|  |  |
| --- | --- |
|  |  |

**Object Detection in Automated Software Testing**

|  |  |
| --- | --- |
| Author  Intern  Extern  Company | Brent Gerets  Bart Stukken  David Vandingenen  Brightest |

Abstract

Content

[1 Introduction 2](#_Toc165376870)

[2 Material and methods 3](#_Toc165376871)

[2.1 Programming language, UI framework, environment 3](#_Toc165376872)

[2.2 Object detection model comparison 3](#_Toc165376873)

[2.2.1 Metrics and models 3](#_Toc165376874)

[2.2.2 Comparison 4](#_Toc165376875)

[2.3 Data augmentation 6](#_Toc165376876)

[3 Results 7](#_Toc165376877)

[3.1 Data augmentation 7](#_Toc165376878)

[3.2 Implementation of models 7](#_Toc165376879)

[3.2.1 YOLOv9 7](#_Toc165376880)

[3.2.2 RT-DETR 7](#_Toc165376881)

[3.2.3 Co-DINO 7](#_Toc165376882)

[3.3 Application 7](#_Toc165376883)

[3.3.1 Dataset creator 7](#_Toc165376884)

[3.3.2 Training 10](#_Toc165376885)

[3.3.3 Testing 11](#_Toc165376886)

[3.3.4 Settings and FAQ 12](#_Toc165376887)

[3.4 API 13](#_Toc165376888)

[3.5 Primary flow 13](#_Toc165376889)

[4 Discussion 14](#_Toc165376890)

[4.1 Implementation of models 14](#_Toc165376891)

[4.1.1 YOLOv9 14](#_Toc165376892)

[4.1.2 RT-DETR 14](#_Toc165376893)

[4.1.3 Co-DINO 14](#_Toc165376894)

[4.2 Application 14](#_Toc165376895)

[4.2.1 Dataset creator 14](#_Toc165376896)

[4.2.2 Training 15](#_Toc165376897)

[4.2.3 Testing 15](#_Toc165376898)

[4.2.4 Settings 15](#_Toc165376899)

[4.3 API 15](#_Toc165376900)

[4.4 Primary flow 15](#_Toc165376901)

[5 Conclusion 15](#_Toc165376902)

[6 Reference list 15](#_Toc165376903)

[7 Attachment 18](#_Toc165376904)

[7.1 Comparison of object detection models 18](#_Toc165376905)

1. Introduction

In recent years, extensive research has resulted in great leaps in the field of machine learning. Many different machine learning models are available to be used by anyone, in a wide variety of applications. One such application that has not received as much attention as others is automated software testing. A reason for this might be that machine learning is a great solution to automate solving problems that previously could only be solved efficiently by humans (e.g. detecting cats in images). Meanwhile, many solutions exist for automated software testing. For example, Selenium is a library available in many languages that facilitates automated testing in the browser [1]. The library allows for interaction with web pages by retrieving the desired elements using XPath, id, class name, etc. [2]. In most cases this works fine. However, being able to retrieve an element using the library is not an assurance that it is visible to the user. Thus, the goal of the project is to discover whether machine learning is a viable solution to this problem. More specifically, object detection is the subcategory of machine learning that applies in this case. The research question can then be formulated as: ‘How can object detection be used in the automated software testing process?’.

In more practical terms, the goal of the project is to develop a proof-of-concept application of the use of object detection in automated testing. This includes deciding which object detection model(s) to use and developing a user interface to facilitate the training and testing of the object detection models on different web pages, as well as an API that can be used to perform object detection inference in an automated testing environment. Developing an object detection model is not in the scope since there are many pre-trained models available that can be applied to the project. The input of the model consists of screenshots of different pages of the website with bounding boxes around the elements that are to be detected. The model should be able to detect the elements at different positions on the screen and different scales, most other variations should not be detected since this should make the test fail (e.g. an element in the wrong colour should not be detected).

A few papers on the topic of user interface element detection have been published [3, 4, 5, 6, 7, 8]. However, all except one share a common problem that prevents their solutions from being applied to this project. Namely, they attempt to detect as many UI elements as possible and classifying them in certain categories (e.g. text, image, button, slider…). However, the aforementioned use case requires the detection of specific elements to confirm whether they are being displayed and subsequently perform actions on them. For instance, it is not the desired result to classify all buttons on a web page as buttons, but rather to detect all instances of a specific button. *Yeh et al.* [8] did propose a solution for detecting UI elements by providing screenshots of the specific elements. Their solution looks promising but unfortunately the paper was published in 2009, making the technology used in the paper significantly outdated. Nonetheless their paper can provide some useful insights.

First, section 2 discusses the materials and methods that were employed in the project. More specifically, the programming language and frameworks (2.1), object detection models (2.2), and data augmentation (2.3). Next, section 3 presents the results that were achieved. Section 4 then proceeds to examine and discuss these results. Finally, section 5 formulates a conclusion and reflection.

1. Material and methods

This section discusses the materials and methods that were employed to achieve the results presented in section 3. It also covers the reasoning behind the choice to use these materials and methods over other alternatives.

* 1. Programming language, UI framework, environment

Often one of the first decisions is which programming language and/or frameworks will be used. Python (object-oriented) is used as the programming language for the application for several reasons. Firstly, Python is the most popular language when it comes to machine learning and data science as a whole, thanks to its ease of use and rich collection of libraries and frameworks [9]. Moreover, Python is and has been the most popular programming language in general in recent years, according to the TIOBE index [10].

NiceGUI [11] is used to create the user interface of the application. It uses the browser as the frontend of the Python code which uses Vue and Quasar, though the application will exclusively be run locally and is not actually a website. NiceGUI also allows Tailwind classes to be used for styling. The reason for choosing NiceGUI is its modern look compared to native frameworks such as Tkinter [12]. It also allows for hot reloads when making changes (though they take a few seconds) and building pages is very straight-forward. NiceGUI is a new framework but is maintained and frequently updated by the open-source community.

Google Colab [13] was used to speed up training by making use of its cloud GPUs, specifically the T4 GPU with 15GB of VRAM. Around 300 ‘compute units’ were purchased (€11.19 per 100 units at 1.76 units per hour) during the course of the project to increase the GPU usage limits. These compute units also unlocked better GPUs with more VRAM (A100 16GB, L4 22.5GB) but also cost over twice and a half times as much and are significantly harder to get access to without a Colab Pro+ or Enterprise subscription. The PXL was also contacted to determine whether it was possible to provide GPUs, which it unfortunately was not. Outside of training, IntelliJ’s PyCharm was used as IDE during development.

* 1. Object detection model comparison

To perform object detection, pretrained models are used and fine-tuned on a dataset containing the webpage elements. A large variety of pretrained models is available, so to avoid spending unnecessary time testing all of them, a selection is made based on the most important qualifiers. These were deemed to be accuracy, speed, and availability.

* + 1. Metrics and models

This section discusses which models will be compared and which metrics will be used to compare them. Firstly, accuracy determines how good a model is at performing object detection on a specific dataset. In particular, the popular Common Objects in Context (COCO) dataset [14]. Every model that was researched performed a benchmark on the COCO dataset. The COCO validation set was the most common, a few used the test-dev set, and some others didn’t specify. The accuracies between these different COCO sets are generally very similar. The accuracy on COCO is expressed as the average precision (AP) with an Intersection over Union (IoU) threshold ranging from 0.5 to 0.95 with steps of 0.05 [15]. This threshold determines when a bounding box is accepted as correct. The IoU is calculated as follows [16]:

This means that if the bounding box produced by the model and the true bounding box align perfectly, the IoU will be 1. The AP is usually represented as a percentage and is sometimes also called mean average precision (mAP).

Secondly, speed refers to how much time a model spends to detect objects in an image. This is dependent on the quality of hardware that is used. Consequently, it is difficult to compare the speed of different models without performing benchmarking for all of them. However, the number of Floating-Point Operations (FLOPs[[1]](#footnote-1)) that the model must perform on an image can function as an estimate of efficiency and complexity [17]. Therefore, it can also function as an estimate of how much time a model needs to process an image. Some models unfortunately did not mention the number of FLOPs so the Frames Per Second (FPS) were used as the alternative.

