GENERATIVE MODELS FOR PREDICTIVE UI DESIGN TOOLS

BY

JASON SITU

THESIS

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science in the Graduate College of the University of Illinois at Urbana-Champaign, 2019

Urbana, Illinois

Adviser:

Assistant Professor Ranjitha Kumar
ABSTRACT

User interface (UI) design is a central part of the mobile app creation process, which involves specifying the elements that should be placed on a screen, and how they should be arranged and styled. This paper introduces a generative model approach to predictive design for mobile UI layouts. Given a partial UI design, the model predicts the next UI element that should be added to the layout. Moreover, the model can be used queried multiple times in succession to autocomplete an entire UI screen. To power this design interaction, we present two types of models: generative adversarial networks (GANs) [7] and variational auto-encoders (VAEs) [15]. We train the GAN and VAE models over 1949 mobile UIs that represent a variety of screen types (e.g. Login, Onboarding), and compare both models along standard and design-based metrics, identifying key tradeoffs. Finally, we present a mobile UI mockup tool that leverages the GAN-based model to support a predictive design workflow.
To my parents, for their love and support.
ACKNOWLEDGEMENTS

First, I would like to express my very great appreciation to my advisor, Dr. Ranjitha Kumar, for her valuable support, patience guidance, and enthusiastic encouragement throughout my graduate student journey. Also, thank you Dr. Alexander Schwing for his insight, advice, and support in this research. Without their support, this research would be impossible.

Additionally, I would like to thank you Thomas Liu, Mark Craft, Deniz Arsan, Oliver Melvin, and Yuan Ting Hu for their help in this work. Finally, I wish to thank you my parents for their forever love and support.
TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION ............................................................................................................. 1

CHAPTER 2: RELATED WORK ........................................................................................................... 3
  2.1 Example Based Design Tools ................................................................................................. 3
  2.2 Data-Driven Design ................................................................................................................. 3

CHAPTER 3: DATA PREPARATION ................................................................................................. 5
  3.1 Labeling Screen Category .................................................................................................... 5
  3.2 Mimicking the UI Creation Process ..................................................................................... 7
  3.3 Data Representation ............................................................................................................. 9

CHAPTER 4: GENERATIVE MODELS FOR PREDICTIVE DESIGN .................................................. 10
  4.1 Pixel Generative Adversarial Networks (PIXEL-GAN) ....................................................... 11
  4.2 Box Variational Autoencoder (BOX-VAE) ........................................................................... 15
  4.3 Evaluation Metrics .............................................................................................................. 17
  4.4 Results .................................................................................................................................. 20

CHAPTER 5: PREDICTIVE MOBILE MOCKUP TOOL .................................................................... 26
  5.1 Integration with Generative Model ....................................................................................... 26
  5.2 Capabilities of the Mockup Tool ......................................................................................... 28

CHAPTER 6: LIMITATIONS AND FUTURE WORK ......................................................................... 29

REFERENCES ................................................................................................................................. 31
CHAPTER 1: INTRODUCTION

User interface (UI) design is a central part of the mobile app creation process. Designers must specify the elements that should be placed on a screen, and how they should be arranged and styled. Designers often use mockup tools like Adobe XD and Balsamiq to develop UI layout specifications; however, these tools can be challenging to use --- especially for novices --- since they either offer blank screens or cookie-cutter templates as starting points.

Recent research systems have presented computational models that scaffold UI layout generation. DesignScape leverages energy-based models to refine existing graphic design layouts, but require that a design already exist [16]. Li et al. introduced a technique for synthesizing novel 2D layouts based on a generative adversarial network approach; while the designs generated from the GAN reflect the patterns expressed in the training data, the model does not provide any control over the resultant design based [21].

This paper introduces a generative model approach to predictive design for mobile UI layouts. Given a partial UI design, the model predicts the next UI element to add to the layout, specifying the element's semantic type (e.g., image, text button), position, and dimensions. The model can be used queried once to predict the next step, or multiple times in succession to autocomplete an entire UI screen.

To power this design interaction, we present two types of models: generative adversarial networks (GANs) [7] and variational auto-encoders (VAEs) [15]. Within a GAN framework, we formulate next-step UI component prediction as an image-to-image translation task. We condition on an input image (partial UI) and generate another image (the next component). In contrast, we formulate VAEs in a discrete data space, representing layouts as collections of UI element bounding boxes. More specifically, we develop a VAE framework that predicts the next component's bounding box conditioned on the last component's bounding box.

We train the GAN and VAE models over 1949 mobile UIs that represent a variety of screen types (e.g. Login, Onboarding), and compare both models along standard metrics such as interaction over union (IOU) and accuracy, as well as, design-based metrics. The GAN-based model performed better under the IOU metric compared to the VAE; however, the VAE model
Figure 1.1: This paper introduces a mobile UI mockup tool that supports a predictive workflow. Given a partial design, an underlying generative model predicts the next UI element that should be placed on the screen. By querying this model multiple times in succession, a designer can auto-complete the entire UI.

... achieved higher accuracy. Finally, we present a mobile UI mockup tool that leverages the GAN-based model to support a predictive design workflow (Figure 1.1).
CHAPTER 2: RELATED WORK

2.1 EXAMPLE BASED DESIGN TOOLS

Our work builds upon previous research on design assistance. Assistance is often provided via examples. Previous works explore how examples can be better presented [16, 20], or how existing examples can be transferred for the target content [17]. Rewire [35] infers a vector representation of screenshots, enabling designers to edit their own designs based on selected examples.

