CS 231N Project Report
Automatic Code Generation from UI Screenshots

Mustafa Abdool
Stanford
San Francisco, CA
moose878@stanford.edu

Himanshu Bhandoh
San Francisco, CA
hbhandoh@stanford.edu

Vincent Ying
Stanford
Palo Alto, CA
vhying@stanford.edu

Abstract

Recent advances in use of neural networks with image understanding and natural language processing have made it possible to jointly learn image and text representations. This has enabled the generation of code from a single screenshot or sketch of an application. [2] We investigate alternatives to the use of standard CNN and LSTMs within the encoder-decoder architecture, including the use of Capsule Networks [4], the use of bi-directional LSTM, and different variations on transfer learning. We show that these approaches can lead to significant improvements from the standard baseline architecture. [9]

1. Introduction

In many modern companies, the bottleneck in the design process involves transforming sketches created by designers into frontend code which can then be rendered in the browser. This whole process can be made more efficient by creating a model which can automatically generate code from a screenshot or sketch, ultimately decreasing the time of each iteration cycle. Such a model for this problem has recently been prototyped by the design team at Airbnb. [8]

The input to the model is a screenshot of the user interface and the output a sequence of tokens in the domain specific frontend language, which represent the code used to generate that specific user interface [2]. Since the output is of variable length and we want to learn joint image and text representations we draw heavily on architectures used in the image captioning domain. Concretely, we learn a representation for both the input image (using CNNs) and the previous sequence of tokens (using RNNs), and then combine both representations to generate the predicted output sequence.

For our project, we will investigate alternatives to the use of standard CNN's and LSTMs within the encoder-decoder architecture. Specifically we will investigate substitution of the encoder CNN with Capsule Networks [4], the addition of bi-directional LSTM within the decoder, and various transfer learning schemes. Our overall goal is to gain a deeper understanding of the encoder-decoder architecture and the contribution of its underlying components within the context of multimodal learning.

2. Related Work

2.1. Screenshot to Code

One of the first attempts at generating domain specific tokens from images without the use of prior heuristics can be found in **Beltramelli** [2]. This approach uses CNNs to learn a representation of a raw image in addition to RNNs to encode the generated text. While this formulation of the problem was quite novel, the components of each subsystem were fairly simplistic. For example, the LSTM network used for text embedding did not use the more recent attention mechanisms to help capture long term dependencies, which should be useful in generating code that must follow a certain structure (such as the opening and closing html tag pairs).

2.2. Image Captioning

The task of automatic code generation from an image is similar in concept to that of image captioning. Recent work in image captioning have enabled deep neural networks to describe objects and their relationships within an image with textual descriptions. This is accomplished with the basic encoder-decoder architecture to generate multimodal embeddings. [1] The encoder is comprised of a CNN and a RNN to jointly learn images and text, and the decoder utilizes a second RNN to predict text solely from a given image.

Vinyals et al. performed image captioning with the use of a CNN encoder for image ingestion and an LSTM RNN decoder for caption generation. To improve performance of the decoder, beam search was utilized to predict the final output sequence. [6]

Karpathy et al. developed a model that generates text descriptions of objects from image/sentence pairs. They combined RCNNs (region based CNN) with BRNN (bidirectional Recurrent Neural Networks) to learn over both image and textual modalities. The sentences were treated as weak labels, where contiguous sentence segments correspond to an unknown location in the image. This multimodal RNN architecture was then utilized to generate descriptions of those distinct, unknown image regions. [3]

3. Dataset and Features

For this problem, we will be utilizing the dataset created in the pix2code paper. The dataset is comprised of 3 subsets for iOS UI (Storyboard), Android UI (XML), and webUI (HTML/CSS). Each subset has 3500 screenshot image/code pairs. The iOS subset is comprised of 760 x 1340 RGB images and Storyboard markup. The Android subset is comprised of 688 x 1070 RGB images and XML markup. The webUI subset is comprised of varying sized RGB images and HTML/CSS markup. [2]

