CONSTRUCTION SITE OBJECT DETECTION API COMPARATIVE STUDY

WONG YEW CHIEN

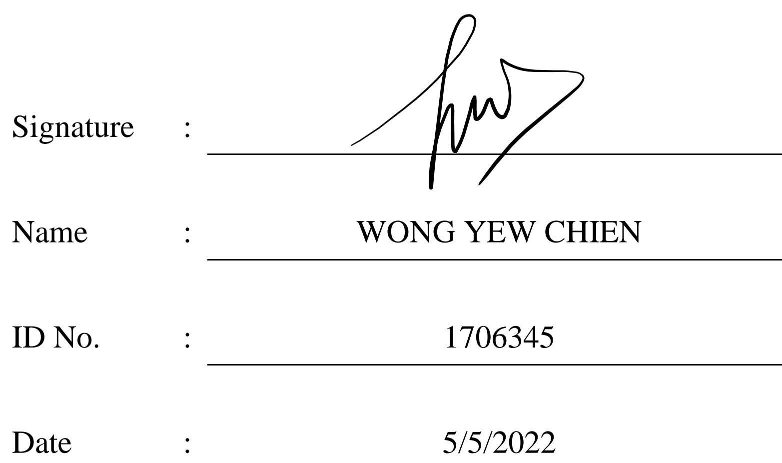
A project report submitted in partial fulfilment of the requirements for the award of Bachelor of Engineering (Hons.) Civil Engineering

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

April 2022

DECLARATION

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



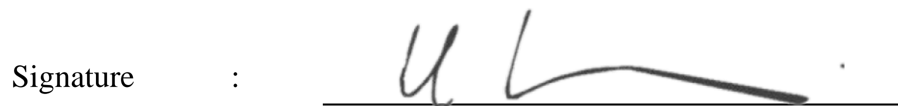
Signature	:	
Name	:	WONG YEW CHIEN
ID No.	:	1706345
Date	:	5/5/2022

APPROVAL FOR SUBMISSION

I certify that this project report entitled **"CONSTRUCTION SITE OBJECT DETECTION API COMPARATIVE STUDY"** was prepared by **WONG YEW CHIEN** has met the required standard for submission in partial fulfilment of the requirements for the award of Bachelor of Engineering (Hons.) Civil Engineering at Universiti Tunku Abdul Rahman.

Approved by,

Signature :



Supervisor : DR. LING LLOYD

Date : 5/5/2022

Signature :

Co-Supervisor : DR. YONG YOKE LENG

Date : 5/5/2022

The copyright of this report belongs to the author under the terms of the copyright Act 1987 as qualified by Intellectual Property Policy of Universiti Tunku Abdul Rahman. Due acknowledgement shall always be made of the use of any material contained in, or derived from, this report.

© 2022, WONG YEW CHIEN. All right reserved.

ACKNOWLEDGEMENTS

I would like to express my gratitude to everyone who has encouraged and helped me to complete this project, especially my brother Wong Yew Lee. Special thanks to my research supervisors, Dr. Ling Lloyd and Dr. Yong Yoke Leng for their invaluable and paramount care, advice, guidance, and patience throughout the development of the research.

Could not have done it without you all.

ABSTRACT

Application of object detection in the construction industry has been extensive, albeit not rich. There is a huge potential in the implementation of object detection. In this project, three object detection models – YOLOv3, Detectron2, and EfficienDet are coded with Python on Google Colab. A modified image dataset that contains 100 images of construction object were prepared. The object classes were person, helmet, vest, and slogan. The image dataset was pre-processed on Roboflow, before being fed into the model. Two out of three models produced meaningful results, whereas one required more data in order to produce results with acceptable level of accuracy. The more data hungry model was dissected and analysed. YOLOv4 model took 1 hour 30 minutes and 4 seconds to train, Detectron2 took 2 hours 42 minutes and 4 seconds, and EfficientDet took 28 minutes and 38 seconds. The longer training time corresponds to a lesser average detection time. YOLOv4 took 4143 milliseconds to detect an image, Detectron2 took 742 milliseconds, and EfficientDet took 305 milliseconds. EfficientDet were the fastest because it does not need to be accurate. The results also suggested that some object detectors were better to detect specific objects. For instance, YOLOv4 can detect hat and slogan better than Detectron2, whereas the latter model can detect person and vest better. Therefore, it is recommended that different object detectors are used depending on the objective of the detection. Among the three object detectors, the highest achieved accuracy for person, hat, slogan, and vest is 94.44%, 78.00%, 81.25%, and 60.00% respectively. All three models successfully detect small and large sized objects. YOLOv4 and Detectron2 were suitable to be applied in the field, but not the EfficientDet model. Surprisingly, the model was able to detect objects that were unannotated in the pre-processing phase. This indicated that the model received enough data on some classes to make predictions by itself. Despite that, more data should be provided in order to produce a more robust object detector. This also means that a better hardware should be acquired to provide better computational power. In this study, the deployment phase was not included, due to the limitation of resources. However, this report shows the necessary steps needed in order to code a working object detector to detect custom image data. In the future, it is recommended that more object classes were to be detected, and the model should move into deployment phase to study its real time accuracy.

TABLE OF CONTENTS

DECLARATION	**ii**
APPROVAL FOR SUBMISSION	**iii**
ACKNOWLEDGEMENTS	**v**
ABSTRACT	**vi**
TABLE OF CONTENTS	**vii**
LIST OF TABLES	**x**
LIST OF FIGURES	**xi**
LIST OF APPENDICES	**xii**

CHAPTER

1	**INTRODUCTION**		**1**
	1.1	General Introduction	1
	1.2	Importance of the Study	2
	1.3	Problem Statement	3
	1.4	Aims and Objectives	3
	1.5	Scope and Limitation of the Study	3
	1.6	Contribution of the Study	4
	1.7	Outline of the Report	4
2	**LITERATURE REVIEW**		**5**
	2.1	Introduction	5
	2.2	Computer vision in Construction Industry	5
	2.3	Image data set	6
	2.4	Performance comparison for various object detector	7
	2.5	Summary	8
3	**METHODOLOGY AND WORK PLAN**		**9**

3.1 Introduction 9
3.2 Overview of the Work Plan 9
3.3 Reviewing of coding platform and parameters 10
3.4 Image gathering 11
3.5 Image pre-processing 11
3.6 Coding 13
3.6.1 Downloading and install model dependencies 13
3.6.2 Importing processed images 14
3.6.3 Determining the upper limit of training parameters 14
3.6.4 Training the model 14
3.6.5 Saving and using the trained model weights 15
3.6.6 Test image inference 15
3.6.7 Result and analysis 16
3.7 Summary 17

4 RESULTS AND DISCUSSIONS 18
4.1 Introduction 18
4.2 Image dataset health check 18
4.2.1 Images and annotations health 18
4.2.2 Class balance check 18
4.2.3 Image size and aspect ratio check 19
4.2.4 Annotations heatmap 19
4.3 Results 20
4.4 EfficientDet 20
4.5 Training time and average detection time 21
4.6 The need for more data? 21
4.7 Accuracy for person 22
4.8 Accuracy for hat 22
4.9 Accuracy for slogan 22
4.10 Accuracy for vest 22
4.11 Input image resolution 23
4.12 Overlapping of detected objects 23

4.13 Detection success for object and suitability of the model to be applied 23
4.14 Model predicting unannotated objects 23
4.15 Summary 24

5 CONCLUSIONS AND RECOMMENDATIONS 25
5.1 Conclusions 25
5.2 Recommendations for future work 25

REFERENCES 26

APPENDICES 28

LIST OF TABLES

Table 2.1: Construction object image datasets 6

Table 2.2: SODA and other construction object image datasets 7

Table 4.1: Results from three objects detectors, YOLOv4, Detectron2, and EfficientDet 20

LIST OF FIGURES

Figure 2.1: mAP of YOLO (v3, v4) for SODA 8

Figure 3.1: Work plan observed in this project 10

Figure 3.2: Image and their corresponding labels 12

Figure 3.3: Cells in Google Colab 13

Figure 3.4: Examples of images that the model recognizes 15

LIST OF APPENDICES

APPENDIX A: YOLOv4.ipynb 28

APPENDIX B: Detectron2.ipynb 31

Appendix C: EfficientDet.ipynb 36

Appendix D: Ground truth images 41

Appendix E: Image dataset health check 46

Appendix F: YOLOv4 results 52

Appendix G: Detectron2 results 57

Appendix H: EfficientDet results 62

Appendix I: YOLOv4 Ground truth-Prediction Table 67

Appendix J: Detectron2 Ground truth-Prediction Table 70

Appendix K: Average detection time 73

CHAPTER 1

INTRODUCTION

1.1 General Introduction

A construction project, be it for landed homes, high rise residentials, commercial lots, infrastructures, and etcetera, is a highly structured venture that requires extreme coordination to achieve maximum productivity. A high efficiency translates to savings in cost, time, and manpower.

Therefore, the adoption of artificial intelligence (AI) is inevitable as we attempt more complex and ambitious projects. In recent years, a surge in the adoption of AI is observed in the construction industry. AI is a hypernym, and it refers to the simulation of computer to imitate the human brain, in order to perform cognitive function like learning, recognition, and problem solving.

One subset of AI is machine learning (ML), and it is the learning of data by statistical techniques without being explicitly programmed to do so. Computer vision falls under the ML. Object detection, as it names suggests, uses computer vision technique to detect specific objects with an acceptable accuracy. In general, there are three elements that are vital in object detection – the Application Programming Interface (API) or model, the algorithm or sometimes called architecture which is the detection method, and the image dataset. All the three elements are distinct, but they are interdependent. In the absence of one element, the whole object detection would not work. When an image library containing images or videos are fed into the detection model, the algorithm runs and starts to localize meaningful objects from an image. The more data i.e., more images the model is exposed to, the more trained it is in detecting the object.

AI implementation is plentiful in other industries such as transportation, healthcare, e-commerce, and etcetera. However, it is the opposite in the construction industry. This is because construction projects rely heavily on previous proven methods rather that new innovative methods that sometimes lack track record of industry success. Construction engineers would not risk security over the vague promise of newer technologies. As of such, there was very little studies and works done regarding AI in construction.

However, the potential of AI is very promising. Innovation like object detection with ML can be very beneficial in large construction sites to ease surveillance and safety management activities. Object detection synergises very well with current implemented technologies like BIM and the use of drones, therefore brining implementation cost to a minimum.

There is an abundance of open source code for object detection models on the Internet. Some tools and hardware are also provided online for free. In this project, the author aims provides a basis of comparison to gauge the performance of three object detector that are based on three distinct architectures.

1.2 Importance of the Study

As mentioned in the previous subtopic, studies and works done regarding AI implementation in construction sites are rare. It is prudent to understand and take part in the development of AI application in construction, therefore this study aims to provide support and encouragement to future civil engineering graduands to pursue programming and computer science skills. Much work is to be done in this field.

Computing power is an important component of object detection. Computing power comes from the capacity of graphics processing unit (GPU) of a computer. As training the model requires huge computing power, the capability of GPU will determine whether the model can be successfully trained or not. On top of that, having more computing power means more accurate and less training time for the model. However, not every individual or organization can commit to purchasing and investing in GPUs that are especially hard to come by in this COVID-19 pandemic time. Thankfully, plenty resources and hardware are available for free online. In this study, we want to know the limitations of our free hardware to detect construction site objects.

To the best knowledge of the author, there are no model publicly and readily available to detect construction site objects. This is because image datasets that contains construction site objects explicitly are not much. Thus, this study explores the necessary steps to be undergone in order to develop a fully working model from scratch, which can successfully detect objects at a construction site. The author also hopes that basing off this project, improvements can be made to enhance the model.

1.3 Problem Statement

The adoption of technology in the construction industry has always been slow. This is because the industry has been deeply rooted in a conservative corporate culture and are traditionally linear in its nature (Lee, 2018). On top of that, the complexity and elevated risk of the industry has solidified the companies' dependence on traditional, proven, and safe method of construction. The priority of the client to ensure the completion of project on time and within budget has also created a construction culture where consultants and contractors often neglect innovation. A radical and fresh change is therefore hard to implement. Despite the rigid nature of construction towards huge change, smaller changes have been embraced. Current practice of implementing robotics, AI, and Internet of Things (IoT) have reduced construction costs by up to 20% (Patil, 2019). Therefore, the problems that are associated with this study can be summarized as follows:

i. Lack of construction site specific object detector.
ii. The absence of comparison done on construction site object detection models, despite the large availability of open-source codes.
iii. Unwillingness of stakeholders to adopt new technologies in construction industry

1.4 Aims and Objectives

The aim of this study is to conduct a comparative study of three different object detector. The aim of the research is based on the objectives listed below:

i. To develop and program models to detect construction site objects.
ii. To quantify the ability of object detection in terms of speed, accuracy, and computing resources needed.

1.5 Scope and Limitation of the Study

Due to time and resources constraints, the scope of this study will only be limited to the quantitative comparison of the models. The deployment beyond the programming phase, e.g., application, testing, and study of the models in a real site were not done in this study. To adhere to COVID-19 movement restrictions and construction site privacy policies, all images were web mined online.

1.6 Contribution of the Study

The limitations of resources faced by this study can be used as a benchmark. Future studies should consider a step-up allocation to their resources, i.e., coding literacy, image dataset, hardware, computing power, and etcetera, if there are plans to produce an improved result. As there are less work done available associated to the coding and training of object detector for construction site objects, this study provides codes and scripts of the models which can be modified heavily. The code to the models are also general and extensive in nature, therefore it can be transformed to detect other objects, even those outside construction sites.

1.7 Outline of the Report

In this report, the remaining part are structured as follows: Chapter 1 introduces the topic, scope and limitations, contributions and importance of the study. Chapter 2 deals with the review of similar works in scientific literatures. Chapter 3 outlines the methodology and work plan to complete this project while Chapter 4 presents and discusses the obtained results. Chapter 5 closes and summarizes the whole report, with recommendations for future work.

CHAPTER 2

LITERATURE REVIEW

2.1 Introduction

In this chapter, publications on deep learning and computer vision in construction, object detection architectures, image data set, and comparisons of a few object detectors are reviewed.

2.2 Computer vision in Construction Industry

Computer vision have been extensively used and researched in the construction industry. It is a fairly new research topic. In a study published by Rad et al. (2020), Deep learning topic represents 37.04% of the publications in the Automation in Construction journal with an average published year of 2016. It also represents 14.81% of the publications in the Advanced Engineering Informatics journal with an average published year of 2013.

Kolar et al. (2018) has noted that computer vision techniques like histogram of oriented gradients, histogram of HSV (hue, saturation, value) colours input for k-nearest neighbors (KNN) classifier, histogram of optical flow, motion boundary histogram, many more were used to detect construction workers, site machinery and construction site's progress. This was way before computer vision-based object detectors.

