From UI Design Image to GUI Skeleton: A Neural Machine Translator to Bootstrap Mobile GUI Implementation

Chunyang Chen[1], Ting Su[1*], Guozhu Meng[1,3*], Zhenchang Xing[2], and Yang Liu[1]
[1]School of Computer Science and Engineering, Nanyang Technological University, Singapore
[2]Research School of Computer Sciecne, Australian National University, Australia
[3]SKLOIS, Institute of Information Engineering, Chinese Academy of Sciences, China
chen0966@e.ntu.edu.sg,{suting,gzmeng}@ntu.edu.sg,zhenchang.xing@anu.edu.au,yangliu@ntu.edu.sg

ABSTRACT

A GUI skeleton is the starting point for implementing a UI design image. To obtain a GUI skeleton from a UI design image, developers have to visually understand UI elements and their spatial layout in the image, and then translate this understanding into proper GUI components and their compositions. Automating this visual understanding and translation would be beneficial for bootstraping mobile GUI implementation, but it is a challenging task due to the diversity of UI designs and the complexity of GUI skeletons to generate. Existing tools are rigid as they depend on heuristically-designed visual understanding and GUI generation rules. In this paper, we present a neural machine translator that combines recent advances in computer vision and machine translation for translating a UI design image into a GUI skeleton. Our translator learns to extract visual features in UI images, encode these features' spatial layouts, and generate GUI skeletons in a unified neural network framework, without requiring manual rule development. For training our translator, we develop an automated GUI exploration method to automatically collect large-scale UI data from real-world applications. We carry out extensive experiments to evaluate the accuracy, generality and usefulness of our approach.

CCS CONCEPTS

• **Software and its engineering**; • **Human-centered computing** → *Graphical user interfaces*;

KEYWORDS

User interface, reverse engineering, deep learning

ACM Reference Format:
Chunyang Chen, Ting Su, Guozhu Meng, Zhenchang Xing, and Yang Liu. 2018. From UI Design Image to GUI Skeleton: A Neural Machine Translator to Bootstrap Mobile GUI Implementation. In *Proceedings of ICSE '18: 40th International Conference on Software Engineering , Gothenburg, Sweden, May 27-June 3, 2018 (ICSE '18),* 12 pages.
https://doi.org/10.1145/3180155.3180222

*Corresponding authors.

ICSE '18, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5638-1/18/05…$15.00
https://doi.org/10.1145/3180155.3180222

1 INTRODUCTION

Mobile applications (apps) are event-centric programs with rich Graphical User Interfaces (GUIs). An app's GUI should not only provide a working interface for user interactions, but also create an intuitive and pleasant user experience. In fact, the later is crucial for an app's success in the highly competitive market [46, 56]. Developing the GUI of an app routinely involves two separate but related activities: design a UI and implement a UI. Designing a UI requires proper user interaction, information architecture and visual effects of the UI, while implementing a UI focuses on making the UI work with proper layouts and widgets of a GUI framework. A UI design can be created from scratch or adapted from UI design kits [1] or existing apps' GUIs, and it is usually conveyed to developers in the form of design images to implement.

A UI design image depicts the desired UI elements and their spatial layout in a matrix of pixels. To implement a UI design image using a GUI framework, developers must be able to translate the pixel-based depiction of the UI (or parts of the UI) into a GUI skeleton. As illustrated in Figure 1, a GUI skeleton defines what and how the components of a GUI builder (e.g., Android layouts and widgets) should be composed in the GUI implementation for reproducing the UI elements and their spatial layout in the UI design image. This GUI skeleton is like the initial "bootstrap instructions" which enables the subsequent GUI implementation (e.g., setting up font, color, padding, background image, and etc.)

However, there is a conceptual gap between a UI design image (i.e., a UI design in a pixel language) and the GUI skeleton (i.e., the UI design in a language of GUI framework component names). To bridge this gap, developers need to have a good knowledge of a GUI framework's components and what visual effects, interactions and compositions these components support in order to create an appropriate GUI skeleton for different kinds of UI elements and spatial layouts. If developers do not have this knowledge, the GUI implementation will become stucked, because modern GUI implementation cannot be achieved by hardcode-positioning some texts, images and controls. This is especially the case for mobile apps that have to run on a wide range of screen sizes.

To overcome the knowledge barrier between UI design image and GUI skeleton, developers may attempt to figure out what and how the GUI components should be composed for a UI design image through a trial-and-error approach. Although modern GUI builders provide strong interactive support (e.g., drag & drop, what-you-see-is-what-you-get) for creating a GUI implementation, this type of trail-and-error attempt would be very cumbersome and frustrating. First, a mobile app's GUI often involves many GUI components and complex spatial layout (see Figure 6(b)). Second, a complex

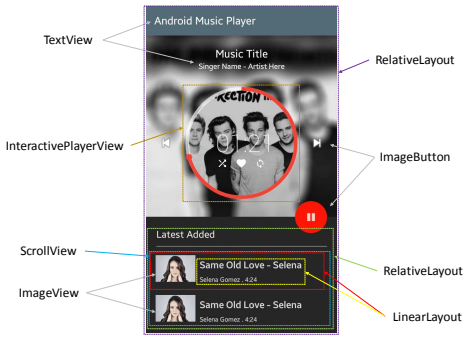


Figure 1: Translating a design image into a Android GUI skeleton (not all)

GUI framework supports dozens of layouts and widgets (some may be interchangeable) and flexible composition of these layouts and widgets. Developers can easily get lost during the trial-and-error of an unfamiliar GUI framework.

Alternatively, developers can learn from GUI framework tutorials or existing GUI implementations. To that end, they must be able to find some tutorials or GUI implementations that implement the UI designs that are similar to the desired UI. Finding such tutorials or GUI implementations through the UI design image is a challenging image search task. It is also difficult to formulate a concise, accurate text query of the UI design and the needed GUI components for using information retrieval (IR) methods. Developers can also seek solutions for implementing a UI design from the developer community (e.g., Stack Overflow), but they may not always be able to obtain useful advices in time.

The UIs of apps can be very sophisticated to support complex tasks, and they may undergo many revisions during the apps' lifespan. Considering millions of apps being developed and maintained, automating the translation from UI design to GUI implementation would be beneficial for mobile app development. Some tools [30, 45] can automatically generate the GUI implementation given a UI design image. This automatic, generative approach overcomes the limitations of the trial-and-error, search-based or ask-developer-community approaches for transforming UI design image into GUI skeleton. However, existing tools are rigid because they depend on hand-designed visual understanding and GUI generation templates which incorporate only limited UI-image-to-GUI-skeleton translation knowledge.

In this work, we present a deep learning architecture that distills the crowd-scale knowledge of UI designs and GUI implementations from existing apps and develop a generative tool to automatically generate the GUI skeleton given an input UI design image. Our generative tool can be thought of as an "expert" who knows a vast variety of UI designs and GUI skeletons to advise developers what and how the components of a GUI framework should be composed for implementing a UI design image.

To build this "expert", we must tackle two fundamental challenges. First, to be a knowledgeable expert, the generative tool must be exposed to a knowledge source of a vast variety of UI designs and GUI skeletons from a large number of apps. Second, to advise developers how to translate a UI design into a GUI skeleton, the generative tool must capture not only the UI elements contained in a UI design image, but it also must express how these UI elements relate to each other in terms of a composition of the GUI components. In this paper, we present an automated GUI exploration technique for tackling the first challenge in knowledge source, and develop a neural machine translator that combines recent advances in computer vision and machine translation for tackling the second challenge in visual understanding and skeleton generation. The neural machine translator is end-to-end trainable using a large dataset of diverse UI screenshots and runtime GUI skeletons that are automatically collected during the automated GUI exploration of mobile app binaries.

We implement an Android UI data collector [52, 53] and use it to automatically collect 185,277 pairs of UI images and GUI skeletons from 5043 Android apps. We adopt this dataset to train our neural machine translator and conduct unprecedented large-scale evaluation of the accuracy of our translator for UI-image-to-GUI-skeleton generation. Our evaluation shows that our translator can reliably distinguish different types of visual elements and spatial layouts in very diverse UI images and accurately generate the right GUI components and compositions for a wide range of GUI skeleton complexity. We also apply our translator to the UIs of 20 Android apps that are not in our training set, and this study further confirms the generality of our translator. Through a pilot user study, we provide the initial evidence of the usefulness of our approach for bootstraping GUI implementations.