Lastly, the availability of a model determines how easy it is to get the model up and running. The availability was split into three categories from low to high availability: ‘download’, ‘library’, and ‘library without the need for data loading’. A model that is only available through download is the hardest to get up and running since the entire setup must be figured out and performed by the developer. A model that is available directly through a library or framework is easier to use because their documentation tends to be relatively thorough. However, the data must still be loaded in and modified appropriately by the developer. Some libraries don’t require this and simply use a configuration file, these models are easiest to use and set up. The reason some degree of importance is attached to availability is that high availability ensures the model is functioning correctly and at its best performance.

There is large collection of object detection models available. Comparing every one of them however would be a waste of time. A considerable portion of models are already outdated and easily surpassed by newer state-of-the-art models. The determine which models to compare, three different avenues where explored. First, a brief literature study of object detection model reviews was conducted to ascertain the most popular models. From this literature study can be concluded that YOLO, Faster R-CNN, and SSD are three of the most popular models [18, 19, 20, 21, 22]. Another avenue trough which to find popular models is to check which models are available directly from machine learning frameworks such as PyTorch [23], TensorFlow [24] and Ultralytics [25]. These models are Faster-CNN, SSD, FCOS, RetinaNet, YOLO, RT-DETR, and more. Lastly, another way to find models is to study the COCO leaderboard [26]. Most of the models that appear in a published paper and have benchmarked on the COCO test-dev set appear on this leaderboard. Currently, Co-DETR is the best performing model in terms of accuracy. Co-DETR and some other state-of-the-art models were found on the leaderboard and compared.

* + 1. Comparison

Taking inspiration from the work of *Ouchra and Belangour* [27], a Weighted Scoring Model (WSM) was used to determine which models should be the focus of the project. Table 1 contains all the models that were compared together with their accuracy score, speed score, availability score and total weighted score. The scores were derived from their respective specifications. As mentioned in the previous section, the accuracy was determined using the reported accuracy of the model on the COCO dataset. The accuracy score was then calculated as a value between 0 and 100, where the model with the lowest accuracy has a score of 0, and the highest a score of 100, the rest in between. The speed score was calculated in the same way, lowest number of FLOPs gets a score of 100, highest a score of 0. If the FLOPs were not reported, the FPS was used as a fallback. As mentioned in the previous chapter, the availability was split into three categories. The lowest availability category receives a score of 0, the middle category 50, and the highest availability category gets a score of 100. Attachment 7.1 contains the complete table with all the accuracy, FLOPs and FPS metrics, as well as the sources of the data. Finally, a weight between 0 and 1 was assigned to the accuracy, speed, and availability. This weight represents the expected significance of each metric in the context of the project. Since speed is not imperative, it was assigned a weight of 0.3, while accuracy was assigned a weight of 0.6. The availability receives a very small weight of 0.1 since it is the least important factor that determines the quality of the model. These weights are by no means set in stone, but changing the weights slightly results in the same models receiving the highest score. The weighted total score is then once again a score between 0 and 100. The top 3 models are highlighted in the table below: YOLOv9-E, RT-DETR-X, and Co-DINO. The accuracy of YOLOv9-E and RT-DETR-X is around 10% lower than Co-DINO, which is significant. However, the combination of superior speed and availability places these models right behind Co-DINO. Even if the weights are changed slightly (while still reflecting their importance in the project), the YOLOv9, RT-DETR and Co-DINO models always end up in the top 5. In the case of YOLOv9 and RT-DETR a variant other than the highlighted one could be used since the scores are very similar and the slight increase in accuracy might not be worth the decreased speed. Usually, variants of a model perform similarly. However, Co-DETR is a special case since it is more of a technique that can be applied to existing DETR models. Therefore, the two Co-DETR variants in the table perform very differently due to Co-DINO and Co-Deformable-DETR being completely different DETR models. From this weighted scoring model comparison can be concluded that YOLOv9, RT-DETR, and Co-DINO should be the focus of the project. Unfortunately, a lot of problems were encountered with Co-DINO and it ended up not being implemented in the application.

Table 1: Weighted Score Model comparison of popular and/or state-of-the-art object detection models (full version: 7.1).

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Model name** | **Model variant** | **Accuracy score** | **Speed score** | **Availability score** | **Weighted total score** |
| YOLOX | s | 42.30 | 98.98 | 0.00 | 55.07 |
| m | 57.95 | 96.33 | 0.00 | 63.67 |
| l | 64.79 | 91.72 | 0.00 | 66.39 |
| x | 68.22 | 84.59 | 0.00 | 66.31 |
| YOLOv7 |  | 68.46 | 94.59 | 0.00 | 69.45 |
| X | 72.62 | 89.78 | 0.00 | 70.50 |
| W6 | 76.77 | 80.19 | 0.00 | 70.12 |
| E6 | 79.95 | 71.44 | 0.00 | 69.40 |
| D6 | 80.93 | 54.99 | 0.00 | 65.06 |
| E6E | 82.15 | 52.94 | 0.00 | 65.17 |
| YOLOv8 | n | 34.47 | 100.00 | 100.00 | 60.68 |
| s | 53.06 | 98.88 | 100.00 | 71.50 |
| m | 66.01 | 96.06 | 100.00 | 78.43 |
| l | 72.62 | 91.17 | 100.00 | 80.92 |
| x | 75.06 | 85.95 | 100.00 | 80.82 |
| YOLOv9 | S | 57.70 | 98.98 | 100.00 | 74.32 |
| M | 68.95 | 96.16 | 100.00 | 80.22 |
| C | 72.86 | 94.69 | 100.00 | 82.12 |
| **E** | **79.22** | **89.64** | **100.00** | **84.42** |
| RT-DETR | L | 72.86 | 94.29 | 100.00 | 82.00 |
| **X** | **77.26** | **87.29** | **100.00** | **82.55** |
| Co-DETR | **Co-DINO** | **100.00** | **84.76** | **0.00** | **85.43** |
| Co-Deformable-DETR | 86.31 | 51.99 | 0.00 | 67.38 |
| Faster R-CNN | R50-FPN | 33.74 | 75.28 | 50.00 | 47.83 |
| R50-FPN | 41.56 | 13.17 | 50.00 | 33.89 |
| R101-FPN | 45.97 | 8.85 | 50.00 | 35.23 |
| X101-FPN | 48.41 | 2.79 | 50.00 | 34.88 |
| R-FCN |  | 20.29 | 0.00 | 0.00 | 12.18 |
| SSD | 300 VGG16 | 0.00 | 34.24 | 50.00 | 15.27 |
| FCOS | R50-FPN | 40.59 | 89.80 | 50.00 | 56.29 |
| RetinaNet | R50-FPN | 32.27 | 70.77 | 50.00 | 45.60 |
| R50 | 37.90 | 11.93 | 50.00 | 31.32 |
| R101 | 42.05 | 11.93 | 50.00 | 33.81 |
| InternImage | T | 63.33 | 85.26 | 0.00 | 63.57 |
| S | 64.79 | 81.32 | 0.00 | 63.27 |
| B | 66.26 | 72.24 | 0.00 | 61.43 |
| L | 80.44 | 21.60 | 0.00 | 54.74 |
| XL | 80.68 | 0.00 | 0.00 | 48.41 |
| **Weight** |  | **0.60** | **0.30** | **0.10** |  |

* 1. Data augmentation

Data augmentation is the practice of expanding a dataset by performing certain transformations on the elements of the dataset. This allows the model to learn invariant features and prevents overfitting [28]. It also increases the size of the dataset, which is important in this project since the dataset only consists of a few screenshots of the website. There are numerous augmentations that can be performed such as resizing, rotating, translating, changing colour, flipping, adding noise, etc. [29]. However, a lot of the augmentations are not desired in this project. For example, changing the colour of the images in the dataset is not desired since the model should only detect the elements in the correct colour. The same goes for flipping, rotating, and shearing. Nonetheless, there are a few data augmentations that can be applied in the context of this project. Firstly, the images can be resized to allow the model to detect the elements at different sizes. This is desired since changing the size of the browser window could change the dimensions of the elements within. It is important to resize only the part of the image within the bounding box. Resizing the entire image would be pointless since the image must be resized again before being input into the model. Secondly, the part of the image withing the bounding box can be translated to a different position inside the image to prevent the model from overfitting to a specific position. Lastly, even though the model shouldn’t be invariant to colour, noise can still be added to prevent overfitting and to allow the model to still detect the element when there are other elements in the vicinity. These are the most important data augmentations that can be applied. It should be noted that these augmentations should be applied to each other as well. For example, the original images are randomly resized. Then all the images, including the resized ones, are randomly translated. Finally, noise is added to all the images. Depending on the number of times an augmentation is applied, this can exponentially increase the size of the dataset.

