Derendering HTML Web-pages

Colin Dolese
Stanford
cdolese@stanford.edu

Jaak Uudmae
Stanford
juudmae@stanford.edu

Abstract

Traditional methods in computer vision generally do not produce image representations which are interpretable outside of their framework, or are otherwise specific to a particular object class. Recent solutions to this problem have represented images in a way understandable by a graphics engine. One obvious application for this procedure is in web page interfaces and their corresponding HMTL/CSS code.

Inspired by the lack of research in this area, and the clear implications for HTML, we developed a method for "derendering" images of simple HTML pages into the associated HTML. To do this, we designed a vector representation of HTML ("VectorHTML") which could be easily procedurally generated and then translated into actual HTML. This allowed us to generate hundreds of HTML page training images, which we then segmented and trained on via a support vector machine method. Ultimately we were able to accurately reproduce the HTML for our simple HTML page inputs. Our final implementation is only a proof-of-concept model, and would need to be extended to account for more complex HTML in order to function in real world contexts.

1. Introduction

Current methods for obtaining interpretable image representations from input images using encoding-decoding neural net frameworks require simple object and scene inputs in order to function. To overcome this, researchers have recently began exploring the use of deterministic rendering functions on structured image inputs to obtain easily interpretable image representations, allowing for the analysis of much more complicated inputs. One recent paper by Wu et al. [3] published in 2017 has made considerable progress on the problem at hand.

One obvious application of these methods is the synthesis of HTML web pages. At a high level, the successful implementation of these methods in this case would allow for a screen shot of an HTML page as input to produce an HTML script for producing that exact page as output. Ideally, this would then allow for a basic reconstruction of any HTML page simply by possessing an image of the page. Furthermore, the output HTML would break the input image into the core page framework and discard the unnecessary page-specific information. As such, this procedure would allow for rapid acquisition of complex HTML page templates modeled on existing pages without the need to decipher overly detailed HTML available via the pages source code.

Our team explored the possibilities of applying these techniques in this context. Specifically, we first addressed very simple HTML pages with only basic web page structures. We then implemented our image analysis pipeline so that it achieved high accuracy on our basic inputs. This model is a proof-of-concept that shows that HTML can be reconstructed using just an image.

Although our inputs were simple, we aimed to make them reflect the underlying building blocks behind all HTML pages. Therefore this project will provide a foundation that could be extended to account for increasingly more complex web pages.

The code to our project is available at: `https://github.com/ColinDolese/CS231Project`

2. Background and Related Work

Our initial inspiration came from a paper by Wu et al. [3] who developed a system for extracting a "compact, expressive, and interpretable representation of scenes" from image input. To do this, they designed an image encoding system which they referred to as "SceneXML" that could be interpreted by any graphics engine. Their use of SceneXML is what motivated us to develop our "VectorHTML" (see Experiment) to encode HTML in a simplified format.

Given an image input of a simple scene, Wu et al.'s method first segments the image and then generates attribute proposals on each segment via a neural network, encoding the image as SceneXML. Finally the SceneXML is decoded to reproduce the original image.

We were directly inspired by this work flow for our own project. Although we used a support vector machine rather than a neural net for the encoding step, we performed the same segmentation-encoding-decoding structure.

Ultimately the best results achieved by Wu et al. saw 42.74% errors on attribute prediction and 21.55% errors on pixels in the image reconstruction. These results were on images referred to as "abstract scenes." When testing on images from the video game "Minecraft" the results improved (26.41% and 5.05% respectively), which they attribute to the narrower use case. Although these errors seem high, Wu et al. demonstrate that they are improvements over other methods. Nonetheless, we expect to achieve greater accuracy in our experiments given the much narrower scope of our project.

3. Technical Approach

3.1. Process Outline

In order to train and test our algorithm we generated our own data. Technically we broke our process into four distinct sub-problems which follow closely the methods found in Wu et al. [3]:

1. Procedural Generation of VectorHTML and gathering HTML page images (generating data)
2. Proposal Generator on input images into segments (Encoder)
3. Object Interpreter on proposals (Encoder)
4. Translate VectorHTML output into HTML and Test for Correctness (Decoder)

3.2. Process Implementation

The technical implementation for each of these stages was developed in Python. For procedural generation of our training data, we designed our own encoded version of HTML we refer to as "VectorHTML." VectorHTML is the data that is actually procedurally generated and which is produced as output by the regression algorithm. The final VectorHTML output is then converted to actual HTML by our decoder algorithm. This process is described in detail in section 4.

