the last layers of the deep learning model to adapt to the new task. The process of retraining a pretrained model to fit a new task is called transfer learning. This method is often preferred because it contains more world knowledge from the get-go, and retraining will require significantly less computational resources and time. One additional benefit of using pretrained models is that the FastAI library will handle all the preprocessing of data according to the already trained resnet50 model.

Now, to perform transfer learning with the FastAI library, start by adjusting the learner seen in chapter 5.1.1.3, Table 18, to the following code snippet shown in Table 19, and define fine-tune method of the object instead for the fit-one-cycle. Remember to swap the xresnet50 to the resnet50 pretrained model. Also, use the “freeze_epochs” parameter to freeze all layers except for the last one for 3 epochs in the beginning of the training.

Table 19: Adjusted code snippet for transfer-learning a pretrained ResNet50 model.

```
learn = vision_learner(dls, resnet50, metrics=accuracy).to_fp16()
learn.fine_tune(9, freeze_epochs=3)
```

Freezing layers is helpful to prevent the previously trained layers to be updated in the initial training of the new model, thus keeping their pre-trained knowledge to help the new layer(s) learn the task-specific features more efficiently. The new layers will adjust their weights to fit the pre-learned features, resulting in a faster convergence and prevent overfitting. The pre-learned layers will normally contain information that is more general, and helpful in initial training. After a few epochs, all layers are unfrozen, and the pre-trained layers will start adjusting its weights to fit the specific new task.

The number of epochs is the same as for the non-pretrained model, freeze for 3 epochs, then run 9 unfreeze, total of 12. As seen in Figure 56 the accuracy of the pre-trained model is a lot higher than the non-pretrained model. It is arguable that the training should have been limited to a total of 9 epochs, as the 5th unfrozen epoch gives the highest accuracy, and the validation loss seems to get worse after unfrozen epoch 5. Improved training loss and worsening validation loss is a clear indication that the model is overfitting to the training data. Figure 57 also illustrates that the training loss keeps decreasing, but the validation loss starts getting worse at a certain point, indicating overfitting. From looking at this plot in Figure 57, it might even be argued that the training should have stopped after unfrozen epoch 3, as this was the

point where accuracy was at 96.7% which is high, and the validation loss was at its lowest on 0.095.

The number of epochs is not taken into account for this task as its primary goal is to test whether a non-pretrained or pretrained model should be used for this project.

epoch	train_loss	valid_loss	accuracy	time
0	2.896408	0.995524	0.736264	00:11
1	1.646108	0.458982	0.868132	00:12
2	1.059790	0.362602	0.886447	00:11

epoch	train_loss	valid_loss	accuracy	time
0	0.278839	0.299870	0.890110	00:14
1	0.195774	0.238112	0.915751	00:14
2	0.161242	0.203671	0.937729	00:14
3	0.136923	0.095021	0.967033	00:14
4	0.109998	0.158871	0.963370	00:14
5	0.082578	0.104049	0.974359	00:14
6	0.065645	0.113539	0.959707	00:14
7	0.047687	0.122736	0.959707	00:14
8	0.041199	0.119036	0.959707	00:15

Figure 56: Pretrained resnet50 training result.

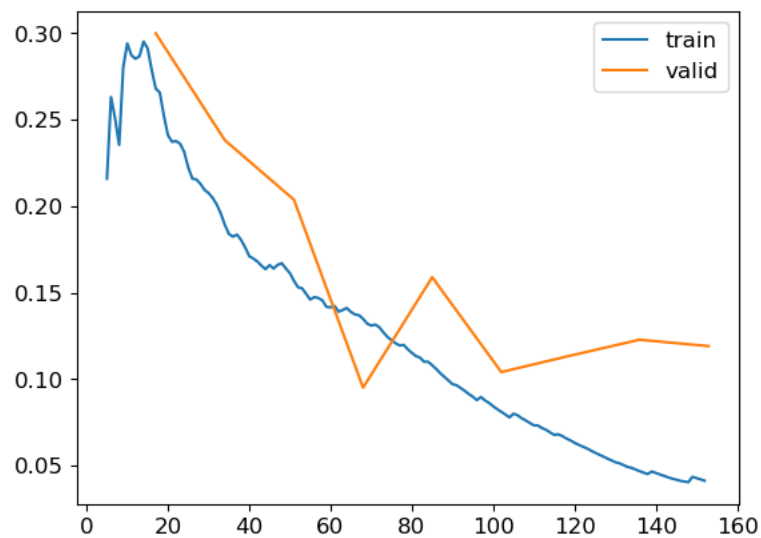


Figure 57: Training and validation loss plot resnet50.

5.1.1.5 Comparing the non-pretrained and pretrained model

As expected, transfer learning a pre-trained model is the clear winner in this case. A quick comparison of the confusion matrixes of both the non-pretrained (Figure 58) and pretrained (Figure 59) model indicates that the non-pretrained model is a lot more confused on the classifications. There are a lot more miss-classifications, especially when it comes to different type of valves in Figure 58.

The pre-trained transfer learning approach will be used further in this project. Full source-code for this single-label classification non-pretrained and pretrained problem can be found in Appendix E. Note that the augmentation part is included in the full source-code even though it is not used for this first step of the project. Only used for testing different results.

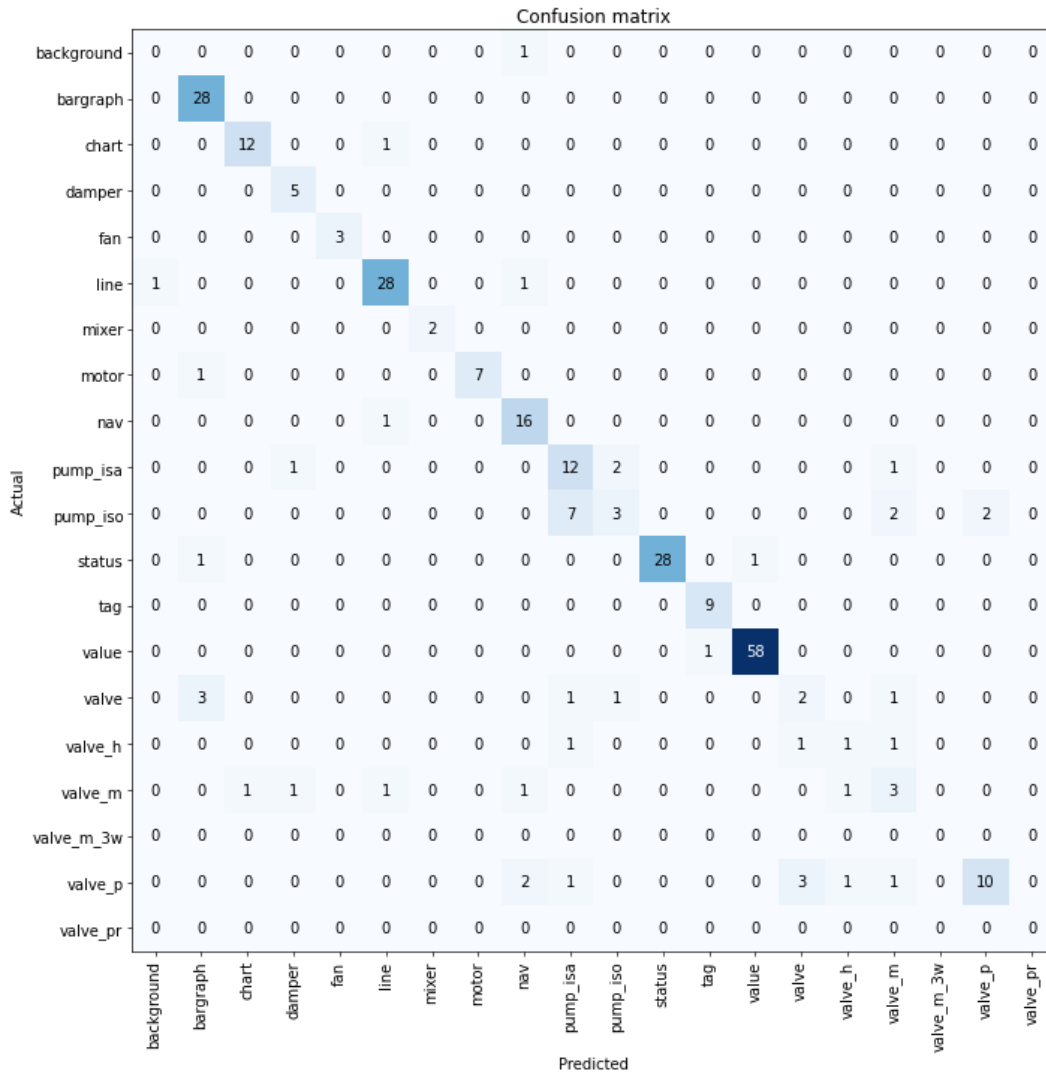


Figure 58: Non-pretrained xresnet50 confusion matrix.

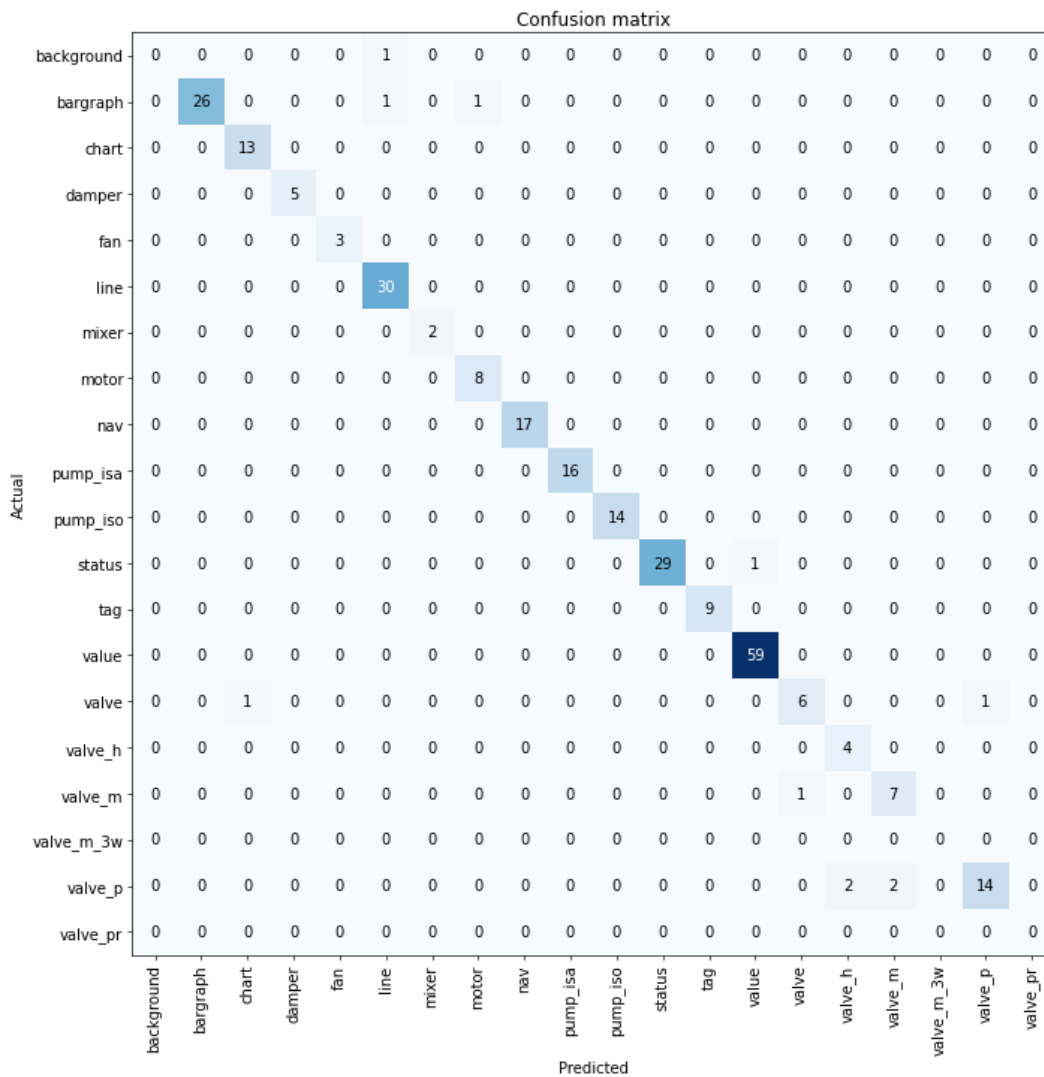


Figure 59: Pretrained resnet50 confusion matrix.

5.1.1.6 Augmenting dataset

There was no reason for doing any augmentation on the data for this test, but this will be an important technique later in the projects as it is a useful technique when having few objects in the training data [77].

5.1.2 Multi-label classification

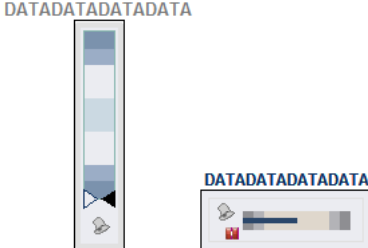
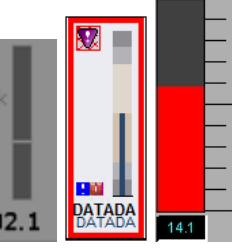


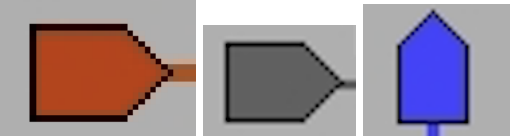


Multi-label classification is the task of recognizing the categories of objects in images where the image may not contain just one type of object, but also multiple or no objects. This “no objects” feature is useful in most real-world cases as it will give the model an option of not classifying images if it can’t find any objects that it is not trained to find. The multi-label classification approach differs from the single-label image classifier in that the latter will always attempt to classify an object in an image as one of its trained classes. It will also give the model the opportunity to indicate that there is some possibility that there might occur

multiple objects in one image. This is helpful in cluttered images where some objects can be close to or even overlaying each other.

5.1.2.1 Assemble data

The object classes are the same as for chapter 5.1.1.1 Table 15, with additional multi-class folders shown in Table 20. Each class is separated with a space on the folder name.

Table 20: Additional multi-label classifier classes and dataset examples.

Class name / folder name	Example from dataset
bargraph tag	
bargraph value	
fan tag	
motor tag	
nav line	
nav tag	
pump_isa line	

pump_isa tag	
pump_isa tag line	
pump_iso line	
pump_iso tag	
status tag	
value tag	
value tag line	
valve line	
valve_3w line	
valve_h line	
valve_h tag	
valve_h_3w line	
valve_m bargraph	
valve_m line	

valve_m tag	
valve_m tag line	
valve_m_3w line	
valve_m_3w tag	
valve_p line	
valve_p tag line	
valve_p tag value	
valve_p value	
valve_p line value	

A script for converting this folder information into a specification CSV file is created as explained in chapter 4.2.4.2.1. This script will iterate through all folders, renaming the files according to the folder name i.e., labeling the files, then putting folder name and file name in the CSV format as shown in example Table 9. A snippet of the “items.csv” specification file loaded into a Pandas data-frame is shown in Figure 60.

	fname	labels	is_valid
0	pump_isa 8.png	pump_isa	False
1	pump_isa 61.png	pump_isa	True
2	pump_isa 45.png	pump_isa	False
3	pump_isa 69.bmp	pump_isa	False
4	pump_isa 2.png	pump_isa	False

Figure 60: Snippet of CSV specification file loaded into Pandas data frame.

5.1.2.2 Setting up DataBlock

The main difference between executing the single-label classification and this multi-label classification is the csv file and type of learner needed to train the model. The multi-label classifier requires a “get_x” and “get_y” function for defining the input (fname) and the output (labels), and a splitter function for splitting the test and validation objects (is_valid) as shown in Table 21.

Table 21: Split dataset into train and validation sets, link input (get_x) and output (get_y).

```
def get_x(r): return path/'train'/r['fname']
def get_y(r): return r['labels'].split(' ')

def splitter(df):
    train = df.index[~df['is_valid']].tolist()
    valid = df.index[df['is_valid']].tolist()
    return train,valid
```

Then by creating a DataBlock variable specifying MultiCategoryBlock, the dataloader can be called as shown in Table 22. Notice how the batch size (bs) is set to 16 for this to limit computation stress.

Table 22: Defining DataBlock for specific task and calling dataloaders with the pandas data-frame as input, specifying batch size of 16.

```
data = DataBlock(blocks=(ImageBlock, MultiCategoryBlock),
                 splitter=splitter,
                 get_x=get_x,
                 get_y=get_y,
                 item_tfms = RandomResizedCrop(224, min_scale=0.35))
dls = data.dataloaders(df, bs=16)

dls.show_batch(nrows=1, ncols=3)
```

Figure 61 shows part of a batch for the first data loader. To normalize the data, the items are transformed by using the random resize and cropped by scaling each object. This is not a

good approach for this project as a lot of the key item features are then lost. To change this, a new DataBlock is created with variable “item_tfms” changed to scale with zero padding as shown in Table 23.

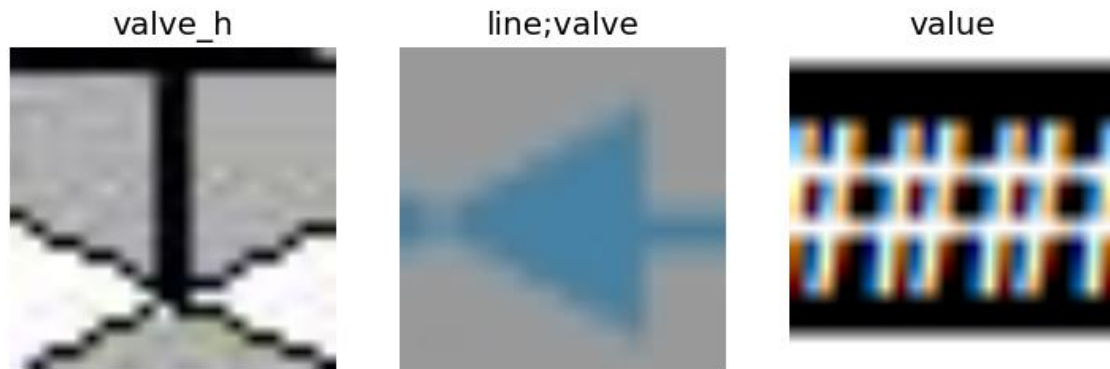


Figure 61: Batch of first data loader resize scaled with cropping.

5.1.2.3 Augmenting dataset

When creating the new DataBlock, an augmentation transform can also be applied to the dataset using the “batch_tfms” variable shown in the code snippet in Table 23.

Table 23: New DataBlock with changed item_tfms and a batch_tfms.

```
data = data.new(
    item_tfms=Resize(224, ResizeMethod.Pad, pad_mode='zeros'),
    batch_tfms=aug_transforms(size=224, min_scale=1, mult=2, max_warp=0,
    do_flip=True, flip_vert=True, max_zoom=1.0, pad_mode="zeros",
    max_rotate=0)
)
dls = data.dataloaders(df, bs=16)

dls.show_batch(nrows=1, ncols=3)
```

A batch of augmented items is shown in Figure 62. The items are now scaled with padding, leading to black edged pixels not containing any information. This worked best for this project, as the key features of each object is kept at scale. Note that black pixels not containing any information will take up computing resources, not considerable in this case. The items are also flipped vertically and horizontally to provide more variety in the training. No zoom or wrap applied as it is not likely for any objects in the image classification.

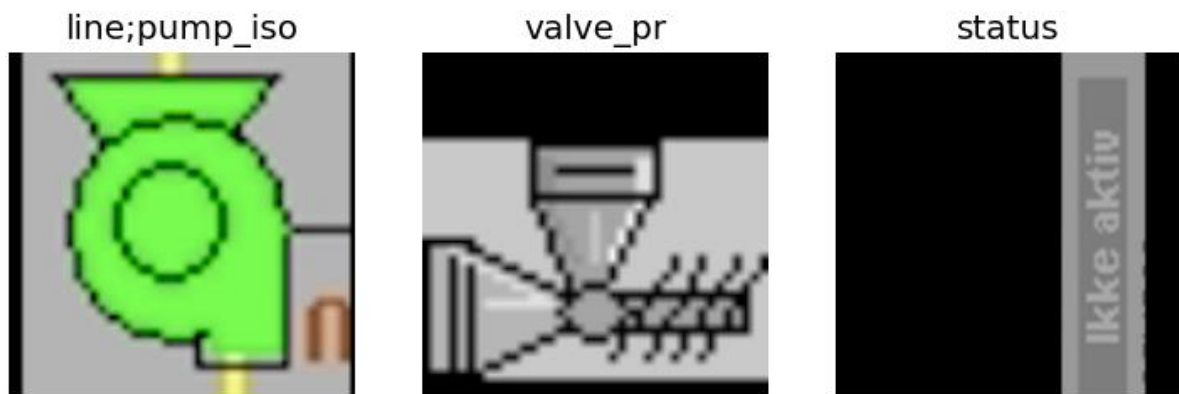


Figure 62: Batch of second data loader with resize scaled with padding.

5.1.2.4 Training a base model

Define a learner with the resnet50 model that was chosen in the previous step of single-label classification. The metrics is set to an accuracy multiplier with the default threshold of 0.5, see Table 24. The threshold is important because it lets the accuracy function know at what threshold (percent certainty) should the activation function classify an object as 1 or 0, also known as firing the neuron. Any value above the threshold is classified as 1 and below as 0.

Table 24: Define learner with dataloaders, model, metrics and threshold for the metrics.

```
learn = vision_learner(dls, resnet50, metrics=partial(accuracy_multi,
    thresh=0.5))
```

Then use the FastAI learning rate finder to find the optimal learning rate, see the method call in Table 25. Learning rate and back propagation is discussed in detail in chapter 4.2.5.2.

Table 25: Using the find learning rate function from the fastai library.

```
lr_min,lr_steep = learn.lr_find(suggest_funcs=(minimum, steep))
```

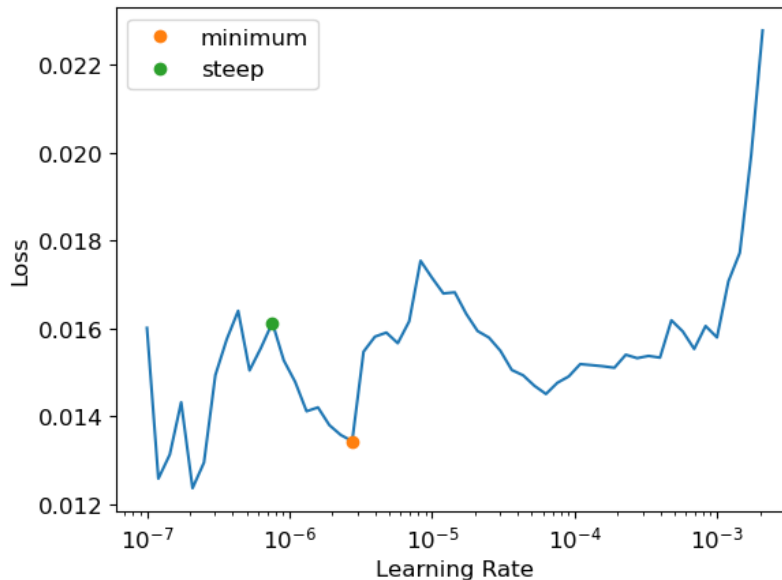


Figure 63: Suggested learning rate by FastAI learner.

When dealing with single-label classification, the suggested learning rate would be in-between the minimum and the steepest point of the loss. This is not the case for a multi-label classifier, as this point indicates a value that is way too low, resulting in small steps during the training, low convergence (step size), slow learning and a bad model. The perfect point for the learning rate would be just when the loss keeps climbing, somewhere between 10^{-4} and 10^{-3} . A learning rate of $3e-3$ is selected for the fine-tuning of the model as shown in Table 26.

Table 26: Calling the learn fine tune function with num epochs, learning rate, and freeze epochs parameters.

```
learn.fine_tune(7, base_lr=3e-3, freeze_epochs=4)
```

5.1.2.5 Validating base model

As seen in Figure 64, the accuracy looks really promising with a baffling score of 99.48%. However, there is one thing to be aware of and that is the threshold defined for the learner. When the threshold is set to low, it will often be prone to select wrong labeled objects, and if the threshold is too high, it will only select objects where the model is very confident [39]. In this first base iteration, the threshold might be wrong, and should be checked before retraining.

epoch	train_loss	valid_loss	accuracy_multi	time
0	0.939007	0.676521	0.619493	00:12
1	0.700046	0.366277	0.873397	00:09
2	0.275587	0.082042	0.971277	00:09
3	0.134296	0.062810	0.977552	00:09

epoch	train_loss	valid_loss	accuracy_multi	time
0	0.076247	0.051901	0.981735	00:13
1	0.065639	0.058618	0.979643	00:13
2	0.052635	0.032294	0.988706	00:13
3	0.040029	0.023979	0.991355	00:13
4	0.029238	0.018087	0.994562	00:13
5	0.019938	0.014984	0.994004	00:13
6	0.015683	0.014193	0.994841	00:13

Figure 64: Multi-label classification base training result.

5.1.2.6 Retraining

Before retraining, there are a few methods to find a better threshold for the accuracy metrics. FastAI library provides a threshold finder that takes the predictions, targets, and a suggested threshold value as input.

```
xs = torch.linspace(0.05,0.98,29)
accs = [accuracy_multi(preds, targs, thresh=i, sigmoid=False) for i in xs]
plt.plot(xs, accs);
```

After iterating through each threshold in the “xs” linspace, a somewhat smooth curve is represented. This smooth curve Figure 65 indicates that all values between 0.2 and 0.95 should give a decent accuracy score, and picking one of these values will not (in theory) overfit the validation set. Choosing 0.445 as accuracy threshold might give the best accuracy score, but since the goal is to only label the categories that the model is confident about, a higher accuracy threshold of 0.8 is chosen. One important part and reason for choosing this 0.8 as threshold is the smoothness of the curve at this point. Choosing a point that is not within the smoothness of the curve can result in choosing a hyperparameter prone to outliers, thus resulting in false accuracy and overfitting to the validation set [39]. So, choosing 0.445 as the threshold might not be a good idea after all.

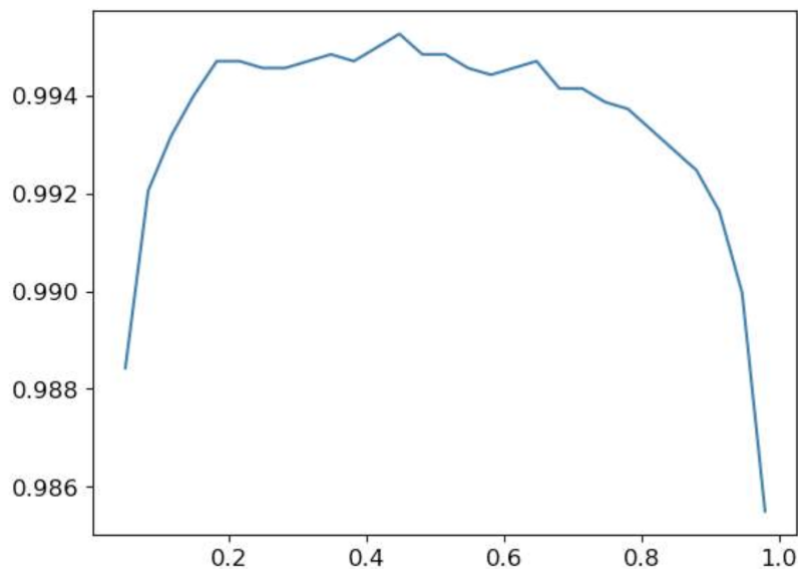


Figure 65: Accuracy threshold plot.

Define a new learner with the new parameters and retrain. The final code snippet is provided in Table 27.

Table 27: Defining learner and start fine tuning the model.

```
learn = vision_learner(dls, resnet50, metrics=partial(accuracy_multi,
    thresh=0.8))
learn.fine_tune(7, base_lr=3e-03, freeze_epochs=4)
```

5.1.2.7 Validating the result of the retrained model

The training result shown in Figure 66 scores almost the same as the base model. An accuracy of 99.33% is 0.15% less than the base model with a score at 99.48%. The important thing about the retraining result is that the model is more confident in its classification, and there is less chance that the model was overfitted during training. The loss plot shown in Figure 67 also indicates that there is no reason to believe that the model overfits. Both the training and validation loss keeps dropping until it flats out during the last two training epoch. There is no confusion matrix developed and available in the FastAI library for this multi-label classification, and no reason for creating a custom function for this. A snippet of the validation batch can be seen in Figure 68 where all the validation objects are correctly classified for that batch.

epoch	train_loss	valid_loss	accuracy_multi	time
0	0.928491	0.665094	0.918851	00:10
1	0.703712	0.381671	0.967234	00:09
2	0.279441	0.084621	0.966537	00:09
3	0.134432	0.065587	0.974345	00:09

epoch	train_loss	valid_loss	accuracy_multi	time
0	0.076124	0.050609	0.982153	00:13
1	0.066656	0.046086	0.983268	00:13
2	0.054059	0.035523	0.987591	00:13
3	0.038443	0.022828	0.990937	00:13
4	0.029542	0.019041	0.991495	00:13
5	0.020481	0.013791	0.993726	00:13
6	0.016304	0.013670	0.993307	00:13

Figure 66: Multi-label classification retraining result.

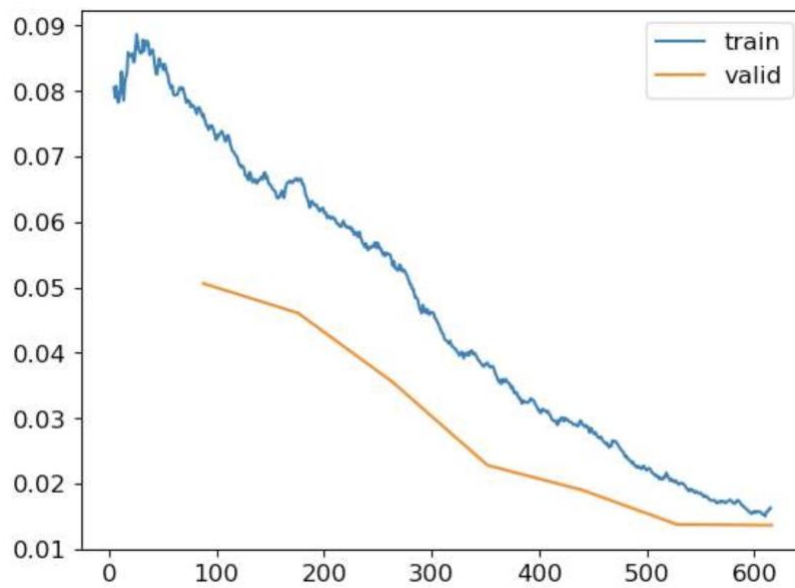


Figure 67: Loss plot multi-label classification retraining.

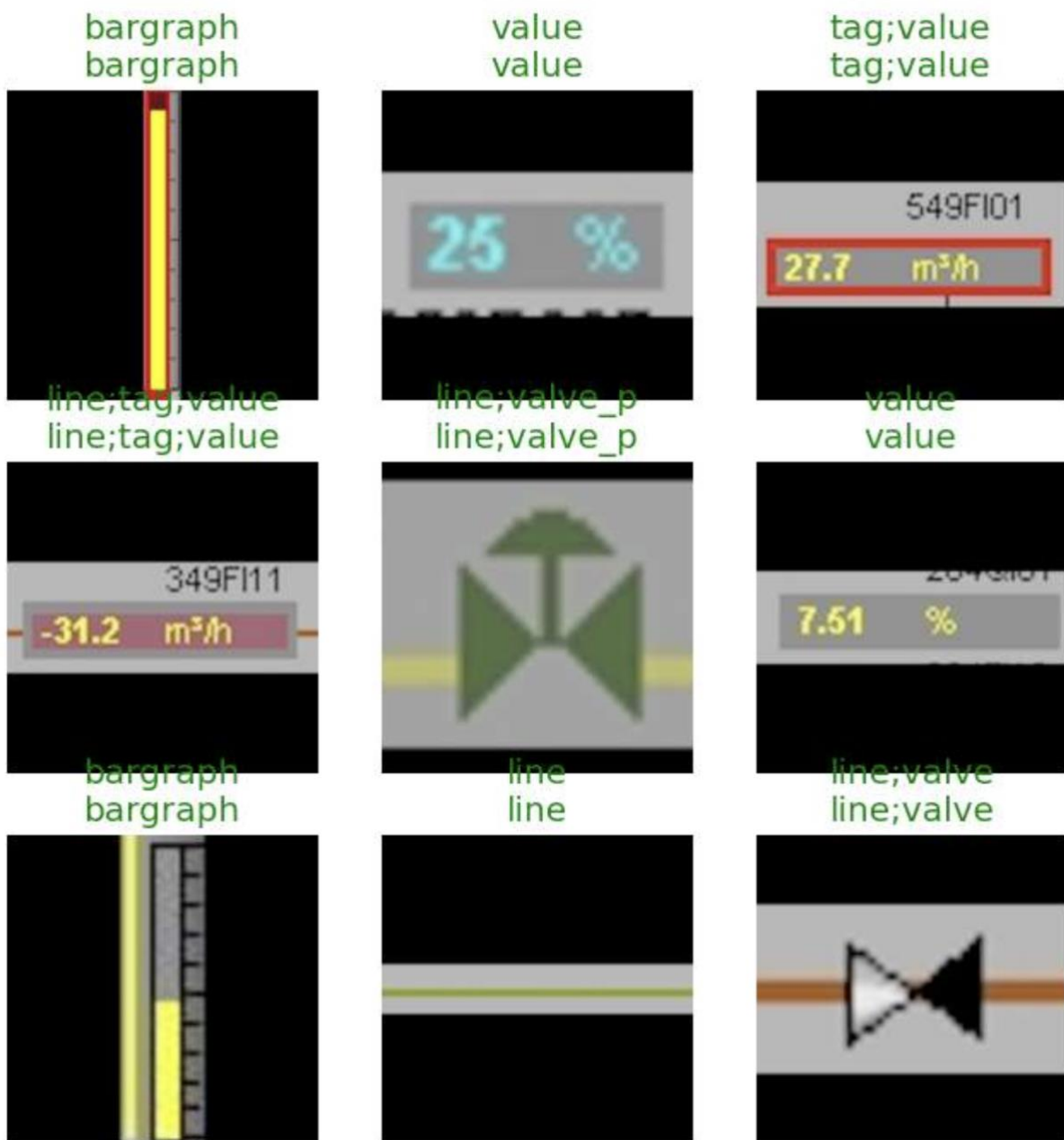


Figure 68: Multi-label classification validation result batch.

