Enhancing Automated Testing With GUI Rendering Inference for Mobile Applications

Ehsan Abdollahi

A Thesis
in
The Department
of
Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of
Master's (Computer Science) at
Concordia University
Montréal, Québec, Canada

February 2025

Concordia University

School of Graduate Studies

This is to certify that the thesis prepared

By: **Ehsan Abdollahi**

Entitled: **Enhancing Automated Testing With GUI Rendering Inference for Mobile Applications**

and submitted in partial fulfillment of the requirements for the degree of

Master's (Computer Science)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining commitee:

_____ Chair
Dr. Shin Hwei Tan

_____ Examiner
Dr. Shin Hwei Tan

_____ Examiner
Dr. Xinxin Zuo

_____ Supervisor
Dr. Yang Wang

_____ Co-supervisor
Dr. Tse-Hsun (Peter) Chen

Approved by _____
Dr. Poullis Charalambos, Graduate Program Director

10 February 2025 _____
Dr. Mourad Debbabi, Dean
Faculty of Engineering and Computer Science

Abstract

Enhancing Automated Testing With GUI Rendering Inference for Mobile Applications

Ehsan Abdollahi

Increasingly complex mobile applications require more accurate methods of automated GUI testing. Traditional testing frameworks relying on fixed delays or pixel-based image comparison methods have a lot of limitations. Most of the time, these methods misclassify GUI rendering states, which leads to false positives. This thesis proposes a new approach to these issues by the inference of the rendering state of GUIs. Drawing on large-scale pre-trained image classification models like Vision Mamba, it enables the accurate classification between rendered GUIs. It does this through a fine-tuning process of a large model. It also involves more sophisticated model-training techniques to ensure that the best is obtained. Instead, this system architecture's semantic examination of GUI elements goes deep into more meaningful matches of context and visual information well beyond anything at the level of pixels.

The efficiency and accuracy of GUI testing will increase significantly with the proposed approach. That is different from fixed throttles that introduce unnecessary delays: it guarantees automated tests execute on fully rendered GUIs, which, in turn, reduces false positives. With this enhancement, development teams will save time and resources. Deep learning-based classification introduces a dynamic system that changes according to various GUI rendering scenarios; therefore, it is also more robust than what is already available.

The contributions of this thesis are three-fold: it proposes a new deep learning-based approach for inferring GUI rendering states, develops a high-quality dataset to support fine-tuning models, and performs an in-depth comparative analysis of large image classification models for GUI testing.

Statement of Originality

I hereby declare that I am the sole author of this thesis. All ideas and inventions attributed to others have been properly referenced. I understand that my thesis may be made electronically available to the public.

Acknowledgments

First and foremost, I would like to express my deepest gratitude to my supervisor, Dr. Yang Wang and my co-supervisor, Dr. Tse-Hsun (Peter) Chen, for your exceptional guidance, patience, insights, and encouragement. Without your supervision and invaluable support, this thesis would not have been possible.

Special thanks to my thesis examiners, Dr. Xinxin Zuo and Dr. Shin Hwei Tan for their extremely valuable and constructive suggestions.

Lastly, I want to thank my family for their constant support and understanding during this period. Your love and encouragement have been my driving force.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

Automated testing represents one of the core activities of the software development life cycle due to its ability to detect defects much faster, help in improving code quality, and allow smoother development flows. However, with the increase in mobile applications getting so feature-rich and graphically complex, the importance of GUI testing has also evolved to be very significant. Unlike traditional software testing, GUI testing allows confirmation that all visible screen controls, such as buttons, text boxes, and images, all work as they should. It ensures that the users will have proper navigation, correct visual appearance, and error-free interaction with the application.

However, GUI rendering has been considered as one of the key challenges in GUI testing. Rendering here refers to visually displaying the user interface on a screen. In a mobile application, GUIs can either be fully or partially rendered and depend on network speed, hardware limitations, and software performance [14]. If automated testing on a partially rendered GUI proceeds, it may miss critical bugs or fail due to the partial loading of elements [44], [10]. This challenge calls for more enhanced mechanisms to infer the rendering state of GUIs before testing.

This thesis is motivated by the general inefficiency of the existing GUI testing frameworks, which either depend on fixed time delays known as throttles or simplistic image comparison methods, such as the Structural Similarity Index [34], [14].

The SSIM has been chosen as the baseline for GUI rendering state classification because it is one of the most used metrics in image quality assessment. SSIM is a metric designed to compare the structural similarity between two images, making it suitable for detecting differences in rendering states. Another work [14] employed SSIM for GUI testing, classifying completely and partially rendered GUIs; thus, arguing that the presence of a structural difference in pixel-level composition

within images may point out to its rendering state.

However, SSIM has the limitation of over-dependence on pixel-level similarities that cannot provide much semantic difference between fully and partially rendered GUIs. For instance, rendering artifacts or suboptimal lightning conditions can trigger minor changes of pixel values that are leading to SSIM self-miscategorization, unable to interpret by SSIM for a deeper visual context in which such pixel values should relate. Moreover, SSIM does not provide higher-level analysis related to structural relationships among various GUI elements, basic for valid inference of the correctness of the rendering states.

These limitations further indicate that a more sophisticated approach is required. This thesis leads to a new direction of GUI rendering inference, which attempts to classify full and partial GUI renderings precisely. That would prevent the early runs of tests by a method perhaps proposed in this work that could improve testing efficiency and further ensure the reliability of automated tests. The core contribution of this thesis is developing a system that leverages the most recent deep learning models and advances in computer vision to classify GUI rendering states accurately. Unlike SSIM, which relies on pixel similarity, the proposed approach uses models analyzing semantic features in GUI images, thus enabling more precise classification.

It leverages large-scale image classification pre-trained models-Vision Mamba, Vision Transformers, and ResNet-trained on huge image datasets and fine-tuned on a selected dataset of GUI images from RICO, one of the largest public datasets on mobile application user interfaces. It would be desirable to develop, through a carefully designed fine-tuning process, a model that could classify GUI images as either fully or partially rendered with high precision and recall.

1.2 Research Hypothesis

Thesis Statement: This thesis hypothesizes that deep learning-based methods can significantly enhance the accuracy, efficiency, and reliability of automated GUI testing by accurately inferring GUI rendering states. By addressing the limitations of traditional approaches like SSIM and fixed throttling, this research aims to improve testing workflows and outcomes through the use of advanced computer vision techniques.

1.3 Thesis Overview

This section presents an overview of the thesis, including a summary of each chapter.

1.3.1 Chapter 2: Background and Literature Review

This chapter summarizes several of the basic concepts and approaches used as the basis for GUI testing. The importance of GUI testing for delivering the best possible user experience is underlined. How increased complexity of GUI rendering, device fragmentation, and missing objects in layout files are imposing new challenges to conventional ways of testing. The chapter covers a wide spectrum of automated GUI testing techniques, from random and model-based to system- and learning-based methods. The discussions involve record-and-replay frameworks, vision-based testing, and the integration of state-of-the-art technologies like large language models. These will empower better, more efficient, scalable, semantically richer testing strategies that come closer to human perception, handle diverse GUIs, and adapt to ever-evolving app ecosystems.

This chapter will go into detail about the challenges of classic test tools, particularly with regard to cross-device and cross-platform, introduce the emergence of vision-based and AI-driven approaches, and strongly emphasize how such methods based on vision analysis, object detection, and large language models solve some of the challenges presented by pure pixel-level approaches. That is, such developments provide a higher degree of accuracy in context-aware GUI testing, hence extending the scope of testing and enhancing reliability and adaptability. Jointly, they form the path for test frameworks to be increasingly more robust, keeping pace with rapid development and diversity seen in today's mobile apps.