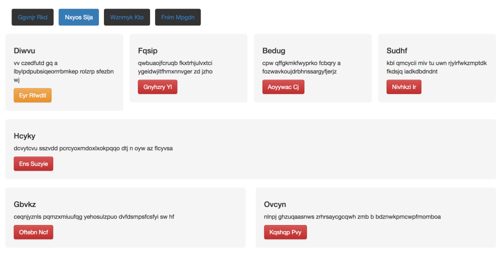

Ggvnjr Rkd
Nxyos Sija
Wznmyk Kto
Fnim Mpgdn
Diwvu
vv czedfutd gq a lbylpdpubsiqeorrrbmkep rolzrp sfezbn wj
Eyr Rfwdtl
Fqsip
qwbuaojfcruqb fkxtrhjulvxtci ygeidwjitfhmxnnvger zd jzho
Gnyhzry Yl
Bedug
cpw qffgkmkfwyprko fcbqry a fozwavkoujdrbhnssargyfjerjz
Aoyywac Cj
Sudhf
kbl qmcycii miv tu uwn rjylrfwkzmptdk fkdsjq iadkdbdndnt
Nivhkzi Ir
Hcyky
dcvytcvu sszvdd pcrcyoxmdoxlxokpqqo dtj n oyw az flcyvsa
Ens Suzyie
Gbvkz
ceqnjyznls pqmzxmiuufqg yehosulzpuo dvfdsmpsfcsfyi sw hf
Oftebn Ncf
Ovcyn
nlnpj ghzuqaasnws zrhrsaycgcqwh zmb b bdznwkpmcwpfmomboa
Kqshqp Pvy

Figure 1. Sample screenshot input to model

```
<START> header { btn-inactive , btn-active , btn-inactive , btn-inactive } row { quadruple { small-title , text , btn
-orange } quadruple { small-title , text , btn-green } quadruple { small-title , text , btn-red } quadruple { small-t
itle , text , btn-red } } row { single { small-title , text , btn-green } } row { double { small-title , text , btn-o
range } double { small-title , text , btn-green } } <END>
```

Figure 2. Sample sequence of domain specific language tokens to generate screenshot

4. Methods

4.1. Baseline Methodology

We train an encoder-decoder network with one portion of the encoder input being generated from the output of the raw image fed through a CNN and the other portion as a sequence of outputs from an LSTM network with the current sequence of tokens. The CNN network uses a simple sequential model comprising of convolutional, fully-connected, and dropout layers. The outputs from these two subnetworks are then concatenated together, fed into another LSTM network, and then a dense fully connected layer which outputs the final probability distribution over all tokens.

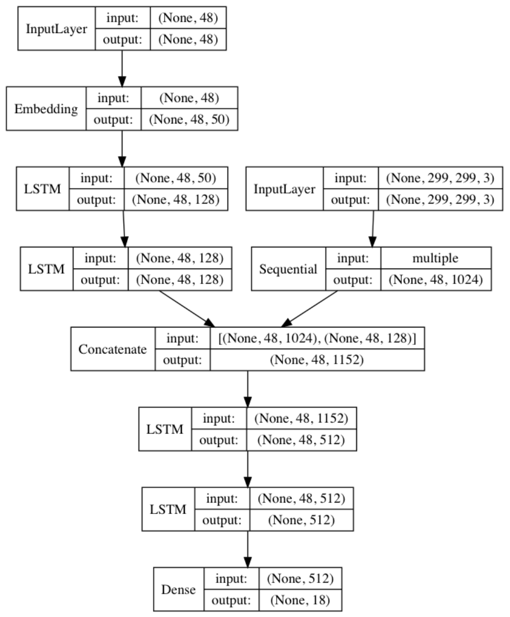

InputLayer | input: (None, 48) | output: (None, 48)
Embedding | input: (None, 48) | output: (None, 48, 50)
LSTM | input: (None, 48, 50) | output: (None, 48, 128)
InputLayer | input: (None, 299, 299, 3) | output: (None, 299, 299, 3)
LSTM | input: (None, 48, 128) | output: (None, 48, 128)
Sequential | input: multiple | output: (None, 48, 1024)
Concatenate | input: [(None, 48, 1024), (None, 48, 128)] | output: (None, 48, 1152)
LSTM | input: (None, 48, 1152) | output: (None, 48, 512)
LSTM | input: (None, 48, 512) | output: (None, 512)
Dense | input: (None, 512) | output: (None, 18)

Figure 3. Baseline architecture of the encoder decoder network

The structure of the LSTM decoder can be expressed mathematically in the following manner

$$p = \text{CNN}(I) \tag{1}$$

$$q_t = \text{LSTM}(x_t) \tag{2}$$

$$r_t = (q, p_t) \tag{3}$$

$$y_t = \text{softmax}(\text{LSTM}'(r_t)) \tag{4}$$

At training time, we feed in input data consisting of the raw UI image along with a sequence of previous tokens (max length of 48) and require the network to predict the next token in the sequence. Then, we use softmax loss between the output probability distribution over the all tokens and the ground truth of the next token.

