Converting Web Pages Mockups to HTML Using Machine Learning

Tiago Bouças[1], António Esteves[2][a]
Centro ALGORITMI, School of Engineering, University of Minho, Campus de Gualtar, Braga, Portugal
tiagoalvesboucas@gmail.com[1], *esteves@di.uminho.pt*[2]

Keywords: Deep learning, convolutional neural network, recurrent neural network, YOLO, Web page mockup

Abstract: Converting Web pages mockups to code is a task that developers typically perform. Due to the time required to accomplish this task, the time available to devote to application logic is reduced. So, the main goal of the present work was to develop deep learning models to automatically convert mockups of Web graphical interfaces into HTML, CSS and Bootstrap code. The trained model must be deployed as a Web application. Two deep learning models were built, resulting from two different approaches to integrate in the Web application. The first approach uses a hybrid architecture with a convolutional neuronal network (CNN) and two recurrent networks (RNNs), following the encoder-decoder architecture commonly adopted in image captioning. The second approach is focused on the spatial component of the problem being addressed, and includes the YOLO network and a layout algorithm. Testing with the same dataset, the prediction's correction achieved with the first approach was 71.30%, while the second approach reached 88.28%. The first contribution of the present paper is the development of a rich dataset with Web pages GUI sketches and their captions. There was no dataset with sufficiently complex GUI sketches before we start this work. A second contribution was applying YOLO to detect and localize HTML elements, and the development of a layout algorithm that allows us to convert the YOLO result into code. It is a completely different approach from what is found in the related work. Finally, we achieved with YOLO-based architecture a prediction's correction higher than reported in the literature.

1 Introduction

For many years, machines only replaced the human being in tasks that involved force. With the use of mechanized force many jobs were lost, but others more linked to cognitive ability were gained. Today, artificial intelligence has reached an impressive development, which is mainly due to the recent computational development. Huge complexity tasks are now performed faster and more efficiently by machines. Professions that involve repetitive tasks risk disappearing, while others will be completely remodeled.

This paper presents the development and deployment of deep learning models to convert graphical user interface (GUI) sketches, elaborated with the Balsamiq Mockups application, into HTML, CSS and bootstrap code. Converting GUI sketches to code is a task commonly performed by programmers. Due to the time consumed by this task, it becomes impossible to devote more time to application logic. On the other hand, it also becomes a repetitive and tedious task. The developed models will make it easier and faster for programmers to work, as they automatically generate code from an outline of the application interface made with Balsamiq Mockups. After receiving the code, the user just needs to add JavaScript code, replace the default text and customize the appearance of the generated page.

The first contribution of the present work is the development of a rich dataset with Web pages GUI sketches and their captions. Due to the lack of datasets with sufficiently complex GUI sketches, we decided to construct our own dataset. A second contribution was applying YOLO to detect and localize HTML elements, and the development of a layout algorithm that allows us to convert the YOLO result into code. It is a completely different approach from what is found in the related work.

After presenting the related work in section 2, it is described the followed methodology. Section 3.1 summarizes the dataset used to train and test the DL models. The next two sections present two distinct approaches to address the mockups conversion problem. The hybrid approach, described in section 3.2, follows an encoder-decoder architecture. The second approach is presented in section 3.3 and uses YOLO.

[a] https://orcid.org/0000-0003-3694-820X

Section 3.4 briefly describes the deployment of the models through a web application. In section 4 we present the realized experiments and the achieved results. This section allows us to understand which one is the best combination of neural networks in the hybrid approach and the best accuracy achieved by the YOLO approach. Two prediction examples are presented next. The paper ends with the conclusions and future work (section 5).

2 Related Work

Image captioning is much more than image recognition or classification. Captioning has additional challenges such as recognizing dependencies between objects that are part of the same image and creating sequential text. (Hossain et al., 2018), (Mullachery and Motwani, 2016), and (Srinivasan et al., 2018) are examples of works that allow automatic captioning, applied to photographs taken in everyday life, through the combination of convolutional and RNNs.

