# Deeply Reinforcing Android GUI Testing with Deep Reinforcement Learning


Yuanhong Lan
State Key Lab for Novel Software Technology, Nanjing University
Nanjing, China
yhlan@smail.nju.edu.cn

Yifei Lu
State Key Lab for Novel Software Technology, Nanjing University
Nanjing, China
lyf@smail.nju.edu.cn

Zhong Li
State Key Lab for Novel Software Technology, Nanjing University
Nanjing, China
mg1733033@smail.nju.edu.cn

Minxue Pan*
State Key Lab for Novel Software Technology, Nanjing University
Nanjing, China
mxp@nju.edu.cn

Wenhua Yang
College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics
Nanjing, China
ywh@nuaa.edu.cn

Tian Zhang
State Key Lab for Novel Software Technology, Nanjing University
Nanjing, China
ztluck@nju.edu.cn

Xuandong Li
State Key Lab for Novel Software Technology, Nanjing University
Nanjing, China
lxd@nju.edu.cn



## ABSTRACT

As the scale and complexity of Android applications continue to grow in response to increasing market and user demands, quality assurance challenges become more significant. While previous studies have demonstrated the superiority of Reinforcement Learning (RL) in Android GUI testing, its effectiveness remains limited, particularly in large, complex apps. This limitation arises from the ineffectiveness of Tabular RL in learning the knowledge within the large state-action space of the App Under Test (AUT) and from the suboptimal utilization of the acquired knowledge when employing more advanced RL techniques. To address such limitations, this paper presents DQT, a novel automated Android GUI testing approach based on deep reinforcement learning. DQT preserves widgets' structural and semantic information with graph embedding techniques, building a robust foundation for identifying similar states or actions and distinguishing different ones. Moreover, a specially designed Deep Q-Network (DQN) effectively guides curiosity-driven exploration by learning testing knowledge from runtime interactions with the AUT and sharing it across states or actions. Experiments conducted on 30 diverse open-source apps demonstrate that DQT outperforms existing state-of-the-art testing approaches in both code coverage and fault detection, particularly for large, complex apps. The faults detected by DQT have been reproduced and reported to developers; so far, 21 of the reported issues have been explicitly confirmed, and 14 have been fixed.



*Corresponding author.



Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.



*ICSE '24, April 14–20, 2024, Lisbon, Portugal*



© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.



ACM ISBN 979-8-4007-0217-4/24/04…$15.00
https://doi.org/10.1145/3597503.3623344



## CCS CONCEPTS

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

## KEYWORDS

Android testing, deep reinforcement learning, graph embedding

**ACM Reference Format:**
Yuanhong Lan, Yifei Lu, Zhong Li, Minxue Pan, Wenhua Yang, Tian Zhang, and Xuandong Li. 2024. Deeply Reinforcing Android GUI Testing with Deep Reinforcement Learning. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597503.3623344


## 1 INTRODUCTION

In recent years, rapid advancements in mobile technology have led to the widespread presence of mobile applications (hereafter called apps), playing an indispensable role in our daily lives. Statistical data [1] reveals that the Google Play store [19] offers over 3.8 million Android apps, with nearly 3,000 new apps added daily. As the last test barrier [24] to ensure app quality, Android GUI testing has witnessed tremendous development in the last decade, with various promising approaches proposed showing encouraging results. However, as market demands and user experience expectations increase, mobile apps are expanding rapidly in scale and complexity. Like *Signal*, a popular open-source communication app with over 1,200 releases on GitHub, has approximately quadrupled its Executable Lines Of Code (ELOC) over the past three years. The augmenting complexity of such apps results in the presence of numerous hard-to-reach states and difficult-to-trigger events, posing a significant challenge for Android GUI testing.

Existing approaches employ various strategies for Android GUI testing, but their effectiveness in addressing the challenge remains limited. Random strategies [26, 43] generate pseudo-random events but struggle to navigate complex transitions without adequate guidance. Model-based approaches [30, 41, 61], which build a model of

the App Under Test (AUT) through static or dynamic analysis, encounter challenges in constructing precise models or handling non-deterministic transitions [54]. Systematic strategies [15, 18, 45], using techniques like symbolic execution or evolutionary algorithms for targeted purposes (e.g., code coverage), suffer from scalability constraints due to state or path explosion [48, 61]. Supervised learning strategies, which rely on pre-existing data to train a model for test generation [34, 37, 40, 42] or action validity prediction [14], exhibit limitations when facing unfamiliar apps or action selection.

Reinforcement Learning (RL) has recently achieved remarkable results [12, 36, 54, 58, 66]. RL-based strategies do not require constructing an explicit model of the AUT or pre-training a testing guidance model. Instead, learning from runtime testing experiences, they dynamically optimize testing policies and leverage testing knowledge for further exploration. The testing knowledge is generally related to identifying key actions that can potentially lead to more untested states and transitions. Typically, this is associated with Q-values that indicate actions' importance. Moreover, these strategies are flexible and adaptable, effectively testing various apps and handling dynamic changes of the AUT, like non-deterministic transitions. Additionally, their lightweight black-box advantage enables testing apps without source code reference.

However, facing increasingly complex state-action spaces of Android apps, existing RL-based strategies still suffer from several limitations. A significant challenge is the insufficient sharing of testing knowledge between states or actions, leading to inefficient test cases, unnecessary repetitions, and compromised testing effectiveness. Specifically, Tabular RL-based strategies [36, 46, 54], although able to learn from testing experiences and identify key actions, are limited by updating individual state-action pairs in a table. The learning for each action within the same state is treated separately, and every time a new state is encountered, the learning process begins anew. This hinders the sharing of testing knowledge across similar states or actions. To address this limitation, Deep RL [12, 14, 58, 66] has been adopted. However, the benefits of continuous state-action spaces and the potential knowledge-sharing capabilities of deep networks remain largely unexplored. Currently, one-hot or coarse encodings are commonly adopted, with widgets' structural and semantic information disregarded, raising difficulty in identifying similar states or actions. Besides, conventional networks struggle with highly complex state-action spaces of Android apps, leading to underfitting and inaccurate model predictions. Additionally, guided by coarse-grained reward functions, the model's perception of the environment and grasp of details remain limited, resulting in inaccurate evaluations.

To illustrate the importance of knowledge sharing, Figure 1 presents a motivating example from *Signal*, a large, complex app comprising over 84 activities and hundreds of pages. The figure displays two pages with "More Options" buttons (widgets in red boxes) of similar functionalities. Upon first entering the state $s_\alpha$, a default action probability distribution is adopted, indicating the statistical likelihood of selecting each action of $s_\alpha$. This can be uniform or other distribution, depending on preset configurations or available prior knowledge. During testing, the action distribution of $s_\alpha$ is continually updated, raising the probability of clicking the "More Options" button (probabilities in red), as clicking this button can navigate to multiple other pages, making it valuable for RL-based

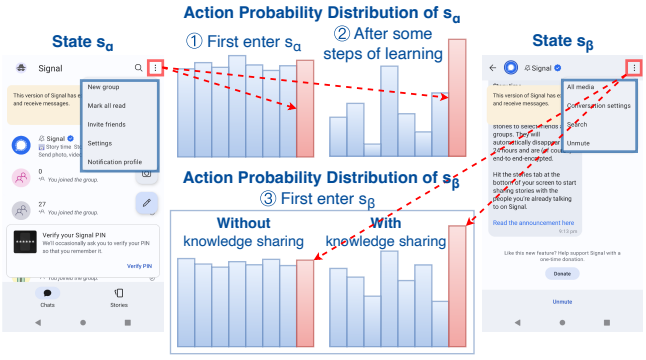


**Figure 1: A Motivating Example.**

approaches that prioritize key actions leading to numerous paths of other app functionalities. However, when encountering a new state $s_\beta$—which also contains a "More Options" widget—existing RL-based approaches typically adopt a preset distribution for actions in new states, requiring the policy to learn from scratch once again. This can negatively reduce testing effectiveness, particularly for large, complex apps with numerous states and actions.

To address the aforementioned challenges and enhance testing efficiency for large, complex Android apps, we believe the key lies in effectively learning and sharing testing knowledge across similar states or actions during testing. With this objective in mind, we introduce a novel approach, DQT (Deep Q-network Testing), based on Deep RL, which excels at handling a multitude of states and actions with various similarities and distinctions. However, transforming the mobile testing problem into a Deep RL problem is a non-trivial task, particularly when considering the goal of knowledge sharing, which necessitates adequate encoding of states and actions, an optimal network architecture for learning within a complex state-action space, and an effective reward function that guides the testing process to explore more states. To address these requirements, we meticulously design DQT to incorporate state-of-the-art graph embedding techniques [63, 72], which preserve widgets' semantic and structural information, enabling the identification of both similar states and actions. We collect over 100,000 samples from a diverse set of apps to train a robust Graph Neural Network (GNN) for graph embedding. Besides, DQT utilizes a specially designed DQN with carefully selected design enhancements for GUI testing. This model is expressive enough to handle complex state-action spaces while maintaining lightweight. Moreover, it leverages the concept of curiosity-driven exploration and employs a fine-grained dynamic reward function crucial for learning significative Q-values to provide effective guidance in exploring the app. It is the first reward function for GUI testing that considers the curiosity of activity, state, transition, and testing time simultaneously. We implemented DQT as a testing tool and conducted experiments on a benchmark of 30 open-source apps with an average ELOC of approximately 30,000. The results demonstrate that our approach outperforms existing state-of-the-art Android GUI testing approaches in both code coverage and fault detection. DQT has found 164 unique faults among the 30 apps, with over 5 unique faults discovered per app. So far, 21 of the faults have been explicitly

confirmed by developers, and 14 have been fixed. Moreover, 15 of the 21 confirmed faults are found by DQT for the first time, which demonstrates the effectiveness of DQT in revealing elusive faults.

In summary, we make the following main contributions:

- We propose employing deep reinforcement learning with graph embedding to enable effective testing knowledge sharing across states or actions, thereby improving mobile testing.
- We introduce the DQT approach, which utilizes graph embedding for state and action representation, a specially designed deep Q-network architecture to exploit app knowledge, and a fine-grained dynamic reward function to guide the testing process.
- We conducted extensive experiments on a benchmark of 30 diverse open-source apps with an average ELOC of approximately 30,000. The results demonstrate the superiority of our approach in terms of both code coverage and fault detection. Our tool and experimental data are publicly available[1] for future research.