1.3.2 Chapter 3: Infer the GUI Rendering State

This chapter presents the challenges to be resolved with the precise measurement of GUI rendering states in tests; the limitations of having a fixed throttling period are also discussed, aside from the SSIM-based method called ADAT. While ADAT uses a light CNN for extracting full and partial render results from screenshots, this approach suffer from incorrect automatic dataset labeling to train its network. The chapter addresses this with a better approach: fine-tuning Vision Mamba, a deep-learning model that exhibits much better robustness, accuracy, and generalization when trained on a manually curated subset of RICO. This approach does not only remove the need to label a large dataset but also outperforms alternative models such as Vision Transformers and ResNet.

By demonstrating the performance of Vision Mamba on complicated rendering cases, it shows how semantically rich models, coupled with data-driven metrics, hold a superior edge over pure pixel similarity-based metrics. The section wraps up with an assurance of more resilient, scalable GUI state classification using the architectural abilities of Vision Mamba in testing for both efficiency and reliability.

1.3.3 Chapter 4: Results of the Study

This chapter presents a comprehensive evaluation of the proposed GUI rendering state classification approach by comparing the performance metrics of Vision Mamba, Vision Transformers, and ResNet. In contrast to an SSIM-based approach, such as ADAT, which is shown to be capable of producing a high labeling error rate, Vision Mamba significantly reduces misclassifications with its fine-tuned, manually curated dataset. This refined dataset and advanced architectural features of the model together help it to generalize better on complex GUI states and outperform Vision Transformers and ResNet on accuracy and stability.

1.3.4 Chapter 5: Thesis Contributions and Future Work

In this chapter, we summarize the contributions of this thesis and discuss several potential directions for future work.

1.4 Thesis Organization

The rest of this thesis is organized as follows: Chapter 2 discusses this thesis's background and literature review. Chapter 3 presents our study on GUI rendering state classification for Android applications. Chapter 4 describes the results of our approach. Chapter 5 concludes the thesis and discusses the potential directions for future work.

Chapter 2

Background and Literature Review

In this chapter, we first present the foundational concepts related to GUI testing, the role of GUI rendering in software development, and the various approaches to automated GUI testing. We also present a detailed literature review of previous works and highlight gaps in the current state of research that this thesis aims to address.

2.1 Introduction to GUI Testing

Background of Graphical User Interface Testing. The Graphical User Interface (GUI) has become a crucial component of mobile applications, serving as the primary connection between apps and end users and directly influencing user experience. Poor quality in the GUI can diminish the overall value and functionality of the mobile app. Extensive research has focused on GUI testing as a reliable approach to maintaining app quality. Through thorough GUI testing, developers can confirm that mobile apps' visual and interactive aspects fulfill functional requirements while offering a smooth and intuitive user experience.

As mobile applications evolve rapidly, their inherently GUI-intensive and event-driven characteristics make the Graphical User Interface of a mobile app critically important [4]. The mobile devices' inherent limitations, such as small screen sizes and limited user attention, demand intuitive and efficient interfaces. A robust mobile app GUI enhances user experience and facilitates the execution of complicated tasks with minimal friction and cognitive load [28]. A well-designed GUI, a crucial bridge between apps and end users, becomes critical in improving user satisfaction, reducing potential errors in user operations, and ultimately determining the app's success in a competitive marketplace [2]. In this context, the GUI serves as a visual layer and an essential intermediary between user intent and app functionality. As a result, even a single bug in the GUI of mobile apps

can lead to significant consequences. A faulty GUI undermines the overall user experience and can cause potential misbehavior within the app [30]. In some cases, a GUI bug can even expose users to security risks, potentially putting sensitive data in jeopardy [47]. Thus, ensuring GUI quality is a significant part of the whole mobile app quality assurance.

The software testing community has long been dedicated to enhancing the effectiveness and efficiency of mobile app GUI testing in all aspects, with efforts since the previous century [33]. GUI testing is a vital aspect of mobile app testing, primarily aimed at verifying that the app's GUI meets specified design requirements. This involves checking that essential information is displayed in the correct locations, and that test events trigger the intended responses by simulating real user actions [2, 28, 38]. GUI testing generally includes examining the functionality, appearance, and interactivity of GUI elements (such as buttons, text boxes, and menus) to confirm the smooth operation of the user interface and maintain a consistent user experience. GUI testing is challenging due to the intricate interactions between various GUI components, fragmented operating environments, custom widget designs, and the need for accurate simulation of user operations. As various app analysis technologies have advanced, GUI testing has progressively shifted from manual to automated testing, with automated techniques now playing a central role.

The integration of intelligent algorithms has significantly advanced mobile app GUI testing. Traditional GUI testing methods primarily rely on the source code or layout files of the app under test to gather information for generating or executing tests. However, these conventional methods encounter several challenges. First, the well-known "fragmentation problem" [46] of mobile platforms—referring to the variety of operating environments for mobile apps—requires GUI testing to adapt to diverse environments. This adaptation is challenging due to different implementations and underlying system support across platforms. Second, traditional techniques for GUI element detection rely on obtaining the runtime layout structures from the apps [9]. However, certain GUI elements are absent from layout files [50]. For instance, Canvas widgets, commonly used to customize widget styles and content, often do not appear in these layout files. Canvas widgets may contain elements like buttons, text fields, and others, but these embedded widgets are not displayed in the GUI layout files. Additionally, some widgets frequently refresh their content, posing a challenge for GUI testing techniques that attempt to locate preset targets after they've been updated. This requires deciding whether to identify these targets by content or location. Inaccurate widget identification can ultimately lead to failures in accurately simulating user actions.

In short, there is a significant gap between the information retrieved from layout files and what is displayed at the GUI level. Based on the mental model [7], which underpins visual metaphor design in mobile apps, a more effective approach in GUI testing is to gather information from a visual perspective [3]. This "what you see is what you get" approach aligns with how users perceive the

interface. With advancements in computer vision technologies, vision-based GUI testing approaches have emerged to address these challenges. These methods analyze and interpret the GUI of the app under test from a visual perspective, using app GUI screenshots for understanding. They offer several advantages: First, app developers often design similar GUI content and layouts across different environments, facilitating universal analysis and comprehension of the app GUI. Second, GUI element detection improves by directly analyzing app GUI screenshots like human testers do. Third, user operations can be simulated based on this vision-based detection of GUI elements.

2.2 Literature Review

Automated GUI testing involves using automated tools and scripts to execute GUI tests. This automates the testing process, including tasks such as test generation, interaction execution on the GUI, and checking whether the system's responses align with expected outcomes. In this section, we review these aspects: 1) GUI test generation, 2) GUI test record & replay, 3) GUI testing framework 4) Fundamental for GUI testing, 5) Challenges in GUI Testing.

2.2.1 Automated GUI Test Generation

As software applications have become increasingly complex, the manual creation of test cases presents a significant challenge, often resulting in a labor-intensive and time-consuming process. The advent of automated GUI test generation has, therefore, gained substantial importance in the software development field. Automated test generation aims to streamline the testing process, enabling developers to address software's multifaceted and evolving aspects with greater efficiency and consistency. This approach reduces the time required to create and execute test cases. It enhances the reliability of the tests, as they are less prone to oversights or inconsistencies that can arise in manual processes. Through automated GUI test generation, developers can maintain rigorous quality control standards, ultimately supporting delivering high-quality software applications that meet user expectations and functional requirements.

This section examines approaches introduced over recent years, all aimed at automating the creation of test cases that effectively capture a wide range of GUI interaction scenarios. These automatically generated test cases play a dual role: they significantly enhance test coverage while also serving as powerful tools for detecting hidden errors and flaws within the application. For testing professionals, this automated approach provides a more efficient and exhaustive method of identifying latent issues, ultimately contributing to improved software quality and reliability. Figure 1 illustrates the general workflow involved in GUI test generation.

