



POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

EXECUTIVE SUMMARY OF THE THESIS

Anomaly Detection in GUIs applied to websites

LAUREA MAGISTRALE IN COMPUTER SCIENCE ENGINEERING

Author: DAVIDE VOLTA

Advisor: PROF. FRATERNALI

Academic year: 2022-2023

1. Introduction

When working on a website or web app, a portion of the development time is spent checking that no regression happened after deploying an update; the amount of time "wasted" testing for regressions grows linearly with the product size, as more and more features are added. A regression may be categorized as either *functional* (i.e. something doesn't behave as it previously did) or *visual* (i.e. something doesn't look as it previously did). This work focuses on the latter type and proposes a deep-learning-based approach that can automatically detect a good portion of them, which could potentially allow QA testers to focus on subtler and/or behavioral errors, where a human might be better suited.

The specific website that I worked on was an ABB product called "*ABB Ability Energy Manager*". ABB Ltd. is a Swedish-Swiss multinational corporation whose main industry is electrical equipment such as breakers; Energy Manager is a web-based tool that allows users to monitor the status of their electrical systems, including charts and real-time data. We focused on its Dashboard page which is the most dynamic and complex in terms of layout: it is made of widgets that a user can add, remove, and shuffle freely.

The method presented in this thesis consists

of automatically segmenting the webpage into its most significant elements and then training an anomaly detection model (i.e. *PaDiM*) on each of them. The combined result of running these models shows considerably better anomaly detection performance when compared to just training the same model on the full webpage.

2. Method

PaDiM [2], which stands for Patch Distribution Modeling, is an innovative approach for anomaly detection and localization in images, with a particular focus on applications in industrial inspections. The primary objective of PaDiM is to automatically identify and pinpoint anomalies or unexpected patterns in images, enabling constant quality control without the need for manual intervention. What sets PaDiM apart from other methods is its use of a pre-trained Convolutional Neural Network (CNN) for feature extraction. It leverages this CNN to describe each position in an image patch as a multivariate Gaussian distribution. Additionally, PaDiM takes into account the correlations between different semantic levels of the CNN, enhancing its ability to detect and localize anomalies effectively. PaDiM has proven to be highly efficient and effective in anomaly detection and localization tasks, surpassing existing state-of-

the-art methods on datasets like MVTec AD and ShanghaiTech Campus. Importantly, PaDiM's efficiency is notable, as it maintains low time and space complexity during testing, making it well-suited for practical industrial applications. As for the actual anomaly detection model training, we relied on *anomalib* version 0.3.7 (at the time of writing, the latest available version is 0.4.0) [1] (<https://github.com/openvinotoolkit/anomalib>). From the README in their repository:

Anomalib is a deep learning library that aims to collect state-of-the-art anomaly detection algorithms for benchmarking on both public and private datasets. Anomalib provides several ready-to-use implementations of anomaly detection algorithms described in the recent literature, as well as a set of tools that facilitate the development and implementation of custom models. The library has a strong focus on image-based anomaly detection, where the goal of the algorithm is to identify anomalous images, or anomalous pixel regions within images in a dataset.

