

---

# 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 techniques can be leveraged to automatically generate code given a graphical user interface screenshot as input. Our model is able to generate code targeting three different platforms (i.e. iOS, Android and web-based technologies) from a single input image with over 77% of accuracy.

## 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 features and logic of the software they are building. Moreover, the computer languages used to implement such GUIs are specific to each target platform; 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 system that can automatically generate platform-specific computer code given a GUI screenshot as input. We extrapolate that a scaled version of our method could potentially end the need for manually-programmed GUIs.

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. Our model is able to generate computer code from the pixel values of the input image alone. That is, no engineered feature extraction pipeline is designed to pre-process the input data. Our experiments demonstrate the effectiveness of our method for generating computer code for 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* can be used as such to generate code written in different target languages simply by being trained on a different dataset. A video demonstrating our system is available online<sup>1</sup>.

Our second contribution is the release of our synthesized datasets consisting of both GUI screenshots and associated source code for three different platforms. They will be made freely available<sup>2</sup> upon publication of this paper to foster future research.

## 2 Related Work

The automatic generation of programs using machine learning techniques is a relatively new field and program synthesis in a human-readable format have only been addressed very recently. A recent example is DeepCoder [2], a system able to generate computer programs by leveraging statistical

<sup>1</sup><https://uizard.io/research#pix2code>

<sup>2</sup><https://github.com/tonybeltramelli/pix2code>

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. [11] 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 thus reduce the size of the search space.

Although the generation of computer programs is an active research field as suggested by these breakthroughs, code generation from visual inputs is still an unexplored research area. This paper is, to the best of our knowledge, the first work attempting to address this very problem. 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 [19, 18, 3, 9] have shown that deep neural networks are able to learn latent variables describing objects in an image and generate a variable-length textual description of the objects and their relationships. All these methods rely on two main components. First, a *Convolutional Neural Network (CNN)* transforming the raw input image into an intermediary learned representation. Second, a *Recurrent Neural Network (RNN)* performing language modeling on the textual description associated with the input picture. Combining both neural network architectures allows the generation of image captions with impressive results.

### 3 pix2code

The task of generating code given a GUI screenshot as input can be compared to the task of generating English textual descriptions given a scene photography as input. 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 screenshot 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 text) of the objects represented by these variables.

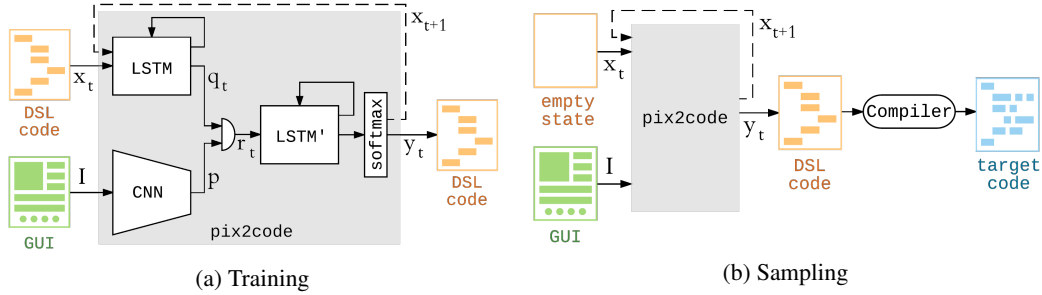
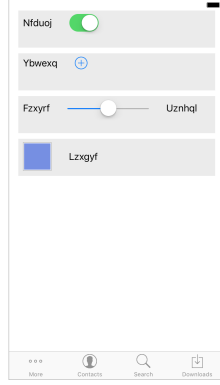


Figure 1: Overview of the *pix2code* model architecture. During training, the GUI screenshot picture is encoded by a CNN-based vision model; the sequence of one-hot encoded tokens corresponding to DSL code is encoded by a language model consisting of a stack of LSTM layers. The two resulting encoded vectors are then concatenated and fed into a second stack of LSTM layers acting as a decoder. Finally, a *softmax* layer is used to sample one token at a time; the output size of the *softmax* layer corresponding to the DSL vocabulary size. Given an image and a sequence of tokens, the model (i.e. contained in the gray box) is differentiable and can thus be optimized end-to-end through gradient descent to predict the next token in the sequence. The input state (i.e. a sequence of tokens) is updated at each prediction to contain the last predicted token. During sampling, the generated DSL code is compiled to the desired target language using traditional compiler design techniques.

#### 3.1 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 [14, 10]. 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 1.

The input images are initially re-sized to  $256 \times 256$  pixels (the aspect ratio is not preserved) and the pixel values are normalized before to be fed in the CNN. No further pre-processing is performed. To encode each input image to a fixed-size output vector, we exclusively used small  $3 \times 3$  receptive fields which are convolved with stride 1 as used by Simonyan and Zisserman for VGGNet [15]. 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.



