Abstract

The process of developing the user interface (UI) of an application tends to be time-consuming, as it usually relies on the sequential input of several people, requiring numerous back and forth iterations until a final version is obtained. A common first step of this process is a hand-made sketch on a whiteboard. Having an automatic tool that, during the design brainstorming phase, could instantly convert UI sketches into the respective code for that UI and generate it, would speed up the whole process. It would allow teams to iterate through different UI designs significantly faster and, consequently, enable the agile experimentation and end-visualisation of different prototype designs, increasing productivity. Current solutions for this problem tend to rely on digital sketches rather than hand-made ones. However, the abstract nature of UI sketches and their diversity suggest the use of machine learning techniques to tackle this problem. Our approach to this task was to split it into two different parts: first detect the position and class of each element in the drawing using a deep neural network, followed by a program generation algorithm to output the final code to be compiled. The best-performing setup achieves 87% in terms of mean average precision (mAP). This work is a first step towards building a deep learning architecture that can correctly interpret and infer human sketches, with the final goal of making the prototyping phase of a UI design more agile and efficient.

Keywords: Hand-Drawn Sketch, UI, Computer Vision, Deep Learning, Program Generation

1. Introduction

Conveying information through drawings is common to a wide variety of situations, in particular in engineering. However, despite some specific solutions, there is a gap between generic human drawings and their interpretation by a computer. This is the case during the idealisation phase of a user interface (UI): the process of developing the UI of an application tends to be long and tedious, usually relying on the sequential input of several people (designers, front-end developers, project managers, etc.) leading to numerous back and forth iterations, until a final version is obtained.

The creative process usually starts with a draft on a whiteboard, followed by digitisation to a graphics editor developing program, and finally being converted into a hand-coded programming language. Even implementing the UI code is a time-consuming and often repetitive task, which prevents developers from focusing on the actual core functionalities of the application and on creating something that makes it unique.

Having an automatic tool that, during the design brainstorming phase, could convert the produced sketches into code for that UI and generate it, would allow the agile experimentation and end-visualisation of several competing designs on the go, increasing the team’s productivity.

The current UI design prototyping solutions rely on digital sketches, rather than hand-made ones. However, they do not tackle the situation in question: the ability to easily and rapidly generate the UI code based on an initial hand-drawn sketch. The main difficulty lies in the need to make the connection between the pixel-based abstract hand-drawn sketches and the reasoning behind the accurate user interface code. In the past, there were attempts to solve this problem by extracting hand-crafted features from sketch images followed by feeding them to a classifier. However, these hand-crafted features are based on heuristics and thus could not fully adapt to the abstract nature and high variability of sketches.

We propose a novel approach to this problem: use deep machine learning techniques to generate UI code automatically from sketches, by creating a model that is able to detect and recognise UI elements and their relative positions, so that from this representation it is possible to generate the code for the corresponding UI.
2. State-of-the-art

Object Detection: Frameworks based on region proposals are composed of several independent stages: region proposal generation, feature extraction using a convolutional neural network (CNN), classification, and bounding box regression. As a result, this sort of architectures tends to be complex, thus hard to optimise. In contrast, one-step frameworks, based on global regression, map directly from image pixels to bounding box coordinates and class probabilities, resulting in a high reduction in computing time, while maintaining good accuracy.

This is the approach followed by the You Only Look Once (YOLO) algorithm [1], which directly predicts bounding boxes and class probabilities in a single network evaluation. It divides the input image into a $S \times S$ grid, and, for each cell of the grid, predicts $B$ bounding boxes along with a confidence score as well as the class probabilities.

A new version, YOLOv2 [2], improved upon the previous version both in terms of accuracy as well as speed. The main innovations are the following: batch normalisation; pre-defined anchor boxes; fine-grained features - which, to counter the problem of detecting small objects, concatenates the output of the current layer with the output of a previous one; multi-scale training - since with the use of anchor boxes, the network can take images of different sizes (there are no fully-connected layers at the end); architecture - using Darknet-19.