The workflow illustrates the iterative process of automated GUI test generation, starting with

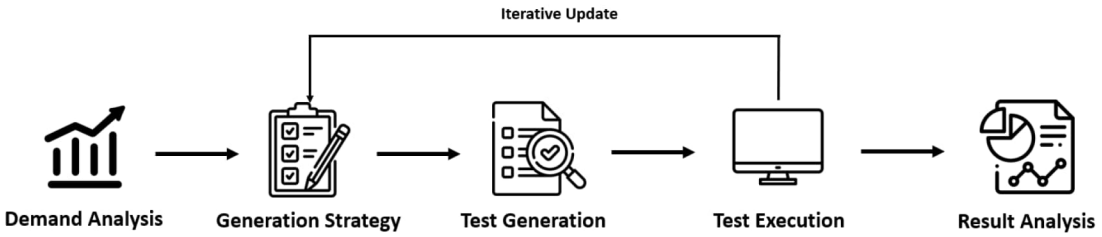


Figure 1: General workflow of GUI test generation.

Demand Analysis, where testing requirements are identified based on system needs. These insights inform the Generation Strategy, where a detailed plan for generating test cases is formulated. This strategy is then implemented during the Test Generation phase, creating specific test cases. The generated tests proceed to the Test Execution stage, where they are run against the application to evaluate its behavior. Results from the execution are processed in the Result Analysis phase, where findings are reviewed for accuracy and insights. Based on the analysis, iterative updates refine the test generation strategy to improve the process, ensuring the system meets evolving requirements effectively and efficiently.

GUI test generation methodologies can be categorized into four primary approaches: random-based, model-based, system-based, and learning-based[51]. Each approach brings distinct methods for generating test cases tailored to different testing needs.

Random-based approaches utilize test generation by randomly selecting input values or sequences of operations. This method is particularly effective for uncovering unexpected exceptions and identifying boundary conditions within a system. Additionally, random-based strategies enable rapid and comprehensive testing, allowing developers to assess system responses across a broad range of scenarios without predefined parameters. Owing to its inherent randomness, random testing proves valuable in identifying issues that application developers may not have anticipated. This approach facilitates the discovery of previously unknown potential problems, thereby enhancing the robustness and reliability of the application. Zeng et al. [53] conducted an industry case study on the widely used Monkey tool. This study applied random-based GUI test generation to WeChat and highlighted several limitations of the Monkey tool when deployed in an industrial environment. While random testing effectively explores a wide range of possible test paths, it also introduces the issue of test case redundancy. The lack of inherent structure in this approach often results in the generation of test cases that are imprecise and insufficiently targeted, diminishing their overall accuracy and efficiency in identifying specific issues within the system.

Model-based strategies use abstract models to depict the interactions, behavior, or structure of

an application under test. Test cases are systematically created using these models. Code coverage and path analysis were examples of static and dynamic analysis approaches that were the mainstay of early model-based test-generating systems. Although these techniques work well for identifying simple mistakes, they usually don't produce test cases with extensive coverage, which restricts their ability to assess complicated systems fully.

System-based techniques for creating test cases are gaining popularity, especially in extensive application projects since the complexity of mobile applications necessitates a more significant focus on system-level issues in GUI test-generating techniques. With an emphasis on tracking system-wide changes and interactions, this method thoroughly evaluates the application's functionality, performance, and security. System-based testing provides a strong foundation for assessing the stability and integrity of the program overall since it can be used in a range of scenarios, such as sensor data leaks, crash testing, and code update impacts.

Learning-based techniques have been increasingly included in many testing methodologies due to the quick development of machine learning and deep learning technologies and the increasing complexity of application systems. As a result, learning-based methods for creating GUI tests have become more popular. These techniques use ongoing data collection to improve and optimize GUI test production over time, enabling them to adjust to applications' changing features and structure. This flexibility improves GUI testing's long-term efficacy. Additionally, as GUIs have advanced in sophistication, GUI images include a wide range of information, such as layouts, color schemes, icons, and other visual components. Learning-based techniques overcome the drawbacks of conventional code and text analysis methods by enabling vision-based GUI testing.

2.2.2 GUI Test Record & Replay

The main goal of all the recording and replaying techniques is to capture user interactions with the graphical user interface. To verify how the software behaves in various situations, these exchanges are captured as test scripts that may be played back. Introducing such techniques has significantly simplified creating test scripts by reducing the technical skill requirements for testing experts and enabling non-technical users to participate in GUI testing. The main workflow of the GUI test record-and-replay procedure is shown in Figure 2 [51].

This workflow begins with Script Record, where user interactions with the application are captured to create a test script. The recorded script then moves to the Saving and Editing phase, where it can be stored and modified as needed to address specific testing scenarios. Next, the script is executed during the Script Replay stage, simulating the recorded interactions on the application. Finally, in the Verification and Reporting phase, the test results are analyzed to verify application behavior, and detailed reports are generated to document any identified issues or confirm successful

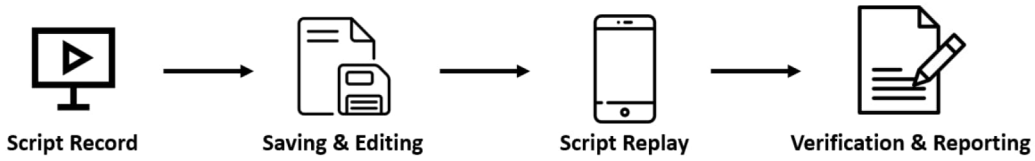


Figure 2: General workflow of GUI test record and replay.

outcomes.

Initially, the primary purpose of GUI test record-and-replay was to support regression testing within the same application and platform environment, typically assuming similar external conditions and overlooking the issue of mobile app fragmentation [46]. These approaches have been refined to minimize the number of recorded events, manage replay errors through image processing, and integrate GUI exploration with record-and-replay functionalities. However, the increasing variety of mobile devices and platforms has highlighted the fragmentation problem, necessitating advancements in GUI test record-and-replay methods to facilitate cross-device and even cross-platform migration.

Cross-device approaches aim to address variations across different types of devices—such as smartphones, tablets, and desktops—as well as across brands (e.g., Huawei, Samsung, Xiaomi) and OS versions (e.g., Android 7, Android 9, Android 10) within the same operating system environment. This enables developers to create test scripts on one device and replay them on various other devices within the same OS to verify app functionality across device types.

Cross-platform approaches, in contrast, extend the capability to record and replay across disparate operating systems, such as Android, iOS, and Web-based platforms. This allows the creation of test scripts on one OS and their deployment across different platforms to ensure app consistency and compatibility [51].

Additionally, with the growth of mobile applications, there is growing interest in extending test migrations to different applications with analogous functionalities. Cross-app approaches facilitate record-and-replay functionality between different applications, particularly when apps share similar features. This capability enables capturing and simulating user interactions across multiple apps, making it helpful in validating standard scenario-based tests across diverse applications.

2.2.3 GUI Testing Framework

In the progression of GUI testing, while targeted research focusing on specific components such as test script debugging, test generation, and expected outcome evaluation has yielded valuable insights, practitioners increasingly seek a holistic framework that allows for the independent execution of the entire GUI testing process. Rather than manually integrating disparate testing procedures, professionals are inclined toward a comprehensive solution that streamlines the process from start to finish. Consequently, this section on testing frameworks presents a selection of notable frameworks developed in recent years, each designed to provide a cohesive, all-in-one approach that supports the efficient execution of the complete GUI testing workflow. Figure 3 [51] illustrates the general workflow of a GUI testing framework.

Figure 3: General workflow of GUI testing framework.

This workflow begins with Demand Analysis, where testing requirements and objectives are identified based on the system's specifications and user needs. Next is the Test Generation phase, where specific test cases are developed to address the identified requirements. These tests are then executed and verified during the Execution and Verification stage to ensure the application's functionality and performance meet expectations. Following this, the Test Maintenance phase focuses on updating and refining test cases to accommodate changes in the application. Finally, the Coverage Evaluation step assesses the extent and effectiveness of the testing process, ensuring that all critical components and scenarios have been adequately addressed.