Our contributions in this work are as follows:

- We develop a deep-learning based generative tool for overcoming the knowledge barrier for translating UI images to GUI skeletons.
- Our generative tool combines CNN and RNN models for learning a crowd-scale knowledge of UI images and GUI skeletons from a large number of mobile apps.
- We develop an automated GUI exploration framework to automatically build a large dataset of UI images and GUI skeletons for training the deep learning models.
- We show our tool's robust visual understanding and GUI skeleton generation capability through large-scale experiments, and provide initial evidence of our tool's usefulness by a pilot user study.

2 PROBLEM FORMULATION

We formulate the UI-image-to-GUI-skeleton generation as a machine translation task. The input i to the machine translator is a UI design image (can be regarded as a UI design in a pixel language, e.g., RGB color or grayscale pixels). As shown in Figure 2, the machine translator should be able to "translate" the input UI design image into a GUI skeleton, i.e., a composition of some container components (i.e., the non-leaf nodes) and atomic components (i.e., the leaf nodes) of a GUI framework.

A GUI skeleton can be regarded as the UI design in a GUI framework language whose vocabulary consists of the component names of the GUI framework, such as Android's *RelativeLayout*, *TextView*, *ImageButton*, and two special tokens (e.g., brackets "{" and "}") expressing the composition of GUI components. As shown in Figure 2, a component hierarchy can be represented as an equivalent token sequence via depth-first traversal (DFT) and using "{" and "}" to enclose a container's contained components in the token sequence. In this work, we use the token sequence representation of the GUI skeleton as the output of the machine translator.

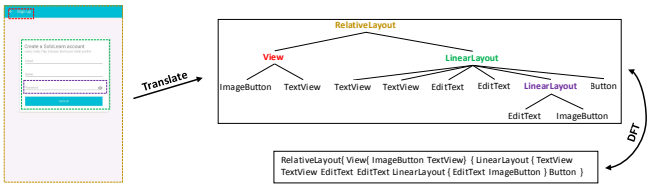


Figure 2: An example of UI-image-to-GUI-skeleton generation

3 NEURAL MACHINE TRANSLATOR

Unlike normal machine translation tasks where both source and target languages are text data, our machine translation task requires joint processing of image and text data. Furthermore, unlike text which is a sequence of words, our input UI design image contains the spatial layout information of UI elements, and our output GUI skeleton is a hierarchical composition of GUI components. Taking into account these characteristics of our task, we design a neural machine translator which integrates a vision Convolutional Neural Network (CNN) [35, 57], a Recurrent Neural Network (RNN) encoder and a RNN decoder [15, 60] in a unified framework. As shown in Figure 3, given an input UI image, the vision CNN extracts a diverse set of image features through a sequence of convolution and pooling operations. The RNN encoder then encodes the spatial layout information of these image features to a summary vector C, which is then used by the RNN decoder to generate the GUI skeleton in token sequence representation.

3.1 Feature Extraction by Vision CNN

To learn about visual features and patterns of numerous UI elements from a vast amount of UI images, we need a sophisticated model with capability of visual understanding. Convolutional Neural Networks (CNNs) constitute one such class of models. CNN is inspired by the biological findings that the mammal's visual cortex has small regions of cells that are sensitive to specific features of visual receptive field [31, 32]. These cells act as local filters over the input space and they visually perceive the world around them using a layered architecture of neurons in the brain [23]. CNN is designed to mimic this phenomenon to exploit the strong spatially local correlation present in images.

A CNN is a sequence of layers that transform the original image spatially into a compact feature representation (called *feature map*) [35, 37]. In our CNN architecture, we use two main types of layers: *convolutional layer (Conv)* and *pooling layer (Pool)*, following the pattern $Conv \rightarrow Pool$. We stack a few $Conv \rightarrow Pool$ layers to create a deep CNN, because a deep CNN can extract more powerful features of the input images [35, 57, 61]. We do not use fully connected layers, since we want to preserve the locality of CNN features in order to encode their spatial layout information later.

3.1.1 *Convolutional Layer*

A convolutional layer accepts an input volume $I \in \mathbb{R}^{W_I H_I D_I}$ where W_I, H_I and D_I are the width, height and depth of the input volume respectively. If the first convolutional layer takes as input an image, then W_I and H_I is the width and height of the image, and D_I is 1 for gray-scale image, or 3 for RGB color image (i.e., red, green, blue channels respectively). Each cell of the input volume is a pixel value from 0 to 255. The cell values of the input volume for the subsequent convolutional layers depend on the convolution and pooling operations of the previous layers.

A convolutional layer performs convolution operations using filters (or kernels). A filter is a neuron that learns to look for some visual features (e.g., various oriented edges) in the input. The filters in a layer will only be connected to the local regions of the input volume. The spatial extent of this local connectivity is called the receptive field of the neuron (i.e., filter size). The extent of the connectivity along the depth is equal to the depth D_I of the input volume. The convolution operation performs dot products of a filter and the local regions of the input followed by a bias $b \in \mathbb{R}$ offset. We apply the non-linear activation function $ReLU(x) = max(0, x)$ [44] to the output of a convolution operation. We perform zero-padding around the border of the input volume so that the information at the border will also be preserved in convolution [49].

Below is an example of applying a 3×3 filter to a 3×3 region of a gray-scale image (i.e., $D_I = 1$) followed by $ReLU$ activation:

$$Conv\left(\begin{bmatrix} p_1 & p_2 & p_3 \\ p_4 & p_5 & p_6 \\ p_7 & p_8 & p_9 \end{bmatrix}, \begin{bmatrix} w_1 & w_2 & w_3 \\ w_4 & w_5 & w_6 \\ w_7 & w_8 & w_9 \end{bmatrix}\right) = max(0, \sum_{n=1}^{9} p_n w_n + b)$$

where p_n is the pixel value and w_n is the weight of the filer. The resulting convolution value represents the activation of a neuron over a region of the input image. Intuitively, this value represents the likelihood of a neuron "seeing" a particular visual feature over the region. During the model training, the CNN will learn filters that activate when they "see" various visual features, such as an edge of some orientation on the first convolutional layer, and shape-like patterns (e.g., rectangle, circle) and more abstract visual patterns (e.g., image region, text) on higher layers of the network [61].

A convolutional layer can have K filters. The K filters that are applied to the same region of the input produce K convolution values. These K values form a *feature vector*, representing the observations of all K neurons over this particular region of the image. A filter is applied to each possible local regions of the input, specified by the stride S (the number of pixels by which we slide the filter horizontally and vertically). This produces a *kernel map* containing the values of performing a convolution of the d-th filter over the input volume with a stride of S. Intuitively, a kernel map represents the observations of a neuron over the entire image. All K kernel maps form a *feature map* of a convolutional layer.

3.1.2 *Pooling Layer*

Pooling layers take as input the output feature map of the preceding *Conv* layers and produce a spatially reduced feature map. They reduce the spatial size of the feature map by summarizing the values of neighboring groups of neurons in the same input kernel map. A pooling layer consists of a grid of pooling units spaced S pixels apart, each summarizing a region of size $Z \times Z$ of the input volume. Different from the filters in the *Conv* layers, pooling units have no weights, but implement a fixed function. In our architecture, we adopt *1-max pooling* [43] which takes the maximum value in the $Z \times Z$ region. As the pooling layer operates independently on every input kernel map, the depth dimension of the output feature map remains the same as that of the input feature map.

Pooling layers progressively reduce the spatial size of the feature map to reduce the amount of parameters and computation in the network, and hence to also control overfitting. Meanwhile, pooling also keeps the most salient information as it preserves the maximum value of each region in the depth slice. It also benefits to the

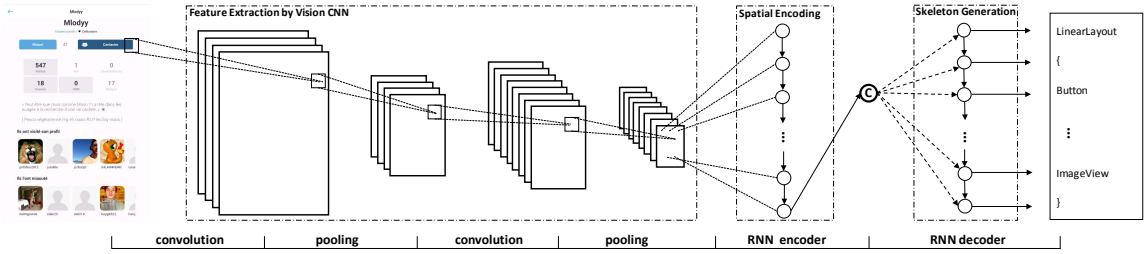


Figure 3: Architecture of Our Neural Machine Translator for UI-Image-to-GUI-Skeleton Generation

invariance to shifting, rotation and scaling. Even if the image is shifted/rotated/scaled by a few pixels, the output by max operation will still stay the same when pooling over a region.