With the advancement of both hardware and software, a new approach for object detectors which extracts large number of image features and uses that feature for image classification, were then adopted. Convolutional neural network (CNN) – often regarded as the most powerful technology to process and solve computer vision related problems, was one of the examples (Kolar et al., 2018). One benefit of CNN algorithm is that they have self-learning ability from a given dataset. Pairing CNN with visual sensors like surveillance camera, unmanned air vehicles, and etcetera can play a huge role in the management of construction site. Zhu et al. (2017) in his publication, has examined workforce, detected equipment, and tracked construction sites with video processing technology.

Xiao et al. (2019) has successfully built a model that detects truck and excavators using YOLOv3. The author noted that deep learning methods have huge

potential in construction automation, as it will help us to understand the situation at site. Effective visualization is also one important area as construction sites are often disorderly and misinformation will impact real-time decision making.

Deep learning methods can also be used to detect defects in building. In a study published by Perez et al. (2019), normal building and its defects such as mould, detoriation, and stain were detected using the VGG-16 network pre-trained on ImageNet. In this study, transfer learning is used.

2.3 Image data set

Deep learning algorithms are data hungry (Duan et al., 2022). According to Duan et al. (2022), construction is a unique process that brings challenge to the gathering and labelling of the images. And thus, well-annotated construction images are hardly seen in larger and more famous datasets. Existing construction site image datasets are just too small and incomplete, with fewer classes. On top of that, the way how construction is structured and carried out also makes it hard for the untrained eye to correctly identify and annotate images taken at sites. Therefore, there is a need to build a public image dataset in construction research. (Xiao et al., 2019).

There are two types of image datasets in computer vision, general and domain dataset. General datasets are used by the public, which contains natural and everyday life objects. Domain dataset on the other hand, contains categories of specific fields. Table 2.1 shows various the various types of general datasets. All these datasets are constantly being renewed and updated, allowing excellent object detection models to be coded in the future.

Table 2.1: Construction object image datasets

Dataset	Images count	Classes
Mnist	70,000	NIL
PASCAL VOC	11,000	20
Microsoft COCO	160,000	91
ImageNet	14 million	20,000

Aerial images also provide a good set of data for machine learning. Datasets like the TAS, SZTAKI-INRIA, NWPU VHR-10, VisDrone, SpaceNet MVOI, DIOR, iSAID, and DOTA are mostly aerial images. The total image ranges from 800 – 20,000 images (Ding et al., 2021). It can be useful to build object detectors for satellites and drone applications in overcoming problems related to congested and densely packed objects.

Despite the large image datasets available, most of it lacks proper categories within the context of construction. Therefore, in recent years, many construction objects image datasets were established. Tajudeen et al. (2014) has created an image dataset of five construction machineries – excavators, loader, bulldozers, and backhoe diggers. In total, there are a few thousand images of the machineries with different views. Kolar et al. (2018) has created a training dataset of 4,000 images for guardrails.

In a study to construct an object detector from development to deployment by Arabi et al. (2019), the author used the AIM image dataset, which are originally from ImageNet. Paired with image crawling, a total of 3,271 images of loader, excavator, dump truck, concrete mix truck, roller and grader were created. In another study by Xiao et al. (2019), 5,000 images were collected.

In 2022, Duan et al. has created the Site Object Detection dAtaset (SODA) which contains 19,846 images and 286,201 objects. By far, it is the largest construction object image dataset. Table 2.2 shows the comparison of SODA with other construction object image dataset.

Table 2.2: SODA and other construction object image datasets

Dataset	Images counts	Objects	Classes	Size
SODA	19,846	286,201	15	1920 × 1080
MOCS	41,668	222,861	13	1200 × __
ACID	10,000	15,767	10	608 × 608
CHV	1,330	9,209	-	608 × 608

2.4 Performance comparison for various object detector

A slight change in object detection algorithm can affect the results produced. In a study conducted by Duan et al. (2022), the author tested the SODA dataset with YOLOv3 and YOLOv4 algorithm. The mean average precision (mAP) for YOLOv3 and YOLOv4 are 71.22% and 81.46% respectively. This shows that the v4 version is more

accurate than the v3 version. Figure xx shows the average precision and the mAP of all the 15 classes in the SODA dataset.

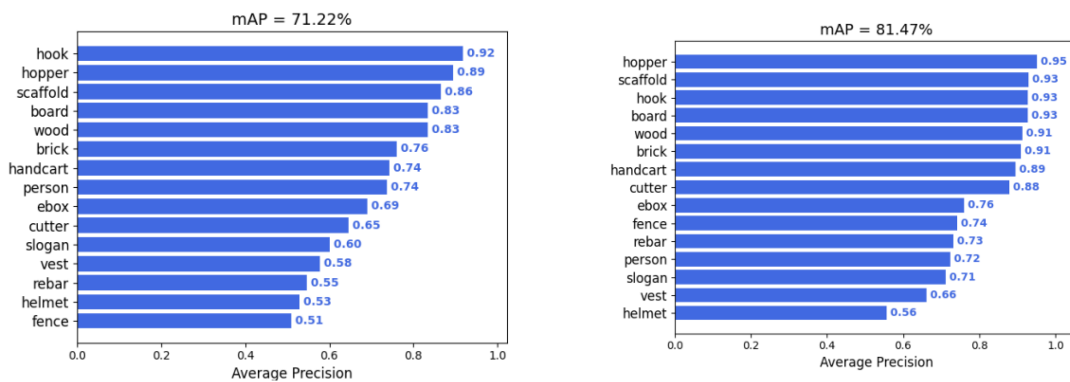


Figure 2.1: mAP of YOLO (v3, v4) for SODA

In a publication by Intellica.AI (2019), the author coded and compared between three models – YOLOv3, Faster R-CNN and SSD to detect a specific logo. Input images of size 840 × 840 were used. The training time for YOLOv3, Faster R-CNN and SSD were 10 hours, 18 hours, and 14 hours respectively. During the analysis, it was found that the accuracy is 100% for YOLOv3, 86.67% for Faster R-CNN, and none for SSD. This is because the loss value for SSD is too large compared to the other two models. Despite training the SSD model for three times the iteration of the other two, it still failed (Intellica.AI, 2019).

Nath et al. (2020) has concluded that between YOLOv2 and v3, the latter has a better accuracy in detecting a combined Pictor-2 dataset. Pictor-2 is a dataset that contains mainly construction site objects. In the study, YOLOv3 scores a mAP of 77.3%. The author concluded that the model's performance is able to match state-of-the art models, illustrating its consistency in detecting common construction objects.

2.5 Summary

In essence, while the application of object detection is extensive in construction industry, it is fairly new. The availability of large public image dataset was scarce, up until recently. Comparisons of object detectors is crucial to understand its strength and limitation.

CHAPTER 3

METHODOLOGY AND WORK PLAN

3.1 Introduction

This chapter explains the methodology and work plan taken to obtain desired results. Every step of the stage is detailed in sections below.

3.2 Overview of the Work Plan

Figure 3.1 shows the work plan for this project. The first step was to review all the necessary requirements for the work to start. It also means to review the coding platform and all the important parameters to be used as grounds for comparison. Then the next step was gathering images and pre-process them. Image dataset is perhaps the most important step of the study. The quality of the images gathered will affect the overall accuracy. Once the images were processed, coding works can begin. The first step in coding was to download and install model dependencies. This step ensures the correct environment are set up. Subsequently, images were imported into the notebook and all the necessary organization are done. The model was then trained. If the model fails, the upper limit of training parameters was tuned to ensure smooth running within the capability of the hardware. A model is considered to be successfully trained once it reaches the defined epoch or iterations from the previous steps. The weighs which is akin to the brain of the model were exported and saved for future use. Test images were then fed to the model to obtain desired results. Once all three object detectors had inferred and output the images, the test images were downloaded and saved locally (in the workstation). Analysis was done on the results without the use of code. If there are no results as a consequence of model incompetence, it needs to be retrained.

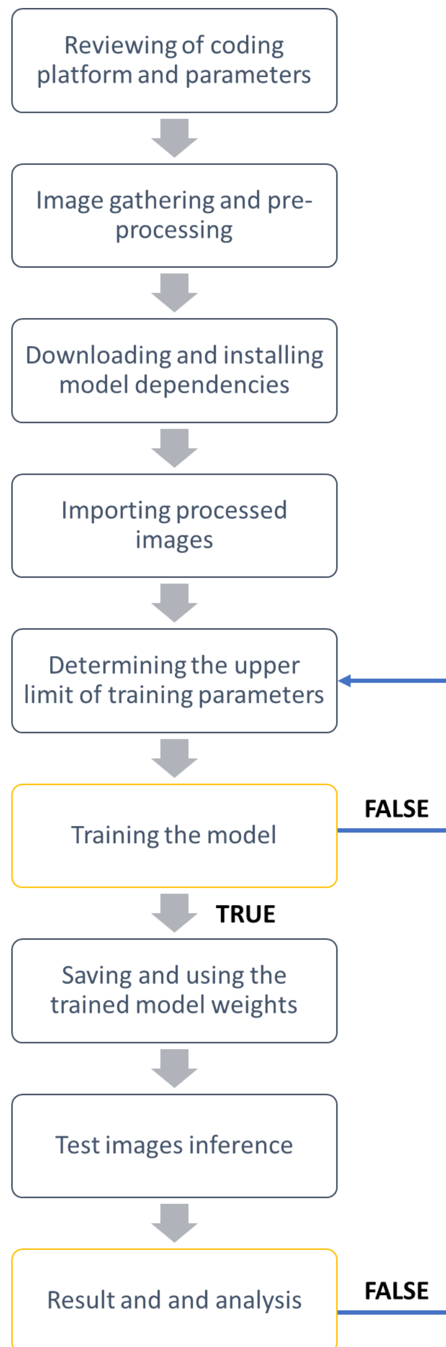


Figure 3.1: Work plan observed in this project

3.3 Reviewing of coding platform and parameters

Python will be used as it is a common programming language used in object detection. Google Colab platform was used because it provides free access to hardware online. As it is a free version, a Tesla P100 is provided. Initially, Jupyter Notebook was also considered but was not used in the end as the Tesla P100 is more powerful GPU.

Three object detection models were selected – YOLOv4, Detectron2, and EfficientDet. The coding of these models is relatively easy, hence the reason why it is picked for this project.

To sufficiently quantify the ability of the object detector, various aspect has to be considered. The main parameters in this project are the accuracy, speed, and overall resources consumption. 10 detailed grounds for comparison are listed below:

i. Training time
ii. Average detection time (s/img)
iii. The need for more data
iv. Accuracy of detected objects
v. Input image resolution
vi. Overlapping of detected objects
vii. Frequency of overlapping
viii. Detection success for small objects
ix. Detection success for large objects
x. Suitability of the model to be applied

3.4 Image gathering

Initially, the plan was to web mine images from the Internet by taking screenshots from Google Earth Pro, drone imageries from various sources, and other relevant images.

But in late February, we stumbled upon Site Object Detection dAtaset (SODA) and has decided to use that dataset due to its complete annotation and large database. The image gathering process took more than 48 hours of continuous downloading. In total, we gathered 19,846 images with a size of 24.5GB.

3.5 Image pre-processing

Image pre-processing involves the curation, compression, and labelling of the gathered images. As we do not have enough computing power to feed the model with 19,846 images, 100 images were chosen at random for this project. This 100 image dataset is named as Construction objects v2.

To hasten the pre-processing phase, an annotation website called Roboflow is used. Roboflow allows seamless pre-processing of the images including labelling, exposure and brightness settings, rotation, and many more processes. Construction

objects v2 dataset were uploaded into the website and the following steps were taken – annotate or labelling, train-test split, auto-orient and resize, and augmentations.

Image annotating or labelling is where designated boxes are placed around objects intended to be detected, which are then named accordingly. This process creates a separate .txt file which contains the labels and coordinates of the bounding boxes. Again, to save on resources, the number of classes to be detected is reduced. The initial SODA dataset contains 15 classes. In this project, it is reduced to 4 classes, namely person, vest, helmet, and slogan. The process is done by removing unwanted labels from already annotated images, on the Roboflow website. Figure 3.2 shows an example of the image and their labels for this project.

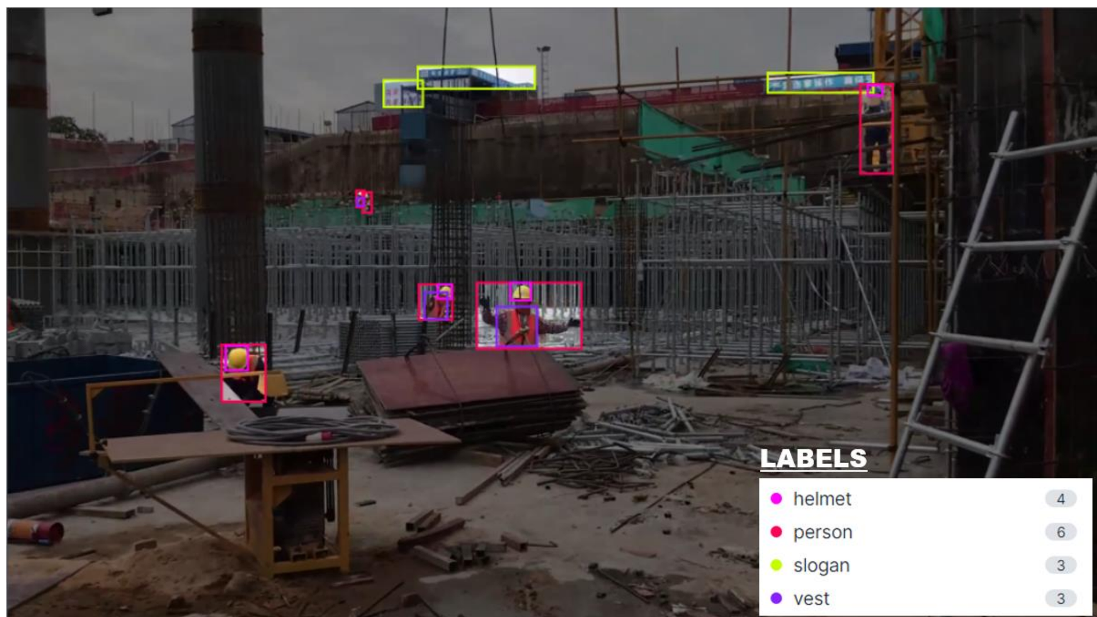


Figure 3.2: Image and their corresponding labels

The image size is standardized by compressing all 100 images to 416 × 416 sized pictures. This step also helps to cut down the training time for the model. It is then split into train and test data randomly with a 9:1 ratio. Train data are images that are fed into the to-be-trained model, while test data are images used to measure the accuracy of the model later on. Two augmentation process were then applied. The first process rotates the train images between -15° and +15° degrees and the second process brightens it by -15% and +15%. This process diversifies the train data from 70 images to 230 images of various orientation and brightness. The final dataset contains 240 images with a train-test ratio of 23:1.

