pix2code: Generating Code from a Graphical User Interface Screenshot

Tony Beltramelli
Uizard Technologies
Copenhagen, Denmark
tony@uizard.io

ABSTRACT

Transforming a graphical user interface screenshot created by a designer into computer code is a typical task conducted by a developer in order to build customized software, websites, and mobile applications. In this paper, we show that deep learning methods can be leveraged to train a model end-to-end to automatically reverse engineer user interfaces and generate code from a single input image with over 77% of accuracy for three different platforms (i.e. iOS, Android and web-based technologies).

CCS CONCEPTS

• **Human-centered computing** → **Graphical user interfaces**; **User interface programming**; • **Computing methodologies** → *Neural networks*; Scene understanding; Object recognition;

KEYWORDS

User Interface Reverse Engineering; Automated Software Engineering; Deep Neural Networks

ACM Reference Format:
Tony Beltramelli. 2018. pix2code: Generating Code from a Graphical User Interface Screenshot. In *EICS '18: ACM SIGCHI Symposium on Engineering Interactive Computing Systems, June 19–22, 2018, Paris, France.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3220134.3220135

1 INTRODUCTION

The process of implementing client-side software based on a *Graphical User Interface (GUI)* mockup created by a designer is the responsibility of developers. Implementing GUI code is, however, time-consuming and prevent developers from dedicating the majority of their time implementing the actual functionality and logic of the software they are building. Moreover, the computer languages used to implement such GUIs are specific to each target runtime system; thus resulting in tedious and repetitive work when the software being built is expected to run on multiple platforms using native technologies. In this paper, we describe a deep artificial neural network optimized end-to-end with stochastic gradient descent to simultaneously learn visual features, sequences, and spatio-temporal relationships to generate variable-length strings of tokens from a single GUI image as input.

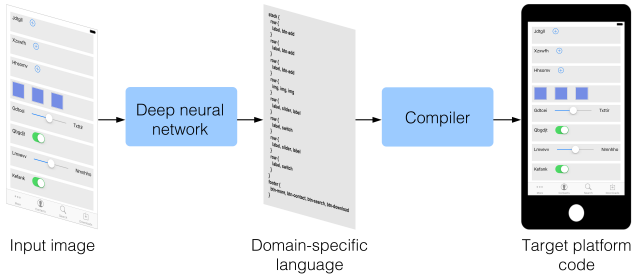
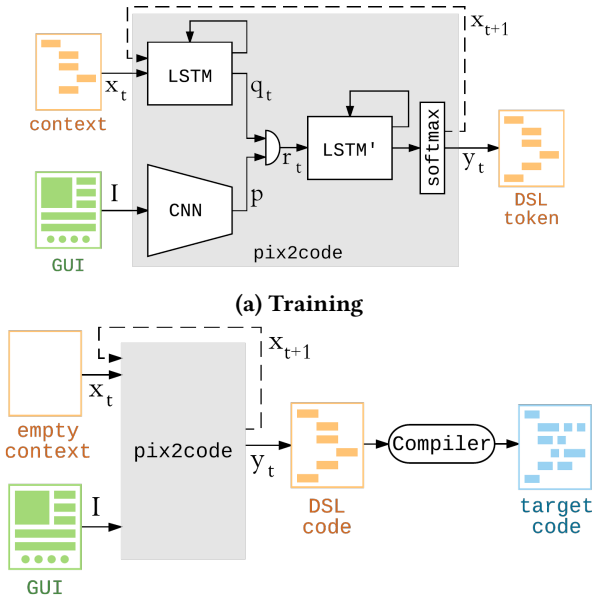


Figure 1: Deep neural network for user interface reverse engineering from a single image input.