(a) iOS GUI screenshot

```
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) Code describing the GUI written in our DSL

Figure 2: An example of a native iOS GUI written in our DSL.

### 3.2 Language Model

We designed a simple DSL to describe GUIs as illustrated in Figure 2. 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 by our DSL. 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 from using one-hot encoded vectors; eliminating the need for word embedding techniques such as word2vec [12] that can result in costly computations.

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 [8] proposed the *Long Short-Term Memory (LSTM)* neural architecture in order to address this very problem. The different LSTM gate outputs can be computed as follows:

$$i_t = \phi(W_{ix}x_t + W_{iy}h_{t-1} + b_i) \quad (1)$$

$$f_t = \phi(W_{fx}x_t + W_{fy}h_{t-1} + b_f) \quad (2)$$

$$o_t = \phi(W_{ox}x_t + W_{oy}h_{t-1} + b_o) \quad (3)$$

$$c_t = f_t \bullet c_{t-1} + i_t \bullet \sigma(W_{cx}x_t + W_{cy}h_{t-1} + b_c) \quad (4)$$

$$h_t = o_t \bullet \sigma(c_t) \quad (5)$$

With  $W$  the matrices of weights,  $x_t$  the new input vector at time  $t$ ,  $h_{t-1}$  the previously produced output vector,  $c_{t-1}$  the previously produced cell state's output,  $b$  the biases, and  $\phi$  and  $\sigma$  the activation functions *sigmoid* and *hyperbolic tangent*, respectively. The cell state  $c$  learns to memorize information by using a recursive connection as done in traditional RNN cells. The input gate  $i$  is used to control the error flow on the inputs of cell state  $c$  to avoid input weight conflicts that occur in traditional RNN because the same weight has to be used for both storing certain inputs and ignoring others. The output gate  $o$  controls the error flow from the outputs of the cell state  $c$  to prevent output weight conflicts that happen in standard RNN because the same weight has to be used for both retrieving information and not retrieving others. The LSTM memory block can thus use  $i$  to decide

when to write information in  $c$  and use  $o$  to decide when to read information from  $c$ . We used the LSTM variant proposed by Gers and Schmidhuber [5] with a forget gate  $f$  to reset memory and help the network model continuous sequences.

### 3.3 Combining Models

Our model is trained in a supervised learning manner by feeding an image  $I$  and a 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 1, 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 [7]. 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 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:

$$p = CNN(I) \tag{6}$$

$$q_t = LSTM(x_t) \tag{7}$$

$$r_t = (q, p_t) \tag{8}$$

$$y_t = softmax(LSTM'(r_t)) \tag{9}$$

$$x_{t+1} = y_t \tag{10}$$

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.

### 3.4 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 code input file used for training was 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 sequence of  $T = 48$  tokens. While the 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  $\langle START \rangle$  and  $\langle END \rangle$  are used to respectively prefix and suffix the DSL code files similarly to the method used by Karpathy et al. [9]. 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:

$$L(I, X) = - \sum_{t=1}^T x_{t+1} \log(y_t) \tag{11}$$

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 [17] 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 [7]. To prevent overfitting, a dropout regularization [16] 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 [21]. Our model was trained with mini-batches of 64 image-sequence pairs.