3.6 Coding

The coding was done on Google Colab with Python. Appendix A, B, and C are codes to the YOLOv4, Detectron2, and EfficientDet model. All three models are coded from scratch, and no one is identical to the other. However, all three models have similar checkpoints or milestones of achievement which are vital to their next steps. Another benefit of using Google Colab is that the codes can be broken down into cells and executed individually. Progress is saved by the cells. Meaning, if there are changes to any part of the entire code, there is no need to run the code from the top in order to observe the changes. Figure 3.3 shows cells on Google Colab for YOLOv4 model. The first cell (in black) contains the first line of the code. It was executed, and its memory saved in the first cell. The output is in grey. The subsequent codes can be added onto the next cells. Therefore, if any changes were to be made into the second cell, the progress of the first cell will not be discarded.

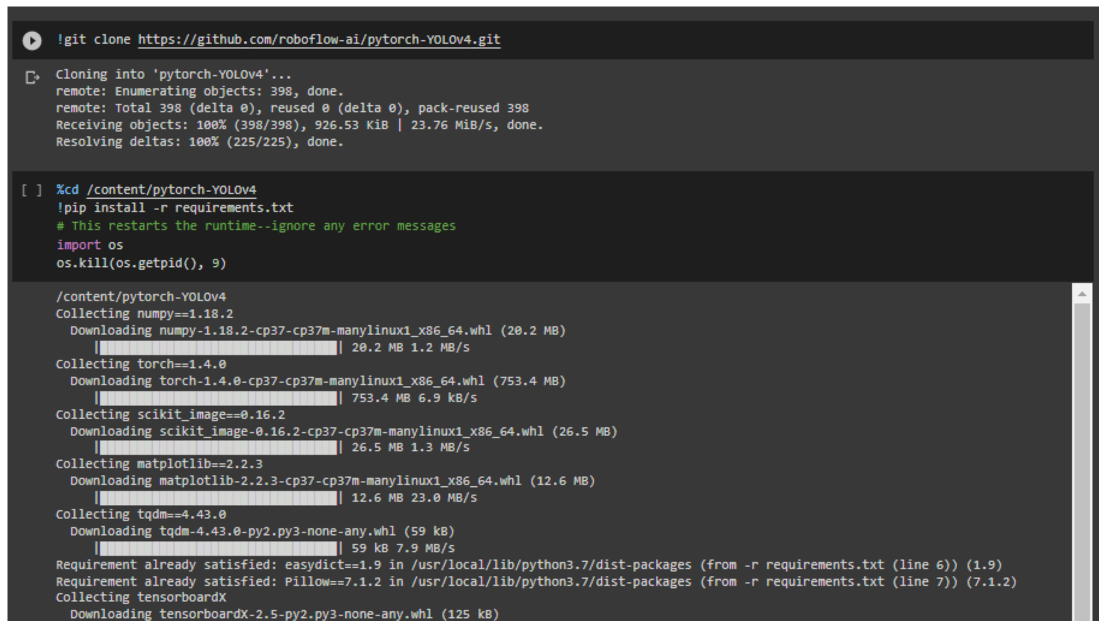


Figure 3.3: Cells in Google Colab

3.6.1 Downloading and install model dependencies

Commands like *!git clone, !pip install, import*, and *!gdown* of the Python language were used. Some model requires the cloning of an entire repository, hence the use of *!git clone* command. *!pip install* installs a certain file for the model to work. *import* brings in a certain requirements once a repository is cloned while *!gdown* downloads

files from a shared Google drive. The correct installation and setting up of environment is essential for the object detector to work.

3.6.2 Importing processed images

As our images have already been processed on Roboflow, we can then import it to our program. On the Roboflow website, the image can be exported in specific formats required by our object detectors, in the form of a link. The link can be pasted on our Colab notebook. A few simple commands unzips the link, and creates a directory on our notebook, which we can then proceed to the next step. In some object detectors like the YOLOv4, additional line of codes like the *!mkdir* and *%cp* command are needed to create and organize new directories for our images.

3.6.3 Determining the upper limit of training parameters

Different models have different parameters to be adjusted. For instance, you can change the batch size, batch subdivisions number, learning rate, and epoch on the YOLOv4 models, the max iterations, steps, and evaluation period for the Detectron2 model, and the learning rate and value interval for EfficientDet model. The values here can be tweaked and changed in order to obtain the best fit. If the model is not accurate enough, parameters like epoch and iterations can be stepped up. Conversely, if the model is overfitted, the value should be stepped down.

3.6.4 Training the model

Running the code kick starts the training process. In this stage, there is not much to do but to allow the model to read and recognize the images. The model recognizes the object by comparing the labelled image with the .txt file. As a human, imagine looking at a picture while holding the image description side by side. This is the exact process that the computer is going through. Figure 3.4 shows some of the images that the model recognizes during training. As observed, some of the pictures are not in upright position – they are rotated slightly due to the augmentations done during image pre-processing phase.

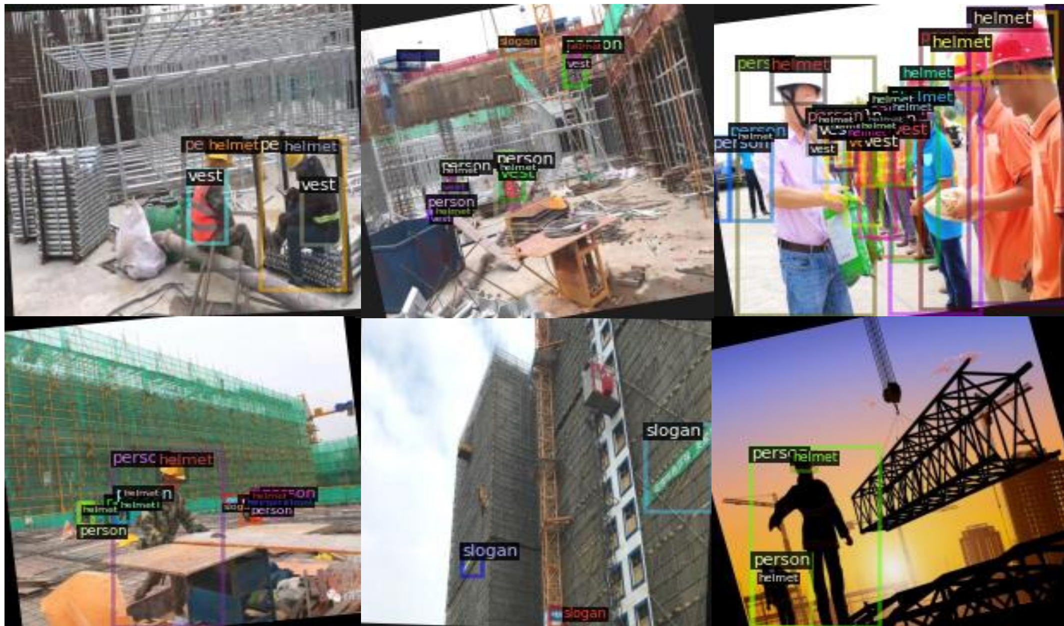


Figure 3.4: Examples of images that the model recognizes

The training time can take up to a few hours, therefore the workstation has to be turned on throughout the training. Maximum computing power are used here, therefore bottlenecks happened frequently. In the event of bottlenecks or model running out of memory, training parameters needs to be changed to a lower value.

3.6.5 Saving and using the trained model weights

During this stage, the model is said to have contained trained weights. Weights are important as it is akin to the brain of the model. If the weights were not saved, the whole training must be repeated again. Saving the trained weights can be done by mounting user's Google drive and exporting it. At the next round of inference, the weights can be loaded externally without needing to retrain the whole object detector.

3.6.6 Test image inference

The test images are then fed through the model for inference. Test images are images that the model have not seen before, it is a fresh set of images used to determine the accuracy of the object detector. The test images were labelled from 1 to 10. The model will take some time to infer the images before outputting images with bounding boxes and its prediction e.g., person, vest, helmet, or slogan. Appendix D shows all the annotated test images.

3.6.7 Result and analysis

The images were labelled 1 – 10, according to their original, clean, annotated version. This step is to ensure correct comparison with their ground truths, and the carrying out of meaningful analysis. The quality of the detection was then observed and analyzed. The 10 grounds for comparison, as mentioned in subsection 3.3 are used here. Most of the criteria is qualitative, expect for the average detection time and accuracy of test data which has to involve some calculations. All calculations were done manually on Microsoft Excel. Appendix F, G, and H are the results produced by YOLOv4, Detectron2, and EfficientDet.

3.6.7.1 Average detection time

Average detection time was calculated by taking the sum of time required to detect all images divided by the number of images. It is expressed in image per milliseconds. For Detectron2, as we could implement the Tensorboard function to the code, the average detection time was shown in the model itself, therefore eliminating the need to calculate separately. Appendix K shows the detection time and their average detection time taken.

3.6.7.2 Accuracy of test data

The accuracy of the test data prediction was defined as the ability of the model to correctly predict the test images. Since the test images were labelled, the annotations or bounding boxes are the ground truth. Appendix D shows the ground truths.

To calculate the accuracy, firstly all the number of ground truths in the images were noted down and compiled into a table. Total number of ground truths are calculated. Then, the downloaded test images from the models were analysed one by one, by comparing it to the ground truth version. A correct prediction i.e., correct prediction of Person yields one point. The percentage of the accumulation of correct predictions over total correct and wrong predictions (number of ground truth) gives the accuracy of the model. The equation is expressed as:

$$Accuracy = \frac{Correct\ predictions}{Correct\ predicitons + Wrong\ predictions} \times 100\%$$

Appendix I and J are the ground truth-predictions table for YOLOv4 and Detecrtron2.

3.7 Summary

An overview of the work plan is observed in this chapter. Detailed and specific steps were reported in each subsection. In this project, Python were used as the main programming language and the whole model is coded on Google Colab. A modified SODA image dataset was used, and 10 test images were used to acquire the desired results. The results for all three object detectors are presented and discussed in the next chapter.

CHAPTER 4

RESULTS AND DISCUSSIONS

4.1 Introduction

In this project, three working object detectors were used on 10 test images. In this chapter, 10 criteria that were used as grounds for comparison were presented and discussed. They are the training time, average detection time (ms/img), the need for more data, accuracy of detected objects, input image resolution, overlapping of detected objects, frequency of overlapping, detection success for small objects, detection success for large objects, and suitability of the model to be applied. The image dataset health check will also be discussed to allow a better understanding on the overview of the image dataset

4.2 Image dataset health check

The image dataset health check function that is readily available on the Roboflow website provides extra information that is useful for our decision making. In this subsection, we will analyse some of the statistics.

4.2.1 Images and annotations health

Figure E-1 of Appendix E shows the images and annotations health. Our uploaded dataset contains 100 images with 0 missing annotations and 0 null examples. A null example is when an image contains no annotations. Meaning, all the images are annotated and have their annotations file ready. In total, there are 1,239 annotations across 4 classes. On average, 1 image contains 12.4 annotations. The average picture size before it was resized is 2.07 megapixel. Smallest picture size is 0.11 megapixel, while the largest picture size is 12.73 megapixel.

4.2.2 Class balance check

Having a good class balance is important because we want our model to learn evenly. The class balance of the image dataset is generally healthy. From figure E-2 of Appendix E, we can see that there are 457 person, 388 helmet, 219 vests, and 175 slogan annotations. In total, they make up the 1,239 annotations. Person and helmet

are well represented, except for vest and slogan which are underrepresented. To overcome this, more images should be added until the class balance evens out.

4.2.3 Image size and aspect ratio check

Figure E-3 of Appendix E shows the aspect ratio distribution. The images in our dataset are more wide than square. Wide images constitute 92% of our image dataset, while square and tall images constitute 1% and 7% respectively. Moving on to the dimension insights, as shown in Figure E-4 of Appendix E, 63% of the images are jumbo size – they have a size of more than 1024 × 1024. 27% are large images, and 10% are medium images. Size information matters because it helps in resizing decisions. The median size acts as the maximum permissible limit for image size. Resizing is crucial because some models can only detect a certain image size.

Figure E-5 of Appendix E depicts the distribution of image sizes. The x-axis is the width, and the y-axis is the height. A directly proportional line passes through the origin. Any point that lies on this line is a square image. If the height is significantly larger than the width, it is a tall image. Conversely, if the width is significantly larger than the height, it is a wide image. Images that are too wide or too tall can affect our resizing process and subsequently our results. This is because the image will tend to overstretch along the shorter dimension, disfiguring our to-be-detected objects. In our dataset, there are no image that are too wide or too tall.

The median image size is 1920 × 1080, which is deemed as wide. Thus, 1920 × 1080 is the largest resizing limit. Any resized images with sizes below the median image size is acceptable. 416 × 416 size is chosen.

4.2.4 Annotations heatmap

Figure E-6 to Figure E-9 of Appendix E depicts the annotation heatmaps for all the classes. Heatmaps helps to visualizes where the objects in the images, without having to search through individual annotations. This is vital as we do not want the object to be cropped out during resizing and the heatmaps tells us exactly where our objects are according to the classes. As shown in Figure E-6 of Appendix E, Slogans are distributed throughout the images, as it can be in hanged mid-air or placed on the ground. Person heatmap, as shown in Figure E-7 of Appendix E were concentrated from the lower part up until around two third of the image. Helmet heatmap is generally centred with an emphasis at the top of the image, as shown in Figure E-8 of

Appendix E, as some workers are working while crouching or sitting. Vest on the other hand is centred but focused on the lower part, which is still logical. Some vests were detected at the upper left hand corner, due to some workers working on elevated heights. Figure E-9 of Appendix E shows the heatmap for vests.

4.3 Results

Table 4.1 shows the results obtained from three object detectors, YOLOv4, Detectron2, and Efficient Det.

Table 4.1: Results from three objects detectors, YOLOv4, Detectron2, and EfficientDet

	YOLOv4	**Detectron2**	**EfficientDet**
Training time	1h 30m 4s	2h 42m 28s	28m 38s
Average detection time (ms/img)	4143	742	305
Need more data?	NO	NO	YES
Accuracy of test data			
Person	75.93%	94.44%	-
Hat	78.00%	74.00%	-
Slogan	81.25%	0.00%	-
Vest	30.00%	60.00%	-
Input image resolution	416×416	416×416	416×416
Overlapping	YES	NONE	YES
Frequency of overlap	LESS	-	OFTEN
Detection success for small sized objects	YES	YES	YES
Detection success for large-sized objects	YES	YES	YES
Can this model be applied yet?	YES	YES	NO

4.4 EfficientDet

The epoch or training duration for EfficientDet were initially set at 100 epochs. However, the computing power of the Tesla P100D were insufficient to reach that amount. Therefore, the epochs were brought down. Trial and error epoch values of 80,

65, 50, 45, 35, 20, and 15 were used. It is found that the epoch limit bottlenecks at 17. Therefore, 15 epochs were set as the upper limiting factor. Despite the training success of the model, it produces results that are not quite favourable. Appendix H shows the results for EfficientDet. As observed from Figures H-1 to H-10 of Appendix H, there were multiple overlapped predictions which are not ideal for analysis. There were also plenty wrong predictions.