1. Results
   1. Implementation of models
      1. YOLOv9
      2. RT-DETR
      3. Co-DINO

After learning how to use the MMDetection toolbox that was used to create the Co-DETR models and installing the correct Python version and required packages, Co-DINO was successfully tested on the COCO validation set. A new model configuration file was created that inherited the Co-DINO configuration [30], which in turn inherits the 5-scale ResNet-50 configuration [31]. The new configuration loads the checkpoint of the 5-scale Swin-L 16 epochs Co-DINO model (provided through Google Drive [32]) that was also pre-trained on the Objects365 dataset and has an mAP of 64.1 on COCO-val [33]. The configuration was set up to use the COCO dataset standard, but with different classes. The parent configurations also had to be slightly modified to allow for a different number of classes. This configuration was functional but required too much VRAM and was therefore failing before the first epoch could start. To prevent this issue, the backbone was frozen (no new weights calculated) by increasing the number of frozen stages from 1 to 4 in the original Co-DINO config file. The maximum image size was also decreased to a lower resolution. These solutions combined reduced the peak VRAM usage to 12GB.

The model was trained on several datasets created by using different parameters as input for the data augmentation script that was developed beforehand. With every dataset, the model would get to 90+% accuracy suspiciously quickly, often already right after the first epoch. After a few epochs to accuracy was 99+%. The greatest number of epochs that has been trained on a single dataset is 10 epochs. The last epoch had an accuracy of 99.82% Testing the model on new screenshots after training confirmed that something was wrong, since nothing was being detected apart from an incorrect label with less than 1% confidence every few screenshots. Work on Co-DINO was halted due to not much time being left until the end of the semester, and the much longer training time compared to other models (1 epoch per hour vs. 1 epoch every few minutes).

* 1. Application

An application was created to facilitate the training of the models on the websites of choice. The application also hosts a dataset creator which is used to create a dataset that can be used to train a model on a specific website. In addition, the prediction capabilities of the trained models can be manually tested using the application. Also, a settings page allows users to change default paths. Finally, there is a frequently asked questions page with usage information. The following sections discuss the individual pages of the app in further detail.

* + 1. Dataset creator

Training a model requires a dataset as input. Creating a dataset is not straightforward, especially when the user is unfamiliar with the model. To make this process simpler, a dataset creator was developed. Creating a dataset using the dataset creator requires three steps: configuration, drawing bounding boxes, and augmentation. Each of these steps is represented by a separate page which can be navigated between using a ‘stepper’.

The configuration step contains a radio menu with two options: ‘new dataset’ (Figure 1) and ‘existing dataset’ (Figure 2). The former prompts the user for a dataset name and save location. The latter requires the user to select a dataset save file (generated by the dataset creator). All file prompts in the application make use of a modified version of the local file picker example found in the NiceGUI Github repository [34] (Figure 3). A custom file picker is necessary due to the limitations of the native file picker. Mainly the fact that it does not allow the selection of a folder. Once a name and save location has been picked, or a save file has been successfully imported, the button to advance to the next step becomes available.

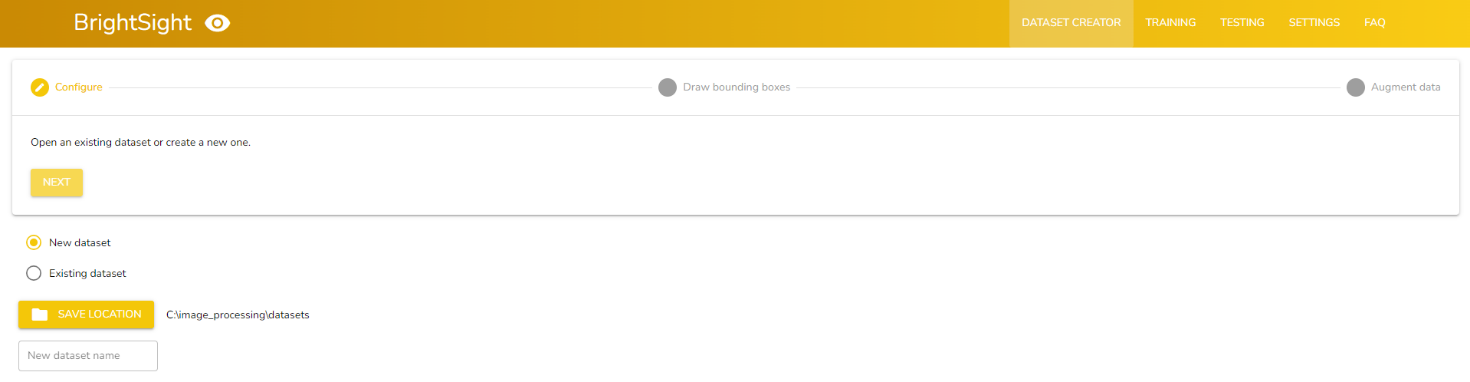


Figure 1: Dataset creator configuration step, new dataset.

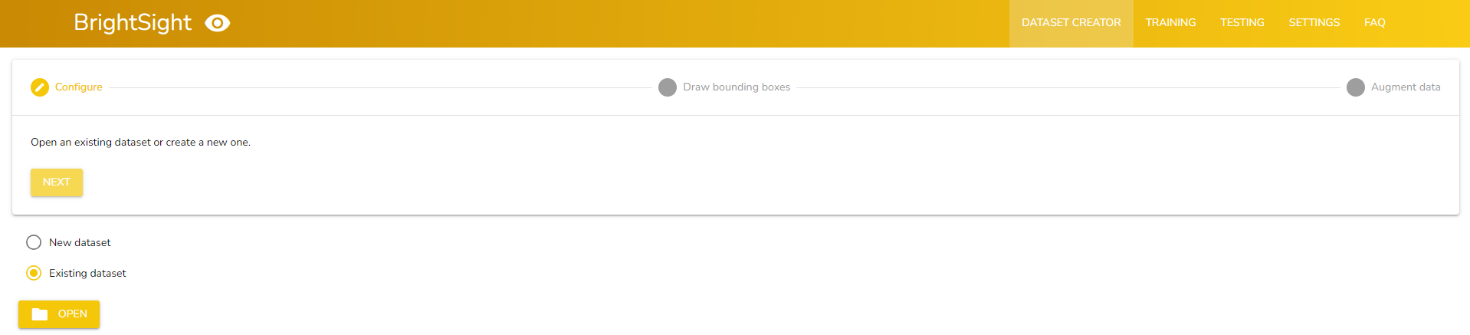


Figure 2: Dataset creator configuration step, existing dataset.

A screenshot of a computer

Description automatically generated

Figure 3: Custom local file picker.

