ULDGNN: A Fragmented UI Layer Detector Based on Graph Neural Networks

Jiazhi Li
Zhejiang University
Hangzhou, China
lijz@zju.edu.cn

Tingting Zhou
Alibaba Group
Hangzhou, China
miaojing@taobao.com

Yunnong Chen
Zhejiang University
Hangzhou, China
chen_y@zju.edu.cn

Yanfang Chang
Alibaba Group
Hangzhou, China
suchuan.cyf@alibabainc.com

Yankun Zhen
Alibaba Group
Hangzhou, China
zhenyankun.zyk@alibabainc.com

Lingyun Sun
Zhejiang University
Hangzhou, China
sunly@zju.edu.cn

Liuqing Chen
Zhejiang University
Hangzhou, China
chenlq@zju.edu.cn

Abstract

While some work attempt to generate front-end code intelligently from UI screenshots, it may be more convenient to utilize UI design drafts in Sketch which is a popular UI design software, because we can access multimodal UI information directly such as layers type, position, size, and visual images. However, fragmented layers could degrade the code quality without being merged into a whole part if all of them are involved in the code generation. In this paper, we propose a pipeline to merge fragmented layers automatically. We first construct a graph representation for the layer tree of a UI draft and detect all fragmented layers based on the visual features and graph neural networks. Then a rule-based algorithm is designed to merge fragmented layers. Through experiments on a newly constructed dataset, our approach can retrieve most fragmented layers in UI design drafts, and achieve 87% accuracy in the detection task, and the post-processing algorithm is developed to cluster associative layers under simple and general circumstances.

1 Introduction

Graphic User Interface (GUI) builds a visual bridge between software and end users through which they can interact with each other. A good GUI design makes software more efficient and easy to use, which has a significant influence on the success of applications and the loyalty of its users.

Figure 1. The original view hierarchy is complicated before merging fragmented layers in the UI icon and it is simplified a lot after merging layers and using a single node to represent a UI icon. The dashed rectangles in the UI icon represent fragmented layers.

This manuscript aims to detect and correct issues in the GUI design drafts automatically, which is the first step of intelligent front-end code generation. Some previous work has already attempted to adopt the deep learning technique...
to generate maintainable UI view code and logic code automatically from design drafts [4, 3, 1]. The main reason for intelligent code generation is that GUI implementation is a time-consuming process for developers and they can hardly devote majority of the time to developing unique features of an application or website. To free the front-end developers from tedious and repetitive work, researchers aim at developing an intelligent code generation system in which the quality of the original design draft plays a vital role. However, the designers mainly focus on the aesthetic of their design and usually ignore some design standards. It is important to correct them before the intelligent code generation.

We are mainly concerned about correcting UI layer merging issues in design drafts which contain rich multimodal UI information and are usually produced by designers with UI design software such as Sketch. We can directly access each layer’s type, position, size, and visual images by traversing UI in a design draft. The main reason of UI merging issues is that UI designers usually stack layers to design their creative patterns such as a background pattern and a UI icon, which not only increases the burden of UI layout calculation but also impairs the maintainability of generated code. The whole pattern should be implemented as a single element. For example, as shown in Fig. 1, the view hierarchy in a design draft is very complicated without merging fragmented layers in the UI icon. After the merging process, the UI layout is simplified greatly as we can use a single node to represent a complicated pattern. Furthermore, as shown in Fig.9, the generated DOM tree without merging fragmented layers has complex nested structure and it is simplified a lot after merging layers in the UI icon, which improves the readability and maintainability of generated front-end code a lot. In summary, it is significant to merge fragmented layers in design drafts to guarantee generating front-end code of high quality. However, it is a great challenge to detect and merge associative fragmented layers.

As described above, we aim to propose an approach to detect and merge all associative fragmented layers in a design draft, which can improve the maintainability and readability of corresponding generated code. We try to solve the problem in two steps. Firstly, all layers in the UI design draft are classified into two classes by a deep learning method, which can judge whether a layer is fragmented or not. We then design a rule-based post-processing algorithm to cluster the predicted trivial layers which belong to the same pattern.

Specifically, we propose a model composed of two main parts for detecting fragmented layers, which are a backbone network and a graph neural network model. The backbone network is used to encode multimodal information of local regions within each layer’s boundary. Each design draft has an underlying hierarchy to organize layers. The hierarchy is represented by a JSON file. We first traverse the hierarchy and construct a new graph structure for it based on each layer’s position and size information. Each node in the graph represents a layer in the UI layout and its state vector is initialized with the multimodal information encoded by the backbone network. The graph neural networks update and refine vertex states through a message-passing framework. In this way, every layer finally knows about the neighbors around it and how it belongs to the UI layout. We construct a graph for the UI layout and use a GNN model to learn each layer’s features because layers in the same pattern usually have strong relationships with each other. They may have similar shapes, colors, and geometrical relations such as parallelism and inclusion. To fully exploit the relation between layers, we train the GNN model so that each layer can obtain a better final representation that contains information about how it interacts with others. An MLP classifier is finally utilized to classify each layer based on the learned representation vectors.