3.2 Spatial Layout Encoding by RNN Encoder

Our neural machine translator takes as input only a raw UI image, requiring no detailed annotations of the structure and positions of visual elements. Therefore, given the feature map outputted by the vision CNN, it is important to localize the relative positions of visual features within the input image for the effective generation of proper GUI components and their compositions. To encodes spatial layout information of CNN features, we run a Recurrent Neural Network (RNN) encoder over each of the feature vectors of the feature map outputted by the vision CNN.

The input to a RNN is a sequence of vectors (e.g., words in a sentence in the application of RNNs to Natural Language Processing tasks). To apply the RNN model in our task, we convert the feature map $FM \in \mathbb{R}^{WHD}$ outputted by the vision CNN into a sequence of D-dimensional feature vectors (D is the depth of the feature vector). The length of this image-based sequence is $W \times H$ (i.e., the width and height of the feature map). The conversion can be done by scanning the feature map along the width axis first and then the height axis, or vice versa. In order to capture the row (or column) information in the sequence, we insert a special vector at the end of each row (or column) (can be thought of as a "." in text), which are referred to as positional embeddings.

An RNN recursively maps an input vector x_t and a hidden state h_{t-1} to a new hidden state h_t: $h_t = f(h_{t-1}, x_t)$ where f is a non-liner activation function (e.g., a LSTM unit discussed below). After reading the end of the input, the hidden state of the RNN encoder is a vector C summarizing the spatial layout information of the whole input feature map. Modeling long-range dependencies between CNN features is crucial for our task. For example, we need to capture the dependency between the bottom-right and top-left features of a visual element in an image-base sequence. Therefore, we adopt Long Short-Term Memory (LSTM) [29]. An LSTM consists of a memory cell and three gates, namely the input, output and forget gates. Conceptually, the memory cell stores the past contexts, and the input and output gates allow the cell to store contexts for a long period of time. Meanwhile, some contexts can be cleared by the forget gate. This special design allows the LSTM to capture long-range dependencies, which often occur in image-based sequences.

3.3 GUI Skeleton Generation by RNN Decoder

Based on the RNN-encoder's output summary vector C, the target tokens of the GUI framework language (i.e., the names of GUI components and the special tokens { and }) are then generated by a decoder. The token sequence representation of a GUI skeleton

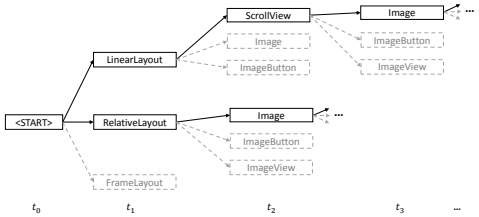


Figure 4: An illustration of beam search (beam width=2)

can be converted to a component hierarchy via depth-first traversal (DFT) as seen in Figure 2. As the length of the generated token sequence varies for different UI images, we adopt a RNN decoder which is capable of producing a variable-length sequence. The hidden state of the decoder at time t is computed as $f(h_{t-1}, y_{t-1}, C)$ (f is also LSTM). The conditional distribution of the next token y_t is computed as $P(y_t|\langle y_{t-1}, ..., y_1\rangle, C) = softmax(h_t, y_{t-1}, C)$, where $softmax$ function produces valid probabilities over the language vocabulary. Note that the hidden state and the next token are not only conditioned on past contexts, but also the summary vector C of the CNN features of the input image.

3.4 Model Training

Although our neural machine translator is composed of three neural networks (a vision CNN, a spatial layout RNN encoder, and a GUI skeleton generation RNN decoder), these networks can be jointly trained end-to-end with one loss function. The training data consists of pairs of UI images i and corresponding GUI skeletons s (see Section 4 for how we construct a large-scale training dataset). The GUI skeleton is represented as a sequence of tokens $s = \{s_0, s_1, ...\}$ where each token comes from a GUI framework language (i.e., the names of GUI components and the two special tokens { and }). Each token is represented as a one-hot vector.

Given a UI image i as input, the model tries to maximize the conditional probability $p(s|i)$ of producing a target sequence of the GUI skeleton $s = \{s_0, s_1, ...\}$. Since the length of s is unbounded, it is common to apply the chain rule to model the joint log probability over $s_0, ..., s_N$, where N is the length of a particular sequence as:

$$\log p(s|i) = \sum_{t=0}^{N} \log p(s_t|i, \langle s_0, ..., s_{t-1}\rangle) \quad (1)$$

At training time, we optimize the sum of log probabilities over the whole training set using stochastic gradient descent [11]. RNN encoder and decoder backpropagates error differentials to its input, i.e., the CNN, allowing us to jointly learn the neural network parameters to minimize the error rate in a unified framework.

3.5 GUI Skeleton Inference

After training the neural machine translator, we can use it to generate the GUI skeleton s for a UI design image i. The generated GUI skeleton should have the maximum log probability $p(s|i)$. Generating a global optimal GUI skeleton has an immerse search space. Therefore, we adopt the beam search [34] to expand only a limited set of the most promising nodes in the search space. As illustrated in Figure 4, at each step t, the beam search maintains a set of the k (beam-width) best sequences as candidates to generates sequences of size $t+1$. At the step $t+1$, the neural machine translator produces a probability distribution over the language vocabulary for expanding each of the current candidate sequences to new candidate sequences of size $t+1$. The beam search keeps only the best k sequences among all new candidate sequences of size $t+1$. This process continues until the model generates the end-of-sequence symbol for the k best sequences. Then, the top-1 ranked sequence is returned as the generated GUI skeleton for the input UI image.

4 COLLECTING LARGE-SCALE MODEL TRAINING DATASET

To train our neural machine translator, we need a large set of pairs of UI images and GUI skeletons from existing mobile apps. This requires us to explore the apps' GUIs, take UI screenshots, obtain runtime GUI component hierarchies, and associate screenshots with component hierarchies. Although some tools (e.g., Apktool [7], UI Automator [24]) may assist these tasks, none of them can automate the whole data collection. Inspired by automated GUI testing techniques [16], we develop an automated technique, termed Stoat [53], to explore the GUIs. During exploration, the UI screenshots paired with their corresponding runtime GUI component hierarchies will be automatically dumped. The dumped UI images and corresponding GUI component hierarchies are like the example in Figure 2.

4.1 Exploring Application GUIs

Mobile apps are event-centric with rich GUIs, and users interact with them by various actions (*e.g.*, `click`, `edit`, `scroll`). Stoat emits various UI events to simulate user actions, and automatically explore different functionalities of an app. To thoroughly explore an app's GUIs, our data collector tries to identify *executable GUI components* (*e.g.*, *clickable*, *long-clickable editable*, *scrollable*) on the current UI and infer actions from these components' type. For example, if the UI contains a `Button`, Stoat can simulate a `click` action on it. However, mobile platforms like Android also permit developers to implement actions in the app code. For example, a `TextView` widget may be registered with a `LongClick` action, which is invisible in the UI. Without incorporating these implicit actions, we may fail to execute some app functionalities (*i.e.*, miss some UI images). To overcome this issue, we integrate static analysis method (e.g., [9]) to scan app code and detect actions that are either registered with UI widgets (*e.g.*, `setOnLongClickListener`) or implemented by overriding class methods (*e.g.*, `onCreateOptionsMenu`).