In the second step (Figure 4), screenshots of the website must be uploaded. On each screenshot, bounding boxes can be drawn around every instance of each element that must be detected by clicking and dragging on the screenshot. These bounding boxes are also given a label to represent which element they encompass. When drawing a new bounding box, the selected label is assigned to it. Only one screenshot is displayed at a time on the canvas, which also allows for panning and zooming using respectively the right mouse button and the scroll wheel. The canvas can be returned to its original position using the button above it on the right. It is possible to switch between screenshots, edit labels, edit bounding boxes, and remove screenshots, labels, and bounding boxes. Labels can be edited by clicking the edit icon, the name and colour of the label can then be modified. The changes take effect immediately. Bounding boxes can be edited by clicking on them inside the canvas or selecting them in the list. Once selected, a bounding box can be moved, resized, and deleted. Finally, every time a destructive action is about to be taken (such as removing a label, and thus also all the bounding boxes with that label) a dialog is displayed to confirm the action (Figure 5). The divider between the lists and the canvas can be moved to increase the size of either. When the user is satisfied with their work, they can continue to the final step.

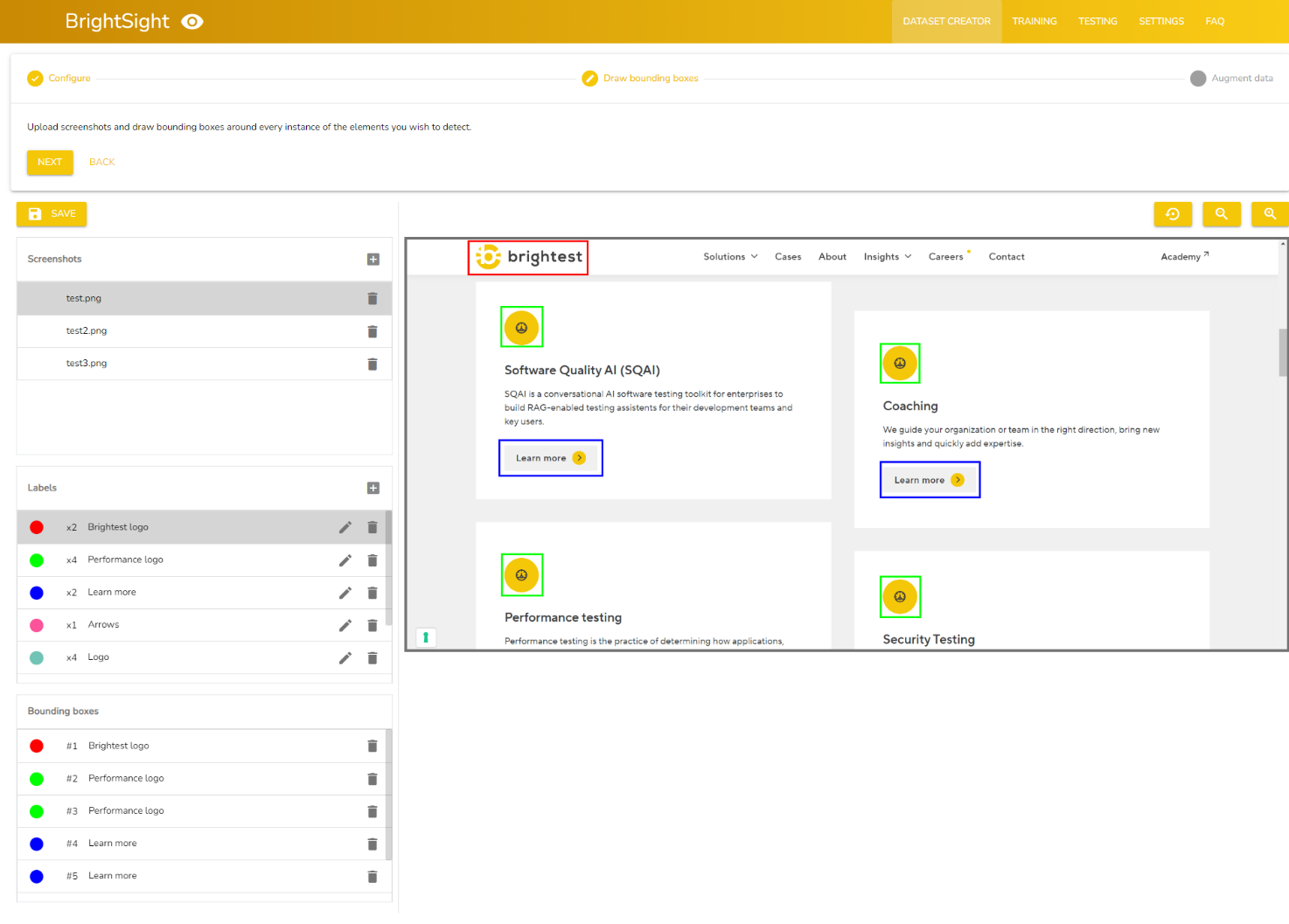


Figure 4: Dataset creator, drawing bounding boxes step.

A screenshot of a computer

Description automatically generated

Figure 5: Confirmation dialog.

The data augmentation step (Figure 6) extracts the bounding boxes from the screenshots which, together with the screenshots, undergo a series of augmentations to increase the size of the dataset and the variety within it. The augmentation page allows the user to select how many times each of the three augmentations (resizing, cropping, translating) will be applied to each image, and what the range of these operations will be. For example, a resize amount of 5 and resize range of 0.4 to 0.9 means every image is resized 5 times, with a random factor between 0.4 and 0.9. Pixels also have a random chance to be turned into noise, which can be configured. Lastly, there is a screenshot multiplier which multiplies the amount of times each of the three augmentations is executed, but only for the full screenshots. The goal of this is to increase the ratio of screenshots to bounding box images. A table containing the amount of input images and output images is displayed and dynamically updated. Once the configuration is completed, the data augmentation can be started. This step can take a while depending on the number of screenshots and bounding boxes, as well as the configured amounts for each of the augmentations. Two progress bars are displayed, one for the overall progress of the data augmentation, and one for the current step that the algorithm is performing. There is also a textbox which provides more feedback on the progress. After the data augmentation is completed, the provided export path will contain the dataset images and the required dataset files in different formats for training and testing.