Our first contribution is *pix2code*, a novel approach based on Convolutional and Recurrent Neural Networks allowing the generation of computer code from a single GUI screenshot as input; effectively reverse engineering user interfaces from a single image as depicted in Figure 1. No engineered feature extraction pipeline nor expert heuristics was designed to process the input data; our model learns from the pixel values of the input image alone. Our experiments demonstrate the effectiveness of our method to reverse engineer GUIs from various platforms (i.e. iOS and Android native mobile interfaces, and multi-platform web-based HTML/CSS interfaces) without the need for any change or specific tuning to the model. In fact, *pix2code* is designed as a general purpose solution to reconstruct GUIs and can be used as such to support different target platforms simply by training the model on a different dataset. To the best of our knowledge, this paper is the first work attempting to

EICS '18, June 19–22, 2018, Paris, France

ACM ISBN 978-1-4503-5897-2/18/06…$15.00
https://doi.org/10.1145/3220134.3220135



(a) Training

(b) Sampling

Figure 2: Overview of the *pix2code* model architecture.

address the problem of user interface reverse engineering from visual inputs by leveraging Machine Learning to learn latent variables instead of engineering complex heuristics. A video demonstrating our system is available online[1].

Our second contribution is the release of our synthesized datasets consisting of both GUI screenshots and associated source code for three different platforms. Our datasets and our *pix2code* implementation are publicly available[2] to foster future research.

2 RELATED WORK

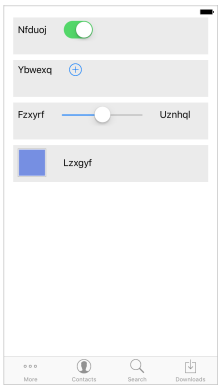
User interface reverse engineering has been an active field of research within the Human-Computer Interaction and Software Engineering communities. As extensively detailed by Ramón et al. [13], current state-of-the-art methods suffer from the fact that they are enabled by carefully engineered pipelines applied to a given domain. As a consequence, these solutions are tightly coupled with a given platform, environment, or language and cannot be easily reused in another context.

The closest methods to our work are performing user interface reverse engineering by analyzing the pixel values of rendered GUIs. In fact, all GUIs—regardless of platform, environment, or language—are rendered in displays using pixels. Pixel values are easy to access and a valuable source of information to reverse engineer GUIs. Nguyen et al. [12] demonstrated a method to reconstruct Android user interfaces from screenshots and Dixon et al. [3] a similar approach to reverse engineer desktop user interfaces running on Microsoft Windows Vista by analyzing pixel values. However, their methods rely entirely on engineered heuristics to identify user interface components from the raw image. Successfully developing such heuristics require a considerable amount of expert knowledge of the given domain; thus making it hard to apply the same technique to reverse-engineer GUIs for another domain. Because of this limitation, we can study the Machine Learning literature for possible solutions.

The automatic generation of programs using Machine Learning techniques is a relatively new field of research and program synthesis in a human-readable format have only been addressed very recently. A recent example is DeepCoder [1], a system able to generate computer programs by leveraging statistical predictions to augment traditional search techniques. In another work by Gaunt et al. [4], the generation of source code is enabled by learning the relationships between input-output examples via differentiable interpreters. Furthermore, Ling et al. [10] recently demonstrated program synthesis from a mixed natural language and structured program specification as input. It is important to note that most of these methods rely on *Domain-Specific Languages (DSLs)*; computer languages (e.g. markup languages, programming languages, modeling languages) that are designed for a specialized domain but are typically more restrictive than full-featured computer languages. Using DSLs thus limit the complexity of the programming language that needs to be modeled and reduce the size of the search space.

Although the generation of computer programs is an active research field as suggested by these breakthroughs, program generation from visual inputs is still a nearly unexplored research area. In order to exploit the graphical nature of our input, we can borrow methods from the Computer Vision literature. In fact, an important number of research [8, 17] have addressed the problem of image captioning with impressive results; showing that deep neural networks are able to learn latent variables describing objects in an image and their relationships with corresponding variable-length textual descriptions. All these methods rely on two main components. First, a *Convolutional Neural Network (CNN)* performing unsupervised feature learning mapping the raw input image to a learned representation. Second, a *Recurrent Neural Network (RNN)* performing language modeling on the textual description associated with the input picture. These approaches have the advantage of being differentiable end-to-end, thus allowing the use of gradient descent for optimization.

[1] https://uizard.io/research#pix2code
[2] https://github.com/tonybeltramelli/pix2code



(a) iOS GUI

```
stack {
    row {
        label, switch
    }
    row {
        label, btn-add
    }
    row {
        label, slider, label
    }
    row {
        img, label
    }
}
footer {
    btn-more, btn-contact, btn-search, btn-download
}
```

(b) DSL code describing the GUI

Figure 3: An example of a native iOS GUI written in our markup-like DSL.