# 2 BACKGROUND

## 2.1 Reinforcement Learning and GUI Testing

Reinforcement Learning (RL) [64] focuses on leading an *agent* learning how to interact with the *environment* to maximize a numerical reward signal. As depicted in Figure 2, this interaction can be formalized as a Markov Decision Process (MDP), which is a 4-tuple $< \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R} >$. At each time step $t = 0, 1, 2...$, the *agent* takes an action $a_t \in \mathcal{A}$ based on the current state $s_t \in \mathcal{S}$. Then the *environment* changes according to the transition function $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S}$ and reaches state $s_{t+1} \in \mathcal{S}$ with a step reward $r_t$ evaluated by the reward function $\mathcal{R}$. By analogy, GUI testing can also be modeled this way. A *testing tool* generates an event $a_t$ for the current GUI $s_t$. The *AUT* reacts, reaches the updated GUI $s_{t+1}$, and returns a step reward $r_t$ determined by a pre-defined reward function $\mathcal{R}$.

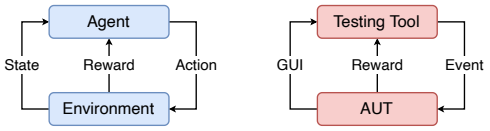


**Figure 2: Markov Decision Process.**

## 2.2 Deep Q-Network

Deep RL [64] employs deep neural networks to learn function approximation, which alleviates the potential intractability and inefficiency of learning for each state and action pair independently in large state and action spaces, as seen in Tabular RL.

Q-learning [69] is a model-free RL algorithm based on Q-function $Q^\pi(s_t, a_t) = \mathbb{E}[G_t|s_t, a_t, \pi]$, indicating the expected discounted cumulative reward $G_t = \sum_{k=t}^{\infty} \gamma^{k-t} r_k$ after taking action $a_t$ at state $s_t$ with the policy $\pi$. Here, $\gamma \in [0, 1]$ is a discount rate typically set to less than 1 for model convergence. DQN [51, 52] is a deep RL algorithm that combines the strengths of both deep networks and Q-learning. Based on the Bellman equation and the target network $\pi^-$ updated from the online network $\pi$ every several steps [52], the optimization target of DQN can be represented as:

$$Q^\pi(s_t, a_t) \leftarrow r_t + \gamma \max_{a'} Q^{\pi^-}(s_{t+1}, a'), \tag{1}$$

where a mean square error is usually adopted as the loss function.

Many improvements [8, 17, 33, 50, 60, 65, 68] have been proposed to enhance DQN further. In designing our DQN for effective mobile testing, we refer to Double Deep Q-Network (DDQN) [65], Dueling DQN [68] and prioritized experience replay [60]. DDQN [65] mitigates the issue of overestimation by delegating action selection responsibility from the target network $\pi^-$ to the online network $\pi$, which has been proved to attain a better evaluation of Q-values:

$$Q^\pi(s_t, a_t) \leftarrow r_t + \gamma Q^{\pi^-}(s_{t+1}, \underset{a'}{argmax}\, Q^\pi(s_{t+1}, a')). \tag{2}$$

The Dueling DQN [68] improves the evaluation of Q-values based on the refinement of network architecture. As depicted in Equation 3, Q-values are now computed by two parts, state values evaluated by state function $V$ and normalized action values evaluated by action function $A$ along with a mean normalization term that ensures the sum of all actions under a state is always zero.

$$Q^\pi(s, a) = V^{\theta,\beta}(s) + A^{\theta,\alpha}(s, a) - \frac{1}{|\mathcal{A}|} \sum_{a'} A^{\theta,\alpha}(s, a'). \tag{3}$$

The network architecture is segmented into three sections accordingly: a shared feature extraction network $\theta$ for feature extraction, a state estimator network $\beta$ for state evaluation, and an action estimator network $\alpha$ for action evaluation. The network $\theta$ passes the extracted features to both the networks $\beta$ and $\alpha$ simultaneously. This bifurcated evaluation of states and actions allows for effective recognition and discrimination between actions, particularly when the actions are numerous and similar. Based on experience replay [52], prioritized experience replay [60] identifies significant experiences, assigns them with higher priorities, and replays them more frequently, achieving more efficient learning.

# 3 APPROACH

## 3.1 Problem Formulation

In this work, we focus on automated Android GUI testing. As discussed in Section 2.1, this problem can be formalized as an MDP with a 4-tuple $< \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R} >$ [54, 58]. ① A state $s \in \mathcal{S}$ represents the current status of the AUT. Since we focus on black-box GUI testing, states are defined by AUT's GUI status. Specifically, we adopt the GUI hierarchy to describe the current GUI status, which has proven effective [30, 40, 54, 58, 61]. While presenting that finer state and comparison granularity benefit testing [5, 30, 54, 62], we model the XML-style hierarchy tree in fine granularity as a widget graph and retain widgets' attributes as they contain important semantic and structural information. Then, with advanced graph embedding techniques, we achieve an informative representation of the current state (see Section 3.3). ② An action $a \in \mathcal{A}$ is an executable event of a state. This paper does not differentiate events and actions, as they are interchangeable in our context. DQT extracts executable events of a state by analyzing the widgets in the GUI hierarchy and their attributes, such as *clickable*, *long-clickable*, *checkable*, and *scrollable*. To explore a larger state space, we also incorporate several system events as a supplement, e.g., *rotating screen*, *changing volume*, and *pressing home*. All the actions will be

[1] https://github.com/Yuanhong-Lan/DQT

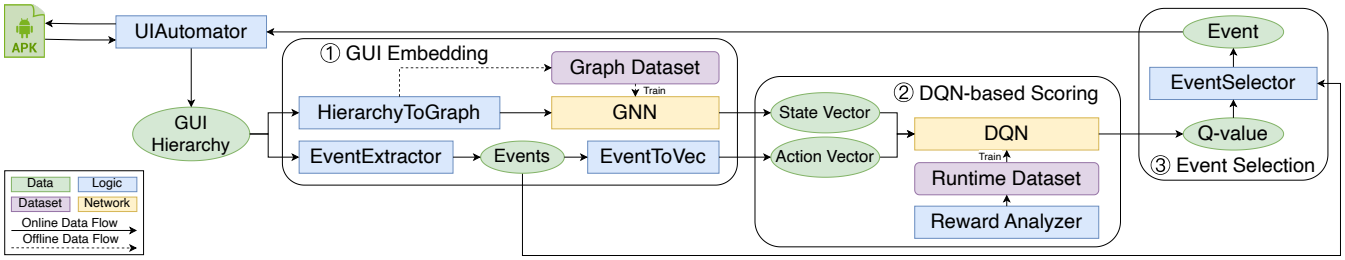


**Figure 3: The Workflow of DQT.**

further encoded into action vectors (see Section 3.3). ③ After an action is performed, the app transitions to a target state determined by the transition function $\mathcal{P}$. Dependent on the app, function $\mathcal{P}$ is implicitly embedded within the AUT. Note that non-deterministic transitions often arise due to changes in app settings, which remain an open issue. DQT employs robust deep networks and the Bellman equation to mitigate this challenge. ④ A reward function $\mathcal{R}$ determines the reward gained after taking an action. Vastly influencing the RL model, a well-defined reward function is critical for test effectiveness. Based on curiosity-driven exploration, DQT resorts to reward shaping [70], designing a fine-grained dynamic reward function, which will be further explained in Section 3.4.

## 3.2 An Overview of DQT

Figure 3 provides an overview of DQT. Given an AUT, the entire testing workflow is organized as a continuous loop until the time limit expires. In each loop iteration, DQT dumps the current GUI hierarchy via UIAutomator [27, 32]. Next, a pipeline of three components processes the GUI hierarchy and selects a valuable test action. Component ①: The *GUI Embedding* component converts the XML-style GUI hierarchy tree into a widget graph and employs graph embedding to transform it into a state vector via a GNN. Simultaneously, executable GUI events are extracted and encoded into action vectors. Component ②: With the informative vector representations, the *DQN-based Scoring* component employs a specially designed DQN to score the actions. It evaluates each action of the current state with Q-values, identifying key actions that lead to more app functionalities. A reward analyzer is leveraged to score each interaction via a reward function that implements the notion of curiosity-driven exploration. This DQN is trained on a batch of runtime data sampled from a buffer of testing interactions (the source and target Activities, the States, the corresponding Actions causing the transitions, and the gained step Rewards). Through suitable embeddings and the utilization of DQN, the updating of Q-values for comparable states or actions simultaneously is made possible. This facilitates the sharing of testing knowledge and enhances the recognition of significant or futile actions in different states. Component ③: The *Event Selection* component chooses a GUI event, system event, or another random event according to given probabilities. When picking a GUI event, it prioritizes the one with the highest Q-value. Finally, the selected event is executed via UIAutomator, and this test interaction is stored in the buffer.

The above workflow enables DQT to efficiently explore the targeted state-action space and make well-informed decisions about which actions to take, ultimately improving the effectiveness of the Android GUI testing process. In the following sections, we will further elaborate on these three main components of DQT.

## 3.3 GUI Embedding

Although utilizing GUI screenshots as input for DQN is an alternative, it neglects informative widget properties (such as *clickable* indicating whether a widget is clickable, and *class* revealing widget type) that imply crucial structural and semantic information. Consequently, following prior works [30, 54, 58], we adopt GUI hierarchies. However, directly incorporating raw GUI hierarchies and extracted events into DQN is infeasible; thus, we employ *GUI embedding* to generate vector representations of actions and states.

**Action Embedding.** Actions are derived from the properties of widgets, e.g., a widget that is *clickable* and *long-clickable* signifies two actions: click and long-click. We represent an action with two parts: the node vector of the widget (widgets serve as nodes in the GUI hierarchy) upon which the action is performed and an action type one-hot encoding. The latter is straightforward, where 9 types have been defined: *click*, *long-click*, *edit* via number, *edit* via text, *scroll* in four different directions, and *back*. For node vectors, we screen out essential widget attributes and categorize them into four groups, applying specific encoding strategies for each:

- *Bounds.* The *bounds* attribute is crucial as it indicates the position of the widget. It is especially meaningful when a specific widget is placed in a significant location. The raw *bounds* attribute comprises four integers representing a widget's top-left and bottom-right coordinates. We further enhance the location information by adding a central point, and all coordinates are normalized by the screen's width and height.
- *Boolean Attributes.* Boolean attributes describe event executability or widgets' current status. Seven important boolean attributes are considered, including four for event executability (i.e., *clickable*, *long-clickable*, *checkable*, and *scrollable*) and three for widget status (i.e., *enable*, *checked*, and *password*). All of them are conventionally mapped into 1 (true) and 0 (false).
- *Content-Free Text Attributes.* Text attributes play an indispensable role in identifying similar widgets. The value of attributes like *resource-id*, a unique identifier for the widget, is less important, while attributes like *text* imply a wealth of semantic information and need more consideration. Thus, instead of focusing solely on whether two strings are identical [54, 58], we categorize text attributes into two groups: *Content-Free Text Attributes*

and *Content-Related Text Attributes*. Attributes *class* and *resource-id* are classified as *Content-Free Text Attributes*. They are hashed into 8-digit numbers and normalized by their digit scale. Note that as this kind of encoding only cares about identity, it avoids the disturbance from some irrelevant information to semantics, e.g., a package name as a prefix of the *resource-id* values.