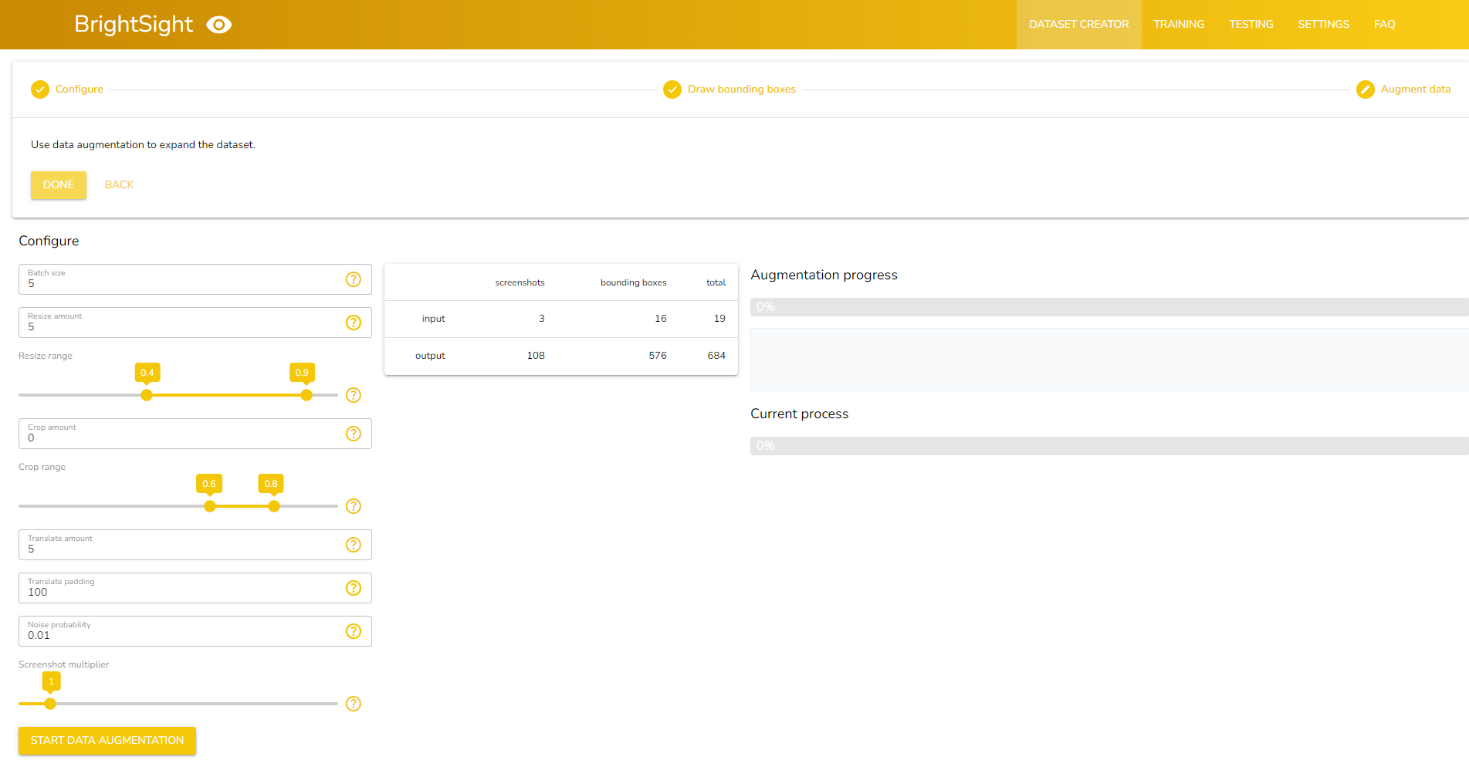


Figure 6: Dataset creator, data augmentation step.

The dataset creator automatically saves the current dataset when switching between steps, and a manual save button is available in the second step (Figure 4). To stop the user from accidentally leaving without saving during the second step, some JavaScript code is used to show a confirmation notification when leaving the page. Some inputs inside the application have a question mark icon displayed on the right side. When hovering over this icon, a tooltip is displayed with information about the value that the input expects. These hints can be turned off in the settings. The dataset creator can also be used on its own without using the rest of the application, since it outputs config files that are compatible with Ultralytics as well as COCO-style annotation files.

* + 1. Training

The training page is shown in Figure 7. Before starting training, a name must be given to the training run and models to train must be selected (either YOLOv9 and/or RT-DETR). Then a dataset file (the one generated by the dataset creator) must be selected and the batch size and number of epochs must be configured. During training the progress bar show the training progress and the console below shows some training output. The trained model is saved in the directory that can be configured in the settings.

A screenshot of a computer

Description automatically generated

Figure 7: Training page.

* + 1. Testing

The testing page requires the user to either create a new test run (Figure 8) or open an existing saved test run (Figure 9). A test run consists of a name, a training run that contains trained models, and a URL to the website to be tested.

A yellow and white rectangular object

Description automatically generated

Figure 8: Testing page, new test run.

A yellow and white rectangular object

Description automatically generated

Figure 9: Testing page, existing test run.

Once a test run is created or opened. Two buttons appear, one to open the browser and one to take a screenshot. Screenshots can only be taken when the browser is opened. The browser is opened using Selenium. In here the user can use the website and at some point press the ‘take screenshot’ button. The screenshot will then be used as input for the models. Once completed (usually a couple seconds per model) the original screenshot and the model outputs are shown (Figure 10). The output images contain bounding boxes with labels and confidence percentage. An image can be enlarged by clicking it. In this view, navigation between the outputs is still possible using the menu on the bottom (Figure 11). Out of the enlarged image view, it is possible to navigate between screenshots using the menu in the top right corner. The page also shows tables containing the speed of each model, the average confidence and amount detected per object per model.

A screenshot of a computer

Description automatically generated

Figure 10: Testing page, after taking a screenshot.

A screenshot of a computer

Description automatically generated

Figure 11: Testing page, enlarged image view.

* + 1. Settings and FAQ

The testing page (Figure 12) allows the user to change the default directories for the datasets, trained models and test runs. It also allows for the hints (icon with tooltip next to input field) to be turned off.

A yellow and white rectangular object

Description automatically generated

Figure 12: Settings page.

The frequently asked questions or FAQ page (Figure 13) contains answers to some questions that a new user might have on how to operate the application, sorted per page.

A screenshot of a computer

Description automatically generated

Figure 13: FAQ page.

* 1. API
  2. Primary flow

A diagram of the primary flow is shown in Figure 14. The flow consists of three broad steps, starting with the dataset creator. In here a new dataset can be created or an existing dataset can be opened and modified. The dataset is created by drawing bounding boxes on screenshots of the website and performing data augmentation. The next step is training. A name for the training run must be entered and the models to train must be selected. The dataset file (output from previous step) must be set, as well as the number of epochs to train and the batch size. After training a model it is possible to test it. This is done by selecting which training run to use (output from the previous step) and which URL the browser should open. It is also possible to open a saved test run and continue working with it. The browser is opened by Selenium when the ‘open browser’ button is pressed. Screenshots of the website can be taken via the application. The application then runs inference on the screenshot using all the selected models. The output screenshots are shown in the application and saved.

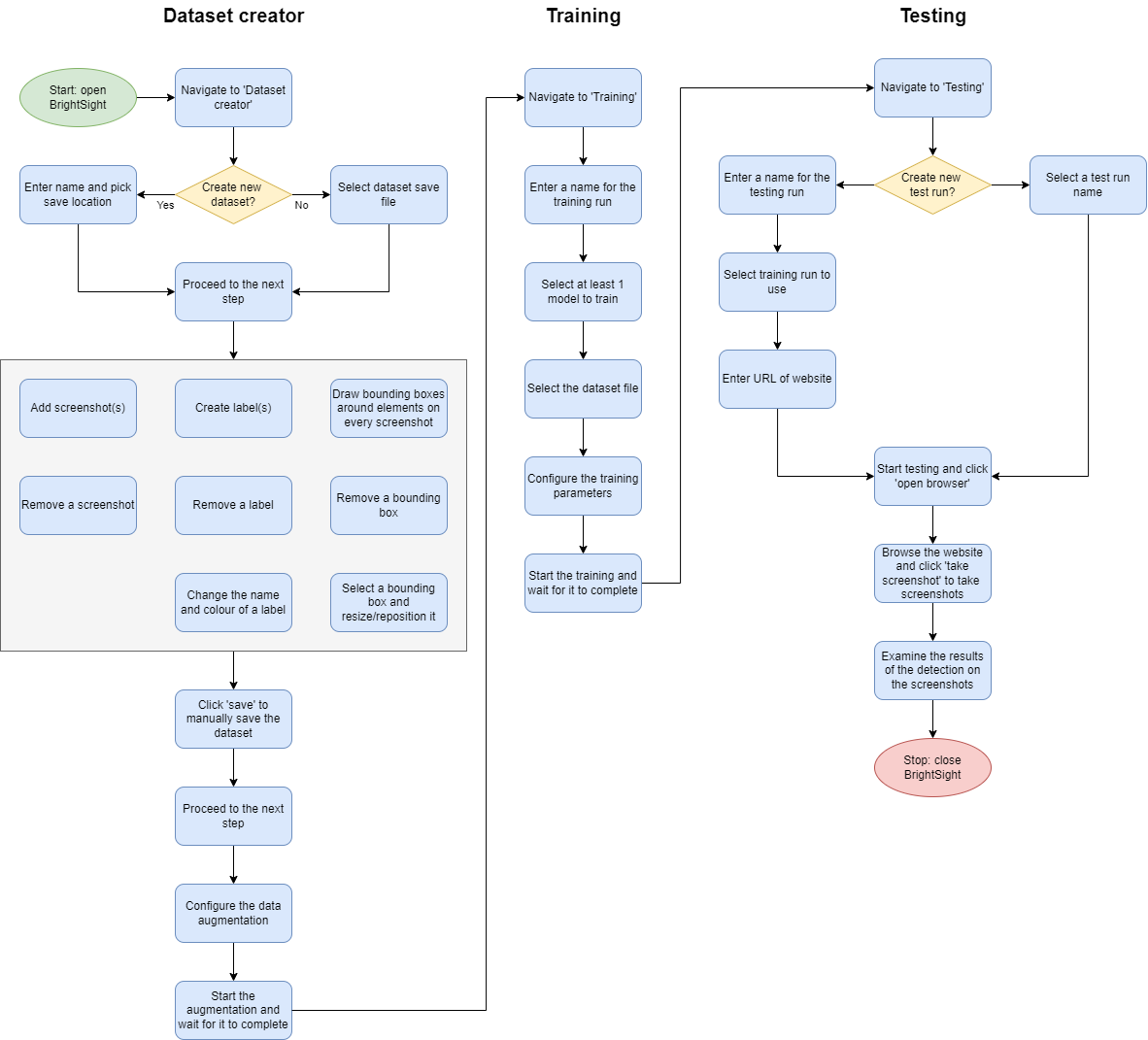


Figure 14: Primary flow diagram of the application.