We conduct experiments on a proposed dataset containing 4644 graphs to evaluate our proposed model and explore the performance of different CNN backbones and GNN models. We also visualize the results of our post-processing algorithm where layers in the same pattern are bounded with the same color. Through quantitative and qualitative analysis, our work can serve as a basic approach to solving the layer merging problem. In summary, the main contributions of this paper are:

- This is the first work trying to detect and merge fragmented layers in design drafts. We also construct an open dataset which is used for the evaluation of our proposed method.
- We propose a method of constructing the graph for a UI design draft and utilizing a GNN model to solve the layer merging problem based on both visual information and underlying multimodal hierarchy information.
- We propose a UI layer merging pipeline to cluster fragmented layers contained in the same pattern, which can simplify and facilitate the high-quality UI code generation process.

2 Related work

2.1 Intelligent Code Generation

It is time-consuming and tedious to implement GUI code. To facilitate GUI code generation, researchers turn to machine learning techniques trying to generate code intelligently. Nguyen et al. [24] identified UI elements in a
given input bitmap via computer vision and OCR technique to reverse-engineer Android user interfaces. Beltramelli et al. [4] and Jain et al. [15] generated computer tokens from GUI screenshots based on Convolutional and Recurrent Neural networks. Chen et al. [7] proposed a neural machine translator to translate a UI design image into a GUI skeleton which would be beneficial for bootstrapping mobile GUI implementation. Moran et al. [23] presented a data-driven approach for prototyping software GUIs automatically which was capable of detecting GUI components and hierarchy generation. Chen et al. [10] automatically generated a generative adversarial network to generate GUI designs automatically. Some work [31, 29] utilized a generative adversarial network to generate GUI designs more quickly by a code retrieval-based approach.

### 2.2 UI Issue Detection

To guarantee the quality and correctness of generated code, there are some code static linting tools to correct programming errors and style errors. For example, styleLint [25] is an open-source plugin which helps detect errors and enforces style conventions based on over 170 built-in rules. Some work [20] detects the final GUI displaying issues, such as text overlaps, blurred screens, and missing images, to guide developers to fix the bug. Chen et al. [8] developed a deep learning-based model to predict missing labels of image-based buttons. Zhao et al. [35] formulated the GUI animation linting problem as a classification task and proposed an auto-encoder to solve the linting problem. Different from static linting, Baek et al. [2], Mirzaei et al. [21] and Su et al. [29] can dynamically analyze a mobile app GUI by a multi-level GUI Comparison Criteria. Several surveys [18, 34] compared different tools used for GUI testing of mobile applications. Recently, Degott et al. [12] adopted the reinforcement learning technique for automatic GUI testing. White et al. [32] utilized computer vision techniques for identifying GUI widgets in screenshots to improve GUI test.

### 2.3 Graph Neural Networks

Recent years have witnessed a great surge of promising graph neural networks (GNN) being developed for a variety of domains including chemistry, physics, social sciences, knowledge graphs, recommendations, and neuroscience. The first GNN model was proposed in [14], which is a trainable recurrent message passing process. To generalize the convolution operation to non-Euclidean graphs, these work [6, 11, 17] defined spectral filters based on the graph laplacian matrix. The learned filters in these spectral approaches depend on the graph structure, so they cannot generalize to a graph with different structures. Spatial-based models define convolutions directly on the graph vertexes and their neighbors. Monti et al. [22] presented a unified generalization of CNN architectures to graphs. Hamilton et al. [15] introduced GraphSAGE, a method for computing node representations in an inductive manner which operates by sampling a fixed-size neighborhood of each node and performing a specific aggregator over it. Some methods attempted to enhance the original models with anisotropic operations on graphs, such as attention [30] and gating mechanisms [5]. Xu et al. [43] aimed at improving upon the theoretical limitations of the previous model. Li et al. [19] and Chen et al. [9] tried to overcome the over-smoothing problem when GCN goes deeper.

Researchers also have great interest in utilizing graph neural networks to tackle computer vision tasks, such as 3D object detection [28], skeleton-based action recognition [31], semantic segmentation [24]. Inspired by these work, we also attempt to introduce a graph neural network model to our proposed pipeline to detect issues in the UI design drafts. More details are described in Section 3.