5.1.2.8 Exporting and importing model

Exporting a model is quick and easy with the FastAI library. In one line of code the model can be exported as a .pkl file as shown in Table 28.

Table 28: Exporting deep learning model using the FastAI library.

```
learn.export(fname="v2_multiobj_classifier.pkl")
```

Then the exported model can be loaded back in as shown in Table 29, and tested on test sets individually or in batches.

Table 29: Importing a deep learning model using the torch library.

```
learn_inf = torch.load(path/'v2_multiobj_classifier.pkl',  
                       map_location='cuda:0')
```

5.1.2.9 Testing

After loading the model back in as a learner, objects from an unseen test set can be predicted by calling the predict method on the learner. Predictions can be made individually as shown in Table 30 or by loading a dataset on to the learner. Loading large datasets into the learner utilizing the full power of parallel computations on the GPU will be further utilized in the sliding window pyramid approach. The full source-code for this multi-label classification can be found in Appendix F.

5 Result

Table 30: Predicting labels of single object by loading an image into a path, then calling predict method on that object path. The tensorbase shows all predictions in a tensor and their score. By calling the dataloaders vocabulary it will list all the different class labels. The model predicts “line” and “pump_isa” which is correct and fits the tensor number 5 (line) and number 9 (pump_isa) class (counting from 0).

```
num = 42
imP_path = Path('/home/user/git/classify_multi_obj/test/'+str(num)+".PNG")
imP = Image.open(imP_path)
imP

Out:


In:
learn_inf.predict(imP_path)

Out:
((#2) ['line', 'pump_isa'],
 TensorBase([False, False, False, False, False, True, False, False, False,
 True, False, False, False, False, False, False, False, False, False,
 False, False]),
 TensorBase([1.9278e-04, 6.2927e-04, 3.0733e-04, 8.4767e-04, 1.7660e-03, 5.
 1715e-01, 6.3016e-04, 2.0605e-03, 1.0679e-03, 5.2957e-01, 2.9166e-03, 1.847
 8e-03, 3.1291e-01, 1.5407e-01, 8.5539e-05,
           1.4498e-04, 1.1626e-03, 4.3624e-04, 3.6069e-02, 4.6947e-03, 2.
 3830e-04, 1.7925e-04]))

In:
learn_inf.dls.vocab

Out:
['background', 'bargraph', 'chart', 'damper', 'fan', 'line', 'mixer', 'moto
r', 'nav', 'pump_isa', 'pump_iso', 'status', 'tag', 'value', 'valve', 'valv
e_3w', 'valve_h', 'valve_h_3w', 'valve_m', 'valve_m_3w', 'valve_p', 'valve_
pr']
```


5.2 Object detection

When talking about object detection compared to image classification, the main difference is that object detection classifies objects as well as locate them. The objects whereabouts is then indicated in some way using bounding box rectangles or a segmentation mask. This is an important feature as it gives the user, whether the user is a person or an autonomous system, the ability to detect, analyze and perform action. For this project, object detection is important, as the user will have to get a report from the computer analysis on what objects exist, number of objects and location of each object in the picture.

A challenging aspect with object detection is the way training data is obtained. Instead of individually obtained snippets of objects as for image classification, it requires full scaled images annotated with bounding boxes or segmentation masks. The task of annotating a large quantity of images is tiresome, and large companies often outsource such jobs to data labeling services. This is not possible within the timeframes for this project, so a clever approach using the previously trained multi-label classifier with more traditional sliding window algorithm in combination with other computer vision techniques is tested and used as a baseline for a semi-automatic annotation tool.

5.2.1 Sliding window, image pyramid scaling and NMS

The general idea is combining these three computer vision techniques for snipping a large quantity of images (objects) from a picture, storing the snipped image's location, then further classify each snippet using the multi-label classification model derived in chapter 5.1.2.

5.2.1.1 Image pyramid scaling

One major problem using a sliding window algorithm for snipping out objects within a picture is the possibility of RoI misplacement or wrong RoI scale. This is basically selecting a RoI window that is too small for an object that the user wants classified as shown in Figure 69, or that the RoI step size happen do pass in a position where only part of the object is visible as shown in Figure 70.

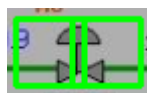


Figure 69: RoI bounding box misplacement and overlapping.



Figure 70: RoI bounding box misplacement an incorrect scale.

This results in RoI snippets that lack information about the objects, and the image classifier will not be able to classify any object within that RoI. Thus, not being able to detect that object within the picture frame.

There are two ways of reducing the chance of this RoI misplacement or wrong RoI scale, and that is to create a pyramid scale of the main picture and/or reducing the RoI step size. The first method is to create scaled version of the original window, where the first scale is the original scale, then it will reduce in size for each iteration of the sliding window algorithm,

keeping the RoI scale the same. On the smaller scaled images, the RoI will appear larger, thus covering more of the picture in each frame, making it more likely to contain larger objects. The implemented code for the image pyramid is shown in Table 31 and an illustration of how the image pyramid is scaled is shown in Figure 71.

Table 31: Image pyramid scale function implemented in python.

```
# Scales the image in given pyramid scales
def image_pyramid(image, scale=1.5, minSize=(128, 128)):
    # yield the original image
    yield image
    # keep looping over the image pyramid
    while True:
        # compute the dimensions of the next image in the pyramid
        w = int(image.shape[1] / scale)
        image = imutils.resize(image, width=w)
        # if the resized image does not meet the supplied minimum
        # size, then stop constructing the pyramid
        if image.shape[0] < minSize[1] or image.shape[1] < minSize[0]:
            break
        # yield the next image in the pyramid
        yield image

# Initialize the image pyramid
pyramid = image_pyramid(image, scale=PYR_SCALE, minSize=window_size_sq)
```

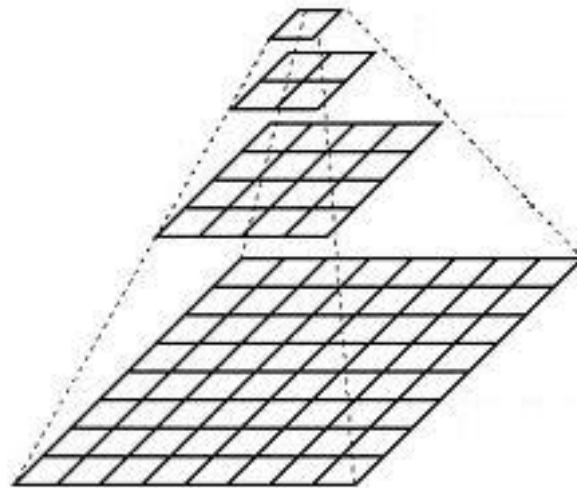


Figure 71: Illustration of image pyramid scale, borrowed from [78]. A single square could be thought of as the RoI window sliding over each scaled image in the pyramid.

The second method, decreasing RoI step size usually just decreases the chance of RoI misplacement, as there is no difference in RoI scale. The step size is defined in the sliding window algorithm in the next chapter 5.2.1.2. By reducing the step size, the sliding window algorithm will pick out more image snippets from the original picture, and therefore having to classify more images, resulting in a slower algorithm, more computing power, and prolonged

execution time. It will also result in objects being classified multiple times, causing more bounding boxes and noise. These are the main benefits and downsides to using such techniques as pyramid scaling and reduced step size.

It is important to note that the sliding window snipping algorithm uses the CPU, and the image classification uses the GPU computation. The more pyramid scaled samples of the original picture and smaller RoI step size will result in more snippets and more hardware computations.

5.2.1.2 Sliding window algorithm

The sliding window algorithm is basically a preset sized RoI iterating over a picture at a preset step size as shown in the defined function in Table 32. Each step will perform a frame snippet that snips and stores the RoI. These snippets are stored as pixel values in an array as seen in the for-loop execution in Table 33. When the sliding window algorithm has iterated over the all the pyramid scaled versions of the picture, the preloaded multi-label classification model is used to classify all sippets stored in the array. The loading of the array into the model and predict call is shown in Table 34. Finally, the predictions need to be split into different arrays for further post processing and drawing bounding boxes on the original images as shown in Table 35.

Table 32: Sliding window function.

```
# Moves sliding windows
def sliding_window(image, step_size, window_size):
    for y in range(0, image.shape[0]-window_size[1], step_size[1]):
        for x in range(0, image.shape[1]-window_size[0], step_size[0]):
            yield (x, y, image[y:y + window_size[1], x:x + window_size[0]])
```

Table 33: Code execution using a for loop for each image in the pyramid to slide and snip RoI.

```

rois = []
locs = []
# Classify the image
for image in pyramid:

    # determine the scale factor between the *original* image
    # dimensions and the *current* layer of the pyramid
    scale = W / float(image.shape[1])

    for (x, y, window) in sliding_window(image, step_size, window_size_sq):
        if window.shape[0] != window_size_sq[1] or window.shape[1] !=
window_size_sq[0]:
            continue

        # scale the (x, y)-coordinates of the ROI with respect to the
        # *original* image dimensions
        x = int(x * scale)
        y = int(y * scale)
        w = int(window_size_sq[0] * scale)
        h = int(window_size_sq[1] * scale)

        # Resize the window to the size expected by the model
        window = cv2.resize(window, (224,224))
        window = np.array(window)
        rois.append(window)
        locs.append((x, y, x + w, y + h))

```

Table 34: Loading all the data into the multi-label classifier model for prediction.

```

# Load data and predict using the Multi-label classifier model
test_dl = learn_inf.dls.test_dl(rois)
preds = learn_inf.get_preds(dl=test_dl)

```

Table 35: Splitting the predictions into separate arrays for post processing and bounding boxing on the image.

```
# Creating dict, splitting predictions into labels, scores and region
arrays
labels = learn_inf.dls.vocab
label = []
score = []
classified_regions = []
for i in range(len(preds[0])):
    x1,y1,x2,y2 = locs[i]
    label = (labels[preds[0][i].argmax()]) #.argmax orig
    score = (preds[0][i].max())
    classified_regions.append((x1, y1, x2, y2, label, score))
```

5.2.1.3 Non-maximum suppression

The theory behind NMS is explained in chapter 4.4. It is an important step for this sliding window pyramid scale classifier as the result from the pure classification is messy, containing thousands of elements. The messy result before post processing with NMS is shown in Figure 72.

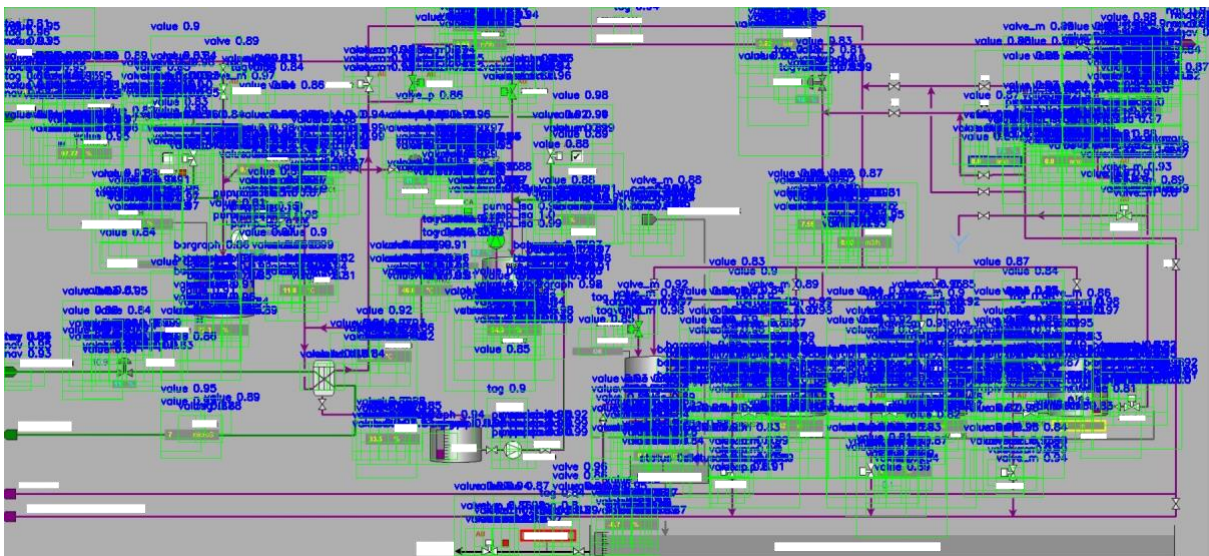


Figure 72: Result of sliding window pyramid scaling multi-label classification algorithm without any NMS post processing.

Now, the solution to this noisy output is to implement NMS to suppress overlapping bounding boxes that have low scores as shown in Figure 73. Traditional NMS can be ruthless when it comes to suppressing overlapping boxes, so a method called Soft NMS [79] is tested. Soft NMS works similar to regular NMS, but it scores each individual overlapping bounding box with a weight depending on how much it overlaps the box with the highest score. So, all bounding boxes are kept during suppression and contribute to the final solution, but the boxes with a lot of overlap compared to the one with the highest score is penalized with a worse score. The final threshold decides what boxes to keep. The advantage of Soft NMS is that it is

more forgiving than plain NMS and keeps more bounding boxes in this case. A problem with the result shown in Figure 73 when applying Soft NMS, is that since this is a custom solution, and there are no ground truth boxes telling the algorithm what should be where, the NMS algorithm suppresses all boxes that are below a certain score, without taking the label into account. So overlapping boxes of different label might suppress each other. This can also be seen from the code snippet provided in Table 36 as it only takes the bounding boxes and the scores as input.

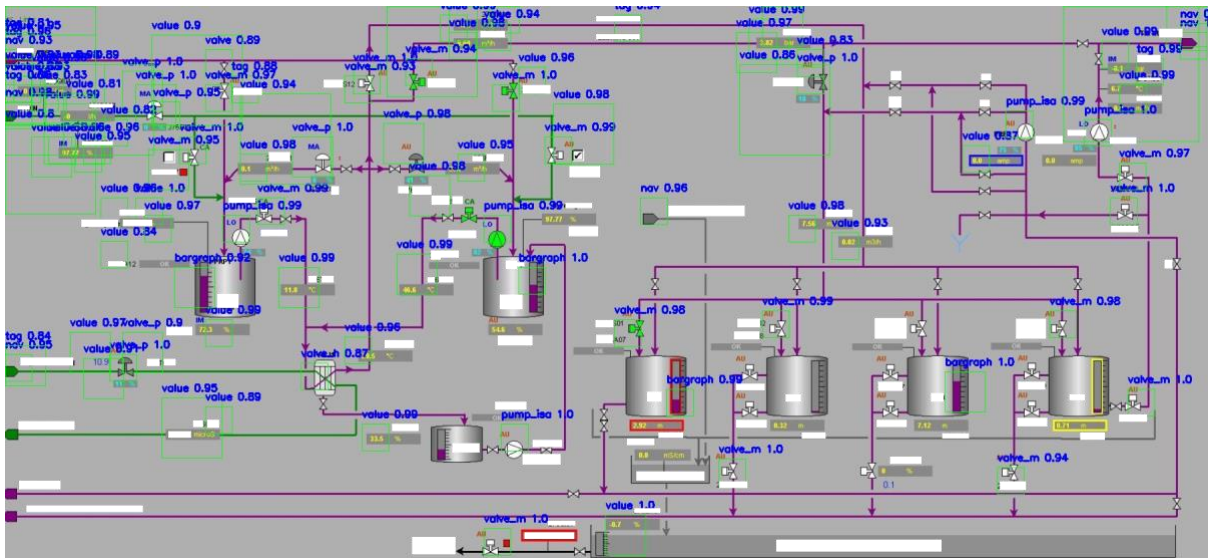


Figure 73: NMS post processing of object predictions.

Table 36: Defining a function for NMS to modify the data to fit the cv2 softNMSBoxes method.

```
# Used for NMS, merging/removing overlapping boxes with low score
def merge_bounding_boxes(bboxes, scores, scoreThreshold=0.1,
nms_threshold=0.1):
    # create a list to store the indices of the bounding boxes to keep
    keep = []
    # Convert the bounding boxes to a format that can be used by the
    # cv2.dnn.NMSBoxes function
    #bboxes = [box.astype("int") for box in bboxes]
    bboxes = [np.around(box).astype("int") for box in bboxes]
    # use the cv2.dnn.NMSBoxes function to suppress overlapping bounding
    # boxes
    scores = np.array(scores, dtype="float")
    indices = cv2.dnn.softNMSBoxes(bboxes, scores, scoreThreshold,
nms_threshold)
    # keep the indices of the bounding boxes that were not suppressed
    for i in indices:
        keep.append(i)
    # return the indices of the bounding boxes to keep
    return keep
```

5.2.1.4 Labeled non-maximum suppression

This is a method invented for this specific object detection case during this project. The labeled non-maximum suppression is based on the same principle as the standard Soft NMS, with a small modification in the first iteration of the algorithm. Instead of sorting all bounding boxes based on the confidence level, it sorts each labeled bounding box separately. In the first iteration it only looks at e.g., “pump” labeled bounding boxes, and in the next one e.g., “valve” labeled bounding boxes. This is achieved in a for-loop, iterating over the obtained bounding boxes. The altered Jaccard index formula is shown in equation (12).

$$J_{label}(A_{label}, B_{label}) = \frac{|A_{label} \cap B_{label}|}{|A_{label} \cup B_{label}|} \quad (12)$$

The code provided in Table 37 first sorts the boxes and scores by labels, then iterate through each label class performing NMS and finally writes the result to a copy of the original image. This code snippet utilizes methods from the OpenCV library for all advanced tasks such as soft NMS, bounding box drawing and putting the labels on the image. It is important to be aware that there exist a lot of open-source packages and solutions that are available for free, that will help improve efficiency and quality of your projects. The final result of the labeled soft NMS is shown in Figure 74. Full source-code for this sliding window algorithm is available in Appendix G.

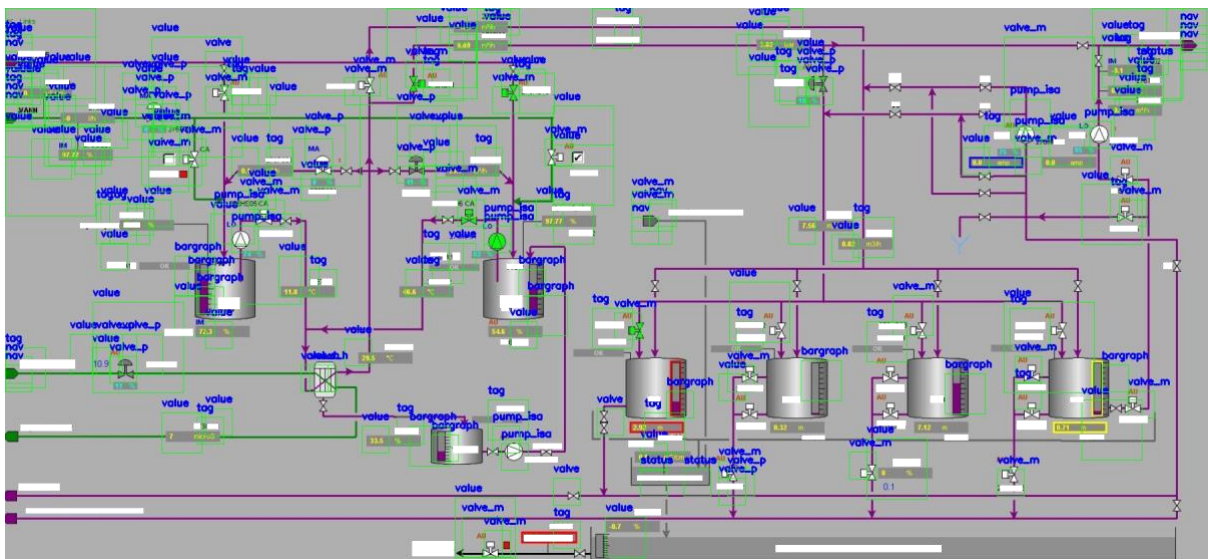


Figure 74: Labeled soft NMS post processing result.

Table 37: Sorting the boxes and scores by labels. Then performing NMS on each label class individually. Final for-loop writes the results to a copy of the original image to provide the final bounding box result.

```
# Group the boxes by label
grouped_boxes = defaultdict(list)
for box, label in zip(boxes, labels):
    grouped_boxes[label].append(box)

# Group the scores by boxes
grouped_scores = defaultdict(list)
for score, label in zip(scores, labels):
    grouped_scores[label].append(score)

# Perform NMS on each group
copy5_image = orig.copy()

for label, boxes in grouped_boxes.items():
    result = []
    scores = []

    #scores = [0.95] * len(boxes) # score of each box, set to 1.0 for
    #simplicity
    scores = grouped_scores[label]
    scores = np.array(scores, dtype="float")
    indices = cv2.dnn.softNMSBoxes(boxes, scores, score_threshold=0.1,
        nms_threshold=0.1)

    #print(scores)
    result.extend([boxes[i] for i in indices[1]])
    for (x1, y1, x2, y2) in result:
        cv2.rectangle(copy5_image, (x1,y1), (x2,y2), (0,255,0), 1)
        cv2.putText(copy5_image, label, (x1, y1-10),
            cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255, 0, 0), 2)
```

5.2.1.5 Mean average precision score and time efficiency

Since this method for object detection is custom made, there are no “out of the box” method for validating how good it performs. By utilizing the mean Average Precision (mAP) method used for more modern object detection methods such as R-CNN and YOLO, there should be a possibility to check accuracy of this sliding window approach. The mAP will be calculated using a human annotated image as ground truth, and comparing the predictions made by the custom object detector.

“The mAP is calculated by finding Average Precision(AP) for each class and then average over a number of classes.” – Deval Shah [80]

It is calculated based on metrics for precision, recall and IoU. The mathematical expression for mAP is shown in equation (13), where n is number of classes, k is class, and AP is the average precision of k class. The code for performing this calculation is added as Appendix

L. The general idea behind creating a custom mAP is to load the manually annotated text file and the predicted text file into a script, where a couple of functions is defined to perform these calculations.

$$mAP = \frac{1}{n} \sum_{k=1}^n AP_k \quad (13)$$

As expected, this sliding window approach scores terribly. The resulting average mAP score of 5 individual images was 11.36%. From the results shown in Figure 74, it is clear that the fixed window size rarely fits the actual object, so the IoU score, and confusion matrix calculation will be bad for almost all objects. Even if the prediction method can give a reference to what and where an object is located, it is not able to bound that object with precision. This could have been improved by combining bounded boxes of predictions with a certain score while performing NMS, but this will not be tested since there exist more modern approaches to object detection that are more interesting to investigate. This method is also inefficient when it comes to time, as it takes about 1 minute to analyze and detect objects for one image. This is of course dependent on the system hardware.

5.2.2 Semi-Automated annotation tool (Program 1)

As previously mentioned, the image classification with sliding window algorithm is suboptimal because it is slow, prone to errors and computational heavy to perform. This is because the attempted use of image classification in combination with computer vision techniques to create an object detector is not optimized for the given task. There are more modern solutions designed specifically for object detection in large images such as one-stage or two-stage detectors. The problem with these methods is the obtaining of training data for such models, as it requires annotation of large quantity of complex images. This then, requires software for annotation and is considered boring, time-consuming manual labor. However, in the first iteration for this project, a sliding window algorithm in combination with a multi-label image classifier was created. Now, wouldn't this serve as a great tool for doing a lot of this manual annotation? Of course, since this is a sub optimal object detection it will require a hardware savvy computer, and all the annotation will require some manual modifications before used as training data for a one-stage or two-stage detector. This was the initial inspiration and fundamental idea for the semi-automated annotation software (Program 1) that is developed during this project and tested in the following subchapters.

The software is designed with some basic concepts from the Unified Process methodology. Mainly focusing on the elaboration phase with design and documentation as shown in chapter 4.5.1. The testing during construction is done iterative as the main focus is to get a working software with the essential requirements as quickly as possible. This software serves a greater purpose as an engineering tool for the next phase of the project, and it is estimated that it will take an engineer approximately 10 minutes annotating a complex image using this tool compared to 40 minutes with a regular tool. The estimated time of 30 minutes saved can be attributed to the fact that many objects are pre-bound before the engineer starts adjusting, but more significantly, all label classes are already defined within the image, which can be easily replicated for any objects requiring modifications or those that may have been overlooked by the algorithm.

5.2.2.1 Program 1 - Testing

The software is developed using the Python language and packages such as, “ipywidgets” and “tkinter” for GUI components such as message box prompts and file dialogs. It is implemented with around thousand lines of code in three separate files. The source code is available in Appendix H. The result is a piece of software that does the following:

Launching the software brings up a Main Menu window shown below in Figure 75. The user will start with uploading an image by clicking the “Upload Image” button, then be prompted with a folder window shown in Figure 76.

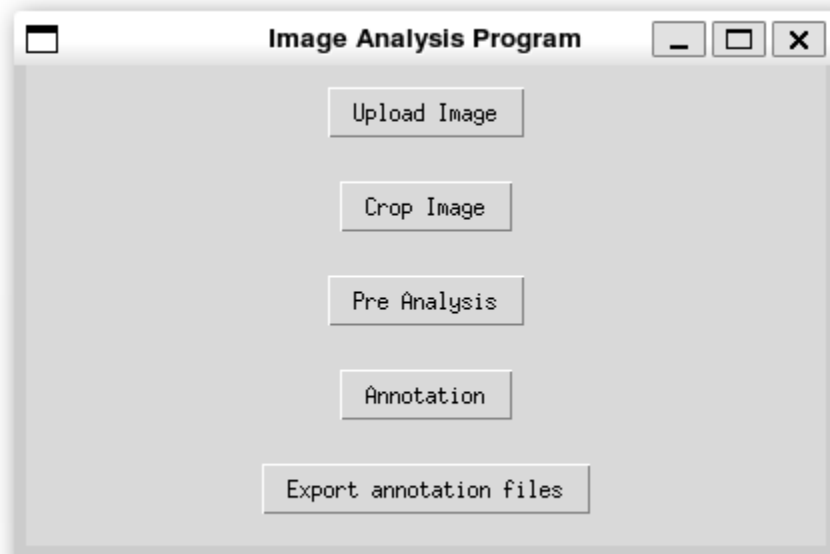


Figure 75: Main menu display.

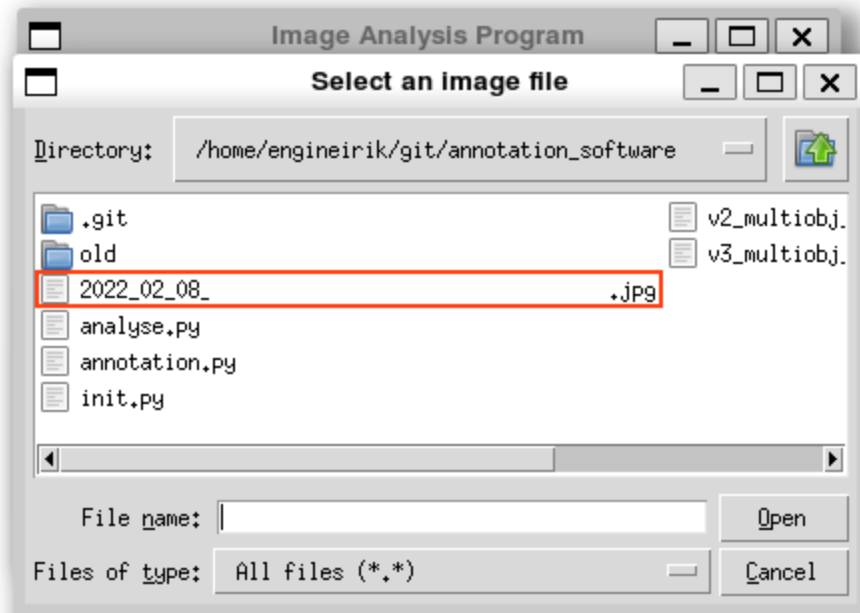


Figure 76: Upload image file prompt.

After uploading an image, the user can perform some cropping action by clicking the “Crop Image” button. A separate cropping window will pop up where the user can resize the image using the sliders at the bottom of the window as shown below in Figure 77. When the window is exited, a successful prompt is shown as Figure 78.

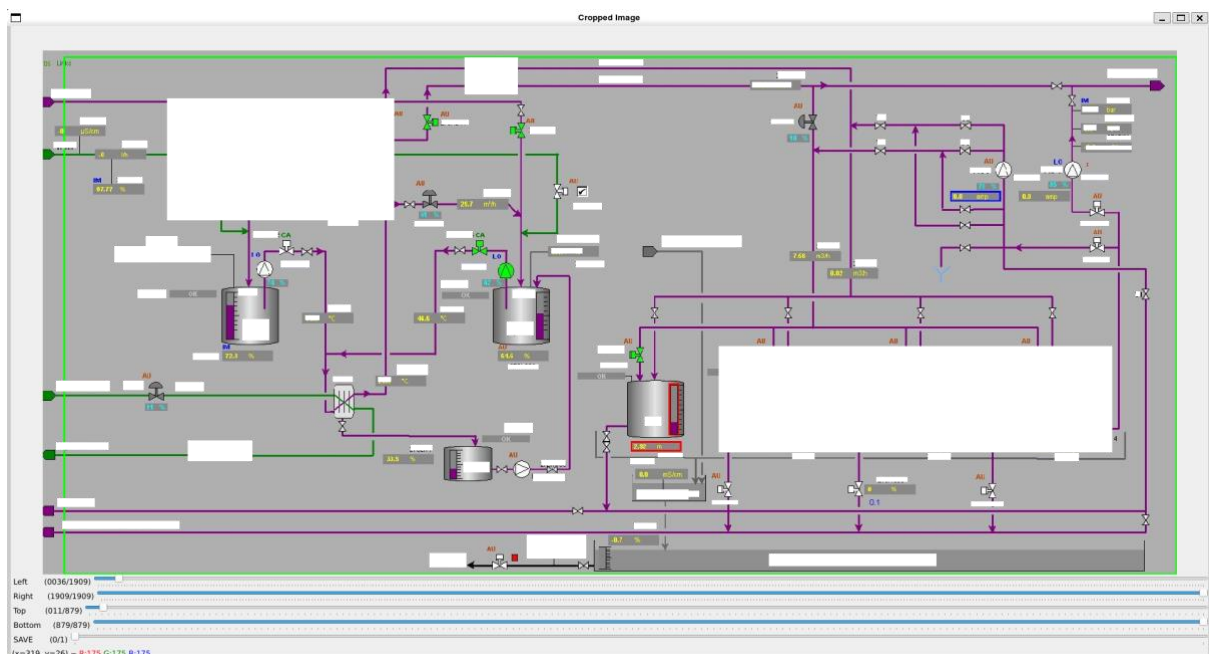


Figure 77: Image crop window.

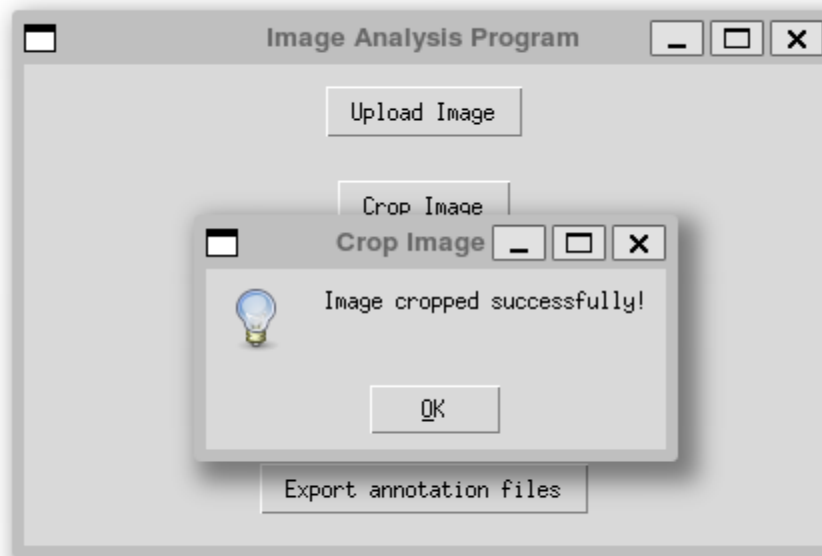


Figure 78: Crop successful prompt.

From the Main Menu, an option to pre-analyze the image using the sliding window algorithm in combination with image classification model trained in chapter 5.2.1 can be selected. The user has the option to upload a custom “.pkl” model file as shown in Figure 79 and Figure 80, that will be used in the classification.

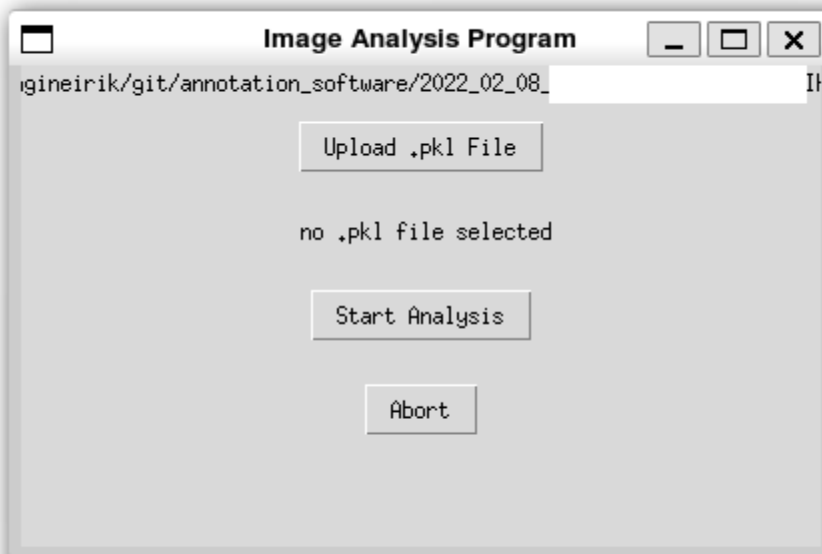


Figure 79: Image analysis display.