Despite being inaccurate, the model can still work, provided that a better hardware was used. This allows for higher epochs and thus a more accurate object detector. Another workaround is to provide more datasets for the object detector to learn. This step can be done even at lower epochs.

4.5 Training time and average detection time

The training times for YOLOv4, Detectron2, and EfficientDet were 1 hour 30 minutes 4 seconds, 2 hours 42 minutes 28 seconds, and 28 minutes and 38 seconds, respectively. Detectron2 took the longest time, whereas EfficientDet took the quickest time. The reason EfficientDet took the least time is because it was trained at a lower epoch i.e., 15 epochs.

In most object detectors, longer training time translated to faster average detection time. Detectron2 who took the longest time to train, detected objects with an average time of 742 milliseconds per image. YOLOv4 trailed with the ability to detect at 4143 milliseconds per image. Despite having the least training time, EfficientDet took only 305 milliseconds to detect images. This is faster than Detectron2 and YOLOv4. One possible explanation is that EfficientDet does not predict images with the same accuracy as the other object detectors. It is fast because it does not need to be accurate.

4.6 The need for more data?

Fundamentally, more data means a more accurate object detector. For this project, EfficientDet requires more images not because for fine tuning, but to raise its prediction accuracy to a more acceptable level. While an increase in dataset could definitely benefit the other two object detectors, it also means more work to annotate and label the images.

4.7 Accuracy for person

Accuracy is perhaps the most important criteria for object detectors. It is also the easiest benchmark to comprehend. A person is perhaps the most commonly detected objects. In this project, the accuracy for person is 75.93% for YOLOv4 and 94.44% for Detectron2. Detectron2 has a higher accuracy for person due to its detection algorithm and typical use case. Therefore, if the goal for the object detector at sites is to detect person, Detectron2 is more recommended.

4.8 Accuracy for hat

Hat or safety helmet is an important PPE to have at construction sites. The detection for hat can ensure all safety measures are being observed at sites. The accuracy for hat detection in this project is 78% and 74% for YOLOv4 and Detectron2. In the case for hat, YOLOv4 leads as the best object detector. The colour and condition of the hats does not affect the accuracy for object detection, therefore a possible reason for a low accuracy on hat detection is the occlusion and orientation of the object.

4.9 Accuracy for slogan

Slogan refers to the safety warnings at over sites. Safety warnings are essential to prevent mishaps. Essentially, detecting slogans has more to do with detecting words and sentences rather than objects. Technically, words placed in a sentence big enough at sites can constitute as slogan according to the object detectors. In this experiment, the accuracy for slogan is 81.25% for YOLOv4. Detectron2 has a 0% accuracy, meaning it does not predict correctly any of the ground truth. This can be attributed to the fact that slogan is underrepresented as indicated in Figure E-2. More images that contain slogan should be added to existing dataset.

4.10 Accuracy for vest

Vest could be the hardest to detect as it may be mistaken as clothing on a worker. The colour of the vest sometimes became monochromic with the body of the worker which furthers decrease the accuracy of the object detector. Our model detectors detected 30% accuracy for YOLOv4, and 60% for Detectron2. Again, the low accuracy may be due to the underrepresenting of vest in the image dataset.

However, it should be noted that vest could always be underrepresented with respect to the other classes because at sites, safety vest is not necessary unless one is

working at high altitude or foggy workspace. Workers can always wear safety helmets but not necessarily safety vests. Therefore, to truly overcome the underrepresenting of vest, images of only worker with vests but without helmet are needed. This is not possible as it is commonly known that safety helmets are a must at construction sites. The only workaround is to add more images until the model can detect vests at higher accuracy.

4.11 Input image resolution

The input image resolution was standardized at 416 × 416 size. Initially, this is a prerequisite for YOLOv4 to work. However, it was made consistent across all three models so that it can produce a meaningful analysis. An image dataset with varying image sizes can consume more resources in terms of computing power. The same goes to image resolution of larger size for the input.

4.12 Overlapping of detected objects

Overlapping is a common but unfavourable occurrence where a model predicts the same object more than once. Figure F-10 of Appendix F and Figure H-10 of Appendix H are some examples of overlapping. The occlusion, orientation, and brightness of an object in relation to the background can also cause overlapping.

Both the YOLOv4 and EfficientDet model has overlapped predictions. Detectron2, however, does not. The overlapping frequency for the YOLOv4 is less compared to the overlapping frequency of EfficientDet.

4.13 Detection success for object and suitability of the model to be applied

All three models successfully detect large and small objects. Therefore, during pre-processing phase, the bounding boxes can be of any sizes for these models. The YOLOv4 and Detectron2 model are suitable to be deployed for real life application, provided that more enhancements are made. On the other hand, EfficientDet are not suitable as the accuracy are still unknown.

4.14 Model predicting unannotated objects

To our surprise, the object detector has learned enough on certain classes to make predictions that are outside the labelled ground truth. For instance, by comparing Figure D-4 and G-4, some predictions made were not actually part of the annotations.

Some predictions are correct, but not all. The most common unannotated prediction that is correct are the person class. This is perhaps due to the fact that the human shape is easily distinguishable whereas objects like slogan are not.

4.15 Summary

We concluded that the three object detectors work. EfficientDet could not produce significant results as it needs more computing power and images. In terms of training time, Detectron2 took the longest, followed by YOLOv4, and EfficientDet. Longer training time means faster detection time. This is evident in Detectron2 that took an average of 742 milliseconds to detect. YOLOv4 took an average of 4143 milliseconds. EfficientDet took only 305 milliseconds, far surpassing the other model. However, it is fast because it does not need to be accurate. In terms of accuracy, Detectron2 is more accurate than YOLOv4 in detecting person and vest. YOLOv4 is more accurate than Detectron2 in detecting hat and slogan. Thus, an object detector must be chosen according to the objective of the detection.

CHAPTER 5

CONCLUSIONS AND RECOMMENDATIONS

5.1 Conclusions

In conclusion, three object detectors – YOLOv4, Detectron2, and EfficientDet were successfully coded. The training time was 1 hour 30 minutes and 4 milliseconds for YOLOv4, 2 hour 42 minutes and 28 milliseconds for Detectron2, and 28 minutes and 38 milliseconds for EfficientDet. Despite working, EfficientDet still requires more data in order to achieve acceptable accuracy. The EfficientDet model in this project were not able to produce meaningful results. More images are required. The accuracy for person, helmet, slogan, and vest, are 75.93%, 78.00%, 30.00%, and 81.25% for YOLOv4, and 94.44%, 74.00%, 60.00%, and 0% for Detecron2. The fastest average detection time is EfficientDet with 0.279 milliseconds, followed by Detecron2 with 742 milliseconds, and YOLOv4 at 3.74seconds.

5.2 Recommendations for future work

Computing power and robust image dataset is key to achieve a decent object detector. Therefore, a better hardware is recommended. A better allows longer, faster and more accurate training phase. The image dataset used in this project can also be expanded with two methods. The first method is to utilize the entire SODA dataset with the augmentations described in Chapter 2. The second method is to collect and compile more images, on top of the existing image dataset.

More objects should also be detected. In this project, only person, helmet, slogan, and vest are detected. In the future, objects like scaffold, electric box, bricks, machineries, and etcetera should be detected to produce a more comprehensive and complete object detector. Post development, the model should move into deployment phase e.g., real time detection at sites to test its reliability. Compatibility and cohesiveness with visual input device like mobile phones, drones, and video camera may be tested in the future.

It is also recommended that future Civil Engineering students to have better comprehension with Python coding ability. This will allow for more complex and steadfast model to tackle more challenging and demanding object detection problems.

REFERENCES

Ansari Rad, S., & Arashpour, M., 2020. *A Critical Review of Machine Vision Applications in Construction. Proceedings of the 37th International Symposium on Automation and Robotics in Construction (ISARC).*

Arabi, S., Arya, H., & Anuj, S., 2019. A deep learning-based solution for construction equipment detection: from development to deployment [online] Available at: https://arxiv.org/abs/1904.09021 [Accessed 12 July 2021]

Benedek, C., Descombes, X., & Zerubia, J., 2012. Building Development Monitoring in Multitemporal Remotely Sensed Image Pairs with Stochastic Birth-Death Dynamics. [online] Available at: https://hal.inria.fr/hal-00730552 [Accessed 12 July 2021]

Brilakis, I., Park, M. W., & Jog, G., 2011. Automated vision tracking of project related entities. *Advanced Engineering Informatics*, 25(4), 713–724.

IntellicaAI., Comparative Study of Custom Object Detection Algorithms., 2021. [online] Available at: https://intellica-ai.medium.com/a-comparative-study-of-custom-object-detection-algorithms-9e7ddf6e765e [Accessed 19 July 2021]

Chen, S., Zhan, R., & Zhang, J., 2018. Geospatial Object Detection in Remote Sensing Imagery Based on Multiscale Single-Shot Detector with Activated Semantics. Remote Sensing, 10(6), 820. MDPI AG. [online] Available at: http://dx.doi.org/10.3390/rs10060820 [Accessed 21 July 2021]

DOTA. (n.d.). A Large-Scale Benchmark and Challenges for Object Detection in Aerial Images. [online] Available at: https://captain-whu.github.io/DOTA/dataset.html [Accessed 5 September 2021]

Duan, R., Deng, H., Tian, M., Deng, Y., & Lin J., 2022. SODA: Site Object Detection dAtaset for Deep Learning in Construction. [online] Available at: https://arxiv.org/abs/2202.09554 [Accessed 6 April 2022]

Heitz, G., & Koller, D., 2008. Learning Spatial Context: Using Stuff to Find Things. Lecture Notes in Computer Science, 30–43. [online] Available at: https://doi.org/10.1007/978-3-540-88682-2_4 [Accessed 7 April 2022]

Kolar, Z., Chen, H., & Luo, X., 2018. Transfer learning and deep convolutional neural networks for safety guardrail detection in 2D images. *Automation in Construction*, 89, 58–70.

Lee, P., 2018. The Future of Construction - Embracing Technology and Innovation. Lexology. [online] Available at: https://www.lexology.com/library/detail.aspx?g=146ad57e-cc21-45c3-8c97-0696bab98d77 [Accessed 2 January 2022]

Li, K., Wan. G., Cheng, G., Meng, L., Han, J., 2019. Object Detection in Optical Remote Sensing Images: A Survey and A New Benchmark [online] Available at: https://arxiv.org/abs/1909.00133 [Accessed 5 February 2022]

Nath, N. D., & Behzadan, A. H., 2020. Deep Convolutional Networks for Construction Object Detection Under Different Visual Conditions. *Frontiers in Built Environment*, 6.

Patil, G., 2019. *Applications of Artificial Intelligence in Construction Management. 6th National Conference On Technology & Innovation: Disrupting Businesses, Transforming Market.*

Perez, H., Tah, J. H. M., & Mosavi, A., 2019. Deep Learning for Detecting Building Defects Using Convolutional Neural Networks. Sensors, 19(16), 3556.

Tajeen, H., & Zhu, Z., 2014. Image dataset development for measuring construction equipment recognition performance. *Automation in Construction*, 48, 1–10.

Zamir, S. W. (2019, May 30). iSAID: A Large-scale Dataset for Instance Segmentation in Aerial Images. [online] Available at: https://arxiv.org/abs/1905.12886 [Accessed 14 March 2022]

Zhu, P., 2018. Vision Meets Drones: A Challenge. [online] Available at: https://arxiv.org/abs/1804.07437 [Accessed 13 March 2022]

Zhu, Z., Ren, X., & Chen, Z., 2017. Integrated detection and tracking of workforce and equipment from construction jobsite videos. *Automation in Construction*, 81, 161–171.

APPENDICES

APPENDIX A: YOLOv4.ipynb

```
# Install YOLOv4 Dependencies
!git clone https://github.com/roboflow-ai/pytorch-YOLOv4.git
-----
%cd /content/pytorch-YOLOv4
!pip install -r requirements.txt
# This restarts the runtime--ignore any error messages
import os
os.kill(os.getpid(), 9)
-----
# downloading the yolov4 weights that have already been converted
 to PyTorch
!pip install --upgrade --no-cache-dir gdown
!gdown https://drive.google.com/uc?id=1li3dsDc9EuLwbN-
9rZ408gWs6cD1-Yhf
-----
#Import and Register Custom YOLOv4 Data
%cd /content/
# This cell exports the dataset to Colab via the link provided
!curl -
L "https://app.roboflow.com/ds/uCXMgffbyX?key=qhQQA1miJR" > robof
low.zip; unzip roboflow.zip; rm roboflow.zip
-----
%cp train/_annotations.txt train/train.txt
%cp train/_annotations.txt train.txt
%mkdir data
%cp valid/_annotations.txt data/val.txt
%cp valid/*.jpg train/
-----
def file_len(fname):
  with open(fname) as f:
    for i, l in enumerate(f):
      pass
  return i + 1
```

```
-----
num_classes = file_len('train/_classes.txt')

# Printing the number of classes to be detected
print(num_classes)
-----
#Train Custom YOLOv4 Detector
# Start training
#-b batch size (keeping this low (2-
4) for training to work properly)
#-
s number of subdivisions in the batch, this was more relevant for
 the darknet framework
#-l learning rate
#-g direct training to the GPU device
#pretrained invoke the pretrained weights that we downloaded abov
e
#classes - number of classes
#dir - where the training data is
#epoch - how long to train for
!python ./pytorch-YOLOv4/train.py -b 2 -s 1 -l 0.001 -g 0 -
pretrained ./yolov4.conv.137.pth -classes {num_classes} -
dir ./train -epochs 50
-----
# Looking at the weights that our model has saved during training
!ls checkpoints
-----
#Inference with YOLOv4 Saved Weights
# Choose random test image
import os
test_images = [f for f in os.listdir('test') if f.endswith('.jpg'
)]
import random
img_path = "test/" + random.choice(test_images);
-----
%%time
# The epoch can be changed here for inference
```

```
!python /content/pytorch-
YOLOv4/models.py {num_classes} checkpoints/Yolov4_epoch50.pth {im
g_path} test/_classes.txt
```

```
#visualize inference
from IPython.display import Image
Image('predictions.jpg')
```