And this is the configuration file we used:

```
dataset:
  name: screenshots
  format: folder
  path: dataset/<hostname>/<page
↳ name>/<component ID>/screenshots
  normal_dir: normal # name of the
↳ folder containing normal images.
  abnormal_dir: abnormal # name of the
↳ folder containing abnormal images.
  normal_test_dir: null # name of the
↳ folder containing normal test
↳ images.
  mask: difference_masks
  task: segmentation # classification or
↳ segmentation
  extensions: null
  split_ratio: 0.2 # ratio of the normal
↳ images that will be used to create a
↳ test split
  image_size: 256
  train_batch_size: 4
```

```
test_batch_size: 4
num_workers: 8
transform_config:
  train: null
  val: null
create_validation_set: true
tiling:
  apply: false
  tile_size: null
  stride: null
  remove_border_count: 0
  use_random_tiling: False
  random_tile_count: 16
model:
  name: padim
  backbone: resnet18
  pre_trained: true
  layers:
    - layer1
    - layer2
    - layer3
  normalization_method: min_max #
↳ options: [none, min_max, cdf]
metrics:
  image:
    - F1Score
    - AUROC
  pixel:
    - F1Score
    - AUROC
  threshold:
    image_default: 3
    pixel_default: 3
    adaptive: true
visualization:
  show_images: False # show images on
↳ the screen
  save_images: True # save images to the
↳ file system
  log_images: True # log images to the
↳ available loggers (if any)
  image_save_path: null # path to which
↳ images will be saved
  mode: full # options: ["full",
↳ "simple"]
project:
  seed: 42
  path: ./results
```

```

logging:
  logger: [] # options: [tensorboard,
↪ wandb, csv] or combinations.
  log_graph: true # Logs the model graph
↪ to respective logger.

optimization:
  export_mode: null #options: onnx,
↪ openvino

# PL Trainer Args. Don't add extra
↪ parameter here.
trainer:
  accelerator: auto # <"cpu", "gpu",
↪ "tpu", "ipu", "hpu", "auto">
  accumulate_grad_batches: 1
  amp_backend: native
  auto_lr_find: false
  auto_scale_batch_size: false
  auto_select_gpus: false
  benchmark: false
  check_val_every_n_epoch: 1 # Don't
↪ validate before extracting
↪ features.
  default_root_dir: null
  detect_anomaly: false
  deterministic: false
  devices: 1
  enable_checkpointing: true
  enable_model_summary: true
  enable_progress_bar: true
  fast_dev_run: false
  gpus: null # Set automatically
  gradient_clip_val: 0
  ipus: null
  limit_predict_batches: 1.0
  limit_test_batches: 1.0
  limit_train_batches: 1.0
  limit_val_batches: 1.0
  log_every_n_steps: 50
  max_epochs: 1
  max_steps: -1
  max_time: null
  min_epochs: null
  min_steps: null
  move_metrics_to_cpu: false
  multiple_trainloader_mode:
↪ max_size_cycle
  num_nodes: 1
  num_processes: null

```

```

num_sanity_val_steps: 0
overfit_batches: 0.0
plugins: null
precision: 32
profiler: null
reload_data_loaders_every_n_epochs: 0
replace_sampler_ddp: true
sync_batchnorm: false
tpu_cores: null
track_grad_norm: -1
val_check_interval: 1.0 # Don't
↪ validate before extracting
↪ features.

```

We divided the whole process into 3 different steps:

1. Acquire images (*acquire*)
2. Train models (*train*)
3. Check the web page for anomalies (*verify*)

2.1. Acquire images

The *acquire* step consists of collecting screenshots of the web page and the individual components, together with binary masks that highlight the anomalous regions (masks and related problems are covered in detail in *Ground truth generation*). Normal images are called *normal* and images where an error has been artificially generated are called *abnormal*.

The following pseudo-code illustrates how the dataset generation works:

Algorithm 1 Dataset generation

```

1: while dataset target size not reached do
2:   Load target webpage
3:   Take a screenshot
4:   Generate a blank full-page anomaly mask

5:   Identify components
6:   for component in components do
7:     Take a screenshot
8:     Perform random alteration
9:     Generate a difference mask
10:    if pixelDifference > threshold then
11:      Binarize the difference mask
12:      Store the normal screenshot, the altered one and the binary anomaly mask
13:      Paste the binary mask on the existing full-page mask
14:    else
15:      Discard component
16:    end if
17:  end for
18:  Take a post-alteration full-page screenshot
19:  Store the pre-alteration full-page screenshot, the altered one and the resulting composite binary anomaly mask
20: end while

```

Figure 1 shows the generated directory structure. Simply put, a directory is generated for each website, which will contain directories for all the analyzed pages. Every page directory is made of a full-page directory (which holds screenshots and masks for the complete page) and several other directories for the detected components.

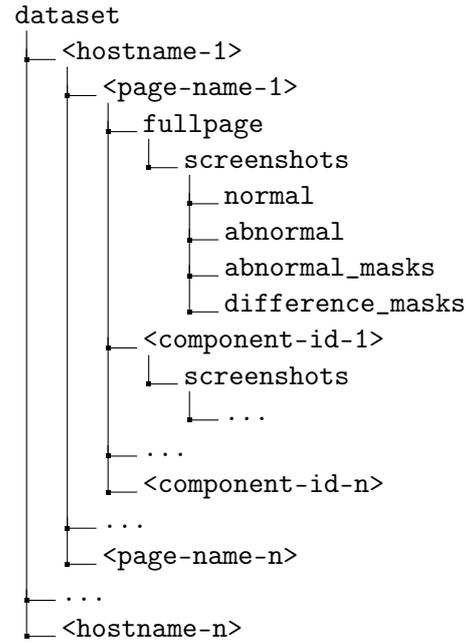


Figure 1: Dataset directory structure

All of the processing is performed at a fixed resolution of 1920x2160: this allowed us to focus on the issue at hand since elements would be easily identifiable by their XY coordinates, which would not change between runs. Potential improvements are discussed in *Conclusions*)

2.1.1 Ground Truth generation

One of the requirements that we decided on pretty early on was that the whole system needed to be as autonomous and flexible as possible, which meant that manually taking screenshots and drawing masks was out of the question.