3 PIX2CODE

The task of generating computer code written in a given programming language from a GUI image can be compared to the task of generating English textual descriptions from a scene photography. In both scenarios, we want to produce a variable-length strings of tokens from pixel values. We can thus divide our problem into three sub-problems. First, a computer vision problem of understanding the given scene (i.e. in this case, the GUI image) and inferring the objects present, their identities, positions, and poses (i.e. buttons, labels, element containers). Second, a language modeling problem of understanding text (i.e. in this case, computer code) and generating syntactically and semantically correct samples. Finally, the last challenge is to use the solutions to both previous sub-problems by exploiting the latent variables inferred from scene understanding to generate corresponding textual descriptions (i.e. computer code rather than English) of the objects represented by these variables.

Vision Model

CNNs are currently the method of choice to solve a wide range of vision problems thanks to their topology allowing them to learn rich latent representations from the images they are trained on [9]. We used a CNN to perform unsupervised feature learning by mapping an input image to a learned fixed-length vector; thus acting as an encoder as shown in Figure 2.

The input images are initially re-sized to 256×256 pixels (the aspect ratio is not preserved) and the pixel values are normalized before to be fed in the CNN. No further preprocessing is performed. To encode each input image to a fixed-size output vector, we exclusively used small 3×3 receptive fields which are convolved with stride 1 as used by Simonyan and Zisserman for VGGNet [14]. These operations are applied twice before to down-sample with max-pooling. The width of the first convolutional layer is 32, followed by a layer of width 64, and finally width 128. Two fully connected layers of size 1024 applying the *rectified linear unit* activation complete the vision model.

Language Model

We designed a simple lightweight DSL to describe GUIs as illustrated in Figure 3. In this work we are only interested in the GUI layout, the different graphical components, and their relationships; thus the actual textual value of the labels is ignored. Additionally to reducing the size of the search space, the DSL simplicity also reduces the size of the vocabulary (i.e. the total number of tokens supported by the DSL). As a result, our language model can perform token-level language modeling with a discrete input by using one-hot encoded vectors.

In most programming languages and markup languages, an element is declared with an opening token; if children elements or instructions are contained within a block, a closing token is usually needed for the interpreter or the compiler. In such a scenario where the number of children elements contained in a parent element is variable, it is important to model long-term dependencies to be able to close a block that has been opened. Traditional RNN architectures suffer from vanishing and exploding gradients preventing them from being able to model such relationships between data points spread out in time series (i.e. in this case tokens spread out in a sequence). Hochreiter and Schmidhuber proposed the *Long Short-Term Memory (LSTM)* neural architecture in order to address this very problem [7].

Decoder

Our model is trained in a supervised learning manner by feeding an image I and a contextual sequence X of T tokens $x_t, t \in \{0 \dots T-1\}$ as inputs; and the token x_T as the target label. As shown on Figure 2, a CNN-based vision model encodes the input image I into a vectorial representation p. The input token x_t is encoded by an LSTM-based language model into an intermediary representation q_t allowing the model to focus more on certain tokens and less on others [6]. This first language model is implemented as a stack of two LSTM layers with 128 cells each. The vision-encoded vector p and the language-encoded vector q_t are concatenated into a single feature vector r_t which is then fed into a second LSTM-based model decoding the representations learned by both the vision model and the language model. The decoder thus learns to model the relationship between objects present in the input GUI image and the associated tokens present in the DSL code. Our decoder is implemented as a stack of two LSTM layers with 512 cells each. This architecture can be expressed mathematically as follows:

$$p = CNN(I) \tag{1}$$
$$q_t = LSTM(x_t) \tag{2}$$
$$r_t = (p, q_t) \tag{3}$$
$$y_t = softmax(LSTM'(r_t)) \tag{4}$$
$$x_{t+1} = y_t \tag{5}$$

This architecture allows the whole *pix2code* model to be optimized end-to-end with gradient descent to predict a token at a time after it has seen both the image as well as the preceding tokens in the sequence. The discrete nature of the output (i.e. fixed-sized vocabulary of tokens in the DSL) allows us to reduce the task to a classification problem. That is, the output layer of our model has the same number of cells as the vocabulary size; thus generating a probability distribution of the candidate tokens at each time step allowing the use of a *softmax* layer to perform multi-class classification.