More recently, further improvements were presented for a new network, YOLOv3 [3], namely in terms of class prediction (using multi-label classification), prediction across scales, and a more powerful network architecture.

One of the main advantages of the YOLO networks is the availability of pre-trained weights on large datasets which are publicly available.

Hand-drawn sketches feature extraction: In Sketch-a-Net [4], the authors developed a neural network specialised for sketches, rather than using a network that was designed for natural photos. It uses large filters (of size 15x15) in the first layer of the CNN, with the argument that larger filters are able to capture more structured context rather than texture information; it removes the commonly used local response normalisation (LRN, presented in [5]), which is inspired by the form of lateral inhibition found in real neurons, arguing that this tool provides brightness normalisation which is not a concern in hand-drawn sketches; it also uses larger pooling size (3x3 with stride 2), which leads to overlapping pooling areas. With these features, the authors argue that the network is more suitable for hand-drawn sketches, in contrast with the more traditional approaches.

3. Dataset

3.1. UI elements

A web platform was the basis for the experiments to be conducted due to the fact that it allows easier implementation and experimentation. It is a very mature and stabilised platform whilst being open source, thus making it the ideal platform for a proof of concept.

The UI elements chosen were based on the most commonly used elements to create webpages and, more specifically, form pages.

In order to achieve a final representation for each element, an iterative process was necessary. Initially, there was a vast range of possible representations for each element. However, since increasing this range of possible options increases the difficulty of the network to correctly identify the element (as it creates more diversity, thus more complexity), at each iteration, the representation of each element was simplified and differentiated from the rest, in order to improve the accuracy of the network. Then, those changes were tested in order to understand their impact and the representation was improved based on those results. Despite this simplification, there was a concern to retain an universal representation, so that it is evident which element each representation stands for. After all the iterations, the result was a fixed and distinguishable, but intuitive representation for each element, as depicted in Figure 1.

![UI elements](image)

Figure 1: UI hand-drawn representation.

3.2. Dataset generation

Computer-generated dataset: In order to generate a large labelled dataset, simulated hand-drawn UI sketches were generated through a python script. This script allowed to generate as many labelled examples as necessary, with the benefit that if it was necessary to adjust some detail in the examples generated, it could promptly be altered. Furthermore, it is easily scalable: a computer program can effortlessly generate a large number of different UI designs and the labelling is promptly available. On the other hand, it is not trivial to mimic human drawings in a computer, due to their inherent abstract nature and variability, which is relatively hard for a computer to simulate.

Beyond the randomness of the elements size and their position on the page, some details were taken...
into consideration at the time of the definition of the webpage in order to increase the variability, while making the images as close as possible to human drawing. These details include the thickness and curvature of the border of each element as well as of the page itself; small random rotations on each element; the font and the size used to write text in the button element; the density and gradient of the filled box in the input text element; the shape, aspect ratio, and number of the circles in the radio element.

Figure 2 presents an example of the output of the single element generation algorithm, as well as an example output for the multiple-element webpage generation algorithm, along with the respective ground truth bounding boxes.

![Figure 2: Computer-generated webpages.](image)

**Human-generated dataset:** In order to create a dataset with human-generated drawings, dataset generation sessions were organised. Participants were asked to draw webpages using the elements provided. This solution aimed to tackle the challenge of having varied and realistic/representative examples in the dataset, as several people were included in the generation sessions and the sketches were performed by humans. In total, throughout the several sessions, we manage to generate and label 1205 examples of a single element webpage and 521 examples of multiple elements in the webpage, with the help of almost 50 participants.

In order to label the images for the object detection task, the labelImg tool, available at github.com/tzutalin/labelImg was used, since it is mandatory to define the bounding box of each element in the page as well as its class. With this tool, a .xml file is created according to the YOLO specific format.

### 3.3. Data pre-processing

Before feeding the data to the network, a sequence of pre-processing steps were applied. The images were rescaled to a standard size of 225x225px for image classification and 500x500px for object detection.