Another line of work focuses on converting screenshots to code, bridging the gap between designers and front-end programmers [4, 27]. In comparison, our systems aid designers in producing UI design prototypes through step-by-step guidance. Instead of strictly following examples, we produce predictions based on the current progress of the design and update predictions as users interact with the system.

2.2 DATA-DRIVEN DESIGN

Method-wise, our system generates predictions one element at a time. This allows us to suggest new elements for the UI in an interactive manner. This idea is also used to infer a drawing program stroke by stroke [5] and suggesting new shapes to add to a 3D scene [34]. Mobile UIs are highly structured, with elements placed inside a view hierarchy. To account for this structure, we generate semantic annotations [17] for UI elements and represent the screen as a multi-channel image. From this structured data, we are able to learn design patterns from data [36].

In graphic design, automating layout is a classic problem [10]. Using heuristic visual cues and design principles to optimize single-page layouts have shown to be effective [29] and extend well to an interactive tool [30]. There has recently been success in extracting semantic structure from documents [39], as well as being able to extend layout generation to generate mobile app UIs [21]. This resembles our approach in representing the data in different ways as they represent layouts as pixel-based images or bounding boxes respectively.

Similar to layout generation, interior scene synthesis is another problem closely related to automating layout. Early approaches in scene synthesis focused on optimization and hand-crafted
design principles [24]. Interior scene synthesis has explored different representations such as 3D arrangements [6] and orthographic top-down view [38].
CHAPTER 3: DATA PREPARATION

We used the semantically annotated UIs from Liu et al. based on the 72k unique UIs from the RICO dataset [43]. Along the side, Android view-hierarchy is provided for each UI that represents structural information like component's class, bounding box position, and etc. However, both research contributions lack an indication about the design purpose for the UI screens. For each screen that has a specific UX purpose to the user, we assigned a label to that screen.

To validate our results, we used Welcome, On-boarding, Sign-Up, and Logins screens that appear at the beginning of mobile apps. To ensure we labeled all of the 4 screen categories in our dataset, we leveraged the semantics present in our dataset; we could restrict our search in the dataset by searching for UI screens with component types and text that often appear in these 4 screen categories.

After this search validation, we ended up finding 726 Welcome screens, 447 Logins screens, 284 Sign-up screens, and 492 On-boarding screens. We also performed an 80-20 split with random shuffle of this particular dataset that we will be using to train and test our models in the later section. These data separated into 580, 357, 227, and 393 screens in the train set; 146, 90, 57, and 99 screens in the test, in a total of 1557 training screens and 392 testing screens.

3.1 LABELING SCREEN CATEGORY

Liu et al. presented an icon labeling webapp to assign a label for an icon classification task. We borrow their approach and extend it to build a similar webapp and the assign screen categories we mention above. Our webapp consists of two blocks: front-end and back-end.

The front-end implementation uses a popular framework, ReactJS, and Material UI from Google to reduce some workload needed to configure user interface styling. In the top of the webapp, there is a status section showing the number of UIs is assigned in each category. The database status allows our researchers to understand the overall progress of the labeling task.

Below the database status, there are two input fields which allow users to enter a unique or a unique UI number. This allows the user quickly jumping to a particular app. This function helps dividing labeling work that user can start at a specific location. A screen category status bar is below the two input fields to show what screen category is labeled in the current app. We
Figure 3.1: A screen labeling webapp that we build to assign label for each unique UI in order to gather our training data.

only want to assign one screen category to a UI per app. To be specific, this means only one UI will be labeled as Login in one app, even multiple screens can be login. We just want to learn the design from each app. If we see a screen category is highlighted in the status, we know this particular app will not going assign a new screen for this highlighted screen category. This can help to search for more screen of a particular screen category.

For each UI in an app, we will show both the original UI (left) and its corresponding semantified screen (right). Below each UI, we have a list of chip-based buttons for each of the screen category. When a screen is not assigned to a category, all buttons are green except those screen categories is assigned in the current app. We will gray out a button to indicate a particular screen category is no longer needed to a label. If the user clicks on a green button, this button
will immediately change from green to orange color, which is to indicate the UI is assigned to this screen category. Consequentially, all other buttons except the one user click on will be grayed out and the final database status and screen category status will also update.

When the user finishes labeling an app or want to next app, there two long vertical buttons at the two side. User can click one of the vertical buttons to navigate back and forth. A screenshot of our screen labeling webapp is shown in Figure 3.1.

We have another web page will show all UIs of the labeled screen from each category. This page has a similar layout like the main labeling page, except there is only screen category selection at the top. Buttons below each UI in this page will be clickable to us to clean up miss labeled screen. One user on a button to fix the mislabeled screen, this screen will disappear in the current to reflect it is updated in our database.

To support these interactions, we use NodeJS server to communicate between front-end and a MongoDB database. We also use NodeJS to host the front-end webpage. Our backend server mainly contains blocks of RESTful API functions to enable our designed communication. We have extra functions to handle other requests like navigate to next app when user click on the vertical button.

3.2 MIMICKING THE UI CREATION PROCESS

In order to understand the process of how designers create UI, we performed 10 video-based user studies by watching videos of designers creating UIs. Two researchers from our team independently watched 5 videos to observe patterns sharing across each video and came up with an initial list of patterns. Using a consensus-driven approach, they met and discussed the difference until they reached an agreement.

Following this process, our final heuristics of mimicking UI creation are under the following rules:
Using heuristic rules, we recomposed the steps of UI creation process and display in semantic representation.