### 3 Methodology

In this section, we describe the proposed approach exploiting the rich multimodal information to detect the UI issues described in Section 1. As illustrated in Fig. 2, we first construct a graph representation for the UI layout in which each node stands for a layer (Section 3.2). All nodes are initialized with their encoded multimodal information as described in Section 3.3. We detect all layers which should be merged through a graph neural network followed by an MLP classifier (described in Section 3.4). It is a two-class classification task. A post-processing algorithm described in Section 5.1 is then designed to cluster the positive samples. After fragmented layers are detected and clustered, which should be merged, they are combined into UI components within some specific context and can be represented by a single layer which can significantly reduce the number of layers in the UI design drafts and help improve the quality of generated front-end code.

### 3.1 Data Introduction

The input data to our algorithm are artboards contained in files with .sketch suffix. The files are created by Sketch which is a popular UI draft design software. Each artboard represents the UI design of the Android or iOS mobile phones. The layers in the artboard are organized by a tree structure. We traverse the tree in a pre-order manner to get all layers which are represented by leaf nodes. Each layer has its own multimodal information including type, size, position, and visual images. The size and position are
Figure 2. Architecture overview: the UI layout is converted into a graph and each node represents a layer. Then a graph neural network is utilized to classify each node after which a post-processing algorithm is designed to merge associative layers.

utilized to construct a graph representation for the UI layout and all multimodal information is encoded into feature vectors to initialize each node in the graph.

3.2 Graph Construction

After the traversal of the artboard, we obtain all layers’ type, position, and size information. We construct a new tree based on the inclusion relationship between layers. The tree can be constructed online while traversing the artboard. In the beginning, an empty root node is constructed which denotes an empty canvas. When layer A is visited, it is inserted into the tree and it will finally be added into node B’s children if node B’s bounding box contains A while none of B’s children contains A.

As described in Fig. 3, a directed graph is constructed based on the new tree representing the UI layout. Actually, a tree data structure is a special kind of graph, so we keep the directed edges in the tree and meanwhile, we construct a complete undirected graph for those sibling nodes in the tree. We use the inclusion relationship to construct a tree due to the following considerations: 1) layers combined into a background usually are contained by a large layer, 2) an empty layer contains other layers to represent a UI component, 3) UI designers usually use some empty layers to split the UI layout into different parts and layers in different parts will never be merged together. The generated graph based on the new tree cuts off interactions between layers in different parts. However, there exist some circumstances, for example, a WIFI icon, under which layers are ranged in a straight line or spread over the entire UI layout and they are not contained by some large layers but can be merged into one component within some specific context. Therefore we just construct a complete graph for those sibling nodes in the tree so that the GNN model can detect these components automatically.

3.3 Feature Extraction

In this section, we describe how to encode the multimodal information to initialize each node in the graph.

We use a one-hot code to denote the layer’s type and embed it into the embedding space by a parameter matrix. For the position and size of layers, we just simply use another matrix to encode the information. All vectors are concatenated together to form an inherent representation of each layer.

It is hard to classify layers only based on the type, position, and size information. We human beings judge the UI issues mainly from shapes, colors, or some other visual features of layers in the UI layout. And we can use vision knowledge to infer the context which helps us detect issues. It is even a challenge for human beings to detect which layers should be merged together based only on the type, size, and position information. Thus it is critical to extract each layer’s rich visual information from the UI screenshots.

There are two methods to extract corresponding features from the images based on the layer’s position and size. As
Figure 3. Graph construction: The UI layout is first converted into a tree and we connect edges between sibling nodes. Each node represents a layer and the tree is constructed based on the inclusion relationship.

Figure 4. The first row shows that we extract visual features from corresponding regions in the feature map with RoI pooling. The second row shows that we propagate the cropped layer image through a CNN backbone to obtain its visual features.

shown in Fig 4, the first straightforward way is that we transform the image data inside the layer region to a fixed size regardless of the aspect ratio, and the visual features are computed by propagating the RGB matrix through a convolutional neural network followed by a fully connected layer. Inspired by [13], the second method is that we can use the RoI pooling operation to extract features. Firstly, the UI screenshot is fed into a CNN backbone to obtain a feature map. We then divide the corresponding layer window with size $H \times W$ into $h \times w$ grids. Each grid with size $H/h \times W/w$ is max pooled to output a maximum value. In this way, we can convert the features inside any valid layer region into a small feature map with a fixed spatial extent of $h \times w$. Both $h$ and $w$ are set to be 5 in our experiments. A fully connected layer is finally adopted to obtain a high-dimensional feature vector for each UI layer.

The three feature vectors described above are concatenated together to form the initial representation of the nodes in the graph. Through the graph neural network model, each node in the graph finally learns how it belongs to the graph based on the types, positions and visual features of its neighbors.