In (Balog et al., 2017), authors have shown that it is possible to train a deep neuronal network (DNN) to predict program properties from inputs and outputs. In (Mou et al., 2015), it is presented a study showing that it is possible to convert an intention, described textually, to C code. Through recurrent networks, it is possible to understand the user's intent and to generate part of the code. Their model does not always generate completely correct code.

Work in (Deng et al., 2017) demonstrates that DNNs, including CNNs and RNNs (Tan and Wang, 2018), achieve better results when compared to classical techniques, such as OCR, even in handwritten data. This project allows us to generate LaTeX code for an equation given its image. They follow an attention-based approach to highlight important features present in the provided image, something that human being does very well. Their goal is to avoid losing relevant information (Xu et al., 2015).

Currently there is a great curiosity about what can be achieved with automatic code generation. The author of (Beltramelli, 2017) showed that it is possible to generate HTML and CSS code from Web pages GUI sketches. Given a dataset with images and the associated code, a CNN makes it is possible to extract the characteristics of the images and an RNN allows him to obtain the image description. The decoder RNN was trained with supervised data, including images and the respective code. The output from the RNN is compiled in order to get functional HTML code. This work is only focused on the layout and ignores the textual part.

The system developed in (Capece et al., 2016) implements a deep learning (DL) currency recognizer, based on a client-server architecture. They show that we can obtain a good CNN accuracy in currency recognition. However, it requires a relatively large dataset, since CNNs do not perform well with little data. Users can photograph a coin and send the image to the server. The provided image is then classified with the trained model.

3 Methodology

This section presents the development and deployment of deep learning models to convert graphical user interface sketches, elaborated with the Balsamiq Mockups application, into code.

3.1 Dataset

The lack of datasets with sufficiently complex GUI sketches led us to create one from scratch. The Web pages GUI sketches were designed with the aid of the Balsamiq Mockups tool. The developed dataset includes the most commonly used Bootstrap components, such as images, videos, buttons, navigation bars, and tables. It consists of 1100 images, 1000 for training and 100 for testing. Covering only a subset of the Bootstrap components is due to the amount of sketches that would be necessary to create in order to support all the components. The elements contained within the upper navigation bar are recognized as independent elements by the DL model. The same does not happen with the side navigation bar, where the recognition of internal elements is not performed for the sake of keeping the dataset smaller.

The inputs for the DL models developed on the two approaches reported in this paper are different. So, while in the hybrid approach it was necessary to caption the images with DSL code, in the YOLO approach XML annotations were created.

3.2 The Hybrid Approach

The hybrid approach follows a encoder-decoder architecture, similar to the common architecture used in image captioning with DNNs (Vinyals et al., 2015)(Vinyals et al., 2017). The encoder consists of two neural networks: a CNN that receives an image as input and a RNN that receives text as input. The decoder neuronal network, which is not necessarily the same as the encoder, plays the exact opposite role of the encoder: receives the concatenated feature vector as input and outputs the closest match based on the input. Encoder and decoder are trained together and work to reduce the cost function.

The hybrid approach architecture combines a CNN with two RNNs to receive an image as input and to generate the caption for that image. In this case, the captioning of sketches is done with DSL code, instead of the usual textual description in a natural language such as Portuguese or English. Given the complexity and size of HTML and CSS code, it was necessary to create a DSL to turn automatic code generation easier.

3.2.1 Domain Specific Language

Domain specific languages (DSL) (Kosar et al., 2015) are programming languages with limited expressiveness, focused on a particular application domain. Limited expressiveness means that the language only serves the minimum requirements for the application domain it was designed. The opposite is a general purpose language, such as Java, C or Python. The fact that a programming language is built specifically to solve problems in a given domain facilitates its interpretation, since it is composed of elements and relations that directly represent the logic of that domain. Using a DSL in the hybrid approach was essential, as it would be difficult a model to learn generating HTML and CSS code correctly, due to its complexity. The DSL simplifies the code generation and facilitates the task of the DL model (LeCun et al., 2015). Listing 1 contains the DSL code for a simple sketch consisting of an image and two text blocks.

```
content {
    row {
        col {
            image
        }
        col {
            p
        }
    }
    row {
        p
    }
}
```

Listing 1: An example of DSL code.