1. Depth-first approach - Any components nested in a container will being designed before moving to another container at the same hierarchical level
2. Left to Right - Components are placing in right-handed writing order
3. Top to Bottom - Designers usually start placing components at top of the screen and end at the bottom

Order of these rules matter, precedence applies from 1st rule to 3rd rule.

In an iterative fashion, we remove components in reverse order as designers would place them to create our synthetic sequence of UI screens. Using structural information in the view hierarchy, we removed components one by one starting from bottom to top, right to left, to inside out order. The first removed component represents the last step of creating a UI, while the last removed component represents the first step of an UI. Thus, the reverse of this removal process would mimic the UI creation process. For both of our representations, using each of the view hierarchies, we created a sequence of screens for each step in the creation process. An example UI with semantic annotation screen is shown in Figure 3.2.

By following this process of UI creation, we can create pairs of data that illustrate the relationship between a partial UI and its next-step component. For partial UIs after the first step of creation, we make a copy and remove all components except the last added component to represent the next component of previous partial UI.

Throughout and irrespective of the particular representation, we use $x$ to denote the current screen design, i.e., the current UI, and we refer to the next component to be added to the partial UI as $y$. 
3.3 DATA REPRESENTATION

To train models using the view hierarchies, we assess two different representations for the screens: (1) a pixel representation and (2) a bounding box representation. For both representations, each component on the UI screen is assigned one out of 25 possible categories based on the semantic component types from Liu et al. [37].

3.3.1 PIXEL REPRESENTATION

Pixel representations have recently been used with great successes for tasks involving generative models [12, 40, 40]. We evaluated several different ways to represent our data in pixel space such as representing each component type on its own depth layer, or representing each level of the view hierarchy on its own depth layer. We found that representing each component with semantic masks worked the best. This finding is consistent with similar tasks such as document layout generation [21] or semantic structure extraction from documents [39], where semantic mask representations were found to work well too.

In the end, we used the semantified UIs from Liu et al. [21]. Each UI is originally represented via a $128 \times 72 \times 3$ image, but we padded the UIs to a dimension of $128 \times 128 \times 3$ because our deep nets operated on square images.

3.3.2 BOUNDING BOX REPRESENTATION

Bounding box representations have worked well for tasks like document layout generation [21]. In this representation, each component on the screen is represented with its bounding box and a one-hot encoding vector $c \in R^C$ where $C$ is the total number of class types. There are a total of 25 semantic component types and therefore $C$ is 25. The bounding box of our components are represented with the top-left and the bottom-right coordinates of the bounding box $u_1, v_1, u_2, v_2$. This representation is able to encode the bare minimum information for each of the screens.
CHAPTER 4: GENERATIVE MODELS FOR PREDICTIVE DESIGN

Generative modeling refers to techniques which capture the underlying yet unknown distribution of samples subsumed in a given dataset. Many techniques have been proposed from classical algorithms like k-means and Gaussian mixture models to more complex latent variable techniques like hidden Markov models, latent Dirichlet allocation and restricted Boltzmann machines. In common to all those methods is direct construction of a probability distribution defined in the data domain, often also referred to as the output space domain. Sparsity challenges those methods since the domain is often high-dimensional, e.g., the space of all possible images, and the size of the dataset is small compared to the dimensionality of the domain.

To address this concern, two methods, Generative Adversarial Networks (GANs) [7] and Variational Auto-Encoders (VAEs) [15], have recently been proposed which rely on the “manifold assumption”: that the data within a dataset lies on a low-dimensional manifold within the high-dimensional domain.

Since both techniques were introduced, many research advances for Generative Adversarial Networks (GANs) and Variational auto-encoders have significantly improved the ability to model high-dimensional data, e.g., realistically looking images which resemble real images subsumed in a given dataset. Some of those advances are theoretically motivated while others are empirically verified [1, 2, 3, 8, 9, 13, 22, 25, 26, 28, 31].

Although both GANs and VAEs accomplish the same task, there are many trade-offs between both models. In spirit similar to a non-cooperative game, GANs pit a generator deep net, which transforms a sample drawn from a simple distribution (e.g., a Gaussian) to the data domain, against a discriminator deep net, which is tasked to differentiate real samples from artificial ones obtained from the generator. The discriminator is tasked to accurately differentiate while it's the generators job to make this differentiation as hard as possible. Note, for GANs the output of the generator is used as an input for the discriminator. This makes GANs particularly suitable if the desired output is continuous as end-to-end backpropagation is possible.

In contrast, VAEs follow the auto-encoder model, i.e., data is encoded into a latent space via a deep net and subsequently reconstructed as accurately as possible via the decoder deep net. Since the data is used as input and output only, this technique is particularly suitable for data in discrete spaces.
In addition to this structural difference, other aspects are important to consider as well. For instance, due to a saddle-point optimization, GANs are often hard to optimize and training of a GAN may take much longer time than training of a VAE.

The original techniques and recent advances are particularly successful for modeling of images. A classical example learns generators or decoders for hand-written digits given the MINST dataset [19]. Upon training, generator or encoder are able to produce artificial images of hand-written digits that look just like those written by humans.

In spirit similar to those examples are the methods analyzed in this paper. We use a modified GAN or VAE framework and extend its architecture to support next step UI component prediction. More specifically, we will ask the GAN or VAE to learn the underlying relationship of sequences of the UI creation process that we created above.

Our next-step component prediction can be thought of as an image-to-image translation task. We condition on an input image (partial UI) and generate another image (the next component). To learn this image-to-image mapping, we use a Pixel GAN which learns a conditional generative model given as a condition the current screen. We found the pixel representation to work well.