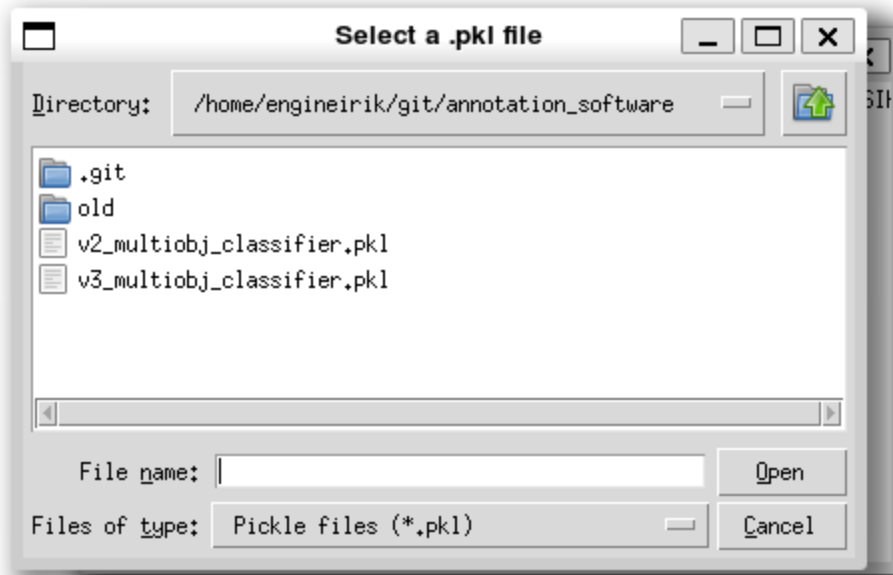


Figure 80: Select classification model .pkl file prompt.

Select model file and click “Start Analysis”. The analysis progress is displayed during the object as shown in Figure 81 and Figure 82.

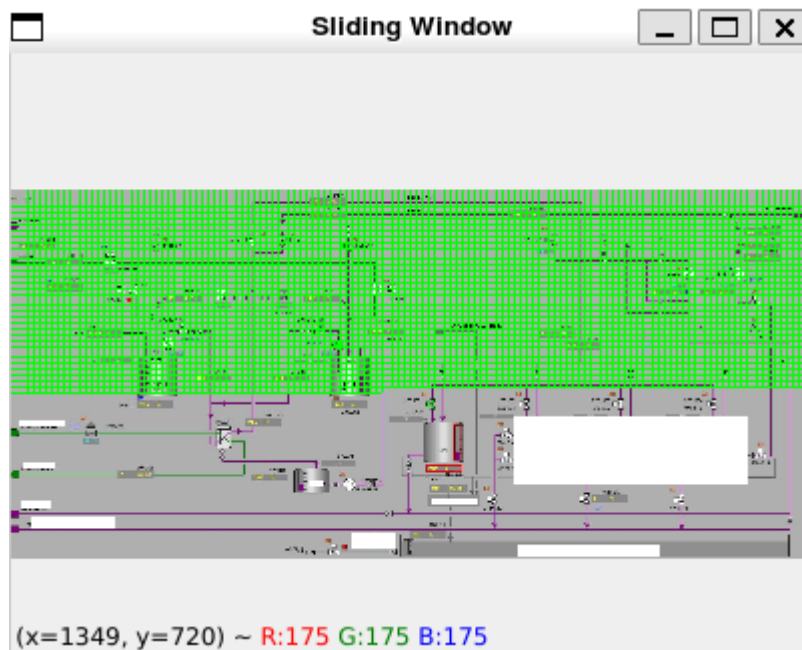


Figure 81: Sliding window progress window.

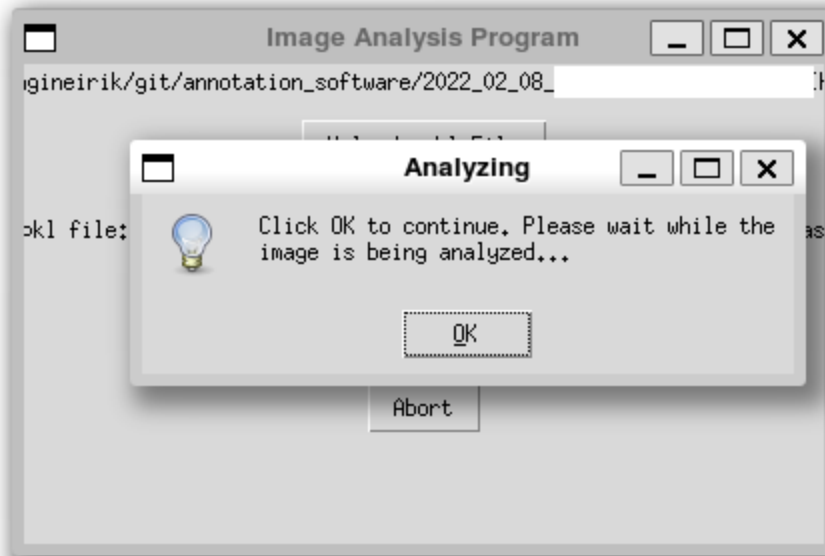


Figure 82: Analysis in progress prompt.

The pre-analysis is further written to a csv file within the software and shown in the annotation window seen in Figure 83, where the user can do modifications, such as delete, add new, edit, and move bounding boxes. An example of adding a new bounding box and enter a label is shown in Figure 84.

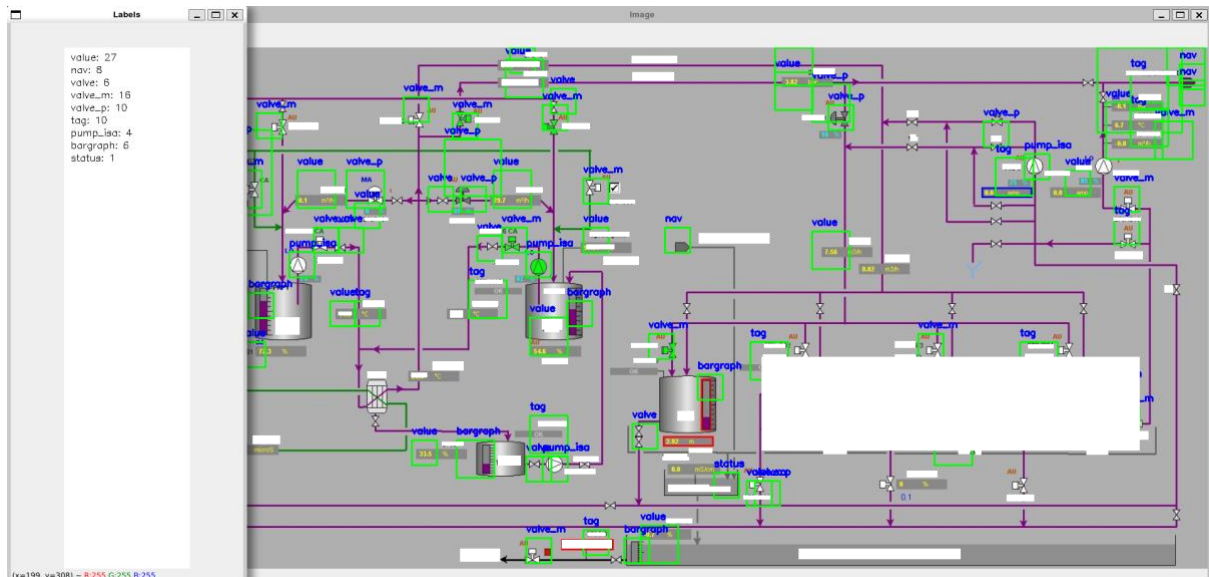


Figure 83: Annotation window.

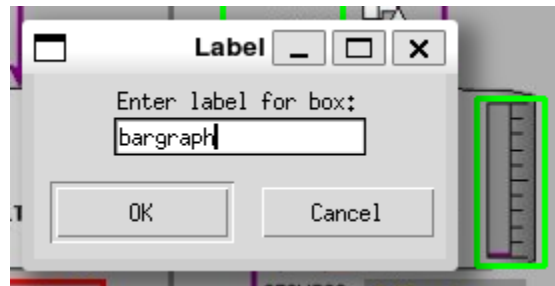


Figure 84: New bounding box or edit label entry prompt.

After quitting the annotation window, the raw image file and annotation (.xml, csv, or txt) file can be exported from the Main Menu window. A folder path request prompt is shown in Figure 85 where the user must specify where these files are going to be exported. If the export was a success, the success prompt is displayed Figure 86.

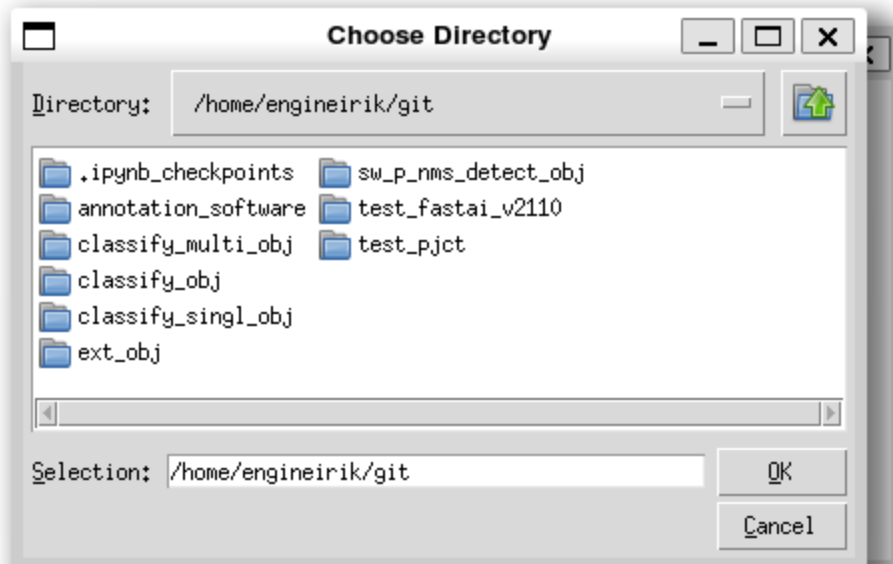


Figure 85: Export annotation files prompt.



Figure 86: Export successful prompt.

This now serves as an annotation software for creating training data for a one-stage or two-stage detection method. A complete manually adjusted annotated image using the annotation tool is shown in Figure 87. This annotation can be fed directly to a one-stage or two-stage detector as training or validation data. An example of annotation xml file in Pascal VOC format listing all bounding boxes and their labels is shown in Figure 88. Pascal VOC format is the annotation formatting often required for object detection models using ResNet architecture. An example of annotation text file in YOLO formatting listing all bounding boxes and label indexes is shown in Figure 89. This is the format required for use in one-stage detection methods such as YOLO. A more elaborated explanation of this YOLO format is provided in chapter 4.3.3. Remember that in YOLO format, the labels are converted to indexes (numbers), so a specification file that lists all labels and their related index is also needed.

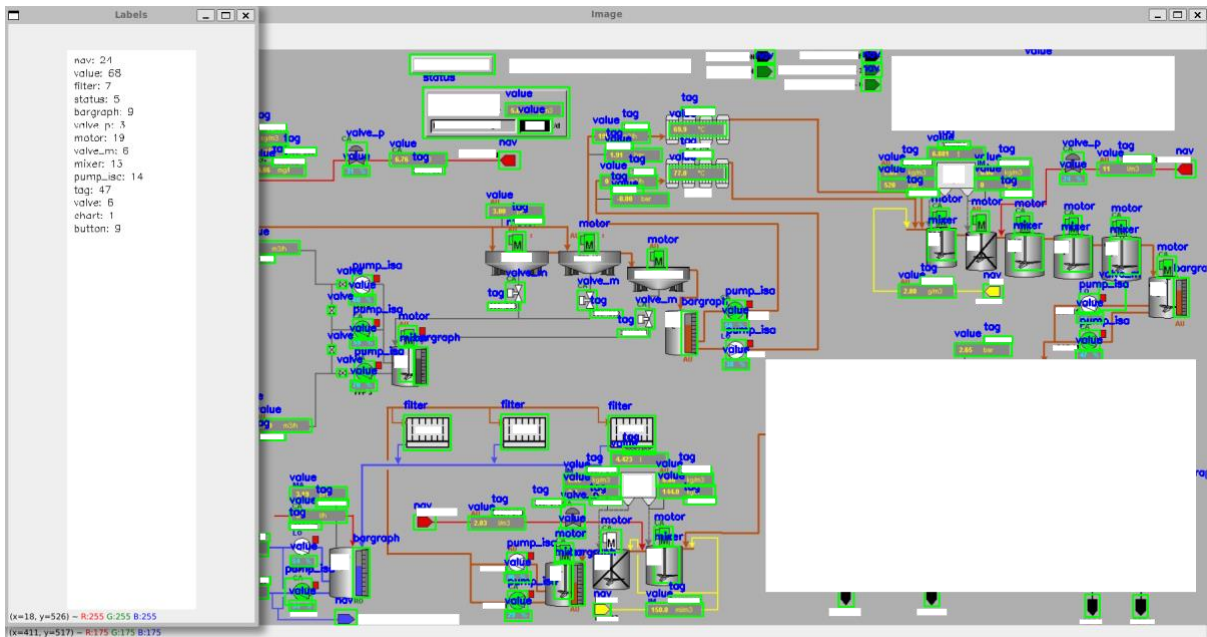


Figure 87: Fully annotated image using the pre-analysis as starting point with manual adjustments.

```

1 <annotation>
2   <folder>/home/engineirik/git/annotation_software</folder>
3   <filename>2022_02_08_<redacted>.png</filename>
4   <size>
5     <width>829</width>
6     <height>1800</height>
7     <depth>3</depth>
8   </size>
9   <object>
10    <name>value</name>
11    <pose>Unspecified</pose>
12    <truncated>0</truncated>
13    <difficult>0</difficult>
14    <bndbox>
15      <xmin>1638</xmin>
16      <ymin>87</ymin>
17      <xmax>1728</xmax>
18      <ymax>177</ymax>
19    </bndbox>
20  </object>
21  <object>
22    <name>value</name>
23    <pose>Unspecified</pose>
24    <truncated>0</truncated>
25    <difficult>0</difficult>
26    <bndbox>
27      <xmin>58</xmin>
28      <ymin>146</ymin>
29      <xmax>148</xmax>
30      <ymax>236</ymax>
31    </bndbox>
32  </object>

```

Figure 88: Snippet of annotation specification .xml file.

```

1 15 0.589444 0.902968 0.665556 0.121005
2 8 0.012222 0.223744 0.024445 0.034247
3 8 0.147778 0.221461 0.026666 0.034247
4 8 0.286666 0.221461 0.026666 0.034247
5 8 0.426666 0.223744 0.026666 0.034247
6 8 0.563334 0.227169 0.026666 0.034247
7 8 0.702222 0.228311 0.026666 0.034247
8 7 0.035556 0.273402 0.024444 0.076484
9 7 0.125556 0.273402 0.024444 0.076484

```

Figure 89: Snippet of YOLO formatted annotation file.

As an additional adjustment to the software, it is added the possibility of uploading pre-annotated images with annotations files into it. This gives the user the opportunity to take previously annotated images and adjust them if needed or even take images where predictions have been made and improve their annotation for further improved training. This is an idea that was thought of during the next step of the project and will be looked more into in the coming chapters. The general idea is that the software checks if there is an annotation file available in the same directory as where the image was uploaded from, and if it already exists a file with the image name and suffix “_annotation.txt” it will load that file into the working csv file of the software and have a preloaded annotation state.

5.2.3 YOLO one-stage detector

The chosen detection method for this project is a one-stage detector network called YOLO. YOLO is short for “You Only Look Once”. More accurately, the YOLOv8 network is selected as it is state of the art, released just a few months prior to this project. More in depth theory of this method can be found in chapter 4.3.2.

5.2.3.1 Preparations

Ultralytics provide a library for YOLO detector methods that is easy to use and easy to implement in software. It can be run directly from the CLI/terminal or as function calls in Python code. The most important foundation to get this working is to have the correct folder structure, annotated data and specification file as shown in chapter 4.3.3 and in example Figure 90. Start by creating a new virtual Python environment and install the Ultralytics library. The installation will automatically also include PyTorch with GPU support if the environment is set up to implement CUDA with Dynamic Linking Library (dll) support. Then create a project and set up the folder structure as shown in Figure 90. Put the annotated training images in the “train” folder, and the validation images in the “val” folder. Remember to include the label map file to the project folder. To automate the process of generating the specification.yaml file, a Python script is created to enter all paths, and list all class labels in correct order, the script is provided in Table 38. The output from the script is a yaml file with paths to the training and validation folder, how many object classes exist, and the class names in chronological order. Within the “train” and “val” folder there are two subfolders that contain images and labels as shown in Figure 44. This yaml file is loaded into the method when calling the train on the YOLO model. A snippet of the specification file for this project (the output of the script in Table 38) is provided in Figure 45.

Name	Date modified	Type	Size
.git	20.03.2023 18:57	File folder	
.ipynb_checkpoints	17.03.2023 16:09	File folder	
runs	13.03.2023 14:02	File folder	
test	02.04.2023 23:30	File folder	
train	20.03.2023 17:49	File folder	
val	20.03.2023 17:49	File folder	
.gitignore	17.03.2023 16:13	Git Ignore Source ...	1 KB
create_yaml.ipynb	20.03.2023 18:56	Jupyter Source File	4 KB
data_custom.yaml	20.03.2023 16:37	Yaml Source File	1 KB
label_map.txt	20.03.2023 16:36	Text Document	1 KB
predict.py	17.03.2023 15:41	Python Source File	1 KB
scripts.txt	13.03.2023 14:56	Text Document	1 KB
train.py	13.03.2023 14:15	Python Source File	1 KB
yolo_tvt.ipynb	20.03.2023 18:56	Jupyter Source File	397 KB
yolov8m.pt	13.03.2023 13:48	PT File	50 897 KB
yolov8n.pt	13.03.2023 13:48	PT File	6 382 KB
yolov8x.pt	17.03.2023 16:15	PT File	133 660 KB

Figure 90: YOLOv8 custom model folder structure for training, validation, and prediction.

Table 38: Script for automate the specification file generation.

```

train_path = os.path.abspath("train")
valid_path = os.path.abspath("val")

# Open the input file for reading
with open("label_map.txt", "r") as f:
    lines = f.readlines()

# Extract the names from each line
names = [line.split()[0] for line in lines]

# Create a dictionary with the names field
data = {"names": names}

# Write the names to the output file as an array
with open("data_custom.yaml", "w") as f:
    f.write("train: " + train_path)
    f.write("\n")
    f.write("val: " + valid_path)
    f.write("\n")
    f.write("\n")
    f.write("nc: " + str(num_classes))
    f.write("\n")
    f.write("\n")
    f.write("names: [")
    f.write(", ".join(f'"{name}"' for name in names))
    f.write("]\n")

```

5.2.3.2 Improve preprocessing of training and validation data

The idea is, since the input data to the YOLO model is of size 640x640 pixels, the training images will be downscaled a lot and padded with zero boundaries. Since the original size of each image is around 1920x1080 pixels and is downscaled by 80.25%, (equation (14), depending on cropping performed during annotation) there is a risk of losing valuable pixel information due to aliasing [81]. This is a general issue that could arise when working with images containing tiny objects with thin contour lines, where “each pixel counts”. Even though the down sampling methods of the Ultralytics library probably utilizes anti-aliasing techniques such as smoothing or blending methods, it still felt like it was necessary to address this issue.

$$100\% - \left(\left(\frac{640}{1920} \right) * \left(\frac{640}{1080} \right) * 100\% \right) \approx 80.25\%$$

(14)

A program for splitting the original annotated image into two separate images and splitting the annotation data in two documents was created. The code for this task is quite extensive and required a lot of checks to make sure that the split of the annotation is put on the correct image and rescaled (normalized) to the new split image sizes. Full source code is provided in Appendix I. The split result without labels is shown in Figure 91.

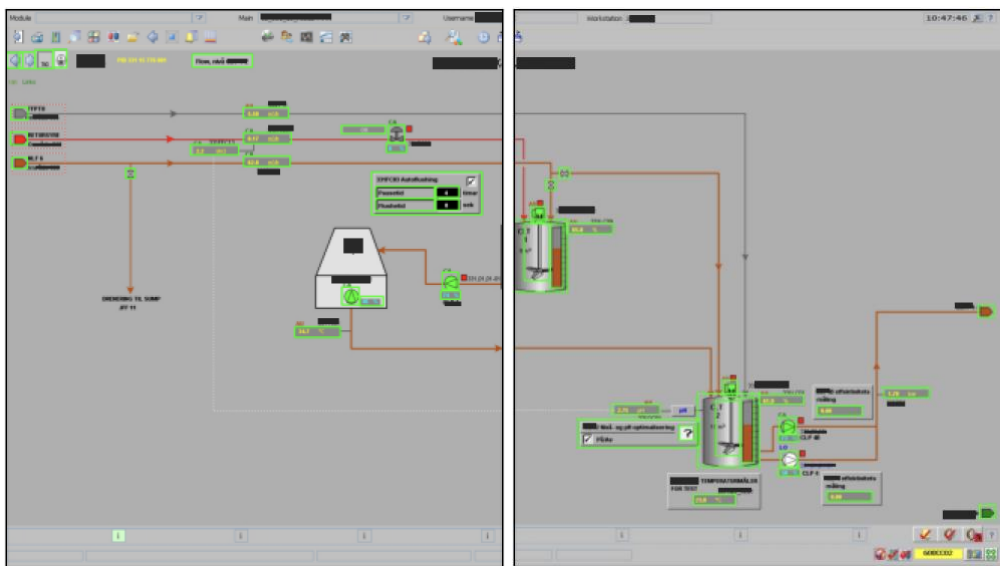


Figure 91: Result of image and annotation split 1 and 2.

Doing this split, automatically double the training data annotated. The prediction images can still be in their original full screen size.

5.2.3.3 Training a base model

Calling the functions for training the YOLO model is executed from a Jupyter notebook script shown in Table 40. First import the library, then import a model, then call the train function with the yaml specification file “data_custom.yaml”. Add additional parameters such as limiting the batch size (to prevent GPU out of memory issues), image input size of 640px, how many epochs to train, and a parameter called patience which basically tells the

function to not stop the training before there has been 100 epochs of no improvement in mean average precision (mAP). The YOLO network also has options for size of model to be used trained. After testing medium, large, and extra-large with the same 21 training and 3 validation images it was clear that the extra-large outperformed the two others. The testing of the three models can be seen in Table 39.

Table 39: Testing different size models. All runs are performed with parameters: patience=150, batch=8.

Runs	Model	Dataset	Epochs	Early stopping	mAP50 score
1	medium	21 train, 3 val	1000 epochs	270 epochs	80.8%
2	large	21 train, 3 val	1000 epochs	388 epochs	83.0%
3	xlarge	21 train, 3 val	1000 epochs	326 epochs	90.5%

The extra-large model is a total of 130 mebibytes (megabytes), so it is quite large compared to the medium of just 51 mebibytes. It is important to remember that a pretrained model is used, so the training function transfer learns the model based on the new data provided in this project.

Table 40: Load model and initialize training of YOLOv8 extra-large model.

```

from ultralytics import YOLO

# Importing the model, medium, large, or extra large
model = YOLO("yolov8x.pt") #m, l or x

# Calling the train method and initialize training
model.train(data="data_custom.yaml", batch=8, imgsz=640, epochs=500,
            workers=1, patience=100)

```

For the first iteration, 12 training images are annotated using the semi-automatic annotation tool developed in the previous project iteration in chapter 0. The images are split to a total of 24 images, where 3 is validation and 21 is training. It is important to select the 3 validation images carefully and make sure they contain a large variation of the data available in the training data. All the training and validation images are from the same site (factory), so they are quite similar when it comes to symbols, layout and colors. The first iteration gave a descent validation result (with respect to the small number of data) of approximately 90.5%. An overview of the training iterations and score can be viewed in Table 42.

5.2.3.4 Predicting the first objects from unseen images

Calling a prediction on a test set is just as simple as calling the training method. The training function stores the best and last model from the training session in a subfolder called “/runs/detect/train/weights”. These models can be directly loaded into a model parameter as shown in Table 41. Then a predict method is called, with some parameter specifications on

how to show and where to store the predictions. All these parameters are listed in the official Ultralytics documentation.

Table 41: Load the best model from training, run a test on images using the predict method.

```
# Specify the path to the best model that was stored during training
nr = 15
best_model = "/home/engineirik/git/yolov8_custom/runs/detect/train" +
             str(nr) + "/weights/best.pt"

# Import the model
model = YOLO(best_model)

# Call the prediction method, specify confidence level and test img path
model.predict(conf=0.3, source="/home/engineirik/git/yolov8_custom/test",
             line_thickness=2, save=True, save_txt=True)
```

A cropped result from the first prediction is shown in Figure 92. The figure clearly indicates that the prediction model works, but there are room for improvements. There are some missing classifications on valves, valves, and navigation buttons. Also, the tag above the value showing 50.8 degrees Celsius is false positive showing two predictions of the same object.

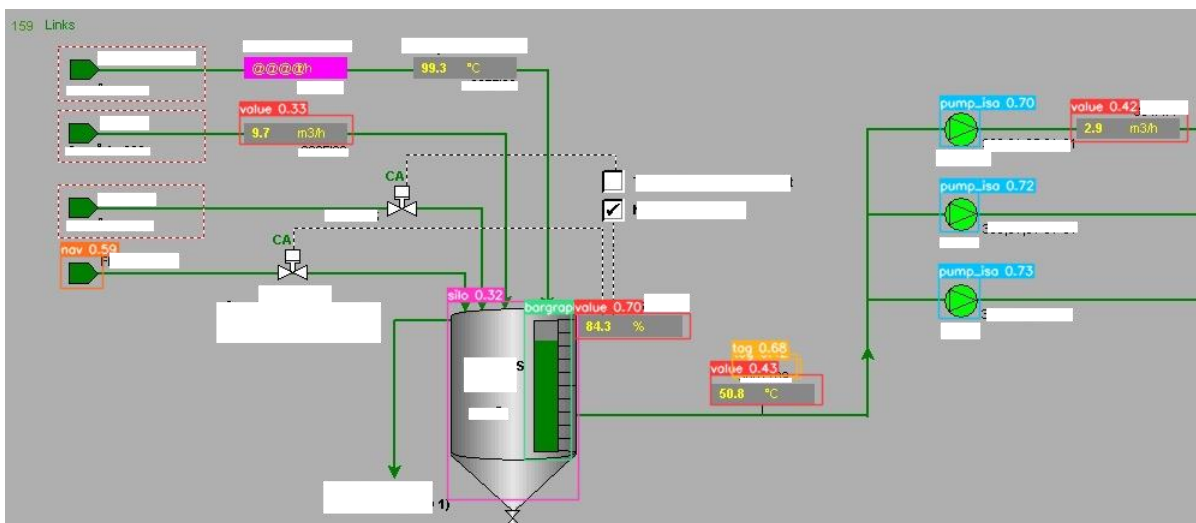


Figure 92: Cropped version of the first predicted test image using the YOLOv8 model.

A notable feature of this approach is that it provides both predicted object images and their associated annotation data in text files, even when making predictions on previously unseen images. This led to the previously mentioned idea for improving the efficiency of annotating more data by modifying the previously developed semi-automated annotation software (Program 1).

5.2.3.5 Adjust and retrain

As mentioned, since the predict method provides the object predicted images with annotation files as shown in example Figure 93, it is easy to take these images, upload them into the annotation software with the annotation data, and perform improvements to the predictions i.e., creating new annotated training data. Note, the test data will now be training data, so new test data needs to be acquired for the next test run.

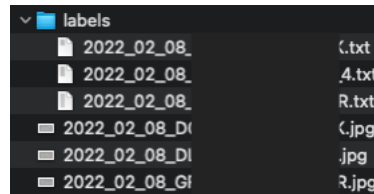


Figure 93: Predicted images from YOLO model with annotation files.

Minor adjustments to the annotation software were needed to check if already existing annotation file exist for the uploaded file selected in Figure 94. This is achieved with a simple “if statement” in the annotation software, and a method that is writing the annotation data from the uploaded YOLO formatted file to the working csv file. The implementation of these changes can be found in the original annotation software source-code provided in Appendix H. Next the user can click the “Annotation” button shown in Figure 75 to make adjustments to the annotation. An example of the prediction made in Figure 92 uploaded to the annotation software with related annotation data is shown in Figure 95.

Do the manual adjustments needed as shown in Figure 96. It is worth noting that the “tag” label is removed from the prediction class dictionary. This is because it caused confusion in the prediction, it was complex to annotated and tag extraction will be handled using optical character recognition (OCR) instead.

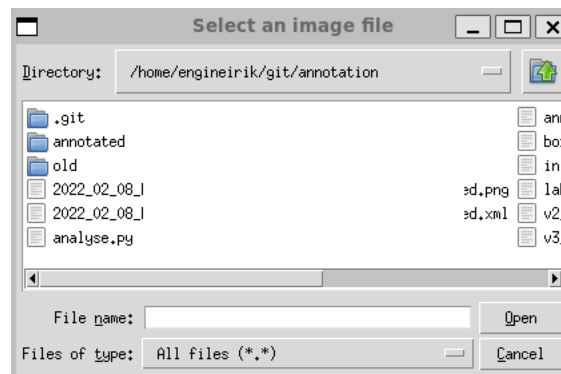


Figure 94: Image file and annotation file in “load image file” prompt in annotation software. The annotation data in this case is .xml as it was converted from .txt to .xml beforehand. This is not necessary in the latest update of the annotation tool (Program 1) as it supports .txt annotation files as well. Remember that .xml is PASCAL VOC format, and .txt is YOLO format. They provide almost the same info in similar formats, so can easily be converted.

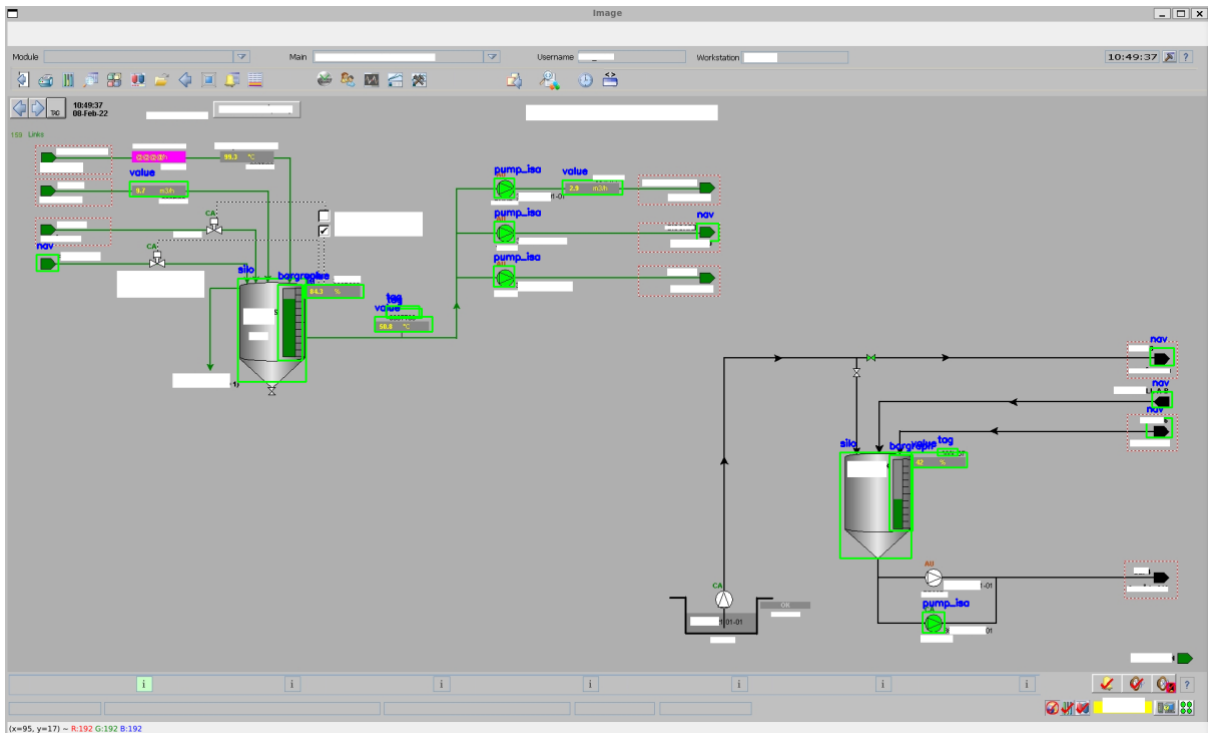


Figure 95: Loaded the YOLOv8 predicted image into the annotation software with the annotation file provided from the prediction method.