In addition to the set of regular images, the training process also requires a set of anomalous images paired with anomaly masks, so that image-wise and pixel-wise performance metrics can be calculated.

Since the anomaly detection models we can use are all unsupervised, the results are not directly influenced by anomalous images and the related anomaly masks, but metrics are. AUROC and F1 scores can get pretty low in 2 cases:

1. Anomalous images are not actually anomalous (see Figure 2)
2. The anomaly mask is not accurate enough (see Figure 3)

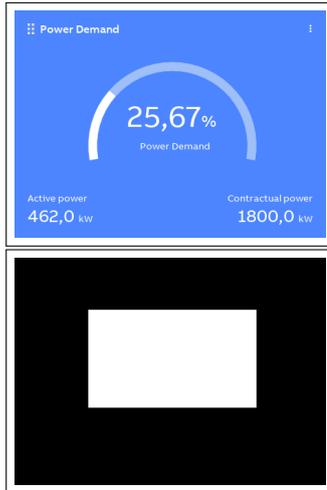


Figure 2: Altered component and the generated anomaly mask (white regions are considered anomalous). This is a case where the result is not anomalous at all. The changed element was probably fully occluded by other elements.

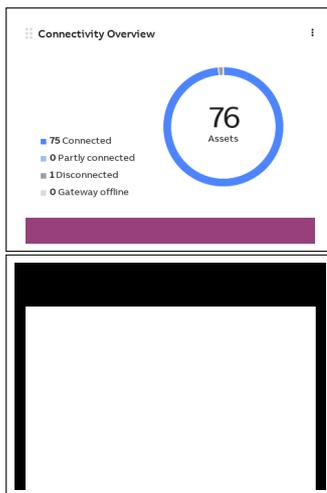


Figure 3: Altered component and the generated anomaly mask (white regions are considered anomalous). In this case, the mask shows an anomalous region bigger than what we can see from the image. The changed element was probably partially occluded by others.

The anomalous images are automatically generated by simply picking an HTML element in the component and changing some of its properties (e.g. *background-color*).

As for anomaly masks, our first approach was to simply draw a rectangle where the changed element was, but this presented a big flaw: it did not work where there was occlusion (i.e.: HTML elements overlapping), which meant that a lot of

the collected "anomalous" images fell into one of the two cases presented above (again, see Figures 2 and 3). We therefore revised the anomaly generation process and based it on pixel-wise difference: any difference greater than 0 results in a white pixel. An example of such a mask is shown in Figure 4.

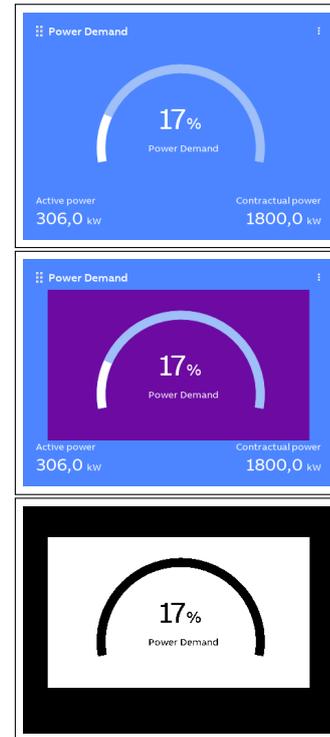


Figure 4: Normal screenshot, altered screenshot, and the resulting difference mask. As you can see, this mask generation method is more accurate and only highlights visible differences (in this case, the bar and the text did not change so they are ignored).

These difference masks solve problem number 2. Also, we avoid storing abnormal screenshots that have a low difference value (< 0.025), measured as:

$$\frac{\# \text{ of different pixels}}{\# \text{ of pixels}}$$

This solves problem number 1.

2.2. Train models

During the training step, the program trains a model for each component found in the dataset. Some validation samples (*screenshots*) plus a summary .csv file are also generated. The resulting directory structure is illustrated in Figure 5.

