1. INTRODUCTION

The deep learning model is used to automate front-end development. We use the current deep learning algorithms, along with synthesized training data, to start exploring artificial front-end automation. In this project, we'll teach a neural network how to code a basic HTML and CSS website based on a picture of a design mockup.

1.1 Here's a ***quick overview*** of the process.
1) Give a design image to the trained neural network
2) The neural network converts the image into HTML markup
3) Render the output

1.2 We'll build the neural network in **three iterations.**
First, we'll make a bare minimum version to get a hang of the moving parts. The *second version*, HTML, will focus on automating all the steps and explaining the neural network layers. In the *final version*, Bootstrap, we'll create a model that can generalize and explore the LSTM layer.

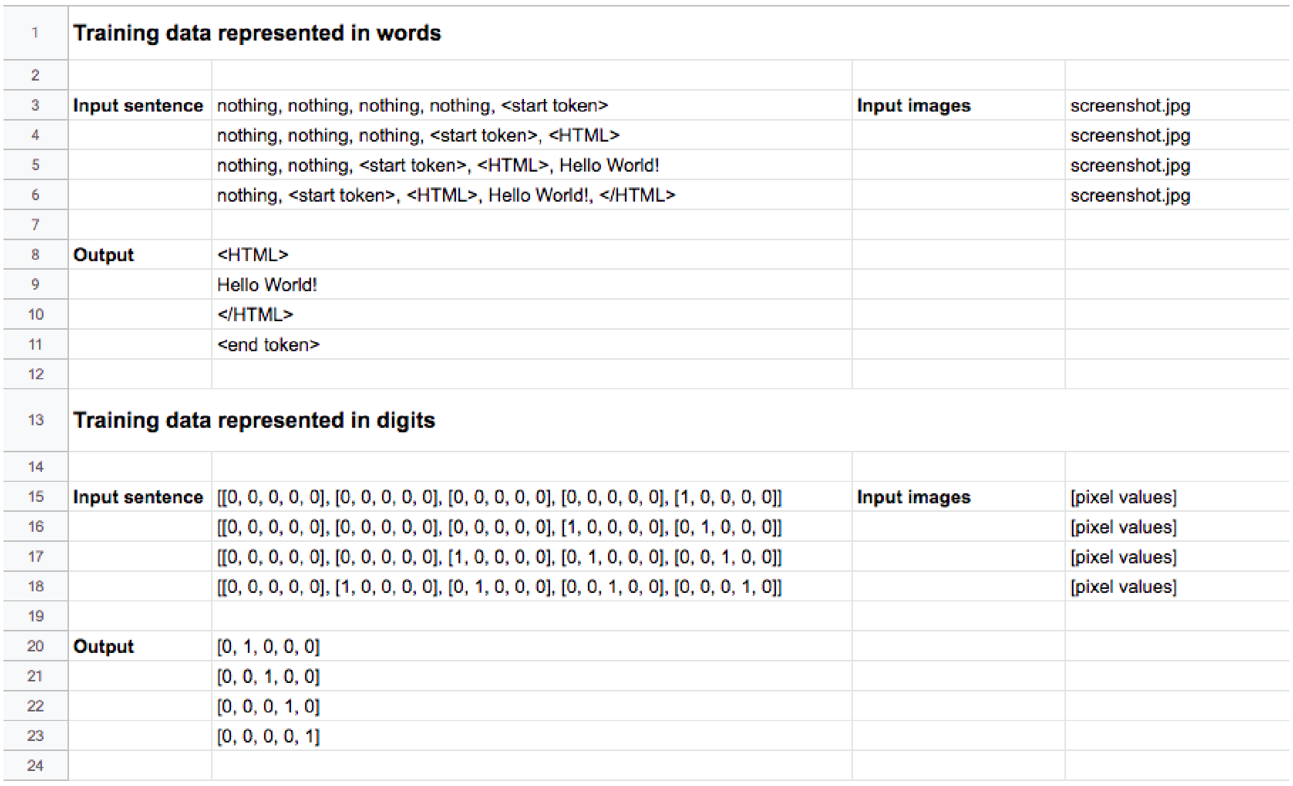
2. INSPIRATION

The models are based on Beltramelli's pix2code paper and Jason Brownlee's image caption tutorials. The code is written in Python and Keras, a framework on top of TensorFlow. The links for the same are :

3. CORE LOGIC

We want to build a neural network that will generate HTML/CSS markup that corresponds to a screenshot. When you train the neural network, you give it several screenshots with matching HTML.