Considering the two different approaches for data generation, there is a relevant discrepancy when it comes to the background colour and light shades/saturation: the computer-generated images have a pure white background with the lines drawn being almost black, whereas the images taken with a camera have a huge variability when it comes to background colour, consistency of the lines drawn, gradient, saturation, contrast, etc. In order to reduce this distance between the two-generation approaches, a binarization method [6] was included.

### 4. Implementation

#### 4.1. Image Classification

**Dataset for image classification:** The dataset created was comprised with images from both generation alternatives, computer and human.

In relation to the computer-generated examples, absolute positioning was used as it allows more freedom when positioning the object in the page, and there is not the problem of overlapping objects.

In order to understand how it would affect the performance of the network, hand-drawn examples were also introduced in the training set. The objective was to include more realistic examples, rather than only computer-generated ones and, consequently, increase the variety of the dataset whilst making it more realistic. Data augmentation techniques were used to increase the number of hand-drawn examples, namely by applying skew, distortion, and rotation to the original images, as demonstrated in Figure 3. This was performed using the tool which provides a broad set of possible augmentation techniques for images, available at github.com/mdbloice/Augmentor.

![Figure 3: Augmented hand drawn examples.](image)

In total, there were 10000 computer-generated examples and 1205 hand-drawn examples. The computer-generated ones were only used for training, whilst the hand-drawn were split between train, validation, and test sets. The proportions chosen were 60% - 20% - 20%, yielding 720 - 242 - 243 examples, respectively. As mentioned, the 720 hand-drawn examples for the training set were augmented to obtain a final number of 5000 examples.
**Network architecture and training:** For the task of image classification, a network was constructed based on the state-of-the-art for sketch recognition, Sketch-a-Net [4].

In order to avoid overfitting, it was used the early stopping method to avoid overfitting or underfitting of the network to the training set. This was achieved by monitoring the model’s performance, namely, the loss value on the validation set and the training stopped when a minimum value was found, which was controlled using a patience hyperparameter, which corresponds to the number of epochs with no improvement after which the training will stop.

### 4.2. Object Detection

**Dataset for object detection:** The challenges presented to this task are similar to those described for image classification, but slightly exacerbated since localising and classifying several objects is inherently harder. The dataset was composed of both hand-drawn images as well as computer-generated images, following the same philosophy.

In the case of computer-generated examples, the main difference was the fact that more than one object had to be positioned in the page and, consequently, relative positioning was used, as it allows to more easily create reasonable webpages while guaranteeing non-overlapping items. Using this method, 10 000 images were generated.

The total number of hand-drawn webpages was 521. Since this number is relatively small to train a large neural network, some approaches were used to expand it. First of all, some outliers were manually removed, for two main reasons: first, there were pages that made no sense considering the task in hand which was to design a legitimate webpage; secondly, some of the uploaded images did not have enough quality to be processed, as either the lighting conditions were unacceptable, or the resolution was too low. This led to the removal of 24 images bringing the size of the dataset down to 497 images. Nevertheless, the images were carefully chosen in order to ensure a balance between sufficient variability and realistic examples.

One approach to increase the number of examples was to include images of webpages with a single element, which had already been drawn for the task of image classification and thus were readily available. Consequently, 200 more images were appended to the dataset, 50 per class, making it a total of 697 examples.

From this group, 20% of the dataset (140 images) were randomly chosen to be used as test set. The remaining 557 images were used for training.

Data augmentation was used to increase the amount and diversity of data available for training. However, besides augmentation (using a combination of scaling, rotation, shearing), it was also necessary to adjust the bounding box of each element, according to the transformation. Using this method, it was possible to produce nine augmented images per each original image, thus increasing the dataset size by a factor of 10.

Finally, the total number of images to be used for training and validation was 5570 hand-drawn images and 10 000 computer-generated images.

**Network architecture and training:** We opted for the YOLO [2] algorithm, since it is the method that arguably presents the best results both in accuracy and in computational speed. Concerning the several versions of YOLO, we opted for the YOLOv2 since it has some significant improvements when compared to the initial version. The YOLOv3 version [3], which achieves even better results, has some upgrades compared to the YOLOv2, however, we came to the conclusion that the advantages of YOLOv3 are not worth if we take into account the added complexity, and consequently, the YOLOv2 version was chosen, as it was the one that presented the best balance between accuracy, computational time, and complexity.