Figure 5 shows an example of using Stoat to explore an Android app (*Budget [12]*) and collect the required data. Starting from the *Main Page* which lists the balance of each expense group (e.g., Baby, Bill, Car), Stoat can "click" ↑ which opens the *Distributed Page*. On the *Distributed Page*, Stoat can "edit" the amount of money distributed to an expense group (e.g., Baby). It can also "scroll" the

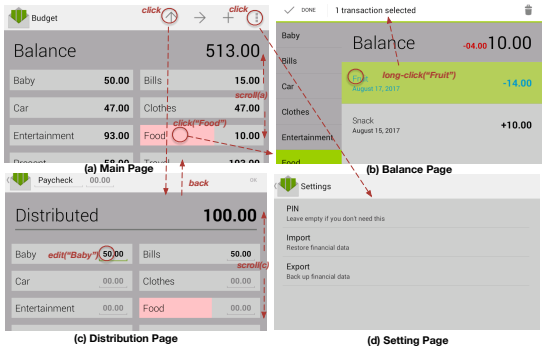


Figure 5: Automatically exploring an app's GUIs

Distributed Page to show more expense groups or "click" the back button (the hardware back button on the phone) to go back to the *Main Page*. On the *Main Page*, Stoat can also "click" an expense group which opens the *Balance Page* or "click" the setting button which opens the *Setting Page*. On the *Balance Page*, it can "longclick" a transaction (e.g., Fruit) to select it.

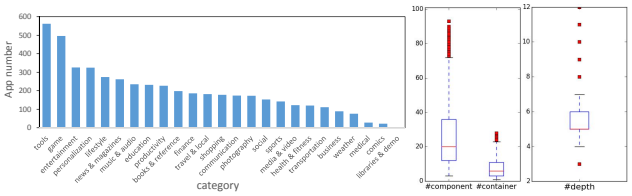
4.2 Prioritizing UI exploration

Figure 5 shows there are often several executable components/actions on a UI. To determine which component/action to execute, Stoat implements a *prioritized UI exploration* method. We conduct a formative study of the GUIs of 50 Google Play apps from 10 categories (*e.g.*, *Communications*, *Personal*, *Tools*), and summarize three key observations that can affect the UI exploration performance: (a) *Frequency of action*. Each action should be given chance to execute. When an action is more frequently executed than others, its priority should be lowered. (b) *Number of subsequent UIs*. If an action exhibits more subsequent UIs after its execution, its should be prioritized in future so that more new functionalities can be visited. (c) *Type of action*. Different types of actions should be given different priorities. For example, a *hardware back* or a *scroll* action should be executed at right time. Or, it may discard the current UI page and prevent the execution of other normal actions (*e.g.*, `edit`).

Based on these observations, we assign each action on an executable component on a UI with an execution weight, and dynamically adjust this value at runtime. The action with the highest weight value will be queued as next action to execute. The weight of an action is determined by the formula below: $execution\_weight(a) = (\alpha * T_a + \beta * C_a)/\gamma * F_a$ where a is the action, T_a is the weight of different types of actions (1 for normal UI actions (e.g., `click`, `edit`), 0.5 for `hardware back` and `scroll`, and 2 for menu selection), C_a the number of unexplored components on the current UI, F_a is the times that a has been executed, and α, β and γ are the weight parameters which can be determined empirically.

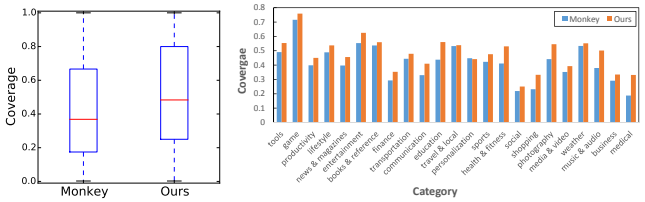
4.3 Excluding Duplicate UIs

After simulating an action, our data collector takes the screenshot of the current UI and dump its runtime GUI component hierarchy. As seen in Figure 5, the same UI may be visited many times, for example, to execute different features on the *Main Page*, or to view the same expense group on the *Balance Page* again. Furthermore, after an action, the app may stay on the same UI, for example, after editing Baby amount on the *Distribution Page*. To collect as diverse UI images and GUI component hierarchies as possible, we should avoid collecting such duplicate UIs.



(a) App category (b) GUI skeleton complexity

Figure 6: Android UI dataset statistics



(a) AC distribution (b) Average AC over app category

Figure 7: Effectiveness of Automated GUI Exploration

To that end, we compare the newly collected pair of UI image and GUI component hierarchy $\langle i_{new}, s_{new} \rangle$ with already collected pairs $\langle i_a, s_a \rangle$. As comparing images is time-consuming, we compare component hierarchies. We convert GUI component hierarchies into their token sequence representation by a depth-first traversal, and then compute the hash values for the resulting token sequences. Only if the hash value of s_{new} does not match that of any s_a, the newly collected pair $\langle i_{new}, s_{new} \rangle$ will be kept. Otherwise, it will be discard. For example, when going back from the *Balance Page* to the *Main Page*, the *Main Page* UI data will not be collected again. As another example, after editing Baby amount to a different value, the *Balance Page* UI image will be slightly different, but the GUI component hierarchy remains the same. Therefore, the *Balance Page* UI data will not be collected after editing Baby amount.

Some UI actions may change the UI's runtime component hierarchies even the app stays on the same UI after the actions, like deleting an expense group or scrolling the *Distribution Page* so that different numbers of expense groups are visible. If the resulting GUI component hierarchy has not been collected before, the newly collected pair $\langle i_{new}, s_{new} \rangle$ will be kept. In such cases, the UI images before and after the actions may be similar, but will not be identical.

5 CONSTRUCTING ANDROID UI DATASET

We implemented our automated Android UI data collector, Stoat, as described in Section 4. Stoat uses Android emulators (configured with the popular KitKat version, SDK 4.4.2, 768×1280 screen size) to run Android apps. It uses `Android UI Automator` [24, 28] to dump pairs of UI images and corresponding runtime GUI component hierarchies. Soot [20] and Dexpler [9] are used for static analysis. Stoat runs on a 64-bit Ubuntu 16.04 server with 32 Intel Xeon CPUs and 189G memory, and controls 16 emulators in parallel to collect data (each app is run for 45 minutes).

5.1 Dataset of Android Application UIs

We crawl 6000 Android apps [41] with the highest installation numbers from Google Play. 5043 apps runs successfully by Stoat and they belong to 25 categories. Figure 6(a) shows the number of apps in each category. The other 957 apps require extra hardware support or third-party libraries which are not available in the emulator.

Stoat collected totally 185,277 pairs of UI images and GUI skeletons (on average about 36.7 pairs per app)[1]. This UI dataset is used for training and testing our neural machine translator (see Section 6). The collected GUI skeletons use 291 unique Android GUI components, including Android's native layouts and widgets and those from third-party libraries. The box plots in Figure 6(b) shows the complexity of the collected GUI skeletons which varies greatly. On average, a GUI skeleton has 24.73 GUI components, 7.43 containers (non-leaf components) and 5.43 layers (the longest path from the root to a leaf component).

5.2 Effectiveness of Automated UI Exploration

To train a "knowledgeable" neural machine translator, we need a diverse set of UIs. Note that we already exclude duplicate UIs during data collection (see Section 4.3). Therefore, the diversity of the collected UI data depends on Stoat's ability to thoroughly explore an app's GUIs. To confirm the UI-exploration effectiveness of Stoat, we compared it with *Monkey* [25], an automated GUI testing tool developed by Google and released with Android SDKs.

We use *Activity Coverage* (AC), rather than code coverage criteria [54, 62], to evaluate the UI-exploration effectiveness. As android apps are composed of activities, which are responsible for rendering UI pages, AC can measure the percentage of how many different activities (UI pages) have been explored. We randomly selected 400 apps from our crawled apps, and apply both tools on them. To achieve a fair comparison, we allocate the same exploration time (45 minutes) for the two tools. Figure 7(a) shows the AC values achieved by the two tools in box plots, and Figure 7(b) presents the average AC values over different app categories. On average, Stoat achieves 0.513 AC, 12.7% higher than Monkey (0.455). Among all 23 categories of these 400 apps, Stoat also outperforms Monkey.

6 EVALUATION

We evaluate our neural machine translator in three aspects, *i.e.*, accuracy, generality and usefulness as follows.

6.1 Implementing Neural Machine Translator

We implement the proposed neural machine translator with six $Conv \rightarrow Pool$ layers in the CNN model. The first *Conv* layer uses 64 filters, and each subsequent layer doubles the number of filers. This is because higher *Conv* layers have to capture more abstract and diverse visual features [36, 61], and thus need more neurons. Following the CNN layer sizing patterns [49] for vision tasks, we set the filter size 3, the stride 1 and the amount of zero padding 2 for convolutional layers. This setting allows us to leave all spatial downsampling to the *Pool* layers, with the *Conv* layers only transforming the input volume depth-wise (determined by the number of filters of a *Conv* layer). For the pooling layers, we use the most common form of pooling layer setting [49], i.e., pooling units of size 2×2 applied with a stride 2. This setting downsamples every kernel map by 2 along both width and height, discarding 75% of the neurons. For the RNN encoder and decoder, there are 256 hidden LTSM units to store the hidden states.