In contrast to GANs where layouts are represented as colored images, we formulate VAEs in the discrete data space. Specifically, the developed Box VAE predicts the next component position conditioned on the last component position. For the developed Box VAE we found the aforementioned bounding box representation to work well.

We subsequently describe both GAN and VAE methods and their benefits and disadvantages in greater detail.

4.1 PIXEL GENERATIVE ADVERSARIAL NETWORKS (PIXEL-GAN)

As mentioned before, GANs optimize a saddle-point objective which pits a parametric generator G against a parametric discriminator D. Both are typically formulated using deep nets and we specify the employed architectures below. The classical minimax objective for GANs can be formulated as follows:

$$\min_G \max_D \mathcal{L}_{GAN}(D, G) := E_y[\log(D(y))] + E_x[\log(1 - D(G(x)))]$$
Classical GANs adopt the cross-entropy loss function. Following recent works in image-to-image translation [41], we replace this loss with a least squares loss which was shown to stabilize the training process [23]. We use the $a - b$ coding scheme [23] for the discriminator, where $a$ and $b$ are hyper-parameters for the fake data and real data, respectively. The program optimizes the following objectives:

$$\max_D \mathcal{L}_{\text{LSGAN}}(D) := \frac{1}{2} \mathbb{E}_y[(D(y) - b)^2] + \frac{1}{2} \mathbb{E}_x[(D(G(x)) - a)^2]$$ (4.2)

$$\min_G \mathcal{L}_{\text{LSGAN}}(G) := \frac{1}{2} \mathbb{E}_y[(D(G(x)) - c)^2]$$ (4.3)

where $c$ denotes the value that $G$ wants $D$ to believe for the fake data. Following [23], we choose $b$ as the all-ones vector $1$, to state that it is real data. Conversely, we choose $a$ as the all-zeros vector to indicate fake data. Finally, $c$ is the all-ones vector, encouraging $G$ to fool $D$. Thus, our final least squares loss is as follows:

$$\max_D \mathcal{L}_{\text{LSGAN}}(D) := \frac{1}{2} \mathbb{E}_y[(D(y) - 1)^2] + \frac{1}{2} \mathbb{E}_x[(D(G(x)))^2]$$ (4.4)

$$\min_G \mathcal{L}_{\text{LSGAN}}(G) := \frac{1}{2} \mathbb{E}_y[(D(G(x)) - 1)^2]$$ (4.5)

In addition to the least squares GAN loss $\mathcal{L}_{\text{LSGAN}}(G, D)$, we found an additional $\mathcal{L}_{L1}(G)$ loss to be helpful when training the generator. All in all, when training the generator, we therefore address the following objective:

$$\min_G \mathcal{L}_{\text{LSGAN}}(G) + \lambda \mathcal{L}_{\text{L1}}(G)$$ (4.6)

where $\lambda \mathcal{L}_{\text{L1}}(G) = \mathbb{E}_{x,y}[\|y - G(x)\|_1]$. 


For our adversarial loss, we use the mean squared error (MSE) loss combined with an L1 distance loss. We used the L1 loss instead of the L2 loss because the L2 loss is known to produce blurry results [18].

Note that this loss suffers from imbalanced training data. This is exacerbated in our case if components occupy a small fraction of the screen area.

For example, if a generator output doesn't provide any prediction and the ground truth component occupies 2% of the screen size at the input resolution, the L1 loss is very close to zero. As a result, the model without adjustment tends to not predict any UI components.

To rectify this concern, we use a weighted L1 loss which emphasizes wrong component predictions by a factor of $10 \times$ and reduces errors due to wrong background prediction by a factor of $0.1 \times$. In summary, the employed weighted L1 loss function is:

$$\mathcal{L}_{L1}(G) = \frac{\alpha}{|S^+|} \sum_{i \in S^+} |y_i - G_i(x)| + \frac{\beta}{|S|} \sum_{j \in S} |y_j - G_j(x)|$$

(4.7)

where $i$ and $j$ denote the $i^{th}$ and $j^{th}$ pixel on the screen respectively, $S^+ = \{i | y_i \neq (255, 255, 255)\}$, $S = \{i | y_i = (255, 255, 255)\}$, $\alpha$ is the weight factor for the ground truth component location (non-white pixels), while $\beta$ is the weight factor for the ground truth background pixels (white pixels). Note again that $y$ refers to the predicted next screen component while $x$ denotes the current screen design.

In our case, $\alpha = 10$ and $\beta = 0.1$. Due to the reweighting, the predictions in both training and test set contain significantly more UI components compared to using the classical L1 loss.

In the following we discuss architectures for the generator $G$ and the discriminator $D$.

4.1.1 MODEL ARCHITECTURE

Our generator and discriminator architectures are derived from Pix2Pix [12].

The generator $G(x)$ operates on the current screen $x$ and predicts a possible UI design continuation. Specifically, our generator is based on an auto-encoder style “U-Net” architecture. Both the encoder and decoder within the generator use a stack of module-based layers composed of convolution, batch-normalization, and a Leaky Rectified Linear Unit (LeakyReLU) activation function. The encoder consists of 7 modules with a filter size of $4 \times 4$ in each of its
Figure 4.1: The generator and discriminator model architecture in details.

Convolutional layers. Also, normalization is only applied after the first module. The decoder is arranged in the reverse order of the encoder using six transpose convolution layers and ReLU activation functions instead of LeakyReLUs. It is then followed by an upsampling layer, a convolutional layer, and a hyperbolic tangent function to produce a pixel-wise prediction at a resolution of $128 \times 128 \times 3$.