We decided to use darkflow (https://github.com/thtrieu/darkflow), which is an implementation of the darknet into TensorFlow, which allows performing the same operations such as configure the network, load pre-trained weights, train and test the network, and use tensorboard.

In order to train the network, pre-trained weights from the original paper were used (available online at https://pjreddie.com/darknet/yolov2/). Despite these weights having been previously optimised for natural images, they are still advantageous as this leads to a significantly faster and more accurate training process, as well as a higher probability of convergence.
Results evaluation: Mean average precision (mAP) is the standard metric to evaluate the performance of object detection algorithms [7]. It is computed based on the precision and recall values. An object detection is only considered to be correct if the intersection over union (IOU) is greater than 0.5. Note that in the case that multiple objects with an IOU \( \geq 0.5 \) are detected, only one will count as a true positive, whereas the rest will be considered false positives.

4.3. Program Generation
The final step in the pipeline is to use the extracted information from the previous step to generate the code for webpages.

The network outputs the following information: class, confidence score, and the bounding boxes coordinates (left, top, right, bottom). With this information, the element to be used is selected and located on the webpage using absolute positioning. It was decided to use absolute positioning since it is simpler to set the element in the webpage, only requiring the coordinates.

In order to make the end visualisation more appealing, a particular CSS style was chosen for each element, in contrast with the more sketchy style used for the dataset examples. This style was based on one of the styles used by OutSystems.

In figure 5 is presented the complete pipeline.

Figure 5: Complete pipeline used for testing.

5. Results
5.1. Image Classification
The results presented in this Section were obtained on a laptop with the following characteristics: CPU: Intel Core i7-7820HQ @ 2.90GHz; RAM: 16GB @ 2400MHz; GPU: NVIDIA Quadro M1200 (4GB dedicated memory).

Table 1 presents the results obtained using an architecture based on Sketch-a-Net [4]. In order to understand the effect of binarizing the images before feeding them into the network, we performed two different experiments: one where all the human-drawn images were binarized and another where the human-drawn images were used without binarization. The training ratio represents the proportion between computer-generated and hand-drawn images in the training set, respectively.

Concerning the effect of binarization, we can conclude that comparing the (ii) and (iv) cases with (v) and (vi), respectively, the results are very close to each other (only a difference of 1%) and on one case in favour of binarization whereas on the other in favour of non-binarization. Note that when the training set is 50/50, binarization leads better results, whereas when the training set is composed solely by human-drawn images it achieves better results with no binarization.

When the dataset is only composed of computer-generated images (cases (i) and (v)), there is a tendency to overfit to the training data (notice the high accuracy values in training and low in the test set), since the computer-generated images lack variability and diversity; consequently, it is inviting the network to overfit that set of images. However, the results are significantly better if the hand-drawn images are binarized (case (i)). This supports the conclusion that, in the case of need to use computer-generated images to increase the size of the training set, binarizing the hand-drawn images is helpful to achieve better performance since they will be closer to the computer-generated images.

There is a noticeable reduction of overfitting as the proportion of computer-generated/hand-drawn examples is decreased: the difference between training and test accuracy is 26.5% when the train set is composed only by computer-generated images, and it decreases to 1.0% when the train set is composed only by human-generated images. This agrees with the expectations, since including hand-drawn images boosts the richness of the training set, and also due to data augmentation, which further increases variability.

Nonetheless, it is relevant to mention case (iii), where hand-drawn images were used, but without augmentation. In this case, there is a non-expected increase in the difference between train and test accuracy. This might be explained by the fact that the training set is relatively small, increasing the risk of overfitting, which highlights the importance of data augmentation.

5.2. Object Detection - YOLOv2
We trained the networks on the Amazon Web Services platform, namely on the g3s.xlarge machine, with the following characteristics: 1 GPU NVIDIA Tesla M60, 4 vCPU, 30.5 GB RAM.