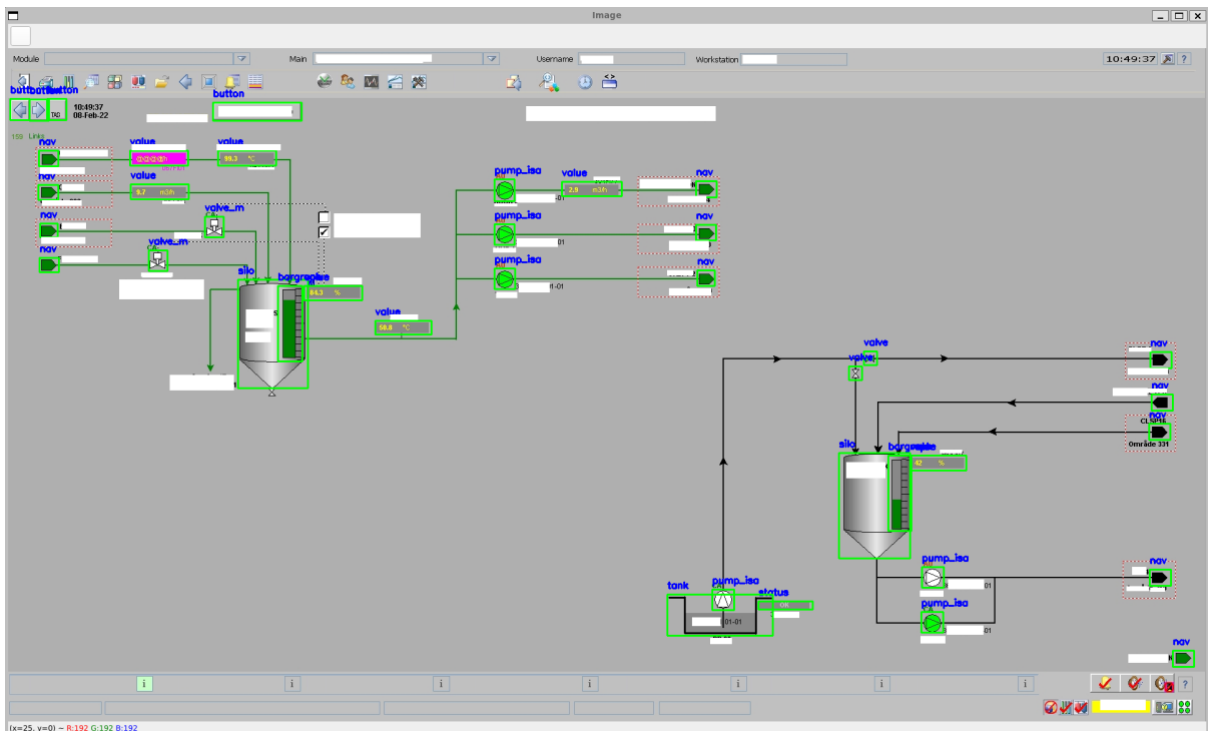


Figure 96: Manually adjusted and improved annotation using the annotation software.

Then the adjusted annotation can be exported, split, and used to retrain the YOLO model to perform even better. Basically, what has happened is that instead of using the semi-automated annotation tool with the sliding window image pyramid scale multi-label image classifier, the

annotation process is improved by using the object detection model (YOLOv8) instead. This whole process could be added as a pipeline to simplify the retraining of the model in a separate annotation tool. This will be suggested as an improvement.

The process of training the model, predicting new images, adjusting annotations, splitting annotation data and images, and retrain the model was done in multiple iterations. After training for a few iterations, the model was tested on a totally new operator interface display images from different sites (factory's) with slight variations in objects, colors, and layout. As expected, the pretrained model did not perform well on these images. This is because the model has been specialized to predict objects from only one site, and therefore are not classified as a generalized model. By annotating and adding some new training data for these new site images, the model performance was improved drastically. At the last iteration of training, a total of 59 training images and 11 validation images was included, the mean average precision (mAP⁵⁰) scored 95.5%. This is where the training iterations stopped as the model result was good enough to continue the project. Arguably, the model could have been trained for 1000 epochs in run 4 Table 42 as the early stopping never occurred. But as shown in Figure 97, the validation result did not really keep improving, so the 1.7% difference in mAP⁵⁰ for run 3 and 4 is likely due to the newly introduced data to make the model more generalized.

Table 42: Iterations of training and validating the YOLOv8 transfer learned model. All runs are performed with parameters: patience=150, batch=8.

Runs	Model	Dataset	Num of factories	Epochs	Early stopping	mAP⁵⁰ score	Note
1	xlarge	21 train, 3 val	1	1000 epochs	326 epochs	90.5%	Tag classes included
2	xlarge	21 train, 3 val	1	1000 epochs	494 epochs	87.1%	Removed tag classes, fixed some errors
3	xlarge	40 train, 8 val	1	1000 epochs	554 epochs	97.2%	Realized non-generalized model
4	xlarge	59 train, 11 val	3	500 epochs	nan	95.5%	Added more data from different sites

5.2.3.6 Final result of training a custom YOLOv8 model

The final iteration of training yielded a satisfactory result of mAP⁵⁰ at 95.5%. The training and validation loss flattens with no significant indication of getting worse or improve, and the mAP⁵⁰ score keeps fluctuating between 90 – 95% as shown in Figure 97.

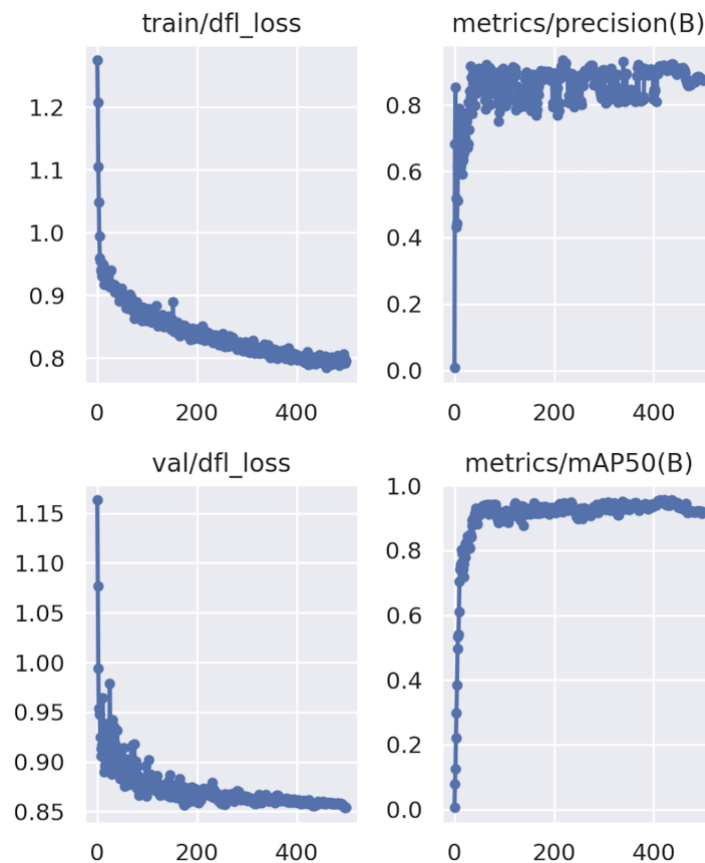


Figure 97: Loss and precision plots for the final training iterations.

The confusion matrix shown in Figure 98 indicates that “damper” is totally misclassified, probably due to the limited numbers of appearance in the data. Also, the normal “valves” are often misclassified as they are quite tiny objects in most occurrences, thus classified as “background”. Also, “conveyors” are often classified as “bargraphs,” and that is not strange due to their similar features. There is a strong correlation between misclassifications and number of instances in the data, Figure 99. Note that “tag” is still in the instance list, just not occurring in any of the data.

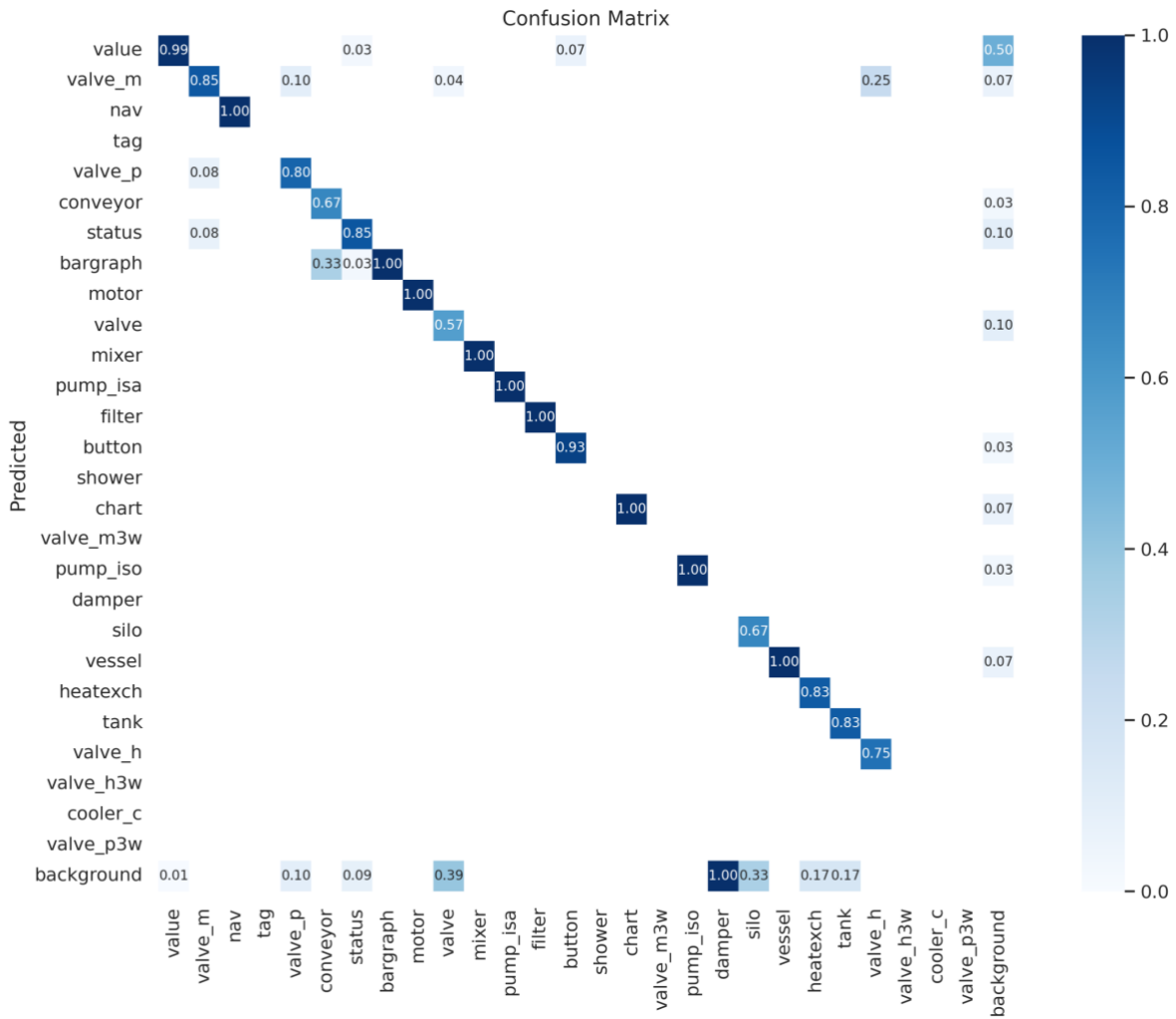


Figure 98: Confusion matrix run 4 of training the custom YOLOv8 model.

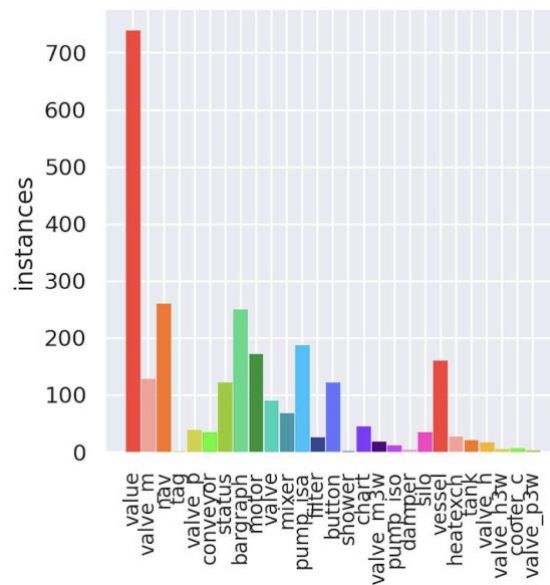


Figure 99: Number of instances in the dataset.

5.2.4 Tag extraction and object linking

To extract the tags from the operator interface images, a technique called optical character recognition (OCR) is used. The library of Pytesseract OCR have methods for using these text recognition methods directly in new applications. A challenge when it comes to the operator interface images is that there are many tags of different formats, and the relative size of the tags is tiny. Some preprocessing steps manipulating scales, contrasts, and colors to get these texts to pop is therefore necessary.

5.2.4.1 OCR – Optical Character Recognition

Often referred to as text recognition. In simple terms, the OCR engine preprocess the image, localize text, character segmentate and recognize, then perform some post processing [19]. The theory behind OCR will not be discussed in detail. The important thing for this project is to only utilize the recognition part of Pytesseract OCR library, and not the post processing dictionary translation. The tags in operator interface images are not of normal words, as they are a combination of different numbers, letters, and symbols. These combinations need to be added in a custom dictionary to filter out all other text combinations. The recognized tags and their position relative to the image needs to be stored in a text file. The position will be normalized between 0 and 1 with respect to the original file size, thus achieving the same formatting as the YOLO annotation.

At first implementation of OCR on the raw image, the OCR engine did not return any valuable information as shown in the left column in Table 43. There could be many reasons why the out of the box solution do not work on these images, but the main guess is that the tags are way to tiny, placement is random, and the colors in the image might not have sufficient contrasts, making it hard to detect them. The solution to these problems was to split the image in four pieces and perform a scale pyramid on each image. Then add some preprocessing steps like gray-scaling, blurring, edge-detection, and dilation. The scale pyramid makes sure that the OCR are applied to different scales of the images and therefore increases chance of localizing text objects. The preprocessed image shown in Figure 100 was then fed to the OCR engine which yielded significantly better result as shown in right column of Table 43.

Table 43: A small part of the returned OCR tag extraction copied from the tag extraction annotation file. Two iterations, before and after manual image preprocessing. Note that the two columns are not related (not same part of image). The left column only returns noise. The right column is selected to match the snipped part of the image that is used as an example in Figure 100.

No image preprocessing: (tag, x_center, y_center, width, height)	Custom image preprocessing: (tag, x_center, y_center, width, height)
2341 0.090365 0.093981 0.003385 0.002315	LSH-1300 0.089461 0.689971 0.030979 0.020058
0201 0.533594 0.083912 0.004687 0.002546	XS-1323 0.183917 0.687645 0.025043 0.020058
0201 0.611393 0.083796 0.003255 0.001852	P-1302 0.258107 0.683430 0.019687 0.023256

21502 0.102148 0.054630 0.005599 0.001852	XS-1325 0.389259 0.680669 0.027215 0.020058
P9212 0.109570 0.096296 0.006120 0.00740	XZ-1302 0.090547 0.757267 0.027649 0.021512

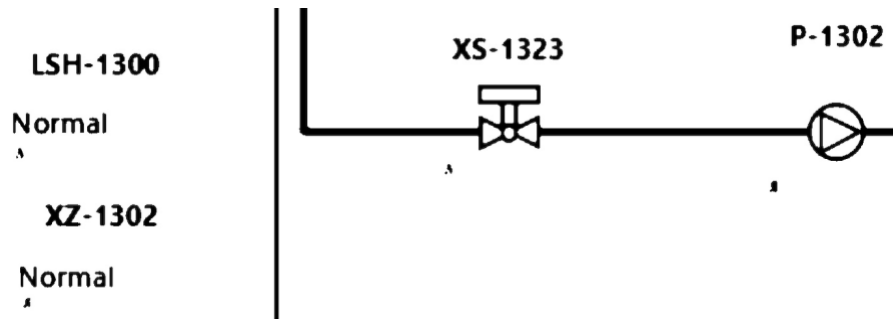


Figure 100: Preprocessed image for text extraction.

The full source-code to this preprocessing and Pytesseract OCR tag detection is available in Appendix J.

5.2.4.2 Minimum Euclidean Distance, linking objects and tags

The idea of tag extraction in combination with object detection is to be able to relate tag and object based on their location. It is fair to believe that tags and objects that are located close to each other would belong together. However, there is some arguments about this generalization such that; Tags can be located far from an object because of status variables on the object that are not visible in the current state of the image. This distance might even be further than the tag's location to an unrelated object. Due to this unknown situation, one tag can be related to many objects in this calculation. It is important to remember that no matter how accurate the object detection and tag extraction is, there will be need for some manual washing after the analysis. Especially on the tag extraction and correlation to objects.

As shown in the previous step, the OCR engine provide the extracted texts with their position converted to a normalized scale 0 to 1 of the original images to match the YOLOv8 object detection location scale. To calculate the distance between center of tag location and object location, a mathematical method called Minimum Euclidean Distance is used. The mathematical expression of this method is shown in equation (15) where γ_{min} denotes the minimum Euclidean distance, x_1 and y_1 is the object coordinates, and x_2 and y_2 is the tag coordinates.

$$\gamma_{min} = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \quad (15)$$

Simply put, the pseudo code methods shown in Table 44 takes the object and tag annotation files as input, then for each object it calculates the proximity of a tag and links the object and tag that is closest to each other. It also checks and excludes objects such as “nav” that is known to not have tags. When a tag is linked to an object, it is removed from the annotation's variable. So, the first iteration finds the object and tag that are absolutely closest, removes it from the list and append it to “linked_objects” parameter. Then the next iteration finds the absolute second closest object and tag, and so on until there are no more unlinked objects and tags. The implementation and source-code can be found in Appendix K, of the final object detection software in the ObjectLinker class.

Table 44: Pseudo code for linking object and tag.

```

def euclidean_dist(x1,y1,x2,y2):
    return mathematical equation (x1,x2,y1,y2)

def linked_object(obj_annot, tag_annot):
    unlinked_obj = copy of obj_annot
    unlinked_tag = copy of tag_annot
    linked_objects = []
    while remaining objects in unlinked_obj:
        filtered_obj = filter out objects with name "nav"
        # Declare variables
        min_dist = large number
        min_obj = None
        min_tag = None
        for object in filtered_obj:
            for tag in unlinked_tag:
                dist = euclidean_dist(x1,y1,x2,y2)
                if dist < min_dist:
                    min_dist = dist
                    min_obj = object
                    min_tag = tag
            if min_obj and min_tag is not empty:
                append min_obj and min_tag to linked_objects
                remove min_obj from unlinked_obj
                remove min_tag from unlinked_tag
    return linked_objects

```

5.2.5 Industrial Component Extraction – ICE (Program 2)

As a last step in this project, a software for hosting the object detection model, tag extraction code and tag-object correlation calculation, is developed. This is the application where the user is going to interact with the product therefore should be developed with best user interface (UI) design practices in mind. A mockup design is drawn with a tool called Figma [82], where all colors, text styles and general design is laid out. The user should be able to upload images, click analyze and get a document containing the performed analysis in return. The software is called Industrial Component Extraction (ICE) software. The entire analysis and design are provided in chapter 4.5.2. This chapter will go through the final testing of the software.

5.2.5.1 Program 2 - Testing

The final result is a user-friendly software for people to interact with as specified by the system sequence diagram. When opening the software, the user is prompt with the initial display where they can upload N number of images from the local machine as shown in Figure 101. A loading screen is displayed during the analysis as shown on left image in Figure 102. When the analysis is done, a display prompting the user to download the analysis document or to perform a new analysis is displayed as shown in right image in Figure 102.

The analysis document has a first page called “Summary” as shown in Figure 103. The summary screen shows an overview of all images that are analyzed, how many components and tags that was detected. For each image analyzed there are two subsequential sheets where the first is an overview of the original image with the bounding boxes for each object detected as shown in Figure 104, and the second data sheet contains a list of each object snipped from the image with object name, tag and localization in the x-y plane as shown in Figure 105. The complete source code is available in Appendix K.

When it comes to performance, putting all the previously mentioned software together (object detection, OCR, minimum Euclidean distance), the most time-consuming part is the image preprocessing before starting tag extraction with OCR. Analyzing 12 images takes around 1 minute and 20 seconds where the object detection only uses around 25ms (on average) for each image. The OCR engine also only capture around 50% of the tags correctly due to the usage of third-party package instead of training and creating a custom model for this task. As explained earlier, this has been attempted compensated with correct preprocessing of the images, where more preprocessing and scales result in improved text detection, but the analysis gets slower. This is a tradeoff between time and precision. Discussion chapter 6 elaborates more around how this could be improved.

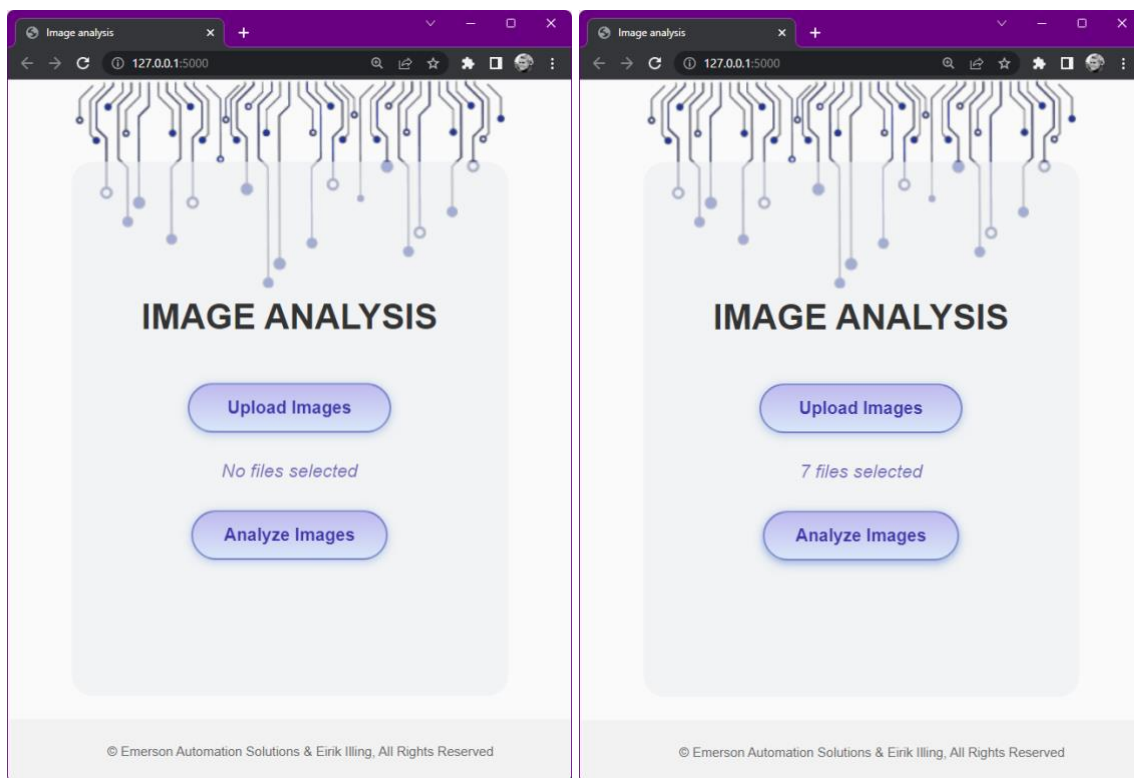


Figure 101: Initial user interface display when opening the software on the left. The user then uploaded 7 images from the local machine as shown on the right.

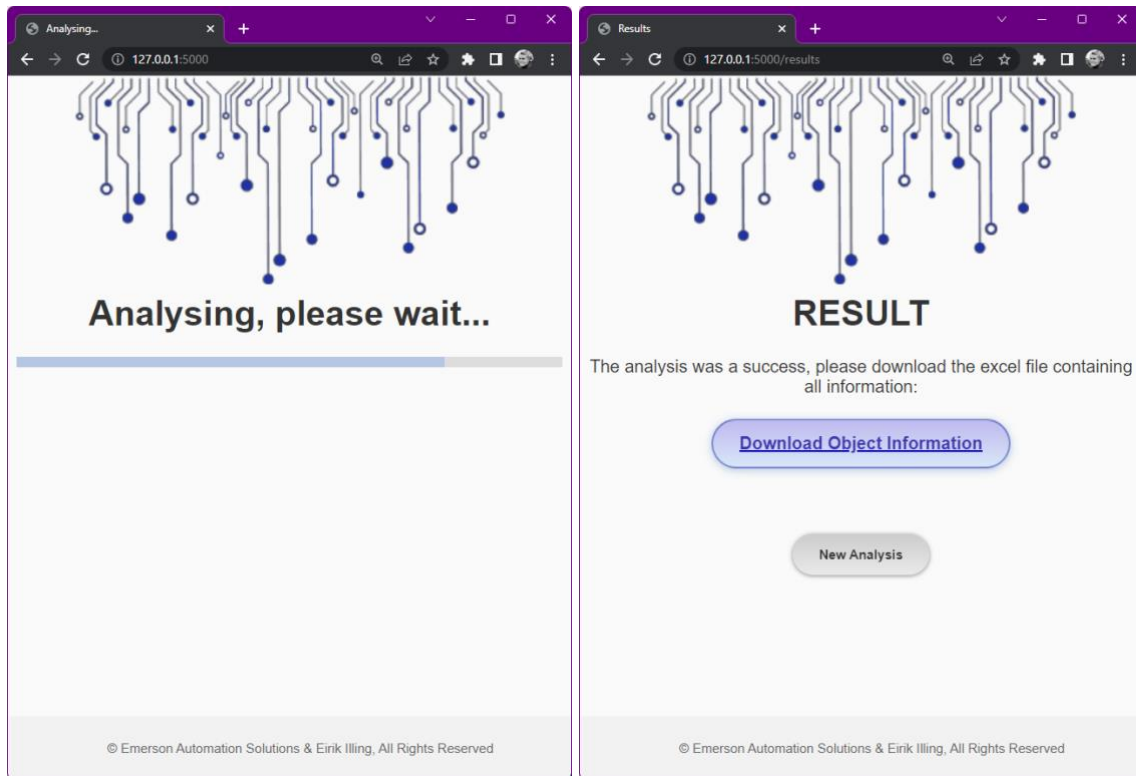


Figure 102: After pressing the “Analyze Images” button, the backend starts to work, prompting the loading display as shown on the left. When the analysis is done, a display letting the user choose to download the result or to perform a new analysis is shown.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
1	Num objec	Num tags	Image	bargraph	button	conveyor	mixer	motor	nav	pump	pump_isa	sil	status	value	valve	valve_m	vessel	
2	104	52	JF_XXX_XX_F	5	10	0	2	7	14	1	7	0	6	30	10	5	7	
3	74	11	KI_XXX_XX_F	2	4	0	0	0	19	1	2	0	1	7	35	0	3	
4	130	57	XXXXXXXX_3i	4	4	1	0	1	18	1	15	3	1	51	1	0	30	
5																		
6																		
7																		
8																		
9																		
10																		
11																		
12																		
13																		
14																		
15																		
16																		
17																		
18																		
19																		
20																		
21																		
22																		
23																		
24																		
25																		
26																		
27																		
28																		
29																		
30																		
31																		
32																		
33																		
34																		
35																		

Figure 103: Summary page of final analysis excel document returned from the ICE software. Listing number of objects and tags for each image and how many of each object type was detected.

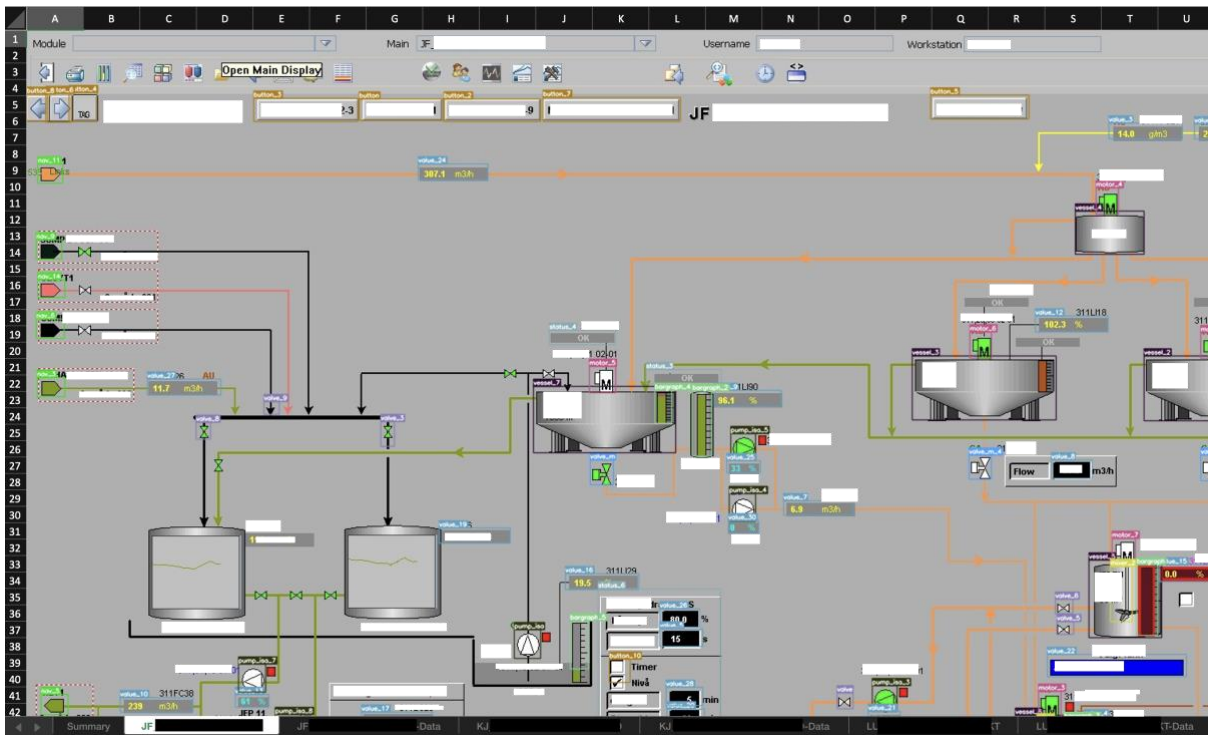


Figure 104: One worksheet for each analyzed original image with bounding boxes for each detected object. The detected objects are bounded by a rectangle and given a label. This figure displays a part of one of the images analyzed. Comparing with the datasheet row 34 shown in Figure 105, the “value_10” object is located in the bottom left corner and bounded with a light blue box. The tag 311FC38 is extracted and linked to the object.

Image	Object	Label	Tag	X min	Y min	X max	Y max
JF_	JF_	_value_8		1282	537	1333	560
JF_	JF_	_value_9	311L190	860	450	944	470
JF_	JF_	_valve_m_2	311HS08	1469	531	1500	563
JF_	JF_	_vessel_7		634	445	812	528
JF_	JF_	_value_10	311FC38	117	834	208	852
JF_	JF_	_value_11	311FC12	1461	117	1554	136
JF_	JF_	_value_12	3	1262	357	1353	377
JF_	JF_	_motor_2	3	1760	377	1787	409

Figure 105: The data sheet of the analyzed image, listing all objects, names and tags with the location x, y, x_max and y_max coordinates on the original image.

6 Discussion

This chapter provide a short discussion revolving the results, comparing the methods in first and second project iteration, and suggesting improvements and further work. A subchapter discussing some “out of the box thinking” is also provided to give some inspiration for further development.

6.1 Comparing sliding window classification and YOLO algorithm for object detection

These two methods of object detection are not easy to compare as one serves as a base line for annotating data, and the other is used for object detection with high accuracy. The initial idea was that the result from each method would provide information that made these equal in some way, and therefore possible to conclude strengths and weaknesses between the two. It is worth noting that the initial idea was thought of in the beginning of the project when knowledge about this technology was limited. As the project progressed, new methods were discovered, providing valuable insights into the advancements made in the field of deep learning image analysis.

It is arguable that starting with the old school image classification in combination with sliding window, pyramid scaling and NMS was a mistake, as the YOLO object detector performed so much better and was easier to implement in the end. Also, if time had been spent on annotating more data for the YOLO algorithm, it would have performed even better and other features could have been added to the final software. However, the first approach provided knowledge about the field, knowledge about object classes that needed to be detected in the custom data, and how to perform full image annotation for the different one-stage and two-stage detectors. It also triggered the idea of creating a semi-automated annotation tool for modern object detection solutions. All in all, it was an important step in the project as it laid the foundation for the result.

6.2 Improvements and future work

Starting with the obvious, the OCR tag extraction method can be improved extensively. Either by creating or using a different method for extracting tags or adding tags as a separate label-class in the object detector. If tags are added as separate class (as tested in the beginning of the project) it will require a lot more training data as tags are the most varying object between sites. The benefit of adding tags as a separate object class would be that each individual tag-object detected could be fed to the OCR engine individually thus hopefully being able to extract more data. The guess why today’s solution performs poorly is because the text is so tiny compared to the other objects and appears quite randomly in the image. The object detection model handles the arbitrary placements quite well, but the OCR does not. The compensating solution to this problem in this project was to create multiple scales and perform OCR on each scaled image, which improved the detection drastically but the more scales and image preprocessing, the more computing power and time will be consumed.

To enhance the final ICE software (Program 2), it would be beneficial to incorporate similar features from the annotation tool, where the user could do some pre-analysis using the YOLOv8 model, then do manual adjustments and retrain the model. This would help improve and generalize the model thereby continuously improving the user tool. The improved software solution could be looked at as a complete pipeline for the user to test, retrain (if necessary) and analyze without interacting with any source-code. The pipeline flow would be as follows:

- User can start by trying to analyze new operator interface images.
- If the analysis is not satisfactory, the user can select a few of the images and do manual improvements on them.
- Then use those manually improved images to retrain the model (transfer learn), to make it fit and adopt to the specific site images. This will also help the model to generalize and learn new objects.
- The user can then retry the analysis on the whole dataset.

This suggested improvement could be added to the engineering tool (Program 1) as well, by substituting the software's sliding window pyramid scale multi-label classifier developed in chapter 5.2.2 with the YOLOv8 algorithm trained in chapter 5.2.3 and adding a model training feature, thus improving the engineering tool. But it would be a lot more interesting to have these features available for all users using the ICE software (Program 2) that will be using this tool to analyze images on a daily basis.