1. Discussion
   1. Implementation of models

The architecture of the application is set up in such a way that new models can be added very easily. A new model class must simply inherit the base model class and implement its methods. The class must then be instantiated inside the model getter method in the training class, and it will be automatically added to the application in all the required places.

* + 1. YOLOv9
    2. RT-DETR
    3. Co-DINO

It is unfortunate that Co-DINO could not be implemented in the application, especially because it is the state-of-the-art at the moment. However, there were simply too many problems which caused training to start working later than the other models. At the point where training was working, there was not enough time left to troubleshoot this model and other aspects of the project had a higher priority. Co-DINO also took about 30 minutes to install in Colab, after which it would still take around an hour per epoch. So, using different datasets without a concrete idea of the problem consumes a lot of time.

The issue of having great accuracy during training but bad detection during prediction is usually an indication of overfitting. However, both other models did not have this problem with the same dataset. Even when predicting on one of the training images, nothing is detected most of the time. Possible future work would be to create an even more varied dataset, which as mentioned in section 6 is difficult for this use case. Of course it is also entirely possible that there is a simple mistake that was made.

* 1. Application
     1. Dataset creator

The dataset creator is an important part of the proof-of-concept application. Not much active time is spent on training and testing, the dataset creator requires the most interaction with the user. Though solutions exist for creating datasets by drawing bounding boxes, they are usually behind a paywall and do not always perform data augmentation. It is also more convenient for the user when everything is contained in a single application and thus there is no need to go to a third-party application. Furthermore, it makes the user experience more consistent between users and allows for more control over the quality of the datasets.

At the very start of development Tkinter was used instead of NiceGUI, but the dated and simplistic look of the GUI prompted a switch to NiceGUI and its contemporary look allowed by the web-based approach. The framework made it very easy to get started and yet it allows some pretty complex things to be achieved, as evidenced by the dataset creator. This complexity is supported by their use of the Vue, Quasar and Tailwind frameworks. However, this also leads into what might be NiceGUI’s biggest problem: incomplete and fragmented documentation. Some things are not covered in the NiceGUI documentation but instead in the Quasar or Tailwind documentation. It is then required to figure out how to apply this documentation to NiceGUI. Another positive point is NiceGUI’s modularity, which makes it easy and convenient to split a page into reusable and encapsulated elements represented by classes. Finally, the framework is not very established yet online, making it difficult to find people encountering the same issues. In conclusion, NiceGUI made it relatively convenient to make a modern looking, responsive GUI with some complex elements.

Care was taken to make the user experience of the dataset creator as smooth as possible with the tools at hand and within the allotted time. Quality-of-life features that users have come to expect of applications were added wherever fitting, though of course not all possible features have been exhausted. Error prevention and handling is applied as much as possible throughout the application. Saved files are checked for validity before being imported. Input fields perform validation and display error messages when necessary. Important results are accompanied by a notification toast on the top of the website.

The dataset creator ended up almost exactly as desired and is very useful, even being used during development to create datasets. The creator can be used to create datasets for models that support the COCO annotation format or the Ultralytics input data format, making it useful even outside the field of web element detection.

Possible future work would include the optimization of the canvas, since it is not uncommon for the canvas to lag slightly. The robustness of the dataset creator, especially when it comes to responsivity, could also be increased.

