Front. Comput. Sci., 2022, 0(0): 1–19
https://doi.org/10.1007/sxxxxx-yyy-zzzz-1

RESEARCH ARTICLE

Automated GUI Widgets Classification

Kabir Sulaiman SAID[1], Liming NIE[1,2](✉), Yuangchang LIN[1], Yaowen Zheng[2], Zuohua DING[1]

1 School of Information Science and Technology, Zhejiang Sci-Tech University, Hangzhou 310018, China

2 School of Computer Science and Engineering, Nanyang Technological University, Singapore 639798, Singapore

Abstract The software engineering community pays much attention to graphical user interface (GUI) widget classification. Prior work on widgets classification primarily used mature methods from the computer vision domain, which involved manually designing the model architecture and then training the architecture using labeled data to produce the final classification model. Unfortunately, manually designing the architecture takes a long time and demands a certain amount of skill. Therefore, an approach for automating the process is crucial. This paper presents the NASW approach for automating model architecture design and generating a widget classification model from a labeled widget dataset. The proposed NASW automatically generates a high-performance classification model NASW_m using Neural Architecture Search (NAS) capabilities. Several experiments are carried out on a large-scale widget dataset with over 235k samples to assess the efficacy of the proposed approach. The proposed method NASW completes the model training 3x faster than the comparative method on the same training data. Meanwhile, the generated model NASW_m achieves a 5% improvement on multiple metrics. Moreover, we discover that the amount of training data has an effect on model performance. With NASW, we can greatly save the time of architecture design and improve classification performance. The successful practice of automation methods on widget classification tasks can provide enlightenment for task automation in the software engineering field.

Keywords GUI, Widgets Classification, Neural Architecture Search, One-shot NAS

1 Introduction

Graphical User Interface (GUI) widgets are essential in present mobile applications (apps). They

Received month dd, yyyy; accepted month dd, yyyy

E-mail: nlm@zstu.edu.cn

are used to interact with users and help users navigate between activities [1, 2]. As most apps interact with their users through the GUI, several aspects of the app development process, such as GUI design and testing, become critical tasks [3].These tasks often require classifying the GUI widgets into their peculiar domain types (e.g., *CheckBox, EditText*, and *ProgressBar*) [1,3]. Therefore, the ability to obtain better widget classification performance has become one of the keys to the success of these tasks.

Researchers in recent years have proposed many techniques for improving widget classification performance [1,4,5]. For example, Anastasia Dubrovina et al. [5] utilized the traditional Computer Vision (CV) technique to classify the GUI widgets. The authors created a small dataset of 258 segmented widgets from five GUI widgets classes. A SVM (Support Vector Machine) classifier was trained using this dataset. Also, Moran, K. et al. [1] proposed a DL (Deep Learning) technique to classify GUI widgets into their domain-specific type. The authors used the deep learning algorithm, a CNN (Convolutional Neural Network) architecture, to classify the GUI widgets. Additionally, they created a large GUI widget dataset that contains 15 widgets classes, which are commonly found in mobile apps.

However, the prior works rely heavily on the manual design of the CNN architecture. This involves many manual efforts to design an exquisite model architecture and often requires specialized skills [6]. It entails many trials and errors, and experimentation is time-consuming and expensive [7]. Although transfer learning can achieve the optimal classification model for some tasks, it is typically better to build a specific model architecture for the best possible performance [8]. Therefore, an approach that can automate the architecture design would be helpful. Researchers and practitioners can leverage such an approach to mitigate the time taken to design an exquisite architecture and improve performance.

This paper proposes the NASW approach to automatically search for exquisite architecture and generate a classification model based on a labeled widget dataset. NASW utilizes the ability of Neural Architecture Search (NAS) to reduce time consumption for designing exquisite architecture and improve performance. To search for architecture based on the widget dataset, we first design a cell-based search space that incorporates a wide diversity of architectures. Second, we use the continuous relaxation technique to transform our search space into continuous search space, drawing inspiration from the one-shot differentiable formulation of NAS [8, 9]. Then, using a gradient descent algorithm, we search for a cell to serve as the architecture's core element. The best cells are stacked to create a candidate architecture. The search and evaluation of candidate architectures are combined into one stage. Finally, based on the validation performance after the search, we choose the best-performing candidate architecture. This architecture is retrained on the large training set to obtain the NASW_m model for the widgets classification.

We conduct experiments on 235,728 samples of the ReDraw [1] GUI widgets dataset, hereafter mentioned as the original dataset. This original dataset is extracted from thousands of Android apps. In addition, we construct another dataset from the original dataset, hereafter referred to as the standard dataset. We created the standard dataset to adjust the class imbalance in the original dataset, which skewed widget types towards certain classes. Thus, we conducted our experiments on the two benchmark datasets, the original and the standard datasets.

Accordingly, we propose three research questions to evaluate NASW from three dimensions: *time, performance*, and *training data size*. The results show that NASW obtained the best architecture in less than an hour in a single search. Also, in terms of training speed, the generated architecture is 3x faster than the comparative method. The results further reveal that the NASW_m surpasses the state-of-the-art methods achieving an F1-score of 90% on the original dataset, also outperforms the mature image classification model, ResNet50, with a precision of 92% on the standard dataset. Our evaluation further illustrates how the NASW_m outperforms ResNet50 as the training data increases. In summary, our paper contributes the following:

- We proposed an approach called NASW for searching an exquisite network architecture for GUI widget classification, which explores the potential of the NAS algorithm in automating software engineering tasks.
- The searched architecture based on NASW not only attains state-of-the-art performance, but the time spent to search and train the architecture is also significantly reduced.
- We constructed a standard dataset with a total of 15 classes and 5k samples in each class to mitigate the noise and the class imbalance in the original large GUI widgets dataset. The code, standard dataset, and trained models are available at our Github repository[1)].

The remaining sections were organized as follows: Section 2 discusses the preliminary, including background and motivation. Section 3 explains the approach. Section 4 describes the experimental setups, including the dataset, research questions, baseline methods, and evaluation metrics. Section 5 discusses the experimental results. Then, Section 6 and Section 7 outline the related work and the threat to the result validity, respectively. Finally, we give the conclusion and future plan in Section 8.

1)https://github.com/talk2kabir/NASW

2 Preliminary

2.1 Background

GUI: GUI is a vital mechanism used to simplify a software environment [10]. It displays objects that carry information and interpret actions that an app user can take. When the user interacts with these objects, they usually change color, size, or visibility.

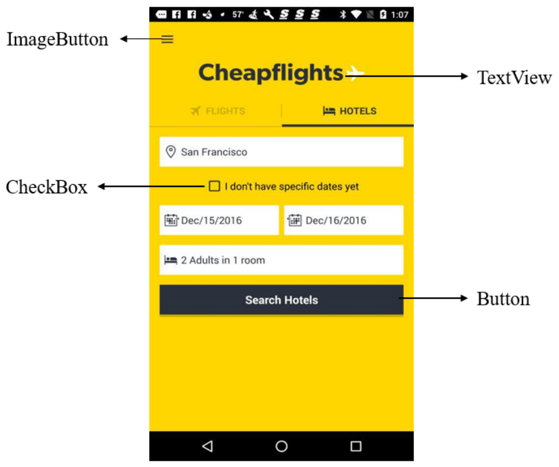

Fig. 1 GUI example

Example 2.1: Figure 1 presents a simple example of the GUI of an app with various widgets. These widgets help the users to interact with the app and perform specific tasks.