We implement our model based on the Torch [17] framework written in Lua. We train the model using randomly selected 90%

[1]Dataset can be downloaded in http://tagreorder.appspot.com/ui2code.html.

of the Android UI dataset (i.e., 165,887 pairs of UI images and GUI skeletons), and fine-tune model hyperparameters (the number of CNN layers, the number of *Conv* layer filers and the number of RNN hidden states) using another randomly selected 3% of Android UI dataset. The model is trained in a Nvidia M40 GPU (24G memory) with 10 epochs for about 4.7 days. At inference time, our translator can generate GUI skeletons for 20 UI images per second, which is about 180 times faster than existing UI reverse engineering techniques [45] based on traditional computer vision techniques (e.g., edge detection [13], Optical-Character-Recognition (OCR) [51]).

6.2 Evaluation Metric

Let $\langle i_t, s_t \rangle$ be a pair of UI image and GUI skeleton in the testing dataset. We say s_t is the ground-truth GUI skeleton for the UI image i_t. Let s_g be the generated GUI skeleton for the i_t using our neural machine translator. Both s_t and s_g are in their token sequence representation (see Section 2) in our evaluation.

The first metric we use is *exact match rate*, i.e., the percentage of testing pairs whose s_t exactly match s_g. Exact match is a binary metric, i.e., 0 if any difference, otherwise 1. It cannot tell the extent to which a generated GUI skeleton differs from the ground-truth GUI skeleton. For example, no matter one or 100 differences between the two GUI skeletons, exact match will regard them as 0.

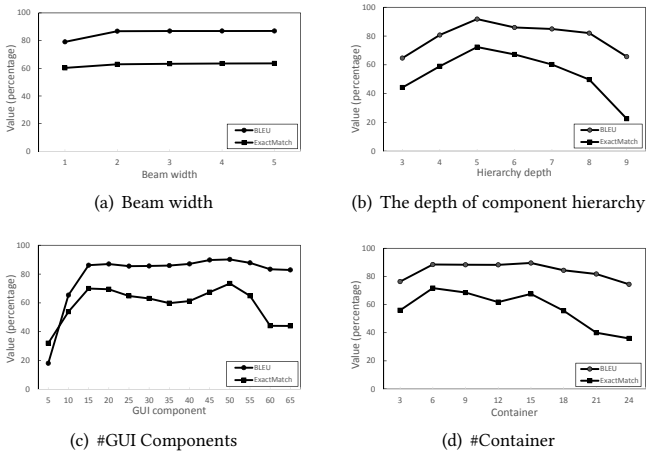
Therefore, we adopt BLEU (BiLingual Evaluation Understudy) [47] as another metric. BLEU is an automatic evaluation metric widely used in machine translation studies. It calculates the similarity of machine-generated translations and human-created reference translations (i.e., ground truth). BLEU is defined as the product of n-gram precisionİ and brevity penalty: $BLEU = BP * \exp\left(\sum_{n=1}^{N} w_n log p_n\right)$ where each p_n is the precision of the n-grams, i.e., the ratio of length n token subsequences generated by the machine translator that are also present in the ground-truth translation. w_n is the weight of different length of n-gram summing to one. It is a common practice [55] to set N as 4 and $w_n = \frac{1}{N}$. BP is the brevity penalty which prevents the system from creating overly short hypotheses (that may have higher n-gram precision). BP is 1 ($c > r$), otherwise $e^{(1-r/c)}$ where r is the length of ground-truth translation, and c is the length of machine-generated translation. BLEU gives a real value with range [0,1] and is usually expressed as a percentage. The higher the BLEU score, the more similar the machine-generated translation is to the ground truth translation. If the translation results exactly match the ground truth, the BLEU score is 1 (100%).

6.3 Accuracy Evaluation

We use randomly selected ~7% of Android UI dataset (10804 pairs of UI images and GUI skeletons) as test data for accuracy evaluation. None of the test data appears in the model training data.

6.3.1 *Overall Performance*

As seen in Figure 8(a), among all 10804 testing UI images, the generated GUI skeletons for 6513 (60.28%) UI images exactly match the ground truth GUI skeletons, and the average BLEU score over all test UI images is 79.09, when the beam width is 1 (i.e., greedy search). Furthermore, for only 9 of all test UI images, our model fails to generate closed brackets. This result shows that our model successfully captures the composition information of container components. When the beam width increases to 5, the exact match rate and the average BLEU score increase to 63.5% and 86.94, respectively. However, the increase after $beam - width = 2$ is marginal. Therefore, we use $beam - width = 2$ in the following experiments considering a balance of computation cost and accuracy.



(a) Beam width (b) The depth of component hierarchy

(c) #GUI Components (d) #Container

Figure 8: Impact of beam-width and generation target complexity

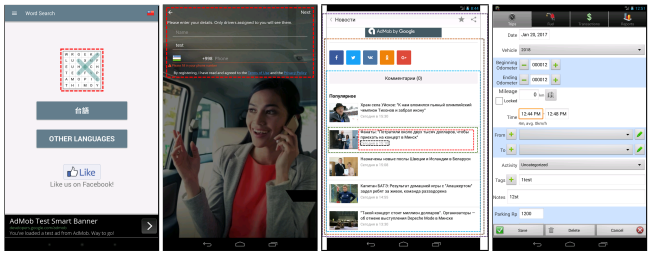
6.3.2 *Performance by Generation Target Complexity*

As we show in Figure 6(b), the ground-truth GUI skeletons (i.e., the targets to generate) in our dataset vary greatly in terms of the number of GUI components (#GUI components), the number of container components (#containers/compositions), and the depth of component hierarchy. These three dimensions define the complexity of GUI skeleton generation tasks. To better understand the capability of our neural machine translator, we further analyze the accuracy of our translator for the ground-truth GUI skeletons with different #GUI components, #containers and depth.

As #GUI components has a wide range, we bucket the ground-truth GUI skeletons by 5 intervals of #GUI components (i.e., 1-5, 6-10, ...). Similarly, we bucket the ground-truth GUI skeletons by 3 intervals of #containers (i.e., 1-3, 4-6, ...). We average the exact match rate or the BLEU score of a generated GUI skeleton and the corresponding ground-truth GUI skeleton for each bucket. Figure 8(b), 8(c) and 8(d) present the results.

Intuitively, the more GUI components and the more containers to generate, the deeper of a component hierarchy to generate, the more challenging a generation task is. However, our results show that our translator works very well for a wide range of generation target complexity. The BLEU score remains very stable (above 80) when there are 15 or more GUI components to generate, 6 to 21 containers to generate, and/or 4 to 8 depth of component hierarchies to generate. The test data in these ranges accounts for 66.75%, 72.37% and 77.54% of all test data, respectively. The exact match rate remains around or above 60% for a relatively narrower range of #GUI components (15-55), #containers (6-15) or the depth of component hierarchies (4-7). Although our translator is less likely to generate an exact match when the GUI skeleton to generate is too complex (> 55 GUI components, > 15 containers and/or > 7 depth), it can still generate a GUI skeleton that match largely with the ground truth (i.e., high BLEU scores).

A surprising finding is that our translator's accuracy degrades when the GUI skeletons is too simple (≤ 10 GUI components, ≤ 3



(a) Text-like image (b) UI elements on background image (c) Deep component hierarchy (d) Complex spatial layout

Figure 9: Examples of visual understanding and generation capability

containers, and/or ≤ 3 depth). We will elaborate the common causes for this result in Section 6.3.4.

6.3.3 *Analysis of Visual Understanding and Generation Capability*

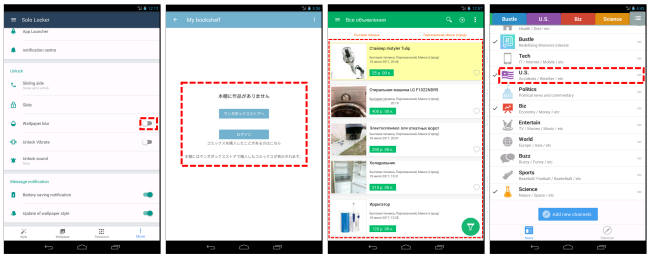
We randomly sampled 10% of the generated GUI skeletons for manual observation. We manually study the differences between these generated GUI skeletons and their ground truth (input UI images). This section analyzes our translator's visual understanding and GUI skeleton generation capability. Section 6.3.4 summarizes common causes of generation errors.