3.2.2 Compiler

A compiler translates high level code to lower level code. For the most popular languages, such as C or Java, the compiler translates a high level language that is understood by the user into a lower level language that the machine can execute. Tools like ANTLR4 let us create our own programming language. The developed compiler was used to convert the described DSL code into HTML and CSS code. The grammar, created to solve the problem of converting sketches to HTML, consists of a set of terminal symbols, including Bootstrap elements such as images, videos, links or tables. These symbols cannot be divided into smaller units, hence the ”terminal” designation. The top navigation bar consists of a set of elements, such as buttons, images, or titles. It is divided into smaller units and is therefore called a non-terminal symbol. Non-terminal symbols consist of combinations of terminal and non-terminal symbols.

3.2.3 Model architecture

The converter follows a different architecture during training and inference. In the training phase, the DL model receives as input a vector that results from concatenating the image features with the correspondent DSL code, while in the inference phase the input vector contains only the image and the `<start>` tag. The DL model follows an encoder-decoder architecture, inspired by the machine translation and image captioning literature. During the encoding step, the input, i.e. an image and the associated DSL code, is transformed into a fixed-length vector. In the decoding step, the encoded vector is interpreted. The decoding task is different during training and inference. During the **training phase**, the decoder receives as input the concatenated vector, which is used to learn the relationship between the image and the associated DSL code (figure 1).

During the **inference phase**, the DL model receives as inputs the image vector and the `<start>` tag. The remaining DSL code will be generated by the model. While the model is generating DSL code for the image, the output sequence grows until it reaches an established maximum number of iterations, or until it generates the `<stop>` tag that terminates the conversion (figure 2).

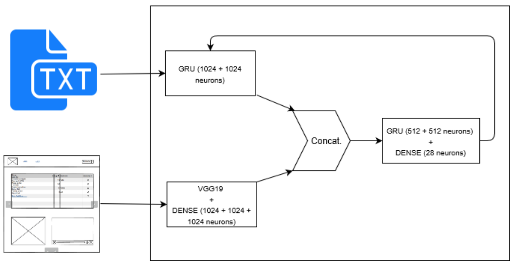

TXT
GRU (1024 + 1024 neurons)
Concat.
GRU (512 + 512 neurons) + DENSE (28 neurons)
VGG19 + DENSE (1024 + 1024 + 1024 neurons)

Figure 1: Hybrid approach architecture during training.

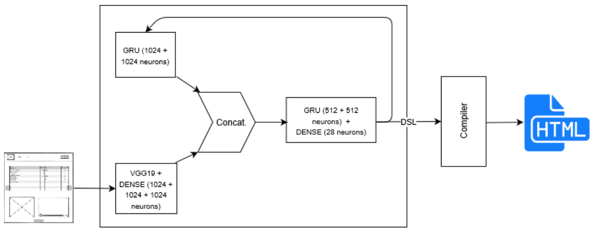

GRU (1024 + 1024 neurons)
Concat.
GRU (512 + 512 neurons) + DENSE (28 neurons)
DSL
Compiler
HTML
VGG19 + DENSE (1024 + 1024 + 1024 neurons)

Figure 2: Hybrid approach architecture during inference.

3.2.4 Metrics

A metric was developed specifically to evaluate the code generated by the DL model. BLEU score is the most commonly used metric in machine translation, but it compares only the generated code with the expected one, word by word (Papineni et al., 2002). In our case, this would neglect the most important aspect, i.e., the number of elements correctly identified.