* + 1. Training
    2. Testing
    3. Settings
  1. API
  2. Primary flow

1. Conclusion
2. Reference list

|  |  |
| --- | --- |
| [1] | Selenium, „Getting started,” [Online]. Available: https://www.selenium.dev/documentation/webdriver/getting\_started/ [Accessed: 02/04/2024]. |
| [2] | Selenium, „Locators,” [Online]. Available: https://www.selenium.dev/documentation/webdriver/elements/locators/ [Accessed: 02/04/2024]. |
| [3] | J. Chen, M. Xie, Z. Xing, C. Chen, X. Xu, L. Zhu en G. Li, „Object detection for graphical user interface: old fashioned or deep learning or a combination?,” in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020. |
| [4] | A. Kumar, K. Morabia, W. Wang, K. Chang en A. Schwing, „CoVA: Context-aware Visual Attention for Webpage Information Extraction,” in *Proceedings of The Fifth Workshop on e-Commerce and NLP (ECNLP 5)*, 2022. |
| [5] | T. Gogar, O. Hubacek en J. Sedivy, „Deep Neural Networks for Web Page Information Extraction,” in *Artificial Intelligence Applications and Innovations*, Cham, 2016. |
| [6] | M. Altinbas en T. Serif, „GUI Element Detection from Mobile UI Images Using YOLOv5,” 2022, pp. 32-45. |
| [7] | M. Xie, „UIED,” [Online]. Available: https://github.com/MulongXie/UIED?tab=readme-ov-file [Accessed: 27/03/2024]. |
| [8] | T. Yeh, T.-H. Chang en R. C. Miller, „Sikuli: using GUI screenshots for search and automation,” in *Proceedings of the 22nd Annual ACM Symposium on User Interface Software and Technology*, New York, NY, USA, 2009. |
| [9] | J. C. Luna, „Top programming languages for data scientists in 2023,” [Online]. Available: https://www.datacamp.com/blog/top-programming-languages-for-data-scientists-in-2022 [Accessed: 02/04/2024]. |
| [10] | TIOBE, „TIOBE Index,” [Online]. Available: https://www.tiobe.com/tiobe-index/ [Accessed: 02/04/2024]. |
| [11] | NiceGUI, „NiceGUI,” [Online]. Available: https://nicegui.io/ [Accessed: 02/04/2024]. |
| [12] | M. Roseman, „TkDocs,” [Online]. Available: https://tkdocs.com/index.html [Accessed: 02/04/2024]. |
| [13] | Google, „Welcome To Colab,” [Online]. Available: https://colab.research.google.com/ [Accessed: 30/04/2024]. |
| [14] | T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár en C. L. Zitnick, „Microsoft COCO: Common Objects in Context,” in *Computer Vision – ECCV 2014*, Cham, 2014. |
| [15] | COCO, „Detection evaluation,” [Online]. Available: https://cocodataset.org/#detection-eval [Accessed: 13/03/2024]. |
| [16] | D. Shah, „Intersection over Union (IoU): Definition, Calculation, Code,” [Online]. Available: https://www.v7labs.com/blog/intersection-over-union-guide [Accessed: 13/03/2024]. |
| [17] | D. Li, „Calculate Computational Efficiency of Deep Learning Models with FLOPs and MACs,” [Online]. Available: https://www.kdnuggets.com/2023/06/calculate-computational-efficiency-deep-learning-models-flops-macs.html [Accessed: 13/03/2024]. |
| [18] | O. E. Olorunshola, A. K. Ademuwagun en C. Dyaji, „Comparative Study of Some Deep Learning Object Detection Algorithms: R-CNN, FAST RCNN, FASTER R-CNN, SSD, and YOLO,” *Nile Journal of Engineering & Applied Science,* vol. 1, p. 70–80, September 2023. |
| [19] | S. A. Sanchez, H. J. Romero en A. D. Morales, „A review: Comparison of performance metrics of pretrained models for object detection using the TensorFlow framework,” *IOP Conference Series: Materials Science and Engineering,* vol. 844, p. 012024, May 2020. |
| [20] | S. Srivastava, A. V. Divekar, C. Anilkumar, I. Naik, V. Kulkarni en V. Pattabiraman, „Comparative analysis of deep learning image detection algorithms,” *Journal of Big Data,* vol. 8, p. 66, 2021. |
| [21] | Z.-Q. Zhao, P. Zheng, S.-T. Xu en X. Wu, „Object Detection With Deep Learning: A Review,” *IEEE Transactions on Neural Networks and Learning Systems,* vol. PP, pp. 1-21, January 2019. |
| [22] | H. Zota en M. Dhande, „A Comparative Study of Widely Used Image Detection Algorithms,” *International Research Journal of Engineering and Technology,* vol. 7, p. 5183–5187, July 2020. |
| [23] | PyTorch, „Models and pre-trained weights: Object detection,” [Online]. Available: https://pytorch.org/vision/stable/models.html#object-detection [Accessed: 13/03/2024]. |
| [24] | Kaggle, „Models,” [Online]. Available: https://www.kaggle.com/models?tfhub-redirect=true&task=17074&fine-tunable=true [Accessed: 13/03/2024]. |
| [25] | G. Jocher en Laughing-q, „Models Supported by Ultralytics,” 12 November 2023. [Online]. |
| [26] | PapersWithCode, „Object Detection on COCO test-dev,” [Online]. Available: https://paperswithcode.com/sota/object-detection-on-coco [Accessed: 13/03/2024]. |
| [27] | H. Ouchra en A. Belangour, „Object Detection Approaches in Images: A Weighted Scoring Model based Comparative Study,” *International Journal of Advanced Computer Science and Applications,* vol. 12, 2021. |
| [28] | B. Zoph, E. D. Cubuk, G. Ghiasi, T.-Y. Lin, J. Shlens en Q. V. Le, „Learning Data Augmentation Strategies for Object Detection,” *CoRR,* vol. abs/1906.11172, 2019. |
| [29] | J. Nelson, „What is Image Preprocessing and Augmentation?,” 26 January 2020. [Online]. Available: https://blog.roboflow.com/why-preprocess-augment/ [Accessed: 20/03/2024]. |
| [30] | Z. Zong, G. Song en Y. Liu, „co\_dino\_5scale\_swin\_large\_16e\_o365tococo.py,” [Online]. Available: https://github.com/Sense-X/Co-DETR/blob/main/projects/configs/co\_dino/co\_dino\_5scale\_swin\_large\_16e\_o365tococo.py [Accessed: 30/04/2024]. |
| [31] | Z. Zong, G. Song en Y. Liu, „co\_dino\_5scale\_r50\_1x\_coco.py,” [Online]. Available: https://github.com/Sense-X/Co-DETR/blob/main/projects/configs/co\_dino/co\_dino\_5scale\_r50\_1x\_coco.py [Accessed: 30/04/2024]. |
| [32] | Z. Zong, G. Song en Y. Liu, „Co-DINO Swin-DETR checkpoint files,” [Online]. Available: https://drive.google.com/drive/folders/1r7bTh-DXkkQsCqkHYAHv4U-2hoEsLEoM?usp=drive\_link [Accessed: 30/04/2024]. |
| [33] | Z. Zong, G. Song en Y. Liu, „Co-DETR,” [Online]. Available: https://github.com/Sense-X/Co-DETR [Accessed: 13/03/2024]. |
| [34] | F. Schindler en R. Trappe, *NiceGUI: Web-based user interfaces with Python. The nice way.,* 2024. |
| [35] | Z. Ge, S. Liu, F. Wang, Z. Li en J. Sun, „YOLOX,” [Online]. Available: https://github.com/Megvii-BaseDetection/YOLOX [Accessed: 13/03/2024]. |
| [36] | C.-Y. Wang, A. Bochkovskiy en H.-Y. M. Liao, „yolov7,” [Online]. Available: https://github.com/WongKinYiu/yolov7 [Accessed: 13/03/2024]. |
| [37] | G. Jocher, A. Chaurasia, F. C. Akyon en Laughing-q, „YOLOv8,” 12 November 2023. [Online]. Available: https://docs.ultralytics.com/models/yolov8/ [Accessed: 13/03/2024]. |
| [38] | G. Jocher, B. Qaddoumi en Laughing-q, „YOLOv9: A Leap Forward in Object Detection Technology,” 26 February 2024. [Online]. Available: https://docs.ultralytics.com/models/yolov9/ [Accessed: 13/03/2024]. |
| [39] | G. Jocher, „Baidu's RT-DETR: A Vision Transformer-Based Real-Time Object Detector,” 12 November 2023. [Online]. Available: https://docs.ultralytics.com/models/rtdetr/ [Accessed: 13/03/2024]. |
| [40] | PyTorch, „Faster R-CNN,” [Online]. Available: https://pytorch.org/vision/main/models/faster\_rcnn.html [Accessed: 13/03/2024]. |
| [41] | Y. Wu, A. Kirillov, F. Massa, W.-Y. Lo en R. Girshick, *Detectron2,* 2019. |
| [42] | J. Dai, Y. Li, Y. Xiong en H. Qi, „R-FCN,” [Online]. Available: https://github.com/daijifeng001/R-FCN [Accessed: 13/03/2024]. |
| [43] | PyTorch, „SSD,” [Online]. Available: https://pytorch.org/vision/main/models/ssd.html [Accessed: 13/03/2024]. |
| [44] | PyTorch, „FCOS,” [Online]. Available: https://pytorch.org/vision/main/models/fcos.html [Accessed: 13/03/2024]. |
| [45] | PyTorch, „RetinaNet,” [Online]. Available: https://pytorch.org/vision/main/models/retinanet.html [Accessed: 13/03/2024]. |
| [46] | W. Wang, J. Dai, Z. Chen, Z. Huang, Z. Li, X. Zhu, X. Hu, T. Lu, L. Lu, H. Li en others, „InternImage,” [Online]. Available: https://github.com/opengvlab/internimage [Accessed: 13/03/2024]. |
| [47] | PapersWithCode, „Torchvision,” [Online]. Available: https://paperswithcode.com/lib/torchvision [Accessed: 13/03/2024]. |
| [48] | W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu en A. C. Berg, „SSD: Single Shot MultiBox Detector,” in *Computer Vision – ECCV 2016*, Cham, 2016. |
| [49] | PapersWithCode, „NAS-FCOS: Fast Neural Architecture Search for Object Detection,” [Online]. Available: https://paperswithcode.com/paper/nas-fcos-fast-neural-architecture-search-for [Accessed: 13/03/2024]. |