We modified the “U-Net” architecture to reduce skip connections which we found to result in the best trade-off between training time and performance. In particular, we only use one skip connection which links the 5th module of the encoder with the 2nd module of the decoder.
The discriminator $D(x)$ computes the likelihood that its input $y$ is a valid design continuation originating from the given dataset as opposed to being artificially generated by the generator. To this end we use a convolutional "PatchGAN" classifier consisting of 4 module layers which share the same form as the encoder in our generator. Those are followed by a padding and a convolution layer to reach a patch size of $8 \times 8 \times 1$.

PatchGAN differentiates two images at the scale of image patches. Different patch sizes were assessed [12] and the authors found $16 \times 16$ to successfully guide the generator to produce a sufficiently sharp image at a resolution of $256 \times 256$. Therefore, we decrease the patch size proportionally to the input size that we use $128 \times 128$. The model architecture of this Pixel GAN is shown in Figure 4.1, including the output size of each layer.

### 4.2 BOX VARIATIONAL AUTOENCODER (BOX-VAE)

For a VAE model we found the bounding box representation to work better in predicting a component position. Recall, the bounding box representation refers to four coordinates that represents a rectangle and a one-hot encoding is used to represent each of the 25 component classes.

A conditional variational auto-encoder [33] maximizes the data log-likelihood \( \log p_y(y|x) \), i.e., the likelihood of predicting the correct design continuation $y$ given current screen design $x$. Note again that we use the bounding box representation in the Box Variational Autoencoder, which differs from the pixel-based representation that is used in the aforementioned Pixel GAN. Specifically, the current screen $x$ with $n$ components is represented as a vector

$$x = [b_1, \ldots, b_n]^T \in \mathbb{R}^{(4m+25) \times n},$$

where $b_i = [u_i, v_i, u_2, v_2, c]$ specifies the location of the $i^{th}$ element on the screen, i.e., top left corner $(u_1, v_1)$ and bottom right corner $(u_2, v_2)$, as well as the one-hot encoded component type $c \in \{0,1\}^C$, where $C = 25$ is the total number of component classes. Note that we discretize the
screen into an $m \times m$ grid and each of $u_1$, $u_2$, $v_1$ and $v_2$ is also a one-hot encoded $m$ dimensional location feature. If the coordinate falls in the $i^{th}$ grid, the $i^{th}$ component of the location feature is 1 and all the others are zero. We set $m$ to 32. The design continuation $y = b_{n+1}$ simply specifies the location and the component type information of the desired prediction. We truncate components for screens with more than $n$ boxes and zero-pad for screens with less than $n$ objects. In our case $n = 2$.

**Figure 4.2:** Two of diagrams demonstrate high level difference between Pixel-GAN and Box-VAE. Pixel-GAN uses pixel-based input, while Box-VAE uses bounding box representation.

Instead of directly maximizing the data log-likelihood, a latent space (manifold) with samples referred to using $z$ is hypothesized, i.e., we obtain the log-likelihood
log \( p_\theta(y | x) \) by marginalizing over the latent space. Here, summation over \( z \) denotes integration over the entire latent space which may be discrete or continuous. In addition, a parametric encoder \( q_\phi(z | y, x) \) is introduced to lower-bound the log-likelihood via

\[
\log p_\theta(y | x) = \log \sum_z p_\theta(y, z | x) \geq \sum_z q_\phi(z | y, x) \log \frac{p_\theta(y | z, x) p(z)}{q_\phi(z | y, x)}
\]

where Jensen's inequality is used to obtain the lower bound on the log-likelihood \( \log p_\theta(y | x) \). Decomposing the ratio yields the final VAE training objective:

\[
\max_{\phi, \theta} -D_{KL}(q_\phi(z | y, x) \| p(z)) + \frac{1}{N} \sum_{i=1}^{N} \log p_\theta(y | x, z_i)
\]

where \( D_{KL} \) is the KL-divergence, and \( N \) is the number of samples \( z_i \sim q_\phi(z | y, x) \) drawn from the encoder. We assume \( p(z) \) to be a Gaussian distribution with zero mean and unit variance. Both the encoder \( q_\phi(z | x, y) \) and the decoder \( p_\theta(y | x, z) \) are 2-layer (multi-layer) perceptrons with ReLU non-linearity and batch normalization [11]. We train with the Adam optimizer [14] with a learning rate of 0.001 for 200 epochs.

The high-level difference between Pixel-GAN and Box-VAE is shown in a diagram in Figure 4.2.

### 4.3 EVALUATION METRICS

We use accuracy and Intersection Over Union (IOU) to evaluate our results. In addition, we also introduce additional variants of these metrics. We introduce a variant which account for multiple next step possibilities as well as variants which account for how much work is needed for the designer to move the predicted component to the ground truth position. To this end we
propose to evaluate design prediction techniques using a variety of new metrics for accuracy and IOU: Lookahead, Centeredness, Closeness, and Centeredness with Closeness.

Figure 4.3. This figure shows how Intersection Over Union is computed between a ground truth element and a predicted element.

Figure 4.4. This figure shows how pixel-wise accuracy is computed.

4.3.1 CLASS

In addition to predicting for the bounding box of the next component, we also predict for the class type of the component. For each of the metrics listed below, we evaluate them with class-specific scores (where an incorrect class type will result in a score of 0) and class-agnostic scores.