It learns by predicting all the matching HTML markup tags one by one. When it predicts the next markup tag, it receives the screenshot as well as all the correct markup tags until that point.



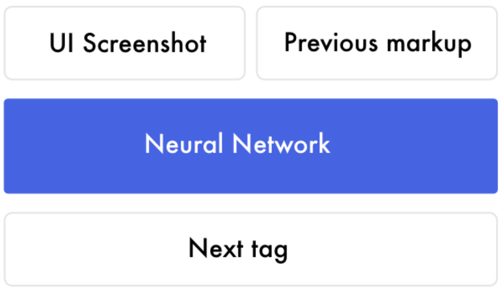
Training data represented in words

Input sentence	nothing, nothing, nothing, nothing, <start token>	**Input images**	screenshot.jpg
	nothing, nothing, nothing, <start token>, <HTML>		screenshot.jpg
	nothing, nothing, <start token>, <HTML>, Hello World!		screenshot.jpg
	nothing, <start token>, <HTML>, Hello World!, </HTML>		screenshot.jpg
Output	<HTML>		
	Hello World!		
	</HTML>		
	<end token>		

Training data represented in digits

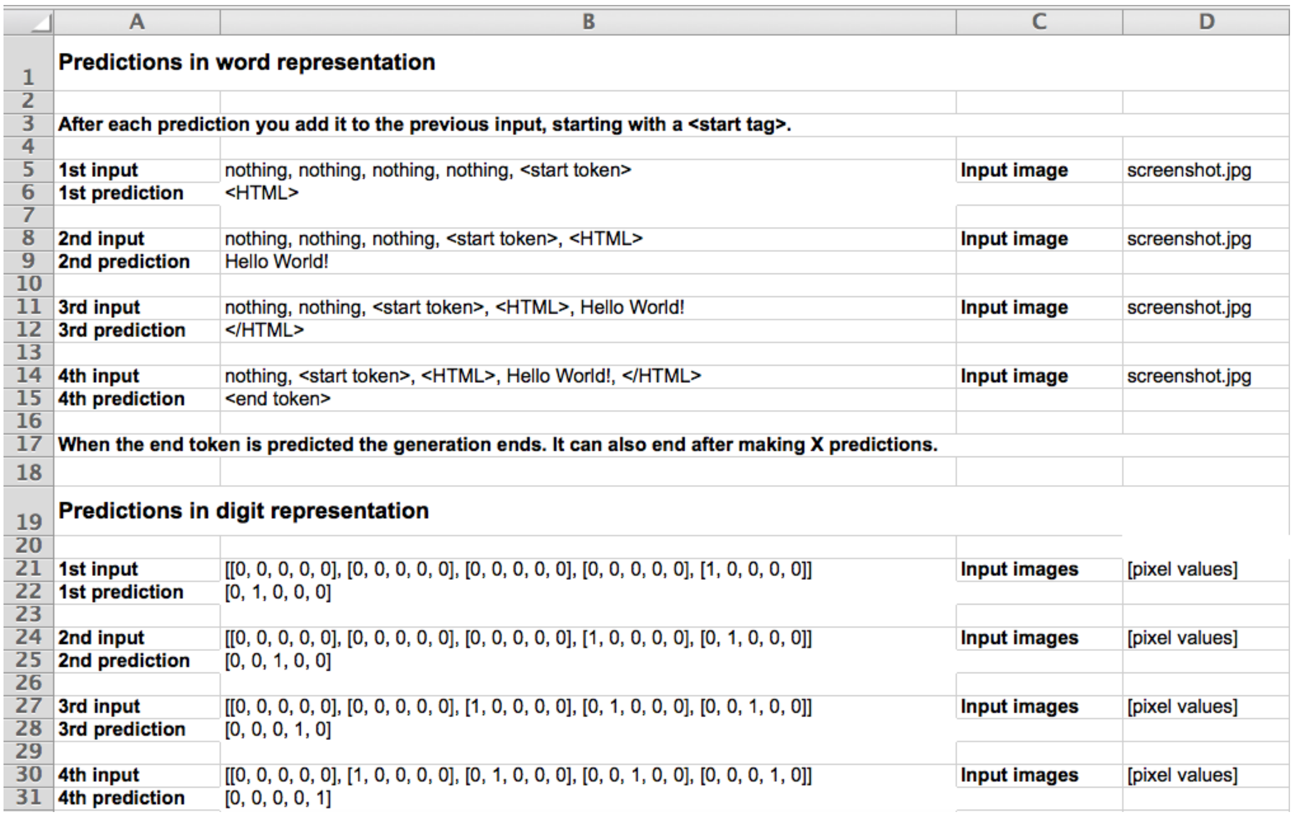
Input sentence	[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [1, 0, 0, 0, 0]]	**Input images**	[pixel values]
	[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [1, 0, 0, 0, 0], [0, 1, 0, 0, 0]]		[pixel values]
	[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [1, 0, 0, 0, 0], [0, 1, 0, 0, 0], [0, 0, 1, 0, 0]]		[pixel values]
	[[0, 0, 0, 0, 0], [1, 0, 0, 0, 0], [0, 1, 0, 0, 0], [0, 0, 1, 0, 0], [0, 0, 0, 1, 0]]		[pixel values]
Output	[0, 1, 0, 0, 0]		
	[0, 0, 1, 0, 0]		
	[0, 0, 0, 1, 0]		
	[0, 0, 0, 0, 1]		