1. Attachment
   1. Comparison of object detection models

Table 2: Comparison of object detection model accuracy

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Model** | | **Accuracy** | | |
| **Name** | **Variant** | **Score** | **mAP (%)** | **Dataset** |
| YOLOX | s | 42.30 | 40.50 | COCO |
| m | 57.95 | 46.90 | COCO |
| l | 64.79 | 49.70 | COCO |
| x | 68.22 | 51.10 | COCO |
| YOLOv7 |  | 68.46 | 51.20 | COCO val |
| X | 72.62 | 52.90 | COCO val |
| W6 | 76.77 | 54.60 | COCO val |
| E6 | 79.95 | 55.90 | COCO val |
| D6 | 80.93 | 56.30 | COCO val |
| E6E | 82.15 | 56.80 | COCO val |
| YOLOv8 | n | 34.47 | 37.30 | COCO val |
| s | 53.06 | 44.90 | COCO val |
| m | 66.01 | 50.20 | COCO val |
| l | 72.62 | 52.90 | COCO val |
| x | 75.06 | 53.90 | COCO val |
| YOLOv9 | S | 57.70 | 46.80 | COCO val |
| M | 68.95 | 51.40 | COCO val |
| C | 72.86 | 53.00 | COCO val |
| **E** | **79.22** | **55.60** | **COCO val** |
| RT-DETR | L | 72.86 | 53.00 | COCO val |
| **X** | **77.26** | **54.80** | **COCO val** |
| Co-DETR | **Co-DINO** | **100.00** | **64.10** | **COCO val** |
| Co-Deformable-DETR | 86.31 | 58.50 | COCO val |
| Faster R-CNN | R50-FPN | 33.74 | 37.00 | COCO val |
| R50-FPN | 41.56 | 40.20 | COCO val |
| R101-FPN | 45.97 | 42.00 | COCO val |
| X101-FPN | 48.41 | 43.00 | COCO val |
| R-FCN |  | 20.29 | 31.50 | COCO val |
| SSD | 300 VGG16 | 0.00 | 23.20 | COCO test-dev |
| FCOS | R50-FPN | 40.59 | 39.80 | COCO test-dev |
| RetinaNet | R50-FPN | 32.27 | 36.40 | COCO minival |
| R50 | 37.90 | 38.70 | COCO val |
| R101 | 42.05 | 40.40 | COCO val |
| InternImage | T | 63.33 | 49.10 | COCO |
| S | 64.79 | 49.70 | COCO |
| B | 66.26 | 50.30 | COCO |
| L | 80.44 | 56.10 | COCO |
| XL | 80.68 | 56.20 | COCO |
| **Weight** |  | **0.60** |  |  |

Table 3: Comparison of object detection model speed

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Model** | | **Speed** | | |
| **Name** | **Variant** | **Score** | **FLOPs (B)** | **FPS** |
| YOLOX | s | 98.98 | 26.80 | 102.04 |
| m | 96.33 | 73.80 | 81.30 |
| l | 91.72 | 155.60 | 68.97 |
| x | 84.59 | 281.90 | 57.80 |
| YOLOv7 |  | 94.59 | 104.70 | 161.00 |
| X | 89.78 | 189.90 | 114.00 |
| W6 | 80.19 | 360.00 | 84.00 |
| E6 | 71.44 | 515.20 | 56.00 |
| D6 | 54.99 | 806.80 | 44.00 |
| E6E | 52.94 | 843.20 | 36.00 |
| YOLOv8 | n | 100.00 | 8.70 |  |
| s | 98.88 | 28.60 |  |
| m | 96.06 | 78.60 |  |
| l | 91.17 | 165.20 |  |
| x | 85.95 | 257.80 |  |
| YOLOv9 | S | 98.98 | 26.70 |  |
| M | 96.16 | 76.80 |  |
| C | 94.69 | 102.80 |  |
| **E** | **89.64** | **192.50** |  |
| RT-DETR | L | 94.29 | 110.00 |  |
| **X** | **87.29** | **234.00** |  |
| Co-DETR | **Co-DINO** | **84.76** | **279.00** |  |
| Co-Deformable-DETR | 51.99 | 860.00 |  |
| Faster R-CNN | R50-FPN | 75.28 | 447.00 | 10.20 |
| R50-FPN | 13.17 |  | 26.32 |
| R101-FPN | 8.85 |  | 19.61 |
| X101-FPN | 2.79 |  | 10.20 |
| R-FCN |  | 0.00 |  | 5.88 |
| SSD | 300 VGG16 | 34.24 |  | 59.00 |
| FCOS | R50-FPN | 89.80 | 189.60 |  |
| RetinaNet | R50-FPN | 70.77 | 527.00 | 24.39 |
| R50 | 11.93 |  | 24.39 |
| R101 | 11.93 |  | 24.39 |
| InternImage | T | 85.26 | 270.00 |  |
| S | 81.32 | 340.00 |  |
| B | 72.24 | 501.00 |  |
| L | 21.60 | 1399.00 |  |
| XL | 0.00 | 1782.00 |  |
| **Weight** |  | **0.30** |  |  |

Table 4: Comparison of object detection model speed

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Model** | | **Availability** | | |
| **Name** | **Variant** | **Score** | **How** | **Where** |
| YOLOX | s | 0.00 | download | [35] |
| m | 0.00 | download |
| l | 0.00 | download |
| x | 0.00 | download |
| YOLOv7 |  | 0.00 | download | [36] |
| X | 0.00 | download |
| W6 | 0.00 | download |
| E6 | 0.00 | download |
| D6 | 0.00 | download |
| E6E | 0.00 | download |
| YOLOv8 | n | 100.00 | lib, no data loading | [37] |
| s | 100.00 | lib, no data loading |
| m | 100.00 | lib, no data loading |
| l | 100.00 | lib, no data loading |
| x | 100.00 | lib, no data loading |
| YOLOv9 | S | 100.00 | lib, no data loading | [38] |
| M | 100.00 | lib, no data loading |
| C | 100.00 | lib, no data loading |
| **E** | **100.00** | **lib, no data loading** |
| RT-DETR | L | 100.00 | lib, no data loading | [39] |
| **X** | **100.00** | **lib, no data loading** |
| Co-DETR | **Co-DINO** | **0.00** | **download** | [33] |
| Co-Deformable-DETR | 0.00 | download |
| Faster R-CNN | R50-FPN | 50.00 | lib | [40] |
| R50-FPN | 50.00 | lib | [41] |
| R101-FPN | 50.00 | lib |
| X101-FPN | 50.00 | lib |
| R-FCN |  | 0.00 | download | [42] |
| SSD | 300 VGG16 | 50.00 | lib | [43] |
| FCOS | R50-FPN | 50.00 | lib | [44] |
| RetinaNet | R50-FPN | 50.00 | lib | [45] |
| R50 | 50.00 | lib | [41] |
| R101 | 50.00 | lib |
| InternImage | T | 0.00 | download | [46] |
| S | 0.00 | download |
| B | 0.00 | download |
| L | 0.00 | download |
| XL | 0.00 | download |
| **Weight** |  | **0.10** |  |  |

Table 5: Comparison of object detection model scores

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Model** | | **Scoring** | | **Data source** |
| **Name** | **Variant** | **Weighted score** | **Rank** |  |
| YOLOX | s | 55.07 | 27 | [35] |
| m | 63.67 | 21 |
| l | 66.39 | 17 |
| x | 66.31 | 18 |
| YOLOv7 |  | 69.45 | 14 | [36] |
| X | 70.50 | 12 |
| W6 | 70.12 | 13 |
| E6 | 69.40 | 15 |
| D6 | 65.06 | 20 |
| E6E | 65.17 | 19 |
| YOLOv8 | n | 60.68 | 25 | [37] |
| s | 71.50 | 11 |
| m | 78.43 | 9 |
| l | 80.92 | 6 |
| x | 80.82 | 7 |
| YOLOv9 | S | 74.32 | 10 | [38] |
| M | 80.22 | 8 |
| C | 82.12 | 4 |
| **E** | **84.42** | **2** |
| RT-DETR | L | 82.00 | 5 | [39] |
| **X** | **82.55** | **3** |
| Co-DETR | **Co-DINO** | **85.43** | **1** | [33] |
| Co-Deformable-DETR | 67.38 | 16 |
| Faster R-CNN | R50-FPN | 47.83 | 30 | [47] |
| R50-FPN | 33.89 | 34 | [41] |
| R101-FPN | 35.23 | 32 |
| X101-FPN | 34.88 | 33 |
| R-FCN |  | 12.18 | 38 | [42] |
| SSD | 300 VGG16 | 15.27 | 37 | [48] |
| FCOS | R50-FPN | 56.29 | 26 | [49] |
| RetinaNet | R50-FPN | 45.60 | 31 | [47] |
| R50 | 31.32 | 36 | [41] |
| R101 | 33.81 | 35 |
| InternImage | T | 63.57 | 22 | [46] |
| S | 63.27 | 23 |
| B | 61.43 | 24 |
| L | 54.74 | 28 |
| XL | 48.41 | 29 |
| **Weight** |  |  |  |  |

1. Don’t confuse FLOPs (Floating-Point Operations) with FLOPS (Floating-Point Operations per Second) [16]. [↑](#footnote-ref-1)