3.4 Graph Neural Networks

A typical graph neural network can be implemented by a messaging-passing framework. Every node updates its state mainly through three steps in the form:

$$m_{ij} = MESSAGE(h_i, h_j)$$
$$a_i = AGGREGATE(m_{ij} : j \in \mathcal{N}_i)$$
$$h'_i = UPDATE(h_i, a_i)$$  \quad (1)$$

Each neighbor of the node prepares a message with $MESSAGE$ function to propagate along the edge. Then each node uses an $AGGREGATE$ function to integrate all messages from its neighbors, and updates its state based on the messages received and the state itself. Usually $MESSAGE$ and $UPDATE$ function are implemented as a multi-perceptron network and the $AGGREGATE$ function should be a permutation invariant function to eliminate the influence of message input order.

Each neighbor is involved in the state updating process differently because layers in the same UI component have more influence on each other than layers outside. Considering the various importance of layers, we introduce the attention mechanism in [30] to our model to learn the importance of different neighbors automatically. It uses a weight matrix $W \in \mathbb{R}^{F \times F'}$ to embed all nodes’ feature vectors through a linear mapping. A shared parameter $a \in \mathbb{R}^{2F'}$ is adopted to calculate the attention coefficients in the form:

$$e_{ij} = \sigma(a(W_h_i, W_h_j))$$  \quad (2)$$

where $\sigma$ is the activation function and $W_h_i$ is the feature vector of each UI layer.
Table 1. Comparisons of different CNN and GNN combinations on four evaluation metrics

| method         | precision | recall | accuracy | f1-score |
|----------------|-----------|--------|----------|----------|
| VGG16+GAT      | 0.880     | 0.868  | 0.867    | 0.874    |
| VGG16+GIN      | 0.862     | 0.860  | 0.852    | 0.861    |
| VGG16+GCN      | 0.855     | 0.886  | 0.860    | 0.871    |
| VGG16+GCNII    | 0.856     | 0.875  | 0.860    | 0.865    |
| ResNet50+GAT   | 0.860     | 0.876  | 0.862    | 0.868    |
| ResNet50+GIN   | 0.860     | 0.879  | 0.863    | 0.869    |
| ResNet50+GCN   | 0.862     | 0.873  | 0.864    | 0.869    |
| ResNet50+GCNII | 0.867     | 0.884  | 0.870    | 0.875    |

$\sigma(.)$ is a LeakyReLU activation function. The coefficients are normalized by a softmax function:

$$\alpha_{ij} = \text{softmax}_{j}(e_{ij}) = \frac{\exp(e_{ij})}{\sum_{k \in N_i} \exp(e_{ik})}$$  \hspace{1cm} (3)

$\alpha_{ij}$ can be used to measure how important node $i$ is to node $j$. The update function of node $i$ can be written as:

$$h_i = \sigma\left(\sum_{j \in N_i} \alpha_{ij}W h_j\right)$$  \hspace{1cm} (4)

Another reason for using graph attention layers in our model is that we need to deal with graphs with diverse topologies. Our model should generalize well to design drafts which are unseen during the training phase because completely different graphs are generated for different UI layouts. For the inductive learning task, it is not suitable to use spectral-based graph neural networks which cannot generalize to graphs with unseen topology. Graph attention networks, however, can directly be applied to inductive learning and can deal with directed graphs. We replace the graph attention model with different spatial-based models to observe their performance in this task.

Through the graph neural network, the final state vectors of each node are fed into an MLP classifier to judge whether a layer should be merged or not.

4 Experiments

4.1 Implementation Details

Data Generation: GUI in mobile Apps are usually scrolled vertically and layer relations are often within local regions. Moreover, a complete artboard may consist of many trivial layers. If we construct a graph for a complete artboard based on the algorithm described in Section 3.2, the large-scale graph may not even fit into the GPU memory. Layers rarely have a long-distance dependency on each other, that is, a layer has a weak relationship with those which are far from it. Given the reasons above, we can split
an artboard into small patches and the model can look into the local regions to detect fragmented layers. Specifically, we scale the height and width of the artboard to a multiple of 750. The artboard is then divided into small windows of size 750×750. A layer will be included in a window if its center is located inside the window. The graph construction algorithm converts the UI layout in the window into a graph which is then fed into the GNN model. Each artboard has a corresponding image which is resized and cut into small patches in the same way. The images serve for each layer’s visual feature extraction.

As the graphs generated in the same artboard may have great similarity, we split the dataset in granularity of artboard to test the model’s generalization ability and we use 80% of the dataset as the training set, 10% as the validation set, and the last 10% as the test set. Finally, a total of 4644 graphs are used for training, and 1048 graphs for validation and test respectively.