Creating a model that predicts word by word is the most common approach today. Notice that for each prediction it gets the same screenshot. So if it has to predict 20 words, it will get the same design mockup twenty times.



Let's focus on the previous markup. **For instance**, we train the network to predict the sentence *"I can code."* When it receives *"I,"* then it predicts *"can."* Next time it will receive *"I can"* and predict *"code."* It receives all the previous words and only has to predict the next word.

The neural network creates features from the data. The network builds features to link the input data with the output data. It has to create representations to understand what is in each screenshot, the HTML syntax, that it has predicted. This builds the knowledge to predict the next tag. When you want to use the trained model for real-world usage, it's similar to when you train the model. The text is generated one by one with the same screenshot each time. Instead of feeding it with the correct HTML tags, it receives the markup it has generated so far. Then, it predicts the next markup tag. *The prediction is initiated with a "start tag" and stops when it predicts an "end tag" or reaches a max limit.*

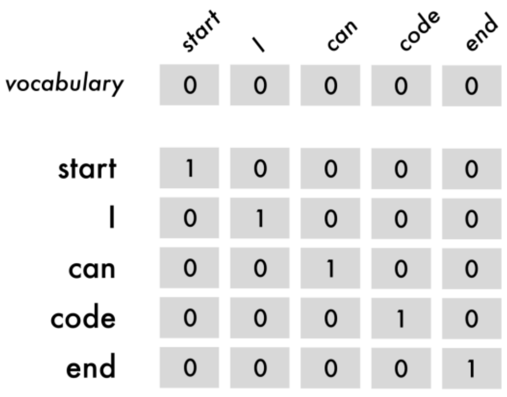


	A	B	C	D
1	**Predictions in word representation**			
2				
3	**After each prediction you add it to the previous input, starting with a <start tag>.**			
4				
5	**1st input**	nothing, nothing, nothing, nothing, <start token>	**Input image**	screenshot.jpg
6	**1st prediction**	<HTML>		
7				
8	**2nd input**	nothing, nothing, nothing, <start token>, <HTML>	**Input image**	screenshot.jpg
9	**2nd prediction**	Hello World!		
10				
11	**3rd input**	nothing, nothing, <start token>, <HTML>, Hello World!	**Input image**	screenshot.jpg
12	**3rd prediction**	</HTML>		
13				
14	**4th input**	nothing, <start token>, <HTML>, Hello World!, </HTML>	**Input image**	screenshot.jpg
15	**4th prediction**	<end token>		
16				
17	**When the end token is predicted the generation ends. It can also end after making X predictions.**			
18				
19	**Predictions in digit representation**			
20				
21	**1st input**	[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [1, 0, 0, 0, 0]]	**Input images**	[pixel values]