The multiclass log loss used can be formulated as

$$L(I, X) = -\sum_{t=1}^{T} x_{t+1} \log(y_t) \tag{5}$$

4.2. Architecture Substitution

4.2.1 Capsule Network

In general, convolutional layers are not spatially invariant with respect to translation, scale, and rotation. Instead of using a standard CNN encoder, we will substitute a Capsule Network to observe any change in performance with a model that is spatially invariant with respect to the input, in our case the UI components in each screenshot.

Capsule Networks, recently introduced by Sabour et al., are composed of groups of neurons (*fig. 4*) that are relatively invariant to spatial transformations through the use of transformation matrices. [4]

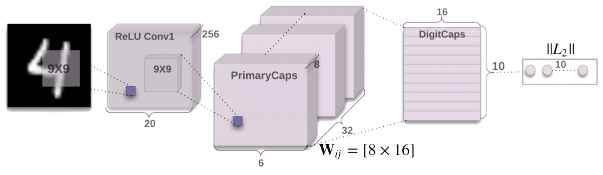


Figure 4. Capsule Network For Digit Classification

The output vector length of a capsule is made to represent the probability that the entity represented by the capsule is present in the current input through the non-linear squashing function below,

$$\boldsymbol{v}_j = \frac{||\boldsymbol{s}_j||^2}{1 + ||\boldsymbol{s}_j||^2}\left(\frac{\boldsymbol{s}_j}{||\boldsymbol{s}_j||}\right) \tag{6}$$

where capsule j has the vector output $\boldsymbol{v}_j$ and a total input of $\boldsymbol{s}_j$.

From the second layer of capsules, the total input of capsule $\boldsymbol{s}_j$ is the weighted sum over all prediction vectors, $\hat{\boldsymbol{u}}_{j|i}$, from the output of the previous capsule layer $\boldsymbol{u}_i$ and a weight matrix $\boldsymbol{W}_{ij}$.

$$\boldsymbol{s}_j = \sum_i c_{ij} \hat{\boldsymbol{u}}_{j|i}, \quad \hat{\boldsymbol{u}}_{j|i} = \boldsymbol{W}_{ij} \boldsymbol{u}_i \tag{7}$$

The c_{ij} in above equation is routing coefficient. This coefficient is governed by the softmax equation, where the initial logits b_{ij} are the log prior probabilities that capsule i should be coupled to capsule j.

$$c_{ij} = \frac{\exp(b_{ij})}{\sum_k \exp(b_{ik})} \tag{8}$$

4.2.2 Bi-directional LSTM

Bi-directional LSTMs involve duplication of the LSTM layer in the network, so that there are two layers side-by-side. The original input sequence remains the input to the first layer and a reversed copy of the input sequence is fed in as input to the second layer. This intuitively mirrors human understanding of language, as tokens on a webpage have different meanings based on context with surrounding tokens.

4.3. Transfer Learning

Transfer learning is a common paradigm when it comes to vision problems. We investigate several variations on transfer learning for our task. For all variations, we use the InceptionV3 network with weights trained on the ImageNet dataset [5]. We choose to use the InceptionV3 network due to its high accuracy on ImageNet, while having a relatively low number of parameters compared to other networks with similar performance (such as InceptionResNetV2).

4.4. Basic transfer learning

This approach follows the conventional method of freezing all weights in the InceptionV3 network, removing the last fully connected layer and replacing it with a trainable fully connected layer. In order to reduce the number of parameters for the new trainable layer, *global average pooling* was used on the output of the InceptionV3 network (after removing the topmost layer used for prediction)

4.5. Fine tuning approach

This method is an extension of the basic approach and also commonly used in transfer learning scenarios. Once we had trained the basic transfer learning to a point where the loss had almost converged we then unfroze some of the earlier layers in the base InceptionV3 network and continued training. When doing this fine-tuning, we used a higher learning rate since we suspected the features learned in the last few layers of the InceptionV3 network were probably not as relevant so we wanted the network to modify them rather quickly. This is because the UI screenshots are quite different than the ImageNet dataset.