Training

The length T of the sequences used for training is important to model long-term dependencies; for example to be able to close a block of code that has been opened. After conducting empirical experiments, the DSL input files used for training were segmented with a sliding window of size 48; in other words, we unroll the recurrent neural network for 48 steps. This was found to be a satisfactory trade-off between long-term dependencies learning and computational cost. For every token in the input DSL file, the model is therefore fed with both an input image and a contextual sequence of $T = 48$ tokens. While the context (i.e. sequence of tokens) used for training is updated at each time step (i.e. each token) by sliding the window, the very same input image I is reused for samples associated with the same GUI. The special tokens $< START >$ and $< END >$ are used to respectively prefix and suffix the DSL files similarly to the method used by Karpathy and Fei-Fei [8]. Training is performed by computing the partial derivatives of the loss with respect to the network weights calculated with backpropagation to minimize the multiclass log loss:

$$L(I, X) = -\sum_{t=1}^{T} x_{t+1} \log(y_t) \tag{6}$$

With x_{t+1} the expected token, and y_t the predicted token. The model is optimized end-to-end hence the loss L is minimized with regard to all the parameters including all layers in the CNN-based vision model and all layers in both LSTM-based models. Training with the *RMSProp* algorithm [16] gave the best results with a learning rate set to $1e-4$ and by clipping the output gradient to the range $[-1.0, 1.0]$ to cope with numerical instability [6]. To prevent overfitting, a dropout regularization [15] set to 25% is applied to the vision model after each max-pooling operation and at 30% after each fully-connected layer. In the LSTM-based models, dropout is set to 10% and only applied to the non-recurrent connections [19]. Our model was trained with mini-batches of 64 image-sequence pairs.

Dataset type		iOS	Android	Web
Synthesizable		26×10^5	21×10^6	31×10^4
Train set	Instances	1500	1500	1500
	Samples	93672	85756	143850
Test set	Instances	250	250	250
	Samples	15984	14265	24108
Error (%)	Greedy	**22.73**	**22.34**	12.14
	Beam 3	25.22	23.58	**11.01**
	Beam 5	23.94	40.93	22.35

Table 1: Dataset statistics and experiments results.

Sampling

To generate DSL code, we feed the GUI image I and a contextual sequence X of $T = 48$ tokens where tokens $x_t \dots x_{T-1}$ are initially set empty and the last token of the sequence x_T is set to the special $< START >$ token. The predicted token y_t is then used to update the next sequence of contextual tokens. That is, $x_t \dots x_{T-1}$ are set to $x_{t+1} \dots x_T$ (x_t is thus discarded), with x_T set to y_t. The process is repeated until the token $< END >$ is generated by the model. The generated DSL token sequence can then be compiled with traditional compilation methods to the desired target language.

4 EXPERIMENTS

Access to consequent datasets is a typical bottleneck when training deep neural networks. To the best of our knowledge, no dataset consisting of both GUI screenshots and source code was available at the time this paper was written. As a consequence, we synthesized our own data resulting in the three datasets described in Table 1. The columns *iOS* and *Android* refer to UIs in the XML format while the column *Web* refers to web-based UIs implemented in HTML/CSS. The row *Synthesizable* refers to the maximum number of unique GUI configuration that can be synthesized using our stochastic user interface generator. The rows *Instances* refer to the number of synthesized (GUI screenshot, GUI code) file pairs. The rows *Samples* refer to the number of distinct image-sequence pairs. In fact, training and sampling are done one token at a time by feeding the model with an image and a sequence of tokens obtained with a sliding window of fixed size T. The total number of training samples thus depends on the total number of tokens written in the DSL files and the size of the sliding window. Our stochastic user interface

generator is designed to synthesize GUIs written in our DSL which is then compiled to the desired target language to be rendered. Using data synthesis also allows us to demonstrate the capability of our model to generate computer code for three different platforms.