For the Encoder, we relied on the Sci-kit Learn library [1] for image segmentation (generating proposals) and regression algorithm. The goal was to find the different HTML elements in the image and use them to fit a multi-regression algorithm that would be able to predict the properties of the given segment

We tested our algorithm for correctness by

$$\begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 4896617 & 0 & 0 & 0 & 24 & 0 & 0 & 1 \\ 13861979 & 0 & 13007 & 0 & 0 & 0 & 3 & 1 \\ 12984208 & 0 & 11294 & 0 & 0 & 0 & 1 & 1 \\ 1187460 & 0 & 128 & 0 & 0 & 0 & 2 & 1 \\ 1009062 & 0 & 0 & 0 & 12 & 0 & 0 & 1 \end{bmatrix}$$

Figure 1. A sample VectorHTML Matrix. Note that attributes with large ranges produce much larger indices than other attributes. To address this, VectorHTML will be normalized in later stages.

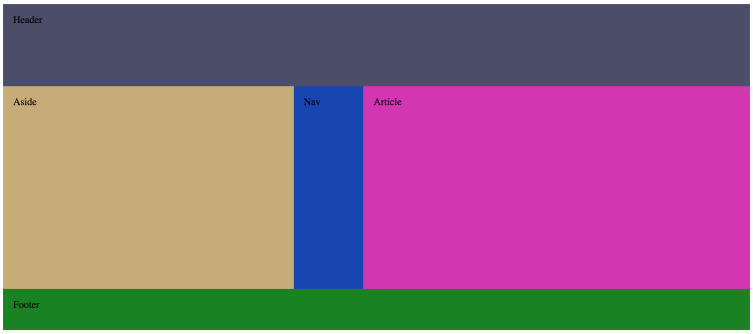

Figure 2. The resulting HTML page from the VectorHTML in Fig. 1

calculating the R^2 score of the MultiRegressor model.

4. Experiment

4.1. Procedural Python Generation

In order to effectively translate HTML page image data into actual HTML, we developed a vectorized version of HTML which we will refer to as "VectorHTML." This method was inspired by similar techniques used by [3] involving a special "SceneXML" to facilitate the encoding/decoding process. During our procedural generation stage, we randomly generate a matrix where each row describes one of six HTML tags (body, header, article, aside, navigation, footer). Each element in these rows then describes an attribute. Specifically, each element is set to an index into a dictionary we have defined containing all possible values for each attribute. If a tag does not have a value set for a given attribute, then the index will be 0 and always refer to an empty value in the dictionary.

Using this method, we are able to procedurally generate as many arbitrary HTML pages as we need. Currently, the resulting HTML is kept very simple. For training we generated 500 of those pages.

4.2. Graphics Engine Decoder

Our generated VectorHTML must be decoded into actual HTML at both ends of our algorithm in order to supply image data as input and to produce actual HTML as output. With VectorHTML clearly defined, decoding VectorHTML was simply a matter of accessing the attribute dictionary and writing a script to produce syntactically correct HTML based on the tags and their attributes.

In addition to simply translating VectorHTML into actual HTML, the decoder will also need to compare ground truth input to the output to