4.6. Transfer Learning Concatenation

4.6.1 InceptionV3 Final (Concatenate final output of InceptionV3 with custom CNN)

As previously mentioned, one issue with transfer learning is that the IncpetionV3 network was trained on a fairly different dataset. However, the information contained in the final layer should still be useful for prediction. In this approach, we apply global average pooling to the final layer of InceptionV3 (after removing the fully connected layer) and concatenate it with the output of the custom CNN network found in the baseline architecture. The motivation behind this was to allow the weights of the custom network to learn information not present in the InceptionV3 features, resulting in a more robust predictor overall.

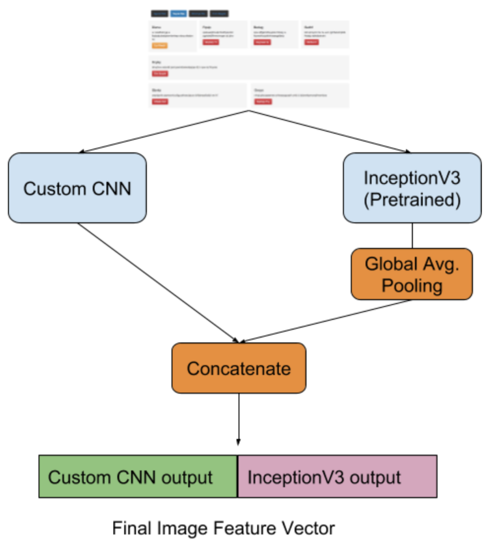


Figure 5. Constructing image embedding using concatenation with InceptionV3 output and custom network

4.6.2 InceptionV3 Intermediate (Concatenate intermediate outputs of InceptionV3 with custom CNN)

In ”Good Practice in CNN Feature Transfer”, the authors explore the approach of extracting features from multiple **intermediate** layers of the pretrained network and concatenating them together to form a new feature vector. [10] We use this approach in conjunction with a custom trained CNN network. The outputs from all intermediate layers are concatenated with the custom network to produce image features. The intuition behind this approach versus the concatenation of the final output of the InceptionV3 network is that the feature extractors represented in the earlier layers of the InceptionV3 network should be *generalizable* and adapted (such as edge detectors or color detectors) towards our task, making it more relevant to the final predictions.

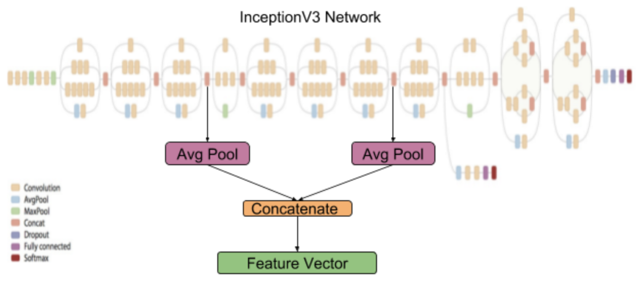


Figure 6. Extracting features from intermediate layers of Inception V3. These are concatenated with output of custom CNN to produce the final image embedding.

5. Experimental Details

5.1. Hyperparameters

For all experiments, the 3,500 training examples of screenshots and generated code were split into 500 examples for the test set and 3,000 for training. During training, 10 percent of the examples were used as a validation set.

The RMSProp optimizer with a learning rate of 0.0001 was found to have the best performance. However, this was changed for the fine-tuning approach in the transfer learning experiments. The learning rate was increased to 0.001 after unfreezing the weights of the top layers in the InceptionV3 network. For the fine-tuning approach that involved concatenating the custom network with the final InceptionV3 output, the learning rate was also increased to 0.001 when the training loss started to plateau.

A batch size of 3 was used for most experiments, which was the largest batch size allowed due to memory constraints. Only for the Capsule Network was the batch size decreased to 1. Due to the increased size of the Capsule Network this was the only batch size that did not result in memory errors.