GUI testing frameworks are developed to automate and streamline the process of GUI testing for mobile applications. These frameworks enable testers to create, execute, and manage test cases to verify the functional accuracy of the app's GUI, identifying potential issues while ensuring the app's stability, functionality, and performance. By incorporating features such as recording and replaying, script automation, and cross-platform and cross-browser compatibility, GUI testing frameworks assist development teams in delivering a high-quality user experience.

From a technical standpoint, GUI testing frameworks can be classified into several types based on traditional methodologies.

- Module- or library-based frameworks focus on encapsulating operations across different modules of the tested application, or they employ a library structure to support cross-module business functionalities.
- Data-driven frameworks distinguish data from test cases, facilitating rapid generation of test cases by varying input data independently.
- Keyword-driven frameworks utilize a tabular or structured format for test case design, translating specific keywords into executable functions.
- Behavior-driven development frameworks prioritize testing based on clearly defined expected behaviors, aligning development and testing with the application's functional requirements.

2.2.4 Element Detection in GUI Testing

This section focuses on two primary areas: GUI element detection and GUI testing evaluation criteria. Image recognition technology, for instance, aids automated testing tools in accurately identifying and interacting with elements on an app's GUI during the testing process. Meanwhile, testing evaluation assists developers in efficiently selecting suitable testing methods. Together, these auxiliary techniques hold the potential to significantly improve the efficiency and precision of GUI testing, ultimately reducing the workload for app developers.

The role of GUI element detection and localization is fundamental in the creation, automated generation, and execution of test scripts. In recent years, various tools and algorithms have been developed to enhance the accuracy of identifying and locating diverse GUI elements—such as buttons, text fields, and drop-down menus—by leveraging technologies like image recognition and positioning algorithms. These tools and algorithms facilitate test generation and record-and-replay processes, empowering testing professionals to efficiently manage and test various mobile applications.

With the rapid expansion of mobile applications, the visual content within these apps has become increasingly complex and central to the testing process. Testers frequently engage in GUI element detection to effectively capture essential information. In GUI test record-and-replay processes, it is often necessary to record the specific GUI elements involved in user actions during the recording phase and to match replay interface elements with recorded interface elements during the replay phase. V2S, developed by Bernal-Cárdenas et al. [20], leverages computer vision technology. It utilizes advanced object detection and image classification techniques to identify and categorize user actions recorded in video format, transforming them into replayable test scenarios.

In the field of GUI testing frameworks, specific automated testing methods frequently depend on the detection of visual information. Yu et al. [52] propose an automated GUI testing approach that leverages image recognition to identify components within GUI images and simulates input signals to

the System Under Test through various input devices. Additionally, element recognition techniques are commonly employed in GUI test report analysis, where test reports often include substantial screenshot data. For instance, Yu [48] introduced CroReG, a tool that utilizes image understanding technology to generate crowdsourced error reports by analyzing error screenshots submitted by crowdsourced workers. Consequently, GUI element detection has emerged as a significant auxiliary technique within GUI testing, generating substantial research.

Chen et al. [9] present an innovative approach to GUI element detection that integrates traditional computer vision techniques for detecting non-text element areas with deep learning models. This combination effectively capitalizes on the strengths of both methodologies, resulting in highly accurate detection outcomes. Figure 4 provides examples illustrating the application of the algorithms described in [9] and [50].

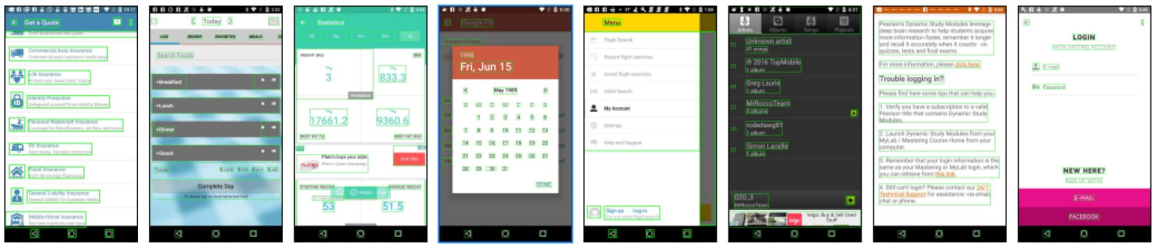


Figure 4: Element detection with the algorithm in [9] and [50]. This figure is taken from Figure 10 of [51].

2.2.5 Challenges of Vision-based GUI Testing

This section explores the challenges associated with vision-based GUI testing. Key topics include GUI widget detection and semantic understanding and the application of large language models in GUI testing.

GUI widget recognition is a fundamental aspect of vision-based GUI testing, enabling the accurate localization and interaction with GUI elements such as buttons, text boxes, and menus. Despite its importance, challenges persist, including adapting to diverse GUI frameworks, dynamic and asynchronous elements, varying resolutions, and display issues like occlusion and overlap [50]. Current recognition techniques—attribute-based, image-based, and AI-based—each face limitations. Attribute-based methods rely on stable attribute values in layout files but struggle with unstable or duplicated attributes, leading to errors [55]. Image-based methods are affected by image quality, similarity, and deformation, resulting in mismatches. AI-based approaches, while promising, require significant labeled data and computational resources. Additionally, integrating GUI widget recognition into testing tools remains complex, requiring careful consideration of compatibility and ease of

integration for end-to-end automation [37].

Semantic analysis of GUI widgets focuses on understanding their meaning, functionality, and interrelationships to enable intelligent test case generation, execution, and verification. However, this process is complicated by the dynamic and asynchronous nature of GUI elements, platform variability, and the inherent ambiguity in user interactions. Methods such as rule-based, model-based, and learning-based approaches have been explored, but they face challenges, including difficulty covering all scenarios, handling exceptions, and adapting to evolving user requirements and feedback [49].

Opportunities for innovation lie in leveraging multimodal fusion to process diverse GUI information simultaneously and establish connections between elements. Knowledge graph technology can map GUI elements and their relationships, enhancing testing intelligence when combined with AI techniques like recommender systems and dialogue systems. Furthermore, big data and cloud computing provide resources for collecting diverse training samples and accelerating model training and inference, meeting real-time and large-scale testing demands. These advancements can significantly improve the robustness, adaptability, and efficiency of GUI testing frameworks.

Large Language Models such as GPT and LLaMA, are high-capacity models with extensive parameters and sophisticated architectures, excelling in both natural language processing (NLP) and multimodal tasks like image classification and segmentation. These capabilities enable LLMs to analyze and understand app GUIs, thereby supporting automated GUI testing. However, challenges remain in their application to GUI testing. LLM outputs, often complex, can be difficult for automated systems to interpret. Additionally, LLMs' generalized pre-training may not meet the domain-specific requirements of GUI testing. Effective test case design—ensuring high coverage, diverse behavior capture, and alignment with expected GUI outputs—presents another significant challenge.

LLMs offer transformative potential for GUI testing by enhancing test generation, script maintenance, and report analysis. For instance, LLMs can automate the generation of high-quality test cases, scripts, and reports, reducing manual effort and errors [13, 31]. Traditional GUI testing separates test exploration and script-based testing migration, but LLMs blur these boundaries by enabling automated exploration to generate logically robust and migratable test scripts. These capabilities extend to scenario-based testing, including recognition, segmentation, and understanding, though practical applications still require script maintenance and optimization.

Additionally, LLMs' multimodal fusion capabilities combine visual and textual data to enhance the depth and accuracy of GUI testing. By analyzing screenshots and user feedback, LLMs can identify layout confusion, color inconsistencies, unclear fonts, functional abnormalities, and performance degradation in GUIs. For example, GPT-4 demonstrates strong caption generation capabilities for app GUI screenshots, surpassing non-LLM models. These abilities enable LLMs to identify and

describe GUI contents precisely, thereby improving defect detection.