22	**1st prediction**	[0, 1, 0, 0, 0]		
23				
24	**2nd input**	[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [1, 0, 0, 0, 0], [0, 1, 0, 0, 0]]	**Input images**	[pixel values]
25	**2nd prediction**	[0, 0, 1, 0, 0]		
26				
27	**3rd input**	[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [1, 0, 0, 0, 0], [0, 1, 0, 0, 0], [0, 0, 1, 0, 0]]	**Input images**	[pixel values]
28	**3rd prediction**	[0, 0, 0, 1, 0]		
29				
30	**4th input**	[[0, 0, 0, 0, 0], [1, 0, 0, 0, 0], [0, 1, 0, 0, 0], [0, 0, 1, 0, 0], [0, 0, 0, 1, 0]]	**Input images**	[pixel values]
31	**4th prediction**	[0, 0, 0, 0, 1]		

4. **“HELLO WORLD” VERSION**

We’ll feed a neural network a screenshot with a website displaying “Hello World!” and teach it to generate the markup. *First, the neural network maps the design mockup into a list of pixel values. From 0–255 in three channels — red, blue, and green.*

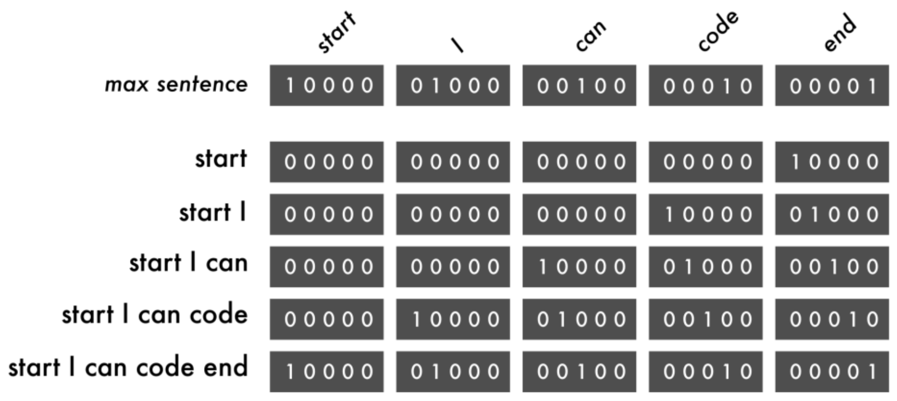
To represent the markup in a way that the neural network understands, I use one hot encoding. Thus, the sentence *“I can code”* could be mapped like the below. The start and end tags are cues for when the network starts its predictions and when to stop.



	start	I	can	code	end
vocabulary	0	0	0	0	0
start	1	0	0	0	0
I	0	1	0	0	0
can	0	0	1	0	0
code	0	0	0	1	0
end	0	0	0	0	1

For the input data, we will use sentences, starting with the first word and then adding each word one by one. The output data is always one word.

Sentences follow the same logic as words. They also need the same input length. Instead of being capped by the vocabulary, they are bound by maximum sentence length. If it’s shorter than the maximum length, you fill it up with empty words, a word with just zeros.



	start	I	can	code	end
max sentence	10000	01000	00100	00010	00001
start	00000	00000	00000	00000	10000
start I	00000	00000	00000	10000	01000
start I can	00000	00000	10000	01000	00100
start I can code	00000	10000	01000	00100	00010
start I can code end	10000	01000	00100	00010	00001

Words are printed from right to left. This forces each word to change position for each training round. This allows the model to learn the sequence instead of memorizing the position of each word.