- *Content-Related Text Attributes*. Regarding *Content-Related Text Attributes text* and *content-desc*, we utilize a renowned multilingual NLP model, *paraphrase-multilingual-MiniLM-L12-v2* trained by Sentence Transformers [57], to capture their semantics. Note that many Android apps typically support multiple languages and may have their attributes manually written or machine-generated. This powerful NLP model is capable of comprehending multi-languages and extracting key semantics of the targeted text, thus breaking the multilingual and source-heterogeneous barriers. Since these vectors are lengthy, we further compress them through average pooling, where a lower compression rate is set for the *text* attribute as it generally contains richer and more pivotal semantic information. Ultimately, *text* and *content-desc* are compressed to 32 and 16 dimensions, respectively.

The vectors corresponding to different attributes are concatenated, resulting in a 77-dimensional node vector for each widget node. Additionally, with a 9-dimensional vector indicating the action type, each action is represented as an 86-dimensional vector. In contrast to encodings [12, 58, 66] that do not differentiate widget semantics or even discard essential attributes, our distinct embedding enables the identification of similar widgets. Consequently, sharing testing knowledge between similar actions, e.g., the same type of actions on similar widgets, becomes feasible.

**State Embedding.** State encoding plays a crucial role in identifying similar states. Existing RL-based strategies [54, 58] typically represent states in two ways: utilizing one-hot encodings of activities and widget availability or modeling a GUI hierarchy as a sequence. However, these representations ignore the structural relations between widgets, hindering the recognition of similar states with similar structures. This further impedes the sharing of testing knowledge between similar states. To preserve the semantic information of widgets and the structural relations between widgets, we model a GUI hierarchy as a widget graph and employ graph embedding to achieve an informative state representation.

Graph embedding, particularly graph-level representation learning, learns informative representations of entire graphs to benefit downstream tasks, achieving remarkable results across various domains [10, 38, 75, 76]. While supervised training methods are prevalent [71, 72, 74], acquiring labeled data can be challenging or even impossible in many domains [63, 73]. Therefore, there has been an increase in employing unsupervised or semi-supervised manners [6, 63, 73]. Notably, we employ unsupervised learning for generality, as it is widely acknowledged that in Android GUI testing, collecting raw GUI hierarchies is straightforward, but labeling is a significant challenge. It is labor-intensive and subjective while resulting in additional biases [6] that reduce models' generality. Specifically, the GUI hierarchy is transformed into a widget graph $G = (\mathcal{V}, \mathcal{E})$ comprising a vertex set $\mathcal{V}$ containing node vectors of widgets and an edge set $\mathcal{E}$ describing relations between parent and child nodes in the GUI hierarchy tree. We construct a training dataset and pre-train a GNN for graph embedding.

To collect a comprehensive training dataset, we followed a series of steps to gather a substantial number of training samples. First, we accessed the Google Play Store [19] and collected the top recommended apps from 16 main categories. We tried configuring them on an Android emulator [20], where 118 apps were successfully installed. Second, we employed a random exploration strategy for each app and stored the GUI hierarchy after each transition. Then, with data deduplication and error removal, we preprocessed these GUI hierarchies. Finally, a training dataset consisting of 101,920 XML-style GUI hierarchies was constructed.

We utilize InfoGraph [63], a cutting-edge unsupervised graph embedding approach. To more effectively exploit structural information, InfoGraph employs GIN [72], which has demonstrated performance comparable to the Weisfeiler-Lehman graph isomorphism test, instead of widely-used GCN [39] that struggles to distinguish intricate graph structures. As our goal of enhancing testing knowledge sharing necessitates the identification of similar states, we adopt InfoGraph for its superior structure recognition capability. We construct our GNN on the foundation of InfoGraph and pre-train it on our training dataset for 1,000 epochs over a week. The output 96-dimensional vectors of our GNN represent the states.

Notably, our fine-grained GUI embedding is the base of our testing knowledge-sharing mechanism, which works well even in intractable scenarios like scrolling up or down on *ListView* widgets. Though states before and after the scroll event are probably recognized as different states with some differences, these states contain widgets that exhibit similar structural and semantic features. With our GUI embedding, DQT identifies these states as highly similar and shares testing knowledge between them.

## 3.4 DQN-Based Scoring

DQT employs a specially designed DQN to learn from runtime testing experiences and share testing knowledge between similar states or actions. It leverages the learned testing knowledge to score every action of the current state, identifying which action is more valuable. Our DQN is designed based on Nature DQN [52], incorporating the following essential design decisions: ① To address the complex state-action space of Android apps, we incorporate action vectors as part of the input and take inspiration from DenseNet [35] and Dueling DQN [68], where the network architecture is enhanced with a dense block and a dueling architecture to increase expressiveness while maintaining lightweight. ② A fine-grained dynamic reward function is designed with four factors: activity, state, transition, and testing time. This reward function provides valuable guidance for testing exploration. ③ Considering the limited training samples (testing interactions) obtained within a constrained test time, we employ DDQN [65] to improve Q-value evaluation and prioritized experience replay [60] with a sliding window to recognize more significant samples, thus enhancing training efficiency. Below, there are further explanations regarding these three design aspects.

**Network Architecture.** Figure 4 illustrates the architecture of our DQN. We introduce three main improvements to the network architecture above Nature DQN [52]. ① As the number of actions varies depending on the current state, we incorporate action vectors as part of the input. Compared to classic DQN approaches [51, 52, 65, 68] that feature fixed-number actions where

each dimension of the network output represents one action, this modification addresses the issue of variable action numbers. Furthermore, instead of ignoring action features, this optimization emphasizes action similarities, improving the testing knowledge sharing across actions. ② Rather than utilizing image-oriented CNNs, we construct a *Dense Block* (inspired by DenseNet [35]) composed of four fully connected layers for feature extraction. DenseNet connects each layer to every other layer in a feed-forward fashion, fostering feature reuse, significantly reducing the number of parameters, and requiring less computation while achieving competitive performance. Although deeper networks typically perform better [31], we prioritize a more compact but efficient network since deeper networks take longer to operate and are harder to converge, which would negatively reduce our testing efficiency within such a limited time. ③ Another challenge lies in accurately and differentially evaluating numerous and similar actions of a state. To tackle this issue, we adopt the Dueling DQN architecture [68] described in Equation 3, where a Q-value is indirectly obtained by a state value and a normalized action value. Unlike traditional DQN that directly achieves the final Q-values, Dueling DQN employs a shared network $\theta$ (our *Dense Block*) for feature extraction and utilizes two independent estimator networks $\beta$ and $\alpha$ (our *Estimators*) for state and action evaluations. This improvement helps capture the differences between actions more effectively and leads to a more accurate evaluation of numerous actions, further enhancing the precision and reliability of our approach. In our implementation, each *Estimator* is equipped with two fully connected layers.

**Reward Function.** Exploration is widely recognized as one of the most challenging problems for RL. To alleviate such challenge, researchers have adopted the notion of curiosity-driven exploration [47, 54, 56, 59, 77]. For GUI testing, our curiosity-driven exploration prioritizes exploring untested app paths and states. Specifically, this notion relies on the curious Q-values judged by the DQN, which is trained via rewards and the Bellman equation. Therefore, a well-defined reward function is critical for our RL-based GUI testing. We propose a fine-grained dynamic reward function. It incentivizes actions that directly lead to new activities or unexplored states, as well as actions that result in less explored states or transitions that have the potential to lead to new ones.

$$r = (R_{base} + R_{state} + \lambda R_{trans}) \times (1 + R_{time}). \quad (4)$$

Equation 4 presents the reward function, which comprises four factors. ① The first is a base reward $R_{base}$ that encourages exploring new activities and unexplored states. We assign a significant reward of 50 to $R_{base}$ when a new activity is reached. To identify new unexplored states, we utilize our graph embedding-based state vectors and compute the cosine similarity between states. We take the **largest** cosine similarity between the current state and the tested states under the same activity, denoted as $\sigma_b$, to determine $R_{base}$. Since the lower $\sigma_b$ is, the more likely it is a new state, we assign a medium reward of 10 when a potential new state is reached ($\sigma_b$<0.85), a small negative reward of -0.5 when the state is overly familiar ($\sigma_b$>0.99), and a low positive reward (1-$\sigma_b$) in other cases. ② The second component is a state exploration rate, denoted as $R_{state}$. This factor encourages exploring less explored states corresponding to less triggered functionalities, which potentially guide to new unexplored states. We compute $\sigma_s$, the **mean** of the cosine similarity between the current state and all other tested states under the same activity and set $R_{state}$=1-$\sigma_s$. ③ The third factor is a transition rate $R_{trans}$. We encourage exploring different transition paths between states since they often correspond to different code segments. For a transition t=($s_t, a_t, s_{t+1}$) where $s_t$ is in activity $act_t$ and $s_{t+1}$ is in $act_{t+1}$, we calculate $\sigma_t$, the mean of the cosine similarity between $t$ and all other tested transitions from $act_t$ to $act_{t+1}$, and set $R_{trans}$ to 1-$\sigma_t$. Note that we assign a weight $\lambda$ to $R_{trans}$, which is set to 0.5 since we consider $R_{state}$ more significant. ④ During testing, positive rewards become incredibly sparse as more and more states are explored. To alleviate such an issue, we introduce a time-dependent reward $R_{time}$. It is proportional to the current testing step, divided by a time factor (e.g., 3600), and only given when $R_{base}$ is positive. With an increasing $R_{time}$, actions directed to unexplored app functionalities are more encouraged even after a long testing time, benefiting further exploration. Additionally, a negative reward of -10 is directly allocated to the reward $r$ when the testing process falls out of the AUT.

**Training Method.** When conducting a new test, we train a testing model about the app from scratch within a limited testing time. To enhance training efficacy, we employ DDQN [65] described in Equation 2, which mitigates the overestimation problem of Nature DQN [52], better-evaluating Q-values. Meanwhile, since our interaction data is time-sensitive, more recent experiences are more valuable. Inspired by prioritized experience replay [60], we propose a fixed-size sliding window covering the most recent testing experiences. We train the network with batches sampled from the sliding window, assigning higher priorities to newer experiences. Moreover, our training parameters[2] refer to previous practices [52, 60].