We find that our translator can reliably distinguish different types of visual elements and generate the right GUI components. Figure 9(a) and 9(b) show two challenging cases. Figure 9(a) shows the setting UI of a puzzle game in which the game icon (highlighted in red box), contains a table of characters. Our translator correctly recognizes the region in the red box as an image and generates a *ImageView* for it instead of *TextView*. The UI in Figure 9(b) contains a background image with some UI elements in the foreground (highlighted in red box). Our translator correctly teases apart the foreground elements and the background image, rather than considering the UI elements as part of the background image.

An interesting observation is that our translator can reliably determine what text elements in UI images look like even when the texts are written in different languages (e.g., Figure 9(c) and Figure 10(b)). During automated UI exploration, different language settings of an app may be triggered so that we can collect UI images containing texts of different languages. For the GUI skeleton generation task, the exact text content does not matter much. Our translator can abstract language-independent text-like features in UI images, which makes it language independent.

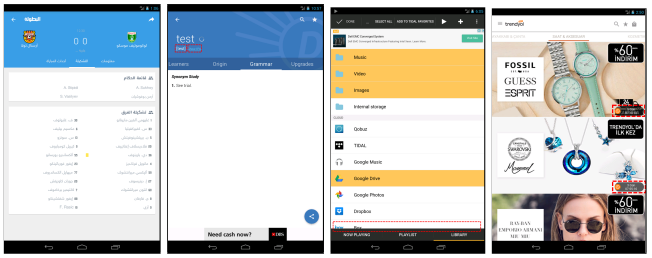
We find that our translator is robust to complex spatial layout of UI elements in a UI design. Figure 9(c) shows a UI design that requires 6 depth of component hierarchy, and Figure 9(d) show a UI design with 60 GUI components that vary in shape, size and alignment. For both cases, our translator generates the exact-match GUI skeleton as the ground truth.

We observe that many differences between the generated and ground-truth GUI skeletons represent alternative implementations, rather than generation errors. For example, the ground truth for the control in the red box in Figure 11(a) is a *ToggleButton*, while our translator generates a *Switch* for the control. *ToggleButton* and *Switch* would be interchangeable for implementing this control. Figure 11(b), 11(c) and 11(d) show three examples of using different Android layouts: compose components in a layout (generated) or



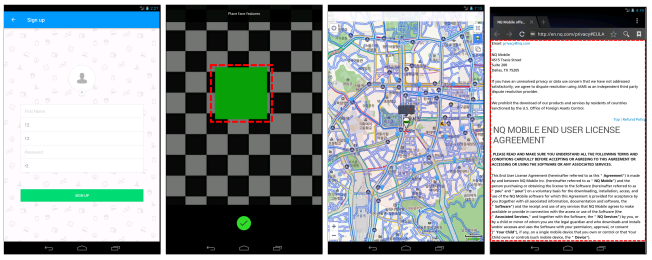
(a) ToggleButton or Switch (b) Using layout or hard-positioning (c) ListView or RecyclerView (d) LinearLayout or RelativeLayout

Figure 10: Examples of alternative implementations



(a) Many similar UI elements (b) Similar texts on one line (c) Partially visible UI elements (d) Image-like UI elements

Figure 11: Common causes of generation errors



(a) Low contrast background (b) Little context information (c) Displayed content as UI to generate (d) Displayed content as UI to generate

Figure 12: Common causes of generation errors (simple UIs)

hardcode position components (ground-truth), use *ListView* (generated) or newer API *RecyclerView* (ground-truth), or use *LinearLayout* (generated) or *RelativeLayout* (ground-truth). Which option is more appropriate for an app depends on the app's development history and usage scenarios, but both options would produce the same spatial layout effects of the UI designs.

6.3.4 *Common Causes for Generation Errors*

Our qualitative analysis identifies some common causes of generation errors. First, when a UI has many similar UI elements (e.g., Figure 11(a)), our translator sometimes may not generate the exact same number of GUI components. Second, when several neighborhood texts in one line use similar fonts and styles (e.g., Figure 11(b)), our translator may regard them as one text component. Third, when a UI element is only partially visible (e.g., covered by suspension menu in Figure 11(c)), our translator may not recognize the blocked UI element. Fourth, our model sometimes cannot discriminate small UI elements on top of a complex image especially when the UI elements has similar visual features to some parts of the image (e.g., the red box in Figure 11(d)).

In addition, we identify two common causes for the degraded accuracy on generating simple UIs. First, UI elements in simple

Table 1: The accuracy results for 20 completely unseen apps

ID	App name	Category	#Installation	#Image	ExactMatch (%)	BLEU
1	Advanced Task Killer	Productivity	50M-100M	89	78.65	95.05
2	ooVoo Video Call, Text&Voice	Social	50M-100M	46	78.26	95.75
3	ColorNote Notepad Notes	Productivity	50M-100M	53	77.36	95.79
4	4shared	Entertainment	100M-500M	57	71.93	88.72
5	Badoo - Meet New People	Social	50M-100M	28	71.42	93.16
6	Mono Bluetooth Router	Music & Audio	1M-5M	79	69.62	99.17
7	Automatic Call Recorder	Tools	50M-100M	59	69.49	94.71
8	Flashlight HD LED	Tools	50M-100M	42	66.67	91.42
9	Solitaire	Game	50M-100M	30	66.67	85.83
10	AVG AntiVirus	Communication	100M-500M	68	60.29	86.66
11	ASKfm	Social	50M-100M	52	59.62	92.90
12	Color X Theme-ZERO launcher	Personalization	1M-5M	49	59.18	78.53
13	Smart Connect	Tools	100M-500M	31	58.06	72.83
14	History Eraser-Privacy Clean	Tools	10M-50M	70	57.14	96.00
15	Pixlr-Free Photo Editor	Photography	50M-100M	59	47.46	77.88
16	SoundCloud-Music & Audio	Music & Audio	100M-500M	49	44.90	80.19
17	Office Documents Viewer (Free)	Personalization	1M-5M	96	42.71	88.04
18	Super Backup : SMS&Contacts	Tools	5M-10M	92	40.22	93.55
19	Photo Effects Pro	Photography	50M-100M	90	37.78	88.14
20	Mobile Security & Antivirus	Tools	100M-500M	69	30.43	67.94
	AVERAGE			60.4	59.39	88.11

UIs often contain little context information for determining the appropriate GUI components for them. For example, the green rectangle in Figure 12(b) is actually a button, but our translator mistakes it as an image. Other design factors like low contrast background in Figure 12(a) could be more problematic for a simple UI with little context information. Second, some mobile apps, like map navigation and web browser, have very simple main UI with just several high-level encapsulated components inside, but the content being displayed in the component can be rather complex. For example, the navigation map in Figure 12(c) and a web page in Figure 12(d) can be displayed by one GUI component such as *MapView* and *WebView*. However, our translator may mistake the content displayed as part of the UI to implement, and generate unnecessary basic GUI components.

6.4 Generality Evaluation

To further confirm the generality of our translator, we randomly select another 20 apps that are not in our UI dataset. To ensure the quantity of test data, apps that we select have at least 1 millon installations (popular apps often have rich-content GUIs). Among these apps, we randomly select 20 apps for which our data collector collects more than 20 UI images. These 20 apps belong to 10 categories. We collect in total 1208 UI images (in average 60.4 per app). We set beam width as 2 for generating GUI skeletons.

Table 1 summarizes the information of the selected 20 apps and the accuracy results of the generated GUI skeletons on the UI images of these apps (sorted by exact match rate in descending order). The average exact match rate is 59.47% (slightly lower than the average exact match rate (63.5%) of Android UI test data), and the average BLEU score is 88.11 (slightly higher than the average BLEU score (86.94) of Android UI test data). These results demonstrate the generality of our translator.