6. Results and Discussion

6.1. Evaluation Metrics

For evaluation metrics, we used BLEU score (4-grams) between the predicted and output sequence mainly because it is more robust to small errors in text generation (such as the ground truth and predicted output differing by one token). BLEU score is also widely used as an evaluation measure in image captioning models, which is quite similar to our problem. In addition, we used greedy search when predicting the output sequence of tokens to reduce computational complexity.

Results for each experiment we have undertaken are given in the following table.

Architecture	Train Loss	Val Loss	BLEU
Baseline	0.0352	0.0366	0.8988
CapsNet	0.0172	0.0372	0.9116
Bi-direction LSTM	0.1017	0.0989	0.9116
Basic Transfer Learning	0.4674	0.2184	0.7979
InceptionV3 Final	**0.0084**	**0.0097**	**1.0000**
InceptionV3 Intermediate	0.0431	0.0453	0.9795
Fine Tuning	0.1912	0.199	0.8343

The different models also converged to the minimum validation loss at different epochs under differing training durations.

Architecture	Min loss epoch	mins / epoch
Baseline	39	12
CapsNet	48	154
Bi-direction LSTM	37	42
Basic Transfer Learning	6	20
InceptionV3 Final	17	30
InceptionV3 Intermediate	14	23
Fine Tuning	2	25

6.2. Baseline

For the baseline, we trained our model for 40 epochs on all three data subsets, and obtained a minimum validation loss of 0.0366 with a BLEU score of 0.8988. The loss curve is given in Figure 2.

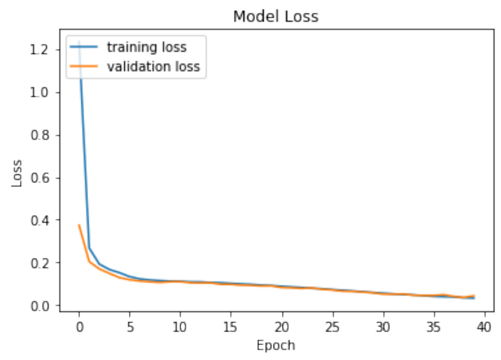
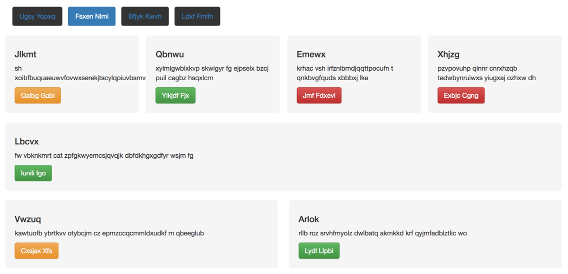


Figure 7. Training and Validation Loss for Baseline

We noticed that the most common mistakes in the baseline network are due to its inability to distinguish between the color of various components, as demonstrated in the example below. While the baseline network tends to usually get the structure of the components correct, it struggles with correctly predicting the color of buttons (note that the text is randomly generated and is not considered as misprediction).



Jlkmt
sh xoibfbuquaeuwvfovwxserekjtscylqpiuvbsmv

Qbnwu
xylmlgwblxkvp skwigyr fg ejpselx bzcj puil cagbz hsqxlcm

Emewx
krhac vsh irfznibmdjqqttpocufn t qnkbvgfquds xbbbxj lke

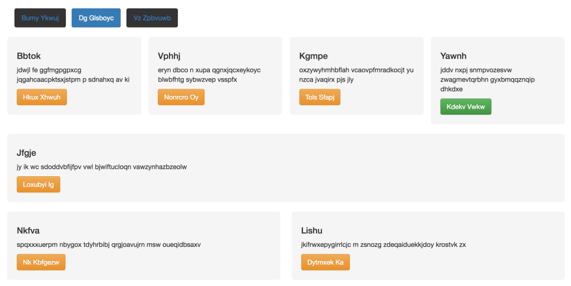
Xhjzg
pzvpovuhp qlnnr cnrxhzqb tedwbynruiwxs yiugxaj ozhxw dh