2.3 Conclusion

This chapter has reviewed foundational concepts in GUI testing, including the importance of GUI rendering in software development and the transition from manual to automated methods. It reviews literature on automated GUI testing methods such as random-based, model-based, system-based, and learning-based approaches, as well as GUI test record-and-replay techniques and testing frameworks. Vision-based GUI testing and advanced technologies like Large Language Models are highlighted as innovative solutions addressing modern challenges. These methodologies enhance efficiency, adaptability, and test coverage, offering robust frameworks for maintaining mobile application quality and user satisfaction.

Chapter 3

Infer the GUI Rendering State

3.1 Introduction

This chapter discusses the methodology to be used to achieve the objectives of the thesis. The objective of the research is to come out with a robust system that will classify fully and partially rendered GUI screens in mobile applications. Accurate classification of rendered GUIs will ensure that the system under test is fully loaded before any automation test starts running. This ensures that testing results are accurate and true to the real state of the application, hence giving better reliability in the automated testing framework of mobile applications.

Our approach leverages deep learning and computer vision by utilizing powerful pre-trained models and methods of transfer learning. In this regard, the backbone of this methodology will be supported by the Rico dataset, which is one of the largest and most diverse collections of GUI screens from real-world mobile applications. The Rico dataset allows us to develop a robust classification model because it includes a rich variety of visual features. This thesis thus proposes, with regards to ensuring accuracy and the ability of our approach for generalization, a multi-step approach entailing dataset preparation, fine-tuning, and comparative analyses for a variety of state-of-the-art deep learning models.

The first step in our methodology is to evaluate the performance of the Structural Similarity Index (SSIM) as a baseline tool for classifying GUI states. The goal is to determine if SSIM can effectively distinguish between fully and partially rendered GUIs within the Rico dataset. This step also helps set a performance benchmark against which more sophisticated methods will be compared later.

With the dataset ready, we finetune Vision Mamba, a competitive computer vision model that has achieved various state-of-the-art performances on benchmark image classification tasks. We decided

to use Vision Mamba since the model is able to handle complex visual features, which makes it an eligible candidate for classifying both fully and partially rendered GUIs. In this work, we thoroughly investigate the performance of Vision Mamba in recognizing rendered GUIs and gauge its potential to be applicable in mobile app testing systems.

To further validate our approach, we evaluate Vision Mamba's performance against two other popular large-scale models: Visual Transformers (ViT) and ResNet. While Visual Transformers adopt attention mechanisms that enable their focus on image regions, residual connections in ResNet allow deeper network architectures without suffering from vanishing gradients. With all the aforementioned settings, we find the most suitable architecture for GUI classification under similar conditions.

Consequently, through such comparisons of the discussed models on a common dataset, we hope to understand many aspects: how accuracy is traded within the models for computational efficiency and implementability. This would give insight into how best to develop an effective lite system that classifies GUI renders states to serve the interests of reliable but efficient automated testing of more mobile applications.

Therefore, the subsequent sections explain in detail each step involved in this methodology: GUI rendering, dataset preparation, and model fine-tuning. We follow this structure to develop a classification system that will obviously outperform traditional methods—like SSIM—but also show practical benefits in real application testing for mobile applications.

3.2 GUI Rendering and Testing

The impact brought about by GUI rendering has been neglected. GUI rendering is the process of generating and displaying a visual frame of an application on a screen. It involves a number of tasks such as page transitions, loading resources from online sources, and conversion of user interface components into pixel-based visuals. These all act upon a structured view hierarchy to properly layer and organize UI elements in a correct manner, as shown in Figure 5. Red bars denote imperfect throttles which inefficiently stagnate on GUIs, or test in partially displayed states, where green bars denote the perfect throttle.

This can take considerable time, based on the code quality of the application under test, the device's performance, and the internet connection speed. Automated testing usually uses some kind of fixed delay, also known as a throttle, between the events to be performed because the GUI usually needs some time to fully render. Optimizing this throttle setting is important, for it will minimize the idle time that will be needed during an automated test, improving the efficiency of the testing process.

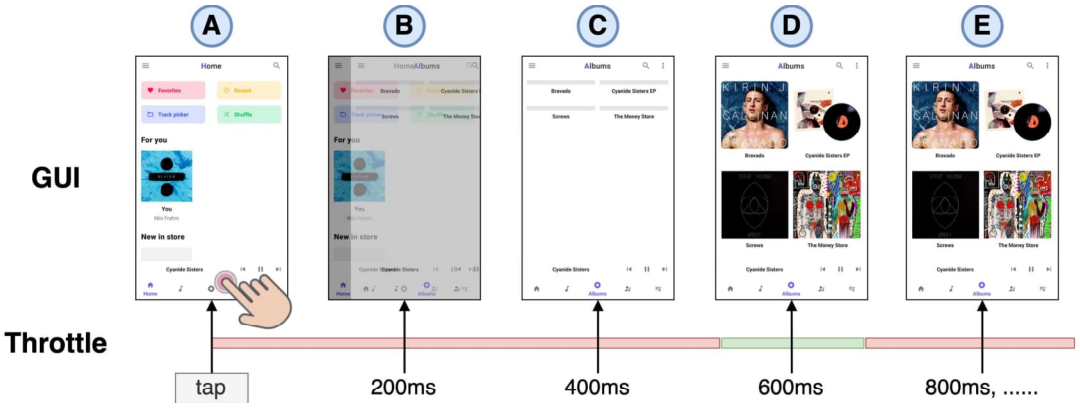


Figure 5: GUI testing that is automated using various throttles. The green bar indicate optimal throttle, while the red bars signify defective throttles that either test partially rendered states or inefficiently stagnate during GUI operations. This figure is taken from Figure 1 of [14].

A fixed throttle may not work equally well for different testing tools, devices, or even different pages of the same application since the complexity of rendering each GUI is different. Whereas a longer throttle will ensure that GUIs are fully rendered, it may lead to inefficiencies by prolonging the testing process with unnecessary idle waiting times, such as those shown in Figure 5-E. Conversely, a too-short setting of throttle would result in partially rendered GUIs, defeating the effectiveness of testing for mainly two reasons.

First, most of the GUI testing tools rely on the visual appearance of the GUI to work correctly. Examples include tools for usability bug detection [30], robot-based testing [37], reinforcement learning-driven application exploration [1], and cross-platform test case migration [41]. All these processes require a fully rendered GUI as input if their results are to be accurate.

Second, the runtime view hierarchy usually does not match the one that is rendered, which introduces mismatches. This can result in a few actions that depend on the view hierarchy to lead to not being executed, which reduces the amount of test coverage. For instance, querying the view hierarchy file for the "Screws" image by coordinates and attempting to tap on it fails when the GUI has not yet been fully rendered, as in Figure 5-C.

To overcome these challenges, an adaptive throttle—such as a 600ms delay shown in Figure 5—can provide a practical trade-off between testing effectiveness and efficiency. To study throttling challenges in automated GUI testing tools, a pilot study on three widely used testing frameworks has been conducted by evaluating their performances on 32 applications [14]. The key aim was to investigate the GUI rendering for various throttle settings. Results indicate that, for a fixed short throttle interval, for example, 200ms, about 24% of events occur while in a partially rendered state. In addition, most of the partially rendered states are transition states, loading states,

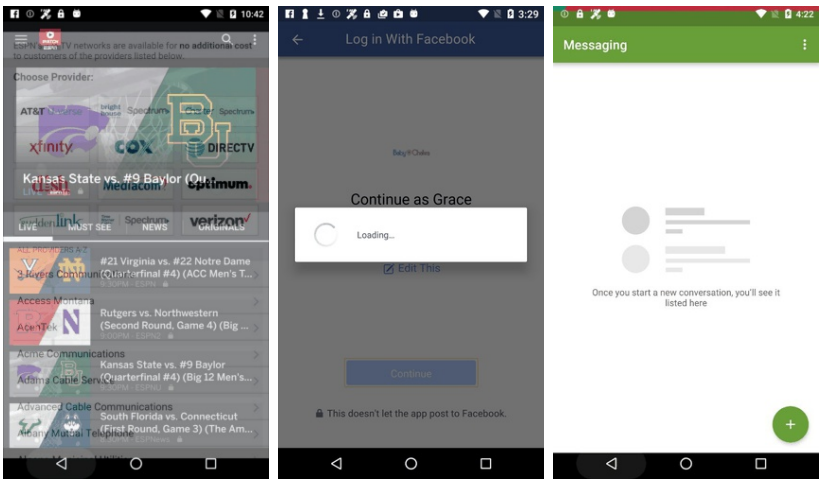
and implicit loading states.