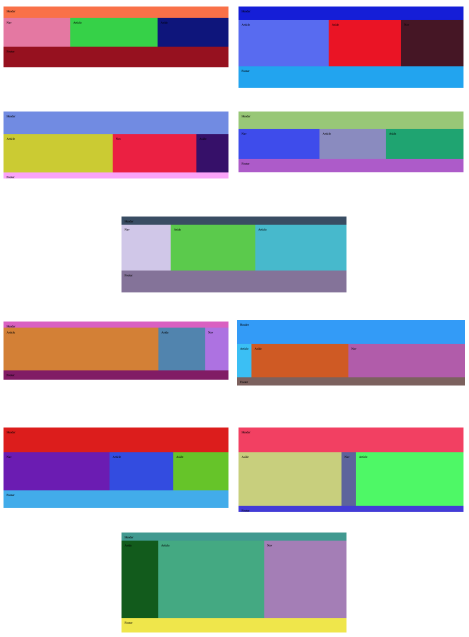

Figure 3. A larger sample of procedurally generated HTML pages

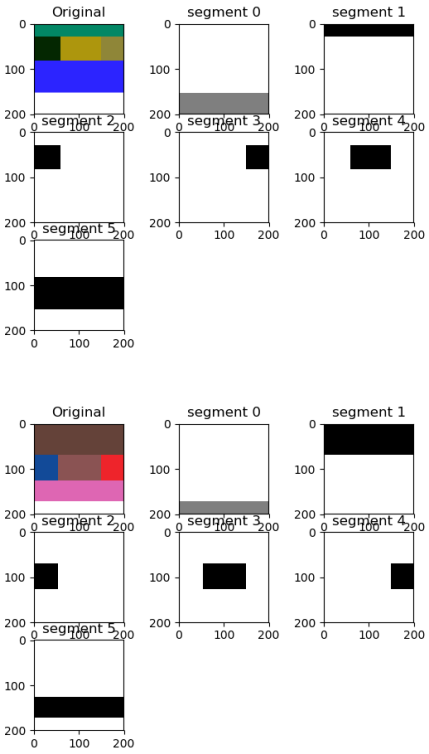

Figure 4. Examples of segmentation.

determine correctness. Given the simple nature of VectorHTML, comparison of input VectorHTML and output VectorHTML is extremely simple. We may also want to explore a more qualitative comparison of input image data and output image data. This is useful for situations when the VectorHTML does not match perfectly but the output image is qualitatively very similar. While initially our VectorHTML is so simple that VectorHTML inputs and outputs should match to achieve qualitative matches, as our inputs become more complex the Neural Net may not always produce perfectly matching VectorHTML but the VectorHTML will still produce a similar looking HTML page.

4.3. Segmentation and Proposal Generator

The goal for this section was to separate one section of the HTML page (one VectorHTML) from the rest so it could be used as an input into the neural network to predict the corresponding VectorHTML. Instead of predicting the whole HTML page in one go, we only predict sections of it, allowing for a more general algorithm. Due to the simplistic design of an HTML page when it has no text, we decided to use a K-means algorithm (MiniBatchKmeans) to find different regions in the image of the HTML page. First, we used Selenium [2] to take a screenshot of the generated HTML. Then, we used that image as an input to an image segmentation algorithm. That image segmentation algorithm uses MiniBatchKMeans to separate the image into six different categories based on the RGB values of the pixels. Six categories were currently chosen since our HTML generation algorithm produces an HTML with five different VectorHTMLs and a body VecorHTML.

The segments do not include color since color can be taken directly from the original image, thus we do not need to predict that. Worth noting is that we also order the segments in a specific order.

Segments are ordered according to rows (where the first pixel of the segment lies), if the rows are equal, then the segment with a lower column value will become first. This is because we want to be able to match the predicted VectorHTMLs with true VectorHTMLs for learning in the next step (regression) and testing our accuracy. The segmentation will also return the colors of each cluster center in order so we can use them when we want to regenerate the HTML page for testing.

Figure 4 also gives an example where the model is prone to errors. The segmentation on the right is correct, but the segmentation on the left side produces an incorrect ordering, where segment 3 and segment 4 are in the reverse order. There's no way of knowing if this happens since k-means algorithms are designed to just find the given amount of clusters and not the order of their appearance.

4.4. Vector Recognition with Regression algorithm

Before we use the segments as inputs to the regression algorithm, we lower the dimensions of it from (400,400,3) to (400,400) by giving each of the three colors (black, white, gray) present in the segments one integer value (0,1,2) instead of using RGB values. We also normalize the values in the VectorHTML according to the range of each possible parameter value to force each category in the VectorHTML to be equally weighted.

Next, for fitting, we need to match each segment with its actual VectorHTML. For that we use the order we get from the segmentation.

We decided to use a MultiOutputRegressor combined with GradientBoostingRegressor from the Sci-kit Learn [1] API, over a neural network due to the simplicity of our HTML code. Even though there's no doubt a neural network could have achieved similar or even better results, we felt like it was unnecessary to implement one if a regressor could do the job much faster and still achieve very good results.

4.5. Testing

In order to test out our model, we generate an HTML page, segment it into six segments and feed each segment to our model. Since each segment corresponds to a specific part of the HTML, we know for sure that some of the values in the VectorHTML have to be 0 for that segment. Therefore we force those values to be zero in the hopes that segmentation produced the segments in the correct order.
Then we need to denormalize the VectorHTML (since we predict normalized values). After that we can assign the colors back to each VectorHTML from the segmentation and generate HTML from the VectorHTML and see the result ourselves.