4.3.2 IOU
IOU is an evaluation metric that reports the area of the overlap between the prediction and ground truth and dividing by the area of union (the area encompassed by both the prediction and ground truth), see Figure 4.3. Note that IOU is a harsh metric, particularly because it only gives credit for the area of overlap and heavily penalizes for any area that is not overlapping. In the case where the prediction is offset from the ground truth, even if there is minimal effort for the designer to correct it, the score may be very low.

4.3.3 PIXEL ACCURACY

Pixel accuracy is an alternative metric beyond IOU which is also often used in semantic segmentation tasks. This metric simply reports the fraction of pixels in the image which are correctly classified (Figure 4.4).

Pixel accuracy is less harsh than IOU but it may be misleading: in cases where the ground truth component is small, the metric is biased to mainly report how well background is identified rather than focusing on the components and classes of interest.

4.3.4 LOOKAHEAD

The aforementioned metrics only take into account the next possible component. However, there may be multiple next step possibilities that would not be accounted for. We propose to give credit for predicting a future component rather than only the next component. To do this, we can use the maximum score out of all of the next components rather than just the next one.

In addition to giving credit to future components, we can also give credit for predicting a component that would appear on a similar looking screen. To find screens that are similar, we use the same method as presented by Liu et al.: we train an autoencoder on the semantified screens. Using the latent dimension from the autoencoder, we use a ball tree data structure to query for the most similar screens. After querying for the 5 most similar screens, we report the best score among the future components of the 5 closest screens.

4.3.5 CENTEREDNESS
We created this metric to approximate the effort of a designer to re-size a prediction to fit the ground truth size. We move the prediction to the center of the ground truth location before computing the score. This is a useful variant because with vanilla IOU or accuracy, if the predicted component is the exact same size as the ground truth but just shifted a bit, there may be a large penalty. However, this variant ignores the location of the predicted and ground truth components.

4.3.6 Closeness

In addition to measuring how much effort is required to re-size a component, we also measure how much effort is required to move a component to its desired location. We compute the distance between the center coordinate of both the prediction and the ground truth. Then, we normalize it to be in the range of [0,1] by dividing it by the maximum possible distance on the screen. Finally, we subtract this value from 1 to report how close an element is from its ground truth location rather than how far away it is.

4.3.7 Closeness

By combining the Centeredness and Closeness metrics, we obtain one metric that represents how much effort a designer would have to spend to change the predicted component into the ground truth component. The simplest way to combine these two metrics is to multiply their values together.

We believe these four variants are more designer-centric and directly assess the necessary effort to correct the predictions of our system.

4.4 RESULTS

To train a Pixel-GAN model in a reasonable time, we downsampled the semantified UIs from 2560 x 1440 to 128 x 72 and padded to 128 x 128. Since the generator output is a square as well, post-processing of the pixel-based output is required to retrieve the original
Table 4.1: Mix and match all combination of evaluation metrics that we proposed to evaluate Box-VAE model. We take samples evaluated in class-agnostic, while the number in parentheses is class-specific.

<table>
<thead>
<tr>
<th>Accuracy</th>
<th>Closestness</th>
<th>Closedness</th>
<th>Centered</th>
<th>IOU</th>
</tr>
</thead>
<tbody>
<tr>
<td>97.15% (68.33%)</td>
<td>66.9% (68.9%)</td>
<td>66.9%</td>
<td>66.9%</td>
<td>66.9%</td>
</tr>
<tr>
<td>86.8% (69.6%)</td>
<td>66.9%</td>
<td>66.9%</td>
<td>66.9%</td>
<td>66.9%</td>
</tr>
<tr>
<td>71.16% (77.99%)</td>
<td>66.9%</td>
<td>66.9%</td>
<td>66.9%</td>
<td>66.9%</td>
</tr>
<tr>
<td>100% (71.97%)</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>70.0% (59.75%)</td>
<td>66.9%</td>
<td>66.9%</td>
<td>66.9%</td>
<td>66.9%</td>
</tr>
<tr>
<td>50.4% (69.6%)</td>
<td>66.9%</td>
<td>66.9%</td>
<td>66.9%</td>
<td>66.9%</td>
</tr>
<tr>
<td>97.15% (68.33%)</td>
<td>66.9%</td>
<td>66.9%</td>
<td>66.9%</td>
<td>66.9%</td>
</tr>
<tr>
<td>86.8% (69.6%)</td>
<td>66.9%</td>
<td>66.9%</td>
<td>66.9%</td>
<td>66.9%</td>
</tr>
<tr>
<td>71.16% (77.99%)</td>
<td>66.9%</td>
<td>66.9%</td>
<td>66.9%</td>
<td>66.9%</td>
</tr>
<tr>
<td>100% (71.97%)</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>70.0% (59.75%)</td>
<td>66.9%</td>
<td>66.9%</td>
<td>66.9%</td>
<td>66.9%</td>
</tr>
<tr>
<td>50.4% (69.6%)</td>
<td>66.9%</td>
<td>66.9%</td>
<td>66.9%</td>
<td>66.9%</td>
</tr>
</tbody>
</table>
Figure 4.5: Examples of 8 test screens obtained from 4 screen categories and two from each category. The middle column shows a partial UI and the ground truth next component along the two prediction outputs directly from Pixel-GAN and Box-VAE respectively, all semantified. In the third column, it is the partial UI of continuation from the middle column.
screen ratio. Since the pixel-based input is an RGB image, the final input size is in 128x128x3 which consists of three stacked layers representing each color channel.