While increasing the throttle interval can reduce problems related to partially rendered states, very long throttles drastically lower testing efficiency. For example, these kinds of throttle intervals reduce testing events by as much as 52.8% during automated exploratory testing, undermining overall effectiveness in the testing process. These results highlight the critical need for balanced throttle configurations to optimize the accuracy and efficiency of automated GUI testing.

3.2.1 Categorizing GUI Rendering State

Fully Rendered State. This means a state where the GUI is loaded, with its associated resources completely displayed and operating. In such a state, it will make sure that the interface is totally functional and complete in a visual manner, which a user or any automation tool in testing may use without having anything like a loading or a transitional indicator showing up.

Transiting State. Figure 6-a depicts how a state is transitioning to the next one, and in this process, two GUI states overlap since the time needed for transition is larger than the throttle interval set. Such overlaps may be due to basically two reasons: First, the throttle interval may be too small to get the GUI fully rendered. Second, the reason for longer rendering might be at the application level due to some development issues. These are issues such as an overdoing of animations or defects in hardware acceleration, which can cause these unexpected delays in rendering.



(a) Transiting state (b) Explicit loading (c) Implicit loading

Figure 6: Examples of partially rendered state.

Explicit Loading State. Figure 6-b represents an explicit loading state. If there is any visual indication, such as a spinning wheel, linear progress bar, or textual hints that show something has just started or is in an ongoing process—a process or rendering—then it is considered an explicit

loading state. Explicit loading states usually signify sensitive data operations, which involve account authentication, money transfers, and file uploads. During this state, the GUI is non-interactive so that no user actions disturb the processes in progress.

Implicit Loading State. Figure 6-c depicts an implicit loading state in which some resources are not displayed because of network latency or defects in the resources. Unlike the explicit loading state, whereby one can clearly identify a loading state, say, through the appearance of a loading bar, the implicit loading state should be deduced by the users or systems from context. In Figure 6-c, for example, gray placeholders or layouts of incomplete resources indicate that the resource contents have not been fully loaded.

3.2.2 Adaptive Throttle

One of the effective approaches to reduce problems with partially rendered GUIs is to increase the throttle interval. In such a way, the time between events would be longer, and thus most of the loading or transition processes would be successfully finished. An experiment has also conducted using Droidbot in order to investigate how different throttle intervals affect the performance of the testing tool. Accordingly, five different throttle intervals—200 ms, 400 ms, 600 ms, 800 ms, and 1000 ms—have been considered and analyzed for their impact on the working and test results of the tool [14].

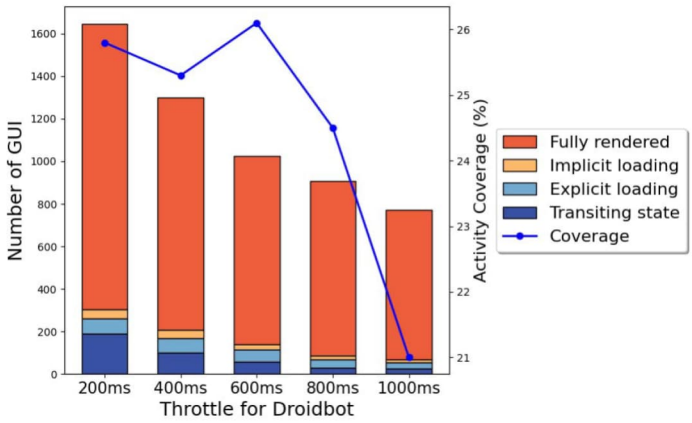


Figure 7: GUIs and activity coverage across various Droidbot throttle settings. This figure is taken from Figure 4 of [14].

Figure 7 depicts the number of GUIs versus the activity coverage results. In this case, the growing throttle intervals clearly reduced the partial rendering states. Concretely, incomplete rendering went from 17% at 200ms, 15%, 14%, and 9%, to 8% at 1000ms throttle intervals. In particular, the problem of transitioning states is considerably reduced, which would indicate that longer throttle intervals

facilitate smoother transitions and loading between events. However, extended throttle intervals also lead to a reduction in the total number of GUIs explored. For example, the number of GUIs decreases progressively with throttle intervals from 1,646 at 200ms to 1,299, 1,023, 907, and 776 at 1000ms, reflecting fewer events executed during runtime.

Therefore, the activity coverage always decreases except for 600ms throttle interval. This makes sense since overly aggressive testing, like 200 or 400ms, will lead to most states being partially rendered and thus testing becomes inefficient. On the other hand, when throttling at the medium pace, such as 600ms, the tool can interact more with fully rendered GUI and therefore explores more activities. However, with higher intervals, for instance, 800ms or 1000ms, the testing will be too slow and therefore inefficient. An optimum throttle interval has to be chosen for effectiveness and efficiency.

These results have highlighted the need for throttle settings to be an integral part of automated testing, pointing toward developing a balanced approach that optimizes effectiveness and efficiency. Automated testing tools must wait until the GUI finishes rendering before continuing to the next events, especially when the application is mostly idle. This stresses the need to develop a dynamic method for adjusting throttle settings during testing. The main challenge is how to tell the difference between a partial and full rendering of the GUIs. Given that such differentiation can easily be performed by a human observer, the approach proposed here is to apply visual cognitive methods for identifying GUI rendering statuses. Image-based approaches are versatile and easy to deploy, assuming that GUI screenshots would normally be available in most automated testing toolsets.

3.2.3 AdaT Approach

AdaT proposes a simple but effective approach that adaptively adjusts the throttle according to GUI screenshots. Since each automated testing tool operates directly on the device, GUI screenshots are synchronously captured to have it detect the current rendering state. Based on the inference mentioned above, AdaT schedules testing events once the GUI is fully rendered and waits for its rendering otherwise. Figure 8 [14] provides an overview of the AdaT framework.

The core of AdaT is on the lightweight CNN-based model that classifies GUI rendering state in three key phases: Data Preparation, GUI Rendering State Classification with the CNN-based model, and Deployment of the model. A large-scale dataset including partially or fully rendered GUIs shall be automatically collected in the first stage. The second stage uses the designed CNN-based model to accurately self-determine the current rendering status.

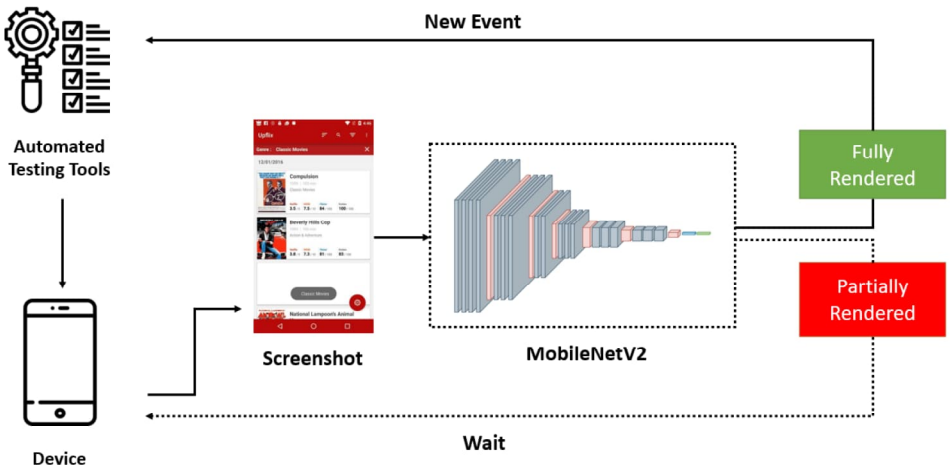


Figure 8: Overview of AdaT approach.