The success of the developed metric is measured by the percentage of elements correctly identified, the placement of the elements on the correct row and column (with a smaller weight), and a penalty associated with the incorrect correspondence between curly braces. Since the metric focus on these points, it produces values closer to reality in our automatic code generation problem. To make the results even more realistic, different weights are assigned to each mentioned component. Due to the low probability of the model generating incorrect curly braces, there is a weight that reduces the result by 5% when the curly braces are in wrong place. The binary variable *correctCBraces* indicates whether there is a correct curly braces matching or not (equation 1).

$$weightCBraces = 0.95 + 0.05 * correctCBraces \quad (1)$$

Equation 2 allows us to evaluate the model output, based on a given weight that penalizes failures in the generation of braces, the number of correctly generated elements, and the correct placement of elements on the generated Web page.

$$weightCBraces * (0.8 * \frac{truePositives}{occurrences} + 0.1 * \frac{corretRows}{occurrences} + 0.1 * \frac{correctColumns}{occurrences}) \quad (2)$$

The *weightCBraces* is computed with equation 1, *occurrences* is the number of elements generated by the model, *truePositives* is the number of elements that are generated correctly, *corretRows* and *correctColumns* are the number of elements placed in the correct row and column, respectively.

3.3 The YOLO Approach

The second approach adopted for converting GUI sketches to code resulted from the fact that the hybrid approach lacked an adequate treatment of the spatial component of sketches, which made it difficult to generate correct layouts. The main tool that allowed us to tackle this challenge was YOLO, a DNN that presents good results in the detection and localization of objects. In this way, we intended to find out which objects are present in the user-provided image. It is also important to know the location of the objects, so that the final layout is as close as possible to the input sketch. Based on the information obtained by YOLO, a layout algorithm has been developed to map the objects detected in the provided image into HTML and CSS code. The layout algorithm takes advantage of CSS's placement and size properties to place elements on the Web page. This allows us to place each object in a position very close to the correct one. The points provided by YOLO allow us to deduce the width and height of the detected objects and thus get HTML pages with a visual layout very close to the input sketch.

3.3.1 YOLO model

YOLO is a deep learning model for real-time object detection and localization, that has evolved through four versions (Redmon et al., 2016) (Redmon and Farhadi, 2016) (Redmon and Farhadi, 2018) (Bochkovskiy et al., 2020). Training accurate object detection models requires many GPUs and using a large batch size. The most updated version of YOLO avoids these inconvenient requirements by making an object detector which can be trained on a single GPU with a smaller batch size. It combines features such as weighted residual connections, cross-stage partial connections, cross mini-batch normalization, self-adversarial training and Mish activation, mosaic data augmentation, DropBlock regularization, and Complete-IoU loss. YOLO generates a list of objects and the correspondent bounding boxes.

3.3.2 Layout algorithm

The CSS properties allow us to position elements on the desired position of a Web page. This makes it relatively easy to map the objects recognized by the model into HTML code. For each detected object, YOLO returns two points that define its bounding box: (x_{min}, y_{min}) and (x_{max}, y_{max}). In the absence of more accurate information, the bounding box is used to set the location and size of the object to place on the Web page. The top-left corner of the object's position, can be specified by the CSS `top` and `left` properties, and assumes the y_{min} and x_{min} values returned by YOLO, respectively. In the same way, the CSS `width` and `height` properties allow us to specify the size of the object on the Web page, and are assigned

$x_{max} - x_{min}$ and $y_{max} - y_{min}$ values, respectively. These CSS properties allow us to generate Web pages visually similar to the input sketches.

The `viewport` is the area where the browser draws the Web page content. We implemented a function to ensure that the size of the Web page elements fit our display. Through simple calculations, this function turns the generated Web pages responsive. For example, given sketches with a fixed-size of 256×256 pixels, equations 3 to 6 convert the coordinates of the detected objects to a variable size viewport, which fits the size of our display.

$$left = (x_{min} * 100vw) \div 256 \quad (3)$$
$$top = (y_{min} * 100vh) \div 256 \quad (4)$$
$$width = (width * 100vw) \div 256 \quad (5)$$
$$height = (height * 100vh) \div 256 \quad (6)$$

Where *vh*, or viewport height, is based on the height of the viewport. A value of $100vh$ is equal to 100% of the viewport height. *vw* is the viewport width and it is based on the width of the viewport.

For each HTML element, a template with HTML and CSS code was defined. The template has tags to delimit the places to be replaced by the values generated by YOLO. As explained before, the CSS properties that define the position and size of HTML elements can be replaced by values obtained by YOLO.