4.6. Evaluation

It would be hard to test out how off we are visually, thus we will generate test data, and use R^2 measure to calculate how accurate our model is on the test data.

$$R^2 = 1 - \frac{\sum_i^N (y_{i_{true}} - y_{i_{pred}})^2}{\sum_i^N (y_{i_{true}} - \frac{1}{N}\sum_j^N y_{j_{true}})^2}$$

The score will be 0, if the algorithm predicts always the expected value of y (meaning it ignores the input data). We want this to be as close to 1 as possible. Note that a negative score is also possible.

4.7. Results

The model trained on 500 examples pages (around 3000 segments) - some of which got rejected due to the failure to produce six segments achieved the score of 0.9525, which we are pleased with. On a test with 100 newly generated example pages (600 segments to predict) we achieved a score of 0.9581. This shows that the model we fit is generalizable to new data, which is exactly what we wanted to do.

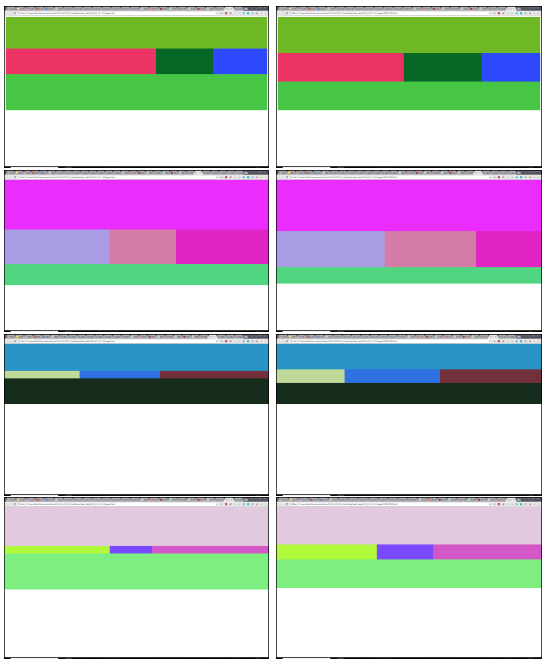

Figure 5. Examples of true HTML vs output of our model

A sample of outputs from our model (following the testing section) is demonstrated in Fig 5,

We can see that while the header, footer, and the body seem to be really accurate in size, the middle HTML elements seem slightly off. That's due to the fact that one of the parameters for the middle elements has dimensions of 15625. This makes relatively small errors seem big in the HTML. Being off by 5% results in a placement that's 780 indices away from the correct placement.

This model is clearly a simplistic representation of the real world task, where the pages are much more cluttered and the different sections might not be separated by color. We have also made some strong assumptions like the number of HTML elements on the web page and the way HTML elements are constructed. This was due to the fact that such work hasn't been done before and we wanted to have a proof-of-concept system that this could work. Unfortunately, due to time restrictions, we weren't able to make our system more complex.

5. Conclusions and Future Work

With this project we have learned how to build an end-to-end model that can take in an image of a HTML page and produce HTML code out of it. We are happy that we had a chance to apply different concepts learned in class (K-Means, importance of dimensions, vector representations) to this project. Using this model, we can use just an image as input to produce modifiable HTML code as output - once we have the VectorHTML representations of each segment, we can modify them as we please, allowing for user freedom.

For future work there are many ways to improve on this project. One of the limitations for us was the slow image capturing from of HTML pages using procedurally generated HTML. This did not allow us to generate significant training data which would have given us a more general model. Another improvement could be made to the segmentation step, where we are currently using K-Means. A segmentation algorithm that doesn't depend fully on color and would give a more reliable ordering could improve the model a lot.

The most important improvement to our model would be introducing more complex HTML inputs and outputs. This would require extending VectorHTML to include more sections (i.e. in addition to body, header, etc) and for each section to include additional attributes. Additionally, as the HTML becomes more complex it would no longer be enough to simply segment on the entire sections themselves, but instead segmentation on individual segments would likely be required. To account for these additions, we would likely need a more complex model for predicting VectorHTML, thus, using a neural network might be a good choice. Another idea would be to use other input to the VectorHTML predictor, maybe using some corner detector or other feature extractor.

References

[1] L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Mueller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler, R. Layton, J. VanderPlas, A. Joly, B. Holt, and G. Varoquaux. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pages 108–122, 2013.

[2] SeleniumHQ. Selenium. `https://github.com/SeleniumHQ/selenium/tree/master/py`.

[3] J. Wu, J. B. Tenenbaum, and P. Kohli. Neural scene de-rendering. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.