In order to allow fair comparisons between the performances of the different setups, the test used was the same for all the experiments. It corresponds to 140 images (20% of the whole dataset) that were randomly selected from the original hand-drawn dataset. Consequently, we constructed four different training sets:

- 100% human-drawn with no data augmentation (557 images). Represented as ♦.
- 100% human-drawn with data augmentation (5570 images). Represented as ◆.
Table 1: Sketch-a-Net accuracy results for different training sets.

<table>
<thead>
<tr>
<th>ID</th>
<th>Training set size</th>
<th>Train ratio (%)</th>
<th>Train (%)</th>
<th>Val (%)</th>
<th>Test (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i)</td>
<td>10000</td>
<td>100/0</td>
<td>96.56</td>
<td>77.08</td>
<td>70.1</td>
</tr>
<tr>
<td>(ii)</td>
<td>10000</td>
<td>50/50</td>
<td>95.97</td>
<td>92.50</td>
<td>90.9</td>
</tr>
<tr>
<td>(iii)</td>
<td>720</td>
<td>0/100</td>
<td>94.46</td>
<td>89.58</td>
<td>84.8</td>
</tr>
<tr>
<td>(iv)</td>
<td>5000</td>
<td>0/100</td>
<td>89.11</td>
<td>91.67</td>
<td>88.1</td>
</tr>
</tbody>
</table>

Binarized

<table>
<thead>
<tr>
<th>ID</th>
<th>Training set size</th>
<th>Train ratio (%)</th>
<th>Train (%)</th>
<th>Val (%)</th>
<th>Test (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(v)</td>
<td>10000</td>
<td>50/50</td>
<td>96.50</td>
<td>88.75</td>
<td>89.7</td>
</tr>
<tr>
<td>(vi)</td>
<td>5000</td>
<td>0/100</td>
<td>92.02</td>
<td>88.33</td>
<td>89.3</td>
</tr>
</tbody>
</table>

Non-Binarized

- 53% human-drawn with data augmentation + 47% computer-generated (10570 images). Represented as ♣.
- 36% human-drawn with data augmentation + 64% computer-generated (15570 images). Represented as ♠.

At testing time, the algorithm, besides the class and the bounding box coordinates, also outputs a confidence value. Usually, a threshold is defined in order to ignore elements with a low confidence value, which is tuned during testing. It is important to realise that if this threshold is high, then the tendency is so that many detections will be ignored, and thus the true positives count will be reduced (increasing precision), whereas if the threshold value is low, then the tendency is so that many detections will be considered, even if they have a low confidence value (increasing recall).

We opted to use all of the training data to train the network, without any validation set. This was possible since there were not many hyperparameters to be tuned but also due to the fact that it was relatively simple to understand when the network had already finished training since, due to the high variability of the network, it was unlikely that it would overfit to the training set.

Tables 2 and 3 present the results for the ♥ and the ♠ training sets. The confidence threshold chosen in both cases was 0.5.

Table 2: Accuracy results for the ♥ dataset.

<table>
<thead>
<tr>
<th>Test (%)</th>
<th>button</th>
<th>input_text</th>
<th>radio</th>
<th>select</th>
<th>mAP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>87.50</td>
<td>80.47</td>
<td>46.80</td>
<td>78.75</td>
<td>73.38</td>
</tr>
</tbody>
</table>

Table 3: Results for the ♠ dataset (threshold=0.5).