GUI Widget: A GUI widget (also known as GUI components) is an atomic graphical element with predefined functionality displayed within a GUI of a software application [1]. Each widget supports specific user interaction and emerges as a visible component of the GUI defined by the theme and produced by the rendering engine. The theme imposes a unified aesthetic design on all widgets, creating a sense of overall cohesion.

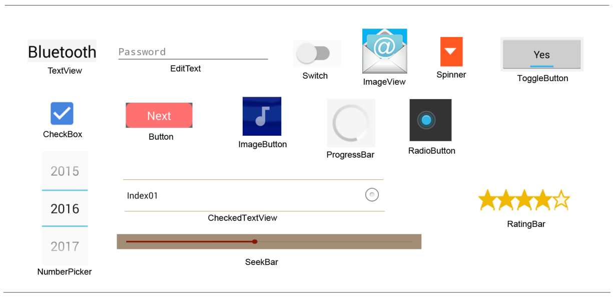

Fig. 2 Examples for GUI widgets

Example 2.2: Figure 2 shows the common GUI widgets found in modern mobile apps. As shown in Figure 2, some of the widgets share some common visual features, such as *Button* and *ToggleButton*.

Widgets classification: GUI widgets classification refers to the process of classifying widgets into their domain-specific types. The image and widgets classification, as the name implies, means to assign a class name to widget/image. Unlike other images, such as Imagenet samples, widgets images are commonly smaller in size and have different shapes. As shown in Figure 2, some widget types, such as *CheckBox*, have a square shape and *Button*, a rectangular shape. The widget classification method supports many software engineering tasks such as code generation and testing. The idea is given a widgets image; the widget classification model aims to identify the specific class of the widget. Current methods used to classify widgets include traditional machine learning (ML) and DL algorithms. A CNN, which is a DL algorithm, is used in several works [1, 3, 4] to classify the widgets into their peculiar domain type (e.g. *ImageButton, CheckBox*). However, manually designing the CNN architecture is typically time-consuming and often challenging to achieve good architecture.

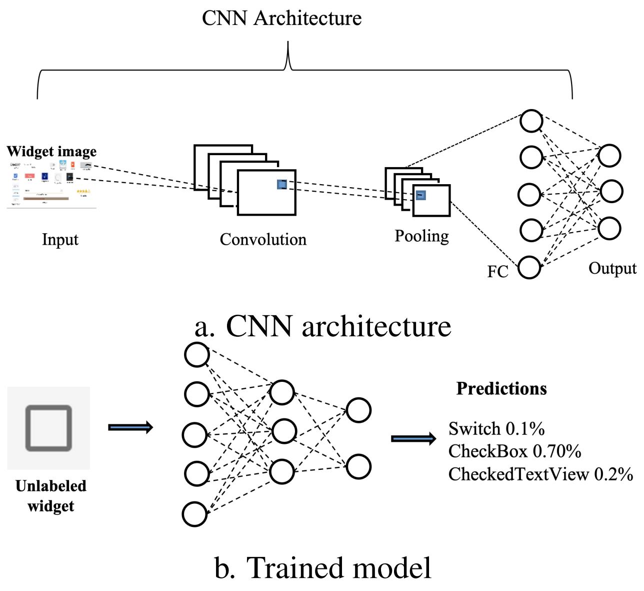

Fig. 3 CNN architecture and classification model

Example 2.3: Figure 3a shows a simple CNN architecture typically used in the classification tasks. During training, the CNN architecture receives an input (e.g., widget image) to learn certain salient features from the input. After the training, the model is used to predict the classes of the widgets that are unseen to the model during the training. Figure 3b illustrates how the trained model is applied to predict the class of *CheckBox* widget.

2.2 Motivation

Previous GUI widgets classification works are limited to manually designed network architectures [1, 3]. For instance, Moran et al. [1] introduced an approach to automatically generate GUI prototypes using a DL model to detect and classify thousands of Android GUI widgets types. Chen et al. [3] proposed combining text-based and non-text-based models to improve the overall performance of GUI widget detection while classifying the widgets with the ResNet50 model. However, these were all implemented based on manually designed CNN architectures. Developing excellent architecture for a specific task is typically time-consuming and necessitates a certain level of proficiency. Therefore, automating the architecture design would help reduce the time required to design an exquisite architecture for the widgets classification task.

The design of novel neural network architectures is crucial for deep learning progress. We observe that NAS has recently attracted lots of attention for its potential to democratize deep learning. The role of NAS in discovering task-specific architecture based on user configurations (e.g., dataset, evaluation metric, etc.) is important for a practical end-to-end deep learning platform. Network architectures discovered using the NAS method have outperformed the manual designs in many domains [11–13]. For example, Chen et al. [11] adopted a NAS method to classify metal defects. Several other works utilized the NAS methods to support different tasks such as regression analysis [12, 13], language modelling [14], and image classification and detection [7, 9, 15].

Inspired by the NAS development, we attempt to explore the possibility of applying the NAS algorithm to (1) mitigate the time taken to design an optimal architecture, (2) and improve the performance of GUI widget classification.

3 Approach

We propose the NASW approach that leverages the differentiable one-shot NAS [8, 9] to automatically search for an effective architecture for GUI widgets classification. Figure 4 shows an overview of the NASW approach. First, we introduce a cell-based search space from which we search for a computational cell as the core element of the architecture. Second, in the architecture search, we find the architecture of a cell by getting the top-k strongest operations from different cell nodes. In this manner, discrete cell-based search space is transformed to continuous architecture search space to optimize a candidate architecture based on its validation set accuracy by gradient descent. The candidate architectures are constructed by stacking the best cells. Finally, we trained the best candidate architecture from scratch to obtain the widgets classification model NASW_m. The subsequent subsections explain the search space, the strategy used to explore the cell-based search space, and model training.

3.1 Cell-based Search Space

A cell is defined as a small, convolutional unit. It is typically stacked many times to construct a complete network architecture. In more technical terms, it is an ordered sequence of N nodes in a directed acyclic graph. Nodes x_i are latent representations (for example, a feature map). Each edge (i, j) between the nodes is connected with an operation O that updates the node x_i. Each cell contains a pair of input nodes and a single output node. The outputs of the previous two cells define the input nodes. The output is the concatenation of the intermediate nodes. The main advantage of the cell-based search space is that it produces smaller, more efficient architectures that can be combined to create larger architecture.