APPENDIX B: Detectron2.ipynb

```
#Install Detectron2 Dependencies
# install dependencies: (use cu101 because colab has CUDA 10.1)
!pip install -U torch==1.5 torchvision==0.6 -
f https://download.pytorch.org/whl/cu101/torch_stable.html
!pip install cython pyyaml==5.1
!pip install -
U 'git+https://github.com/cocodataset/cocoapi.git#subdirectory=Py
thonAPI'
import torch, torchvision
print(torch.__version__, torch.cuda.is_available())
!gcc --version
# opencv is pre-installed on colab
-----
# install detectron2:
!pip install detectron2==0.1.3 -f
https://dl.fbaipublicfiles.com/detectron2/wheels/cu101/torch1.5/i
ndex.html
-----
# You may need to restart your runtime prior to this, to let your
 installation take effect
# Some basic setup:
# Setup detectron2 logger
import detectron2
from detectron2.utils.logger import setup_logger
setup_logger()

# import some common libraries
import numpy as np
import cv2
import random
from google.colab.patches import cv2_imshow

# import some common detectron2 utilities
from detectron2 import model_zoo
from detectron2.engine import DefaultPredictor
from detectron2.config import get_cfg
```

```
from detectron2.utils.visualizer import Visualizer
from detectron2.data import MetadataCatalog
from detectron2.data.catalog import DatasetCatalog
```

```
#Import and Register Custom Detectron2 Data
!curl -L "https://app.roboflow.com/ds/76cWf3eZy2?key=5f10P94N1N" > roboflow.zip; unzip roboflow.zip; rm roboflow.zip
```

```
from detectron2.data.datasets import register_coco_instances
register_coco_instances("my_dataset_train", {}, "/content/train/_annotations.coco.json", "/content/train")
register_coco_instances("my_dataset_val", {}, "/content/valid/_annotations.coco.json", "/content/valid")
register_coco_instances("my_dataset_test", {}, "/content/test/_annotations.coco.json", "/content/test")
```

```
#visualize training data
my_dataset_train_metadata = MetadataCatalog.get("my_dataset_train")
dataset_dicts = DatasetCatalog.get("my_dataset_train")

import random
from detectron2.utils.visualizer import Visualizer

for d in random.sample(dataset_dicts, 3):
    img = cv2.imread(d["file_name"])
    visualizer = Visualizer(img[:, :, ::-1], metadata=my_dataset_train_metadata, scale=0.5)
    vis = visualizer.draw_dataset_dict(d)
    cv2_imshow(vis.get_image()[:, :, ::-1])
```

```
#Train Custom Detectron2 Detector
#Importing our own Trainer Module here to use the COCO validation evaluation during training. Otherwise no validation eval occurs.
```

```
from detectron2.engine import DefaultTrainer
from detectron2.evaluation import COCOEvaluator

class CocoTrainer(DefaultTrainer):

  @classmethod
  def build_evaluator(cls, cfg, dataset_name, output_folder=None)
:

    if output_folder is None:
        os.makedirs("coco_eval", exist_ok=True)
        output_folder = "coco_eval"

    return COCOEvaluator(dataset_name, cfg, False, output_folder)
```

```
#from .detectron2.tools.train_net import Trainer
#from detectron2.engine import DefaultTrainer
# select from modelzoo here: https://github.com/facebookresearch/
detectron2/blob/master/MODEL_ZOO.md#coco-object-detection-
baselines

from detectron2.config import get_cfg
#from detectron2.evaluation.coco_evaluation import COCOEvaluator
import os

cfg = get_cfg()
cfg.merge_from_file(model_zoo.get_config_file("COCO-
Detection/faster_rcnn_X_101_32x8d_FPN_3x.yaml"))
cfg.DATASETS.TRAIN = ("my_dataset_train",)
cfg.DATASETS.TEST = ("my_dataset_val",)

cfg.DATALOADER.NUM_WORKERS = 4
cfg.MODEL.WEIGHTS = model_zoo.get_checkpoint_url("COCO-
Detection/faster_rcnn_X_101_32x8d_FPN_3x.yaml")  # Let training i
nitialize from model zoo
cfg.SOLVER.IMS_PER_BATCH = 4
cfg.SOLVER.BASE_LR = 0.001
```

```
cfg.SOLVER.WARMUP_ITERS = 1000
cfg.SOLVER.MAX_ITER = 1500 #adjust up if val mAP is still rising,
 adjust down if overfit
cfg.SOLVER.STEPS = (1000, 1500)
cfg.SOLVER.GAMMA = 0.05
cfg.MODEL.ROI_HEADS.BATCH_SIZE_PER_IMAGE = 64
cfg.MODEL.ROI_HEADS.NUM_CLASSES = 4 # number of classes + 1

cfg.TEST.EVAL_PERIOD = 500


os.makedirs(cfg.OUTPUT_DIR, exist_ok=True)
trainer = CocoTrainer(cfg)
trainer.resume_or_load(resume=False)
trainer.train()
```

```
# Look at training curves in tensorboard:
%reload_ext tensorboard
%tensorboard --logdir output
```

```
#test evaluation
from detectron2.data import DatasetCatalog, MetadataCatalog, buil
d_detection_test_loader
from detectron2.evaluation import COCOEvaluator, inference_on_dat
aset

cfg.MODEL.WEIGHTS = os.path.join(cfg.OUTPUT_DIR, "model_final.pth
")
cfg.MODEL.ROI_HEADS.SCORE_THRESH_TEST = 0.85
predictor = DefaultPredictor(cfg)
evaluator = COCOEvaluator("my_dataset_test", cfg, False, output_d
ir="./output/")
val_loader = build_detection_test_loader(cfg, "my_dataset_test")
inference_on_dataset(trainer.model, val_loader, evaluator)
```

#Inference with Detectron2 Saved Weights

```
%ls ./output/
```

```
cfg.MODEL.WEIGHTS = os.path.join(cfg.OUTPUT_DIR, "model_final.pth
")
cfg.DATASETS.TEST = ("my_dataset_test", )
cfg.MODEL.ROI_HEADS.SCORE_THRESH_TEST = 0.7   # set the testing t
hreshold for this model
predictor = DefaultPredictor(cfg)
test_metadata = MetadataCatalog.get("my_dataset_test")
                              -----
from detectron2.utils.visualizer import ColorMode
import glob

for imageName in glob.glob('/content/test/*jpg'):
  im = cv2.imread(imageName)
  outputs = predictor(im)
  v = Visualizer(im[:, :, ::-1],
                metadata=test_metadata,
                scale=0.8
                 )
  out = v.draw_instance_predictions(outputs["instances"].to("cpu"
))
  cv2_imshow(out.get_image()[:, :, ::-1])
```

Appendix C: EfficientDet.ipynb

```
#Set up EfficientDet Dependencies
! git clone https://github.com/roboflow-
ai/Monk_Object_Detection.git
                              -----
#Set up library requirements
! cd Monk_Object_Detection/3_mxrcnn/installation && cat requireme
nts_colab.txt | xargs -n 1 -L 1 pip install
                              -----
#fixed version of tqdm output for Colab
!pip install --
force https://github.com/chengs/tqdm/archive/colab.zip
#IGNORE restart runtime warning
!pip install efficientnet_pytorch
!pip install tensorboardX
                              -----
#Import Custom EfficientDet Data
#Outputing dataset in Coco Json format
!curl -
L "https://app.roboflow.com/ds/76cWf3eZy2?key=5f10P94N1N" > robof
low.zip; unzip roboflow.zip; rm roboflow.zip
                              -----
%ls
                              -----
#jpg images and some coco json annotations
%ls train
                              -----
!mkdir construction_objects
!mkdir construction_objects/annotations
!mkdir construction_objects/Annotations
!mkdir construction_objects/Images
                              -----
%cp train/_annotations.coco.json construction_objects/annotations
/instances_Images.json
%cp train/*.jpg construction_objects/Images/
                              -----
```

```
#Train Custom EfficientDet Detector
import os
import sys
sys.path.append("Monk_Object_Detection/4_efficientdet/lib/");
                                  -----
from train_detector import Detector
                                  -----
gtf = Detector();
                                  -----
#directs the model towards file structure
root_dir = "./";
coco_dir = "construction_objects";
img_dir = "./";
set_dir = "Images";
                                  -----
gtf.Train_Dataset(root_dir, coco_dir, img_dir, set_dir, batch_siz
e=8, image_size=512, use_gpu=True)
                                  -----
gtf.Model();
                                  -----
gtf.Set_Hyperparams(lr=0.0001, val_interval=1, es_min_delta=0.0,
es_patience=0)
                                  -----
%%time
gtf.Train(num_epochs=15, model_output_dir="trained/");
                                  -----
#Inference with Detectron2 Saved Weights
import os
import sys
sys.path.append("Monk_Object_Detection/4_efficientdet/lib/");
                                  -----
from infer_detector import Infer
                                  -----
gtf = Infer();
                                  -----
#trained model weights are in here in onxx format
gtf.Model(model_dir="trained/")
                                  -----
```

```
#extract class list from our annotations
import json
with open('train/_annotations.coco.json') as json_file:
    data = json.load(json_file)
class_list = []
for category in data['categories']:
  class_list.append(category['name'])
```

```
class_list
```

```
%%time
test_images = [f for f in os.listdir('test') if f.endswith('.jpg'
)]
import random
img_path = "test/" + random.choice(test_images);
duration, scores, labels, boxes = gtf.Predict(img_path, class_lis
t, vis_threshold=0.2);
```

```
from IPython.display import Image
Image(filename='output.jpg')
```

Export Trained Weights

```
#export trained model and mount Google Drive
from google.colab import drive
drive.mount('/content/drive')
```

```
%mkdir trained_export
%cp ./trained/signatrix_efficientdet_coco.onnx ./trained_export/s
ignatrix_efficientdet_coco_$(date +%F-%H:%M).onnx
%cp ./trained/signatrix_efficientdet_coco.pth ./trained_export/si
gnatrix_efficientdet_coco_$(date +%F-%H:%M).pth
%mv ./trained_export/* /content/drive/My\ Drive/
```

Reloading Trained Weights after Export

```
#export trained model
from google.colab import drive
drive.mount('/content/drive')
```

```
#our fork of the Tessellate-Imaging image detection library
#!rm -rf Monk_Object_Detection
! git clone https://github.com/roboflow-
ai/Monk_Object_Detection.git
                              -----
#Set up library requirments
! cd Monk_Object_Detection/3_mxrcnn/installation && cat requireme
nts_colab.txt | xargs -n 1 -L 1 pip install

#fixed version of tqdm output for Colab
!pip install --
force https://github.com/chengs/tqdm/archive/colab.zip
!pip install efficientnet_pytorch
!pip install tensorboardX
#IGNORE restart runtime warning
                              -----
#recover trained weights
!mkdir '/trained'
!cp '/content/drive/MyDrive/signatrix_efficientdet_coco_2022-03-
31-08:14.onnx' '/trained/signatrix_efficientdet_coco.onnx'
!cp '/content/drive/MyDrive/signatrix_efficientdet_coco_2022-03-
31-08:14.pth' '/trained/signatrix_efficientdet_coco.pth'
                              -----
import os
import sys
sys.path.append("Monk_Object_Detection/4_efficientdet/lib/");
                              -----
from infer_detector import Infer
gtf = Infer();
                              -----
#trained model weights are in here in onxx format
gtf.Model(model_dir="/trained")
                              -----
#download some test data
!curl -
L "https://app.roboflow.com/ds/76cWf3eZy2?key=5f10P94N1N" | jar -
x
                              -----
from google.colab import drive
```

```
drive.mount('/content/drive')
```

```
!ls test
```

```
#extract class list from our annotations
import json
with open('train/_annotations.coco.json') as json_file:
    data = json.load(json_file)
class_list = []
for category in data['categories']:
  class_list.append(category['name'])
```

```
class_list
```

```
%%time
img_path = "/content/test/z1303_jpg.rf.1587c25920f08f4ff394d8b5ca
ce1b2f.jpg";
duration, scores, labels, boxes = gtf.Predict(img_path, class_lis
t, vis_threshold=0.2);
```

```
!ls test
```

```
from IPython.display import Image
Image(filename='output.jpg')
```