<table>
<thead>
<tr>
<th>Test (%)</th>
<th>button</th>
<th>input_text</th>
<th>radio</th>
<th>select</th>
<th>mAP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>88.60</td>
<td>83.74</td>
<td>44.24</td>
<td>77.54</td>
<td>73.53</td>
</tr>
</tbody>
</table>

Analysing both Tables, we can draw some conclusions. It is clear that the element that the network has the hardest time with is the radio: the accuracy for that element is always inferior to the rest. This may be justified by the fact that this element presents the highest variability as one or multiple circles can represent it. Thus, the network has difficulties correctly locating it and, especially when it is composed by several circles. The easiest element tends to be the button, closely followed by the input_text and the select. This order, even though it is relatively close, suggests that the network is more capable of correctly locating and classifying elements that have a more defined and distinct structure, which is in line with what is expected. Note that all the three elements are composed by a rectangular form, what distinguishes them is what is inside that rectangle. The button has “BUTTON” written inside, the input_text has a filled square, and the select has a filled triangle pointing downwards. One would expect that the easiest to distinguish would be the rectangle with written text inside, followed by the square, and only after the smaller triangle, which is in line with the performance of the network.

These results confirm the relevance of the work performed in the image classification task, where the nomenclature and representation of the elements were defined. The high accuracy obtained demonstrates that the chosen representations are adequate for the network to be able to distinguish between elements. These conclusions still hold for the rest of the different setups.

Table 4 shows the results for the same dataset, but with varying values for the confidence threshold: 0.2, 0.5 and 0.7. It leads to the conclusion that the lower the threshold, the higher the mAP. This is self-evident since the lower the threshold, the more detections the network will consider, even if they have low confidence values, and it is enough that some of those extra detections happen to be correct to increase the mAP value further.

However, this metric does not show what happens
with the false positives, which can only increase with lower thresholds. In order to understand this, Table 4, also presents the number of class-wise true positives and false positives, for each threshold. In this Table, it is clear that with the reduction of the threshold, both the number of true positives and false positives increases, as expected.

Table 4: Results for the dataset, for several thresholds.

<table>
<thead>
<tr>
<th>Threshold</th>
<th>Test (%)</th>
<th>button</th>
<th>input_text</th>
<th>radio</th>
<th>select</th>
<th>mAP</th>
<th>True Positive/False Positive</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.2</td>
<td></td>
<td>97.06</td>
<td>85.61</td>
<td>62.31</td>
<td>89.29</td>
<td>83.56</td>
<td>button input_text radio select</td>
</tr>
<tr>
<td>0.5</td>
<td></td>
<td>90.96</td>
<td>79.76</td>
<td>47.02</td>
<td>78.70</td>
<td>74.11</td>
<td>102/2 121/19 74/25 65/1</td>
</tr>
<tr>
<td>0.7</td>
<td></td>
<td>72.32</td>
<td>52.99</td>
<td>25.57</td>
<td>35.37</td>
<td>46.56</td>
<td>81/0 79/7 39/8 29/0</td>
</tr>
</tbody>
</table>

It is somewhat surprising that, when comparing the performances of the datasets with only human-drawn images, the training set with no data augmentation de facto has a better performance. This may be due to the fact that the images the network is tested on are more similar to the training set with no augmentation. Note that the augmentation process, which includes shearing, rotations, and scaling, generates images that are distorted copies of the originals, but at the same time those images are considerably further apart from the test set images than the original ones. It might be true that the network is more able to detect and classify more random images and elements, and thus more able to generalise to completely new examples. However, in this case the test set does not include any of those cases, it only has non augmented examples, and that might be the main reason why the training set with no augmentation outperforms the training set with augmentation. Meaning that, even though the network is more able to generalise that may not help it if the test set is similar to the training set.

Overall, this means that, in what concerns accuracy, it is more relevant to have a training set as similar to the test set as possible, rather than increasing its size and variability by including examples, either from other sources that are different from the test set (computer-generated images) or through data augmentation by distorting the original images. In fact, it is true that in this case, the network is less able to generalise to new examples (the training set is not as rich), but if the real-world examples are well defined, is it worth to train the algorithm with examples that it will never be tested on? Note that we may only draw this conclusion in cases where the training set has a minimum number of examples to be able to train the network and where the test set is relatively simple and well-defined; for cases with more complex real-world application, this reasoning may not hold.

Besides looking at quantitative measures, it is also important to analyse and discuss some individual results, to gain a better grasp of the performance of the network. The following images results are from the YOLOv2 network, with a threshold of 0.2.