### 3.5 Sampling

To generate DSL code, we feed the GUI image  $I$  and a 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  $\langle START \rangle$  token. The predicted token  $y_t$  is then used to update the next sequence of input 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  $\langle END \rangle$  is generated by the model. The generated DSL code can then be compiled with traditional compilation methods to the desired target language.

Dataset type	Synthesizable	Training set		Test set	
		Instances	Samples	Instances	Samples
iOS UI (Storyboard)	$26 \times 10^5$	1500	93672	250	15984
Android UI (XML)	$21 \times 10^6$	1500	85756	250	14265
web-based UI (HTML/CSS)	$31 \times 10^4$	1500	143850	250	24108

Table 1: Dataset statistics. The column *Synthesizable* refers to the maximum number of unique GUI configuration that can be synthesized using our data synthesis algorithm. The columns *Instances* refers to the number of synthesized (GUI screenshot, GUI code) file pairs. The columns *Samples* refers 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 code files and the size of the sliding window which we set to  $T = 48$ .

## 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 that will be open-sourced. Our data synthesis algorithm 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.

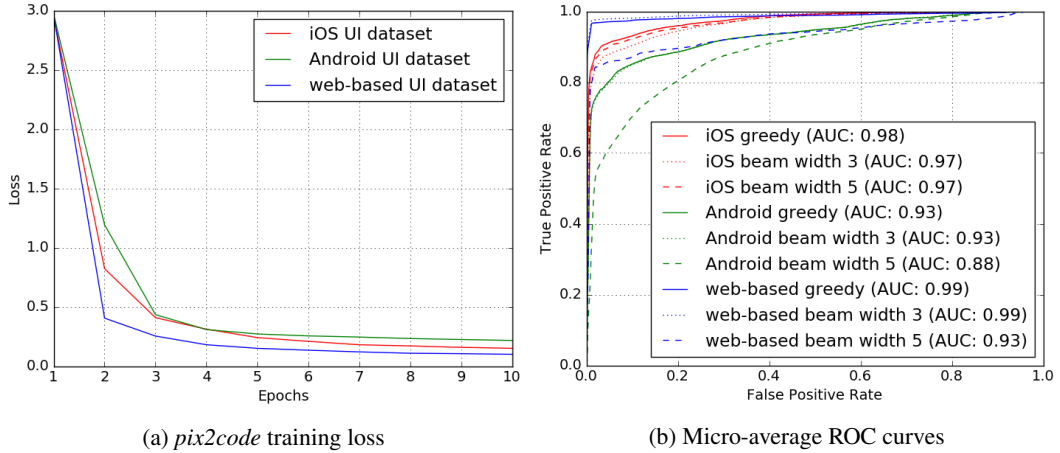


Figure 3: Training loss on different datasets and ROC curves calculated during sampling with the model trained for 10 epochs.

Our model has around  $109 \times 10^6$  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 3. Code generation is performed with both greedy search and beam search 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 can be seen on Table 2.

Dataset type	Error (%)		
	greedy search	beam search 3	beam search 5
iOS UI (Storyboard)	<b>22.73</b>	25.22	23.94
Android UI (XML)	<b>22.34</b>	23.58	40.93
web-based UI (HTML/CSS)	12.14	<b>11.01</b>	22.35

Table 2: Experiments results reported for the test sets described in Table 1.

Figures 4, 5, and 6 show samples consisting of input GUIs (i.e. ground truth), and output GUIs generated by a trained *pix2code* model. 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 assigns 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 manner and can preserve the hierarchical structure of the graphical elements.



Figure 4: Experiment samples for the iOS GUI dataset.

## 5 Conclusion and Discussions

In this paper, we presented *pix2code*, a novel method to generate computer code given a single GUI screenshot as input. While our work demonstrates the potential of such a system to automate GUI programming, 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 quality of the generated code could be drastically improved by training a bigger model on significantly more data for an extended number of epochs. Experimenting with various regularization methods or implementing an attention mechanism [1] could further improve the quality of the generated code.

Using one-hot encoding as we did does not provide any useful information about the relationships between the tokens since the method simply assigns an arbitrary vector representation to each token. Therefore, pre-training the language model to learn vectorial representations of the tokens would allow the relationships between tokens in the DSL to be inferred (i.e. learning word embeddings such as word2vec [12]) 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* [6] have shown to be extremely powerful at generating images and sequences [20, 13, 22]. 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 major drawback of deep neural networks is the need for a lot of training data for the resulting model to generalize well on new unseen examples. One of the significant advantages of the method we described in this paper is that there is no need for human-labelled data. In fact, the network can model the relationships between graphical components and associated code by simply being