We found that pre-training the model with the modified loss and fine-tuning with the traditional loss to improve the results even further. Although the modified loss results in more UI component predictions and increases the consistency of the output, the predictions tend to be much more blurry. By fine-tuning the model with the traditional loss, the model would not only produce components consistently, the components would also be more rectangular, which is intuitive, see Figure 4.5.

We compare the results of the various metrics explained above on our Pixel-GAN and Box-VAE in Table 4.1 and Table 4.2. To more accurately measure the performance of the Pixel-GAN, we use the same post-processing that we did when integrating the model into our mockup tool; we post-process the output to get a bounding box and the class by looking at the median non-white pixel in the prediction.

While our Pixel-GAN learns an almost deterministic mapping between input and output, we are able to sample from the latent dimension in our Box-VAE. Thus, for the Box-VAE, for each of the metrics, we report the best score out of $k$ samples, where $k \in \{1, 5, 10, 20, 100\}$. Since Box-VAE is capable of producing diverse results, it would be unfair to compare to Pixel-GAN on metrics other than best IOU at $k = 1$.

From Table 4.1: we observe the following two trends:

1. For the class-agnostic numbers the Pixel-GAN tends to outperform the Box-VAE on the IOU metrics while the opposite trend is observed for accuracy.
2. For the class-specific numbers the Pixel-GAN tends to outperform the Box-VAE on all metrics except for the lookahead variants.

Compared to the vanilla IOU and accuracy metrics, we see that using our four new variants, the scores for the Pixel-GAN and Box-VAE greatly increase. We see that since Centeredness only considers the area of the prediction and not the location, this score significantly increases the scores for both the Pixel-GAN and Box-VAE. Additionally, we see that Closeness generally has a large score which indicates that most of the predictions are near the ground truth location and would not require the designer to do that much work in moving the prediction. Finally, Lookahead also increases the scores implying that the models making more meaningful predictions than is captured by the vanilla metrics.
<table>
<thead>
<tr>
<th></th>
<th>GAN@1</th>
<th>VAE@1</th>
</tr>
</thead>
<tbody>
<tr>
<td>IOU</td>
<td>26.40% (14.70%)</td>
<td>21.18% (5.08%)</td>
</tr>
<tr>
<td>Lookahead IOU</td>
<td>49.17% (31.47%)</td>
<td>46.90% (14.99%)</td>
</tr>
<tr>
<td>Centered IOU</td>
<td>42.26% (20.00%)</td>
<td>41.58% (10.79%)</td>
</tr>
<tr>
<td>Closeness IOU</td>
<td>88.01% (33.61%)</td>
<td>90.36% (26.47%)</td>
</tr>
<tr>
<td>Centered IOU * Closeness IOU</td>
<td>39.46% (19.28%)</td>
<td>38.70% (10.00%)</td>
</tr>
<tr>
<td>Lookahead Centered IOU</td>
<td>78.13% (53.52%)</td>
<td>77.08% (58.89%)</td>
</tr>
<tr>
<td>Lookahead Closeness IOU</td>
<td>89.14% (64.14%)</td>
<td>85.74% (76.25%)</td>
</tr>
<tr>
<td>Lookahead Centered IOU * Closeness IOU</td>
<td>71.91% (49.79%)</td>
<td>70.79% (49.44%)</td>
</tr>
<tr>
<td>Accuracy</td>
<td>91.68% (33.79%)</td>
<td>93.69% (27.40%)</td>
</tr>
<tr>
<td>Lookahead Accuracy</td>
<td>97.43% (69.55%)</td>
<td>100% (97.86%)</td>
</tr>
<tr>
<td>Centered Accuracy</td>
<td>33.79% (34.14%)</td>
<td>27.40% (28.01%)</td>
</tr>
<tr>
<td>Closeness Accuracy</td>
<td>88.02% (33.61%)</td>
<td>90.36% (26.47%)</td>
</tr>
<tr>
<td>Centered Accuracy * Closeness Accuracy</td>
<td>83.99% (32.45%)</td>
<td>86.78% (25.40%)</td>
</tr>
<tr>
<td>Lookahead Centered Accuracy</td>
<td>95.09% (40.20%)</td>
<td>97.99% (94.86%)</td>
</tr>
<tr>
<td>Lookahead Closeness Accuracy</td>
<td>94.37% (60.89%)</td>
<td>97.31% (88.68%)</td>
</tr>
<tr>
<td>Lookahead Centered Accuracy * Closeness Accuracy</td>
<td>92.12% (36.14%)</td>
<td>95.38% (85.96%)</td>
</tr>
</tbody>
</table>

Table 4.2: Our proposed evaluation metrics and take one sample from Pixel-GAN and Box-VAE. This shows the tradeoff between to GAN-based and VAE-based model. The first number in each metric evaluated in class-agnostic, while the number in parenthesis computed in class-specific.

For the IOU, although Lookahead, Centeredness and Closeness each improve the scores, combined, they increase the scores further. The combination of these variants are the most
insightful to assess quality of the predictions towards designers; this combination measures the possibilities that designers may consider and how much effort they would need to put to correct the model's predictions.

Since Box-VAE can predict diverse components, the more samples we taken from the model the better score it can achieve. Results in Table 4.1 show that highest score recorded is from sampling 100 times from the example.

In the side-by-side comparison from Table 4.2, we observe that the GAN performs better than the VAE because Pixel-GAN is using global screen information rather than local information of the last component.
CHAPTER 5: PREDICTIVE MOBILE MOCKUP TOOL

To demonstrate the efficacy of our approach, we created a mockup tool to integrate and test our model Figure 5.1. The mockup tool that we created includes all of the basic features needed to create mockups of mobile UIs. In the mockup, we have a left-hand panel and canvas that users can interact with. On the left-hand panel, users can to choose between 12 different components and we also display the view hierarchy of the current screen.