Lbcvx
fw vbknkmrt cat zpfgkwyemcsjqvqjk dbfdkhgxgdfyr wsjm fg

Vwzuq
kawtuofb ybrtkvv otybcjm cz epmzccqcmmldxudkf m qbeeglub

Arlok
rllb rcz srvhfmyolz dwlbatq akmkkd krf qyjmfadblztlic wo

Figure 8. Ground truth UI



Bbtok
jdwjl fe ggfmgpgpxcg jqgahcaacpktsxjstpm p sdnahxq av ki

Vphhj
eryn dbco n xupa qgnxjqcxeykoyc blwbfhtg sybwzvep vsspfx

Kgmpe
oxzywyhmhbflah vcaovpfmradkocjt yu nzca jvaqirx pjs jly

Yawnh
jddv nxpj snmpvozesvw zwagmevtqrbhn gyxbmqqznqip dhkdxe

Jfgje
jy ik wc sdoddvbfijfpv vwl bjwiftucloqn vawzynhazbzeolw

Nkfva
spqxxxuerpm nbygox tdyhrbibj qrgjoavujrn msw oueqidbsaxv

Lishu
jkifrwxepygirrlcjc m zsnozg zdeqaiduekkjdoy krostvk zx

Figure 9. Predicted UI

6.3. Capsule Network

The substitution of the CNN encoder with a capsule network increased training time from around 10 min per epoch to more than 2.5 hrs. To Capsule Network has a minimum validation loss at 48 epochs at 0.0372. Due to the model size, the BLEU score could not be calculated. It is also interesting to note that the training time for the Capsule Network is much slower than any of the other methods due to the complex routing algorithm used.

6.4. Bi-directional LSTM

The bi-directional LSTM seems to achieve a slightly better BLEU score (0.91) than the baseline model (0.89) over a similar number of epochs, but it took almost three times as long to train. This is to be expected since we effectively doubled the number of LSTMs.

6.5. Basic Transfer Learning

Overall, the basic transfer learning approach did not perform as well as the baseline method. This was expected as the baseline requires training an entire CNN from scratch, which means it can better adapt to the domain of GUI screenshots unlike the frozen InceptionV3 network. However, one interesting point to note is that the basic transfer learning approach converges to the minimum validation loss much quicker than the baseline (in only 6 epochs vs. 39). This is probably due to the order of magnitude reduction in the number of trainable parameters (from around 143m in the baseline to 8m in basic transfer learning).

6.6. Basic transfer learning with fine tuning

Fine-tuning improved the performance of the basic transfer learning approach, but it was still slightly less than the baseline method. Again, we suspect this is largely due to the fact that the baseline method is more powerful as it involves training an entire CNN from scratch.

There was also a delicate balance between the number of layers to fine-tune in the InceptionV3 network. We found that if too many layers are tuned at once, then overfitting would start to occur only after a few epochs. This would likely be due to the number of increased parameters. We attempted to mitigate this with various strategies, such as adding a dropout layer after the fully connected layer preceding the InceptionV3 network. Overall, we found that fine-tuning the layers in the last two inception modules of the InceptionV3 network gave the best performance.

6.7. Inception V3 Final

Concatenating the output of the final layer of InceptionV3 network (after removing the last fully connected layer) with the output of a custom CNN resulted in the best performance overall, both in terms of validation loss and BLEU score. While this took more training time (in terms of epochs) to reach the minimum validation loss than other transfer learning variations, it still required less training time than the baseline. This is most likely due to the fact that the overall network was able to leverage information from the pretrained InceptionV3 network (probably more global features which might give an indication of color, for example).

The resulting performance of this experiment exceeding the baseline was consistent with our expectations. The concatenation of features derived from the InceptionV3 network increased the information available to the network, thereby increasing performance as expected.

6.8. Inception V3 Intermediate

Concatenating the intermediate outputs of some layers in the InceptionV3 network with the output of the custom CNN architecture achieved a BLEU score that is very close to the optimal value, even though it was slightly worse than the concatenation with the final output of pre-trained InceptionV3. This was somewhat surprising as we initially thought that earlier layers would learn more general features to better improve performance in this problem domain. We suspect that since features are only extracted from earlier layers, the network might still be missing important global or aggregate information that is only represented in the final layer of the pre-trained network.

Furthermore, it is quite likely that there may be some combination of features from intermediate layers that performs even better. However, it is computationally infeasible to enumerate all the possible combinations of intermediate layers to choose. Out of all the combinations we tried, the best performance was achieved when extracting features from the layers after the 3rd, 4th, and 5th Inception modules in the InceptionV3 architectures.