A cell is defined as a small, convolutional unit. It is typically stacked many times to construct a complete network architecture. In more technical terms, it is an ordered sequence of N nodes in a directed acyclic graph. The main advantage of the cell-based search space is that it produces smaller, more efficient architectures that can be combined to create larger architecture. Thus,

Thus, we search for cells as the key components of the architecture. The architecture is then built

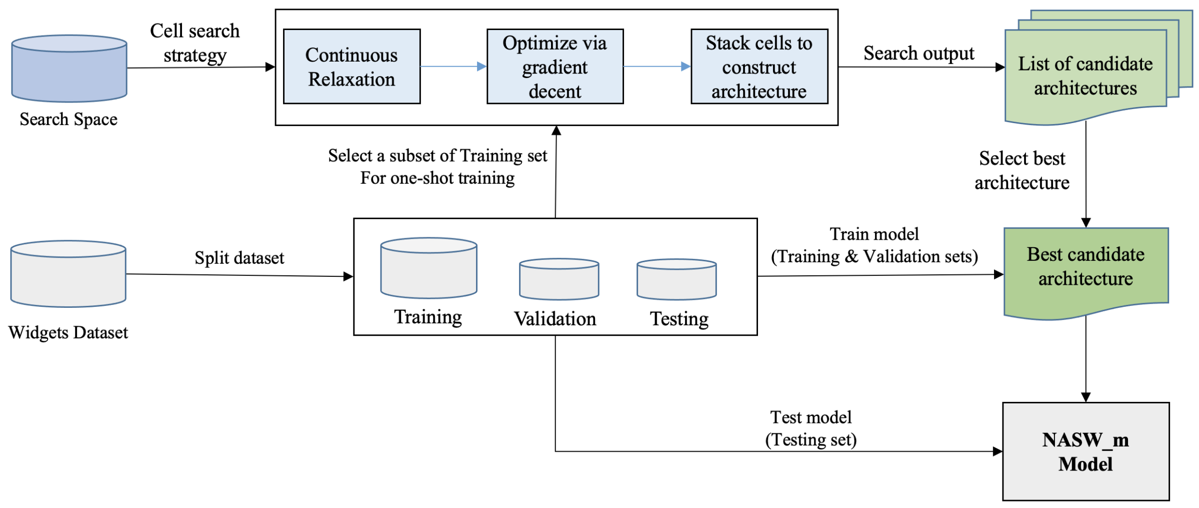
Search Space
Cell search strategy
Continuous Relaxation
Optimize via gradient decent
Stack cells to construct architecture
Search output
List of candidate architectures
Select a subset of Training set For one-shot training
Select best architecture
Widgets Dataset
Split dataset
Training
Validation
Testing
Train model (Training & Validation sets)
Best candidate architecture
Test model (Testing set)
NASW_m Model

Fig. 4 Overview of NASW

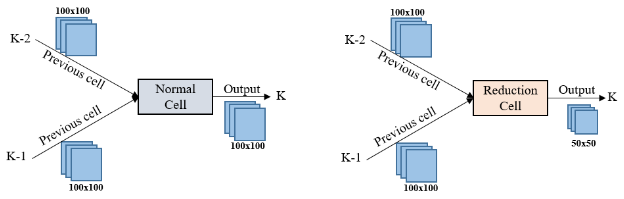

Fig. 5 An abstract view for normal and reduction cells

by stacking the best cells. Normal and Reduction cells are the two types of cell structures that are being searched. Figure 5 illustrates the abstract view of the normal and reduction cells. A *normal cell* is a normal block that computes the feature map of an image. In contrast, a *reduction cell* reduces the feature map dimensions. The convolutions and pooling operations in the normal and reduction blocks have a stride of 1 and 2, respectively.

Consequently, we build a cell-based search space that begins with a 3×3 convolution to generate effective widget classification architecture. The search skeleton body is separated into three stages linked by a network of reduction cells. Each stage stacks the normal cell 4 times. The skeleton is concluded by a classification layer, which transforms network outputs into final predictions. The cell has an ordered sequence of N nodes (feature maps) in this case, with each node connecting all preceding nodes. Each node-to-node edge is connected with eight operations denoted by $\mathcal{O}$. These operations are 3×3 separable convolutions, 3×3 dilated convolutions, 3×3 max pooling, 3×3 average pooling, 5×5 separable convolutions, 5×5 dilated convolutions, identity (skip_connect), and none.

3.2 Architecture Search

Continuous relaxation: Searching over a discrete set of operations is challenging because the model must be trained on a specific configuration before moving on to the next configuration. This process is time-consuming. Therefore, we adopt the continuous relaxation described in [8]. Hence, we relax the categorical selection of a given operation as a softmax across all operations. Consequently, the architecture search reduces to learning a set of mixing probabilities $\alpha = \{\alpha(i, j)\}$, as shown in Figure 6. Following [8], we preserve the top-k most effective operations from different cell nodes. We build each node in the discrete architecture considering non-zero candidate operations obtained from preceding nodes. The discrete architecture is built by

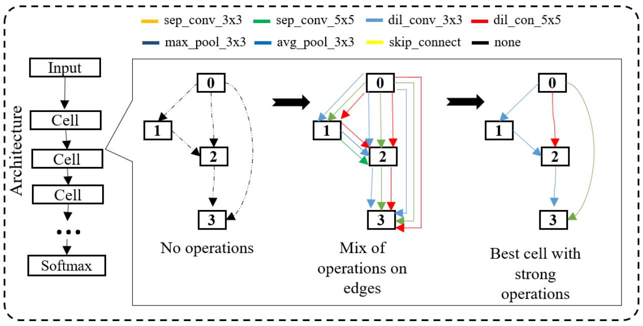

Fig. 6 Overview of cell search

replacing every mixed operation $o^{(i,j)}$ with the most probable operation, i.e., $o^{(i,j)} = argmax_{o \in O} \ \alpha_o^{(i,j)}$, where α is the architecture encoding. Thus, the discrete search space is transformed into a continuous search space in which the gradient descent algorithm works perfectly.

Optimization: We intend to simultaneously learn the weights w and the architecture α within the mixed candidate operations (e.g., convolution filters weights) through continuous relaxation. Accordingly, their validation losses can be effectively optimized through gradient descent. We optimize the validation loss using gradient descent, with T_{loss} and V_{loss} to represent the training and validation losses, respectively. Both T_{loss} and V_{loss} losses are determined by the weights w and the architecture α in the network. The aim for the architecture search is to discover α^* that reduces the validation loss V_{loss} (w^*, α^*), where the weights w^* connected with the architecture are achieved by reducing the training loss $w^* = argmin_w \ T_{loss}(w, \alpha^*)$. This optimization process is a bilevel optimization [9] with α and w as the up and low-level variables, respectively.

Given the optimized weights on the training set, the optimization problem is framed as finding α such that the validation loss is minimized.

$$min_{(\alpha)} \quad V_{loss}(w^*(\alpha), \alpha) \tag{1}$$

$$s.t. \quad w^*(\alpha) = argmin_w \quad T_{loss}(w, \alpha) \tag{2}$$

Thus this method, the entire framework can be differentiated based on layer weights and hyperparameters, enabling end-to-end architecture search tasks easy.

Searching setup: We train the proposed one-shot model covering all candidate operations to search for an optimal architecture. Throughout the search process, the aim is to discover the best set of hyperparameters. For this reason, the training is divided into two parts. The first part is used to optimize network parameters such as convolutional weights. In contrast, the second part is used to optimize hyperparameters. We randomly sampled two subsets of the training set dataset with 10% D_{train} and 2.5% D_{val} samples to speed up the training time. The D_{train} is used to train network weights, while the D_{val} to update hyperparameters. Momentum SGD is used to optimize network weights. In addition, we utilize an initial learning rate of 0.2, weight decay of 3×10^{-4}, and momentum of 0.9. Besides, we optimize the hyperparameters using adam optimizer with a learning rate ξ of 3×10^{-3} fixed, a momentum of (0.5, 0.999), and a weight decay of 10^{-3} In addition, we use Adam optimizer for hyperparameters with a fixed ξ of 3×10^{-3}, a weight decay of 10^{-3}, and a momentum of (0.5, 0.999). The one-shot architecture search step is attained in an end-to-end fashion. In addition, our proposed approach automatically combines the search and evaluation of each candidate architecture in one stage. Also, following the traditional architecture search procedures, we use separate stages for searching an architecture and final model training.