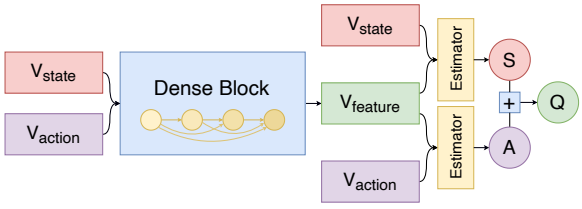


**Figure 4: The Architecture of Our Deep Q-Network.**

## 3.5 Event Selection

With Q-values predicted by the DQN, DQT prefers selecting the most valuable events (i.e., events with the highest Q-values) extracted from the GUI to execute. Meanwhile, introducing system events as a supplement to the regularly extracted GUI events is essential for exploring additional functionalities and triggering complex faults [15, 40, 54, 58, 61]. Moreover, random events (e.g., the events of Monkey [26]) are beneficial for the training of DQN [9, 17, 52]. Accordingly, we first determine the event type, i.e., GUI events, system events, or random GUI events according to Equation 5, with $\tau$ empirically set to a small value of 0.03 in our practice, where DQT pays more attention to GUI events and triggers system events at a very low level (2%), compared to other tools like Q-testing

[2] $\gamma$: 0.99, *learning rate: 0.0003, target update interval: 8, batch size: 16*

(10%) [54] and Stoat (more than 10%) [61].

$$selectEvent(s) = \begin{cases} \text{extracted GUI event}, & 1-\tau, \\ \text{random system event}, & \frac{2}{3}\tau, \\ \text{other random event}, & \frac{1}{3}\tau. \end{cases} \quad (5)$$

When picking extracted GUI events, DQT employs $\epsilon$-greedy policy, randomly selecting one at a probability of $\epsilon$ (e.g., progressively reduced from 0.9 to 0.3 like [52]) and choosing the event with the highest Q-value at a probability of 1-$\epsilon$, as shown in Equation 6.

$$chooseAction(s) = \begin{cases} \arg\max_a Q(s,a), & 1-\epsilon, \\ \text{random GUI event}, & \epsilon. \end{cases} \quad (6)$$

# 4 EVALUATION

DQT is implemented as a block-box Android GUI testing tool in Python, using UIAutomator [27, 32] for UI testing, ADB [28] for device communication, and AAPT2 [22] for packaging. Our DQN is implemented in PyTorch [55] with CUDA [53], while our GNN is built upon publicly available implementation of InfoGraph [63]. Our evaluation investigates the following research questions:

- **RQ1:** ***Code Coverage.*** How effective is DQT in achieving code coverage compared to state-of-the-art testing approaches? Does DQT exhibit enhanced effectiveness on larger-size apps?
- **RQ2:** ***Fault Detection.*** How effective is DQT in detecting faults when compared to state-of-the-art testing approaches? How many faults detected by DQT are unique? Do the detected faults represent real ones in real-world development environments?
- **RQ3:** ***Testing Knowledge Sharing.*** How does the sharing of runtime testing knowledge across states or actions influence test effectiveness? Specifically, to what extent does it contribute to improvements in code coverage and fault detection capabilities?

## 4.1 Experimental Setup

**Experimental Environment.** The experimental environment comprises a workstation with an Intel Core i7-7800X @ 12x 4GHz CPU, 64GB RAM, and an NVIDIA GeForce GTX 1080 Ti GPU. Android emulators [20] are used as experimental devices to ensure reproducibility. The chosen operating system version for these emulators is Android 9.0 (API Level 28), which is more recent and widespread than the versions used in previous studies [15, 30, 42, 54, 58]. Emulators operate concurrently with each configured with a 1GB SDCard to store coverage files and 4GB RAM to ensure the smooth operation of large apps (e.g., *Signal*), where the experiment lasted over two months even with three emulators.

**Benchmark Apps.** The benchmark consists of 30 open-source Android apps of varying sizes sourced from Q-testing [54] and GitHub. These apps were selected to represent a diverse range of applications and sizes. First, apps were collected from the Q-testing benchmark, an improved version of the AndroTest benchmark [11] that has gained widespread recognition[15, 41, 45, 54, 58, 61]. Each app was examined, and those still under active maintenance were included. The selection criterion required at least one commit record within the past three years, resulting in 22 apps. To further diversify the benchmark and minimize potential bias, larger open-source Android apps were sought from GitHub. Using keywords *android app*, search results were sorted by stars in descending order. After excluding unsuitable projects (such as plugins, libraries, and games), 12 projects with over 30,000 CLOC [13] from the top 50 were chosen. Additionally, 4 projects that failed to compile were excluded, as Jacoco [16] was used for ELOC and instruction counting, requiring manual project compilation. Ultimately, 8 apps were selected. The latest versions were adopted to resemble real-world development environments. The *App* column in Table 1 provides additional details about the benchmark apps, including their GitHub/GitLab stars (until Feb. 4, 2023), testing version, number of instructions, ELOC counted by Jacoco [16], and number of activities. The apps are sorted in descending order according to their instruction counts. It is worth noting that most of our benchmark apps are also real-world apps. For instance, 26 of the 30 apps are available on Google Play, containing popular apps like *Signal*, *CoronaWarn*, *DuckDuckGo*, and *Firefox* which have been downloaded more than 10 million times.

**Baseline Approaches.** DQT is compared with six state-of-the-art Android GUI testing approaches: Monkey [26], Stoat [61], APE [30], Q-testing [54], DeepGUIT [12], and ARES [58]. Monkey, a practical tool developed by Google, is commonly employed in real-world development scenarios. Stoat and APE are renowned model-based testing approaches that have received extensive recognition. Q-testing, DeepGUIT, and ARES are recent RL-based testing approaches, with DeepGUIT and ARES further incorporating Deep RL techniques. Attempts were also made to Qdroid [66] and DeepGUI [14]. However, the artifact of Qdroid is not publicly available, while DeepGUI only supports Android 2.3.3 (API Level 10) and cannot support our benchmark apps requiring at least Android 4.4 [21]. To enhance the validity of the experiments, we adopted the provided default settings by each approach, and for ARES, which necessitates numerous settings, we sought guidance from its authors to ensure its potential was fully exploited in our experiments. Following previous works [11, 30, 54], a delay of 200 milliseconds is set between actions for Monkey. As for ARES that incorporates several Deep RL algorithms, we select the best-performing SAC algorithm [58].

Further, we assess the impact of sharing runtime testing knowledge across states or actions on testing effectiveness. While the GUI embedding and the reward function can distinguish states and actions or specify the type of knowledge to be learned, thereby aiding in knowledge sharing, the pivotal element in this process is the DQN that determines which states and actions can share knowledge. We introduced a variation of our tool, denoted as DQT*, where we replaced our DQN with a Q-table utilized in Q-testing. In this variant, all other modules, including *GUI embedding*, *Reward Function*, and *Event Selection*, remain unchanged. In DQT*, each (state, action) pair is treated as a cell in the table. As such, the sharing of runtime testing knowledge across states or actions is no longer possible. We then compared DQT with DQT* to understand the influence of knowledge sharing on test effectiveness.

**Experimental Configurations.** Measures have been taken to minimize external interference and enhance experimental reliability. Apps are evenly distributed among emulators, with each app assigned to a specific emulator to reduce the impact of emulator variability. A preliminary snapshot captured for each emulator initializes the emulator before every test to ensure a consistent initial state. For apps requiring a login for normal operation, namely *Signal*, *K9Mail*, and *Conversations*, we manually logged them into a common test account before capturing the snapshot. Consequently,