3.3 Data Preparation

Extensive datasets are the backbone for understanding GUI rendering states and successfully training deep learning models. However, this is quite expensive because of the manpower needed to label such data. To alleviate this, this phase will aim to automatically collect partially rendered and fully rendered GUIs by utilizing GUI transition screencasts. The purpose is to reduce manual efforts while keeping the datasets powerful enough to train a model. In the subsequent sections, the utilized dataset and the approach used to label the dataset automatically are depicted. However, another question regarding the appropriation of this method will be answered.

3.3.1 RICO Dataset

RICO dataset [11] is one of the most extensive, publicly available datasets of graphical user interfaces for mobile applications. It acts as a fundamental instrument for advancements in research in the realms of human-computer interaction, machine learning, and user experience design. RICO was specifically made to meet the ever-growing demands of structured and scalable datasets within these domains by providing an elaborative set of mobile app screenshots, semantic information, and traces of interaction.

RICO allows researchers to study mobile application behavior, such as finding patterns in design and training algorithms to guess user interactions, by capturing diverse app categories and interface designs. This dataset now includes not only the visual information of interfaces but also detailed metadata for structural attributes, which gives due insight into the hierarchical representation of

UI components. It marked something of a turning point in the study of mobile applications, finally allowing large-scale studies that were previously impossible due to a lack of comprehensive data.

Rico includes 44,418 transition screencasts sourced from over 9,700 distinct Android applications spanning 27 app categories. The screencasts vary in duration, ranging from 0.5 to 50 seconds, and each contains one or more user actions (e.g., tap, scroll) interspersed with periods of inactivity. In this dataset, scrolling actions during transitions are excluded due to their inherent ambiguity. For instance, scrolling within a lazy-loading GUI may result in capturing partially rendered states, whereas scrolling within a pre-loaded GUI may produce fully rendered states. After these adjustments, the dataset is refined to include 36,038 transition screencasts.

The RICO dataset comprises screenshots of user interfaces from Android applications, spanning a wide range of categories such as social media, finance, travel, and gaming. Each screenshot is accompanied by:

- **View Hierarchies:** These provide a tree-like structure of all UI components, detailing their spatial relationships, sizes, and visibility.
- **Semantic Information:** Labels, attributes, and metadata that describe the functionality of each UI element, such as buttons, text fields, and images.
- **Interaction Traces:** User interaction paths that show how users navigate through the application, offering insights into typical usage patterns.

Additionally, the app-level metadata in the dataset include category labels and text descriptions. Such a versatile set of data elements provides the ground for multifarious research opportunities, from unraveling user behavior to understand the training of models for classifying UI elements and predicting the usability of an application interface. The breadth of variety in categories ensures that there is representativeness of real-world, mobile applications, hence enhancing the results' generalization derived from it.

The RICO dataset is one of the keystones in this thesis for addressing challenges in the fields of automated mobile application testing and GUI rendering state classification, in particular, the classification of fully rendered versus partial GUIs to enable a proper performance evaluation of a mobile application. The RICO dataset provides a foundational ground truth for this.

This dataset is a valuable tool for research, but it does have certain limitations. One key drawback is its exclusive focus on Android applications, which reduces its relevance for studies involving iOS or cross-platform analysis. This platform-specific design limits the dataset's ability to support broader, comparative research across multiple operating systems.

Another notable limitation is the dataset's extensive annotations, which, while useful, may fall short in providing the semantic or contextual details required for specific tasks like classifying GUI

rendering states. As a result, researchers may need to engage in extra manual labeling or data preprocessing to make the dataset suitable for these more specialized studies. These challenges point to the necessity of additional data sources or adjustments in research methods to fully leverage the potential of the RICO dataset in more comprehensive research settings.

The RICO dataset has been widely adopted in various domains, highlighting its versatility and importance. Notable applications include:

- **UI Design Analysis:** Identifying design patterns, usability issues, and best practices for mobile app development.
- **Machine Learning:** Training models for tasks such as UI element classification, sequence prediction, and synthetic UI generation.
- **Accessibility Improvements:** Analyzing design patterns to enhance accessibility for users with disabilities.
- **Automated Testing:** Developing tools to test app performance and functionality based on UI layouts and interaction flows.

3.3.2 Transiting Frame Identification

A GUI transition comprises fully or partially rendered sequences of frames. Image processing for finding out the rendering state for every frame of a transitioning screencast employs the Y-Difference or, in short, Y-Diff. This computation measures a perceptual similarity score for two successive frames using the color space of YUV, normally utilized for encoding video. Unlike RGB representation, YUV is similar to how this type of transmission error, akin to compression artifacts, would hide. It is also the closeness to human perceptual efficiency [8], [40]. Y-Diff maps an image's Y (luminance) value difference in a UV color space, commonly becoming one of the major inputs concerning human perception of motion conditions [32].

A transitioning screencast can be represented as a sequence of frames, $f_0, f_1, \ldots, f_{N-1}, f_N$, where f_N denotes the current frame and f_{N-1} the preceding frame. To compute the Y-Difference (Y-Diff) between the current frame f_N and the previous frame f_{N-1}, luminance masks Y_{N-1} and Y_N are extracted by converting the RGB color space into the YUV color space.

Subsequently, the Structural Similarity Index (SSIM) [45], a perceptual comparison metric, is applied to determine the pixel-wise similarity. SSIM evaluates differences based on the local mean, variance, and correlation of luminance values. The resulting SSIM score ranges from 0 to 1, with higher values indicating greater similarity between the frames.

To determine whether a frame is fully or partially rendered, the similarity scores of consecutive frames within the transitioning screencast are analyzed, as illustrated in Figure 9. The initial

step involves grouping frames that belong to the same atomic state through a customized pattern analysis. This step is crucial because discrete states displayed on the screen persist across multiple frames and must be appropriately grouped and segmented.

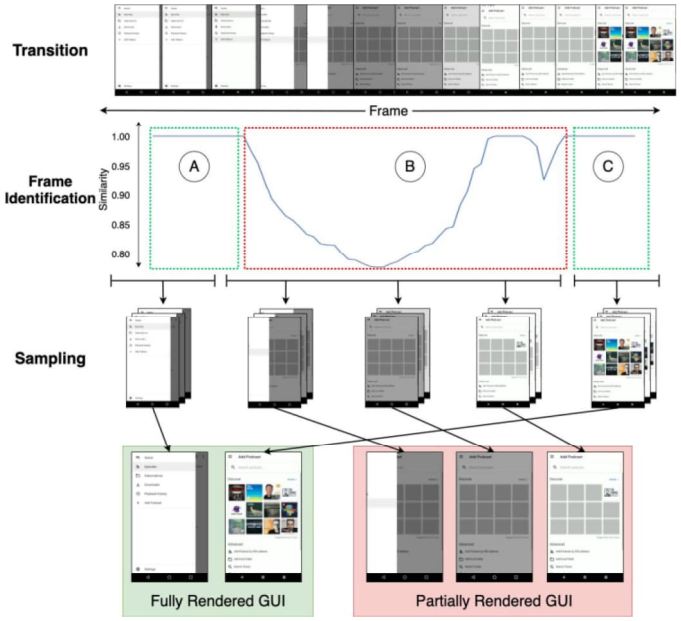


Figure 9: Pipeline for automated data collection that is used in ADAT. This figure is taken from Figure 6 of [14].

The findings indicate that fully rendered GUIs exhibit a steady state characterized by consecutive frames that remain identical or highly similar over an extended duration. Examples of such states are shown in Figure 9 (A) and (C).