7. Conclusion

We tried substituting and augmenting various network components to improve the task of automated code generation from screenshots. Our experiments included the use of capsule networks, bi-directional LSTMS, transfer learning with fine tuning and concatenation. We propose the use of InceptionV3-CNN concatenation based encoder-decoder network, since it gives the best performance on the BLEU score metric for automated screenshot to frontend code generation.

7.1. Future Work

For future work, a modularized approach to the addition of various encoders for image ingestion should improve performance. A separate encoder for image features, colors, fonts, and spatial layout each should generate more accurate prediction models.

To improve the performance of the textual (code) component of the input, an attentional mechanism can be added to the LSTM to focus on important token sequences in the distant past.

Another direction to improve the performance of the model would be the utilization of GANs for the generation of UI screenshots to vary the input dataset. Thinking even more broadly, GANs could even be used to generate completely new types of user interfaces.

In addition, our model can be applied to alternative problem domains that involve UI components generated from other markup and programming languages, e.g. Latex, Javascript, and Python. It would be interesting to investigate how well our best model generalizes to other domains.

8. Appendices

8.1. Network Architecture Details

8.2. Capsule Network

The Capsule Network was comprised of 3 layers a convolutional layer, primary capsule layer, and a feature capsule layer.

1. Conv. layer with with 128 9x9 filters, stride of 1.
2. Primary capsule layer with dimension of 6 with 9x9 filters, stride of 2 for 6 channels.
3. Feature capsule layer with 6 capsules, each of dimension of 6 and routing of 3.

8.3. Custom CNN Architecture

For all convolutional layers ReLU activation was used along with a padding type of *same*.

1. Conv. layer with with 16 3x3 filters, stride of 1
2. Conv. layer with with 16 3x3 filters, stride of 2
3. Conv. layer with 32 3x3 filters, stride of 1
4. Conv. layer with 32 3x3 filters, stride of 2

5. Conv. layer with 64 3x3 filters, stride of 1
6. Conv. layer with 64 3x3 filters, stride of 2
7. Conv. layer with 128 3x3 filters, stride of 1
8. Fully connected layer of size 1024
9. Dropout layer with dropout probability 0.3
10. Fully connected layer of size 1024
11. Dropout layer with dropout probability 0.3

8.4. Encoder and Decoder Architecture

For the encoder that takes as input the current context (sequence of tokens in domain language), the number of units (dimension of output) was 128. For the decoder (which takes as input the concatenation of the image feature vector and the output of the encoder), the number of units was 512.

9. Contributions and Acknowledgements

Each group member contributed equally to the successful completion of this project.

We would like to acknowledge Justin Johnson and the advisors of CS231N for their guidance at the initiation of this project. We would also like to acknowledge Emil Wallner's work on top of the initial Pix2Code prototype by Belltramelli, upon which we based our project (github). [7]

References

[1] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473, 2014.

[2] T. Beltramelli. pix2code: Generating code from a graphical user interface screenshot. *CoRR*, abs/1705.07962, 2017.

[3] A. Karpathy and F. Li. Deep visual-semantic alignments for generating image descriptions. *CoRR*, abs/1412.2306, 2014.

[4] S. Sabour, N. Frosst, and G. E. Hinton. Dynamic routing between capsules. *CoRR*, abs/1710.09829, 2017.

[5] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the inception architecture for computer vision. *CoRR*, abs/1512.00567, 2015.

[6] O. Vinyals, A. Toshev, S. Bengio, and D. Erhan. Show and tell: A neural image caption generator. *CoRR*, abs/1411.4555, 2014.

[7] E. Wallner. Turning design mockups into code with deep learning. `https://blog.floydhub.com/turning-design-mockups-into-code-with-deep-learning/`, 2018.

[8] B. Wilkins. Sketching interfaces - generating code from low fidelity wireframes, 2017.

[9] K. Xu, J. Ba, R. Kiros, K. Cho, A. C. Courville, R. Salakhutdinov, R. S. Zemel, and Y. Bengio. Show, attend and tell: Neural image caption generation with visual attention. *CoRR*, abs/1502.03044, 2015.

[10] L. Zheng, Y. Zhao, S. Wang, J. Wang, and Q. Tian. Good practice in CNN feature transfer. *CoRR*, abs/1604.00133, 2016.