**Table 1: Testing Results for Comparison.**

| App | | | | | | | Average Instruction Coverage (%) | | | | | | | | Total Unique Faults | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ID | Name | Star | Ver. | Inst. | ELOC | Act. | **DQT** | DQT* | APE | QT | ARES | Stoat | DeepG | Monkey | **DQT** | DQT* | APE | QT | ARES | Stoat | DeepG | Monkey |
| 1 | Signal | 23.3k | 6.3.3 | 904470 | 174574 | 84 | 39.58 | 33.28 | 23.95 | 29.11 | 17.84 | 21.78 | 23.23 | 19.24 | 9 | 6 | 0 | 6 | 1 | 8 | 4 | 2 |
| 2 | CoronaWarn | 2.5k | 2.28.3 | 459709 | 70154 | 12 | 30.45 | 17.75 | 11.82 | 29.20 | 21.07 | 14.53 | 10.35 | 25.49 | 2 | 0 | 0 | 2 | 0 | 0 | 0 | 1 |
| 3 | Tachiyomi | 20.8k | 0.13.6 | 412967 | 45341 | 13 | 56.41 | 51.26 | 44.56 | 37.95 | 26.12 | 21.44 | 22.68 | 38.57 | 7 | 5 | 6 | 5 | 1 | 2 | 1 | 6 |
| 4 | DuckDuckGo | 2.9k | 5.140.0 | 363204 | 60207 | 55 | 41.15 | 38.34 | 31.49 | 36.17 | 30.00 | 31.56 | 20.04 | 32.33 | 5 | 5 | 1 | 5 | 1 | 5 | 2 | 2 |
| 5 | Firefox | 6.7k | 110.0a1 | 348086 | 53428 | 17 | 49.91 | 46.18 | 48.94 | 42.21 | 23.91 | 27.48 | 20.39 | 37.34 | 7 | 0 | 0 | 3 | 1 | 5 | 1 | 3 |
| 6 | MyExpenses | 507 | 3.4.5 | 318189 | 50665 | 44 | 42.24 | 35.28 | 38.58 | 37.68 | 28.30 | 20.72 | 21.59 | 35.51 | 3 | 3 | 0 | 4 | 1 | 3 | 3 | 2 |
| 7 | AnkiDroid | 6k | 2.16 | 259648 | 43043 | 36 | 42.27 | 34.85 | 38.65 | 34.89 | 20.85 | 18.48 | 14.49 | 23.35 | 7 | 7 | 4 | 5 | 2 | 1 | 1 | 3 |
| 8 | K9Mail | 7.6k | 6.306 | 208089 | 37149 | 33 | 49.76 | 46.28 | 39.64 | 46.84 | 39.73 | 36.55 | 35.27 | 45.05 | 7 | 5 | 2 | 4 | 2 | 4 | 0 | 7 |
| 9 | Conversations | 4.2k | 2.10.10 | 191099 | 42740 | 36 | 46.73 | 43.79 | 39.69 | 31.51 | 29.88 | 33.06 | 36.83 | 38.12 | 4 | 4 | 2 | 1 | 0 | 4 | 2 | 2 |
| 10 | SuntimesWidget | 249 | 0.14.7 | 182463 | 36360 | 25 | 48.12 | 46.20 | 51.17 | 43.76 | 19.35 | 28.41 | 26.28 | 34.42 | 3 | 2 | 3 | 3 | 1 | 1 | 3 | 2 |
| 11 | NewPipe | 22.6k | 0.23.3 | 149421 | 34710 | 14 | 51.54 | 46.26 | 50.36 | 42.95 | 36.32 | 31.52 | 38.13 | 44.61 | 4 | 5 | 7 | 5 | 1 | 1 | 1 | 4 |
| 12 | AmazeFileManager | 4.3k | 3.7.2 | 136538 | 29671 | 10 | 31.31 | 27.74 | 31.60 | 30.71 | 22.94 | 21.76 | 17.63 | 22.97 | 4 | 4 | 6 | 6 | 1 | 5 | 1 | 3 |
| 13 | MoneyManagerEx | 361 | 2021.05.13 | 127783 | 28637 | 52 | 43.85 | 36.92 | 28.29 | 41.67 | 23.60 | 30.95 | 27.54 | 21.08 | 10 | 4 | 2 | 5 | 5 | 6 | 4 | 3 |
| 14 | BookCatalogue | 366 | 5.3.0-5 | 117790 | 24664 | 35 | 45.93 | 42.68 | 37.35 | 38.83 | 28.50 | 27.61 | 29.71 | 36.62 | 4 | 4 | 4 | 4 | 1 | 4 | 2 | 3 |
| 15 | AntennaPod | 4.7k | 2.7.1 | 114020 | 27207 | 11 | 56.34 | 52.79 | 53.65 | 51.34 | 39.84 | 13.99 | 29.65 | 30.01 | 4 | 3 | 2 | 1 | 1 | 1 | 0 | 5 |
| **Average (App 1-15)** | | | | | | | 45.04 | 40.00 | 37.98 | 38.32 | 27.22 | 25.32 | 24.92 | 32.31 | 5.33 | 3.80 | 2.60 | 3.93 | 1.27 | 3.33 | 1.67 | 3.20 |
| 16 | LBRY | 2.5k | 0.17.1 | 89679 | 16460 | 6 | 51.33 | 48.84 | 51.07 | 41.32 | 26.79 | 42.72 | 38.83 | 49.23 | 3 | 4 | 4 | 4 | 1 | 4 | 0 | 12 |
| 17 | RunnerUp | 636 | 2.4.5.0 | 73479 | 16592 | 17 | 48.30 | 38.88 | 43.72 | 27.87 | 27.88 | 22.75 | 9.34 | 21.89 | 4 | 4 | 4 | 2 | 2 | 1 | 0 | 3 |
| 18 | ConnectBot | 2k | 1.9.8 | 70609 | 8985 | 11 | 31.92 | 29.84 | 31.01 | 27.83 | 25.30 | 21.50 | 28.47 | 27.51 | 4 | 0 | 4 | 1 | 1 | 0 | 2 | 6 |
| 19 | APhotoManager | 210 | 0.8.3 | 54313 | 11451 | 11 | 59.12 | 40.67 | 53.81 | 37.48 | 26.83 | 28.42 | 29.44 | 35.03 | 9 | 0 | 9 | 2 | 0 | 3 | 5 | 2 |
| 20 | Timber | 6.8k | 1.8 | 54218 | 12002 | 6 | 36.34 | 35.74 | 30.24 | 30.33 | 24.24 | 19.24 | 24.79 | 34.68 | 7 | 7 | 3 | 5 | 2 | 1 | 5 | 7 |
| 21 | BetterBatteryStats | 551 | 3.0 | 52560 | 11118 | 12 | 28.09 | 27.98 | 27.83 | 29.09 | 38.47 | 28.85 | 25.49 | 30.21 | 4 | 3 | 1 | 3 | 1 | 3 | 1 | 2 |
| 22 | Vanilla | 966 | 1.1.0 | 47306 | 10477 | 13 | 46.20 | 45.85 | 45.33 | 45.83 | 36.49 | 34.38 | 32.31 | 39.28 | 6 | 4 | 6 | 3 | 5 | 2 | 1 | 6 |
| 23 | AnyMemo | 142 | 10.11.7 | 46259 | 10046 | 28 | 64.38 | 59.31 | 54.67 | 60.27 | 33.20 | 35.66 | 26.10 | 48.98 | 14 | 13 | 8 | 14 | 3 | 12 | 11 | 3 |
| 24 | LoopHabitTracker | 5.9k | 2.1.1 | 45946 | 9139 | 11 | 57.64 | 48.72 | 44.37 | 38.92 | 32.95 | 35.11 | 27.50 | 36.60 | 6 | 6 | 2 | 6 | 3 | 4 | 2 | 5 |
| 25 | KeePassDroid | 1.3k | 2.6.8 | 38874 | 9600 | 15 | 14.32 | 14.21 | 12.28 | 13.29 | 11.95 | 11.39 | 12.90 | 12.09 | 4 | 0 | 0 | 3 | 0 | 3 | 1 | 1 |
| 26 | AlarmClock | 366 | 3.11.00 | 35197 | 5404 | 5 | 81.07 | 76.99 | 81.17 | 71.19 | 69.53 | 61.61 | 63.57 | 73.04 | 5 | 0 | 2 | 4 | 2 | 4 | 1 | 3 |
| 27 | Materialistic | 2.2k | 3.3 | 33858 | 7790 | 23 | 65.75 | 60.81 | 69.64 | 56.31 | 48.53 | 51.26 | 48.54 | 53.32 | 6 | 4 | 3 | 3 | 2 | 1 | 2 | 2 |
| 28 | Notes | 76 | 1.4.1 | 18841 | 3510 | 12 | 52.06 | 41.15 | 50.62 | 48.02 | 39.70 | 21.64 | 44.03 | 43.93 | 7 | 6 | 6 | 5 | 6 | 0 | 5 | 4 |
| 29 | Swiftp | 661 | 3.1.0 | 11778 | 3094 | 4 | 21.54 | 18.92 | 18.81 | 19.53 | 18.62 | 18.70 | 16.29 | 23.84 | 3 | 1 | 1 | 1 | 1 | 2 | 1 | 2 |
| 30 | WhoHasMyStuff | 8 | 1.1.0 | 4849 | 820 | 7 | 69.91 | 66.91 | 68.62 | 69.15 | 64.49 | 62.30 | 67.78 | 58.77 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| **Average (App 1-30)** | | | | | | | 46.79 (±1.42) | 41.83 (±3.03) | 41.76 (±2.19) | 39.71 (±2.90) | 31.11 (±2.87) | 29.18 (±2.56) | 28.97 (±3.96) | 35.77 (±4.00) | 5.47 | 3.67 | 3.10 | 3.87 | 1.63 | 3.03 | 2.10 | 3.53 |

with the same snapshot load, these apps are logged in consistently, ensuring uniformity across different tests. Furthermore, two hours are given to test each app more thoroughly (for Stoat, we follow [54] and allocate one hour for modeling and one for sampling). Each test is repeated five times to mitigate randomness, with the averages as the final results. While DQT is a black-box testing tool capable of testing apps without their source code, to obtain code coverage information, we utilized Jacoco [16], which should be instrumented in the apps' source code. Faults are recorded by monitoring runtime Logcat [25] messages. Our definition of a fault aligns with Q-testing and Stoat, where crash or exception lines in the stack trace indicate a fault. To further enhance the quality of the detected faults, we follow ARES and APE, where only fatal errors are focused, as they significantly impact user experience and have been taken more seriously [23, 29]. Lastly, all identified faults were de-duplicated based on their types and trigger lines to ensure uniqueness.

## 4.2 Experimental Results

**RQ1:** ***Code Coverage.*** Table 1 presents the average instruction coverage for five rounds of testing on 30 open-source apps by DQT, APE, Q-testing (QT), ARES, Stoat, DeepGUIT (DeepG), and Monkey. Generally, DQT achieves the average instruction coverage of 46.79%, outperforming APE (41.76%) by 12.05%, Q-testing (39.71%) by 17.83%, and significantly outperforming Monkey (35.77%) by 30.81%, ARES (31.11%) by 50.40%, Stoat (29.18%) by 60.35%, and DeepGUIT (28.97%) by 61.51%. DQT attains the highest instruction coverage in 24 of the 30 apps. This result is noteworthy, considering over the past decade, Android GUI testing approaches have reached a certain level of development and encountered various bottlenecks [7, 62, 67]. Smaller apps are relatively easier to test due to simpler logic and fewer features [67]. Consequently, larger apps demand more attention. We classify apps with over 20,000 ELOC into a large-app group containing 15 apps shown in the top half of Table 1. Among these large apps, DQT's advantage in instruction coverage over APE, Monkey, ARES, Stoat, and DeepGUIT increase to 18.59%, 39.40%, 65.47%, 77.88%, and 80.74%, which demonstrates DQT's superior exploration capabilities in large, complex apps. Meanwhile, DQT maintains a 17.54% advantage over Q-testing, indicating that Q-testing may also do better in exploring larger apps but is generally less efficient than DQT. Note that achieving higher code coverage in large, complex apps is usually more challenging as every percentage of coverage means more ELOC and app functionalities. On our largest app, *Signal*, DQT exhibits a remarkable 35.97% advantage over Q-testing, which performs best among the other six approaches. This implies that DQT covers 90,000 more instructions than Q-testing within the 2-hour testing period.

To further elucidate DQT's superiority, we examine the scored Q-values of *Signal* as an example. Initially, DQT assigns all actions on the home page ($S_\alpha$ in Figure 1) Q-values close to zero. After 22 testing steps of exploration and learning on this page, with several

actions tried, DQT accurately recognizes the importance of the clicking action on *More Options* (denoted as $a_{m1}$) and assigns it a Q-value of 3.51, higher than 90% of other actions. When entering the dialog page ($S_\beta$ in Figure 1) for the first time (another 32 testing steps later) and identifying the *More Options* clicking action (denoted as $a_{m2}$) on this page, DQT discovers the similar state structure and widget semantic between actions $a_{m1}$ and $a_{m2}$ via its graph embedding. Ultimately, with the testing knowledge learned before, DQT assigns action $a_{m2}$ a Q-value of 14.9, higher than 83% of other actions on the same page. DQT's ability to effectively learn and share testing knowledge among similar states or actions allows it to avoid learning from scratch repeatedly, making it more effective. In contrast, other RL-based approaches are limited by Tabular RL or under-optimized Deep RL, thus unable to share testing knowledge. Even when they have recognized the importance of $a_{m1}$ and assigned it a high Q-value, they cannot recognize the importance of $a_{m2}$ until they thoroughly explore the dialog page.

Figure 5 presents the progressive average instruction coverage for each approach on the 30 apps in five rounds. APE maintains the highest coverage in the initial fifteen minutes, largely benefiting from its rapid event generation. However, DQT surpasses APE thereafter, and the advantage increases throughout the remaining testing period. This is mainly because DQT accumulates more and more testing knowledge as the test progresses, allowing for a more comprehensive evaluation of each state-action pair and thus enhancing testing effectiveness. It is essential to emphasize that obtaining new increments during the later testing stage becomes particularly challenging as most easy-to-trigger app functionalities have been tested. Nevertheless, DQT maintains substantial growth even during the test's latest stages, unequivocally demonstrating the effectiveness of DQT in exploring a broader state space, especially deeply hidden states. Additionally, the last row of the *Average Instruction Coverage* column in Table 1 renders the average standard deviation of these apps for each approach. DQT's smaller standard deviation demonstrates its ability to maintain stability while preserving high exploration capacity. In comparison, APE and Q-testing exhibit greater fluctuations, while Monkey, being random-based, produces the largest range with no surprise.