**Details of Training:** We adopt the VGG16 model pre-trained on the ImageNet dataset as our visual feature extraction backbone. We also conduct experiments by replacing VGG16 with ResNet50. The convolutional neural networks are also trained to adapt to our task. The first ten layers of VGG16 and the first three residual blocks of ResNet50 are fixed during training. The graph attention networks (GAT) consist of four graph attention layers. To stabilize the training process, we use multi-head attention in each layer similar to [30]. In the first three layers, each has 4 attention heads and we concatenate them into a single vector which is then followed by ELU non-linearity. For the last layer, it has 6 attention heads and we average the output. We add a skip connection from the previous attention layer to the current one. The skip connection structure consists of a single MLP layer. We utilize the Adam optimizer with an initial learning rate of 1e-3 and it is reduced by half if the validation loss does not change for 10 epochs until it reaches the minimum value of 1e-6. We implement our algorithm with the Pytorch and Pytorch-Geometric library.

**Evaluation Metric:** We evaluate and compare different models’ performance for UI issue detection with four commonly used metrics: accuracy, precision, recall, and f1-score.

### 4.2 Results

**Quantitative Analysis:** Our GNN model is used to detect fragmented layers in the UI layout which should be merged with others. It is an inductive learning task in which GNN models generalize well to unseen graphs. Therefore GNN models are implemented by a spatial-based method which usually is based on a message-passing framework. We conduct experiments on different GNN models and report their performance based on various evaluation metrics.

We use the first visual feature extraction method introduced in Section 3.3. For popular GCN in [17], it consists of three graph convolution layers which are implemented in a spatial-based manner. For fairness, we also add the same skip connection to the GCN model. For the GCNII model introduced in [9] and the GIN model [33], we use the same configuration in the original papers. To combine different advantages of CNN and GNN, we concatenate the output of CNN and GNN to form the final representation vector used for classification. As shown in Table 1, with VGG16 as the backbone network, the GAT model performs best on most evaluation metrics. The GCN model and GCNII model can outperform other GNN models regarding the recall metric. Although the GIN model is as powerful as WL-test, it can not well adapt to our fragmented layer detection task when combining VGG16 model. When replacing VGG16 with ResNet50 as the backbone network, most GNN models have a slight increase in recall and accuracy metrics. Although the combination of ResNet50 and GCNII outperforms all the other combinations on accuracy and f1-score, it just has a little improvement on the fragmented layer detection task. We finally adopt the combination of VGG16 model and GAT model because they have fewer parameters and training time while achieving comparable performance. It seems that different GNN models perform similarly on the four metrics, however, they play a vital role in the fragmented layer detection task. We compare the performance of a single CNN and a combination of CNN and GNN to validate the effectiveness of the GNN models. More details will be described in Section 4.3.

**Qualitative Analysis:** We visualize some detection results
in the test dataset as shown in Fig. 5. Each fragmented layer, which should be merged, is marked with a green rectangle. We can see that designers usually stack layers to draw a UI icon, a decorative pattern or art font, and a background pattern. These fragmented layers should be merged into a whole part before generating front-end code intelligently. The merging process, which decreases the number of layers a lot, not only eases the burden of layout calculation in the intelligent code generation but improves the quality and maintainability of generated front-end code. Our model can detect a majority of fragmented layers in UI icons, decorative patterns, art fonts and complicated background patterns, and it can serve as a useful visualization tool to remind the designers of merging fragmented layers manually. In Section 5, we design and evaluate a simple post-processing algorithm to merge trivial layers detected by our model to avoid repetitive and time-consuming human effort for merging layers.

Failure Cases Analysis: While our approach can detect the majority of trivial layers without specific context in a UI icon or a simple decorative pattern, there are still some challenging cases where our algorithm performs not well as shown in Fig. 6. For some complicated background patterns which are composed of many tiny layers, it is hard to retrieve all fragmented layers as the layers are spread over the entire background. One possible explanation is that it requires some cognitive knowledge about UI aesthetics and UI design to correctly classify and merge fragmented layers in the background. For example, in Figure 6, human beings conclude that layers bounded by blue rectangles are fragmented mainly based on some high-level knowledge, that is UI visual aesthetic. However, it is difficult for machines to learn about such knowledge, which is a great challenge for classifying layers correctly in the background. In Fig. 6, we also notice that our model fails to detect the large background layer because the layer is occluded by foreground layers. Furthermore, layers may be combined by a boolean operation to form some specific shapes but it does not matter that our model fails to detect these layers, because we can merge them directly if we know layers are combined by a boolean operation.