We manually inspect the apps with low exact match rate (< 50%) or BLEU score (< 80). We observe similar causes for generation errors as those discussed in Section 6.3.4. For example, personalization and photography apps have UIs for users to upload and manipulate documents or images. Similar to the map and web page examples in Figure 12(c) and Figure 12(d), our translator may mistake some content displayed in the UIs as part of the UIs to generate, which results in low exact match rate or BLEU score for these apps. However, although the exact match rate is low for some such apps (e.g., 17-Office Document Viewer, 19-Photo Effects Pro), the BLEU score is high which indicates that the generated GUI skeletons still largely match the ground truth.

The app *Mobile Security & Antivirus* is an interesting case, which has the lowest exact match rate and the lowest BLEU score. We find that its developers use *LinearLayout* or *RelativeLayout* rather randomly when the two layouts produce the same UI effect (similar to the example in Figure 10(d)). In contrast, our translator tends to use one GUI component consistently for a type of UI spatial layout, for example just *LinearLayout*, which results in many mismatches between the generated GUI skeletons and the actual implementation of *Mobile Security & Antivirus*.

6.5 Usefulness Evaluation

We conduct a pilot user study to evaluate the usefulness of the generated GUI skeleton for bootstraping GUI implementation.

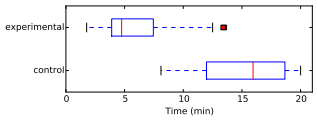
6.5.1 Procedures

We recruit eight PhD students and research staffs from our school. We ask each participant to implement the same set of 5 UI images in Android. We select two relatively simple UI design images, two medium-complex images, and one complex image for the study. Participants need to implement only a skeleton GUI that replicates UI elements and their spatial layout in a UI image, without the need to set up components' properties (e.g., font, color, padding, etc.). The study involves two groups of four participants: the experimental group P_1, P_2, P_3, P_4 who start with the generated GUI skeletons by our tool, and the control group P_5, P_6, P_7, P_8 who start from scratch. According to pre-study background survey, all participants have more than two-years Java and Android programming experience and have developed at least one Android application for their work. Each pair of participants $\langle P_x, P_{x+4} \rangle$ have comparable development experience so that the experimental group has similar expertise to the control group in total. Participants are required to use Android Studio to avoid tool bias and have up to 20 minutes for each design.

We record the time used to implement the UI design images. After each UI image's implementation, participants are asked to rate how satisfied they are with their implementation in five-point likert scale (1: not satisfied at all and 5: highly satisfied). After the experiment, we ask a research staff (not involved in the study) to judge the similarity of the implemented skeleton GUIs to the respective UI images (also five-point likert scale, 1: not similar at all and 5: identical layout). This judge does not know which skeleton GUI is implemented by which group.

6.5.2 Results

Box plot in Figure 13 shows that the experiment group implements the skeleton GUIs faster than the control group (with average 6.14 minutes versus 15.19 minutes). In fact, the average time of the control group is underestimated, because three participants fail to complete at least one UI image within 20 minutes, which means that they may need more time in the real development. In contrast, all participants in the experiment group finish all the tasks within 15 minutes. The experimental group rates 90% of their implemented GUIs as highly satisfactory (5 point), as opposed to 15% highly satisfactory by the control group. This is consistent with the similarity ratings of the implemented GUIs to the UI images given by the judge. On average, the satisfactory ratings for the experiment and control group is 4.9 versus 3.8, and the similarity ratings for the experiment and control group is 4.2 versus 3.65.



Measures	Control	Experimental
Time (minutes)	15.19	6.13*
Satisfactoriness	3.8	4.9*
Similarity	3.65	4.2**

Figure 13: The comparison of the experiment and control group. * denotes $p < 0.01$ and ** denotes $p < 0.05$

We believe the above results are because the generated GUI skeletons by our tool give participants a reliable starting point for GUI implementation. Guided by the generated GUI skeletons, participants are clear about what components to use and how to compose them in a proper order. Then, they mainly need to fix some generation errors and make some adjustment of component layouts. Without the help of the generated GUI skeletons, the control group has to determine in a trial-and-error manner what components to use and how to compose them, which results in the longer implementation time and less satisfactory GUI implementations.

To understand the significance of the differences between the two groups, we carry out the Mann-Whitney U test [22] (specifically designed for small samples) on the implementation time, satisfactory and similarity ratings. The test results in Figure 13 table suggests that our tool can significantly help the experimental group implements skeleton GUIs faster ($p - value < 0.01$), creates more satisfactory GUIs ($p - value < 0.01$) that are more similar to the UI design images ($p - value < 0.05$). According to our observations, starting with the generated GUI skeletons, even the less experienced participants in the experimental group achieve the comparable performance to the most experienced participant in the control group. Although by no means conclusive, this user study provides initial evidence of the usefulness of our approach for bootstraping GUI implementation.

7 RELATED WORK

UI design and implementation require different mindset and expertise. The former is performed by user experience designers and architects via design tools (*e.g.*, Sketch [4], Photoshop [1]), while the latter performed by developers via development tools (*e.g.*, Android Studio [2], Xcode [5]). Our work lowers the transition barrier from UI design images (the artifacts from UI design) to GUI skeletons (the starting point of GUI implementation). Existing tools well support these two phases respectively, but none of them supports effective transition from UI design images to GUI skeletons.

Supporting this transition is challenging due to the diversity of UI designs and the complexity of GUI skeletons (see Figure 6(b)). Some tools [21, 30, 45] use blockwise histogram based features (e.g., scale-invariant feature transform [39]) and image processing methods (e.g., edge detection [13], OCR [51]) to identify UI elements from images. Other tools (e.g., Export Kit [3]) use the metadata of UI elements in complex image formats exported by design tools (e.g., the PSD file by Photoshop) to assist the transition from UI design images to GUI implementations. However, these tools are rigid because they are built on limited, hand-designed rules for visual feature extraction and image-to-GUI transformation.

Different from these rule-based tools, our work is inspired by recent successes of deep learning (DL) models in image classification [35, 57], captioning [19, 36, 58], and machine translation [15, 60]. DL models are entirely data-driven, and do not require manual rule development. The most related work are image captioning techniques, but they take as input natural scene or digital document images and generate a sequence of words in natural languages [36, 58] or markup ones (e.g., latex expressions [19]). In software engineering community, some DL based methods have been proposed to generate code given input-output examples [8, 27], partially completed code [59] or feature descriptions [26, 42], or generate code comments [6] or code-change commit messages [50]. But our work is the first deep learning based technique, trained with real-world App UI data, to convert UI requirements (in the form of UI images) into a hierarchy of GUI components.

Before deploying deep learning models, a high-quality dataset is required for the models to learn important data features for a given task. Computer vision and NLP communities usually adopt crowdsourcing approach to develop such datasets for model development and benchmark [18, 38]. A major contribution of our work is to develop an automated program analysis technique to automatically collect large-scale UI data from real-word apps for implementing and evaluating the deep learning based UI-image-to-GUI-skeleton generation. This makes our work significantly different from existing work [10] which has similar goal but are developed based on artificial UI data generated by rules.

Our approach is generative based on the UI design and GUI implementation knowledge learned from existing apps. An alternative way to reuse such knowledge in existing apps is search-based methods. To use IR based code search methods [14, 33, 40], an accurate, concise description of UI elements and spatial layout in an image is required but hard to achieve. It would be desirable to search GUI implementation by UI image directly but a challenge in image-GUI search is how to match two heterogeneous data. The only work having this flavor is Reiss's work [48]. But this work internally uses templates to transform an input UI sketch into a structured query for matching GUI code. These sketch-to-query templates limit the generality of the approach. Furthermore, the fundamental difference between our generative approach and search-based approaches is that our approach can generate GUI skeletons that are not present in a code base, while the searching method can only return the information available in the code base.

8 CONCLUSION

This paper presents a generative tool for UI-image-to-GUI-skeleton generation. Our tool consists of two integral parts: a deep learning architecture and an automated UI data collection method. Our tool possesses several distinctive advantages: 1) it integrates feature extraction, spatial encoding and GUI skeleton generation into an end-to-end trainable framework. 2) it learns to abstract informative UI features directly from image data, requiring neither handcraft features nor image preprocessing. 3) it learns to correlate UI features and GUI components and compositions directly from UI images and GUI skeletons, requiring no detailed annotations of such correlations. 4) it is trained with the first large-scale UI-image-GUI-skeleton dataset of real-world Android applications. These advantages gives our tool unprecedented speed, reliability, accuracy and generality in over 12000 UI-image-to-GUI-skeleton generation tasks. In the future, we will further test our tool with different UI resolutions and orientations. We will also extend our neural network components to web design and implementation.