The canvas is where all of the components can be placed and moved on. Once components are placed on the canvas, users are able to move, re-size, delete, or alter them (such as changing the text on text buttons). In addition to the basic features of the mockup tool, we allow for next step predictions using our generative models.

5.1 INTEGRATION WITH GENERATIVE MODEL

Although the Box-VAE has an advantage of being able to provide many different outputs by sampling from its latent dimension, because the Pixel-GAN did better in all metrics when sampled once, the final model that was integrated into the mockup tool was the Pixel-GAN. Internally the current state of the screen is stored as a JSON object. An advantage of storing the screen as a JSON object is that it has a tree-like structure and it was how the view hierarchies for the original screens were stored. Thus, it was simple to transform the JSON into the pixel representation or the box representation for the Pixel-GAN and the Box-VAE respectively. To be able to add a prediction to the screen, the frontend required the bounding box of the prediction as well as the class label.

To be able to use the Pixel-GAN with the mockup tool, we needed to pass as input the pixel representation of the current screen. Once we did this, we would get out a pixel representation of the prediction from the model. To be able to render this into our frontend, it was necessary to extract the bounding box as well as the class label from the prediction. The output of the Pixel-GAN although mostly rectangular in shape, would be fuzzy and post-processing was necessary to get the bounding box.

To get the bounding box, we use the maximum sum submatrix algorithm. To use this algorithm, we needed to weight white pixels by $-1$ and colored pixels by $0.1$. By having this
Figure 5.1: The mockup tool that displays a translucent suggested component on the canvas, and its component type and location outlined in red in the view hierarchy.

weighting, we try to have the least amount of white pixels as possible as well as to have the most amount of colored pixels as possible. To get the class label, we use L2 distance from the medium colored pixel with each pixel value of the class labels and use the one with the smallest distance.

In contrast to the Pixel-GAN, integration of the Box-VAE with the mockup tool was a much easier process. The Box-VAE takes as input the bounding box and a one-hot encoding of the class label of the last placed component which can be taken directly from the JSON. The model then outputs the bounding box and one-hot-encoding of the predicted class which could then be directly used by the frontend. Regardless of the model, we found that post-processing the predictions to have their bounding box be aligned with nearby components to both look better as
well as improve the effectiveness of the predictions. When components on the screen are aligned, the models are more likely to produce a prediction.

5.2 CAPABILITIES OF THE MOCKUP TOOL

    From our mockup tool, we provide two main capabilities with our model.

    We first provide a next step prediction whenever anything on the canvas changes. The model is conditioned on the current screen and will predict a next component that will appear as a translucent suggestion for that component Figure 5.1. The prediction is also shown outlined in red in the view hierarchy. When either the translucent component or the predicted component in the view hierarchy is clicked, it is placed on the screen. This general next step prediction is very helpful as it will be prompted on any action that would impact the canvas.

    Besides this general next step prediction, we also provide next step predictions for a specified component type. Whenever a user hovers over one of the component types on the left-hand panel, a prediction for that component type will appear in the same manner as the general next step prediction. This is achieved by having a model conditioned on both the current screen as well as the desired component type. This prediction gives more control to the user by allowing them to request a prediction of a certain type.
CHAPTER 6: LIMITATIONS AND FUTURE WORK

One of our limitations is that the dataset contains only the first set of mobile apps screen categories (e.g. Welcome, Login, Sign-Up, and On-boarding). Also, intensive human labor is required to validate the label even though the search-based approach is used to label the screen category. However, our focus is on presenting how to create novel interactions amongst UI design tools to support a predictive workflow. We want researchers/design tool creators to leverage our system architecture and integrates into their system. We can exploit all screen categories in RICO and Liu et al. to create comprehensive coverage to support the entire mobile app design workflow.

Another limitation in our dataset are the heuristic rules for the creation of train and test data. Designers might have other approaches to create UI in the design process. Designer might start at the middle instead of starting at the top. There can be more complex rule that our heuristic rules did not capture.

Currently, we are only able to predict the semantics for Text Buttons. For example, we can predict button with text Login, Sign-Up with Google, and etc. We also want to predict like what kind of text can be the text box. In the future, we can train models to support more semantic and stylistic predictions.

Similar to [12], our Pixel-GAN generates output in a one-to-one mapping way. In the future, we want to explore approaches that allow our model that produces output that has variability in terms of component position and component class. This reason is in our generator there is no random variable that sample from a particular distribution. With a random variable, it allows our model properly to learn the hidden distribution that underlie in our dataset.

Both Pixel-GAN and Box-VAE suffers to produce results with a high score in vanilla IOU. We want researchers to leverage our approach and explore other generative models that can increase testing performance.

One of the limitations in Box-VAE is that the input screen is discretized into a lower resolution. Similarly, Pixel-GAN also used a downsampled screen resolution in training. We would like to explore using a higher resolution screen in our training process.

We also want researchers to leverage our approach to explore other domains of predictive work besides just the next-step prediction. For example, we can have a system that looks for
potential missing components by predicting what component can add to make UI more appealing design.

The mockup design tool we built is lacking user feedback on how actual designers would use it. We want to perform user studies to answer questions like, how many times designers chose to accept the model prediction, or whether designers find the next step prediction useful in their workflow.
REFERENCES