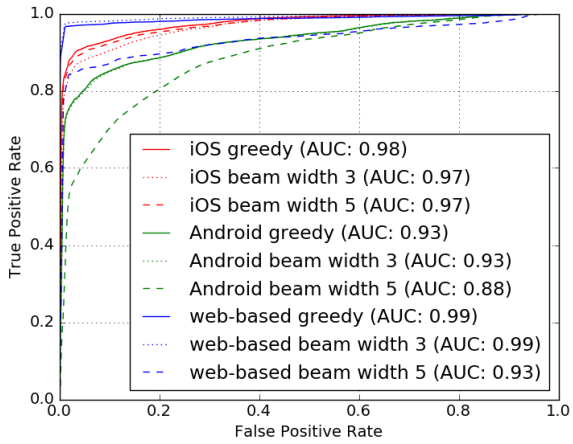
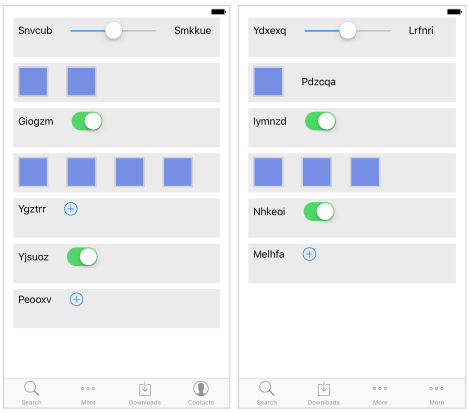


Figure 4: Micro-average ROC curves

Figure 5: ROC curves calculated during sampling after the model is trained for 10 epochs.

Our model has around 109×10^6 parameters to optimize and all experiments are performed with the same model with no specific tuning; only the training datasets differ as shown on Figure 5. Code generation is performed with both greedy search and beam search with various beam sizes to find the tokens that maximize the classification probability. To evaluate the quality of the generated output, the classification error is computed for each sampled DSL token and averaged over the whole test dataset. The length difference between the generated and the expected token sequences is also counted as error. The results reported for the test sets can be seen on Table 1.

Figures 6, 7, and 8 show samples consisting of input GUIs (i.e. ground truth), and output GUIs. These output GUI screenshots are obtained by sampling code with a trained *pix2code* model; the generated DSL code is then compiled to the appropriate target language producing UI code that can be rendered and captured as an image. It is important to remember that the actual textual value of the labels is ignored and that both our data synthesis algorithm and our DSL compiler assign randomly generated text to the labels. Despite occasional problems to select the right color or the right style for specific GUI elements and some difficulties modelling GUIs consisting of long lists of graphical components, our model is generally able to learn the GUI layout in a satisfying



(a) Groundtruth **(b) Generated**

Figure 6: iOS GUI experiment sample.

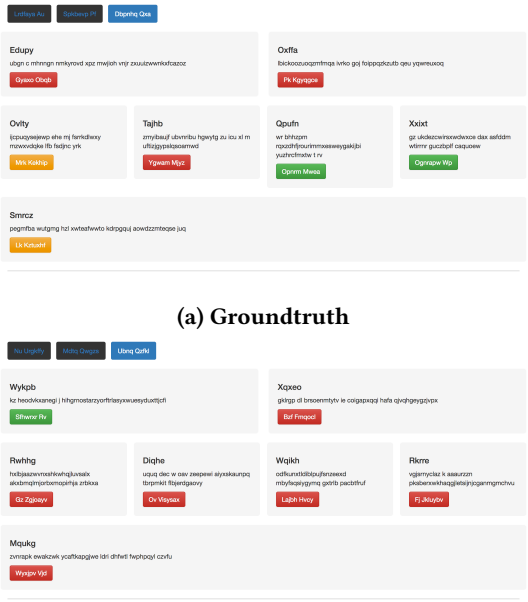
manner and can preserve the hierarchical structure of the graphical elements.

5 CONCLUSION AND DISCUSSIONS

In this paper, we presented *pix2code*, a novel method to generate user interface code by reverse-engineering a single GUI image input. While our work demonstrates the potential of such a system to automate the process of implementing GUIs, we only scratched the surface of what is possible. Our model consists of relatively few parameters and was trained on a relatively small dataset. The accuracy of the system could be drastically improved by training a bigger model on significantly more data. Implementing a now-standard attention mechanism [18] could further improve the performance of our method. Furthermore, our approach still requires an expert to write a transpiler to compile the DSL code to any new target language.