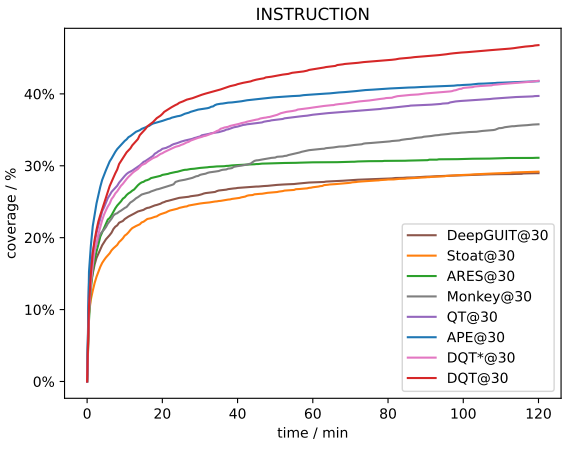


**Figure 5: Progressive Average Instruction Coverage.**

**RQ2: *Fault Detection.*** The *Total Unique Faults* column in Table 1 summarizes the total unique faults detected in five rounds on 30 open-source apps by each approach. Generally, DQT demonstrates a significantly higher fault detection number of 164 compared to APE (93), Q-testing (116), ARES (49), Stoat (91), DeepGUIT (63), and Monkey (106). On average, DQT identifies the most faults per app at 5.47, followed by Q-testing at 3.87. Furthermore, DQT detects the most faults in 24 (with tied first also counted) out of the 30 apps.

Figure 6 illustrates the pairwise comparison of detected unique faults between DQT and the other approaches. DQT successfully detects 41.94% (39/93), 60.34% (70/116), 59.18% (29/49), 51.65% (47/91), 55.56% (35/63), and 33.02% (35/106) unique faults identified by APE, Q-testing, ARES, Stoat, DeepGUIT, and Monkey, respectively. In contrast, the other six approaches can only detect 23.78%, 42.68%, 17.68%, 28.66%, 21.34%, and 21.34% of the 164 unique faults found by DQT. Note that DQT detects more than half of ARES's or DeepGUIT's faults, while ARES or DeepGUIT only detect about one-fifth of DQT's faults, although they all incorporate Deep RL techniques. Among all 361 faults detected by all seven approaches, DQT identifies nearly half (45.43%). Additionally, 57 faults can only be detected by DQT, while the numbers are 30, 18, 14, 16, 15, and 49 for APE, Q-Testing, ARES, Stoat, DeepGUIT, and Monkey, respectively. Despite Monkey's ability to generate extremely fast, random event streams and detect a significant number of stress-related faults, the only detected faults by DQT still outnumber Monkey by 8. These findings emphasize DQT's superior effectiveness in detecting faults in Android apps and highlight the unique value of DQT's fault detection capability compared to other approaches.

Many of DQT's detected faults are deeply hidden in the app requiring long action sequences to trigger. A representative example is a confirmed fault (Fault #3 in Table 2) deeply concealed within the *Signal* app. To expose this fault, users must navigate through at least six different pages: the home page, quick recording page, gallery page, gallery recording page, video editing page, and multi-video editing page, where at least seven steps are needed to trigger the fault from the home page, with any missteps potentially leading to a failure of triggering. DQT successfully detects it, as DQT identifies many similar states with similar actions in the *Signal* app, such as the quick recording page and gallery recording page, as well as the video editing page and multi-video editing page. Specifically, DQT leverages the testing knowledge acquired during the exploration of

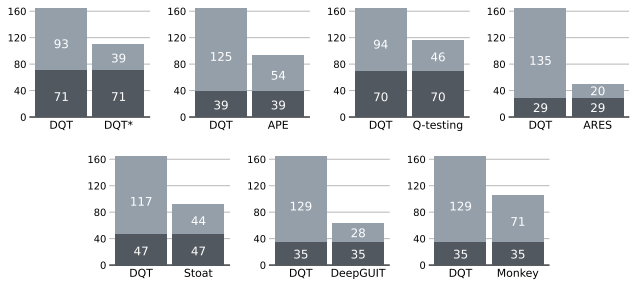


**Figure 6: Pairwise Comparison of Detected Unique Faults.**

**Table 2: Confirmed Faults Found by DQT.**

| ID | App | Star | GPI | Type | URL | FF | Status | RT | IT |
|---|---|---|---|---|---|---|---|---|---|
| #1 | Signal | 23.3k | 100m+ | java.lang.NullPointerException | https://github.com/signalapp/Signal-Android/issues/12651 | ✓ | Fixed | 2 days | 7 months |
| #2 | Signal | 23.3k | 100m+ | java.lang.IllegalStateException | https://github.com/signalapp/Signal-Android/issues/12666 | ✓ | Fixed | 2 days | 3 years |
| #3 | Signal | 23.3k | 100m+ | Application Not Responding | https://github.com/signalapp/Signal-Android/issues/12667 | ✓ | Confirmed | 2 days | - |
| #4 | Tachiyomi | 20.8k | - | java.lang.IllegalArgumentException | https://github.com/tachiyomiorg/tachiyomi/issues/8509 | | Confirmed | 1 day | - |
| #5 | DuckDuckGo | 2.9k | 10m+ | java.util.IllegalFormatConversionException | https://github.com/duckduckgo/Android/pull/2499 | | Fixed | 1 day | 1 month |
| #6 | Firefox | 6.7k | 100m+ | java.lang.IllegalStateException | https://bugzilla.mozilla.org/show_bug.cgi?id=1816288 | ✓ | Confirmed | 1 month | - |
| #7 | Firefox | 6.7k | 100m+ | java.lang.NullPointerException | https://github.com/mozilla-mobile/fenix/issues/28389 | ✓ | Confirmed | 6 days | - |
| #8 | Firefox | 6.7k | 100m+ | java.lang.IllegalStateException | https://github.com/mozilla-mobile/fenix/issues/28390 | ✓ | Confirmed | 6 days | - |
| #9 | AnkiDroid | 6k | 10m+ | java.lang.NullPointerException | https://github.com/ankidroid/Anki-Android/issues/12936 | | Fixed | 1 day | 6 months |
| #10 | NewPipe | 22.6k | - | java.lang.NullPointerException | https://github.com/TeamNewPipe/NewPipe/issues/9518 | ✓ | Fixed | 1 day | 10 months |
| #11 | NewPipe | 22.6k | - | Application Not Responding | https://github.com/TeamNewPipe/NewPipe/issues/9529 | ✓ | Confirmed | 1 day | - |
| #12 | BookCatalogue | 366 | 100k+ | android.content.ActivityNotFoundException | https://github.com/eleybourn/Book-Catalogue/issues/877 | | Fixed | 1 day | 12 years |
| #13 | BookCatalogue | 366 | 100k+ | android.content.ActivityNotFoundException | https://github.com/eleybourn/Book-Catalogue/issues/883 | ✓ | Fixed | 7 days | 10 years |
| #14 | Vanilla | 966 | 500k+ | android.content.ActivityNotFoundException | https://github.com/vanilla-music/vanilla/issues/1147 | ✓ | Fixed | 11 days | 7 years |
| #15 | LoopHabitTracker | 5.9k | 5m+ | android.content.ActivityNotFoundException | https://github.com/iSoron/uhabits/issues/1589 | ✓ | Fixed | 1 day | 6 years |
| #16 | AlarmClock | 366 | 1m+ | android.content.ActivityNotFoundException | https://github.com/yuriykulikov/AlarmClock/issues/500 | ✓ | Fixed | 2 months | 4 years |
| #17 | AlarmClock | 366 | 1m+ | java.lang.RuntimeException | https://github.com/yuriykulikov/AlarmClock/issues/501 | | Confirmed | 2 months | - |
| #18 | Notes | 76 | 10k+ | java.lang.NullPointerException | https://github.com/SecUSo/privacy-friendly-notes/issues/132 | ✓ | Fixed | 12 days | 2 months |
| #19 | Notes | 76 | 10k+ | java.lang.NullPointerException | https://github.com/SecUSo/privacy-friendly-notes/issues/133 | ✓ | Fixed | 12 days | 2 months |
| #20 | Notes | 76 | 10k+ | java.lang.NullPointerException | https://github.com/SecUSo/privacy-friendly-notes/issues/138 | ✓ | Fixed | 3 months | 5 months |
| #21 | Swiftp | 661 | 1k+ | android.content.ActivityNotFoundException | https://github.com/ppareit/swiftp/issues/174 | | Fixed | 6 months | 3 years |

the video editing page to expedite the exploration of the subsequent multi-video editing page, ultimately executing the most valuable action that triggers the fault. In contrast, without the ability to share testing knowledge, the other six approaches are typically trapped on any of the six pages and fail to detect the fault.

Furthermore, to ascertain whether the faults detected by DQT correspond to real-world issues and are valuable for developers, we have reproduced and reported the faults detected by DQT to developers. Given the time-consuming nature of preparing issue reports and our limited time, we reported 36 issues. To date, 21 of these reported issues have been explicitly confirmed by developers, with 14 already fixed. We only consider an issue confirmed if developers provide explicit text responses or tag it as a fault. Table 2 presents these faults in descending order of app size, along with app name, GitHub stars, Google Play Installations (GPI), fault type, issue URL, first found by DQT or not (FF), current status, the duration between reporting and developers' replies (RT), and the duration between faults' first injection and developers' fixes (IT). Among the 21 confirmed faults, 15 were reported for the first time, indicating that DQT effectively uncovers faults not easily found by users or developers. Furthermore, DQT respectively uncovers 3 faults in apps *Signal* and *Firefox*, two world-famous apps with millions or tens of millions of active users. Developers have shown great concern for the reported faults, with 10 issues confirmed within 2 days. For instance, Fault #15 in *LoopHabitTracker* was confirmed within one day and fixed within two days. Among the 23 latest faults reported in *LoopHabitTracker* over the past four months, this fault is one of the only two confirmed and fixed by developers. Additionally, Fault #6 in *Firefox* was assigned priority *P2*, the second-highest priority among *Firefox*'s five-level priority system. The developer has announced plans to fix it in the app's next release. These examples demonstrate the importance of the faults detected by DQT to developers. Moreover, we traced the commit records of these fixed faults to find out when they were first introduced into the app. We discovered that in the largest app, *Signal*, Fault #1 had been injected over seven months, and more surprisingly, Fault #2 had been present in the app for over three years. Furthermore, Fault #13 in *BookCatalogue* had been injected over ten years and was first found by DQT. Given that most of these apps have undergone extensive maintenance, such as *Signal*, which updates nearly every day recently. Meanwhile, they have amassed large user bases while being rigorously tested by various testing approaches. These findings demonstrate DQT's ability to reveal deeply hidden faults, which is particularly crucial for large, complex apps.