3.3.3 Model architecture

Like in the hybrid approach, the developed model follows a different architecture during training and inference. During **training**, YOLO receives a file with the identification of the images, the objects included in each image and their bounding boxes. In addition to the image identification, the model receives the following information for each object: $x_{min}, y_{min}, x_{max}, y_{max}$, and the object class. Each image contains one or more objects (figure 3).

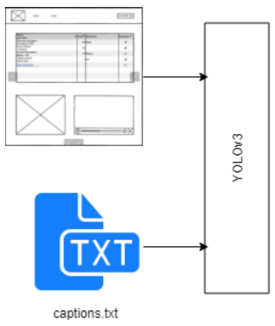


Figure 3: YOLO approach architecture during training.

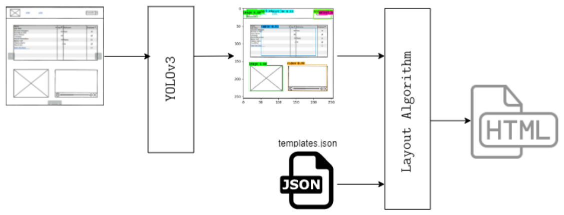


Figure 4: YOLO approach architecture during inference.

During the **inference phase**, the model is only fed with an image for which it must generate HTML and CSS code. Since the model only predicts the bounding box and the object's class, it was necessary to apply a layout algorithm to convert the YOLO output to code. The layout algorithm receives the list of bounding boxes generated by YOLO, which contains the location and the object's class $(x_{min}, y_{min}, x_{max}, y_{max}, class)$, and the file with the HTML elements templates. The algorithm converts the information about objects to HTML and CSS code. We always get a file with functional code. To simplify the task of handling the end result by users, the HTML, CSS and Bootstrap codes are placed in a single HTML file (figure 4).

3.3.4 Metrics

Two metrics were developed for the YOLO approach, allowing us to evaluate the quality of the generated HTML code. Quality refers to the match between the image provided as input and the HTML and CSS page generated by the model as output. The first metric measures the **accuracy**, i.e., the number of elements correctly detected divided by the number of identified elements. The second metric measures **precision**, and counts the number of elements correctly identified on each class. Both metrics consider the intersection between the predicted and the true bounding boxes. Through the ratio between the interception and the union of the bounding boxes (IoU), which measures the percentage of coincidence between these regions, the error in generating bounding boxes is penalized. The IoU is also called Jaccard index and Jaccard similarity coefficient (Fletcher and Islam, 2018).

The metrics applied in both approaches were sought to be similar. Thus, a weight of 80% was assigned to the number of elements correctly identified and 20% was applied to the correct localization of objects. Including the IoU value in the global metric, ensures that when we minimize this metric we are minimizing the size and position errors associated with the identified elements.

Equation 7 calculates the mean IoU over the N true objects. The algorithm compares each true bounding box with the predicted one and realizes which of the predicted regions correspond to the true region. The comparison is based on the object category and the IoU value. Equation 8 measures the accuracy, through the number of correct predictions, while equation 9 accounts for the precision of the C classes of objects, with a weight of 80%, and for the IoU mean, with a weight of 20%.

$$\overline{IoU} = \frac{\sum_{i=1}^{N} IoU_i}{N} \tag{7}$$

$$acc = 0.8\, \frac{TruePositives}{occurrences} + 0.2\, \overline{IoU} \tag{8}$$

$$p = 0.8\, \frac{\sum_{i=1}^{C} \frac{TruePositives_i}{TruePositives_i + FalsePositives_i}}{C} + 0.2\, \overline{IoU} \tag{9}$$

3.4 Deep Learning Models Deployment

React is a declarative, efficient, and flexible JavaScript library for building user interfaces. React introduced the concept of component-based architecture, where each component manages its own state. The components can be seen as small independent parts, which together constitute the user interface.