Algorithm 1 NASW Architecture Search

```
Input: Super-network, A subset of the train set $D_{train}$ and $D_{val}$, A mixed operation parameterized by $\alpha^{(i,j)}$ for each edge $(i, j)$,
Output: An architecture $\alpha^*$
  while not converged do
      Train the super-network                                  ▷ Train
      $\nabla\alpha V_{loss}(w - \xi\nabla_w T_{loss}(w, \alpha), \alpha)$   ▷ Optimize $\alpha$
      $\nabla_w T_{loss}(w, \alpha)$                                   ▷ Optimize $w$
  end while
  Based on the learned $\alpha$, generate the final architecture.
```

3.3 Model training

The one-shot for architecture search produces a list of candidate architectures and their accuracy on the validation set. In this step, we first select the best candidate architecture based on the validation accuracy. Then, we train the best architecture to obtain the final widgets classification model NASW_m. This architecture consists of the learned cells stacked together to form the larger model. In addition, we used an SGD optimizer with an initial ξ of 0.2 (annealed down to zero), a weight decay of 3×10^{-5}, a norm gradient clipping at 1, and a momentum of 0.9. We run the training on the entire benchmark training set and report the outcomes of the trained model NASW_m on the testing dataset. Note that the testing data was unseen to the model during the architecture search and model training.

4 Experiment Setup

This section details the settings for evaluating the proposed approach NASW, including the research questions, Dataset, Comparative Methods, and Evaluation Metrics.

4.1 Research Questions

- **RQ1. How long does it take to search and train an optimal architecture?** The reason for setting this RQ is to investigate the time taken to search for an excellent architecture based on the benchmark dataset and the training time against the comparative method.
- **RQ2. How effective is the proposed NASW against the comparative methods?** The purpose of setting this RQ is to compare the efficacy of the proposed NASW to that of handcrafted network architectures.
- **RQ3. What is the impact of the amount of training dataset?** The reason for setting this RQ is to investigate the impact of dataset training size on the performance of the proposed NASW against the baseline method.

4.2 Benchmark Dataset

We conduct experiments on a GUI widget dataset, an extensive collection of 15 GUI widget types commonly found in mobile apps (Figure 2). The dataset consists of organic and synthetically generated data hereafter referred to as the **original dataset**[2]. The organic data is extracted from various android apps. However, because some components are used more frequently than others in certain apps, some widget types, such as *TextView* and *ImageView*, account for more than 70% of the organic dataset. Thus, the synthetically generated data augment those classes with low support to mitigate the class imbalance. In general, the dataset consists of a total of 191,300 samples of organic data across 15 classes and 101,911 synthetically generated data. We split the organic dataset into 75% training set, 15% validation set, and 10% testing set for a fair comparison with the

[2] https://zenodo.org/record/2530277#.YWQxVElfiUk

previous work [1]. Then, we add the synthetically generated data to the organic training set until each class has at least 5,000 samples to augment those classes with low support. Finally, the training set contains organic data samples and synthetic data samples. In contrast, the validation and testing set only contains the organic data samples.

However, we noticed a lot of noises in some classes of the dataset. For instance, we found that more than 50% of the samples labeled *CheckedTextView* do not belong to that class. Some *ImageButton* were also labeled as *ImageView* and vice-versa. This may affect the model performance in certain classes. Thus, we construct a **standard dataset** containing 5,000 samples in each widget class to mitigate this problem. To construct the standard dataset, we first manually remove the irrelevant and mislabeled samples from each GUI widget class in the organic data. Next, we randomly select some relevant samples from the synthetic data and add them to those classes that do not contain up to 5000 cleaned samples. This standard dataset is used to investigate whether the data cleaning positively impacts the model performance. To evaluate the proposed approach, we split the standard dataset into 70% training set, 20% validation set, and 10% testing set.

Thus, in this study, we experiments on both the *original* and the *standard* datasets. Table 1 explains the statistics of the benchmark dataset.

4.3 Comparative Methods

This section explains the main comparative methods used in evaluating our proposed method.

ReDraw: ReDraw [1] is the latest approach that automates the transformation process of GUI mockup into code by enabling proper prototyping of GUIs through detection, classification, and assembly tasks. We consider only the classification task of the ReDraw in this paper. It employs a deep neural network to classify widgets into their peculiar domain types. ReDraw utilizes a hand-crafted architecture similar to AlexNet [16] but with three instead of five convolutional layers.

BOVW (Bag of Visual Words): BOVW [17] is utilized as a baseline technique to measure the performance of the ReDraw CNN classifier. BOVW is achieved as a baseline SVM for the image classification approach. This method extracts image features using the Speeds-Up Robust Feature algorithm, clusters related features together using K-means clustering, and uses an SVM trained on the resulting features.

ResNet50: ResNet50 is a ResNet [18] variant that includes 48 Convolution layers, one Max Pool layer, and one Average Pool layer. The pre-trained network can be load trained on more than 1m image samples from the ImageNet dataset. The network was trained to classify the samples into 1000 classes, including keyboard, car, pencil, and various animals. Accordingly, the pre-trained model has learned a variety of rich features for a variety of images.

To assess the performance of the proposed NASW, we compare the above comparative methods in RQ2. Additionally, to verify the efficiency of the NASW and the impact of the training dataset size on the method, we only compared ResNet50 in RQ1 and RQ3. The reason for that is, ResNet50 provides the source code, while the other two algorithms do not provide reproducible programs.

4.4 Metrics

The efficacy of the architecture is evaluated on the test set after the training. The intuitive approach to measuring performance accuracy is to compute the ratio between correctly classified samples and

Table 1 Dataset Statistics

			Original Data			Standard Data		
Widget Type	Organic (O)	Synthetic (S)	Train (O+S)	Valid (O)	Test (O)	Train	Valid	Test
Button	16,007	-	12,007	2,400	1,600	3,500	1,000	500
CheckBox	1,650	7,762	5,000	247	165	3,500	1,000	500
CheckedTextView	3,424	6,472	5,000	505	337	3,500	1,000	500
EditText	5,643	5,743	5,000	846	567	3,500	1,000	500
ImageButton	8,693	3,445	6,521	1,306	866	3,500	1,000	500
ImageView	53,324	-	39,983	7,996	5,345	3,500	1,000	500
NumberPicker	378	8,714	5,000	57	38	3,500	1,000	500
ProgressBar	406	17,722	5,000	60	39	3,500	1,000	500
RadioButton	1,293	8,025	5,000	194	129	3,500	1,000	500
RatingBar	219	8,835	5,000	33	22	3,500	1,000	500
SeekBar	405	8,694	5,000	61	40	3,500	1,000	500
Spinner	20	8,985	5,000	3	2	3,500	1,000	500
Switch	373	8,718	5,000	56	37	3,500	1,000	500
TextView	99,200	-	74,087	15,236	9,877	3,500	1,000	500
ToggleButton	265	8,796	5,000	40	26	3,500	1,000	500
Total	191,300	101,911	187,598	29,040	19,090	52,500	15,000	7,500

the entirety of samples in the testing set. However, when working with imbalanced datasets, this might produce deceptive results because models can perform well in classes with copious data while their performance in minority classes is concealed. We computed the F1-score in addition to the accuracy and precision to place each class on an equal footing when contributing to the model score, regardless of size. F1-score is the harmonic mean for precision and recall scores.