**RQ3:** ***Testing Knowledge Sharing.*** As depicted in Table 1 (DQT and DQT*), with testing knowledge sharing, the average instruction coverage among all the benchmark apps increases from 41.83% to 46.79% by 11.86%, while in the large app group, a higher increment of 12.60% from 40.00% to 45.04% has been observed. Accordingly, knowledge sharing also improves the ability of fault detection, where the average number of detected unique faults increases from 3.67 to 5.47, with a general improvement of 49.05%. As shown in Figure 6, DQT detects 64.55% unique faults detected by DQT* while this number is only 43.29% inversely. It is reasonable, as the absence of knowledge sharing necessitates that DQT* undergoes a fresh learning process each time it encounters a new state, thereby diminishing testing efficiency. Unlike DQT* that learns each (state, action) pair separately, DQT efficiently learns features of AUT's state-action space and shares testing knowledge between actions or states. Consequently, a greater number of hard-to-reach states and difficult-to-trigger events can be tested within a limited time. It is worth noting that, even without the knowledge-sharing mechanism, DQT* achieves a comparable performance compared to state-of-the-art testing approaches in terms of both code coverage and fault detection. Furthermore, as presented in Figure 5, DQT* initially exhibits performance that closely mirrors that of Q-testing in the first 40 minutes. However, it eventually surpasses Q-testing with an advantage continuously expanding as the testing progresses. This

demonstrates the effectiveness of our proposed graph embedding technique in distinguishing states and actions, and the dynamic reward function in guiding the testing process.

### 4.3 Threats to Validity

**Internal Threats.** The main internal threat comes from the settings for the seven testing approaches, which are essential to the testing results. To mitigate the threat, we adopt the original implementations and default settings for the six comparison approaches. As for DQT, we take up the default parameters of InfoGraph and refer to the parameters of previous works for our DQN.

**External Threads.** The external thread mainly derives from selecting our benchmark apps, which may not be general and practical enough. We alleviate the threat by referring to related works and filtering out outdated apps. We further enrich our benchmark by collecting popular large-scale open-source apps from GitHub.

## 5 RELATED WORK

Similar to recent works [7, 54, 58], we classify the existing automated Android GUI testing approaches into four categories.

**Random Testing.** These approaches [26, 43] employ random strategies to generate test inputs for Android apps. Monkey [26], a practical Android GUI testing approach that functions as a stress-testing tool, generates pseudo-random streams of user events at a breakneck pace, performing favorably on some benchmark apps [11]. However, without proper guidance, its performance is limited by redundant and ineffective events. In contrast, DQT extracts executable events from the current GUI and employs a powerful DQN to guide the testing process. Meanwhile, DQT utilizes Monkey to introduce extra randomness that benefits the exploration of the AUT and the training process of our Deep RL model.

**Model Based Testing.** With static or dynamic analysis, model-based approaches [2, 3, 5, 30, 41, 49, 61] build models of the AUT and derive test cases from them. Stoat [61] combines static and dynamic analysis to construct a stochastic Finite State Machine model, while APE [30] moves forward and dynamically optimizes its model with runtime information. Nevertheless, building a precise model of the AUT is challenging, especially for large, complex apps. Additionally, non-deterministic transitions [54] further interfere with models' accuracy. By contrast, instead of relying on an explicit model of the AUT, DQT adopts model-free Deep RL, learning from testing experiences and leveraging runtime testing knowledge for testing.

**Systematic Testing.** Systematic approaches [4, 15, 18, 44, 45, 48] apply techniques such as symbolic execution and evolutionary algorithms to generate testing inputs for targeted purposes (e.g., code coverage). SynthesiSE [18] builds a dynamic symbolic execution engine for enhancing the testing of Android apps. TimeMachine [15] proposes time-travel testing, where the evolved states could be resumed when needed to maximize state space exploration and code coverage. Such approaches usually suffer from state or path explosion, limiting their scalability and generality. Contrastingly, as a black-box testing approach that does not rely on any source code of AUT, DQT is much more versatile and compatible.

**Machine Learning Based Testing.** Machine learning techniques have been applied to Android GUI Testing. Many approaches train models from prior data by supervised learning and employ them for generating new tests [34, 37, 40, 42] or predicting valid test actions [14]. Given that historical data may not consistently be reliable when applied to rapidly evolving and continually iterative real-world applications, the effectiveness of approaches using supervised learning to generate new tests could be limited. DeepGUI [14] employs supervised learning to address a different task of predicting the validity of a specific action at a particular location within the GUI. The objective is to reduce the occurrence of invalid test actions in tools like Monkey. While many Android testing approaches, including our own, rely on UIAutomator to extract executable actions and thus do not encounter the problem that DeepGUI targets, the combination of DeepGUI that obtains valid actions and approaches that generate effective tests can be beneficial when applied to other platforms such as IOS and web.

Recent approaches [12, 36, 54, 58, 66] propose to use reinforcement learning to generate tests. TESTAR [36] and Q-testing [54] utilize Tabular RL while Q-testing further adopts a scenario division model for curiosity-driven testing. However, separated by table cells, runtime testing knowledge could not be shared between states or actions. Although Deep RL has been introduced [12, 58, 66], its rich potential has not been fully exploited. For instance, Qdroid [66] fails to consider the encoding of GUI actions while handling GUI states and classifies all kinds of widgets into only four distinct groups, using the quantity of these widget groups to represent a state. This coarse-grained GUI state encoding cannot effectively differentiate between two different GUI states. The output of Qdroid is also coarse, being one of the four types of widget groups instead of a specific action. DeepGUIT [12] only considers the index attribute for vectorizing GUI actions, posing challenges in effectively sharing testing knowledge among similar GUI actions, as DeepGUIT tends to treat actions with similar indexes as similar, which is clearly not aligned with reality. Additionally, these approaches have not thoroughly investigated the structure of the DQN and the reward function of reinforcement learning. Similarly, with one-hot and coarse encodings, conventional network, and coarse-grained reward function, ARES [58] also undergoes insufficient structural and semantic information usage. Compared to these approaches, instead of adopting a table or a simple multilayer perceptron, we employ a specially designed DQN guided by a fine-grained dynamic reward function for better state-action evaluation. Moreover, rather than abstracting states with one-hot or coarse encodings, we model a state as a graph and harness the strength of graph embedding to exploit a better understanding of states and actions.

## 6 CONCLUSION

In this paper, we present DQT, a Deep RL-based automated Android GUI testing approach. DQT leverages advanced graph embedding techniques to facilitate the identification of similar states or actions. Moreover, DQT employs a specially designed DQN guided by a fine-grained dynamic reward function, enabling testing knowledge sharing between states or actions. Experiments demonstrate DQT's superior performance among existing state-of-the-art approaches.

## ACKNOWLEDGMENTS

This research was supported by the National Natural Science Foundation of China under Grant Nos. 62032010, 62232014, and 61972193.

# REFERENCES


[1] 42matters. 2023. *42matters: Google Play Statistics and Trends 2023.* Retrieved February 27, 2023 from https://42matters.com/google-play-statistics-and-trends

[2] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M. Memon. 2012. Using GUI ripping for automated testing of Android applications. In *IEEE/ACM International Conference on Automated Software Engineering, ASE'12, Essen, Germany, September 3-7, 2012.* ACM, 258–261.

[3] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Bryan Dzung Ta, and Atif M. Memon. 2015. MobiGUITAR: Automated Model-Based Testing of Mobile Apps. *IEEE Softw.* 32, 5 (2015), 53–59.

[4] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. 2012. Automated concolic testing of smartphone apps. In *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE'12, Cary, NC, USA - November 11 - 16, 2012.* ACM, 59.

[5] Young Min Baek and Doo-Hwan Bae. 2016. 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, Singapore, September 3-7, 2016.* ACM, 238–249.

[6] Yunsheng Bai, Hao Ding, Yang Qiao, Agustin Marinovic, Ken Gu, Ting Chen, Yizhou Sun, and Wei Wang. 2019. Unsupervised Inductive Graph-Level Representation Learning via Graph-Graph Proximity. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019.* ijcai.org, 1988–1994.

[7] Farnaz Behrang and Alessandro Orso. 2020. Seven Reasons Why: An In-Depth Study of the Limitations of Random Test Input Generation for Android. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020.* IEEE, 1066–1077.

[8] Marc G. Bellemare, Will Dabney, and Rémi Munos. 2017. A Distributional Perspective on Reinforcement Learning. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017 (Proceedings of Machine Learning Research, Vol. 70).* PMLR, 449–458.

[9] Nicolò Cesa-Bianchi, Claudio Gentile, Gergely Neu, and Gábor Lugosi. 2017. Boltzmann Exploration Done Right. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA.* 6284–6293.

[10] Hsin-Pai Cheng, Tunhou Zhang, Yixing Zhang, Shiyu Li, Feng Liang, Feng Yan, Meng Li, Vikas Chandra, Hai Li, and Yiran Chen. 2021. NASGEM: Neural Architecture Search via Graph Embedding Method. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 35. 7090–7098.

[11] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated Test Input Generation for Android: Are We There Yet? (E). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015.* IEEE Computer Society, 429–440.

[12] Eliane F. Collins, Arilo Claudio Dias-Neto, Auri Vincenzi, and José Carlos Maldonado. 2021. Deep Reinforcement Learning based Android Application GUI Testing. In *35th Brazilian Symposium on Software Engineering, SBES 2021, Joinville, Santa Catarina, Brazil, 27 September 2021 - 1 October 2021.* ACM, 186–194.

[13] Albert Danial. 2021. *cloc: v1.92.* https://doi.org/10.5281/zenodo.5760077

[14] Faraz Yazdani Banafshe Daragh and Sam Malek. 2021. Deep GUI: Black-box GUI Input Generation with Deep Learning. In *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021.* IEEE, 905–916.

[15] Zhen Dong, Marcel Böhme, Lucia Cojocaru, and Abhik Roychoudhury. 2020. Time-travel testing of Android apps. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020.* ACM, 481–492.

[16] EclEmma. 2022. *JaCoCo Java Code Coverage Library.* Retrieved May 21, 2022 from https://www.eclemma.org/jacoco/index.html

[17] Meire Fortunato, Mohammad Gheshlaghi Azar, Bilal Piot, Jacob Menick, Matteo Hessel, Ian Osband, Alex Graves, Volodymyr Mnih, Rémi Munos, Demis Hassabis, Olivier Pietquin, Charles Blundell, and Shane Legg. 2018. Noisy Networks For Exploration. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings.* OpenReview.net.

[18] Xiang Gao, Shin Hwei Tan, Zhen Dong, and Abhik Roychoudhury. 2018. Android testing via synthetic symbolic execution. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018.* ACM, 419–429.