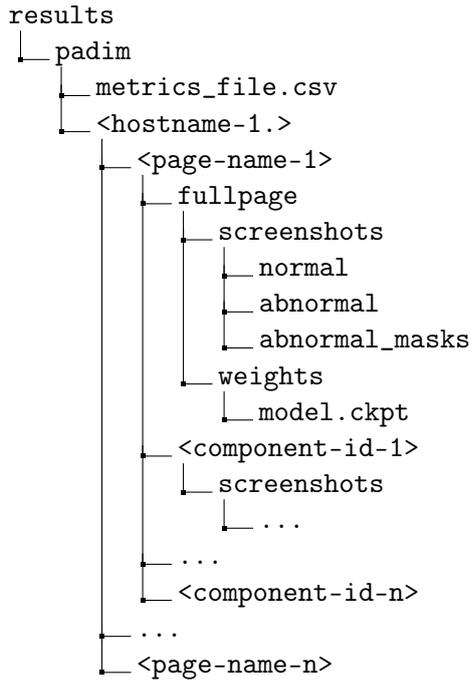


Figure 5: Training output

3. Evaluation

3.1. Data set

The dataset we used to conduct the evaluation contained 827 screenshots for each of the 36 significant components identified, plus another 827 for the whole page. Every screenshot also has a corresponding generated anomalous version, plus a binary anomaly mask. The directory structure is presented in Figure 1).

Our program automatically split our dataset into a training portion (80%) and an evaluation one (20%).

3.2. Results

The results are presented in the following tables (p is short for *pixel*, while i is short for *image*). The first row refers to the whole page (hence the ID *fullpage*), while the others refer to the identified components (whose ID is generated by concatenating their X and Y coordinates in this case).

3.2.1 Training

ID	pAUROC	pF1	iAUROC	iF1
fullpage	0.77	0.32	1.0	1.0
30.20	0.87	0.93	1.0	1.0
1642.20	0.94	0.72	0.87	0.91
17.152.040	1.0	1.0	1.0	1.0
20.83	0.69	0.68	0.86	0.91
150.84	0.71	0.74	0.88	0.91
280.84	0.74	0.73	0.88	0.91
392.84	0.72	0.7	0.86	0.91
507.84	0.7	0.68	0.85	0.91
615.84	0.75	0.74	0.87	0.91
764.84	0.71	0.69	0.85	0.91
202.36	0.98	0.97	1.0	1.0
232.29	0.95	0.91	1.0	1.0
452.040	1.0	1.0	1.0	1.0
3.092.040	1.0	1.0	1.0	1.0
4.262.040	1.0	1.0	1.0	1.0
5.462.040	1.0	1.0	1.0	1.0
7.572.040	1.0	1.0	1.0	1.0
8.472.040	1.0	1.0	1.0	1.0
9.852.040	1.0	1.0	1.0	1.0
11.662.040	1.0	1.0	1.0	1.0
493.39	0.96	0.76	1.0	1.0
644.39	0.95	0.79	1.0	1.0
744.41	0.97	0.98	1.0	1.0
18.052.070	1.0	1.0	1.0	1.0
1.697.160	0.72	0.76	0.88	0.91
1.662.163	0.69	0.77	0.91	0.91
48.217	0.97	0.78	1.0	1.0
35.264	0.65	0.32	0.62	0.91
497.264	0.8	0.52	0.77	0.91
1.420.948	0.82	0.54	0.73	0.91
958.264	0.87	0.49	0.98	0.98
1.420.264	0.89	0.46	0.98	0.98
35.606	0.73	0.3	0.65	0.91
958.948	0.89	0.46	0.94	0.94
351.290	0.8	0.51	0.75	0.91
4.971.290	0.87	0.62	0.8	0.91

3.2.2 Evaluation

ID	pAUROC	pF1	iAUROC	iF1
fullpage	0.77	0.34	1.0	1.0
30.20	0.87	0.93	1.0	1.0
1642.20	0.93	0.71	0.85	0.91
17.152.040	1.0	1.0	1.0	1.0
20.83	0.75	0.75	0.89	0.91
150.84	0.73	0.76	0.89	0.91
280.84	0.73	0.71	0.87	0.91
392.84	0.7	0.68	0.87	0.91
507.84	0.7	0.68	0.85	0.91
615.84	0.7	0.69	0.85	0.91
764.84	0.69	0.67	0.86	0.91
202.36	0.98	0.97	1.0	1.0
232.29	0.95	0.91	1.0	0.99
452.040	1.0	1.0	1.0	1.0
3.092.040	1.0	1.0	1.0	1.0
4.262.040	1.0	1.0	1.0	1.0
5.462.040	1.0	1.0	1.0	1.0
7.572.040	1.0	1.0	1.0	1.0
8.472.040	1.0	1.0	1.0	1.0
9.852.040	1.0	1.0	1.0	1.0
11.662.040	1.0	1.0	1.0	1.0
493.39	0.96	0.75	1.0	1.0
644.39	0.95	0.78	1.0	1.0
744.41	0.96	0.98	1.0	1.0
18.052.070	1.0	1.0	1.0	1.0
1.697.160	0.72	0.75	0.88	0.91
1.662.163	0.65	0.77	0.89	0.91
48.217	0.97	0.78	1.0	1.0
35.264	0.66	0.31	0.61	0.91
497.264	0.82	0.6	0.77	0.91
1.420.948	0.84	0.63	0.78	0.91
958.264	0.88	0.45	0.99	0.97
1.420.264	0.82	0.43	0.98	0.97
35.606	0.73	0.32	0.6	0.91
958.948	0.77	0.37	0.96	0.94
351.290	0.82	0.55	0.77	0.91
4.971.290	0.91	0.66	0.79	0.91