Accuracy: The ratio of the overall number of correct predicted widget class to the sum of all predictions:

$$Accuracy\ (Acc.) = \frac{TP + TN}{TP + TN + FP + FN} \tag{3}$$

Precision: The precision metric measures how exact the model is when some widget groups are detected. It is the ratio of positive observations against false positives within a specific widget class and is calculated as:

$$Precision\ (Pre.) = \frac{TP}{TP + FP} \tag{4}$$

Recall: Recall measures what proportion of a particular widget class was predicted correctly. It is calculated as the ratio of accurately predicted positive widget observations to all observations in the actual widget class:

$$Recall\ (Rec.) = \frac{TP}{TP + FN} \tag{5}$$

Where TP represents true positives or cases where the highest network widget prediction is correct, and FP represents false positives or cases where the highest network prediction is incorrect

F1-Score: The metric can be viewed as a weighted average of recall and precision, with 1 as the best and 0 the worst. It is used to determine the best tradeoff of precision and recall. The F1 score is calculated as:

$$F1 - score = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{6}$$

4.5 Parameters and Environment

The experiments were carried out on a PC equipped with a GTX 2080Ti single Nvidia GPU accelerator. The PC has an Intel i7-7700K CPU, 64 GB of RAM, and is running Ubuntu 18.04 LTS. The proposed approach and baseline were implemented in Python using the PyTorch ML framework. In the baseline method, we apply the advanced optimization algorithm SGD to discover the optimal network weights. Finally, due to memory constraints, we train the ResNet50 with a batch size of 64, but the proposed method permits up to 128 batches.

5 Experiment Results

5.1 RQ1: Time

Method: In response to RQ1, we search for an optimal architecture using a small sample of the GUI widgets dataset. The sizes of the GUI widget images vary across classes. We normalize the data by using channel mean and standard deviation for preprocessing. Also, we resize the images into 256 pixels then crop them to 224 pixels before feeding them to the network; this helps to reduce the architecture search and model training time and consistency. We run the architecture search five times for 50 epochs using different seed values. At the end of each search, we select the best performing candidate architecture based on the validation accuracy. Next, we choose the best architecture among the five best candidate architectures we selected at the end of each search. Then, we train the best architecture on the *standard dataset* for 50 epochs to generate the final classification model NASW_m. Additionally, we train ResNet50 architecture to illustrate the effectiveness of our NASW approach. Finally, we report the time taken to search and train the best architecture and the performance.

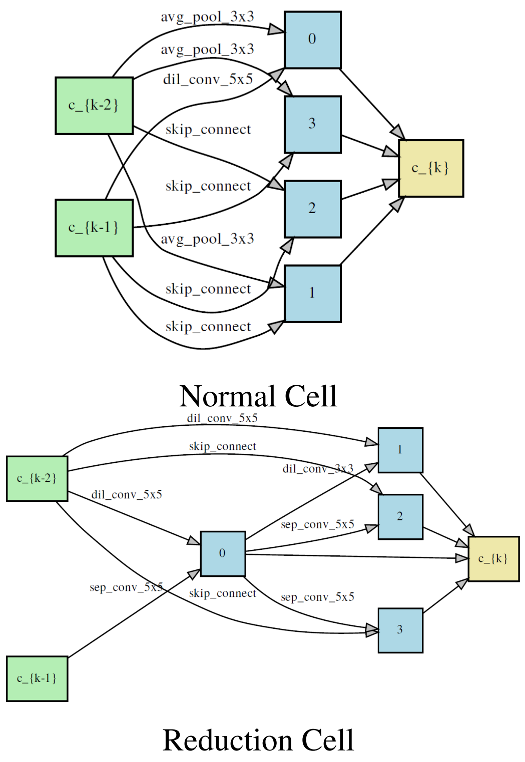

Fig. 7 Normal and reduction cells found on GUI widgets dataset

Result: Figure 7 shows the *normal cell* (cells that compute the feature map) and *reduction cell* (cells that reduced the feature map dimension) of the best architecture found on the widgets dataset. Also, Table 2 shows the time taken to search for an optimal architecture based on the GUI widgets dataset. As shown in the Table 2, each search takes less than an hour to obtain the best architecture based on the given data and the seed value. Additionally, the results show that the architecture obtained using the seed value as 0, is best performing architecture based on the validation accuracy. Also, this architecture has a smaller number of parameter size than the other generated architectures. In addition, Table 3 reveals the performance of the NASW_m model from the best NASW architecture and ResNet50 and the time taken to train both. The NASW_m model achieves a higher precision of 91.9% than ResNet50 with 90.3% on the standard dataset. This improvement shows our NASW approach's ability to learn better on the GUI data since the architecture is generated based on the GUI widgets dataset. Additionally, the results also reveal that training the proposed NASW is about 3x faster than ResNet50. It takes only 3 hours to train the NASW architecture on the *original dataset* while ResNet50 takes up to 10 hours. Even training on the *standard dataset*, NASW is about 2.5x faster than ResNet50. The training speed was due to the params size. The best generated NASW architecture has a params size of 2.47MB, while the baseline method has 89.79MB. Thus, our proposed NASW method trains faster than the comparative method.

Table 2 Architecture search time

Architecture	Seed	Accuracy	Time (Min)	Param.
NASW	0	**84.15**	**57.13**	**2.47 MB**
NASW1	1	82.49	57.31	3.93 MB
NASW2	2	83.93	55.27	3.85 MB
NASW3	3	83.51	54.16	3.49 MB
NASW4	4	83.61	55.50	3.67 MB

Table 3 Training time and performance

	Original Data		Standard Data	
Method	Time	Precision	Time	Precision
ResNet50	619 Mins	0.858	181 Mins	0.903
NASW_m	184 Mins	0.906	**73** Mins	**0.919**

Summary: Our proposed NASW approach takes less than an hour to obtain an optimal model architecture for widgets classification. This shows the effectiveness of the NASW method in automating the architecture engineering and thus, reduce the time-consuming task of designing the architecture manually. Besides, the generated architecture trains faster than the comparative method.

5.2 RQ2: Performance

Method: To answer RQ2; first, we train the NASW and ResNet50 on the *original dataset*. For a fair judgment with the prior work ReDraw [1], we apply the same data segmentation method on the *original dataset* to train the models. The training set consists of organic and synthetic data, while the validation and testing data are all randomly selected from the organic data. We train the searched NASW model architecture and the comparative method for 50 epochs. To assess the effectiveness of the trained model NASW_m, we compute the average precision and recall. Moreover, considering the data imbalance in the original dataset, we calculate the average F1-score and compare the results against the comparative methods. Note that the ReDraw code has not been made available to reproduce the result at the time of conducting this experiment. Thus, we utilized the results and confusion matrix reported by the authors Moran et al. [1] to compute the F1-scores and other metrics.