The green boxes represent detections considered
Table 5: Accuracy results for the several datasets.

<table>
<thead>
<tr>
<th>Train Set</th>
<th>Test (%)</th>
<th>button</th>
<th>input_text</th>
<th>radio</th>
<th>select</th>
<th>mAP</th>
</tr>
</thead>
<tbody>
<tr>
<td>♥</td>
<td>98.17</td>
<td>91.16</td>
<td>65.89</td>
<td>93.18</td>
<td>87.10</td>
<td></td>
</tr>
<tr>
<td>♦</td>
<td>97.28</td>
<td>88.76</td>
<td>59.50</td>
<td>91.78</td>
<td>84.33</td>
<td></td>
</tr>
<tr>
<td>♣</td>
<td>97.06</td>
<td>85.61</td>
<td>62.31</td>
<td>89.29</td>
<td>83.56</td>
<td></td>
</tr>
<tr>
<td>♠</td>
<td>94.47</td>
<td>90.55</td>
<td>53.74</td>
<td>88.72</td>
<td>81.87</td>
<td></td>
</tr>
</tbody>
</table>

correct, whereas the blue boxes correspond to the ground-truth. If there is incorrect detection, either due to a low IOU or a wrong classification, it is marked with a red bounding box.

Figure 6: Detection with unexpected predictions.

Figure 6 illustrates two topics of interest. The first one is that sometimes the network makes some unexpected mistakes, such as the input text detected at the bottom of the page. This is probably a result of lowering the threshold since it is likely that this detection would have a low confidence value and thus would not be considered with a higher threshold. However, by visually scanning through the test set results, it is possible to notice that this sort of errors are considerably rare.

The second, which occurs often, is when the evaluation metric considers wrong a detection if its IOU with the ground-truth is less than 50%. This is the reason why the other three red boxes are considered wrong detections. However, if one was to use this algorithm, in practice, these would not be incorrect detections. Nonetheless, this threshold has to be defined in order to produce a metric, and having an IOU of 50% is a common choice in the literature, in order to consider a detection as correct. This demonstrates that sometimes it is more important to look at qualitative results, rather than just at a number that summarises the results.

One of the main obstacles of the representation chosen was that the radio element might be composed of several circles, which hinders the performance of the network in this particular element, as mentioned previously. Figure 7 illustrates this, which has a long array of circles that constitute a single radio element, and, since most training examples had from one to three or four circles, this particular case reveals to be even harder than usual for the algorithm to interpret (remember that YOLOv2 uses predefined sizes for its anchor boxes). In turn, what the network tries to do is to split the array into more than one part, thus making multiple detections for the same element. Nonetheless, the network is able to predict two bounding boxes that have an IOU of more than 50% with the ground-truth.

Figure 7: Image with a long array of radio elements.

Finally, the fact that some of the images had deficient quality hurt the network results: some even had to be up-sampled, which means that the image is pixelized, and thus it is hard to distinguish the
elements. Additionally, in some cases, the lighting conditions were poor, which then affects the preprocessing step, further spoiling the image that is to be given to the object detection algorithm. A low-resolution example is illustrated in Figure 8. Nonetheless, the algorithm is still capable of performing three correct detections, out of four.

5.3. Program Generation

The final step in the pipeline, after detecting every element’s class and position in the sketch, is to use that information to generate the actual webpage. There is no evaluation metric to evaluate this task since it is accomplished through a straightforward approach.

We now present the final result, which includes the initial image, the processed image which serves as the input of the algorithm, with the respective bounding boxes identified by the object detection algorithm and, on the right side, the final outcome, a screenshot of the generated webpage.

Figures 10 and 9 present examples for which the generated webpage has a large similarity with the initial sketch,

![Initial image](image1) ![Detections](image2) ![Final output](image3)

Figure 9: Accurate generation example.