In the hello world version, we use three tokens:

```
1. start
2. <HTML>  <center>  <H1>Hello World!</H1>    </center> </HTML>
3. end
```

A token can be anything. It can be a character, word, or sentence. Character versions require a smaller vocabulary but constrain the neural network. *Word level tokens tend to perform best.*

Output

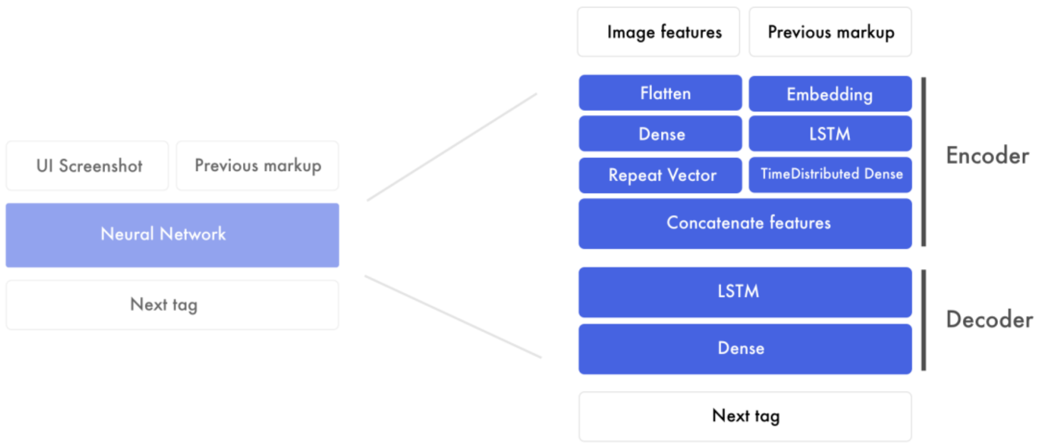
10 epochs: start start start
100 epochs: start <HTML><center><H1>Hello World!</H1></center></HTML> <HTML><center><H1>Hello World!</H1></center></HTML>
300 epochs: start <HTML><center><H1>Hello World!</H1></center></HTML> end

5. <u>HTML VERSION</u>

In this version, we’ll automate many of the steps from the Hello World model. This section will focus on creating a scalable implementation and the moving pieces in the neural network.

5.1 Overview

If we expand the components of the previous graphic it looks like this.



There are two major sections.

First, the **encoder**. This is where we create image features and previous markup features. Features are the building blocks that the network creates to connect the design mockups with the markup. At the end of the encoder, we glue the image features to each word in the previous markup.

The **decoder** then takes the combined design and markup feature and creates a next tag feature. This feature is run through a fully connected neural network to predict the next tag.

5.2 Design mockup features

Since we need to insert one screenshot for each word, this becomes a bottleneck when training the network. Instead of using the images, we extract the information we need to generate the markup.

The information is encoded into image features. This is done by using an already pre-trained convolutional neural network (CNN). The model is pre-trained on Imagenet. We extract the features from the layer before the final classification.

5.3 Markup features

In the *hello world version*, we used a *one-hot encoding* to represent the markup. In this version, we'll use a *word embedding* for the input and keep the one-hot encoding for the output.

The way we structure each sentence stays the same, but how we map each token is changed. One-hot encoding treats each word as an isolated unit. Instead, we convert each word in the input data to lists of digits. These represent the relationship between the markup tags.

The dimension of this word embedding is eight but often varies between 50–500 depending on the size of the vocabulary. The eight digits for each word are weights similar to a vanilla neural network. They are tuned to map how the words relate to each other.

This is how we start developing markup features. Features are what the neural network develops to link the input data with the output data.