Result: Table 4 presents the performance comparison of our NASW_m model on the *original dataset* against the comparative methods. Our proposed NASW method achieves an average F1-score of 90% that outperforms the comparative methods. Although ReDraw achieves nearly the same precision, the proposed method achieves a better recall and F1-score. We calculated the F1-score to ensure that each class contributed equally to the model scores, despite the size.

Table 4 Performance comparison with the comparative methods on the original dataset

Method	Accuracy	Precision	Recall	F1-score
ReDraw	0.834	**0.913**	0.554	0.690
BOVW	0.646	0.569	0.528	0.546
ResNet50	0.862	0.858	0.862	0.854
NASW_m	**0.906**	0.906	**0.905**	**0.899**

Table 5 Impact of training data size

Method	Dataset Size	Precision	Improvement	Recall.	Improvement	F1-score	Improvement
	100%	0.858	-	0.862	-	0.854	-
ResNet50	75%	0.846	-	0.849	-	0.837	-
	50%	0.836	-	0.842	-	0.830	-
	25%	0.823	-	0.828	-	0.816	-
	100%	0.906	+5.59%	0.905	+4.99%	0.899	+5.27%
NASW_m	75%	0.869	+2.72%	0.873	+2.83%	0.866	+3.46%
	50%	0.849	+1.56%	0.854	+1.43%	0.845	+1.81%
	25%	0.819	-0.49%	0.824	-0.48%	0.814	-0.25%

Summary: The proposed NASW method outperforms the comparative methods achieving an F1-score of 90% on the *original dataset*. Because the *original dataset* is skewed toward certain classes, we calculated the F1-score to ensure that each class contributed equally to the model score, irrespective of size. The performance improvement is perhaps from the ability of the proposed method to learn more salient features from the GUI widgets dataset since the network architecture is generated based on the widgets dataset.

5.3 RQ3: Impact of training data size

Method: We set RQ3 to verify the impact of the training size on algorithms performance. First, we prepare the training data needed for the experiment. Four training data sizes are set, i.e., 100%, 75%, 50%, and 25% of the original dataset, as shown in Table 5. The samples in these data were chosen at random from the training partition in the *original dataset* to fulfill the ratio. Moreover, the validation set and testing set specified in Section 4.2 remain unchanged. Second, we train the NASW and ResNet50 with different sizes of the training set, respectively. Some of the algorithms mentioned above, i.e., ReDraw and BOVW, do not provide reproducible code or executable programs. We only compare our proposed method with ResNet50.

Result: Table 5 shows the performance of our proposed NASW and ResNet50 trained models on four training data sizes. Precision, recall, and F1-score are the three metrics that are compared. The results show 5.59%, 4.99%, and 5.27% relative improvement on three metrics against ResNet50 on the 100% training size. Besides, the results show a relative decrease in the performance as the training data decreases. The performance loss is understandable since, without enough training data, the deep learning model cannot efficiently learn the GUI widgets' important features. Meanwhile, we find that the models' relative performance is consistent across the data sizes. The NASW_m outperforms the others as the training data increases.

Summary: The evaluation result shows that the amount of training data has an impact on the performance of the models. This is because deep learning requires sufficient training data in order to learn salient features from the dataset. Nevertheless, the result shows that our NASW approach still outperforms the comparative method as the training size increases.

5.4 Discussion

5.4.1 Necessity of constructing the standard dataset

We constructed the *standard dataset* to mitigate the class imbalance and the noise in the *original wid-*

gets dataset as described in Section 4.2. The imbalanced class challenge is a deep learning phenomenon. This phenomenon happens when the total number of specific data classes is considerably greater than that of other data classes [19]. This problem is prevalent in many areas [20, 21], including image recognition and text categorization. Most standard statistical classification methods aim to reduce the total error rate on the training data by assuming a balanced distribution class or even misclassification losses. When challenged with skewed data, these classification approaches prioritize the majority class, negatively impacting minority classes [22]. Under-sampling or random over-sampling strategies are used to balance the class distribution. However, they do not take data cleaning into account [19]. Several researchers have carried out experiments to show that the overall imbalance ratio is not the main source of classification issues only; it is also related to other data distribution factors, such as the presence of noise [23]. Usually, when there is noise and class imbalance in a dataset, the performance of the minority class is masked. Thus, we built the *standard dataset* to examine the impact of the data cleaning and the balanced class distribution in our proposed method. We manually remove the irrelevant widgets from the original dataset classes to build the *standard dataset* with 5k samples in each class. Our experiment on the *standard dataset* illustrates the impact of the data cleaning in improving the approach NASW performance.

5.4.2 Widget types

As presented in Android Official Website[3)], GUI widgets are typically categorized into four types: information, collection, control, and hybrid. There are more than 100 widgets. Our benchmark dataset consists of only 15 widgets. Three of them belong to information type(*ImageView, ProgressBar, and TextView*), (*CheckedTextView*) belong to hybrid type , and the remaining ones are all control widgets, such as *Button, Switch*, and *ToggleButton*. Obviously, the effectiveness of our proposed method NASW can only be proved on the used benchmark dataset. In the future, we plan to label all widgets and verify the effectiveness of NASW.

5.4.3 Choice of technology

Our work is the first attempt of NAS technology in the task of widget classification, and has achieved a certain improvement compared to the human designed model architectures. As reported results in Section 5 answering to three RQs, the proposed method NASW not only completes the model training 3x faster than the comparative method on the same training data, but the generated model NASW_m achieves a 5% improvement on multiple metrics. This is mainly due to the use of the differentiable one-shot NAS [8, 9]. In fact, in recent years, NAS algorithm has made great progress and been applied in several fields [24, 25] (e.g., semantic image segmentation [26], object detection [27], etc.). In future work, we will try different types of NAS algorithms to further improve the performance of this task.

6 Related Work

In this section, we discuss some related work on GUI widgets tasks and image classification using NAS.

GUI Widgets classification: The design, testing, and implementation of a GUI are all important components of software engineering (SE). Many SE

[3)] https://developer.android.com/guide/topics/appwidgets/overview tasks, such as GUI automation and testing [28–30], advanced GUI interactions [31,32], search engines [33–35], and GUI code generation [1,36,37], rely on identifying GUI widgets in a GUI image. Detecting and classifying GUI widgets into their peculiar domain types is required for many of these tasks. A CNN architecture was used in previous work [1,3] to classify images of GUI widgets into their domain-specific classes.

Large-scale image identification and classification have benefited from the advancement of CNNs [16, 38]. Significant performance increases have been reported across many computer vision obstacles as DL continues to advance. The majority of these increases can be attributed to manually constructed model architecture [16, 18, 38]. From a large set of labeled training images, such as the ILSVRC dataset [39], these DL algorithms can automatically discover robust, prominent properties of image categories. ReDraw citeMoran2018, for example, uses a hand-crafted CNN to classify the GUI widgets.