The developed DL models were deployed as a Web application. The application interface was divided into 4 components: the `header`, the `frameview` that shows in real-time the code changes done on the editor, the `card` describing the steps to be taken when loading an image in the main page, and the `CodeMirror`, an external component used as text editor. The application consists of only two pages, the home page and the code editor. The home page consists of the `header` and 3 `card` components, which work as a tutorial on how to use the application. The editor page contains the text editor, implemented with a `CodeMirror` component, and the `frameview`. The user can create, edit or remove content from the text editor and automatically view changes in the `frameview`. This feature saves users time, avoiding having to open the HTML page to view its content.

4 Experiments and Results

This section presents the experiments conducted during the development of the mockup's converter, as well as their results. Experiments include evaluating different convolutional neural networks (VGG16, VGG19, ResNet-50, etc.) on the task of feature extraction, the difference between GRU- and LSTM-based recurrent networks, assess the impact of RNNs optimized with CUDA on the training time (via `CuDNN`), and the comparison of the results obtained with both presented approaches. Both approaches were trained with the same images and tested in the same scenarios, in order to allow an appropriate comparison. The image captions vary between a DSL description (hybrid approach) and an XML description of the objects plus the respective bounding boxes (YOLO approach). Although the languages are different, the captions are equivalent, as they describe the same image.

Encoder CNN	**Enc. RNN**	**Dec. RNN**	**Acc.(%)**
InceptResNetV2	cudnngru	cudnngru	12.73
InceptionV3	cudnngru	cudnngru	32.66
ResNet-50	cudnngru	cudnngru	34.75
vgg16	cudnngru	cudnnlstm	60.36
vgg16	cudnnlstm	cudnnlstm	61.9
vgg16	GRU	GRU	66.73
vgg16	cudnngru	cudnngru	67.35
vgg19	cudnngru	cudnngru	71.30

Table 1: Results from the hybrid approach experiments.

Encoder CNN	**Encoder RNN**	**Decoder RNN**	**Time (min)**	**Acc. (%)**
inception3	cudnngru	cudnngru	239	26.16
vgg16	cudnngru	cudnngru	346	60.27
resnet-50	cudnngru	cudnngru	373	20.43
vgg16	cudnnlstm	cudnnlstm	373	26.21
vgg16	cudnngru	cudnnlstm	374	56.15
vgg19	cudnngru	cudnngru	423	63.12
vgg19	cudnngru	cudnnlstm	429	59.09
vgg16	gru	gru	565	63.26

Table 2: Results from hybrid approach after 20 iterations.

4.1 Hybrid approach experiments

The experiments carried out aimed to select the best combination of neural networks, based on the accuracy between the real sketches and those generated by the model in the test dataset. The best combination will be compared with the YOLO-based network.

The `cudnnlstm` and `cudnngru` are normal RNN cells, optimized with CUDA, for faster training on GPUs. CUDA provides an interface that makes it easy to explore the parallelism available on GPUs. The `CuDNN` library provides ML tools with an implementation of Nvidia's GPU-optimized linear algebra operations. According to tables 1 and 2, and for the same number of epochs, the CUDA version takes considerably less time to train, maintaining a similar accuracy. It is also verified that the final precision is slightly higher when using the RNN's CUDA version.

GRU cells are relatively newer and simpler than LSTMs. According to the results obtained with the hybrid experiments (table 1), GRU-based RNNs train faster and have better results than LSTMs. Table 2 also show that, with the same number of epochs, the GRU-based RNNs present better results. Although with more epochs, the CUDA version ends up showing better results. LSTMs have the advantage of being able to memorize information from longer sequences, due to their more complex architecture.

After several attempts with different neuronal networks, with and without transfer learning, and with different hyperparameters, 71.30% accuracy was the best result obtained with the test set. This result was obtained using transfer learning, model weights from the `Imagenet` dataset, fine-tuning, and freezing the weights of the first layers.

Due to a GPU memory limitation, 8GB on the Nvidia GTX 1070, some networks could not be tested without fine-tuning. For example, when using the 200-layers `InceptionResNetV2` (Szegedy et al., 2016), it was necessary to freeze more than half of the layers to reduce the required memory. This ended up limiting the experiments carried out and, consequently, the results. Only the `VGG16` and `VGG19` networks allowed satisfactory results.