[19] Google. 2021. *Google Play.* Retrieved December 12, 2021 from https://play.google.com/store/apps

[20] Google. 2021. *Run apps on the Android Emulator.* Retrieved December 14, 2021 from https://developer.android.com/studio/run/emulator

[21] Google. 2021. *Sign-in on Android devices running Android 2.3.7 or lower will not be allowed starting September 27.* Retrieved June 27, 2023 from https://support.google.com/android/thread/118703101/sign-in-on-android-devices-running-android-2-3-7-or-lower-will-not-be-allowed-starting-september-27

[22] Google. 2022. *AAPT2.* Retrieved January 3, 2023 from https://developer.android.com/studio/command-line/aapt2

[23] Google. 2022. *Crashes.* Retrieved November 10, 2022 from https://developer.android.com/topic/performance/vitals/crash

[24] Google. 2022. *Fundamentals of testing Android apps.* Retrieved May 5, 2022 from https://42matters.com/google-play-statistics-and-trends

[25] Google. 2022. *Logcat command-line tool.* Retrieved January 3, 2023 from https://developer.android.com/studio/command-line/logcat

[26] Google. 2022. *UI/Application Exerciser Monkey.* Retrieved April 25, 2022 from https://developer.android.com/studio/test/other-testing-tools/monkey

[27] Google. 2022. *Write automated tests with UI Automator.* Retrieved August 9, 2022 from https://developer.android.com/training/testing/other-components/ui-automator

[28] Google. 2023. *Android Debug Bridge (adb).* Retrieved January 7, 2023 from https://developer.android.com/studio/command-line/adb

[29] Google. 2023. *ANRs.* Retrieved February 14, 2023 from https://developer.android.com/topic/performance/vitals/anr

[30] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Chun Cao, Chang Xu, Yuan Yao, Qirun Zhang, Jian Lu, and Zhendong Su. 2019. Practical GUI testing of Android applications via model abstraction and refinement. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019.* IEEE / ACM, 269–280.

[31] Kaiming He and Jian Sun. 2015. Convolutional neural networks at constrained time cost. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015.* IEEE Computer Society, 5353–5360.

[32] Xiaocong He, Yuanyuan Zou, Qian Jin, Xu Jingjie, and Xia Mingyuan. 2020. *Python wrapper of Android uiautomator test tool.* Retrieved October 25, 2021 from https://github.com/xiaocong/uiautomator

[33] Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Gheshlaghi Azar, and David Silver. 2018. Rainbow: Combining Improvements in Deep Reinforcement Learning. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 32. 3215–3222.

[34] Gang Hu, Linjie Zhu, and Junfeng Yang. 2018. AppFlow: using machine learning to synthesize robust, reusable UI tests. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018.* ACM, 269–282.

[35] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. 2017. Densely Connected Convolutional Networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017.* IEEE Computer Society, 2261–2269.

[36] Thorn Jansen, Fernando Pastor Ricós, Yaping Luo, Kevin van der Vlist, Robbert van Dalen, Pekka Aho, and Tanja E. J. Vos. 2022. Scriptless GUI Testing on Mobile Applications. In *22nd IEEE International Conference on Software Quality, Reliability and Security, QRS 2022, Guangzhou, China, December 5-9, 2022.* IEEE, 1103–1112.

[37] Nataniel P. Borges Jr., Maria Gómez, and Andreas Zeller. 2018. Guiding app testing with mined interaction models. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems, MOBILESoft@ICSE 2018, Gothenburg, Sweden, May 27 - 28, 2018.* ACM, 133–143.

[38] Nikhil Varma Keetha, Chen Wang, Yuheng Qiu, Kuan Xu, and Sebastian A. Scherer. 2022. AirObject: A Temporally Evolving Graph Embedding for Object Identification. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2022, New Orleans, LA, USA, June 18-24, 2022.* IEEE, 8397–8406.

[39] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings.* OpenReview.net.

[40] Yavuz Köroglu, Alper Sen, Ozlem Muslu, Yunus Mete, Ceyda Ulker, Tolga Tanriverdi, and Yunus Donmez. 2018. QBE: QLearning-Based Exploration of Android Applications. In *11th IEEE International Conference on Software Testing, Verification and Validation, ICST 2018, Västerås, Sweden, April 9-13, 2018.* IEEE Computer Society, 105–115.

[41] Duling Lai and Julia Rubin. 2019. Goal-Driven Exploration for Android Applications. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019.* IEEE, 115–127.

[42] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2019. Humanoid: A Deep Learning-Based Approach to Automated Black-box Android App Testing. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019.* IEEE, 1070–1073.

[43] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: an input generation system for Android apps. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013.* ACM, 224–234.

[44] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. 2014. EvoDroid: segmented evolutionary testing of Android apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014.* ACM, 599–609.

[45] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: multi-objective automated testing for Android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*. ACM, 94–105.

[46] Leonardo Mariani, Mauro Pezzè, Oliviero Riganelli, and Mauro Santoro. 2012. AutoBlackTest: Automatic Black-Box Testing of Interactive Applications. In *Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Montreal, QC, Canada, April 17-21, 2012*. IEEE Computer Society, 81–90.

[47] Pietro Mazzaglia, Ozan Çatal, Tim Verbelen, and Bart Dhoedt. 2022. Curiosity-Driven Exploration via Latent Bayesian Surprise. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 36. 7752–7760.

[48] Nariman Mirzaei, Hamid Bagheri, Riyadh Mahmood, and Sam Malek. 2015. SIG-Droid: Automated system input generation for Android applications. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*. 461–471.

[49] Nariman Mirzaei, Joshua Garcia, Hamid Bagheri, Alireza Sadeghi, and Sam Malek. 2016. Reducing combinatorics in GUI testing of android applications. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. ACM, 559–570.

[50] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. 2016. Asynchronous Methods for Deep Reinforcement Learning. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016 (JMLR Workshop and Conference Proceedings, Vol. 48)*. JMLR.org, 1928–1937.

[51] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. 2013. Playing Atari with Deep Reinforcement Learning. *CoRR* abs/1312.5602 (2013).

[52] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin A. Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. 2015. Human-level control through deep reinforcement learning. *Nat.* 518, 7540 (2015), 529–533.

[53] NVIDIA. 2021. *CUDA Toolkit Documentation v11.3.0*. Retrieved October 20, 2021 from https://docs.nvidia.com/cuda/archive/11.3.0/

[54] Minxue Pan, An Huang, Guoxin Wang, Tian Zhang, and Xuandong Li. 2020. Reinforcement learning based curiosity-driven testing of Android applications. In *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*. ACM, 153–164.

[55] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*. 8024–8035.

[56] Deepak Pathak, Pulkit Agrawal, Alexei A. Efros, and Trevor Darrell. 2017. Curiosity-Driven Exploration by Self-Supervised Prediction. In *2017 IEEE Conference on Computer Vision and Pattern Recognition Workshops, CVPR Workshops 2017, Honolulu, HI, USA, July 21-26, 2017*. IEEE Computer Society, 488–489.

[57] Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP 2019, Hong Kong, China, November 3-7, 2019*. Association for Computational Linguistics, 3980–3990.

[58] Andrea Romdhana, Alessio Merlo, Mariano Ceccato, and Paolo Tonella. 2022. Deep Reinforcement Learning for Black-box Testing of Android Apps. *ACM Trans. Softw. Eng. Methodol.* 31, 4 (2022), 65:1–65:29.

[59] Nikolay Savinov, Anton Raichuk, Damien Vincent, Raphaël Marinier, Marc Pollefeys, Timothy P. Lillicrap, and Sylvain Gelly. 2019. Episodic Curiosity through Reachability. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net.

[60] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. 2016. Prioritized Experience Replay. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*.

[61] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, stochastic model-based GUI testing of Android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*. ACM, 245–256.

[62] Ting Su, Jue Wang, and Zhendong Su. 2021. Benchmarking automated GUI testing for Android against real-world bugs. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*. ACM, 119–130.

[63] Fan-Yun Sun, Jordan Hoffmann, Vikas Verma, and Jian Tang. 2020. InfoGraph: Unsupervised and Semi-supervised Graph-Level Representation Learning via Mutual Information Maximization. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net.

[64] Richard S. Sutton and Andrew G. Barto. 2018. *Reinforcement Learning - An Introduction (second edition)*. MIT Press.

[65] Hado van Hasselt, Arthur Guez, and David Silver. 2016. Deep Reinforcement Learning with Double Q-Learning. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*. AAAI Press, 2094–2100.

[66] Thi Anh Tuyet Vuong and Shingo Takada. 2019. Semantic Analysis for Deep Q-Network in Android GUI Testing. In *The 31st International Conference on Software Engineering and Knowledge Engineering, SEKE 2019, Hotel Tivoli, Lisbon, Portugal, July 10-12, 2019*. KSI Research Inc. and Knowledge Systems Institute Graduate School, 123–170.

[67] Wenyu Wang, Dengfeng Li, Wei Yang, Yurui Cao, Zhenwen Zhang, Yuetang Deng, and Tao Xie. 2018. An empirical study of Android test generation tools in industrial cases. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*. ACM, 738–748.

[68] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas. 2016. Dueling Network Architectures for Deep Reinforcement Learning. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016 (JMLR Workshop and Conference Proceedings, Vol. 48)*. JMLR.org, 1995–2003.

[69] Christopher J. C. H. Watkins and Peter Dayan. 1992. Technical Note Q-Learning. *Mach. Learn.* 8 (1992), 279–292.

[70] Eric Wiewiora. 2010. *Reward Shaping*. Springer US, Boston, MA, 863–865.

[71] Zhang Xinyi and Lihui Chen. 2019. Capsule Graph Neural Network. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net.

[72] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2019. How Powerful are Graph Neural Networks?. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net.

[73] Minghao Xu, Hang Wang, Bingbing Ni, Hongyu Guo, and Jian Tang. 2021. Self-supervised Graph-level Representation Learning with Local and Global Structure. In *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event (Proceedings of Machine Learning Research, Vol. 139)*. PMLR, 11548–11558.

[74] Zhitao Ying, Jiaxuan You, Christopher Morris, Xiang Ren, William L. Hamilton, and Jure Leskovec. 2018. Hierarchical Graph Representation Learning with Differentiable Pooling. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*. 4805–4815.

[75] Zhuohan Yu, Yifu Lu, Yunhe Wang, Fan Tang, Ka-Chun Wong, and Xiangtao Li. 2022. ZINB-Based Graph Embedding Autoencoder for Single-Cell RNA-Seq Interpretations. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 36. 4671–4679.

[76] Guo Zhang, Hao He, and Dina Katabi. 2019. Circuit-GNN: Graph Neural Networks for Distributed Circuit Design. In *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA (Proceedings of Machine Learning Research, Vol. 97)*. PMLR, 7364–7373.

[77] Yan Zheng, Yi Liu, Xiaofei Xie, Yepang Liu, Lei Ma, Jianye Hao, and Yang Liu. 2021. Automatic Web Testing Using Curiosity-Driven Reinforcement Learning. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 423–435.