In these examples, it is possible to observe the effect of the heuristics used when generating the code: the size of the generated elements is smaller than the respective bounding box (note that the bounding box surrounds the component and thus tends to be bigger); the component alignment is also visible since, despite the detected bounding boxes not being perfectly aligned, the code generation is still able to align the elements correctly; the number of sub-components in the radio element is proportional to the height to width ratio of the respective bounding box.

Figure 10 demonstrates another example of the heuristics applied: despite the fact that the bounding boxes generated for one of the radios and one of the input_text having a zero left coordinate, the code generation step corrects this by adding a small margin to those elements. Additionally, it displays the case when the input_text component is considered an input_text with only one line (the two cases on top) and when it is considered a text area, with

![Initial image](image1) ![Detections](image2) ![Final output](image3)

Figure 10: Mostly correct example.

more than one line (the case on the bottom), which depends on the height of the component.

Figures 11 and 12 illustrate how this step acts when the object detection algorithm generates wrong detections.

![Initial image](image1) ![Detections](image2) ![Final output](image3)

Figure 11: Generated webpage from an incorrect detection.

![Initial image](image1) ![Detections](image2) ![Final output](image3)

Figure 12: Example of the ability of the network to deal with a low quality image input.

As mentioned, the final step of code generation follows almost directly, with some minor heuristics introduced, the information that it has been given. This implies that the whole procedure is very much dependent on the object detection algorithm performance to achieve accurate results. Note that, in both Figures 11 and 12, there is a case where an extra element is generated due to a wrongly detected input_text. Figure 12 also demonstrates an occurrence of an element being correctly located but incorrectly classified. However, it is important to note that these cases are relatively sparse: in general, the object detection algorithm tends to have high precision, meaning that most elements it detects are actual correct detections and thus it is infrequent to detect an object that does not exist.
Nevertheless, Figure 11 confirms a previously made argument: considering the fact that some of the detections are classified as an error by the evaluation metric due to a low IOU, namely the radio
d and the top button of the example, when we use those detections for the code generation step, they seem as if they were correctly detected. Therefore, it can be argued that the accuracy that the whole pipeline presents is higher than the number given by the evaluation metric concerning the object detection step. However, it is important to note that, this ultimately depends on the element’s location precision that the user is expecting. Furthermore, Figure 12 also represents the ability of the network to deal with images that have very low quality: out of four elements, the network correctly detected 3, misclassified one and had one extra incorrect detection.

6. Conclusions
The aim of this research project was to “detect and recognise different UI elements, and their relative positions, in a sketch or mockup so that from this representation we can generate the code for that UI”. This subject is of utmost relevance when it comes to productivity and efficiency of teams developing UIs, as, if successful, it would allow rapid iterations and thus a more agile process, especially in the brainstorm and prototyping phase of the development, when there is still an abundance of choices to be made.

Our approach was to split the problem into two steps: object detection to locate and identify every component of the sketch, and then, using that information generate the respective code for the webpage. In order to tackle the problem at hand, and considering that there was no specific previous work that we could follow, we had to start from scratch: define which elements to use and how they would be represented. The next step was to create and optimise an object detection algorithm that was able to pinpoint the position of the elements in the sketch and their class. This was accomplished using a state-of-the-art algorithm, YOLOv2 [2]. In what concerns the training set, there were two distinct approaches to generate it: using a computer program that simulates hand-drawn sketches and also actual human drawings. The final step was the code generation step, where, using the information from the previous phase and absolute positioning, it was possible to generate the webpage that represented the input sketch.

All in all, with the assembling of the complete pipeline, and by looking at the results for both test sets, we can conclude that the ultimate goal of generating a webpage from an UI hand-drawn sketch was successfully achieved. The methods and the philosophy adopted has potential to be followed for future research projects in this area.

Several possible future directions may serve as a continuation and consequently achieve the full potential of this idea, namely: text detection and interpretation using optical character recognition (OCR); enhance the program generation step, using a program synthesis technique; prepare the whole pipeline to be available to be run in real-time with frames from a video; include more elements to the current set and granting the user with more possible choices and diversity of end-designs.

Acknowledgements
This work was supported by OutSystems under a collaboration with Instituto Superior Técnico.

References