Appendix D: Ground truth images

Figure D-1: Image 1

Figure D-2: Image 2

Figure D-3: Image 3

Figure D-4: Image 4

Figure D-5: Image 5

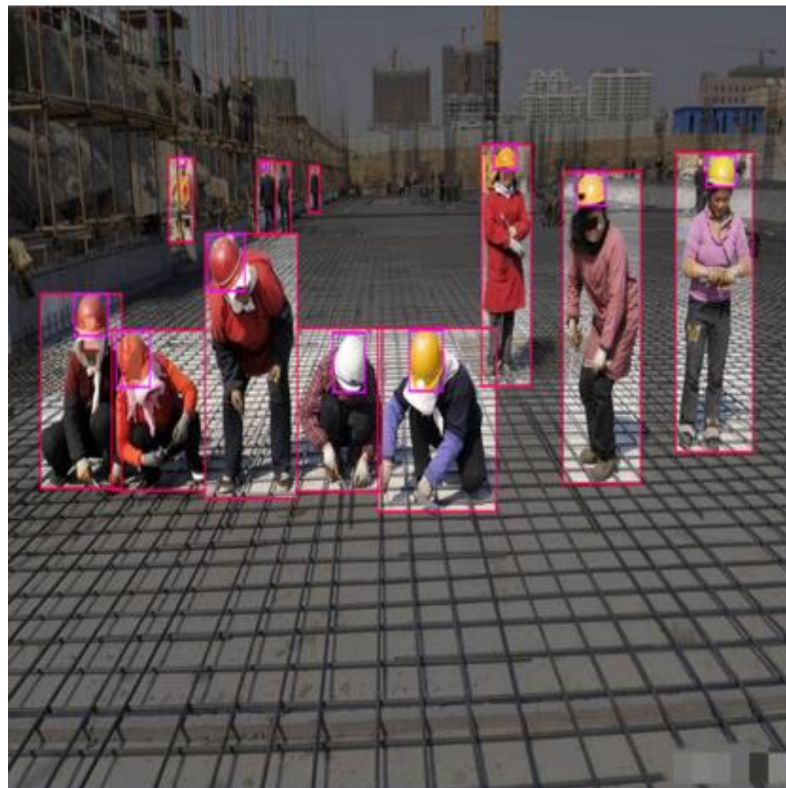


Figure D-6: Image 6

Figure D-7: Image 7

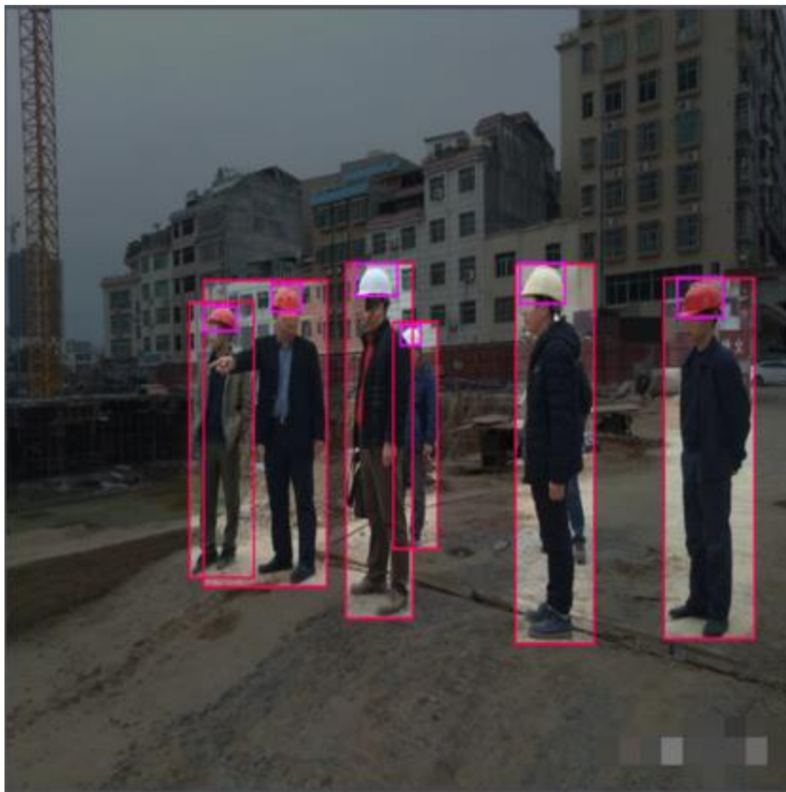


Figure D-8: Image 8

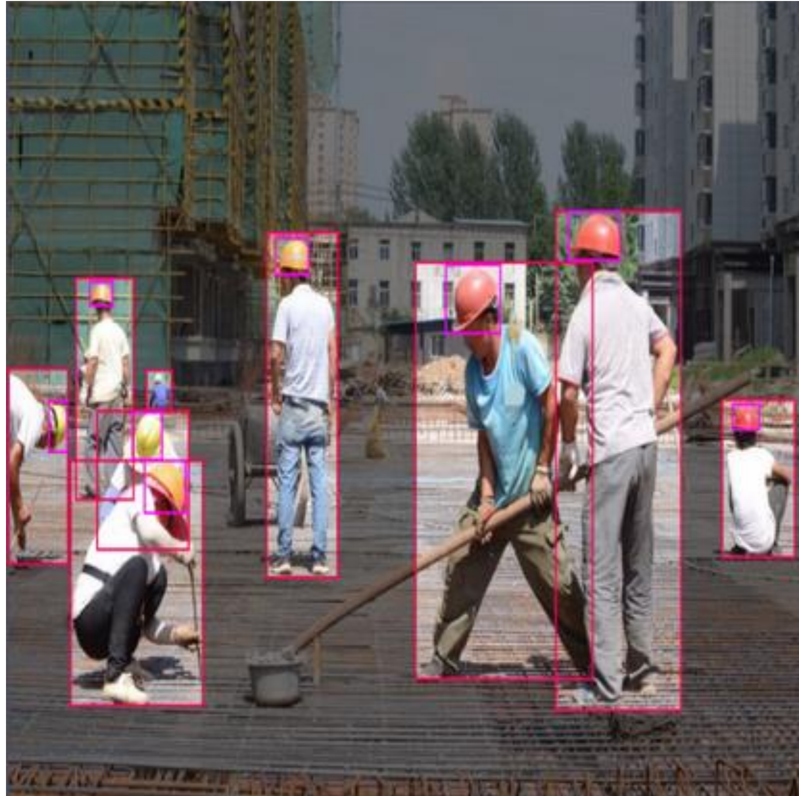


Figure D-9: Image 9

Figure D-10: Image 10

Appendix E: Image dataset health check

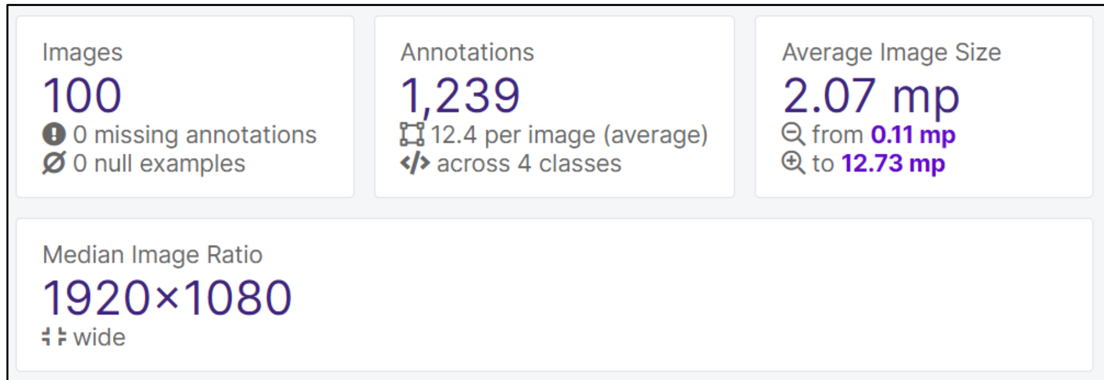


Figure E-1: Images and annotations health

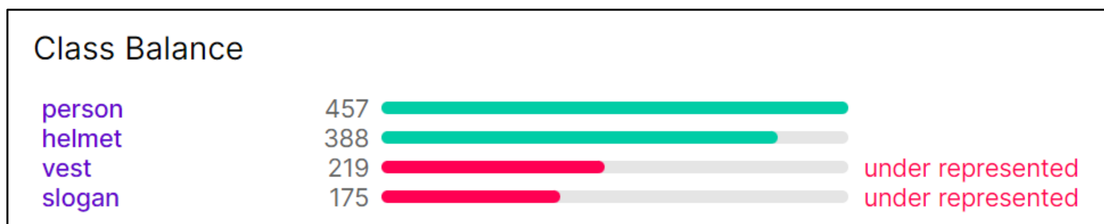


Figure E-2: Class balances

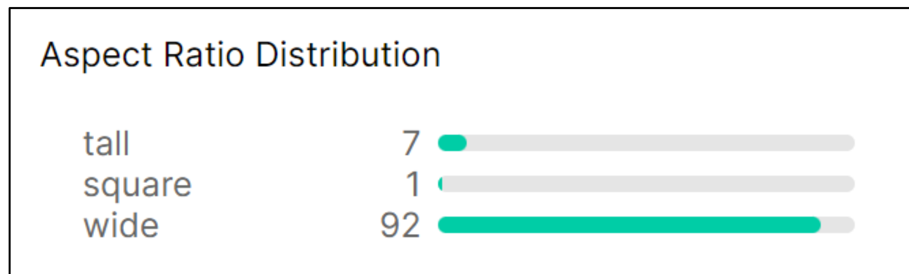


Figure E-3: Aspect ratio distribution

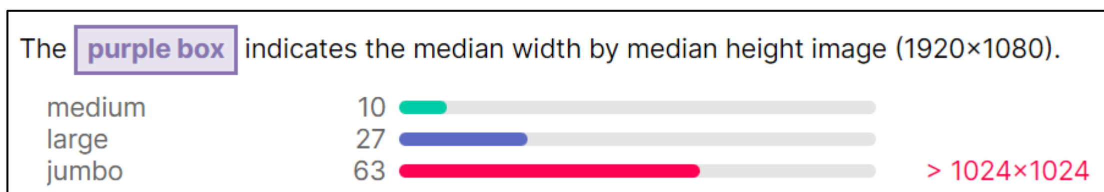


Figure E-4: Image size count

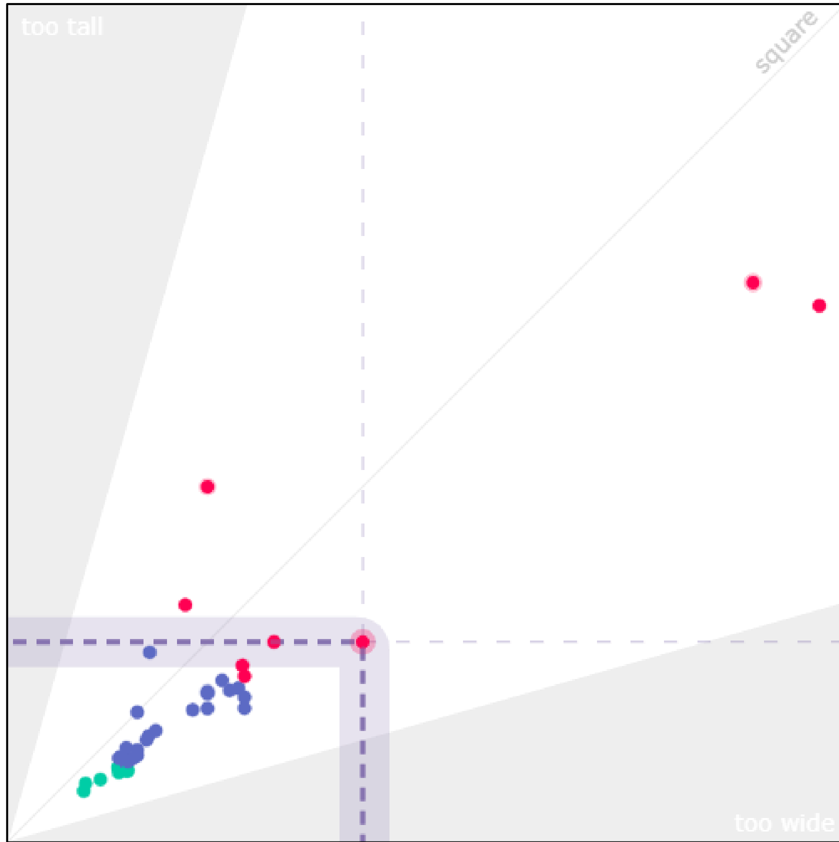


Figure E-5: Distribution of image size

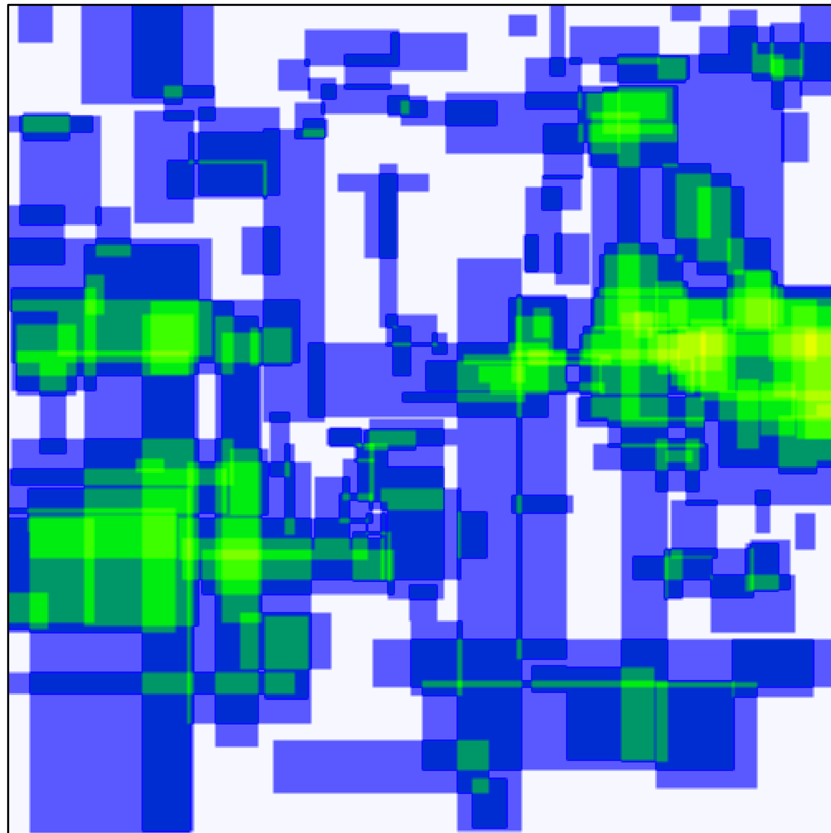


Figure E-6: Annotation heat map for slogan

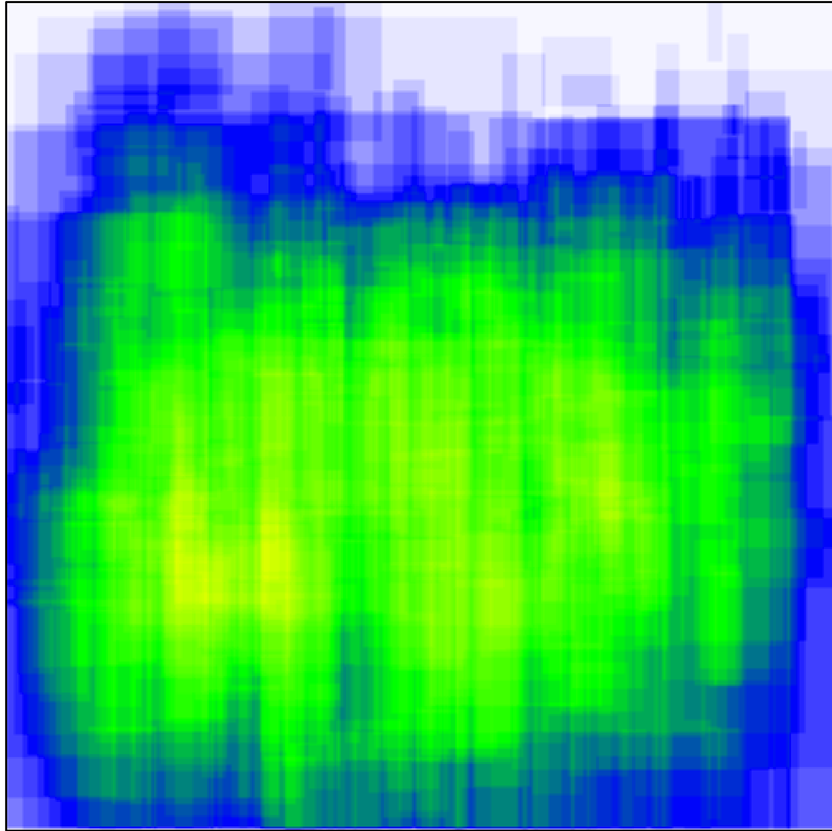


Figure E-7: Annotation heat map for person

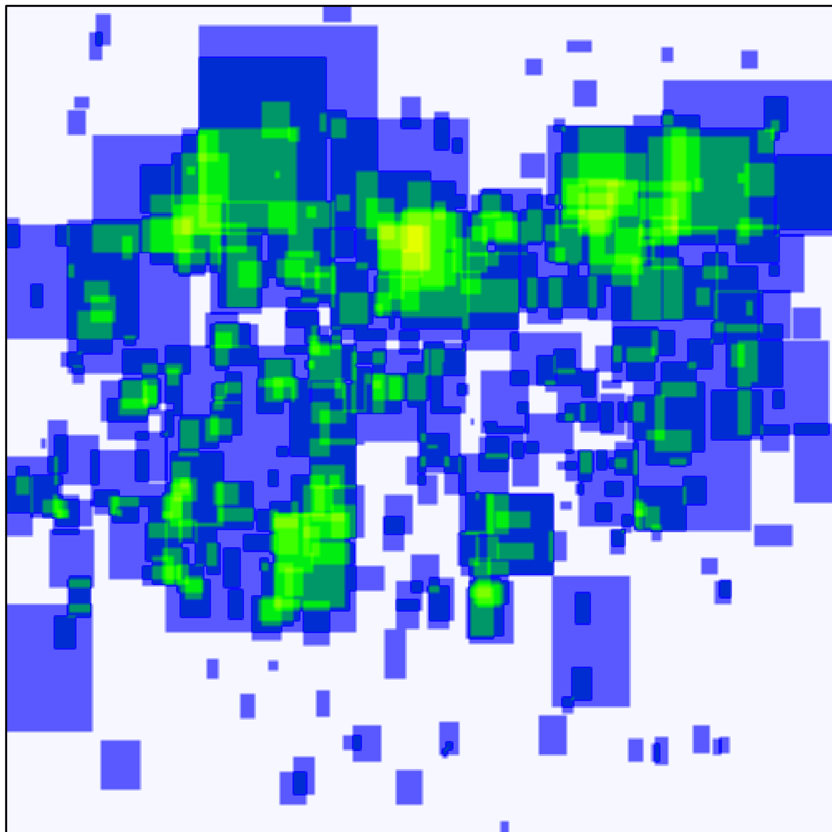


Figure E-8: Annotation heat map for helmet

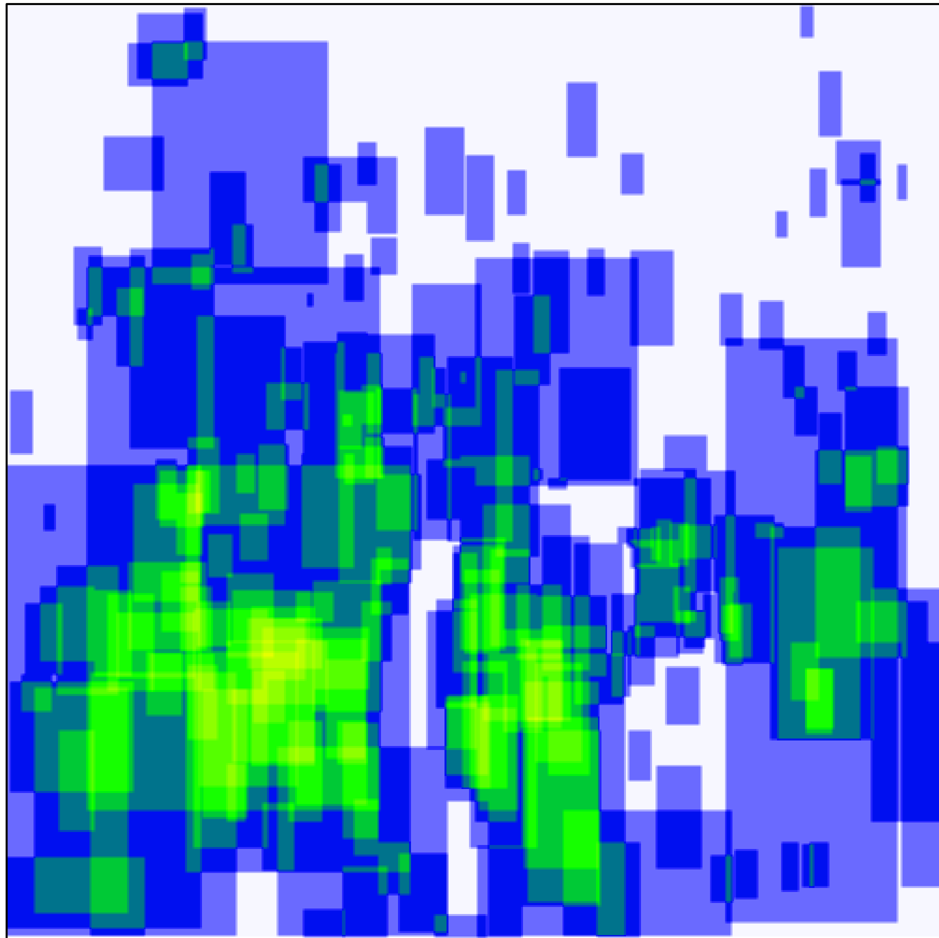


Figure E-9: Annotation heart map for vest

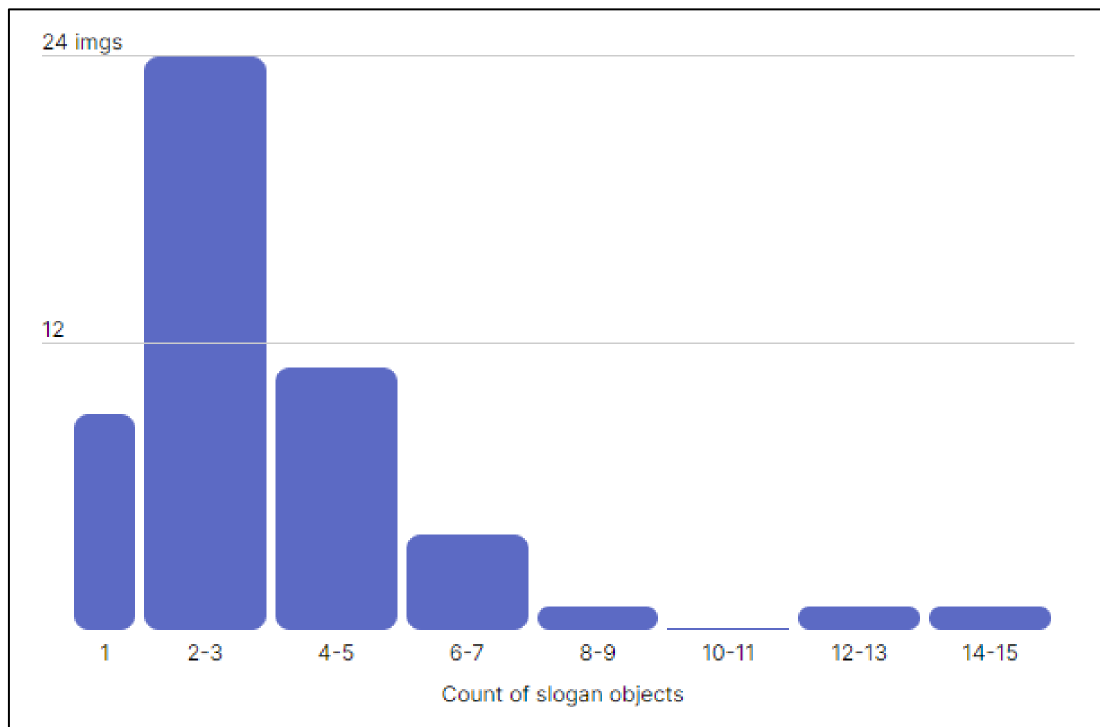


Figure E-10: Count histogram for slogan

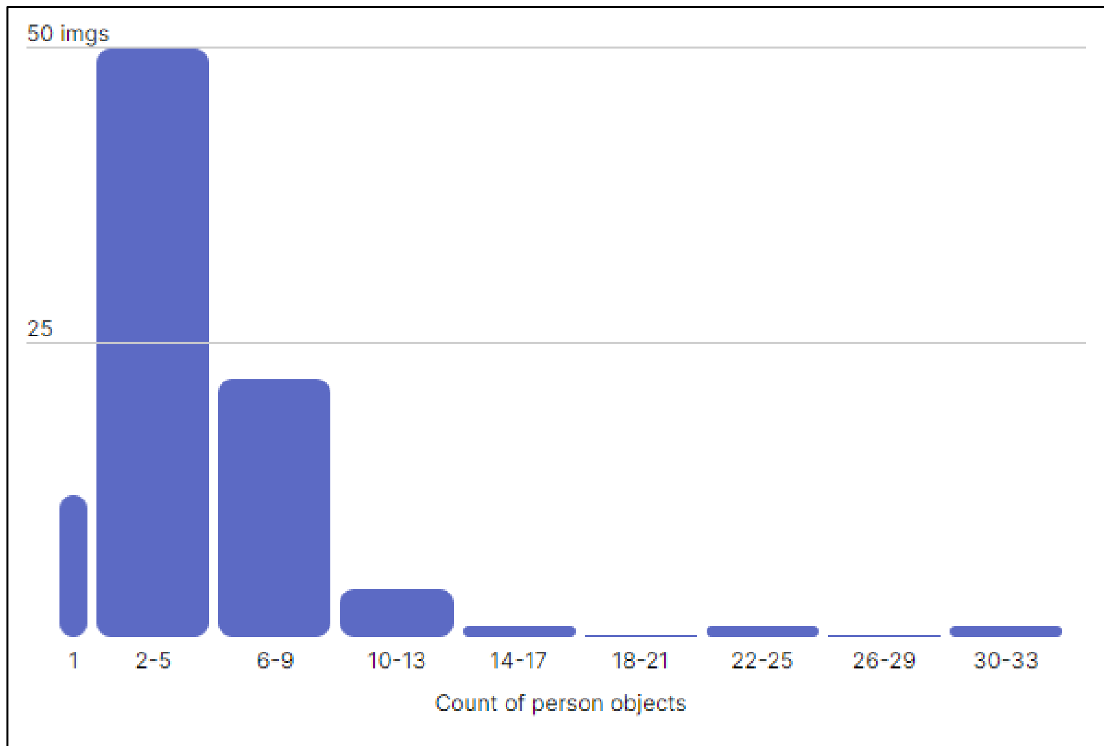


Figure E-11: Count histogram for person

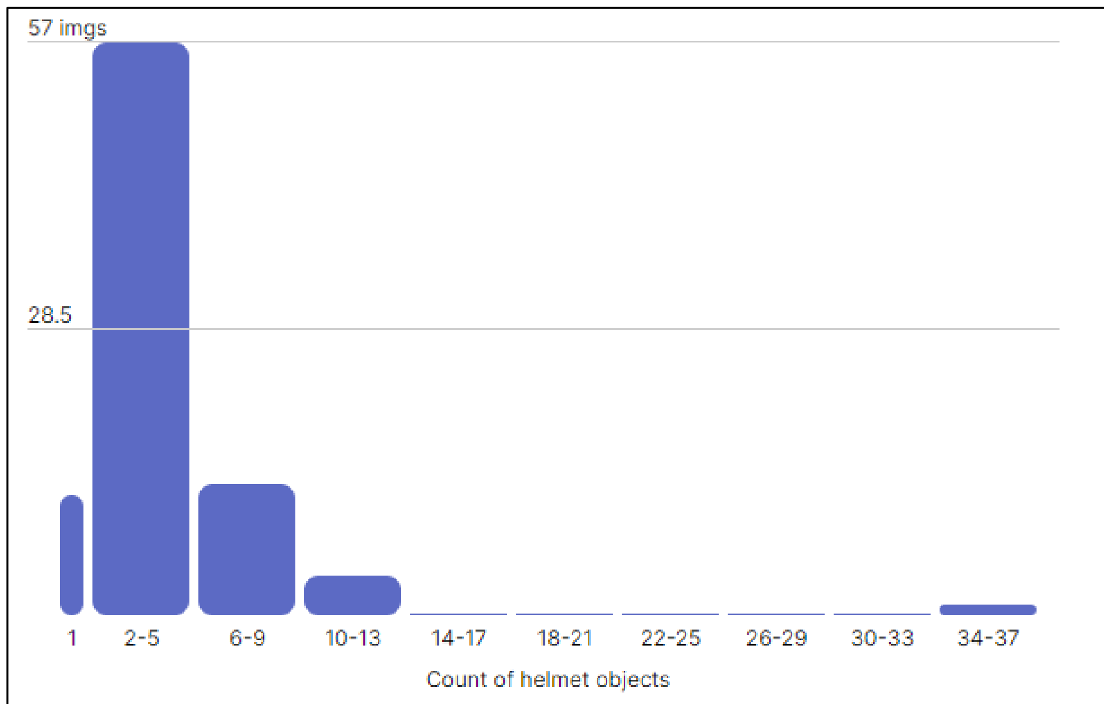


Figure E-12: Count histogram for helmet

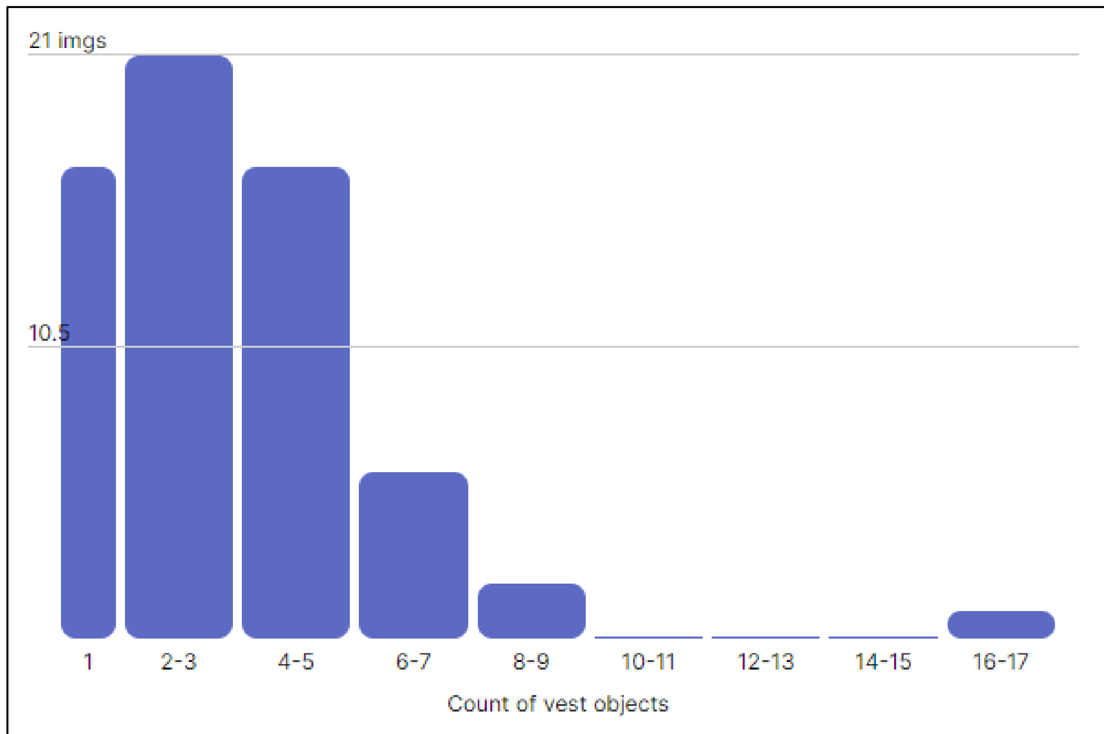


Figure E-13: Count histogram for vest

Appendix F: YOLOv4 results

Figure F-1: YOLOv4 results 1

Figure F-2: YOLOv4 results 2

Figure F-3: YOLOv4 results 3

Figure F-4: YOLOv4 results 4

Figure F-5: YOLOv4 results 5

Figure F-6: YOLOv4 results 6

Figure F-7: YOLOv4 results 7

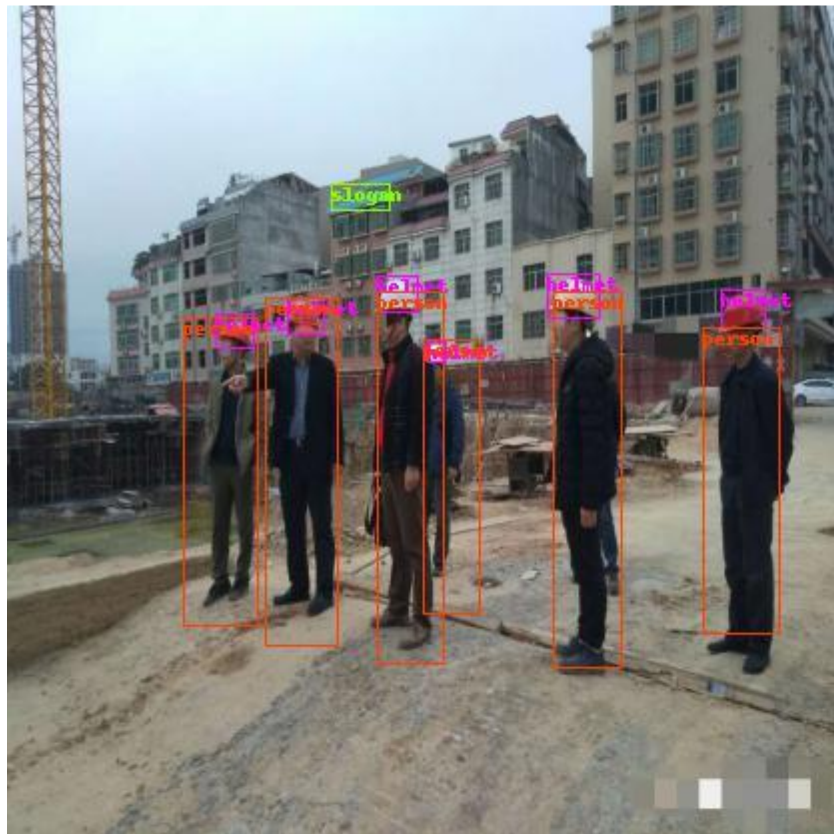


Figure F-8: YOLOv4 results 8

Figure F-9: YOLOv4 results 9

Figure F-10: YOLOv4 results 10

Appendix G: Detectron2 results

Figure G-1: Detectron2 results 1

Figure G-2: Detectron2 results 2

Figure G-3: Detectron2 results 3