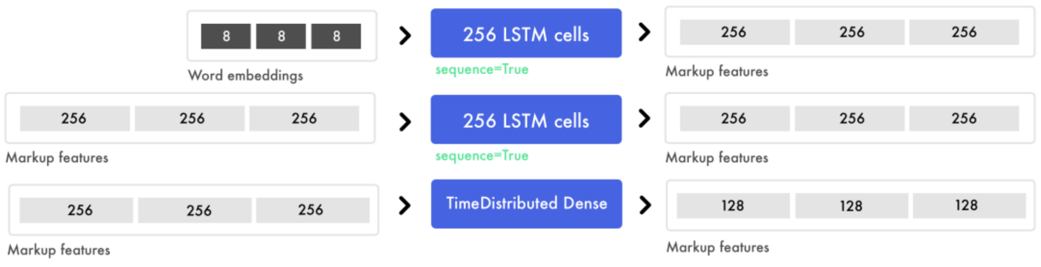
5.4 The Encoder

We'll take the word embeddings and run them through an LSTM and return a sequence of markup features. These are run through a Time distributed dense layer — think of it as a dense layer with multiple inputs and outputs.

In parallel, the image features are first flattened. Regardless of how the digits were structured, they are transformed into one large list of numbers. Then we apply a dense layer on this layer to form a high-level feature. These image features are then concatenated to the markup features.

5.4.1 Markup features

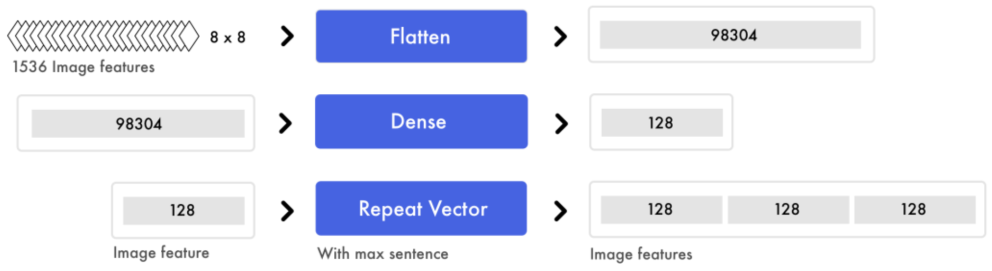
Here we run the word embeddings through the LSTM layer. All the sentences are padded to reach the maximum size of three tokens.



To mix signals and find higher-level patterns, we apply a TimeDistributed dense layer to the markup features. TimeDistributed dense is the same as a dense layer, but with multiple inputs and outputs.

5.4.2 Image features

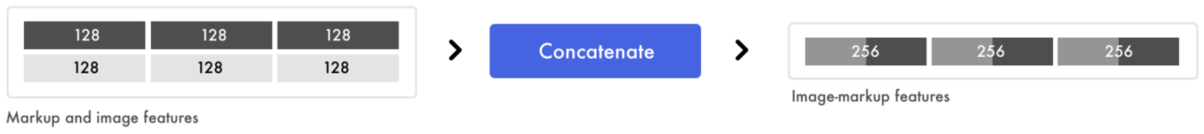
In parallel, we prepare the images. We take all the mini image features and transform them into one long list. The information is not changed, just reorganized.



In this case, we have three markup features. Thus, we end up with an equal amount of image features and markup features.

5.4.3 Concatenating the image and markup features

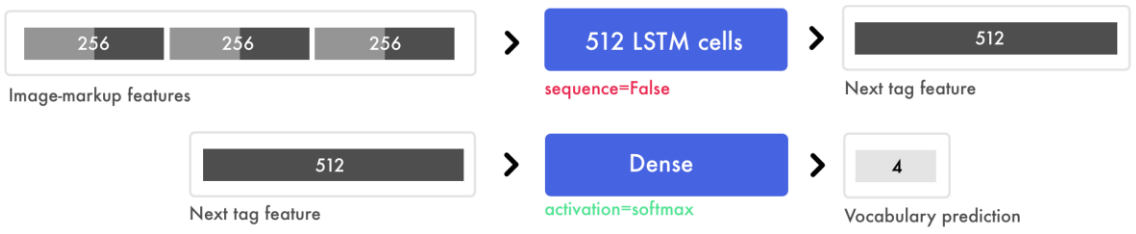
All the sentences are padded to create three markup features. Since we have prepared the image features, we can now add one image feature for each markup feature.