Last but not least, more data. Annotating more images from different sites (factories) would improve the model mAP⁵⁰ score and help generalizing the YOLOv8 model to perform better on a larger variety of images. Also including more object classes and improving class standardization to detect even more objects. This will require more time spent on boring repetitive annotation tasks, but also massively improve the end result. Another option (instead of more training data) could be altering the YOLOv8 network architecture, substituting the classification network with the multi-label image classification model derived in the first iteration of the project. This would require custom network architecture engineering but could end up replacing the need for having to annotated more full-scale images and instead using the already learned features for the single objects.

6.3 Thinking outside the box

It is essential to always leave room for creative and innovative thinking when it comes to technology as it may lead to groundbreaking opportunities and possibilities that might have otherwise gone unnoticed. The technology underpinning this entire project is deep learning, which is an exciting and rapidly evolving field. The developed product in this project creates a foundation for object detection and tag extraction from operator interface images. Now, imagine if this product was used in combination with pixel processing for pipeline detections as discussed by Moon et al. [7] giving a structure and process flow of the image. Then by training a foundation model, preferably a large language model (LLM) on the source-code for the DeltaV Live library, each individual object that was detected in the analysis could map to the object in Live, thus creating a prompt to the LLM. The LLM would then write the entire code for the new Live images. Now, in best case scenario this would create lines and objects placed correctly on images with correct tags entered in the correct configuration fields,

making the need for manual engineering. But now, imagine doing the same for configuration, system control diagrams (SCD's) and P&ID's. Training LLM on configuration, object detection models and text extraction on SCD's and P&ID's (as suggested by Rahul et al. [6] and Paliwal et al. [4]) combining all these methods. Theoretically, it would then be possible to generate all redesigned operator interface images automatically.

A good object detector foundation model created for process graphics can also be used to monitor real-time system images and extract information directly from it without the need of interacting directly with the system logic. This idea can be extremely useful in cases where interfacing with existing communication protocols is not an option. An example of such an application could be old HMI panels controlling machines in process areas where integration or modernization is not an option, so a web camera is placed in front of the HMI panel, monitoring and extracting data to a cloud solution directly.

7 Conclusion

This project has laid the foundation for object detection in operator interface images, providing important steps to optimize processes of obtaining training and validation data, annotating images, training, and testing models, and embedding those models into software. The first approach of using multi-label classification model in combination with traditional computer vision techniques such as sliding window, pyramid scaling and NMS was used to build a semi-automated annotation tool for more modern object detectors. The annotation tool can preprocess images by performing a pre-analysis using the above-mentioned techniques, and the user can make manual changes to the analysis using the annotation feature of the tool. It is estimated an improved efficiency in the annotation task by 75% compared to traditional tools or fully manual work. Finally, exporting the annotation data for either PASCAL VOC xml format, or YOLO text format. Thus, serving as an annotation tool for both two-stage and one-stage detectors. The tool is also implemented with a feature for loading images with already existing annotation files for performing changes or further annotation of pre-annotated data. This is a good feature as modern sub-optimal object detectors can be used to predict unseen images, then the predictions can be uploaded to the tool, “fixed/improved” and exported as more training data for the modern detector models to be transfer learned on.

In the second iteration of the project, it was researched how to utilize a state of the art one-stage detector architecture called YOLO. Training the latest YOLOv8 model with new custom training data generated using the semi-automated annotation tool and used it to predict/detect objects in unseen images. The first prediction iterations on unseen images resulted in 60-70% of all objects annotated with bounding boxes and annotation files. These predictions were uploaded to the previously developed semi-automated annotation tool and fixed, thus providing more training data, quicker. After multiple iterations, with different preprocessing, the YOLOv8 xl model was trained with a mAP⁵⁰ score of 95.5%. The final model is wrapped in a software, developed as a responsive web application, using the Python FLASK framework. The software is developed as a tool for every-day use, thus focusing on UI design best practices, where a user can upload N number of images, perform an analysis, and get a downloadable excel document containing the final analysis. This tool will be especially helpful in a project planning phase for analyzing object types and counts per image or in total, removing boring repetitive tasks, increasing efficiency, and ensuring accurate overview of details with regards to operator interface images. The tool will also help simplifying migration cost estimation for sales personnel and project managers. Remember, more training data equals better tool, so it is recommended to spend some time testing and collecting data to ensure an accurate and generalized model, before relying on it 100%.

References

- [1] J. Charlie, M. Wulandari, and Nurwijayanti, “Classification of Fertilizer Using OpenCV Based on Color Characteristic,” *IOP Conf. Ser. Mater. Sci. Eng.*, vol. 1007, p. 012053, Dec. 2020, doi: 10.1088/1757-899X/1007/1/012053.
- [2] K. Choudhary *et al.*, “Recent advances and applications of deep learning methods in materials science,” *Npj Comput. Mater.*, vol. 8, no. 1, Art. no. 1, Apr. 2022, doi: 10.1038/s41524-022-00734-6.
- [3] “DeltaV™ Live | Emerson US.” <https://www.emerson.com/en-us/automation/control-and-safety-systems/distributed-control-systems-dcs/deltav-distributed-control-system/deltav-live> (accessed May 03, 2023).
- [4] S. Paliwal, M. Sharma, and L. Vig, *OSSR-PID: One-Shot Symbol Recognition in P&ID Sheets using Path Sampling and GCN*. 2021.
- [5] Z. Tian, W. Huang, T. He, P. He, and Y. Qiao, “Detecting Text in Natural Image with Connectionist Text Proposal Network.” arXiv, Sep. 12, 2016. doi: 10.48550/arXiv.1609.03605.
- [6] R. Rahul, S. Paliwal, M. Sharma, and L. Vig, “Automatic Information Extraction from Piping and Instrumentation Diagrams;,” in *Proceedings of the 8th International Conference on Pattern Recognition Applications and Methods*, Prague, Czech Republic: SCITEPRESS - Science and Technology Publications, 2019, pp. 163–172. doi: 10.5220/0007376401630172.
- [7] Y. Moon, J. Lee, D. Mun, and S. Lim, “Deep Learning-Based Method to Recognize Line Objects and Flow Arrows from Image-Format Piping and Instrumentation Diagrams for Digitization,” *Appl. Sci.*, vol. 11, no. 21, Art. no. 21, Jan. 2021, doi: 10.3390/app112110054.
- [8] J. Wang, W. Yang, H. Guo, R. Zhang, and G.-S. Xia, “Tiny Object Detection in Aerial Images,” in *2020 25th International Conference on Pattern Recognition (ICPR)*, Jan. 2021, pp. 3791–3798. doi: 10.1109/ICPR48806.2021.9413340.
- [9] C. Lee *et al.*, “Interactive Multi-Class Tiny-Object Detection.” arXiv, Mar. 29, 2022. Accessed: Apr. 23, 2023. [Online]. Available: <http://arxiv.org/abs/2203.15266>
- [10] “What is SCADA?,” *Inductive Automation*. <http://www.inductiveautomation.com/resources/article/what-is-scada> (accessed May 05, 2023).
- [11] “What is Machine Learning? | IBM.” <https://www.ibm.com/topics/machine-learning> (accessed Feb. 16, 2023).
- [12] O. M. Brastein, “Introduction to Artificial Neural Networks and Deep Learning.” Nov. 01, 2021.
- [13] N.-O. Skeie, “Object-Oriented Analysis, Design and Programming (OOADP) - Analysis Use Cases (2).” USN, Jan. 09, 2018.
- [14] J. H. and S. Gugger, “fast.ai - fastai A Layered API for Deep Learning.” <https://www.fast.ai/posts/2020-02-13-fastai-A-Layered-API-for-Deep-Learning.html> (accessed Feb. 23, 2023).

References

- [15] “Ultralytics | Revolutionizing the World of Vision AI,” *Ultralytics*. <https://ultralytics.com> (accessed Mar. 28, 2023).
- [16] G. Jocher, A. Chaurasia, and J. Qiu, “YOLO by Ultralytics.” Jan. 2023. Accessed: Mar. 24, 2023. [Online]. Available: <https://github.com/ultralytics/ultralytics>
- [17] “torchvision — Torchvision main documentation.” <https://pytorch.org/vision/stable/index.html> (accessed Feb. 23, 2023).
- [18] D. Doan, “Tesseract OCR: What is it and why would you choose it?,” *Klippa*, Oct. 20, 2022. <https://www.klippa.com/en/blog/information/tesseract-ocr/> (accessed Feb. 23, 2023).
- [19] “Tesseract OCR in Python with Pytesseract & OpenCV,” *Nanonets AI & Machine Learning Blog*, Aug. 09, 2022. <https://nanonets.com/blog/ocr-with-tesseract/> (accessed Feb. 23, 2023).
- [20] “About,” *OpenCV*. <https://opencv.org/about/> (accessed Feb. 23, 2023).
- [21] “Top 15 Most Useful Python Modules,” *Codevoweb*, Feb. 03, 2022. <https://codevoweb.com/top-15-most-useful-python-modules/> (accessed Feb. 23, 2023).
- [22] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. Accessed: Feb. 16, 2023. [Online]. Available: <https://www.deeplearningbook.org/contents/intro.html>
- [23] “DEEPLIZARD Interactive Demo - Convolution Operation.” <https://deeplizard.com/resource/pavq7noze2> (accessed Mar. 27, 2023).
- [24] K. O’Shea and R. Nash, “An Introduction to Convolutional Neural Networks.” arXiv, Dec. 02, 2015. Accessed: Mar. 27, 2023. [Online]. Available: <http://arxiv.org/abs/1511.08458>
- [25] “(152) Simple explanation of convolutional neural network | Deep Learning Tutorial 23 (Tensorflow & Python) - YouTube.” https://www.youtube.com/watch?v=zfiSAzpy9NM&ab_channel=codebasics (accessed Mar. 27, 2023).
- [26] BChen, “7 popular activation functions you should know in Deep Learning and how to use them with Keras and...,” *Medium*, Jan. 04, 2021. <https://towardsdatascience.com/7-popular-activation-functions-you-should-know-in-deep-learning-and-how-to-use-them-with-keras-and-27b4d838dfe6> (accessed Mar. 28, 2023).
- [27] K. Fukushima, “Visual Feature Extraction by a Multilayered Network of Analog Threshold Elements,” *IEEE Trans. Syst. Sci. Cybern.*, vol. 5, no. 4, pp. 322–333, Oct. 1969, doi: 10.1109/TSSC.1969.300225.
- [28] K. Fukushima, “Cognitron: A self-organizing multilayered neural network,” *Biol. Cybern.*, vol. 20, no. 3, pp. 121–136, Sep. 1975, doi: 10.1007/BF00342633.
- [29] V. Nair and G. E. Hinton, “Rectified Linear Units Improve Restricted Boltzmann Machines”.
- [30] “A Beginner’s Guide to Restricted Boltzmann Machines (RBMs),” *Pathmind*. <http://wiki.pathmind.com/restricted-boltzmann-machine> (accessed Mar. 28, 2023).

References

- [31] “What is Rectified Linear Unit (ReLU),” *Deepchecks*.
<https://deepchecks.com/glossary/rectified-linear-unit-relu/> (accessed May 08, 2023).
- [32] “Activation Functions in Neural Networks [12 Types & Use Cases].”
<https://www.v7labs.com/blog/neural-networks-activation-functions>,
<https://www.v7labs.com/blog/neural-networks-activation-functions> (accessed Mar. 28, 2023).
- [33] “Image Classification - an overview | ScienceDirect Topics.”
<https://www.sciencedirect.com/topics/engineering/image-classification> (accessed May 08, 2023).
- [34] K. He, X. Zhang, S. Ren, and J. Sun, “Deep Residual Learning for Image Recognition.” arXiv, Dec. 10, 2015. Accessed: Mar. 29, 2023. [Online]. Available: <http://arxiv.org/abs/1512.03385>
- [35] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet Classification with Deep Convolutional Neural Networks,” in *Advances in Neural Information Processing Systems*, Curran Associates, Inc., 2012. Accessed: Mar. 29, 2023. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html
- [36] [Classic] *Deep Residual Learning for Image Recognition (Paper Explained)*, (Jul. 14, 2020). Accessed: Mar. 30, 2023. [Online Video]. Available: <https://www.youtube.com/watch?v=GWt6Fu05voI>
- [37] K. E. Koech, “Cross-Entropy Loss Function,” *Medium*, Jul. 16, 2022.
<https://towardsdatascience.com/cross-entropy-loss-function-f38c4ec8643e> (accessed May 08, 2023).
- [38] “FastAI 05_pret_breeds.”
https://colab.research.google.com/github/fastai/fastbook/blob/master/05_pet_breeds.ipynb (accessed May 08, 2023).
- [39] “FastAI 06_multicat.”
https://colab.research.google.com/github/fastai/fastbook/blob/master/06_multicat.ipynb#scrollTo=invs-Qyn8lSC (accessed Feb. 27, 2023).
- [40] S. Saxena, “Binary Cross Entropy/Log Loss for Binary Classification,” *Analytics Vidhya*, Mar. 03, 2021. <https://www.analyticsvidhya.com/blog/2021/03/binary-cross-entropy-log-loss-for-binary-classification/> (accessed Mar. 31, 2023).
- [41] P. Solai, “Convolutions and Backpropagations,” *Medium*, Apr. 18, 2018.
<https://pavisj.medium.com/convolutions-and-backpropagations-46026a8f5d2c> (accessed Mar. 31, 2023).
- [42] J. Brownlee, “A Gentle Introduction to Object Recognition With Deep Learning,” *MachineLearningMastery.com*, May 21, 2019.
<https://machinelearningmastery.com/object-recognition-with-deep-learning/> (accessed May 08, 2023).
- [43] Z. Zou, K. Chen, Z. Shi, Y. Guo, and J. Ye, “Object Detection in 20 Years: A Survey.” arXiv, Jan. 18, 2023. Accessed: Mar. 23, 2023. [Online]. Available: <http://arxiv.org/abs/1905.05055>

References

- [44] P. Viola and M. Jones, “Rapid object detection using a boosted cascade of simple features,” in *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001*, Dec. 2001, p. I–I. doi: 10.1109/CVPR.2001.990517.
- [45] N. Dalal and B. Triggs, “Histograms of oriented gradients for human detection,” in *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR’05)*, Jun. 2005, pp. 886–893 vol. 1. doi: 10.1109/CVPR.2005.177.
- [46] P. Felzenszwalb, D. McAllester, and D. Ramanan, “A discriminatively trained, multiscale, deformable part model,” in *2008 IEEE Conference on Computer Vision and Pattern Recognition*, Anchorage, AK, USA: IEEE, Jun. 2008, pp. 1–8. doi: 10.1109/CVPR.2008.4587597.
- [47] Y. LeCun, L. Bottou, Y. Bengio, and P. Ha, “Gradient-Based Learning Applied to Document Recognition,” 1998.
- [48] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, “You Only Look Once: Unified, Real-Time Object Detection,” in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Las Vegas, NV, USA: IEEE, Jun. 2016, pp. 779–788. doi: 10.1109/CVPR.2016.91.
- [49] W. Liu *et al.*, “SSD: Single Shot MultiBox Detector,” in *Computer Vision – ECCV 2016*, B. Leibe, J. Matas, N. Sebe, and M. Welling, Eds., in Lecture Notes in Computer Science, vol. 9905. Cham: Springer International Publishing, 2016, pp. 21–37. doi: 10.1007/978-3-319-46448-0_2.
- [50] R. Girshick, J. Donahue, T. Darrell, and J. Malik, “Rich feature hierarchies for accurate object detection and semantic segmentation.” arXiv, Oct. 22, 2014. Accessed: Mar. 23, 2023. [Online]. Available: <http://arxiv.org/abs/1311.2524>
- [51] Y. Verma, “R-CNN vs Fast R-CNN vs Faster R-CNN - A Comparative Guide,” *Analytics India Magazine*, Sep. 10, 2021. <https://analyticsindiamag.com/r-cnn-vs-fast-r-cnn-vs-faster-r-cnn-a-comparative-guide/> (accessed Mar. 23, 2023).
- [52] R. Gandhi, “R-CNN, Fast R-CNN, Faster R-CNN, YOLO — Object Detection Algorithms,” *Medium*, Jul. 09, 2018. <https://towardsdatascience.com/r-cnn-fast-r-cnn-faster-r-cnn-yolo-object-detection-algorithms-36d53571365e> (accessed Mar. 23, 2023).
- [53] “R-CNN vs Fast R-CNN vs Faster R-CNN | ML,” *GeeksforGeeks*, Feb. 28, 2020. <https://www.geeksforgeeks.org/r-cnn-vs-fast-r-cnn-vs-faster-r-cnn-ml/> (accessed Mar. 23, 2023).
- [54] R. Girshick, “Fast R-CNN.” arXiv, Sep. 27, 2015. doi: 10.48550/arXiv.1504.08083.
- [55] S. Ren, K. He, R. Girshick, and J. Sun, “Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks.” arXiv, Jan. 06, 2016. Accessed: Mar. 23, 2023. [Online]. Available: <http://arxiv.org/abs/1506.01497>
- [56] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, “You Only Look Once: Unified, Real-Time Object Detection,” in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Las Vegas, NV, USA: IEEE, Jun. 2016, pp. 779–788. doi: 10.1109/CVPR.2016.91.

References

- [57] P. Jiang, D. Ergu, F. Liu, Y. Cai, and B. Ma, “A Review of Yolo Algorithm Developments,” *Procedia Comput. Sci.*, vol. 199, pp. 1066–1073, 2022, doi: 10.1016/j.procs.2022.01.135.
- [58] J. Solawetz, F. JAN 11, and 2023 10 Min Read, “What is YOLOv8? The Ultimate Guide.,” *Roboflow Blog*, Jan. 11, 2023. <https://blog.roboflow.com/whats-new-in-yolov8/> (accessed Mar. 24, 2023).
- [59] “Papers with Code - COCO test-dev Benchmark (Object Detection).” <https://paperswithcode.com/sota/object-detection-on-coco> (accessed Mar. 24, 2023).
- [60] “YOLOv8 Docs.” <https://docs.ultralytics.com/> (accessed Mar. 24, 2023).
- [61] “YOLOv8 Ultralytics: State-of-the-Art YOLO Models,” Jan. 10, 2023. <https://learnopencv.com/ultralytics-yolov8/> (accessed Mar. 24, 2023).
- [62] Z. Wei, C. Duan, X. Song, Y. Tian, and H. Wang, “AMRNet: Chips Augmentation in Aerial Images Object Detection.” arXiv, Oct. 25, 2020. Accessed: Mar. 24, 2023. [Online]. Available: <http://arxiv.org/abs/2009.07168>
- [63] S. Elfving, E. Uchibe, and K. Doya, “Sigmoid-Weighted Linear Units for Neural Network Function Approximation in Reinforcement Learning.” arXiv, Nov. 01, 2017. Accessed: Mar. 28, 2023. [Online]. Available: <http://arxiv.org/abs/1702.03118>
- [64] “SiLU — PyTorch 2.0 documentation.” <https://pytorch.org/docs/stable/generated/torch.nn.SiLU.html> (accessed Mar. 28, 2023).
- [65] Z. Zheng, P. Wang, W. Liu, J. Li, R. Ye, and D. Ren, “Distance-IoU Loss: Faster and Better Learning for Bounding Box Regression.” arXiv, Nov. 19, 2019. Accessed: Mar. 31, 2023. [Online]. Available: <http://arxiv.org/abs/1911.08287>
- [66] R. Khandelwal, “Different IoU Losses for Faster and Accurate Object Detection,” *Analytics Vidhya*, Aug. 12, 2021. <https://medium.com/analytics-vidhya/different-iou-losses-for-faster-and-accurate-object-detection-3345781e0bf> (accessed Mar. 31, 2023).
- [67] guest_blog, “A Beginner’s Guide to Focal Loss in Object Detection!,” *Analytics Vidhya*, Aug. 28, 2020. <https://www.analyticsvidhya.com/blog/2020/08/a-beginners-guide-to-focal-loss-in-object-detection/> (accessed Mar. 31, 2023).
- [68] “Quickstart - YOLOv8 Docs.” <https://docs.ultralytics.com/quickstart/> (accessed Mar. 24, 2023).
- [69] S. K, “Non-maximum Suppression (NMS),” *Medium*, Apr. 30, 2021. <https://towardsdatascience.com/non-maximum-suppression-nms-93ce178e177c> (accessed Mar. 06, 2023).
- [70] “Intersection over Union (IoU),” *Hasty.ai*. <https://hasty.ai/docs/mp-wiki/metrics/iou-intersection-over-union> (accessed Mar. 06, 2023).
- [71] L. Kühne, “Accessible colors for user interfaces,” *Medium*, Dec. 08, 2020. <https://bootcamp.uxdesign.cc/accessible-colors-for-user-interfaces-b82ba5a837da> (accessed Apr. 20, 2023).
- [72] “Types of Contrast in User Interface Design,” *Tubik Blog: Articles About Design*, Aug. 09, 2021. <https://blog.tubikstudio.com/contrast-in-user-interface-design/> (accessed Apr. 20, 2023).

References

- [73] “WebAIM: Contrast Checker.” <https://webaim.org/resources/contrastchecker/> (accessed Apr. 20, 2023).
- [74] “What is Flask Python - Python Tutorial.” <https://pythonbasics.org/what-is-flask-python/> (accessed May 08, 2023).
- [75] “The fastai book.” fast.ai, Feb. 21, 2023. Accessed: Feb. 21, 2023. [Online]. Available: https://github.com/fastai/fastbook/blob/823b69e00aa1e1c1a45fe88bd346f11e8f89c1ff/05_pet_breeds.ipynb
- [76] “FastAI 07_sizing_and_tta.” https://colab.research.google.com/github/fastai/fastbook/blob/master/07_sizing_and_tta.ipynb#scrollTo=v4l3PhkV-yhc (accessed Feb. 24, 2023).
- [77] P. Soni, “Data augmentation: Techniques, Benefits and Applications | Analytics Steps.” <https://www.analyticssteps.com/blogs/data-augmentation-techniques-benefits-and-applications> (accessed May 04, 2023).
- [78] “Image Pyramid using OpenCV | Python,” *GeeksforGeeks*, May 16, 2019. <https://www.geeksforgeeks.org/image-pyramid-using-opencv-python/> (accessed Apr. 13, 2023).
- [79] N. Bodla, B. Singh, R. Chellappa, and L. S. Davis, “Soft-NMS -- Improving Object Detection With One Line of Code.” arXiv, Aug. 08, 2017. doi: 10.48550/arXiv.1704.04503.
- [80] “Mean Average Precision (mAP) Explained: Everything You Need to Know.” <https://www.v7labs.com/blog/mean-average-precision>, <https://www.v7labs.com/blog/mean-average-precision> (accessed Apr. 27, 2023).
- [81] M. Venturelli, “The dangers behind image resizing,” *Blog - Zuru Tech*, 08 2021. <https://zuru.tech/blog/the-dangers-behind-image-resizing> (accessed May 08, 2023).
- [82] “Figma: the collaborative interface design tool.,” *Figma*. <https://www.figma.com/> (accessed May 04, 2023).

Appendices

Appendix A: Task Description – Master’s Thesis 2023 – Eirik Illing - Signed

Appendix B: GANTT project planning

Appendix C: WBS project planning

Appendix D: Development Environment

Appendix E: Single-Label Classifier (source-code)

Appendix F: Multi-Label Classifier (source-code)

Appendix G: PyrScaled SlidingWindow NMS Classifier (source-code)

Appendix H: Annotation tool, GitHub repo:
https://github.com/engineirik/annotation_software (source-code)

Appendix I: Split Image annotation YOLO Prep (source code)

Appendix J: OCR (source-code)

Appendix K: ICE software, GitHub repo:
https://github.com/engineirik/ice_app (source-code)

Appendix L: mAP calculation (source-code)

Appendix M: Semi-automated annotation tool mockup drawing

Appendix N: UI Figma design for ICE software

Appendix A
Task Description MT-70-23 Eirik
Illing - Signed

FMH606 Master's Thesis

Title: Object detection, information extraction and analysis of operator interface images using computer vision and machine learning

USN supervisor: Associate Professor Ole M. Brastein and Professor Nils-Olav Skeie

External partner: Emerson Automation Solutions, Geir Falkevik

Task background:

Migrating from old outdated human machine interfaces (HMI), process displays or operator graphics to new modern high-performance HMI's (HPHMI) is often time consuming and costly. When creating a proposal for such migration projects, the sales and project team are often given an overview of today's old displays in configuration files or in plain images. If the input is configuration files, the engineers have tools for extracting data directly from these files, resulting in a good estimate of display complexity and therefore a fair time and cost estimate. However, if the input is plain images, the complexity analysis of these displays is done manually by counting custom and non-custom objects in the display, static and dynamic objects, clustering etc. This manual analysis is very time consuming and has a much higher degree of uncertainty that could result in poor time and cost estimates.

Emerson delivers a world known distributed control system (DCS) known as DeltaV. DeltaV comes with a fully integrated operator graphics tool known as DeltaV Operate. This tool has served its purpose for many years for all of Emerson's customer and will continue to do so in many years to come. However, this operator graphics tool is based on older technology and a new and better fully integrated operator graphics tool known as DeltaV Live has come to replace it. DeltaV Live is a state-of-the-art modern stable framework for high performance operator graphics, so migrating from DeltaV Operate to DeltaV Live is in high demand. These migration projects are the foundation for this master's thesis, where Emerson wants to investigate the possibility for creating a tool to do a complexity analysis of old DeltaV Operate operator graphics, to get a good and fair estimate of migration time and cost for its customers.

Task description:

Interim goals:

- Summary of literature review regarding object detection methods in images (containing a large quantity of objects).
- Choose one or more suitable approaches for object detection and object classification to extract components and information from images.
- Describe how to obtain valuable datasets for training, validating, and testing models for this specific task. Look into the possibility of customer adjusted standard dynamo sets for object detection.
- Suggest analytical methods for pre-processing and clean-up/preparations of datasets.
- Develop machine learning models and check the accuracy and repeatability of the models.
- Develop an application focusing on user interface (UI) design for interacting with the model/software.

Student category: IIA (EET, EPE, IIA or PT students)

Is the task suitable for online students (not present at the campus)?

No

Practical arrangements:

This project is reserved for the industry master student at Emerson, Eirik Illing.

Supervision:

As a general rule, the student is entitled to 15-20 hours of supervision. This includes necessary time for the supervisor to prepare for supervision meetings (reading material to be discussed, etc).

Signatures:

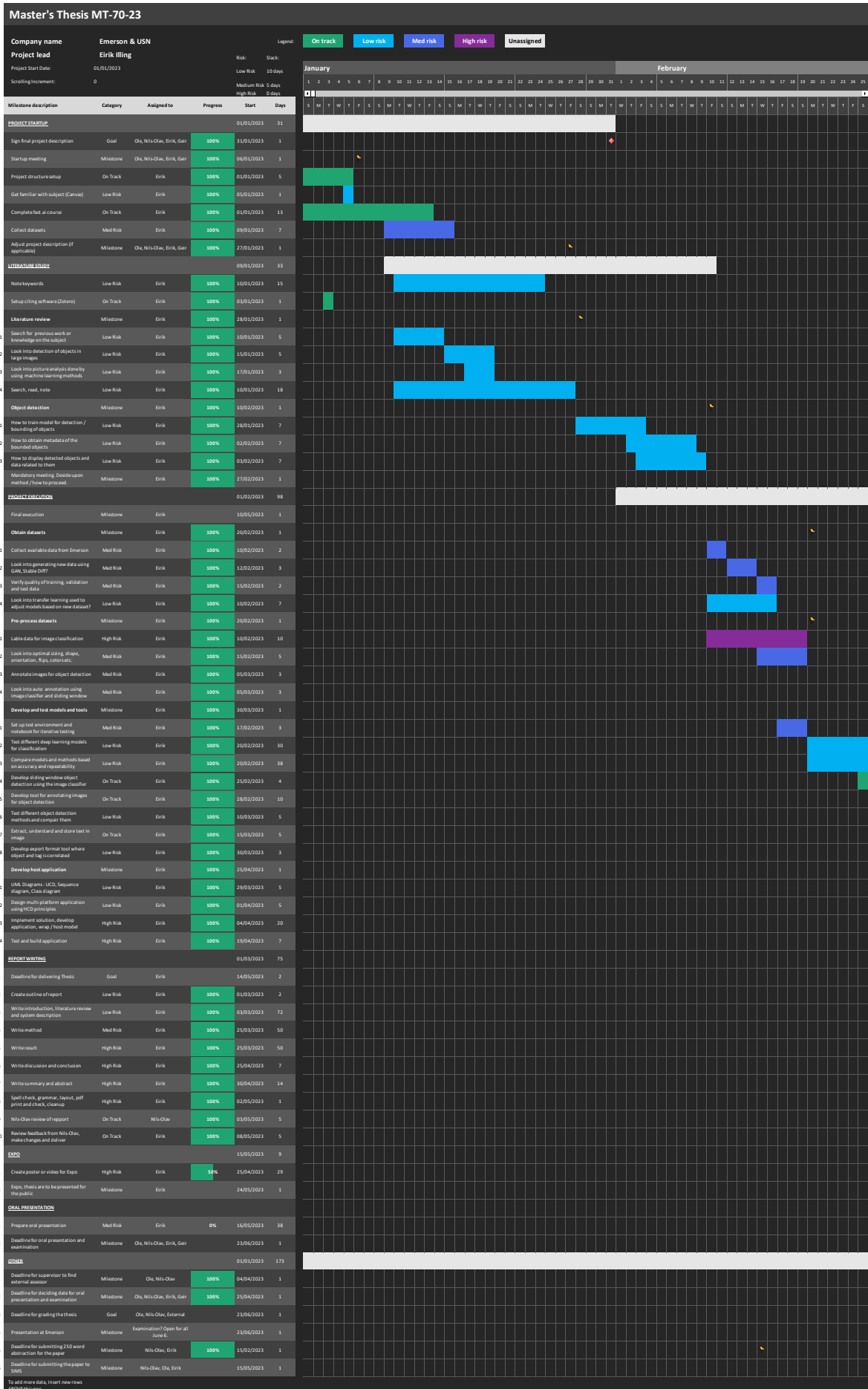
Supervisor (date and signature): *01/02-23*
Ole M. Brastein