4.3 Ablation Study

| Table 2. The performance of our pipeline with and without GNN |
|------------------|----------|----------|----------|----------|
| method           | precision| recall   | accuracy | f1-score |
| VGG              | 0.844    | 0.819    | 0.829    | 0.831    |
| VGG+GAT          | 0.880    | 0.868    | 0.867    | 0.874    |

Single CNN vs CNN+GNN: We remove the GNN model while keep the MLP classifier to validate the effectiveness of GNN. Table 2 reports the performance of the single CNN with the first visual extraction method. We can see that the combination of GNN and CNN outperforms the single CNN model by 4% regarding the accuracy. And with GNN model, we can retrieve more fragmented layers as the table shows a great increase of recall by about 5%. Due to the great improvement on the four metrics, we can conclude that the GNN model plays a vital role in the detection task. Fragmented layer prediction with a single CNN model only focus on features within local regions which ignores layer’s relations with others. It tends to learn the common features of the fragmented layers. When using the spatial-based graph neural networks, each node in the graph can update its own state according to the features of its neighbors. Each layer in the UI layout can finally learn how to interact with layers around, which can benefit for detecting fragmented layers a lot.

| Table 3. Comparisons between two visual feature extraction methods. |
|------------------|----------|----------|----------|----------|
| method           | precision| recall   | accuracy | f1-score |
| RoI              | 0.835    | 0.835    | 0.825    | 0.835    |
| CropCNN          | 0.880    | 0.868    | 0.867    | 0.874    |

1. RoI means that the method extracts local features by the RoI pooling operation.
2. CropCNN means that we use CNN to extract features from cropped layer image directly.

Visual Feature Extraction: We compare the performance of two visual feature extraction methods described in Section 3.3. The results in Table 3 show that using the CNN backbone to extract features directly from the corresponding local region performs better in our task, which outperforms the method of using the RoI pooling operation greatly. One possible explanation is that the layers which belong to the same part usually have some similar visual features such as color and shapes. Layers with similar visual features should be represented by feature vectors with great similarity. The key difference between the two feature extraction methods is where the extracted features come from. For the RoI pooling method, it extracts features from the feature map obtained by propagating the image through a CNN backbone. Similar layers in different positions obtain different feature representations because different local regions in the feature map have great dissimilarity, which explains the poor performance of the RoI pooling method. As for the method of using a CNN backbone directly, it extracts features from the original image directly. Layers with similar visual features obtain similar representations because we propagate similar cropped layer images through the same CNN backbone. Therefore, the nodes in the graph
can be better initialized and our model detects fragmented layers better when adopting the method of extracting features from original images.

| method | precision | recall | accuracy | f1-score |
|--------|-----------|--------|----------|----------|
| LE     | 0.818     | 0.910  | 0.85     | 0.861    |
| VF     | 0.880     | 0.868  | 0.867    | 0.874    |
| LE+VF  | 0.876     | 0.876  | 0.868    | 0.876    |

1 LE denotes that the initial representation contains only layer’s properties.  
2 VF stands for using visual features only.  
3 LE+VF means that we classify a layer from its size, type, position, and visual features.

**Feature Fusion:** Each layer has its own type, x-y coordinates, size, and visual features such as colors and shapes. Here we try to investigate which part of information plays a more important role. We initialize the initial representation vectors of nodes in the graph with three strategies. As Table 4 shows, we remove the CNN backbone and utilize a GNN model only to classify layers based on their types, positions, and sizes (LE). Then we conduct an experiment on detecting layers based on their visual features only (VF). Finally, we fuse these features to initialize each node’s state (LE+VF). Table 4 shows that the type, position, and size information can help retrieve more fragmented layers. However, the model retrieves many irrelevant layers without the help of visual features as the precision is just 81.8%. With visual features only, the model can better guarantee the quality of retrieved layers but the recall decreases a lot by around 4%. When combining all these features, the model has a slight increase regarding recall, accuracy, and F1-score compared with the VF method. In summary, the type, position, and size information can help to retrieve more fragmented layers while the visual features help to remove irrelevant layers.

5 Application

In this section, we utilize our proposed fragmented UI layer detector followed by a post-processing algorithm to address the issue of merging associative layers. We design a rule-based algorithm to judge whether two layers belong to the same pattern in the UI layout. Fig[7] shows how our approach can be utilized as a useful tool to correct issues in design drafts before intelligent code generation with imgcook [1].