Chen et al. [3] utilized image detection techniques to identify the GUI widgets from an image. The authors compared the state of art detection methods and proposed a better approach for the GUI widgets detection. In addition, they adopted the mature ResNet50 pre-trained model to classify the GUI widgets. However, the dataset we used in this work has unique characteristics different from the RICO [35] dataset used in the detection task. The RICO dataset contains GUI screenshots with more than one widget each. However, the GUI widgets dataset contains only one widget as a sample in the dataset.

Nevertheless, while these manually created network architectures have produced promising results in various domains, designing an outstanding architecture for a specific task is time-consuming.

NAS Based classification: In recent years, the NAS technique has gotten much attention. The goal of NAS is to develop automated ways to design neural architectures that can replace the handcrafted ones. Many approaches, including evolutionary algorithms, RL, and one-shot, have achieved optimal performance.

Evolutionary algorithms apply a group of individuals to produce offspring that improve in performance over time. Individuals, like natural organisms, are determined by their genes and genomes. Each gene contains details about the problem being solved. Individuals can reproduce through crossover, resulting in offspring having genes from both parents, according to Kyriakides et al. [40]. In the NAS scenario, genes frequently carry information on each network layer and the links between them. The crossover and mutation operators are the most distinctive amongst their implementations. For example, Miikkulainen et al. [41] use genetic algorithms to evolve exterior skeleton designs and cell types. They employ mutation to add or remove connections and layers and change the parameters of a layer. The poorest performers are eliminated at each generation, while the rest of the population reproduces and is replaced by its offspring.

Furthermore, the RL approach has been successfully used in the design of architectures in NAS. The fundamental distinctions between them are in the definitions of action and state. Their outcome is often calculated depending on the estimated performance of the produced network. The action space, which consists of layer parameter selection, layer type, and inter-layer connections, is dictated by and is generally equivalent to the search space. Then, the state is the current generated architecture. The study by Zoph and Le et al. [7] was one of the first to use the RL approach and was the inspiration for the NAS acronym. Nonetheless, the computational expenses of the evolutionary-based and RL-based techniques are still expensive.

Therefore, some scholars proposed lowering the expenses of evaluating each searched option to speed up the architecture search. One of the early initiatives, Cai et al. [42], was to share weights across searched and new networks. These methods were subsequently expanded into a more sophisticated architectural search framework known as a one-shot. For example, Liu et al. [8] employed a one-shot approach, in which a super-network containing all potential operations was trained once, and then numerous sub-networks were sampled exponentially from it. This technique was implemented in a framework called DARTS. On the evaluation of the ImageNet dataset, DARTS outperformed the existing image classification architectures.

Our work is inspired by recent performances of the NAS-based approaches in image classification. This is a new research area that has recently received much attention. Unlike other NAS methods [7–9], which mainly experiments on the CIFAR10 and ImageNet datasets, we conduct experiments on the GUI widgets dataset to support software engineering tasks such as GUI code generation and GUI Testing. In addition, our algorithm is built based on the one-shot strategy [8, 9] utilizing the cell-based search space.

7 Threat to validity

7.1 Construct Validity

The operationalization of the experimental artifacts poses a threat to construct validity. Our comparative method, ReDraw [1], poses a threat to construct validity. The source code for reproducing the result was not made accessible at the time of writing this paper. As a result, we compare our findings to those reported in the original study. Furthermore, for a fair comparison with the existing technique, we train and evaluate our proposed model using the same data segmentation stated in the ReDraw study.

7.2 External Validity

The generalization of the results poses a threat to external validity. We only experiment on the data extracted from Android Apps, which may be a risk to the external validity of the results and findings. For example, the benchmark dataset used in this paper contains only 15 classes of android GUI widgets. However, we believe these widget categories are common and can easily be extended to other domains such as iOS and Windows apps. In addition, the proposed NASW architecture is searched only on the 15 widely used GUI widget classes. However, we believe the architecture will perform well even if the classes are extended to include other new GUI widgets classes.

8 Conclusion

This paper presents a NAS-based approach for automatically generating network architecture for GUI widgets classification. Our experiment results reveal that (i) the proposed approach NASW can obtain the best network architecture for widgets classification in less than an hour, and (ii) the proposed NASW outperforms the comparative methods which utilize manually designed network architectures. In the future, it is necessary to construct a larger dataset with more than 15 GUI widgets to generalize the classification task better. Additionally, we plan to extend the work to support the GUI testing and GUI code generation tasks.

Acknowledgement

The National Natural Science Foundation of China (Grant 61972359) and the Zhejiang Provincial Natural Science Foundation of China (Grant LY19F020052) supported this research.

References

1. Moran K, Bernal-Cardenas C, Curcio M, Bonett R, Poshyvanyk D. Machine Learning-Based Prototyping of Graphical User Interfaces for Mobile Apps. IEEE Transactions on Software Engineering, 2020, 46(2): 196–221
2. Said K S, Nie L, Ajibode A A, Zhou X. GUI testing for mobile applications: objectives, approaches and challenges. In: 12th Asia-Pacific Symposium on Internetware. 2020, 51–60
3. Chen J, Xie M, Xing Z, Chen C, Xu X, Zhu L, Li G. Object detection for graphical user interface: Old fashioned or deep learning or a combination? In: ESEC/FSE 2020 - Proceedings of the 28th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2020, 1202–1214
4. Chen C, Feng S, Xing Z, Liu L, Zhao S, Wang J. Gallery D.C.: Design search and knowledge discovery through auto-created GUI component gallery. Proceedings of the ACM on Human-Computer Interaction, 2019, 3(CSCW): 1–22
5. Dubrovina A, Kisilev P, Freedman D, Schein S, Bergman R. Efficient and robust image descriptor for GUI object classification. In: Proceedings - International Conference on Pattern Recognition. 2012, 3594–3597
6. Nakai K, Matsubara T, Uehara K. Att-DARTS: Differentiable Neural Architecture Search for Attention. In: Proceedings of the International Joint Conference on Neural Networks. 2020, 1–8
7. Zoph B, Le Q V. Neural architecture search with reinforcement learning. 5th International Conference on Learning Representations, ICLR 2017 - Conference Track Proceedings, 2017
8. Liu H, Simonyan K, Yang Y. DARTS: Differentiable architecture search. 7th International Conference on Learning Representations, ICLR 2019, 2019
9. Xu Y, Xie L, Zhang X, Chen X, Qi G J, Tian Q, Xiong H. PC-DARTS: Partial Channel Connections for Memory-Efficient Architecture Search. arXiv preprint arXiv:1907.05737, 2019
10. Sherrick S Q. Introduction to graphical user interfaces and their use by citis. Technical report, National Inst. of Standards and Technology (CSL), Gaithersburg, MD (United . . . , 1992
11. Chen H, Zhang Z, Zhao C, Liu J, Yin W, Li Y, Wang F, Li C, Lin Z. Depth Classification of Defects Based on Neural Architecture Search. IEEE Access, 2021, 9: 73424–73432
12. Jie R, Gao J. Differentiable Neural Architecture Search for High-Dimensional Time Series Forecasting. IEEE Access, 2021, 9: 20922–20932
13. Mo H, Custode L L, Iacca G. Evolutionary neural architecture search for remaining useful life prediction. Appl. Soft Comput., 2021, 108: 107474
14. Du Q, Xu N, Li Y, Xiao T, Zhu J. Topology-Sensitive Neural Architecture Search for Language Modeling. IEEE Access, 2021, 9: 107416–107423
15. Chen Y, Yang T, Zhang X, Meng G, Xiao X, Sun J. DetNAS: Backbone search for object detection. In: Advances in Neural Information Processing Systems. 2019, 6642–6652
16. Krizhevsky A, Sutskever I, Hinton G E. ImageNet classification with deep convolutional neural networks. Communications of the ACM, 2017, 60(6): 84–90
17. Venegas-Barrera C S, Manjarrez J. Visual Categorization with Bags of Keypoints. In: Revista Mexicana de Biodiversidad. 2011, 179–191
18. He K, Zhang X, Ren S, Sun J. Deep residual learning for image recognition. In: Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition. 2016, 770–778
19. Guan H, Zhang Y, Xian M, Cheng H D, Tang X. Wenn for individualized cleaning in imbalanced data. In: 2016 23rd International Conference on Pattern Recognition (ICPR). 2016, 456–461
20. Yu H, Ni J. An improved ensemble learning method for classifying high-dimensional and imbalanced biomedicine data. IEEE/ACM transactions on computational biology and bioinformatics, 2014, 11(4): 657–666
21. Yang Q, Wu X. 10 challenging problems in data mining research. International Journal of Information Technology & Decision Making, 2006, 5(04): 597–604
22. Fernández A, García S, Luengo J, Bernadó-Mansilla E, Herrera F. Genetics-based machine learning for rule induction: state of the art, taxonomy, and comparative study. IEEE Transactions on Evolutionary Computation, 2010, 14(6): 913–941
23. Napierała K, Stefanowski J, Wilk S. Learning from imbalanced data in presence of noisy and borderline examples. In: International conference on rough sets and