4.2 YOLO approach experiments

The YOLO approach aimed to verify whether an architecture, known for having a high performance in the detection and location of objects, would perform better than our hybrid model. In each iteration of training and validation, a value for the loss is obtained. The loss quantifies how well the model is adapting to both training and test sets, and unlike precision, its value is not a percentage. As the loss function sums the errors, we must minimize its value. Figure 5 shows the training and validation loss along 100 epochs. The analysis of the figure reveals that the curves start to diverge in epoch 78. Table 3 shows the accuracy at 5 checkpoints during training.

YOLO and the layout algorithm achieved very good results. The model reaches the best score at epoch 78. After 254 minutes of training the generated sketches are 88.28% accurate. According to the metric presented in equation 9, the best precision is 88.4%. The third version of YOLO architecture was used. Tests were carried out with different learning rates and optimizers. This approach achieved an excellent 88.28% accuracy in the test set. The HTML code generated by the YOLO approach is much more similar to the provided input than in the hybrid approach. YOLO finds the exact location of the bounding region and the layout algorithm places elements in the correct positions (figures 6 and 7). The same

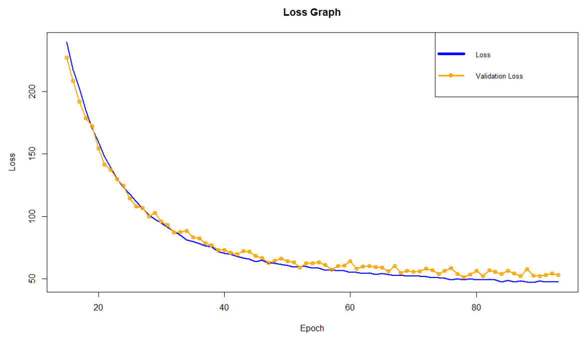
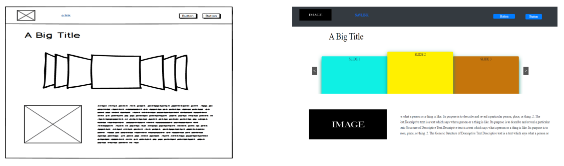


Figure 5: Training/validation loss in YOLO approach.

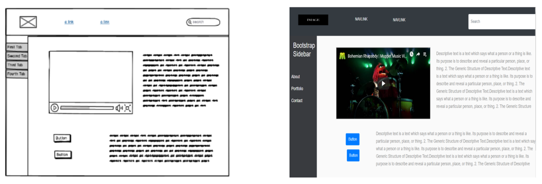
Epoch	48	58	68	78	88
Accuracy (%)	60	74	83	88	83

Table 3: Test accuracy of the YOLO approach.

is not true in the first approach, which does not preserve the margins nor the size of the elements. In the hybrid approach, the elements have size and position assigned by default in a template file.



(a) input mockup (b) output page
Figure 6: First example of prediction with YOLO approach.



(a) input mockup (b) output page
Figure 7: 2^{nd} example of prediction with YOLO approach.

5 Conclusions

In the hybrid approach, the characteristics of the elements to be inserted in the HTML code are stored in a file, containing a template for each type of element commonly found in Web pages. So, the elements do not vary in size, and their position is defined only by a line-column pair, which confines the final appearance of the generated page. The final result is always different from the input mockup. YOLO handles more appropriately the conversion of mockups to code. The YOLO network identifies the elements of a mockup, as well as the respective location. The layout algorithm maps each object recognized by YOLO into HTML and CSS code, based on the coordinates of the bounding box. This algorithm places the elements in a HTML file, using CSS properties that allow placing the elements given their coordinates.

The metrics developed for both approaches apply the same weights: the number of elements generated correctly is weighted 80% and the remaining 20% are applied to the dimensions and positioning of the elements. The hybrid approach achieved as best accuracy 71.30%, while the YOLO approach achieved 88.28% of accuracy and 88.4% of precision. The second approach generates HTML code that contains objects with the correct size and position, which naturally results in Web pages much more similar to the provided mockups. The YOLO approach covered a wide variety of HTML elements and reached an accuracy that outperforms the related approaches.