Using one-hot encoding does not provide any useful information about the relationships between the tokens since the method simply assigns an arbitrary vectorial representation to each token. Therefore, pre-training the language model to learn vectorial representations would allow the relationships between tokens in the DSL to be inferred (i.e. learning word embeddings such as word2vec [11]) and as a result alleviate semantical error in the generated code. Furthermore, one-hot encoding does not scale to very big vocabulary and thus restrict the number of symbols that the DSL can support.

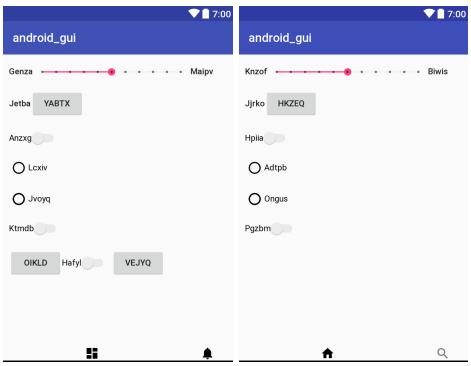
Generative Adversarial Networks (GANs) [5] have shown to be extremely powerful at generating images and sequences [2, 20]. Applying such techniques to the problem of generating computer code from an input image is so far an unexplored research area. GANs could potentially be used as a standalone method to generate code or could be used in combination with our *pix2code* model to fine-tune results.



(a) Groundtruth

(b) Generated

Figure 7: Web-based GUI experiment sample.



(a) Groundtruth **(b) Generated**

Figure 8: Android GUI experiment sample.

6 ACKNOWLEDGMENTS

The authors would like to thank Prof. Jean Vanderdonckt of Université catholique de Louvain for providing valuable feedback about related works.

REFERENCES

[1] Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2016. DeepCoder: Learning to Write Programs. *arXiv preprint arXiv:1611.01989* (2016).

[2] Bo Dai, Dahua Lin, Raquel Urtasun, and Sanja Fidler. 2017. Towards Diverse and Natural Image Descriptions via a Conditional GAN. *arXiv preprint arXiv:1703.06029* (2017).

[3] Morgan Dixon and James Fogarty. 2010. Prefab: implementing advanced behaviors using pixel-based reverse engineering of interface structure. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 1525–1534.

[4] Alexander L Gaunt, Marc Brockschmidt, Rishabh Singh, Nate Kushman, Pushmeet Kohli, Jonathan Taylor, and Daniel Tarlow. 2016. Terpret: A probabilistic programming language for program induction. *arXiv preprint arXiv:1608.04428* (2016).

[5] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative adversarial nets. In *Advances in neural information processing systems*. 2672–2680.

[6] Alex Graves. 2013. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850* (2013).

[7] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[8] Andrej Karpathy and Li Fei-Fei. 2015. Deep visual-semantic alignments for generating image descriptions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 3128–3137.

[9] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.

[10] Wang Ling, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiskỳ, Andrew Senior, Fumin Wang, and Phil Blunsom. 2016. Latent predictor networks for code generation. *arXiv preprint arXiv:1603.06744* (2016).

[11] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*. 3111–3119.

[12] Tuan Anh Nguyen and Christoph Csallner. 2015. Reverse engineering mobile application user interfaces with remaui (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 248–259.

[13] Óscar Sánchez Ramón, Jean Vanderdonckt, and Jesús García Molina. 2013. Re-engineering graphical user interfaces from their resource files with UsiResourcer. In *Research Challenges in Information Science (RCIS), 2013 IEEE Seventh International Conference on*. IEEE, 1–12.

[14] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[15] Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research* 15, 1 (2014), 1929–1958.

[16] Tijmen Tieleman and Geoffrey Hinton. 2012. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning* 4, 2 (2012).

[17] Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan. 2015. Show and tell: A neural image caption generator. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 3156–3164.

[18] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron C Courville, Ruslan Salakhutdinov, Richard S Zemel, and Yoshua Bengio. 2015. Show, Attend and Tell: Neural Image Caption Generation with Visual Attention.. In *ICML*, Vol. 14. 77–81.

[19] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. 2014. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329* (2014).

[20] Han Zhang, Tao Xu, Hongsheng Li, Shaoting Zhang, Xiaolei Huang, Xiaogang Wang, and Dimitris Metaxas. 2016. StackGAN: Text to Photo-realistic Image Synthesis with Stacked Generative Adversarial Networks. *arXiv preprint arXiv:1612.03242* (2016).