Figure G-4: Detectron2 results 4

Figure G-5: Detectron2 results 5

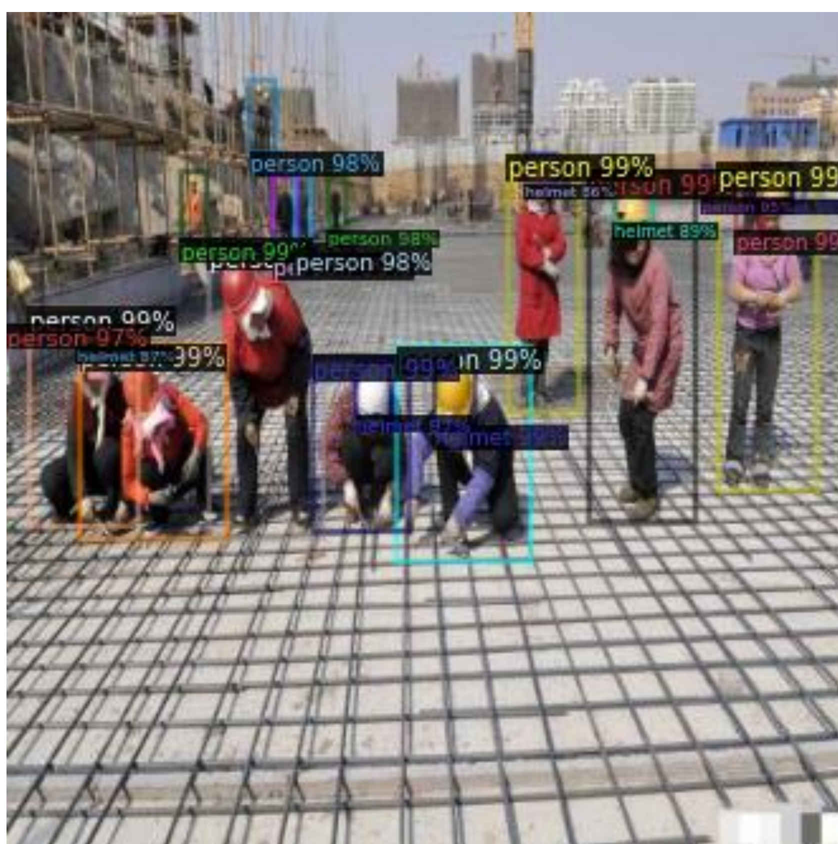


Figure G-6: Detectron2 results 6

Figure G-7: Detectron2 results 7

Figure G-8: Detectron2 results 8

Figure G-9: Detectron2 results 9

Figure G-10: Detectron2 results 10

Appendix H: EfficientDet results

Figure H-1: EfficientDet Results 1

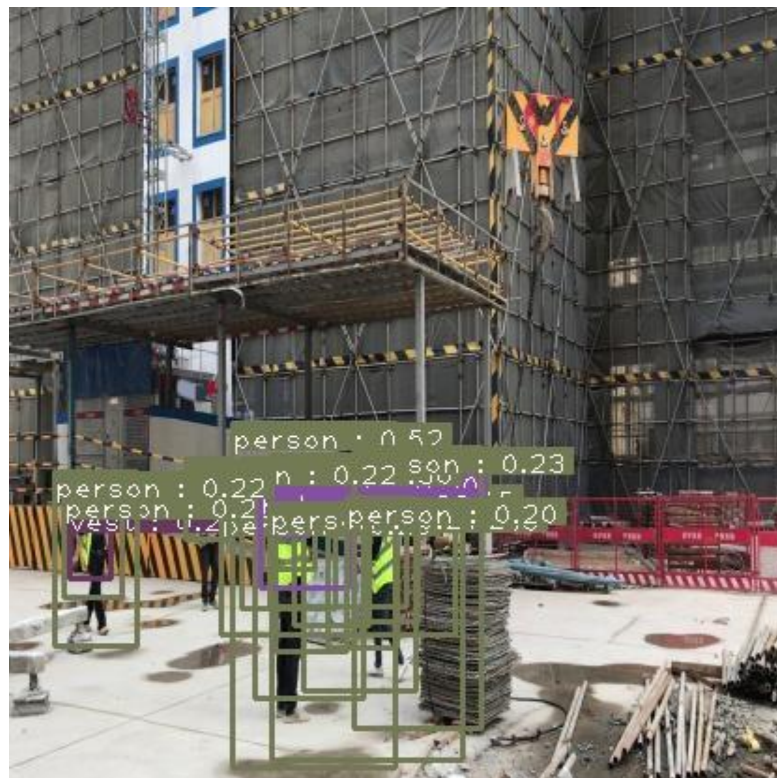


Figure H-2: EfficientDet Results 2

Figure H-3: EfficientDet Results 3

Figure H-4: EfficientDet Results 4

Figure H-5: EfficientDet Results 5

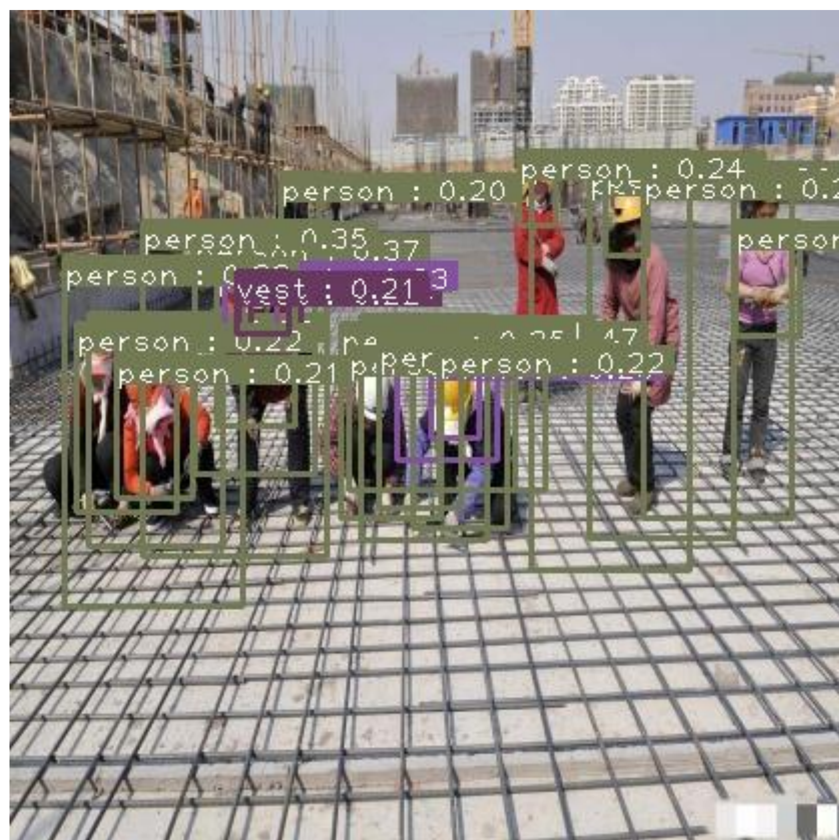


Figure H-6: EfficientDet Results 6

Figure H-7: EfficientDet Results 7

Figure H-8: EfficientDet Results 8

Figure H-9: EfficientDet Results 9

Figure H-10: EfficientDet Results 10

Appendix I: YOLOv4 Ground truth-Prediction Table

PERSON			
Image	Ground Truth	Total	Prediction
1	a	5	1
	b		1
	c		1
	d		0
	e		0
2	a	5	1
	b		1
	c		1
	d		1
	e		1
3	a	7	1
	b		1
	c		1
	d		1
	e		0
	f		0
	g		0
6	a	12	1
	b		1
	c		1
	d		1
	e		1
	f		1
	g		1
	h		1
	i		0
	j		0
	k		0
	l		0
7	a	9	1
	b		1
	c		1
	d		1
	e		1
	f		1
	g		1
	h		0
	i		0
8	a	6	1
	b		1
	c		1
	d		1
	e		1
	f		1
9	a	9	1
	b		1
	c		1
	d		1
	e		1
	f		1
	g		1
	h		1
	i		0
10	a	1	0
	Grand total	54	41
	Accuracy	75.93%	

Table I-1: Person Ground truth-Prediction Table

HELMET			
Image	Ground Truth	Total	Prediction
1	a	5	1
	b		1
	c		1
	d		1
	e		0
2	a	5	1
	b		1
	c		1
	d		1
	e		1
3	a	6	1
	b		1
	c		1
	d		1
	e		1
	f		0
6	f	10	1
	a		1
	b		1
	c		1
	d		1
	e		0
	f		0
	g		0
	h		0
	i		0
7	a	9	1
	b		1
	c		1
	d		1
	e		1
	f		0
	g		0
	h		0
	i		0
8	a	6	1
	b		1
	c		1
	d		1
	e		1
	f		1
9	a	8	1
	c		1
	d		1
	e		1
	f		1
	g		1
	h		1
	i		1
10	h	1	1
	Grand total	50	39
	Accuracy	78.00%	

Table I-2: Helmet Ground truth-Prediction Table

VEST			
Image	**Ground Truth**	**Total**	**Prediction**
1	a	4	1
	b		1
	c		1
	d		1
2	a	5	1
	b		1
	c		1
	d		0
	e		0
3	a	7	1
	b		1
	c		1
	d		1
	e		1
	f		1
	g		0
	Grand Total	16	13
	Accuracy	81.25%	

Table I-3: Vest Ground truth-Prediction Table

SLOGAN			