Student (write clearly in all capitalized letters): *ERIK ILLING*

Student (date and signature): *01/02-23*
Eirik Illing

Appendix B

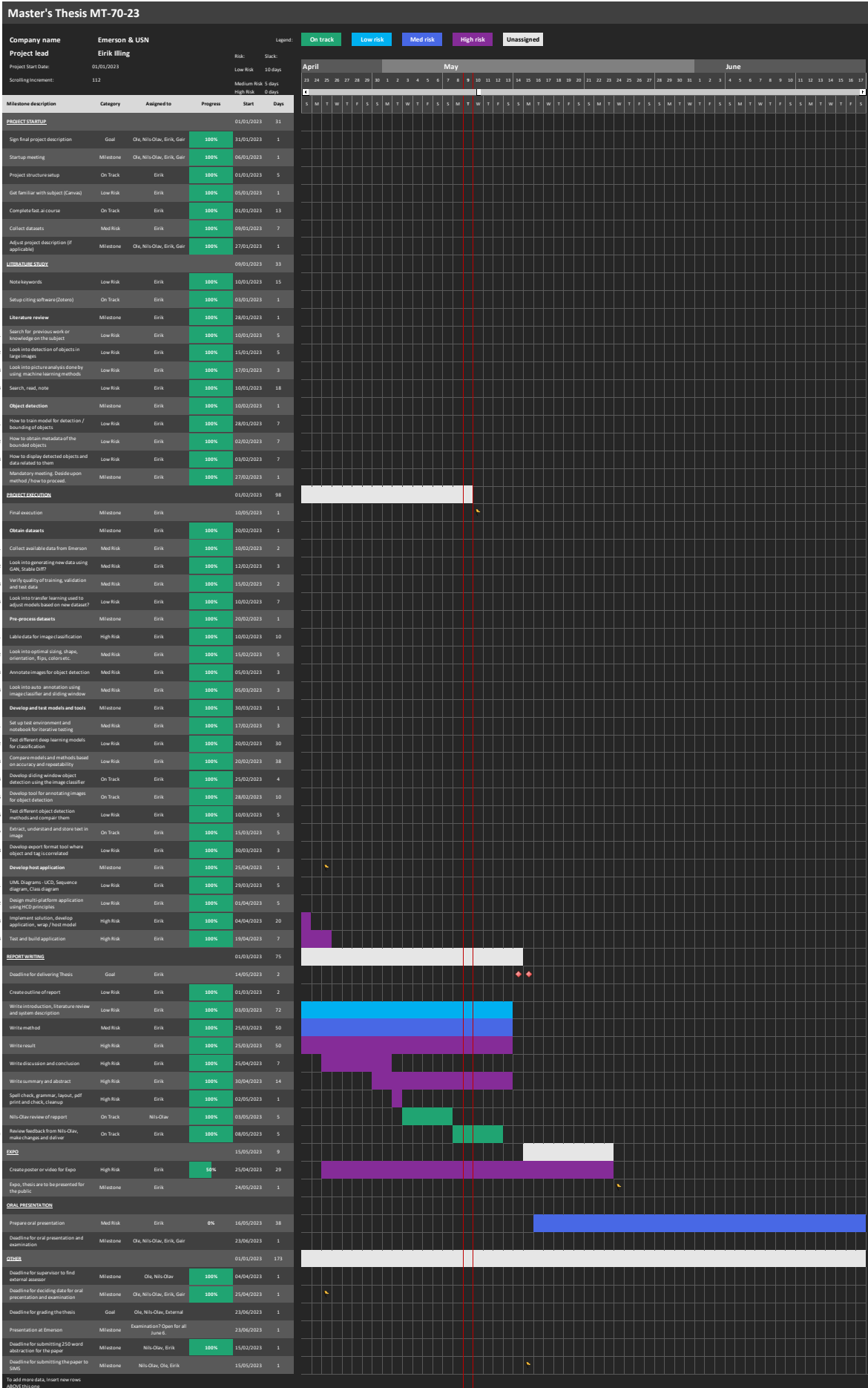
GANTT Project Planning



Master's Thesis MT-70-23					
Company name		Emerson & USN		Legend: On track Low risk Med risk High risk Unassigned	
Project lead		Erik Illing		Risk:	Stack:
Project Start Date:		03/01/2023		Low Risk	10 days
Scrolling increment:		56		Medium Risk	5 days
				High Risk	0 days
February					
March					
April					
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31					
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31					
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31					
1	PROJECT STARTUP				01/01/2023 31
1.1	Sign final project description	Goal	Ch, Nils-Olav, Erik, Ger	100%	31/01/2023 1
1.2	Startup meeting	Milestone	Ch, Nils-Olav, Erik, Ger	100%	06/01/2023 1
1.3	Project structure setup	On Track	Erik	100%	01/01/2023 5
1.4	Get familiar with subject (Canvas)	Low Risk	Erik	100%	05/01/2023 1
1.5	Complete last al course	On Track	Erik	100%	01/01/2023 13
1.6	Collect datasets	Med Risk	Erik	100%	09/01/2023 7
1.7	Adjust project description (if applicable)	Milestone	Ch, Nils-Olav, Erik, Ger	100%	27/01/2023 1
2	LITERATURE STUDY				09/01/2023 33
2.1	Note keywords	Low Risk	Erik	100%	10/01/2023 15
2.2	Setup citing software (Zotero)	On Track	Erik	100%	03/01/2023 1
2.3	Literature review	Milestone	Erik	100%	28/01/2023 1
2.3.1	Search for previous work or knowledge on the subject	Low Risk	Erik	100%	10/01/2023 5
2.3.2	Look into detection of objects in large images	Low Risk	Erik	100%	15/01/2023 5
2.3.3	Look into picture analysis done by using machine learning methods	Low Risk	Erik	100%	17/01/2023 3
2.3.4	Search, read, note	Low Risk	Erik	100%	10/01/2023 18
2.4	Object detection	Milestone	Erik	100%	10/02/2023 1
2.4.1	How to train model for detection / bounding of objects	Low Risk	Erik	100%	28/01/2023 7
2.4.2	How to obtain metadata of the bounded objects	Low Risk	Erik	100%	02/02/2023 7
2.4.3	How to display detected objects and data related to them	Low Risk	Erik	100%	03/02/2023 7
2.5	Mandatory meeting. Decide upon method. How to proceed.	Milestone	Erik	100%	27/02/2023 1
3	PROJECT EXECUTION				03/02/2023 98
3.1	Final execution	Milestone	Erik		10/05/2023 1
3.2	Obtain datasets	Milestone	Erik	100%	20/02/2023 1
3.2.1	Collect available data from Emerson	Med Risk	Erik	100%	10/02/2023 2
3.2.2	Look into generating new data using GAN, Stable Diff?	Med Risk	Erik	100%	12/02/2023 3
3.2.3	Verify quality of training, validation and test data	Med Risk	Erik	100%	15/02/2023 2
3.2.4	Look into transfer learning used to adjust models based on new dataset?	Low Risk	Erik	100%	10/02/2023 7
3.3	Pre-process datasets	Milestone	Erik	100%	20/02/2023 1
3.3.1	Label data for image classification	High Risk	Erik	100%	10/02/2023 10
3.3.2	Look into optimal labeling, shape, orientation, flip, color, etc.	Med Risk	Erik	100%	15/02/2023 5
3.3.3	Annotate images for object detection	Med Risk	Erik	100%	05/03/2023 3
3.3.4	Look into auto-annotation using image classifier and sliding window	Med Risk	Erik	100%	05/03/2023 3
3.4	Develop and test models and tools	Milestone	Erik	100%	30/03/2023 1
3.4.1	Set up test environment and notebook for iterative testing	Med Risk	Erik	100%	17/02/2023 3
3.4.2	Test different deep learning models for classification	Low Risk	Erik	100%	20/02/2023 30
3.4.3	Compare models and methods based on accuracy and repeatability	Low Risk	Erik	100%	20/02/2023 38
3.4.4	Develop sliding window object detection using the image classifier	On Track	Erik	100%	25/02/2023 4
3.4.5	Develop tool for annotating images for object detection	On Track	Erik	100%	28/02/2023 10
3.4.6	Test different object detection methods and compare them	Low Risk	Erik	100%	10/03/2023 5
3.4.7	Extract, understand and store text in images	On Track	Erik	100%	15/03/2023 5
3.4.8	Develop export format tool where object and tag is correlated	Low Risk	Erik	100%	30/03/2023 3
3.5	Develop host application	Milestone	Erik	100%	25/04/2023 1
3.5.1	UML Diagrams - UCD, Sequence Diagram, Class diagram	Low Risk	Erik	100%	29/03/2023 5
3.5.2	Design multiplatform application using i2D principles	Low Risk	Erik	100%	01/04/2023 5
3.5.3	Implement solution, develop application, wrap / host model	High Risk	Erik	100%	04/04/2023 20
3.5.4	Test and build application	High Risk	Erik	100%	19/04/2023 7
4	REPORT WRITING				03/03/2023 75
4.1	Deadline for delivering Thesis	Goal	Erik		14/05/2023 2
4.2	Create outline of report	Low Risk	Erik	100%	03/03/2023 2
4.3	Write introduction, literature review and system description	Low Risk	Erik	100%	03/03/2023 72
4.4	Write method	Med Risk	Erik	100%	25/03/2023 50
4.5	Write result	High Risk	Erik	100%	25/03/2023 50
4.6	Write discussion and conclusion	High Risk	Erik	100%	25/04/2023 7
4.7	Write summary and abstract	High Risk	Erik	100%	30/04/2023 14
4.8	Spell check, grammar, layout, pdf print and check, cleanup	High Risk	Erik	100%	02/05/2023 1
4.9	Nils-Olav review of report	On Track	Nils-Olav	100%	03/05/2023 5
4.10	Review feedback from Nils-Olav, make changes and deliver	On Track	Erik	100%	08/05/2023 5
5	EXPO				15/05/2023 9
5.1	Create poster or video for Expo	High Risk	Erik	50%	25/04/2023 29
5.2	Expo, thesis are to be presented for the public	Milestone	Erik		24/05/2023 1
6	ORAL PRESENTATION				
6.1	Prepare oral presentation	Med Risk	Erik	0%	16/05/2023 38
6.2	Deadline for oral presentation and examination	Milestone	Ch, Nils-Olav, Erik, Ger		23/06/2023 1
7	SMKS				01/01/2023 173
7.1	Deadline for supervisor to find external examiner	Milestone	Ch, Nils-Olav	100%	04/04/2023 1
7.2	Deadline for deciding date for oral presentation and examination	Milestone	Ch, Nils-Olav, Erik, Ger	100%	25/04/2023 1
7.3	Deadline for grading the thesis	Goal	Ch, Nils-Olav, External		23/06/2023 1
7.4	Presentation at Emerson	Milestone	Examination? Open for all June 6.		23/06/2023 1
7.5	Deadline for submitting 250 word abstract for the paper	Milestone	Nils-Olav, Erik	100%	15/02/2023 1
7.6	Deadline for submitting the paper to SMKS	Milestone	Nils-Olav, Ch, Erik		15/05/2023 1

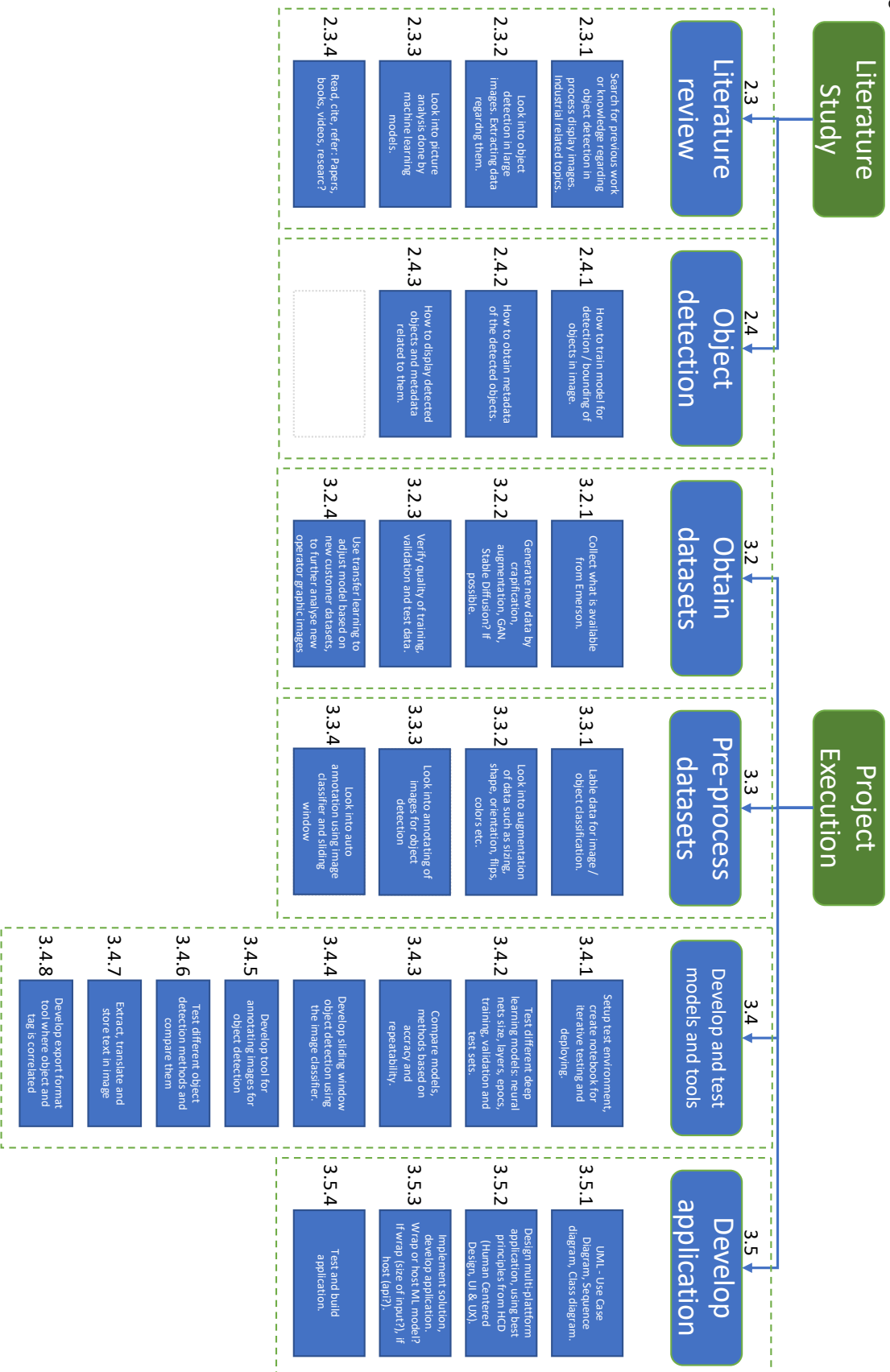


To add more data, insert new rows above this one.



Appendix C

WBS Project Planning



Appendix D
Development Environment
Elaborated

1 Development environment

Machine learning tasks can be computationally heavy to perform. Specially during development of certain applications while training and testing. A decent hardware and software environment is key for efficiency and performance. This development station and environment will be hosted on a local computer in the office, with remote access via TeamViewer. The computer will also be connected to a Raspberry PI4 that is configured to reboot if/after power loss. This Raspberry PI4 can also be reached with TeamViewer, where a wake on LAN magic package can be sent from the Raspberry PI4 to the development station, thus turning it on. The development station is configured with Wake On LAN in bios and on the Ethernet Controller.

1.1 Hardware environment

The most demanding task while developing machine learning models is the training of models and predicting large quantity of information. For this process, GPUs are key components, as they are built to perform complex parallel computation. GPUs are more suited for these kinds of tasks compared to CPU because they are specifically designed for calculations related to graphics and rendering. GPUs are equipped with more cores and higher bandwidth than CPUs, thus able to perform a lot more tasks at once. CPUs are on the other hand equipped with more powerful cores, better suited for sequential processing. One significant difference between these two is that GPUs does not dynamically allocate and dump memory the same way that CPUs does, so memory management is a key factor when working with GPU computation. There are varies methods for handling these “out of memory” error cases when working with machine learning, such as reducing batch size in training, use smaller/less complex model, mixed precision training and killing processes. So, when deciding upon hardware components for machine learning development, GPU and cooling will be the most crucial components.

For this project, an old gaming computer seemed to be a good fit. The computer has a GTX1080 overclocked GPU, an Intel Core i5-8400 processor, 16gib of DDR4 RAM, 250gib M.2 SSD. Table 1 gives an overview of components and part numbers used in the development machine.

Table 1: List of development environment hardware

Part name	Part number	Description
MSI B360I Gaming Pro AC, Socket-1151	B360I GAMING PRO AC	Motherboard
Intel Core i5-8400 Processor	BX80684I58400	CPU
Asus GeForce GTX 1080 Rog Strix	ROG STRIX-GTX1080-A8G-GAMING	GPU
Corsair Vengeance LPX DDR4 2400MHz 16gb	CMK16GX4M2A2400C14	RAM
WD Black SSD 250GB M.2 PCIe	WDS250G2X0C	SSD

Cooler Master MasterWATT 650	MPX-6501-AMAAB-EU	PS
------------------------------	-------------------	----

1.2 Software environment

The pc was reinstalled with Windows 10, student edition. Windows 10 is a perfectly fine multipurpose OS designed for everything from everyday use to development. However, more advanced development requiring a large quantity of open-source packages and flexibility can get tedious when working with Windows. This is mainly because Windows focus on a graphical user interface experience, while developing software often limits itself to working with command line tools. Using some sort of Linux distro therefore seems like a more appealing approach.

One thing to note about Windows is that it has better commercial software and hardware drives support. Some sort of mix, running Windows as main OS and virtualizing an Ubuntu environment is a good idea. However, running Ubuntu as a virtual machine will result in hardware limitations as it is predefined with a specific amount of computing power when set up. A virtual machine also requires some recourses just to run, and this could affect the overall machine performance. It is also tedious to set up, allocate memory and configure file sharing between Windows and virtual machine.

Second option is to dual boot the system with a native Ubuntu distro. This will give the distro full access to computing power, but the disk space needs to be partitioned giving 50/50 to Windows and Ubuntu. File sharing between these two OS's is also a hassle, and it requires the user to turn the machine on and off to switch environment. The hardware drives can also become an issue on the Linux system.

The final and most diffidently best approach is to set up a Windows Subsystem for Linux directly from Windows 10 terminal (CMD). WSL is Microsoft's answer to more flexible open-source Linux environments directly on Windows. Preventing developers from switching to Linux distros as they advance in their carrier and making it more appealing for Linux users to switch too Windows. WSL is a lightweight and integrated solution running Linux on a Windows operating system. It can directly access files and share resources with the Windows host. And since WSL also shares the same kernel as the Windows host, it also inherits the security protections. This is not the case for a virtual machine running on Windows services such as Hyper-V, VirtualBox or Wmware Workstation.

Setting up WSL and installing a distro is easy. Find a good tutorial online, such as the one referred to in this section [1]. Follow it and do adjustments required for different hardware specifications. It is recommended to have some basic understanding of Linux file system and package installation. Otherwise, use the internet to search for help and solve error messages. Start by installing Docker Desktop on Windows, this is handy for containerizing projects running on the Linux kernel using the WSL as backend. It is not required to have Docker installed, but recommended. Next install WSL by running the `wsl --install -d Ubuntu`. Where Ubuntu specifies the Linux distro for installation. Ubuntu will then be installed on the machine, and can be opened by searching for "Ubuntu" in the Windows menu. A new terminal with the Ubuntu terminal will open, representing the Ubuntu machine. Next it is recommended to set up git and connect to a online git source-code storage and management service such as GitKraken or GitHub. Then install Visual Studio Code as a code editor on Windows, and connect it to WSL by adding the Remote Development extension pack. This gives the possibility to open any folder from the Ubuntu terminal in VSC by running the

“code .” command. After the IDE or Code editor is integrated, it is time to install development environment and packages in Ubuntu. Install MiniConda or Mamba, which is lightweight Python Conda package manager. This will give the bare minimum to create Conda environments and start Python development. Create a new Conda environment by running the “conda create -n newEnv” command. It is recommended to work in separate environments when developing to easier manage packages, prevent conflicts and backup. Finally there is one last thing that needs to be taken care of to access the processing power of the GPU hardware both in Windows and on the Ubuntu distro.

Installing packages for NVIDIA CUDA toolkit and cuDNN drivers. Go to the NVIDIA for developers website, download and install the latest CUDA driver on the Windows OS. Then download and install the cuDNN drivers for the Windows OS. Extract the cuDNN drivers from the installation folder and move them into and overwrite exiting driver folders in the \Program Files\NVIDIA GPU Computing Toolkit\CUDA\driver folder on the Windows machine. Both the bin and libnvvp folder need to be added to the Environment Variable path. A complete guide written by Bex T. can be found at [towardsdatascience.com](https://towardsdatascience.com/referenced-here) referenced here [2]. When installation on Windows machine is done, it is recommended to test it locally before installing the same driver support on the WSL Ubuntu system. This was found to be unnecessary in this project.

Next, install the same support on WSL in the Ubuntu terminal using a few simple commands shown in step 16 by Bex T. in [towardsdatascience.com](https://towardsdatascience.com/referenced-here) referenced here [1]. Then install the preferred Machine Learning libraries such as PyTorch, Tensorflow, Keras in the Conda environment created earlier or separate environments. It is recommended to keep some these separated as they may cause conflict with each other. This, however, needs to be tested and researched before use. If a mistake is made and conflicts occur, simply create new Conda environment and reinstall. Remember to install the packages that are supported for WSL and with GPU support. This can be found on the packages official sites. A list of packages used in this project can be seen in In this project, a WSL Ubuntu distro was created, set up with Git and MiniConda and multiple new template Conda environments were created with all packages and GPU functionality. This template is then copied into new development environments for testing and developing. This way, a fresh working environment is always available if something should go wrong in the developing environment. This environment can also be exported to a .yaml file and imported on other machines running a Conda setup on Ubuntu distro.

- [1] B. T, “How to Create Perfect Machine Learning Development Environment With WSL2 on Windows 10/11,” *Medium*, Dec. 09, 2022. <https://towardsdatascience.com/how-to-create-perfect-machine-learning-development-environment-with-wsl2-on-windows-10-11-2c80f8ea1f31> (accessed Feb. 14, 2023).
- [2] B. T, “How to Finally Install TensorFlow 2 GPU on Windows 10 in 2022,” *Medium*, Dec. 09, 2022. <https://towardsdatascience.com/how-to-finally-install-tensorflow-gpu-on-windows-10-63527910f255> (accessed Feb. 14, 2023).

Appendix E

Single-Label Classifier

Jupyter Notebook

```
In [1]: from fastai.vision.all import *
from fastbook import *
from fastai.vision.widgets import *
from fastai.callback.fp16 import *
import warnings
warnings.filterwarnings("ignore")
```

```
In [2]: path = Path("/home/engineirik/git/classify_singl_obj/Classification_small_singleobj")
Path.BASE_PATH = path
path.ls()
```

```
Out[2]: (#20) [Path('pump_isa'),Path('status'),Path('chart'),Path('valve_m'),Path('mixer'),Path('valve_pr'),Path('valve'),Path('valve_p'),Path('valve_m_3w'),Path('valve_h')]...
```

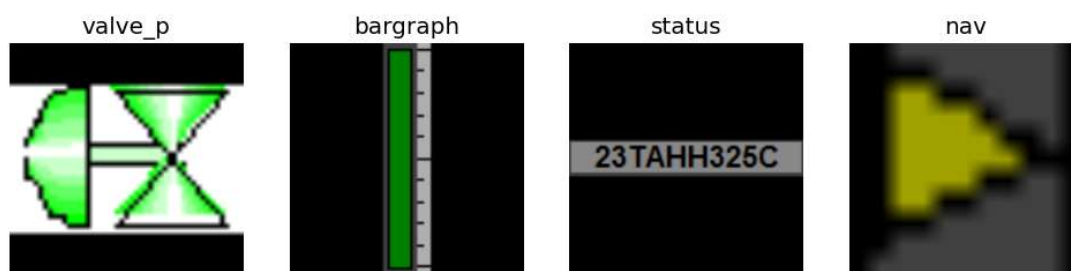
```
In [3]: fns = get_image_files(path/"status")
len(fns)
```

```
Out[3]: 125
```

Model and Preprocessing

```
In [4]: data = DataBlock(
    blocks=(ImageBlock, CategoryBlock),
    get_items=get_image_files,
    splitter=RandomSplitter(valid_pct=0.2, seed=42),
    get_y=parent_label,
    item_tfms=Resize(224, ResizeMethod.Pad, pad_mode='zeros')
)
dls = data.dataloaders(path)
```

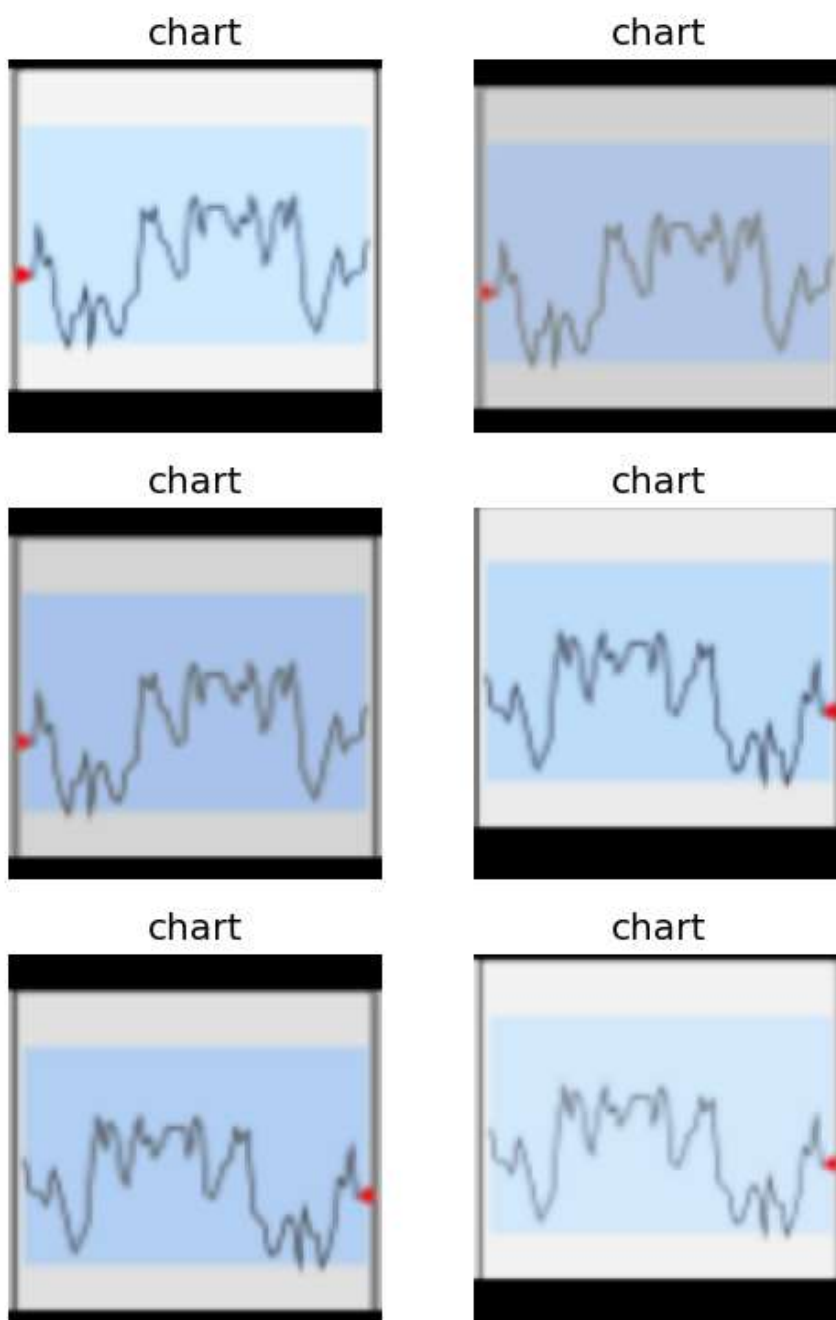
```
In [5]: dls.valid.show_batch(max_n=4, nrows=1)
```



Augmentation

```
In [6]: data = data.new(
    item_tfms=Resize(224, ResizeMethod.Pad, pad_mode='zeros'),
    batch_tfms=aug_transforms(size=128, min_scale=1, mult=2, max_warp=0,
        do_flip=True, flip_vert=False, max_zoom=1.02,
        pad_mode="zeros", max_rotate=0))
```

```
dls = data.dataloaders(path, bs=16)  
dls.train.show_batch(max_n=6, nrows=3, unique=True)
```



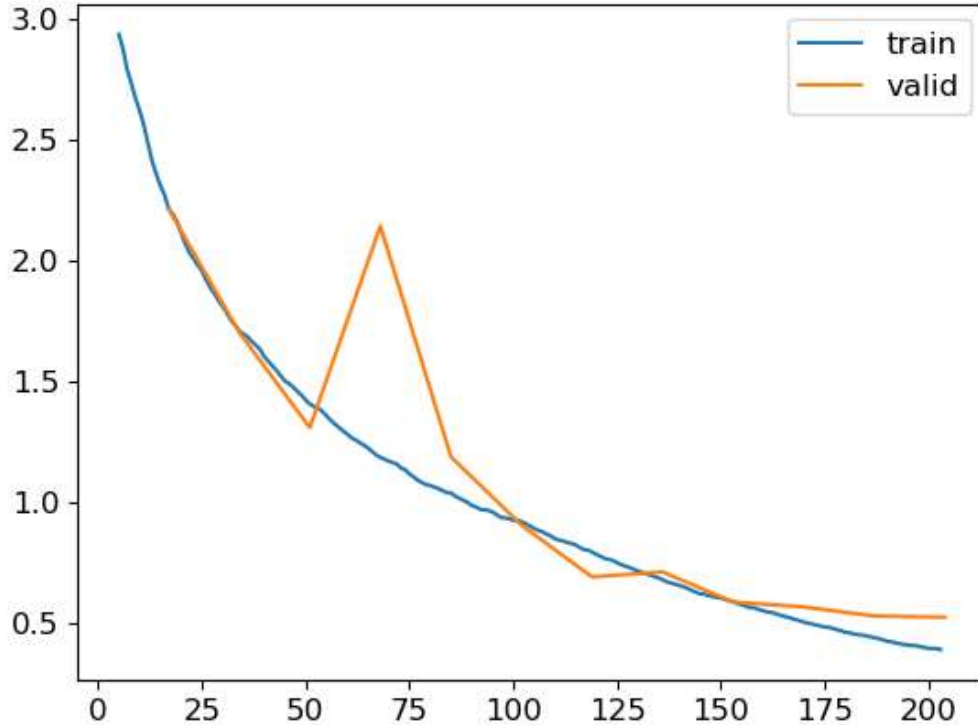
Train non pretrained model

```
In [6]: model = xresnet50(n_out=dls.c)  
learn = Learner(dls, model, loss_func=CrossEntropyLossFlat(),  
               metrics=accuracy)  
#Learn.fine_tune(9, freeze_epochs=3)
```

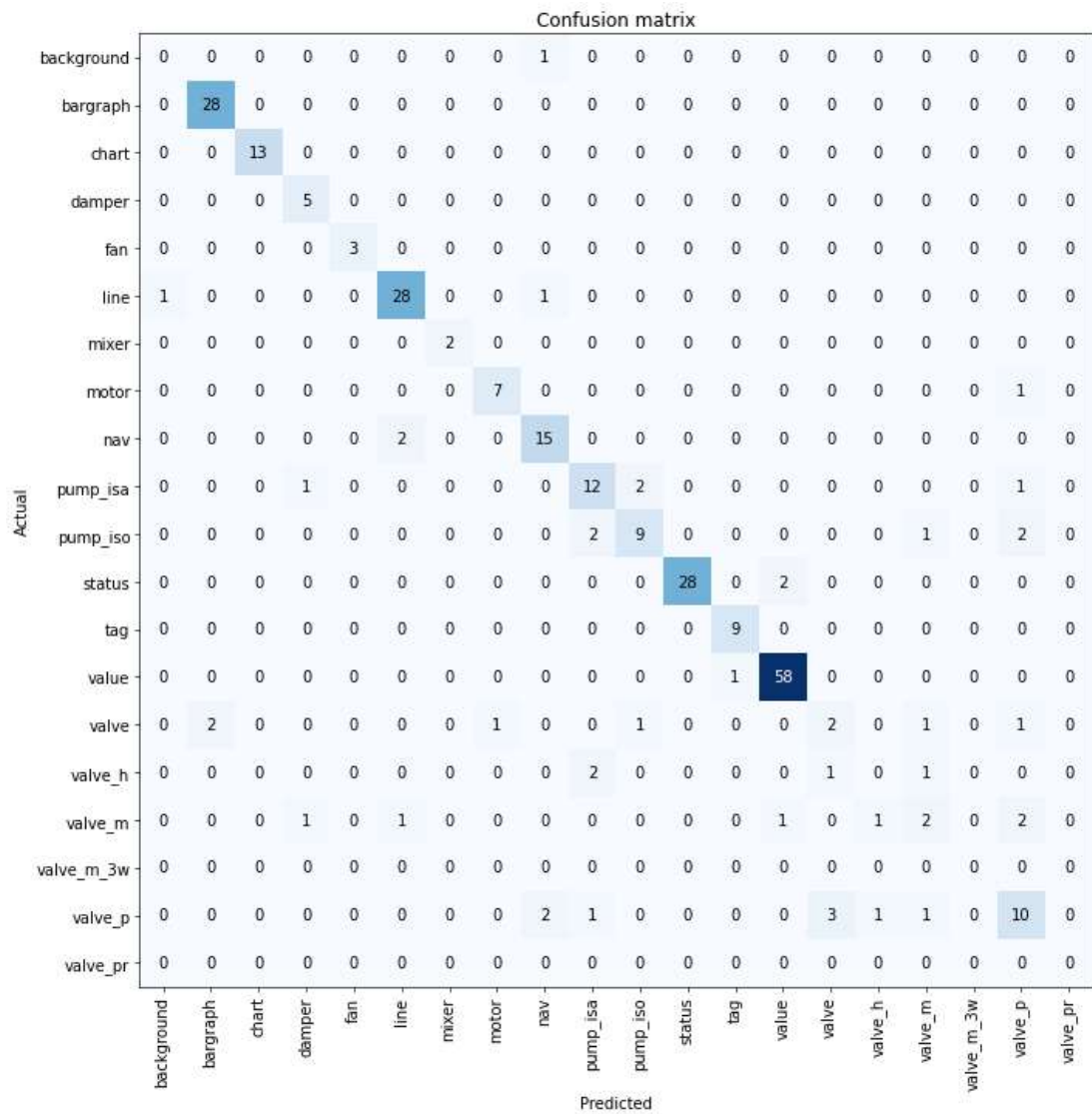
```
learn.fit_one_cycle(12)  
#Learn.fit_one_cycle(5, 3e-3)
```

epoch	train_loss	valid_loss	accuracy	time
0	2.268960	2.210411	0.241758	00:21
1	1.729840	1.700031	0.494505	00:19
2	1.421203	1.306692	0.600733	00:19
3	1.192690	2.140647	0.560440	00:19
4	1.037965	1.185952	0.666667	00:19
5	0.923955	0.903173	0.761905	00:19
6	0.799859	0.688817	0.776557	00:19
7	0.684407	0.709529	0.761905	00:19
8	0.591151	0.585039	0.798535	00:19
9	0.507836	0.564602	0.831502	00:19
10	0.440936	0.527033	0.846154	00:19
11	0.387851	0.522242	0.846154	00:19

```
In [7]: learn.recorder.plot_loss()
```



```
In [8]: inter = ClassificationInterpretation.from_learner(learn)  
inter.plot_confusion_matrix(figsize=(12,12), dpi=60)
```



```
In [9]: #interp.plot_top_losses(20, nrows=5)
interp.most_confused(min_val=1)
```

```
Out[9]: [('valve_p', 'valve', 3),
 ('nav', 'line', 2),
 ('pump_isa', 'pump_iso', 2),
 ('pump_iso', 'pump_isa', 2),
 ('pump_iso', 'valve_p', 2),
 ('status', 'value', 2),
 ('valve', 'bargraph', 2),
 ('valve_h', 'pump_isa', 2),
 ('valve_m', 'valve_p', 2),
 ('valve_p', 'nav', 2),
 ('background', 'nav', 1),
 ('line', 'background', 1),
 ('line', 'nav', 1),
 ('motor', 'valve_p', 1),
 ('pump_isa', 'damper', 1),
 ('pump_isa', 'valve_p', 1),
 ('pump_iso', 'valve_m', 1),
 ('value', 'tag', 1),
 ('valve', 'motor', 1),
 ('valve', 'pump_iso', 1),
 ('valve', 'valve_m', 1),
 ('valve', 'valve_p', 1),
 ('valve_h', 'valve', 1),
 ('valve_h', 'valve_m', 1),
 ('valve_m', 'damper', 1),
 ('valve_m', 'line', 1),
 ('valve_m', 'value', 1),
 ('valve_m', 'valve_h', 1),
 ('valve_p', 'pump_isa', 1),
 ('valve_p', 'valve_h', 1),
 ('valve_p', 'valve_m', 1)]
```