5.1 Post Processing Algorithm

In Section 3.2, a tree in which nodes represent layers is constructed for the UI layout according to the inclusion relationship between layers. We use the tree to facilitate the layer merging process. Based on some prior knowledge, we conclude some rules: 1) layers that are close enough to each other should be merged together; 2) if a non-leaf node in the tree is classified into a positive sample, it is merged with its children with large probability. The reasons are as follows: 1) the distance is small between layers in UI icons, such as a Wifi icon; 2) designers may stack layers inside an empty layer to form a pattern as shown in Fig[8]; 3) layers in the background patterns are contained in a large layer. Therefore, our post-processing algorithm is mainly based on the distance and inclusion relations between layers.

We construct an adjacency matrix for all detected layers, where $A_{ij}$ denotes layer $i$ and $j$ should be merged together. A depth-first or breadth-first search algorithm is then utilized to find all connected components of which each stands for a merging area. Specifically, we traverse all fragmented layers detected by our model and merge layers between which the distance of the centers is below a predefined threshold. These layers are ignored in the following steps. We search the tree in a top-down manner. If a non-leaf node is predicted to be merged, it should be merged with its child nodes which are also positive samples, and we set the corresponding entries in the matrix to be 1.

5.2 Evaluation

We apply the proposed approach to a real application scenario where we need to merge fragmented layers into
a whole part. As shown in Fig.8a, we visualize some results of merging trivial layers in the UI layout design draft. We bound layers that should be merged together with the same color, and we use different colors to distinguish different merging areas. Our post-processing algorithm can easily merge layers in the type of UI icons and simple decorative patterns based on the rules described above. We can also discover that UI icons may be contained in an empty layer and layers in UI icons are close to each other. Designers stack many decorative layers on a large layer to form a background for the UI design draft as shown in the down-left of Fig.8a. As our approach merges layers intelligently, which spares a lot of human effort to correct UI issues, it has a great potential in facilitating high-quality and maintainable code generation as shown in Fig.1. However, it is hard to tell the foreground and background apart as shown in Fig.8b. The algorithm merged the “battery” icon with some background parts together as the rule-based approach ignores the semantic information. We will continue to optimize our algorithm in the future.

(a) Our algorithm can merge fragmented layers in UI icons successfully.

(b) The algorithm falsely merges background layers and icon layers together.

Figure 8. Some visualization results of our approach in a real application scenario. Layers which should be merged together are bounded with the same color. Different colors are used to distinguish different merging areas.

Figure 9. DOM trees generated by imgcook before and after merging fragmented layers in design drafts.

6 conclusion

We have proposed an approach to detect fragmented layers in the UI design draft based on a backbone network and a graph neural network model. We also design a rule-based post-processing algorithm to merge associative trivial layers into a whole pattern. By converting the UI layout into a graph representation, each layer can interact with each other through the GNN model and the final learned vector can better express layers’ multimodal information in the UI layout. Our experiments show that the approach can easily detect and merge associative fragmented layers in icons-related UI elements but performs not very well on retrieving layers in complicated background patterns. We conduct a user study to further demonstrate the effectiveness of our approach to improve the quality of generated code.

In this work, we predict each layer to be positive or negative, which leaves a huge burden on the layer merging process. In the future, we can define a more concrete classification task. Specifically, a layer can be classified into one of the following types, such as a UI icon layer, a background layer, and a word font layer.

References

[1] Alibaba. Intelligent code generation for design drafts, 2021.
[2] Y.-M. Baek and D.-H. Bae. Automated model-based android gui testing using multi-level gui comparison criteria. In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, page 238–249, New York, NY, USA, 2016. Association for Computing Machinery.

[3] F. Behrang, S. Reiss, and A. Orso. Guifetch: supporting app design and development through gui search. pages 236–246, 05 2018.

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

[5] X. Bresson and T. Laurent. Residual gated graph convnets. arXiv preprint arXiv:1711.07553, 2017.

[6] J. Bruna, W. Zaremba, A. Szlam, and Y. LeCun. Spectral networks and locally connected networks on graphs. arXiv preprint arXiv:1312.6203, 2013.

[7] C. Chen, T. Su, G. Meng, Z. Xing, and Y. Liu. From ui design image to gui skeleton: A neural machine translator to bootstrap mobile gui implementation. In 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), pages 665–676, 2018.

[8] J. Chen, C. Chen, Z. Xing, X. Xu, L. Zhu, G. Li, and J. Wang. Unblind your apps:predicting natural-language labels for mobile gui components by deep learning. Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, Jun 2020.

[9] M. Chen, Z. Wei, Z. Huang, B. Ding, and Y. Li. Simple and deep graph convolutional networks. In International Conference on Machine Learning, pages 1725–1735. PMLR, 2020.

[10] S. Chen, L. Fan, C. Chen, T. Su, W. Li, Y. Liu, and L. Xu. Storydroid: Automated generation of storyboard for android apps. In 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), pages 596–607, 2019.