As future work we propose to implement a layout algorithm with division by line and column, as occurs in the `Bootstrap` framework. Since the YOLO approach provides the coordinates of the bounding regions, the algorithm to be developed must be able to find the correct margins, in order to position the elements closer to the coordinate mapping that is being used, thus making the generated code responsive. To get around the biggest problem found in this work, the lack of data, it is proposed to increase the size and diversity of the dataset. This measure aims to improve the object's detection accuracy, but fundamentally to improve the accuracy of the coordinates of the bounding box. It is also planned to increase the variety of supported HTML elements. It is also intended to create metrics for the assessment of precision and recall.

Acknowledgment

This work has been supported by FCT - Fundação para a Ciência e Tecnologia within the R&D Units Project Scope: UIDB/00319/2020.

REFERENCES

Balog, M., Gaunt, A., Brockschmidt, M., Nowozin, S., and Tarlow, D. (2017). Deepcoder: Learning to write programs. *ICLR 2017*.

Beltramelli, T. (2017). pix2code: Generating code from a graphical user interface screenshot.

Bochkovskiy, A., Wang, C.-Y., and Liao, H.-Y. (2020). Yolov4: Optimal speed and accuracy of object detection.

Capece, N., Erra, U., and Ciliberto, A. (2016). Implementation of a coin recognition system for mobile devices with deep learning. *Conf. Signal-Image Technology & Internet-Based Systems*.

Deng, Y., Kanervisto, A., Ling, J., and Rush, A. M. (2017). Image-to-markup generation with coarse-to-fine attention. *Int. Conf. on ML*.

Fletcher, S. and Islam, M. (2018). Comparing sets of patterns with the jaccard index. *Australasian Journal of Information Systems*, 22.

Hossain, M. Z., Sohel, F., Shiratuddin, M. F., and Laga, H. (2018). A comprehensive survey of deep learning for image captioning.

Kosar, T., Bohra, S., and Mernik, M. (2015). Domain-specific languages: A systematic mapping study. *Information and Software Technology*, 71.

LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. Technical report.

Mou, L., Men, R., Li, G., Zhang, L., and Jin, Z. (2015). On end-to-end program generation from user intention by deep neural networks.

Mullachery, V. and Motwani, V. (2016). Image captioning. *arXiv*, abs/1805.09137.

Papineni, K., Roukos, S., Ward, T., and Zhu, W.-J. (2002). Bleu: a method for automatic evaluation of machine translation. pages 311–318.

Redmon, J., Divvala, S., Girshick, R., and Farhadi, A. (2016). You only look once: Unified, real-time object detection. *Conf. on Computer Vision and Pattern Recognition (CVPR)*, pages 779–788.

Redmon, J. and Farhadi, A. (2016). Yolo9000: Better, faster, stronger. *Conf. on Computer Vision and Pattern Recognition (CVPR)*.

Redmon, J. and Farhadi, A. (2018). Yolov3: An incremental improvement. *ArXiv*, abs/1804.02767.

Srinivasan, L., Sreekanthan, D., and A.L, A. (2018). Image captioning - a deep learning approach. *Int. Journal of Applied Engineering Research*, 13.

Szegedy, C., Ioffe, S., Vanhoucke, V., and Alemi, A. (2016). Inception-v4, inception-resnet and the impact of residual connections on learning. *AAAI Conference on Artificial Intelligence*.

Tan, K. and Wang, D. (2018). A convolutional recurrent neural network for real-time speech enhancement.

Vinyals, O., Toshev, A., Bengio, S., and Erhan, D. (2015). Show and tell: A neural image caption generator. In *IEEE Conf. on Computer Vision and Pattern Recognition*, pages 3156–3164.

Vinyals, O., Toshev, A., Bengio, S., and Erhan, D. (2017). Show and tell: Lessons learned from the 2015 MSCOCO image captioning challenge. *IEEE Trans. Pattern Anal. Mach. Intell.*, 39(4):652–663.

Xu, K., Ba, J., Kiros, R., Cho, K., Courville, A., Salakhutdinov, R., Zemel, R., and Bengio, Y. (2015). Show, attend and tell: Neural image caption generation with visual attention.