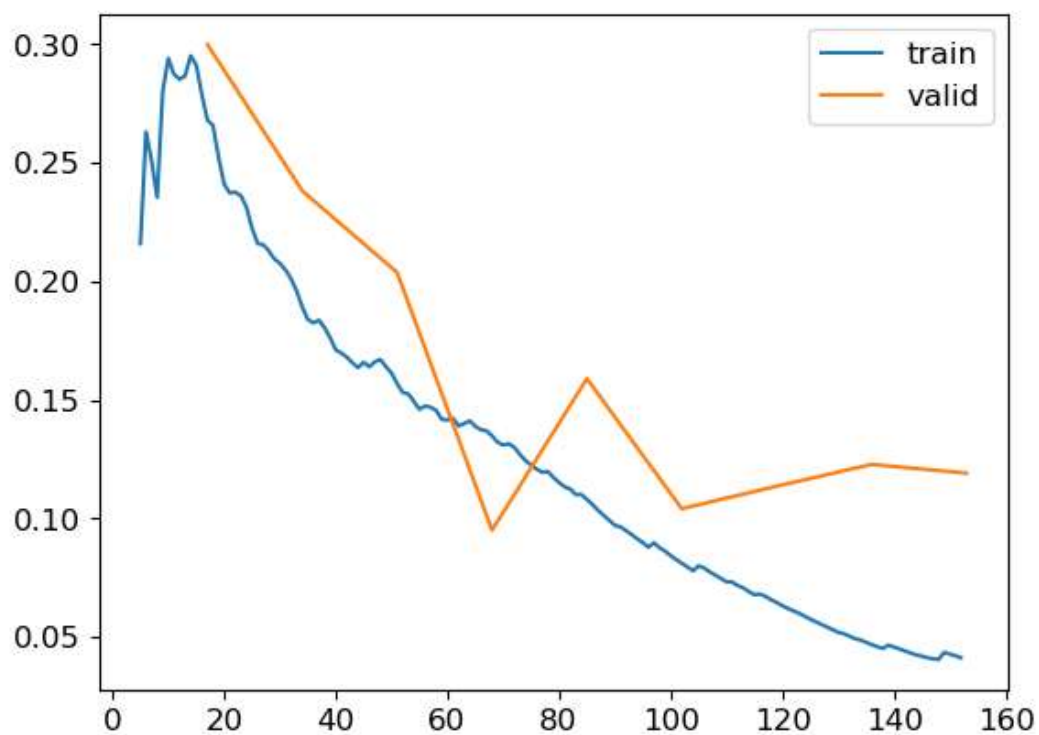
Train pretrained model

```
In [10]: learn = vision_learner(dls, resnet50, metrics=accuracy).to_fp16()
learn.fine_tune(9, freeze_epochs=3)
```

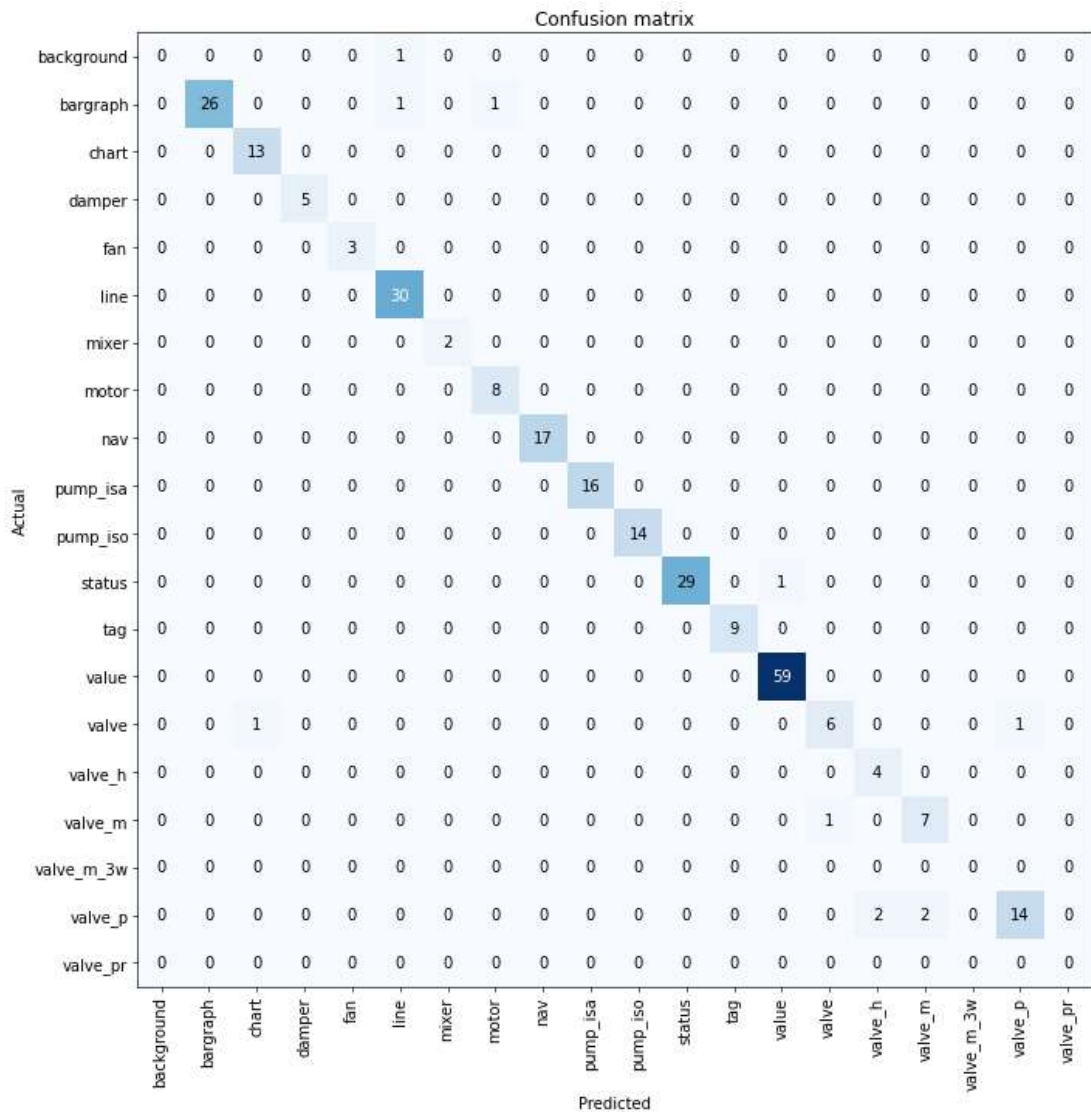
epoch	train_loss	valid_loss	accuracy	time
0	2.896408	0.995524	0.736264	00:14
1	1.646108	0.458982	0.868132	00:14
2	1.059790	0.362602	0.886447	00:14

epoch	train_loss	valid_loss	accuracy	time
0	0.278839	0.299870	0.890110	00:16
1	0.195774	0.238112	0.915751	00:16
2	0.161242	0.203671	0.937729	00:16
3	0.136923	0.095021	0.967033	00:16
4	0.109998	0.158871	0.963370	00:16
5	0.082578	0.104049	0.974359	00:16
6	0.065645	0.113539	0.959707	00:16
7	0.047687	0.122736	0.959707	00:16
8	0.041199	0.119036	0.959707	00:16

```
In [11]: learn.recorder.plot_loss()
```



```
In [12]: interp = ClassificationInterpretation.from_learner(learn)
interp.plot_confusion_matrix(figsize=(12,12), dpi=60)
```



```
In [13]: #interp.plot_top_losses(20, nrows=5)
interp.most_confused(min_val=1)
```

```
Out[13]: [('valve_p', 'valve_h', 2),
          ('valve_p', 'valve_m', 2),
          ('background', 'line', 1),
          ('bargraph', 'line', 1),
          ('bargraph', 'motor', 1),
          ('status', 'value', 1),
          ('valve', 'chart', 1),
          ('valve', 'valve_p', 1),
          ('valve_m', 'valve', 1)]
```

Export model

```
In [14]: filename = "classify_singl_pipeline1.pkl"
```

```
In [16]: learn.export(fname=filename)
path = Path()
path.ls(file_exts='.pkl')
```

```
Out[16]: (#2) [Path('classify_singl_pipeline1.pkl'),Path('simple_classifier_aug1.pkl')]
```

```
In [17]: learn_inf = load_learner(path/filename)
```

```
In [18]: num = 6
imP_path = Path('/home/engineirik/git/classify_singl_obj/test_img/' +
                str(num) + ".png")
imP = Image.open(imP_path)
imP
```

```
Out[18]:
```



```
In [19]: learn_inf.predict(imP_path)
```

```
Out[19]: ('pump_iso',
TensorBase(10),
TensorBase([1.6992e-08, 7.3694e-09, 2.5086e-08, 2.6695e-08, 5.7377e-07, 2.3314e-10, 9.6678e-08, 7.8886e-09, 1.8278e-08, 3.2490e-07, 1.0000e+00, 1.7124e-09, 1.9605e-07, 6.4292e-08, 1.7851e-09, 1.5711e-08, 1.0282e-06, 6.4365e-08, 5.8607e-08, 4.0035e-08]))
```

```
In [20]: learn_inf.dls.vocab
```

```
Out[20]: ['background', 'bargraph', 'chart', 'damper', 'fan', 'line', 'mixer', 'motor', 'nav', 'pump_isa', 'pump_iso', 'status', 'tag', 'value', 'valve', 'valve_h', 'valve_m', 'valve_m_3w', 'valve_p', 'valve_pr']
```

```
In [ ]:
```

Appendix F

Multi-Label Classifier

Jupyter Notebook

Setup

```
In [1]: from fastai.vision.all import *
from fastbook import *
from fastai.vision.widgets import *
from fastai.callback.fp16 import *
import warnings
warnings.filterwarnings("ignore")
```

```
In [2]: path = Path("/home/engineirik/git/classify_multi_obj")
Path.BASE_PATH = path
path.ls()
```

```
Out[2]: (#13) [Path('rename_files_in_folder.ipynb'),Path('.git'),Path('.ipynb_checkpoints'),Path('v2_multiobj_classifier.pkl'),Path('folder_csv_generator.ipynb'),Path('multiobj_classifier.ipynb'),Path('v1_multiobj_classifier.pkl'),Path('test'),Path('items.csv'),Path('copyfiles_from_to_folder.ipynb')]...
```

```
In [3]: df = pd.read_csv(path/'items.csv')
df.head()
```

```
Out[3]:
```

	fname	labels	is_valid
0	pump_isa 8.png	pump_isa	False
1	pump_isa 61.png	pump_isa	True
2	pump_isa 45.png	pump_isa	False
3	pump_isa 69.bmp	pump_isa	False
4	pump_isa 2.png	pump_isa	False

```
In [4]: df.iloc[:,0]
```

```
Out[4]: 0      pump_isa 8.png
1      pump_isa 61.png
2      pump_isa 45.png
3      pump_isa 69.bmp
4      pump_isa 2.png
...
1741   value 240.PNG
1742   value 114.png
1743   value 24.png
1744   value 209.bmp
1745   value 93.png
Name: fname, Length: 1746, dtype: object
```

```
In [5]: df.iloc[0]
```



```

data = DataBlock(blocks=(ImageBlock, MultiCategoryBlock),
                 splitter=splitter,
                 get_x=get_x,
                 get_y=get_y)
dsets = data.datasets(df)
# Again, check the length to make sure it works
len(dsets.valid)

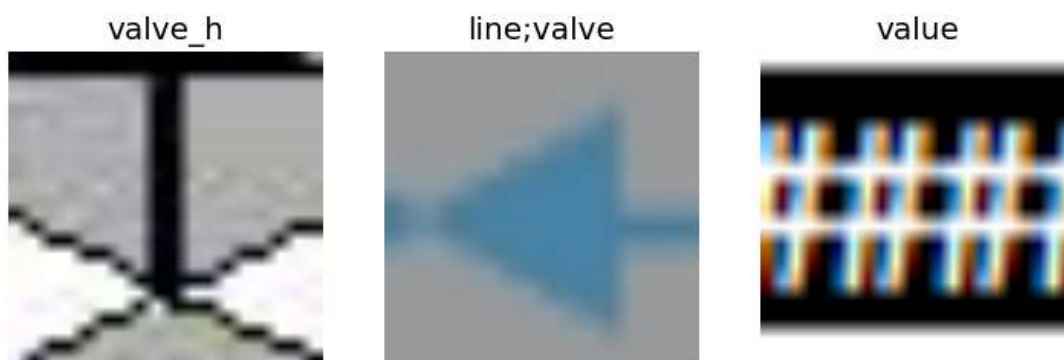
```

Out[10]: 326

```

In [11]: data = DataBlock(blocks=(ImageBlock, MultiCategoryBlock),
                        splitter=splitter,
                        get_x=get_x,
                        get_y=get_y,
                        #item_tfms = Resize(128, ResizeMethod.Pad, pad_mode='zeros')
                        item_tfms = RandomResizedCrop(224, min_scale=0.35))
dls = data.dataloaders(df, bs=16)
dls.show_batch(nrows=1, ncols=3)

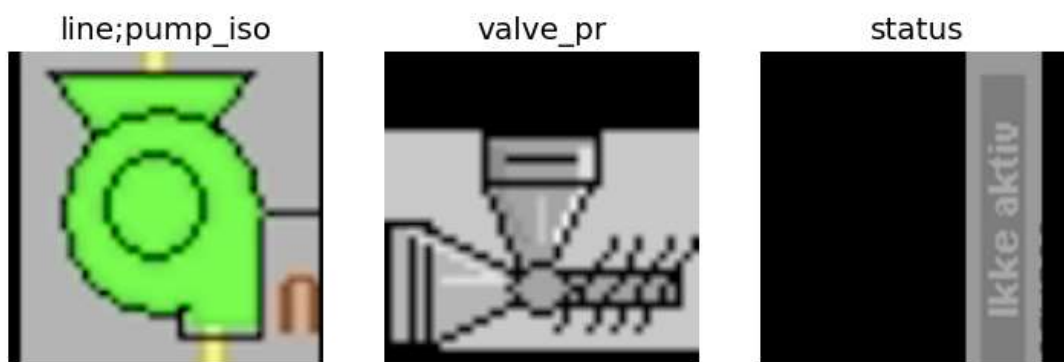
```



```

In [12]: data = data.new(
        item_tfms=Resize(224, ResizeMethod.Pad, pad_mode='zeros'),
        batch_tfms=aug_transforms(size=224, min_scale=1, mult=2,
                                  max_warp=0, do_flip=True,
                                  flip_vert=True, max_zoom=1.0,
                                  pad_mode="zeros", max_rotate=0)
    )
dls = data.dataloaders(df, bs=16)
# Show batch, unique=true will return same object in dif augmentations
dls.show_batch(nrows=1, ncols=3)
#dls.valid.show_batch(max_n=3, nrows=2, unique=True)

```



Train

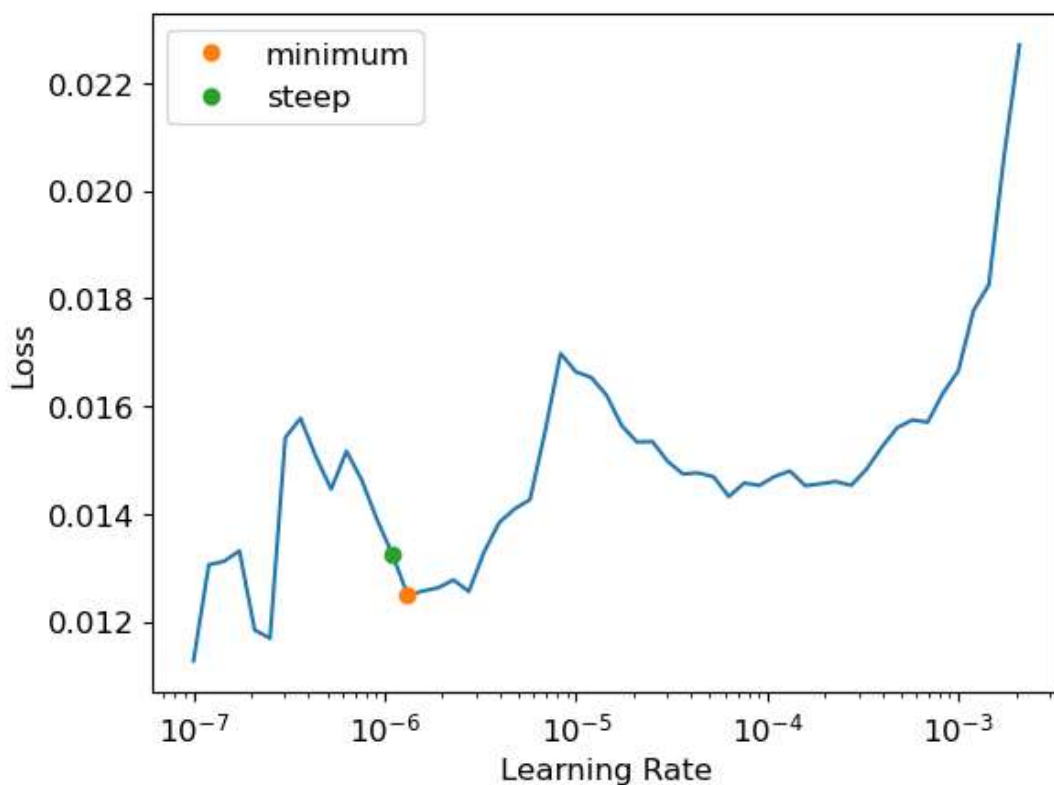
```
In [13]: learn = vision_learner(dls, resnet50, metrics=partial(accuracy_multi, thresh=0.5))  
learn.fine_tune(7, base_lr=3e-3, freeze_epochs=4)
```

epoch	train_loss	valid_loss	accuracy_multi	time
0	0.934290	0.682573	0.615588	00:13
1	0.690487	0.355597	0.887479	00:11
2	0.273667	0.080241	0.971974	00:11
3	0.135597	0.061409	0.979643	00:12

epoch	train_loss	valid_loss	accuracy_multi	time
0	0.079864	0.050636	0.983687	00:16
1	0.068066	0.045865	0.982989	00:16
2	0.051531	0.031916	0.988009	00:16
3	0.039019	0.025347	0.990101	00:16
4	0.029049	0.017391	0.994841	00:16
5	0.020180	0.015425	0.993726	00:16
6	0.017092	0.014262	0.994841	00:16

Learning rate finder

```
In [14]: lr_min,lr_steep = learn.lr_find(suggest_funcs=(minimum, steep))
```

```
In [15]: print(f"Minimum/10: {lr_min:.2e}, steepest point: {lr_steep:.2e}")
Minimum/10: 1.32e-07, steepest point: 1.10e-06
```

Threshold

```
In [16]: # Check the Learning threshold metrics
learn.metrics = partial(accuracy_multi, thresh=0.1)
learn.validate()
```

Out[16]: (#2) [0.014261656440794468,0.9919130206108093]

```
In [17]: learn.metrics = partial(accuracy_multi, thresh=0.99)
learn.validate()
```

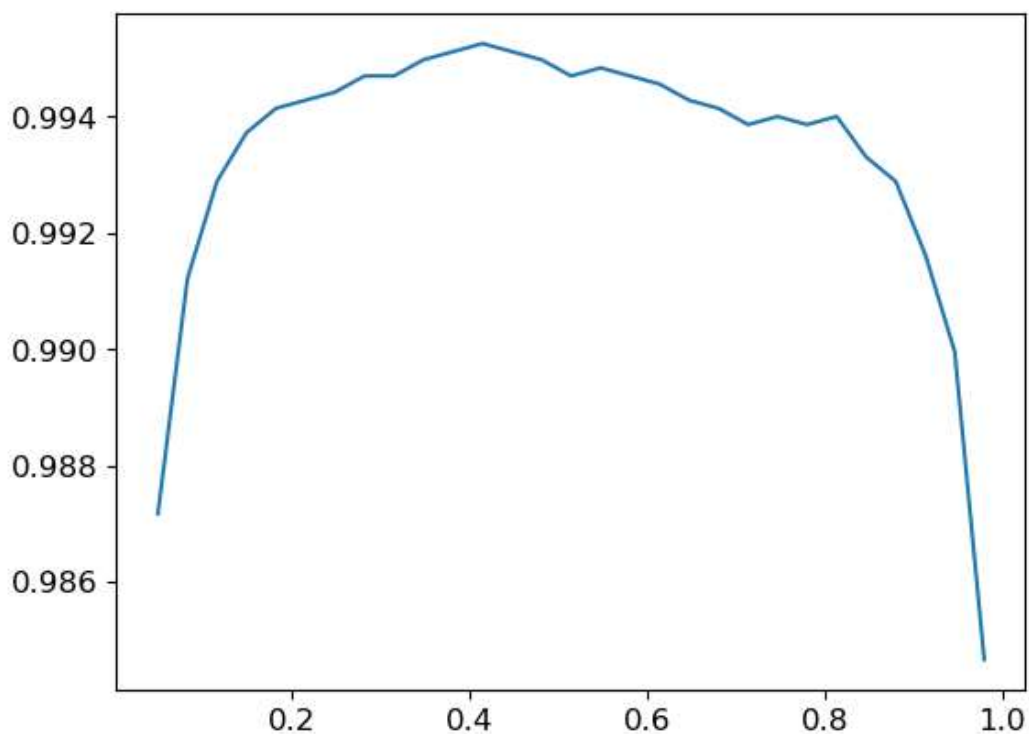
Out[17]: (#2) [0.014261656440794468,0.9796431064605713]

```
In [18]: preds,targs = learn.get_preds()
```

```
In [19]: accuracy_multi(preds, targs, thresh=0.9, sigmoid=False)
#telling it to not apply activation function sigmoid
```

Out[19]: TensorBase(0.9921)

```
In [20]: xs = torch.linspace(0.05,0.98,29)
accs = [accuracy_multi(preds, targs, thresh=i, sigmoid=False) for i in xs]
plt.plot(xs,accs);
```



Re-Train

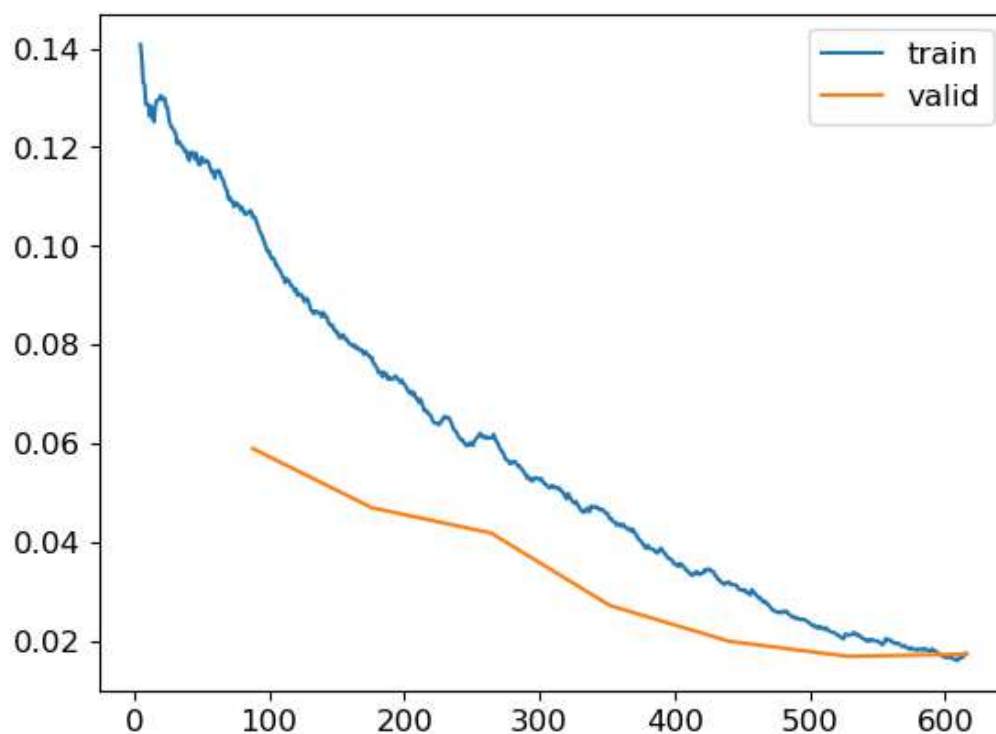
```
In [21]: #Automatically change Learningrate, select num freeze
learn = vision_learner(dls, resnet50, metrics=partial(accuracy_multi, thresh=0.8))
learn.fine_tune(7, base_lr=3e-03, freeze_epochs=4)

#changing the threshold between 90-100 does not really effect the result. Need more
```

epoch	train_loss	valid_loss	accuracy_multi	time
0	0.938557	0.701670	0.906860	00:11
1	0.699822	0.371826	0.969883	00:11
2	0.272820	0.086045	0.967931	00:11
3	0.131944	0.065084	0.974205	00:11

epoch	train_loss	valid_loss	accuracy_multi	time
0	0.079951	0.054881	0.977831	00:16
1	0.065731	0.044130	0.981735	00:16
2	0.055285	0.034544	0.987172	00:16
3	0.039843	0.025235	0.989961	00:16
4	0.029929	0.020573	0.992331	00:16
5	0.021101	0.018158	0.993029	00:16
6	0.016511	0.017393	0.993865	00:16

```
In [19]: learn.recorder.plot_loss()
```



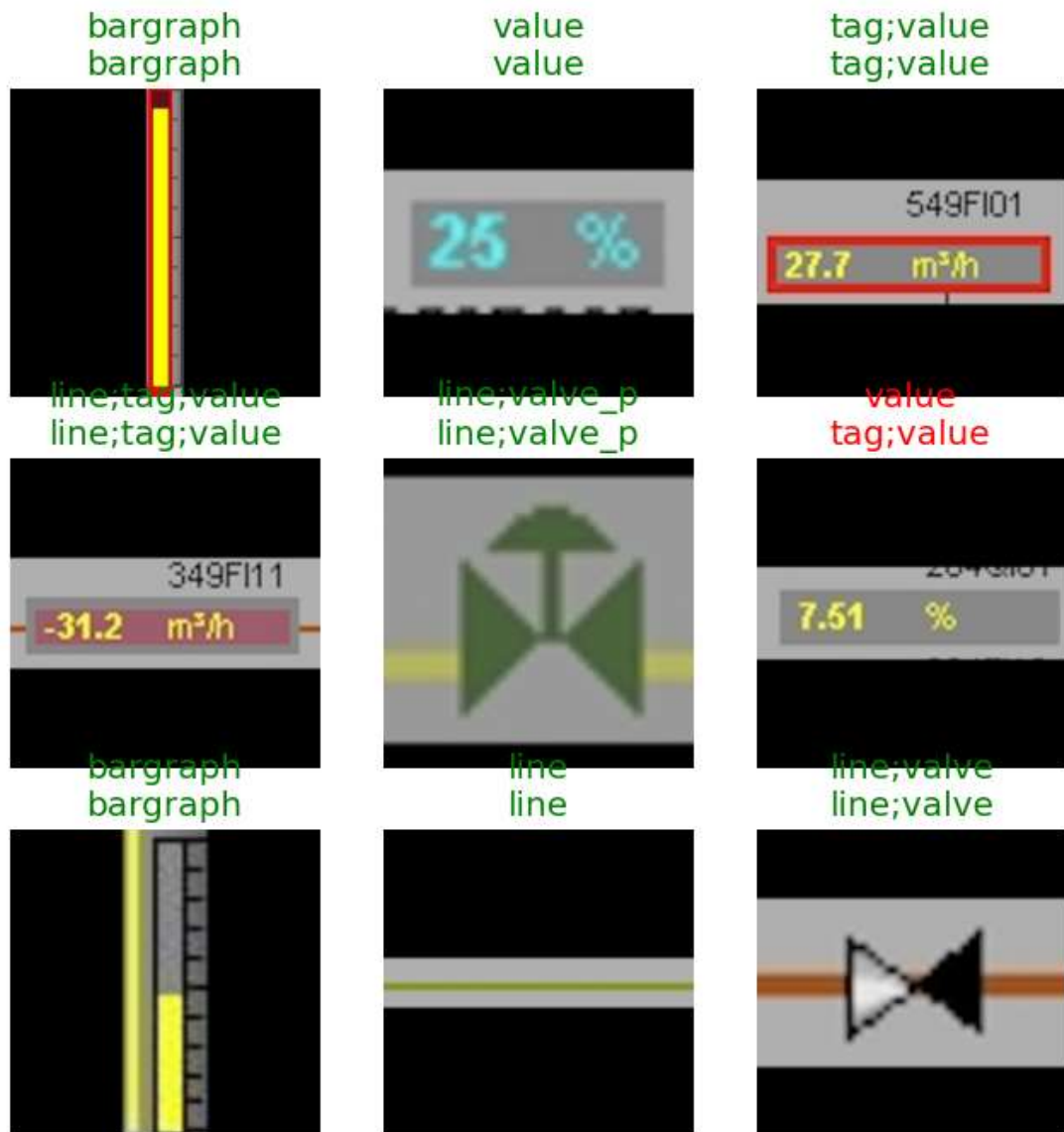
Manually validate training

```
In [22]: interp = ClassificationInterpretation.from_learner(learn)
#interp.plot_multi_top_losses(figsize=(12,12), dpi=60)
#interp.plot_confusion_matrix(figsize=(12,12), dpi=60)
```

```
In [23]: #interp.plot_top_losses(20, nrows=5)
interp.most_confused(min_val=1)
```

```
Out[23]: [('bargraph', 'background', 29), ('background', 'bargraph', 17)]
```

```
In [24]: learn.show_results(ds_idx=1, nrows=3, figsize=(8,8))
```



I would argue that the missclassification above is actually correct as there actually is a tag in the value object (column 3, row 2). So tag;value classification is correct.