[11] M. Defferrard, X. Bresson, and P. Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. Advances in neural information processing systems, 29:3844–3852, 2016.

[12] C. Degott, N. P. Borges Jr, and A. Zeller. Learning user interface element interactions. In Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, pages 296–306, 2019.

[13] R. Girshick. Fast r-cnn. In Proceedings of the IEEE international conference on computer vision, pages 1440–1448, 2015.

[14] M. Gori, G. Monfardini, and F. Scarselli. A new model for learning in graph domains. In Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005., volume 2, pages 729–734. IEEE, 2005.

[15] W. L. Hamilton, R. Ying, and J. Leskovec. Inductive representation learning on large graphs. In Proceedings of the 31st International Conference on Neural Information Processing Systems, pages 1025–1035, 2017.

[16] V. Jain, P. Agrawal, S. Banga, R. Kapoor, and S. Gulyani. Sketch2code: Transformation of sketches to ui in real-time using deep neural network, 2019.

[17] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. arXiv preprint arXiv:1609.02907, 2016.

[18] T. Lämsä. Comparison of gui testing tools for android applications. University of Oulu, 2017.

[19] G. Li, M. Muller, A. Thabet, and B. Ghanem. Deep-gcns: Can gcns go as deep as cnns? In Proceedings of the IEEE/CVF International Conference on Computer Vision, pages 9267–9276, 2019.

[20] Z. Liu, C. Chen, J. Wang, Y. Huang, J. Hu, and Q. Wang. Owl eyes: Spotting UI display issues via visual understanding. CoRR, abs/2009.01417, 2020.

[21] N. Mirzaei, J. Garcia, H. Bagheri, A. Sadeghi, and S. Malek. Reducing combinatorics in gui testing of android applications. In Proceedings of the 38th International Conference on Software Engineering, ICSE ’16, page 559–570, New York, NY, USA, 2016. Association for Computing Machinery.

[22] F. Monti, D. Boscaini, J. Masci, E. Rodola, J. Svoboda, and M. M. Bronstein. Geometric deep learning on graphs and manifolds using mixture model cnns. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 5115–5124, 2017.

[23] K. Moran, C. Bernal-Cárdenas, M. Curcio, R. Bonett, and D. Poshvyvanyk. Machine learning-based prototyping of graphical user interfaces for mobile apps. IEEE Transactions on Software Engineering, 46(2):196–221, 2020.

[24] T. A. Nguyen and C. Csallner. Reverse engineering mobile application user interfaces with remau (t). In 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 248–259, 2015.

[25] C. of stylelint. Stylelint developer documentation, 2020.
[26] X. Qi, R. Liao, J. Jia, S. Fidler, and R. Urtasun. 3d graph neural networks for rgbd semantic segmentation. In Proceedings of the IEEE International Conference on Computer Vision, pages 5199–5208, 2017.

[27] S. P. Reiss. Seeking the user interface. In Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, ASE ’14, page 103–114, New York, NY, USA, 2014. Association for Computing Machinery.

[28] W. Shi and R. Rajkumar. Point-gnn: Graph neural network for 3d object detection in a point cloud. In Proceedings of the IEEE/CVF conference on computer vision and pattern recognition, pages 1711–1719, 2020.

[29] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su. 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, page 245–256, New York, NY, USA, 2017. Association for Computing Machinery.

[30] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio. Graph attention networks. arXiv preprint arXiv:1710.10903, 2017.

[31] Y.-H. Wen, L. Gao, H. Fu, F.-L. Zhang, and S. Xia. Graph cnns with motif and variable temporal block for skeleton-based action recognition. In Proceedings of the AAAI Conference on Artificial Intelligence, volume 33, pages 8989–8996, 2019.

[32] T. D. White, G. Fraser, and G. J. Brown. Improving random gui testing with image-based widget detection. In Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, pages 307–317, 2019.

[33] K. Xu, W. Hu, J. Leskovec, and S. Jegelka. How powerful are graph neural networks? arXiv preprint arXiv:1810.00826, 2018.

[34] S. Zein, N. Salleh, and J. Grundy. A systematic mapping study of mobile application testing techniques. Journal of Systems and Software, 117:334–356, 2016.

[35] D. Zhao, Z. Xing, C. Chen, X. Xu, L. Zhu, G. Li, and J. Wang. Seenomaly: Vision-based linting of gui animation effects against design-don’t guidelines. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, ICSE ’20, page 1286–1297, New York, NY, USA, 2020. Association for Computing Machinery.

[36] T. Zhao, C. Chen, Y. Liu, and X. Zhu. Guigan: Learning to generate gui designs using generative adversarial networks. In 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), pages 748–760, 2021.