current trends in computing. 2010, 158–167
24. Elsken T, Metzen J H, Hutter F. Neural architecture search: A survey. The Journal of Machine Learning Research, 2019, 20(1): 1997–2017
25. Ren P, Xiao Y, Chang X, Huang P Y, Li Z, Chen X, Wang X. A comprehensive survey of neural architecture search: Challenges and solutions. ACM Computing Surveys (CSUR), 2021, 54(4): 1–34
26. Liu C, Chen L C, Schroff F, Adam H, Hua W, Yuille A L, Fei-Fei L. Auto-deeplab: Hierarchical neural architecture search for semantic image segmentation. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. 2019, 82–92
27. Ghiasi G, Lin T Y, Le Q V. Nas-fpn: Learning scalable feature pyramid architecture for object detection. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. 2019, 7036–7045
28. Cardenas C B, Cooper N, Moran K, Chaparro O, Marcus A, Poshyvanyk D. Translating video recordings of mobile app usages into replayable scenarios. Proceedings - International Conference on Software Engineering, 2020, 309–321
29. Yeh T, Chang T H, Miller R C. Sikuli: Using GUI screenshots for search and automation. In: UIST 2009 - Proceedings of the 22nd Annual ACM Symposium on User Interface Software and Technology. 2009, 183–192
30. Qian J, Shang Z, Yan S, Wang Y, Chen L. RoScript: A visual script driven truly non-intrusive robotic testing system for touch screen applications. In: Proceedings - International Conference on Software Engineering. 2020, 297–308
31. Banovic N, Grossman T, Matejka J, Fitzmaurice G. Waken: Reverse engineering usage information and interface structure from software videos. In: UIST'12 - Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology. 2012, 83–92
32. Dixon M, Fogarty J. Prefab: Implementing advanced behaviors using pixel-based reverse engineering of interface structure. In: Conference on Human Factors in Computing Systems - Proceedings. 2010, 1525–1534
33. Bernal-Cardenas C, Moran K, Tufano M, Liu Z, Nan L, Shi Z, Poshyvanyk D. Guigle: A GUI search engine for android apps. In: Proceedings - 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion, ICSE-Companion 2019. 2019, 71–74
34. Nie L, Jiang H, Ren Z, Sun Z, Li X. Query expansion based on crowd knowledge for code search. IEEE Transactions on Services Computing, 2016, 9(5): 771–783
35. Deka B, Huang Z, Franzen C, Hibschman J, Afergan D, Li Y, Nichols J, Kumar R. Rico: A mobile app dataset for building data-driven design applications. In: UIST 2017 - Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology. 2017, 845–854
36. Nguyen T A, Csallner C. Reverse engineering mobile application user interfaces with REMAUI. In: Proceedings - 2015 30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015. 2016, 248–259
37. Chen C, Su T, Meng G, Xing Z, Liu Y. From UI design image to GUI skeleton: A neural machine translator to bootstrap mobile GUI implementation. In: Proceedings - International Conference on Software Engineering. 2018, 665–676
38. Zeiler M D, Fergus R. Visualizing and understanding convolutional networks. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). 2014, 818–833
39. Simonyan K, Zisserman A. Very deep convolutional networks for large-scale image recognition. 3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings, 2015
40. Kyriakides G, Margaritis K. An introduction to neural architecture search for convolutional networks. arXiv preprint arXiv:2005.11074, 2020
41. Miikkulainen R, Liang J, Meyerson E, Rawal A, Fink D, Francon O, Raju B, Shahrzad H, Navruzyan A, Duffy N. Evolving deep neural networks. In: Artificial intelligence in the age of neural networks and brain computing, 293–312. Elsevier, 2019
42. Cai H, Chen T, Zhang W, Yu Y, Wang J. Efficient architecture search by network transformation. In: Proceedings of the AAAI Conference on Artificial Intelligence. 2018

Kabir Sulaiman SAID graduated from Kano University of Science and Technology in Wudil, Nigeria, with a Bachelor of Science in Computer Science in 2016. He is currently a master's student at Zhejiang Sci-Tech University in China, studying computer science and technology. His research interests include software testing and intelligent software development.

Liming NIE is a lecturer at Zhejiang Sci-Tech University, China. From June 2021, he works at Nanyang Technological University as a senior research fellow. He received a Ph.D. degree from the Dalian University of Technology in 2017. His current research interests include Intelligent software development and big code data analysis.

Yuangchang LIN received the B.E. degree in Biomedical Engineering from Shanghai University of Medicine & Health Sciences, Shanghai, China, in 2020. She is currently pursuing an M.S degree in computer science and technology with the Zhejiang Sci-Tech University, China. Her research interests include computer network and software development and testing.

Yaowen Zheng is a research fellow at Nanyang Technological University. His research interests are related to system security. In particular, his research mainly focuses on vulnerability analysis techniques such as fuzzing, dynamic emulation for IoT firmware. He received the Ph.D. in cyberspace security from University of Chinese Academy of Sciences in 2020.

Zuohua DING (Associate Member, IEEE) earned an M.S. in computer science and a Ph.D. in mathematics from the University of South Florida in Tampa, FL, USA, in 1996 and 1998, respectively. He is currently a Professor and the Director of Zhejiang Sci-Tech University's Laboratory of Intelligent Computing and Software Engineering in Hangzhou, China. He has written or co-authored more than 70 articles. System modeling, software reliability prediction, intelligent software systems, and service robots are among his current research interests.