Export model

```
In [25]: learn.export(fname="v5_multiobj_classifier.pkl")
```

```
In [26]: #Learn.export??
```

```
In [27]: path = Path()
path.ls(file_exts='.pkl')
```

```
Out[27]: (#3) [Path('v2_multiobj_classifier.pkl'),Path('v1_multiobj_classifier.pkl'),Path('v5_multiobj_classifier.pkl')]
```

```
In [28]: learn_inf = torch.load(path/'v5_multiobj_classifier.pkl', map_location='cuda:0')
#learn_inf = load_learner(path/'v1_multiobj_classifier.pkl').cuda()
```

```
In [29]: num = 42
imP_path = Path('/home/engineirik/git/classify_multi_obj/test/'+str(num)+".PNG")
imP = Image.open(imP_path)
imP
```

```
Out[29]: 
```

```
In [30]: learn_inf.predict(imP_path)
```

```
Out[30]: ((#2) ['pump_isa', 'tag'],
  TensorBase([False, False, False, False, False, False, False, False, False, True,
  False, False, True, False, False, False, False, False, False, False, False]),
  TensorBase([1.3426e-03, 1.7562e-03, 6.2061e-04, 2.5043e-03, 5.1339e-03, 2.6294e-0
  1, 2.2042e-03, 3.2258e-03, 9.4121e-04, 7.2583e-01, 3.1395e-02, 1.9519e-02, 8.4550e
  -01, 8.5509e-02, 3.9538e-04,
  3.8589e-04, 6.0479e-03, 9.7700e-04, 6.3403e-02, 3.1402e-03, 1.7011e-0
  3, 1.9481e-03]))
```

```
In [27]: learn_inf.dls.vocab
```

```
Out[27]: ['background', 'bargraph', 'chart', 'damper', 'fan', 'line', 'mixer', 'motor', 'na
v', 'pump_isa', 'pump_iso', 'status', 'tag', 'value', 'valve', 'valve_3w', 'valve_
h', 'valve_h_3w', 'valve_m', 'valve_m_3w', 'valve_p', 'valve_pr']
```

Save model

```
In [ ]:
```

Appendix G

Pyramid Scaled Sliding Window NMS Classifier

Jupyter Notebook

```
In [ ]: import cv2
import numpy as np
from fastai.vision.all import *
from fastbook import *
from fastai.vision.widgets import *
from fastai.callback.fp16 import *
import pickle
import argparse
import imutils
import os
os.environ['KMP_DUPLICATE_LIB_OK']='True'
import pytesseract
import keras_ocr
import math
import warnings
warnings.filterwarnings("ignore")
```

In this code, the `sliding_window` function generates windows of a certain `window_size` over an input image, with a step of `step_size`. The `classify_image` function uses the `sliding_window` function to generate windows, resizes them to the expected size for the pre-trained model, and classifies each window using the `model.predict` method. The classified windows are stored in the `classified_regions` list and returned.

```
In [9]: # Define paths
path = Path("/home/engineirik/git/classify_multi_obj")
imP_path = Path('/home/engineirik/git/classify_obj/OperateDisplay/')
#imP_path.ls()
```

```
Out[9]: (2) [Path('/home/engineirik/git/classify_obj/OperateDisplay/2022_02_08_PR_276_10_SYRE_EKSP_LAGER.jpg'),Path('/home/engineirik/git/classify_obj/OperateDisplay/2022_02_08_VL_310_10_TANKER.jpg')]
```

```
In [10]: # Defined to remove attribute error in model
def get_x(r): return path/'train'/r['fname']
def get_y(r): return r['labels'].split(' ')
```

```
In [11]: # Load model
learn_inf = torch.load(path/'v2_multiobj_classifier.pkl', map_location='cuda:0')
# Define the window size and step size
>window_size_hz = (80,30)
>window_size_vc = (30,80)
>window_size_sq = (40,40)
>step_size = (13,13) #1313
>WIDTH = 1800
>PYR_SCALE = 1.5

# Load the input image
>image = cv2.imread(str(imP_path)+'/2022_02_08_PR_276_10_SYRE_EKSP_LAGER.jpg')
>image = imutils.resize(image, width=WIDTH)
>orig = image.copy()
>(H, W) = image.shape[:2]
```

```
In [12]: # Function for showing the image in jupyter notebook
def show_rgb_image(image, title=None, conversion=cv2.COLOR_BGR2RGB):

    # Converts from one colour space to the other. this is needed as RGB
    # is not the default colour space for OpenCV
    image = cv2.cvtColor(image, conversion)

    # Show the image
    plt.imshow(image)

    # remove the axis / ticks for a clean looking image
    plt.xticks([])
    plt.yticks([])

    # if a title is provided, show it
    if title is not None:
        plt.title(title)

    plt.show()
```

```
In [13]: # Moves sliding windows
def sliding_window(image, step_size, window_size):
    for y in range(0, image.shape[0]-window_size[1], step_size[1]):
        for x in range(0, image.shape[1]-window_size[0], step_size[0]):
            yield (x, y, image[y:y + window_size[1], x:x + window_size[0]])
```

```
In [14]: # Scales the image in given pyramid scales
def image_pyramid(image, scale=1.5, minSize=(128, 128)):
    # yield the original image
    yield image
    # keep looping over the image pyramid
    while True:
        # compute the dimensions of the next image in the pyramid
        w = int(image.shape[1] / scale)
        image = imutils.resize(image, width=w)
        # if the resized image does not meet the supplied minimum
        # size, then stop constructing the pyramid
        if image.shape[0] < minSize[1] or image.shape[1] < minSize[0]:
            break
        # yield the next image in the pyramid
        yield image
```

```
In [16]: # Initialize the image pyramid
pyramid = image_pyramid(image, scale=PYR_SCALE, minSize=window_size_sq)
```

```
In [17]: rois = []
locs = []
# Classify the image
for image in pyramid:

    # determine the scale factor between the *original* image
    # dimensions and the *current* layer of the pyramid
    scale = W / float(image.shape[1])
```



```

for (x, y, window) in sliding_window(image, step_size, window_size_sq):
    if window.shape[0] != window_size_sq[1] or window.shape[1]
       != window_size_sq[0]:
        continue

    # scale the (x, y)-coordinates of the ROI with respect to the
    # *original* image dimensions
    x = int(x * scale)
    y = int(y * scale)
    w = int(window_size_sq[0] * scale)
    h = int(window_size_sq[1] * scale)

    # Resize the window to the size expected by the model
    window = cv2.resize(window, (224,224))
    window = np.array(window)
    rois.append(window)
    locs.append((x, y, x + w, y + h))

```

```

In [18]: # Load data and predict using the Multi-Label classifier model
test_dl = learn_inf.dls.test_dl(rois)
preds = learn_inf.get_preds(dl=test_dl)

```

```

In [19]: labels = learn_inf.dls.vocab
label = []
score = []
classified_regions = []
for i in range(len(preds[0])):
    x1,y1,x2,y2 = locs[i]
    label = (labels[preds[0][i].argmax()]) #.argmax orig
    score = (preds[0][i].max())
    classified_regions.append((x1, y1, x2, y2, label, score))
#score
#Locs
#classified_regions[2]

```

```

In [20]: # Number of classifications
len(classified_regions)

```

```

Out[20]: 14416

```

To merge similar bounding boxes into one bounding box with one label, you can use a technique called non-maximum suppression. The idea is to compare each bounding box with all other bounding boxes and remove those that have a high overlap with another bounding box.

```

In [31]: # Used for NMS, merging/removing overlapping boxes with low score
def merge_bounding_boxes(bboxes, scores, scoreThreshold=0.1, nms_threshold=0.1):
    # create a list to store the indices of the bounding boxes to keep
    keep = []
    # Convert the bounding boxes to a format that can be used by the
    # cv2.dnn.NMSBoxes function
    #bboxes = [box.astype("int") for box in bboxes]

```

```

bboxes = [np.around(box).astype("int") for box in bboxes]
# use the cv2.dnn.NMSBoxes function to suppress overlapping bounding boxes
scores = np.array(scores, dtype="float")
indices = cv2.dnn.softNMSBoxes(bboxes, scores, scoreThreshold, nms_threshold)
# keep the indices of the bounding boxes that were not suppressed
for i in indices:
    keep.append(i)
# return the indices of the bounding boxes to keep
return keep

```

No NMS

```

In [32]: # Draw bounding boxes around the classified regions, show all without NMS
boxes = []
scores = []
labels = []

copy3_image = orig.copy()

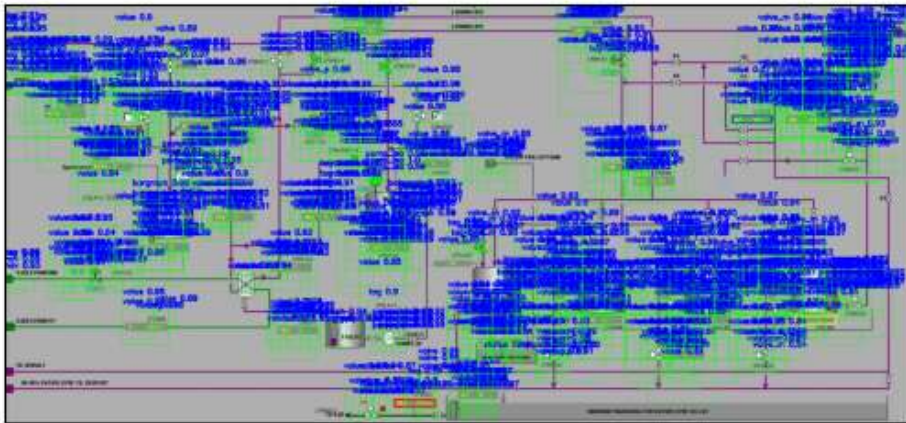
for (x1, y1, x2, y2, label, score) in classified_regions:
    if score >= 0.8 and label != "chart" and label != "line" and label != "arrow":
        boxes.append([x1, y1, x2, y2])
        scores.append(score)
        labels.append(label)
        cv2.rectangle(copy3_image, (x1,y1), (x2,y2), (0,255,0), 1)
        cv2.putText(copy3_image, str(label)+" "+str(round(float(score), 2)),
                    (x1, y1-10), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255, 0, 0), 2)

```

```

In [33]: cv2.imwrite("v2_NONMS.jpg", copy3_image)
show_rgb_image(copy3_image)

```



NMS

```

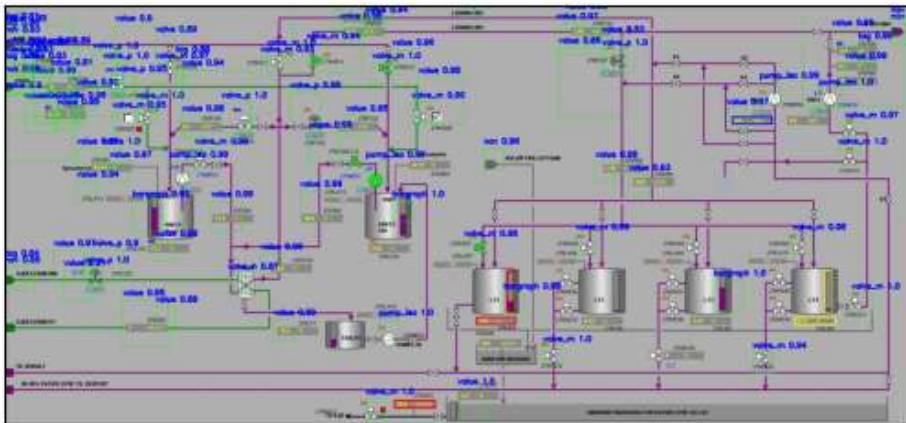
In [34]: # Copy boxes before NMS
keep = merge_bounding_boxes(boxes, scores)
bboxes = [boxes[i] for i in keep[1]]

```

```
bcores = [scores[i] for i in keep[1]]
bblabel = [labels[i] for i in keep[1]]
bcores = np.array(bcores, dtype="float")
```

```
In [35]: # Draw bounding boxes around the classified regions, only the one with highest score
copy4_image = orig.copy()
i = 0
for (x1, y1, x2, y2) in bboxes:
    cv2.rectangle(copy4_image, (x1,y1), (x2,y2), (0,255,0), 1)
    cv2.putText(copy4_image, str(bblabel[i])+" "+str(round(bcores[i], 2)),
                (x1, y1-10), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255, 0, 0), 2)
    i+=1
```

```
In [36]: cv2.imwrite("v2_NMS.jpg", copy4_image)
show_rgb_image(copy4_image)
```



NMS label

```
In [37]: # Group the boxes by label
grouped_boxes = defaultdict(list)
for box, label in zip(bboxes, labels):
    grouped_boxes[label].append(box)

print(len(grouped_boxes))
```

10

```
In [38]: # Group the scores by boxes
grouped_scores = defaultdict(list)
for score, label in zip(scores, labels):
    grouped_scores[label].append(score)

print(len(grouped_scores))
```

10

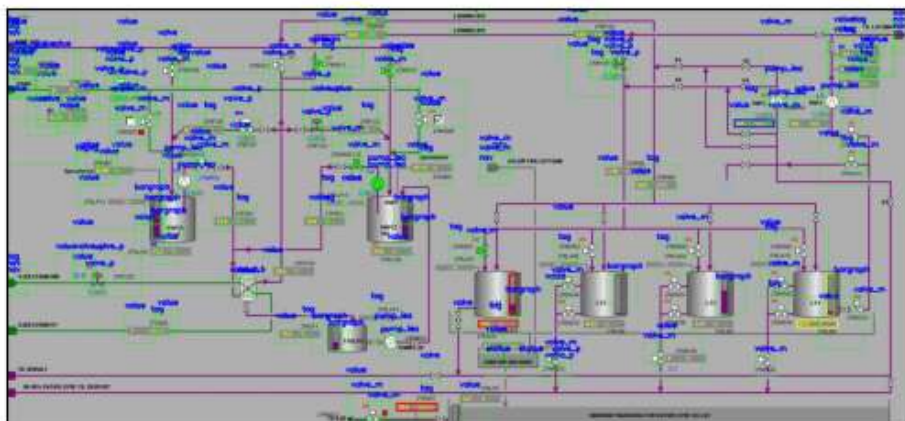
```
In [39]: # Perform NMS on each group
copy5_image = orig.copy()
```

```
for label, boxes in grouped_boxes.items():
    result = []
    scores = []

    #scores = [0.95] * len(boxes) # score of each box, set to 1.0 for simplicity
    scores = grouped_scores[label]
    scores = np.array(scores, dtype="float")
    indices = cv2.dnn.softNMSBoxes(boxes, scores, score_threshold=0.1, nms_threshol

    #print(scores)
    result.extend([boxes[i] for i in indices[1]])
    for (x1, y1, x2, y2) in result:
        cv2.rectangle(copy5_image, (x1,y1), (x2,y2), (0,255,0), 1)
        cv2.putText(copy5_image, label, (x1, y1-10),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255, 0, 0), 2)
```

```
In [40]: cv2.imwrite("v2_LNMS.jpg", copy5_image)
         show_rgb_image(copy5_image)
```



Appendix I

Split Image Annotation YOLO Prep

Jupyter Notebook

```
In [ ]: import cv2
import numpy as np
import matplotlib.pyplot as plt
import os
```

```
In [ ]: def show_rgb_image(image, title=None, conversion=cv2.COLOR_BGR2RGB):
    # Converts from one colour space to the other. this is needed as RGB
    # is not the default colour space for OpenCV
    image = cv2.cvtColor(image, conversion)
    # Show the image
    plt.imshow(image)
    # remove the axis / ticks for a clean looking image
    plt.xticks([])
    plt.yticks([])
    # if a title is provided, show it
    if title is not None:
        plt.title(title)
    plt.show()
```

```
In [ ]: def draw_annotation(image, annotation_file):
    img_height, img_width = image.shape[:2]
    with open(annotation_file, 'r') as f:
        annotation = f.readlines()
    for line in annotation:
        data = line.split()
        x_center, y_center, w, h = map(float, data[1:])
        # Convert normalized coordinates to pixel coordinates
        x = int((x_center - w/2) * img_width)
        y = int((y_center - h/2) * img_height)
        w = int(w * img_width)
        h = int(h * img_height)
        # Draw bounding box
        cv2.rectangle(image, (x, y), (x+w, y+h), (0, 255, 0), 2)
```

```
In [ ]: def split_image(img_path, annotation_path, img_output_path, ann_output_path):
    # Load image and annotation data
    img = cv2.imread(img_path)
    with open(annotation_path, 'r') as f:
        annotation = f.readlines()

    # Get image width and height
    img_height, img_width = img.shape[:2]

    # Split image in half
    left_img = img[:, :img_width//2, :]
    right_img = img[:, img_width//2:, :]
    c_left_img = left_img.copy()
    c_right_img = right_img.copy()

    # Calculate new image widths
    left_width = left_img.shape[1]
    right_width = right_img.shape[1]

    # Split annotation data accordingly
    left_annotation = []
    right_annotation = []
    for line in annotation:
```

```

data = line.split()
x_center, y_center, w, h = map(float, data[1:])
x = (x_center - w/2) * img_width
y = (y_center - h/2) * img_height
w = w * img_width
h = h * img_height
x_center_p = x_center * img_width
y_center_p = y_center * img_height

#x_center /= img_width # divide by full image width
#y_center /= img_height # divide by full image height
#w /= img_width # divide by full image width
#h /= img_height # divide by full image height

if x + w < 0.5*img_width:
    left_annotation.append('{:} {:.6f} {:.6f} {:.6f} {:.6f}\n'.format
        (data[0],
         x_center_p/left_width,
         y_center_p/img_height,
         w/left_width,
         h/img_height))
elif x >= 0.5*img_width:
    right_annotation.append('{:} {:.6f} {:.6f} {:.6f} {:.6f}\n'.format
        (data[0],
         (x_center_p-left_width)/right_width,
         y_center_p/img_height,
         w/right_width,
         h/img_height))
else:
    # Annotation is split in half, need to update coordinates
    if x < 0.5*img_width:
        x_left = x
        w_left = (0.5*img_width)-x
        if w_left > 0.5*w:
            x_center_p = (x_left + w_left/2)
            y_center_p = y_center * img_height
            left_annotation.append('{:} {:.6f} {:.6f} {:.6f} {:.6f}\n'
                .format
                (data[0], x_center_p/left_width,
                 y_center_p/img_height,
                 w_left/left_width, h/img_height))

    if x + w > 0.5*img_width:
        x_right = 0
        w_right = ((x + w) - (left_width))
        if w_right > 0.5*w:
            x_center_p = (w_right / 2)
            right_annotation.append('{:} {:.6f} {:.6f} {:.6f} {:.6f}\n'
                .format
                (data[0], x_center_p/right_width,
                 y_center_p/img_height,
                 w_right/right_width, h/img_height))

    #if x < 0.5*img_width and x + w > 0.5*img_width:
    #left_annotation.append(Line)
    #right_annotation.append(Line)

# Save split images and annotations
img_filename = os.path.basename(img_path)

```

```

img_basename, img_extension = os.path.splitext(img_filename)
left_img_path = os.path.join(img_output_path, img_basename +
                              '_left' + img_extension)
right_img_path = os.path.join(img_output_path, img_basename +
                               '_right' + img_extension)
cv2.imwrite(left_img_path, left_img)
cv2.imwrite(right_img_path, right_img)

ann_filename = os.path.basename(annotation_path)
ann_basename, ann_extension = os.path.splitext(ann_filename)
left_ann_path = os.path.join(ann_output_path, ann_basename +
                              '_left' + ann_extension)
right_ann_path = os.path.join(ann_output_path, ann_basename +
                               '_right' + ann_extension)
with open(left_ann_path, 'w') as f:
    f.writelines(left_annotation)
with open(right_ann_path, 'w') as f:
    f.writelines(right_annotation)

# Draw annotations on split images
draw_annotation(c_left_img, left_ann_path)
draw_annotation(c_right_img, right_ann_path)

# Show split images with annotations
show_rgb_image(c_left_img, title=img_path[:-4]+'_left')
show_rgb_image(c_right_img, title=img_path[:-4]+'_right')

```

```

In [ ]: # Splitt just one image
#img = "yolo/train/images/2022_02_08_AB_268_10_SYRE_ABS_analyzed.png"
#ann = "yolo/train/Labels/2022_02_08_AB_268_10_SYRE_ABS_analyzed.txt"
#split_image(img, ann)

```

```

In [ ]: img_dir = "yolo/train/images"
ann_dir = "yolo/train/labels"
img_split_dir = "yolo/train/images_s"
ann_split_dir = "yolo/train/labels_s"

# Get lists of image and annotation file paths
img_files = os.listdir(img_dir)
ann_files = os.listdir(ann_dir)

```

```

In [ ]: # Loop through image files and call split_image on corresponding annotation file
for img_file in img_files:
    if img_file.endswith('.png'):
        # Get corresponding annotation file
        ann_file = img_file.replace('.png', '.txt')
        if ann_file in ann_files:
            img_path = os.path.join(img_dir, img_file)
            ann_path = os.path.join(ann_dir, ann_file)
            split_image(img_path, ann_path, img_split_dir, ann_split_dir)

```


Appendix J

OCR Prep

Python code


```
# Convert the image to grayscale
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

# Get the height and width of the image
height, width = gray.shape[:2]

# Divide the height and width by 2 to get the dimensions of each quadrant
h = height // 2
w = width // 2

# Create an array to store the 4 quadrants of the image
quadrants = [gray[:h, :w], gray[h, w:], gray[h:, :w], gray[h:, w:]]

# Create a list to store the annotations
annotations = []

# Set the minimum confidence level to 50%
conf_level = 10

# Set the Pytesseract configuration parameters
config = f"--psm 6 --oem 3 -c min_confidence_level={conf_level}"

# Define the scaling factors
scales = [1.5, 2, 4]

def get_range(threshold, sigma=0.33):
    return (1-sigma) * threshold, (1+sigma) * threshold

for scale in scales:
    # Loop over the quadrants
    for j, quadrant in enumerate(quadrants):
        # Randomly scale and rotate the image
        upscaled = cv2.resize(quadrant, None, fx=scale, fy=scale,
interpolation=cv2.INTER_LINEAR)

        #upscaled = cv2.resize(scaled, None, fx=4, fy=4,
interpolation=cv2.INTER_LINEAR)
        q_height, q_width = upscaled.shape[:2]
        # Apply a Laplacian filter to sharpen the image
        laplacian = cv2.Laplacian(upscaled, cv2.CV_8U) #test
        sharpened = cv2.addWeighted(upscaled, 1.5, laplacian, -0.5, 0) #test

        # Apply thresholding to create a binary image
        thresh = cv2.threshold(sharpened, 170, 255, cv2.THRESH_BINARY_INV)[1]
```

```
# Apply kernel to dilate the image
kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (1,1))
# Invert the Canny edges image
edges_inverted = cv2.bitwise_not(thresh)
# Apply dilation to make text more visible
dilated = cv2.dilate(edges_inverted, kernel, iterations=1)

cv2.imwrite(f'img/quadrant_{j}.jpg', dilated)

# Use pytesseract to extract text and bounding boxes from the image
data = pytesseract.image_to_data(dilated,
output_type=pytesseract.Output.DICT, config=config, lang=None) #config=config)
#print(data['text'])

# Loop over the words and concatenate bounding boxes that are close
together
for i in range(len(data['text'])):
    # Extract the text and bounding box coordinates
    text = data['text'][i]
    x, y, w, h = data['left'][i], data['top'][i], data['width'][i],
data['height'][i]

    # Apply the scaling factor used in the loop
    x = x / (scale)
    y = y / (scale)
    w = w / (scale)
    h = h / (scale)

    # Rescale the coordinates and dimensions of the bounding boxes
    if j == 0: # Top-left quadrant
        x_offset = 0
        y_offset = 0
    elif j == 1: # Top-right quadrant
        x_offset = width/2
        y_offset = 0
    elif j == 2: # Bottom-left quadrant
        x_offset = 0
        y_offset = height/2
    else: # Bottom-right quadrant
        x_offset = width/2
        y_offset = height/2

    x_center = (x + x_offset) / width
    y_center = (y + y_offset) / height
    box_width = w / width
```

```
box_height = h / height

if not text:
    continue
if len(text) < 3:
    continue
#if text.replace(" ", "") == "":
#continue

# Check if the text matches any of the desired patterns
matches_pattern = False

for pattern in patterns:
    if re.match(pattern, text):
        matches_pattern = True
        break

if not matches_pattern:
    if re.match(r'\d{3}', text):
        if i+1 < len(data['text']):
            text2 = data['text'][i+1]
            if re.match(r'\d{2}', text2):
                if i+2 < len(data['text']):
                    text3 = data['text'][i+2]
                    if re.match(r'\d{2}-\d{2}', text3):
                        text = text + text2 + "_" + text3
                        matches_pattern = True
                elif re.match(r'\d{2},\d{2}', text3):
                    if i+2 < len(data['text']):
                        text3 = data['text'][i+2]
                        if re.match(r'\d{2}-\d{2}', text3):
                            text = text + text2 + "_" + text3
                            matches_pattern = True
                    elif re.match(r'\d{2}', text3):
                        if i+2 < len(data['text']):
                            text3 = data['text'][i+2]
                            if re.match(r'\d{4}-\d{2}', text3):
                                text = text + "," + text2 + "_" + text3
                                matches_pattern = True
                    elif re.match(r'\d{2},\d{2}', text3):
                        text = text + text2
                        matches_pattern = True
                elif re.match(r'\d{3},\d{2},\d{2}', text3):
                    if i+1 < len(data['text']):
                        text2 = data['text'][i+1]
```

```
        if re.match(r'\d{2}-\d{2}', text2):
            text = text + "_" + text2
            matches_pattern = True
    elif re.match(r'\d{3},\d{2}', text):
        if i+1 < len(data['text']):
            text2 = data['text'][i+1]
            if re.match(r',\d{2}', text2):
                if i+2 < len(data['text']):
                    text3 = data['text'][i+2]
                    if re.match(r'\d{2}-\d{2}', text3):
                        text = text + text2 + "_" + text3
                        matches_pattern = True
    elif re.match(r'\d{2}', text2):
        if i+2 < len(data['text']):
            text3 = data['text'][i+2]
            if re.match(r'\d{2}-\d{2}', text3):
                text = text + "," + text2 + "_" + text3
                matches_pattern = True

    if not matches_pattern:
        continue

    if any(text in annotation for annotation in annotations):
        continue
    else:
        print(text + " " + str(j))
        # Add the annotation to the list
        annotations.append(f"{text} {x_center:.6f} {y_center:.6f}
{box_width:.6f} {box_height:.6f}")

# Save the image with the bounding boxes
img.save('image_with_boxes.jpg')

# Save the annotations to a text file
with open('annotations.txt', 'w') as f:
    f.write('\n'.join(annotations))

# Copy image
copy_img = image.copy()

# Load the bounding box data from the text file CHECK
with open("/home/engineirik/git/ocr/annotations.txt") as f:
    lines = f.readlines()[1:] # Skip the header line
    for line in lines:
```

```
cols = line.strip().split()
x, y, w, h = map(float, cols[1:5])

# Scale the coordinates to the image size
x = x * width
y = y * height
w = w * width
h = h * height

# Draw a rectangle around the object
cv2.rectangle(copy_img, (int(x), int(y)), (int(x+w), int(y+h)), (0, 255,
0), 2)

# Display the image
cv2.imshow("Image with bounding boxes", copy_img)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Appendix L

Custom mAP Calculation

Python code


```
import numpy as np
import os

def compute_iou(box1, box2):
    # Calculate the intersection rectangle
    x1 = max(box1[0], box2[0])
    y1 = max(box1[1], box2[1])
    x2 = min(box1[0]+box1[2], box2[0]+box2[2])
    y2 = min(box1[1]+box1[3], box2[1]+box2[3])
    inter_area = max(0, x2-x1) * max(0, y2-y1)

    # Calculate the union area
    box1_area = box1[2] * box1[3]
    box2_area = box2[2] * box2[3]
    union_area = box1_area + box2_area - inter_area

    # Calculate the IoU
    iou = inter_area / union_area

    return iou

def compute_precision_recall(yolo_data, annotated_data, class_id, iou_threshold):
    tp = 0
    fp = 0
    fn = 0

    num_annotated_objects = np.sum(annotated_data[:, 0] == class_id)

    for i in range(len(yolo_data)):
        if yolo_data[i][0] != class_id:
            continue

        yolo_box = [yolo_data[i][1], yolo_data[i][2], yolo_data[i][3], yolo_data[i][4]]
        max_iou = 0

        for j in range(len(annotated_data)):
            if annotated_data[j][0] != class_id:
                continue
```

```
    annotated_box = [annotated_data[j][1], annotated_data[j][2], annotated_data[j][3], annotated_data[j][4]]
    iou = compute_iou(yolo_box, annotated_box)
    if iou > max_iou:
        max_iou = iou

    if max_iou >= iou_threshold:
        tp += 1
    else:
        fp += 1

fn = num_annotated_objects - tp

if tp + fp > 0:
    precision = tp / (tp + fp)
else:
    precision = 0

recall = tp / (tp + fn)

return precision, recall

def compute_mAP(yolo_file, annotated_file, iou_threshold=0.50):
    yolo_data = np.loadtxt(yolo_file, delimiter=' ')
    annotated_data = np.loadtxt(annotated_file, delimiter=' ')
    class_ids = np.unique(annotated_data[:, 0])
    num_classes = len(class_ids)

    aps = []
    for i, class_id in enumerate(class_ids):
        precision, recall = compute_precision_recall(yolo_data, annotated_data, class_id, iou_threshold)

        ap = 0
        for j in range(11):
            threshold = j / 10
            if recall >= threshold:
```

```
max_precision = 0
for k in range(len(yolo_data)):
    if yolo_data[k][0] != class_id:
        continue

    yolo_box = [yolo_data[k][1], yolo_data[k][2], yolo_data[k][3], yolo_data[k][4]]
    max_iou = 0
    for l in range(len(annotated_data)):
        if annotated_data[l][0] != class_id:
            continue
        annotated_box = [annotated_data[l][1], annotated_data[l][2], annotated_data[l][3], annotated_data[l][4]]
        iou = compute_iou(yolo_box, annotated_box)
        if iou > max_iou:
            max_iou = iou
    if max_iou >= iou_threshold:
        tp = 1
        fp = 0
        precision = tp / (tp + fp)
        if precision > max_precision:
            max_precision = precision

    ap += max_precision / 11

aps.append(ap)

mAP = np.mean(aps)

return mAP

annotated_folder = 'annotated'
preanalyzed_folder = 'preanalyzed'
iou_threshold = 0.5

avgMAP = 0
numFiles = 0
```

```
for annotated_file in os.listdir(annotated_folder):
    if not annotated_file.endswith('.txt'):
        continue
    preanalyzed_file = os.path.join(preanalyzed_folder, annotated_file)

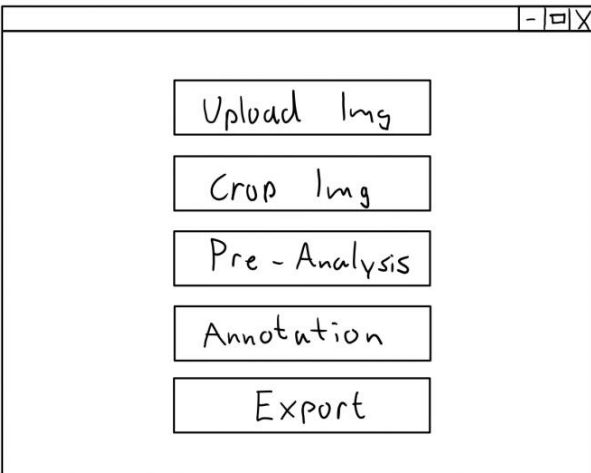

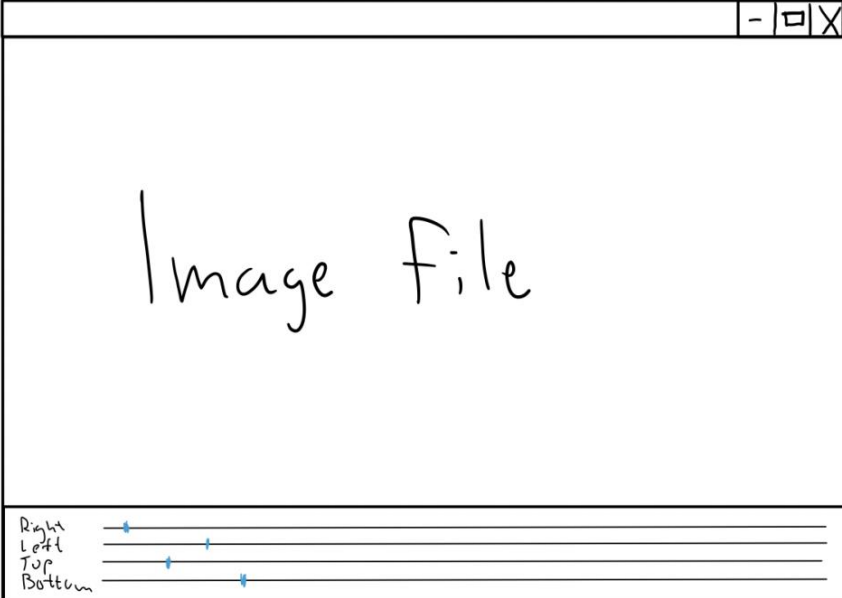
    if not os.path.exists(preanalyzed_file):
        print(f'Error: preanalyzed file {preanalyzed_file} not found')
        continue

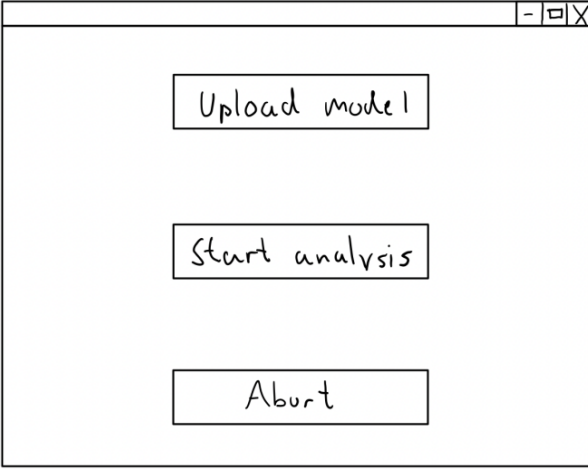
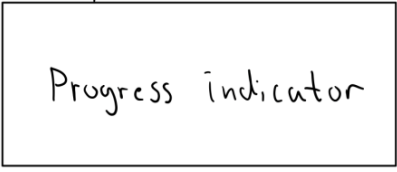
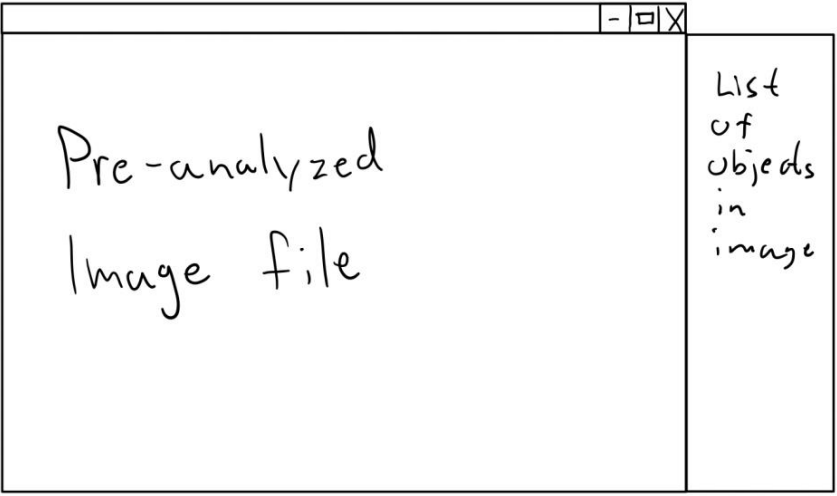

    mAP = compute_mAP(preanalyzed_file, os.path.join(annotated_folder, annotated_file), iou_threshold)
    avgMAP += mAP
    numFiles += 1
    print(f'mAP for file {annotated_file}: {mAP}')

if numFiles > 0:
    avgMAP /= numFiles
    print(f'Average mAP: {avgMAP}')
else:
    print('No files processed')
```

Appendix M
Semi-Automated Annotation
Tool Mockup Design

Table 1: Step by step design mockup of annotation software.



Start menu window	 A hand-drawn mockup of a start menu window. It features a title bar with a close button (X) on the right. The main area contains five vertically stacked rectangular buttons with the following text: "Upload Img", "Crop Img", "Pre - Analysis", "Annotation", and "Export".
Upload image prompt	 A hand-drawn mockup of an upload image prompt. It consists of a single rectangular box containing the text "Open a folder" on the top line and "PROMPT" on the bottom line.
Crop image window	 A hand-drawn mockup of a crop image window. It has a title bar with a close button (X) on the right. The main area contains the text "Image file" in the center. At the bottom, there is a crop control interface with four horizontal lines and blue arrows. The labels "Right", "Left", "Top", and "Bottom" are written vertically on the left side of these lines.

Pre-Analysis window	 A window mockup with a title bar containing a minus sign, a maximize button, and a close button. The window contains three buttons stacked vertically: "Upload model", "Start analysis", and "Abort".
Progress indicator prompt	 A rectangular box containing the text "Progress indicator".
Annotation window	 A window mockup with a title bar containing a minus sign, a maximize button, and a close button. The window is split into two vertical panels. The left panel contains the text "Pre-analyzed Image file". The right panel contains the text "List of objects in image".
Folder save export prompt	 A rectangular box containing the text "Open a folder PROMPT".

Appendix N

UI Figma Design ICE Software

Table 1: Different UI designs for the ICE software

Blue mobile	 The image shows a mobile application interface with a dark blue background. At the top and bottom, there are decorative patterns of white circuit lines and nodes. In the center, there is a white rounded rectangle containing the text "Display Analysis" in bold. Below this, there is a button labeled "Upload Image" with a small icon of a camera. Underneath the button is a text input field with the placeholder text "File name". Below the input field is another button labeled "Analysis".
Light mobile	 The image shows a mobile application interface with a light grey background. At the bottom, there are decorative patterns of light grey circuit lines and nodes. In the center, there is a white rounded rectangle containing the text "Display Analysis" in bold. Below this, there is a button labeled "Upload Image" with a small icon of a camera. Underneath the button is a text input field with the placeholder text "File name". Below the input field is another button labeled "Analysis".