Image	**Ground Truth**	**Total**	**Prediction**
1	a	1	0
3	a	3	1
	b		1
	c		1
4	a	4	0
	b		0
	c		0
	d		0
5	a	2	0
	b		0
	Grand total	10	3
	Accuracy	30.00%	

Table I-4: Slogan Ground truth-Prediction Table

Appendix J: Detectron2 Ground truth-Prediction Table

PERSON			
Image	Ground Truth	Total	Prediction
1	a	5	1
	b		1
	c		1
	d		1
	e		1
2	a	5	1
	b		1
	c		1
	d		1
	e		1
3	a	7	1
	b		1
	c		1
	d		1
	e		1
	f		1
	g		1
6	a	12	1
	b		1
	c		1
	d		1
	e		1
	f		1
	g		1
	h		1
	i		1
	j		1
	k		1
	l		1
7	a	9	1
	b		1
	c		1
	d		1
	e		1
	f		1
	g		1
	h		0
	i		0
8	a	6	1
	b		1
	c		1
	d		1
	e		1
	f		1
9	a	9	1
	b		1
	c		1
	d		1
	e		1
	f		1
	g		1
	h		1
	i		0
10	a	1	1
	Grand total	54	51
	Accuracy	94.44%	

Table J-1: Person Ground truth-Prediction Table

HELMET			
Image	Ground Truth	Total	Prediction
1	a	5	1
	b		1
	c		1
	d		1
	e		1
2	a	5	1
	b		1
	c		1
	d		1
	e		1
3	a	6	1
	b		1
	c		1
	d		0
	e		0
	f		0
6	a	10	1
	b		1
	c		1
	d		1
	e		1
	f		1
	g		0
	h		0
	i		0
	j		0
7	a	9	1
	b		1
	c		1
	d		1
	e		1
	f		0
	g		0
	h		0
	i		0
8	a	6	1
	b		1
	c		1
	d		1
	e		1
	f		1
9	a	8	1
	b		1
	c		1
	d		1
	e		1
	f		1
	g		0
	h		0
10	a	1	1
	Grand total	50	37
	Accuracy	74.00%	

Table J-2: Helmet Ground truth-Prediction Table

VEST			
Image	Ground Truth	Total	Prediction
1	a	4	0
	b		0
	c		0
	d		0
2	a	5	0
	b		0
	c		0
	d		0
	e		0
3	a	7	0
	b		0
	c		0
	d		0
	e		0
	f		0
	g		0
	Grand Total	16	0
	Accuracy	0.00%	

Table J-3: Vest Ground truth-Prediction Table

SLOGAN			
Image	Ground Truth	Total	Prediction
1	a	1	1
3	a	3	1
	b		1
	c		1
4	a	4	1
	b		1
	c		0
	d		0
5	a	2	0
	b		0
	Grand total	10	6
	Accuracy	60.00%	

Table J-4: Slogan Ground truth-Prediction Table

Appendix K: Average detection time

YOLOv4		
Image	Detection Time (ms)	
1	4030	
2	3940	
3	4140	
4	4730	
5	4130	
6	4030	
7	4130	
8	4140	
9	4130	
10	4030	
	4143	**Average**

Table K-1: Average detection time for YOLOv4

EfficientDet		
Image	Detection Time (ms)	
1	646	
2	278	
3	260	
4	261	
5	268	
6	272	
7	268	
8	274	
9	266	
10	260	
	305.3	**Average**

Table K-2: Average detection time for EfficientDet