3.3. Results Discussion

We will first focus on *Image AUROC* and *Image F1*. These two values tell us how good our models are at classifying the image as either anomalous or non-anomalous. If our goal is to simply classify the page as "contains errors" or "does not contain errors", then the numbers tell us that there is no significant improvement obtained from working on individual components compared to the full web page: we already have a 1 in both metrics.

Things look different though when you observe *Pixel AUROC* and *Pixel F1* scores. These measure the ability to locate anomalies in the image. As you can see, the *fullpage* scored 0.77 and 0.32 respectively, but individual components scored mostly much better values, with some exceptions. This means that our method has a higher chance to successfully highlight anomalies in a web page; this is due to having separate models for each component, which reduces variability in the input and allows each model to be much more precise.

Let's examine one of the components that didn't do too well in our evaluation: *35.606*.



Figure 6: Sample data for component 35.606.

This component is made by some drop-down menus and, more prominently, a histogram, which generates a lot of variability in the output, thus requiring the model to see more data when compared to other simpler and more static components. We therefore reasonably expect performance to improve as the dataset size increases. Also, we need to consider that the Energy Manager platform we ran our training against was experiencing some issues that caused long loading times and sometimes graphical bugs; while we put in place some mitigation measures to avoid capturing data that is clearly wrong, some of the screenshots still present issues that most

definitely impact the accuracy of the result. The training process would ideally need to see only correct and valid images, but this is sometimes not technically feasible when using automation. A manual screening would certainly help, but we could not afford to do it for our research effort. At the same time, we pondered over this issue and thought that, if the test target behaves abnormally during training even with mitigation measures put in place, then it would be unreasonable to expect our system to work and a stable condition would need to be achieved before performing any kind of training.

4. Conclusions

In conclusion, this method, which builds upon the foundation of Anomalib and PaDiM, has demonstrated its effectiveness in anomaly detection. It leverages the strengths of these existing techniques while addressing specific challenges and making what we believe are notable contributions to the field. However, as with any evolving technology, certain areas warrant further attention in future research endeavors.

Firstly, one crucial area for improvement lies in handling variable window sizes. The whole training is run at a specific resolution and using the resulting models on elements that differ in size might result in inaccuracies. Also, we rely on fixed locations to match an element to the corresponding model, which means that the test resolution needs to be the same as the training one. Developing mechanisms to dynamically adapt to different window sizes or effectively processing images with varying resolutions would enhance the method's adaptability and applicability across diverse scenarios.

Secondly, the method can benefit from ongoing efforts to dynamically match elements in test images to the learned models. As we just said, our current method of matching an element to its model via its page coordinates is very unreliable: any change in resolution and/or position of the detected elements would break it immediately. We believe this could be implemented by something rather rudimentary like cosine similarity or, maybe, another deep learning model dedicated to computing a similarity score between an image and a given set of images. Pre-existing tools such as Sikuli and JAutomate could also help.

Lastly, leveraging provisional data obtained during training to filter out invalid content in test images presents a valuable avenue for future research. The data acquisition step currently runs based on the assumption that no error will occur and all screenshots taken are considered valid. For example, we could think about acquiring 100-200 images per element, perform training on this initial dataset, and then use the learned models to acquire more screenshots while filtering errors that may occur during the acquisition process, so that the dataset is as clean as possible.

In summary, while we believe we have achieved interesting results, addressing these challenges and exploring these avenues for future work could result in a more robust method that might see some adoption from people who want to monitor their website's appearance.

References

- [1] Samet Akcay, Dick Ameln, Ashwin Vaidya, Barath Lakshmanan, Nilesh Ahuja, and Utku Genc. Anomalib: A deep learning library for anomaly detection, 2022.
- [2] Thomas Defard, Aleksandr Setkov, Angelique Loesch, and Romaric Audigier. Padim: a patch distribution modeling framework for anomaly detection and localization, 2020.