After sticking one image feature to each markup feature, we end up with three image-markup features. This is the input we feed into the decoder.

5.5 The Decoder

Here we use the combined image-markup features to predict the next tag. We use three image-markup feature pairs and output one next tag feature.



Note that the LSTM layer has the sequence set to false. Instead of returning the length of the input sequence, it only predicts one feature. In our case, it's a feature for the next tag. It contains the information for the final prediction.

5.6 The final prediction

The dense layer works like a traditional feedforward neural network. It connects the 512 digits in the next tag feature with the 4 final predictions. Say we have 4 words in our vocabulary: start, hello, world, and end.

The vocabulary prediction could be [0.1, 0.1, 0.1, 0.7]. The softmax activation in the dense layer distributes a probability from 0–1, with the sum of all predictions equal to 1. In this case, it predicts that the 4th word is the next tag. Then you translate the one-hot encoding [0, 0, 0, 1] into the mapped value, say "end".

6. BOOTSTRAP VERSION

In our final version, we'll use a dataset of generated bootstrap websites from the pix2code paper. By using Twitter's bootstrap, we can combine HTML and CSS and decrease the size of the vocabulary.

We'll enable it to generate the markup for a screenshot it has not seen before.

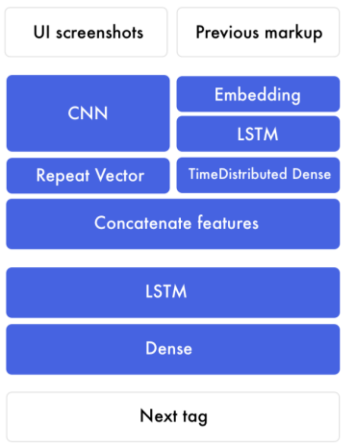
Instead of training it on the bootstrap markup, we'll use 17 simplified tokens that we then translate into HTML and CSS. The dataset includes 1500 test screenshots and 250 validation images. For each screenshot there are on average 65 tokens, resulting in 96925 training examples.

By tweaking the model in the pix2code paper, the model can predict the web components with 97% accuracy (BLEU 4-ngram greedy search).

6.1 An end-to-end approach

Extracting features from pre-trained models works well in image captioning models. The pre-trained models have not been trained on web data and are customized for classification.

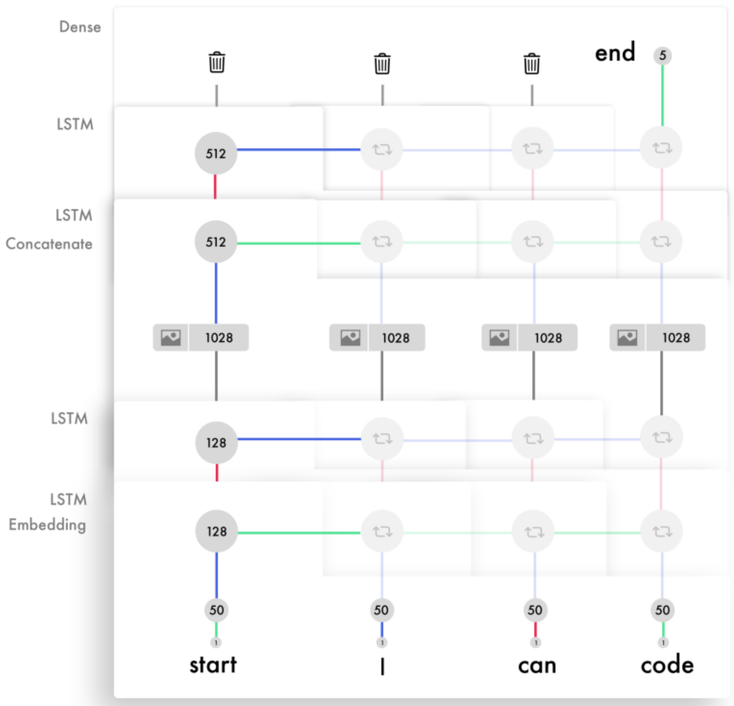
In this model, we replace the pre-trained image features with a light convolutional neural network. Instead of using max-pooling to increase information density, we increase the strides. This maintains the position and the color of the front-end elements.