ACKNOWLEDGMENTS

We appreciate all participants for the human study, and we especially thank Yuekang Li for the constructive discussion in Android UI development. We also thank the GPU support from Nvidia AI center, Singapore to accelerate the experiments.

REFERENCES

[1] 2017. Adobe Photoshop. http://www.adobe.com/Photoshop/. (2017).
[2] 2017. Android Studio. https://developer.android.com/studio/index.html. (2017).
[3] 2017. Convert a psd to android xml ui and java. http://exportkit.com/learn/how-to/export-your-psd/convert-a-psd-to-android-xml-ui-and-java. (2017).
[4] 2017. Sketch:Professional digital design for Mac. (2017). https://www.sketchapp.com/
[5] 2017. Xcode. https://developer.apple.com/xcode/. (2017).
[6] Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A convolutional attention network for extreme summarization of source code. In *International Conference on Machine Learning*. 2091–2100.
[7] Apktool. 2017. Apktool. (2017). Retrieved 2017-5-18 from https://ibotpeaches.github.io/Apktool/
[8] Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2016. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989* (2016).
[9] Alexandre Bartel, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2012. Dexpler: Converting Android Dalvik Bytecode to Jimple for Static Analysis with Soot.
[10] Tony Beltramelli. 2017. pix2code: Generating Code from a Graphical User Interface Screenshot. *arXiv preprint arXiv:1705.07962* (2017).
[11] Léon Bottou. 2012. Stochastic gradient descent tricks. In *Neural networks: Tricks of the trade*. Springer, 421–436.
[12] Budget. 2017. Budget. (2017). Retrieved 2017-5-18 from https://github.com/notriddle/budget-envelopes
[13] John Canny. 1986. A computational approach to edge detection. *IEEE Transactions on pattern analysis and machine intelligence* 6 (1986), 679–698.
[14] Wing-Kwan Chan, Hong Cheng, and David Lo. 2012. Searching connected API subgraph via text phrases. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 10.
[15] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078* (2014).
[16] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated Test Input Generation for Android: Are We There Yet? (E). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*. 429–440.
[17] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. 2011. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*.
[18] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*. IEEE, 248–255.
[19] Yuntian Deng, Anssi Kanervisto, Jeffrey Ling, and Alexander M Rush. 2017. Image-to-Markup Generation with Coarse-to-Fine Attention. In *International Conference on Machine Learning*. 980–989.
[20] Soot Developers. 2017. Soot. (2017). Retrieved 2017-5-18 from https://github.com/Sable/soot
[21] 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.
[22] Michael P Fay and Michael A Proschan. 2010. Wilcoxon-Mann-Whitney or t-test? On assumptions for hypothesis tests and multiple interpretations of decision rules. *Statistics surveys* 4 (2010), 1.
[23] Daniel J Felleman and David C Van Essen. 1991. Distributed hierarchical processing in the primate cerebral cortex. *Cerebral cortex* 1, 1 (1991), 1–47.
[24] Google. 2017. Android UI Automator. (2017). Retrieved 2017-5-18 from http://developer.android.com/tools/help/uiautomator/index.html
[25] Google. 2017. Monkey. (2017). Retrieved 2017-2-18 from http://developer.android.com/tools/help/monkey.html
[26] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 631–642.
[27] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2017. DeepAM: Migrate APIs with Multi-modal Sequence to Sequence Learning. *arXiv preprint arXiv:1704.07734* (2017).
[28] Xiaocong He. 2017. Python wrapper of Android UIAutomator test tool. (2017). Retrieved 2017-5-18 from https://github.com/xiaocong/uiautomator
[29] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.
[30] Ruozi Huang, Yonghao Long, and Xiangping Chen. 2016. Automaticly Generating Web Page From A Mockup.. In *SEKE*. 589–594.
[31] David H Hubel and Torsten N Wiesel. 1962. Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. *The Journal of physiology* 160, 1 (1962), 106–154.
[32] David H Hubel and Torsten N Wiesel. 1968. Receptive fields and functional architecture of monkey striate cortex. *The Journal of physiology* 195, 1 (1968), 215–243.
[33] Iman Keivanloo, Juergen Rilling, and Ying Zou. 2014. Spotting working code examples. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 664–675.
[34] Philipp Koehn. 2004. Pharaoh: a beam search decoder for phrase-based statistical machine translation models. *Machine translation: From real users to research* (2004), 115–124.
[35] 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.
[36] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *Nature* 521, 7553 (2015), 436–444.
[37] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.
[38] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. 2014. Microsoft coco: Common objects in context. In *European conference on computer vision*. Springer, 740–755.
[39] David G Lowe. 1999. Object recognition from local scale-invariant features. In *Computer vision, 1999. The proceedings of the seventh IEEE international conference on*, Vol. 2. Ieee, 1150–1157.
[40] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. 2011. Portfolio: finding relevant functions and their usage. In *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 111–120.
[41] Guozhu Meng, Yinxing Xue, Jing Kai Siow, Ting Su, Annamalai Narayanan, and Yang Liu. 2017. AndroVault: Constructing Knowledge Graph from Millions of Android Apps for Automated Analysis. *CoRR* abs/1711.07451 (2017). arXiv:1711.07451 http://arxiv.org/abs/1711.07451
[42] Lili Mou, Rui Men, Ge Li, Lu Zhang, and Zhi Jin. 2015. On end-to-end program generation from user intention by deep neural networks. *arXiv preprint arXiv:1510.07211* (2015).
[43] Naila Murray and Florent Perronnin. 2014. Generalized max pooling. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2473–2480.
[44] Vinod Nair and Geoffrey E Hinton. 2010. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*. 807–814.
[45] 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.
[46] Greg Nudelman. 2013. *Android design patterns: interaction design solutions for developers*. John Wiley & Sons.
[47] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*. Association for Computational Linguistics, 311–318.
[48] Steven P Reiss. 2014. Seeking the user interface. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 103–114.
[49] Patrice Y Simard, David Steinkraus, John C Platt, et al. 2003. Best practices for convolutional neural networks applied to visual document analysis.. In *ICDAR*, Vol. 3. 958–962.
[50] Ameer Armaly Siyuan Jiang and Collin McMillan. 2017. Automatically Generating Commit Messages from Diffs using Neural Machine Translation. In *Automated Software Engineering (ASE), 2017 32th IEEE/ACM International Conference on*. IEEE.
[51] Ray Smith. 2007. An overview of the Tesseract OCR engine. In *Document Analysis and Recognition, 2007. ICDAR 2007. Ninth International Conference on*, Vol. 2. IEEE, 629–633.
[52] Ting Su. 2016. FSMdroid: Guided GUI testing of android apps. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016 - Companion Volume*. 689–691.
[53] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, Stochastic Model-based GUI Testing of Android Apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, New York, NY, USA, 245–256.
[54] Ting Su, Ke Wu, Weikai Miao, Geguang Pu, Jifeng He, Yuting Chen, and Zhendong Su. 2017. A Survey on Data-Flow Testing. *ACM Comput. Surv.* 50, 1, Article 5 (March 2017), 35 pages.

[55] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*. 3104–3112.

[56] Seyyed Ehsan Salamati Taba, Iman Keivanloo, Ying Zou, Joanna Ng, and Tinny Ng. 2014. An exploratory study on the relation between user interface complexity and the perceived quality. In *International Conference on Web Engineering*. Springer, 370–379.

[57] Yaniv Taigman, Ming Yang, Marc'Aurelio Ranzato, and Lior Wolf. 2014. Deepface: Closing the gap to human-level performance in face verification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 1701–1708.

[58] 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.

[59] Martin White, Christopher Vendome, Mario Linares-Vásquez, and Denys Poshyvanyk. 2015. Toward deep learning software repositories. In *Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on*. IEEE, 334–345.

[60] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. 2016. Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. *arXiv preprint arXiv:1609.08144* (2016).

[61] Matthew D Zeiler and Rob Fergus. 2014. Visualizing and understanding convolutional networks. In *European conference on computer vision*. Springer, 818–833.

[62] Hong Zhu, Patrick A. V. Hall, and John H. R. May. 1997. Software Unit Test Coverage and Adequacy. *ACM Comput. Surv.* 29, 4 (Dec. 1997), 366–427.