trained on image-sequence pairs. Although we used data synthesis in our paper partly to demonstrate the capability of our method to generate GUI code for various platforms; data synthesis might not be needed at all if one wants to focus only on web-based GUIs. In fact, one could imagine crawling the World Wide Web to collect a dataset of HTML/CSS code associated with screenshots of renderer GUIs. Considering a large number of websites already available online and the fact that new websites are created every day, the web could theoretically supply an unlimited amount of training data. We extrapolate that Deep Learning used in this manner could eventually end the need for manually-programmed GUIs.



Figure 5: Experiment samples from the Android GUI dataset.

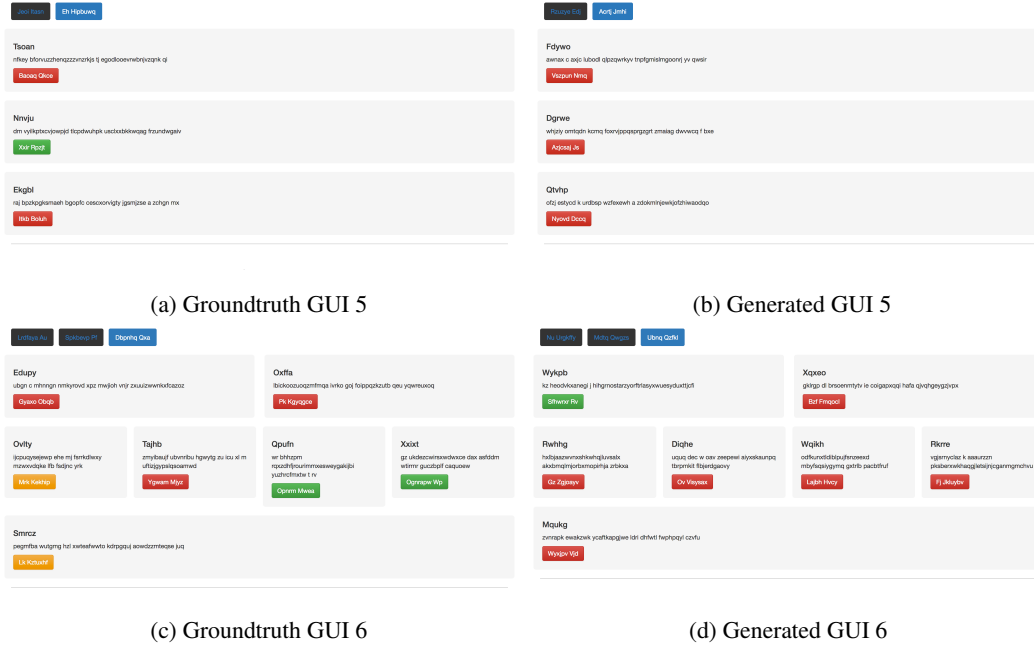


Figure 6: Experiment samples from the web-based GUI dataset.

## References

- [1] Bahdanau, D., Cho, K., and Bengio, Y.: 2014, *arXiv preprint arXiv:1409.0473*
- [2] Balog, M., Gaunt, A. L., Brockschmidt, M., Nowozin, S., and Tarlow, D.: 2016, *arXiv preprint arXiv:1611.01989*
- [3] Donahue, J., Anne Hendricks, L., Guadarrama, S., Rohrbach, M., Venugopalan, S., Saenko, K., and Darrell, T.: 2015, in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp 2625–2634
- [4] Gaunt, A. L., Brockschmidt, M., Singh, R., Kushman, N., Kohli, P., Taylor, J., and Tarlow, D.: 2016, *arXiv preprint arXiv:1608.04428*
- [5] Gers, F. A., Schmidhuber, J., and Cummins, F.: 2000, *Neural computation* **12**(10), 2451
- [6] Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y.: 2014, in *Advances in neural information processing systems*, pp 2672–2680
- [7] Graves, A.: 2013, *arXiv preprint arXiv:1308.0850*
- [8] Hochreiter, S. and Schmidhuber, J.: 1997, *Neural computation* **9**(8), 1735
- [9] Karpathy, A. and Fei-Fei, L.: 2015, in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp 3128–3137
- [10] Krizhevsky, A., Sutskever, I., and Hinton, G. E.: 2012, in *Advances in neural information processing systems*, pp 1097–1105
- [11] Ling, W., Grefenstette, E., Hermann, K. M., Kočiský, T., Senior, A., Wang, F., and Blunsom, P.: 2016, *arXiv preprint arXiv:1603.06744*
- [12] Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., and Dean, J.: 2013, in *Advances in neural information processing systems*, pp 3111–3119
- [13] Reed, S., Akata, Z., Yan, X., Logeswaran, L., Schiele, B., and Lee, H.: 2016, in *Proceedings of The 33rd International Conference on Machine Learning*, Vol. 3
- [14] Sermanet, P., Eigen, D., Zhang, X., Mathieu, M., Fergus, R., and LeCun, Y.: 2013, *arXiv preprint arXiv:1312.6229*
- [15] Simonyan, K. and Zisserman, A.: 2014, *arXiv preprint arXiv:1409.1556*
- [16] Srivastava, N., Hinton, G. E., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R.: 2014, *Journal of Machine Learning Research* **15**(1), 1929
- [17] Tieleman, T. and Hinton, G.: 2012, *COURSERA: Neural networks for machine learning* 4(2)
- [18] Vinyals, O., Toshev, A., Bengio, S., and Erhan, D.: 2015, in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp 3156–3164
- [19] Xu, K., Ba, J., Kiros, R., Cho, K., Courville, A. C., Salakhutdinov, R., Zemel, R. S., and Bengio, Y.: 2015, in *ICML*, Vol. 14, pp 77–81
- [20] Yu, L., Zhang, W., Wang, J., and Yu, Y.: 2016, *arXiv preprint arXiv:1609.05473*
- [21] Zaremba, W., Sutskever, I., and Vinyals, O.: 2014, *arXiv preprint arXiv:1409.2329*
- [22] Zhang, H., Xu, T., Li, H., Zhang, S., Huang, X., Wang, X., and Metaxas, D.: 2016, *arXiv preprint arXiv:1612.03242*