There are two core models that enable this: **convolutional neural networks** (CNN) and **recurrent neural networks** (RNN). The most common recurrent neural network is long-short term memory (LSTM).

6.2 Understanding timesteps in LSTMs

LSTMs are made for input with timesteps. It’s a neural network customized for information in order. If you unroll our model it looks like the image below. For each downward step, we keep the same weights. We apply one set of weights to the previous output and another set to the new input.



The weighted input and output are concatenated and added together with an activation. This is the output for that timestep. Since we reuse the weights, they draw information from several inputs and build knowledge of the sequence.

6.3 Understanding the units in LSTM layers

Each unit in the LSTM layer learns to keep track of different aspects of the syntax. Below is a visualization of a unit that keeps tracks of the information in the row div. This is the simplified markup we are using to train the bootstrap model.

Each LSTM unit maintains a cell state. Think of the cell state as the memory. The weights and activations are used to modify the state in different ways. This enables the LSTM layers to fine-tune which information to keep and discard for each input. In addition to passing through an output feature for each input, it also forwards the cell states, one value for each unit in the LSTM.

6.4 Test accuracy

It's tricky to find a fair way to measure the accuracy. Say we compare word by word. If our prediction is one word out of sync, we might have 0% accuracy. If we remove one word which syncs the prediction, we might end up with 99/100.

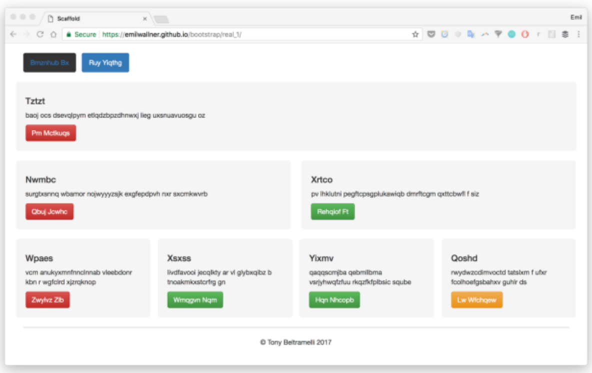
We thus used the BLEU score. It breaks the sentence into four n-grams, from 1–4 word sequences. In the below prediction "cat" is supposed to be "code."

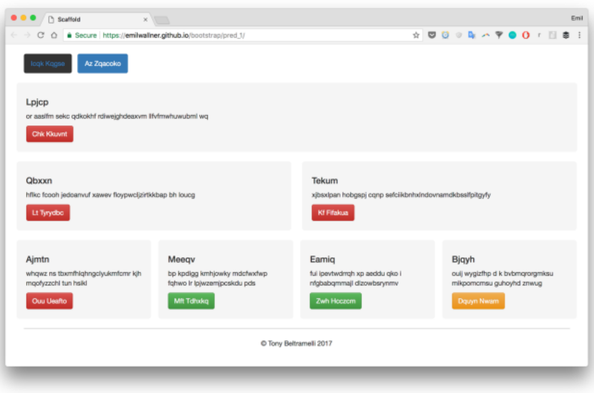
1-gram:	<start>, I, can, cat, <end> (4/5)
2-gram:	<start> I, I can, can cat, cat <end> (2/4)
3-gram:	<start> I can, I can cat, can cat <end> (1/3)
4-gram:	<start> I can cat, I can cat <end> (0/2)

To get the final score, we multiply each score with 25%, (4/5) * 0.25 + (2/4) * 0.25 + (1/3) * 0.25 + (0/2) * 0.25 = 0.2 + 0.125 + 0.083 + 0 = 0.408 . The sum is then multiplied with a sentence length penalty. Since the length is correct in our example, it becomes our final score.

6.5 Output

These are the few snapshots of the orginal and predicted web pages by the model.


Original


Prediction