In contrast, partially rendered GUIs exhibit significant differences between consecutive frames, indicating a rapid transition from one screen to another. For instance, Figure 9 (B) illustrates a scenario where a button click initiates the fading out of the current GUI. During this transition, the similarity score decreases sharply. Subsequently, as the next GUI fades in, the similarity score increases. The observations reveal that a typical characteristic of partially rendered GUIs is a brief stabilization of the similarity score between two sharp declines, as depicted in Figure 9 (B).

The brief, steady duration observed is attributed to resource loading within the GUI, consistent with the earlier observation of the implicit loading state. Two thresholds have been empirically established to differentiate between fully rendered and partially rendered GUIs: a similarity score of 0.992 to determine whether two frames are similar and a sequence of five consecutive frames to signify a steady state.

3.3.3 Analyzing SSIM for Auto-labeling

The Structural Similarity Index Measure (SSIM) is a widely used metric for assessing image quality by comparing structural information between images. While SSIM effectively measures differences in luminance, contrast, and structure, it has notable limitations that render it inadequate for distinguishing between fully and partially rendered mobile application screenshots. Therefore, this section reviews the reasons for SSIM's inappropriateness for auto-labeling the dataset.

First, SSIM relies on the assumption that the human visual system is highly adapted for extracting structural information from the images. On the other hand, partially rendered images include complex and various artifacts related to missing elements or incomplete layouts that may not affect structural components in the way SSIM checks. The consequence may, therefore, be a high similarity score using SSIM, while such partially rendered images may be incomplete for human observers [36].

Second, SSIM is only sensitive to local luminance and contrast changes; therefore, misleading assessments can arise. Indeed, many partially rendered images have irregularities that, in many cases, do not affect the general luminance or contrast; the SSIM will most likely not pick up those critical differences and might not recognize partial rendering issues. This is a serious limitation for mobile application interfaces since users must depend on a fully and correctly rendered presentation of all interface elements [36].

Also, the fact that SSIM is based on pixel-wise comparisons significantly influences the tendency to misinterpret complex visual content. Dynamic content in screenshots, such as animations or other interactive elements, may induce variations in SSIM, which then misinterprets them as structural differences, though the rendering is complete. Such misinterpretation may lead to false positives where fully rendered images are wrongly classified as partially rendered [35].

Additionally, SSIM is not suited for changes in scale, rotation, and similar geometric transformations of the images; hence, in cases where such transformations are introduced between native app interfaces, it will also not be reliably applicable. The partial rendering of views might include elements unaligned or unscaled appropriately; hence, the design of SSIM could not consider such geometric differences [35].

Whereas SSIM provides a basic measure for the structural similarity of two images, inherent limitations—like being irresponsive against some artifacts and frequently being fooled about dynamic content and geometrical transformation—make SSIM an intuitively naive and unreliable method for segmentation among fully and partially rendered mobile applications. Therefore, this type's limitations nurtured our interest in alternative directions, namely manual annotation of a small dataset to fine-tune a large deep learning model previously trained on a large dataset. Such an approach is further detailed in the next section.

3.4 Model Fine-tuning

In this section, three cutting-edge deep learning models are introduced: Vision Mamba, Vision Transformers, and ResNet. All of them have been pre-trained on large-scale datasets. Later, they were fine-tuned with a small dataset that was labeled manually. Subsequent sections discuss each model and its implementation details.

3.4.1 Vision Mamba

Recent research has been very instrumental in bringing state space models into the spotlight. Coming off from the pioneering work of the Kalman filter model [22], the more recent SSMs exhibit an impressive ability to deal with long-range dependencies and have benefited from parallelized training. Innovative approaches such as linear state-space layers (LSSL) [17], structured state-space sequence model (S4) [16], diagonal state space (DSS) [19], and S4D [18] are all models designed to process sequential data much more effectively across diverse tasks and modalities by design, focusing on modeling power with extended-range dependencies. State space models are efficient at handling very long sequences using convolutional and near-linear computations. Methods like 2-D SSM [5], SGConvNeXt [27], and ConvSSM [39] integrate SSM frameworks with CNNs or Transformer architectures for the efficient processing of two-dimensional data.

Recent developments, such as Mamba [15], further extend the time-varying parameters to state space models (SSMs) and develop a hardware-optimized algorithm that significantly boosts the efficiency of both training and inference. The scaling behavior of Mamba thus has very good competitiveness or is even preferred over Transformers when applied for language modeling. On the other hand, a systematically thorough SSM-based network as a backbone for an effective treatment of the visual information embodied by both images and videos is yet to be provided.

ViTs have been performing commendably well in learning representation from visual information, be it large-scale self-supervised pre-training or showing their best on downstream tasks. Compared to CNNs, the key advantages of Vision Transformers (ViTs) are that they provide every patch of an image with a data- or patch-dependent global context through self-attention mechanisms. This is in direct contrast to the CNNs, which apply identical parameters, such as convolutional filters, equally at all positions.

Another important advantage of ViTs is that they reflect a modality-agnostic modeling strategy: they treat an image as a sequence of patches without inductive bias in two dimensions. The latter property has positioned ViTs as one of the popular architectures for multimodal applications [6], [25], [29]. While powerful, the Transformer self-attention mechanism also brings up limitations regarding speed and memory efficiency. Particularly, this becomes an issue in cases requiring long-range visual

dependencies—for example, high-resolution images processing.

Following the success of Mamba in language modeling, a strong extension of this success is foreseen in the sphere of vision. This, in particular, involved the design of a generic and efficient visual backbone with the advanced state space model approach. Despite its merits, Mamba has two major barriers: unidirectional modeling and no positional awareness. Limitations are overcome with the Vision Mamba model to combine strengths from both: continuous state space models, thus modeling data-dependent global visual contexts in a bidirectional manner, and position embeddings capture location-aware visual recognition.

The input image is divided into patches that are linearly projected into vectors for the processing of the Vision Mamba model. The image patches are treated as sequential data in the Vim blocks where the proposed bidirectional selective state space will efficiently compress the visual representation. Also, the position embedding in the Vim block helps to give the location-based information that allows the model Vision Mamba (Vim) to be much more robust for dense prediction tasks. Currently, the model is at a stage where Vision Mamba is trained on a task involving supervised classification of images using the ImageNet dataset.

Pre-trained Vim forms the backbone for learning the visual representation sequentially to support various downstream dense prediction tasks like semantic segmentation, object detection, and instance segmentation. Like Transformers, the model can be pre-trained in an unsupervised way with large-scale visual data by introducing the Vision Mamba-Vim. Using the greater efficiency of Mamba allows much economic pre-training for the Vision Mamba on a larger scale.

Comparatively, being among the approaches that are SSM-driven for vision tasks, a good deal of the design focuses on making the Vision Mamba purely SSM-based: that is, processing images in order and presenting it as a more viable way to develop a generic efficient backbone. As the first purely SSM-driven approach, Vision Mamba demonstrates remarkable performance in dense tasks with bidirectional compression models combined with positional awareness. When compared to the widely regarded Transformer-based model DeiT [42], the Vision Mamba (Vim) demonstrates superior performance in ImageNet classification tasks.

Besides the advantages in performance, Vision Mamba shows much higher efficiency in GPU memory and inference time when processing high-resolution images. This allows Vim to directly perform sequential visual representation learning without 2D priors, such as the 2D local window used in ViTDet [26], for high-resolution visual understanding tasks. Surprisingly, Vim outperforms DeiT in these scenarios.

3.4.1.1 Method Introduction

The primary objective of Vision Mamba (Vim) is to adapt the advanced state space model (SSM), specifically Mamba [15], for applications in computer vision. This section begins by outlining the preliminaries of SSM, providing foundational context. It then offers an overview of Vim, followed by a detailed explanation of how the Vim block processes input token sequences. Subsequently, the architectural specifics of Vim are elaborated, culminating in an analysis of the model's efficiency.

State space models (SSMs), such as structured state space sequence models (S4) and Mamba, draw inspiration from continuous systems that transform a one-dimensional function or sequence $x(t) \in \mathbb{R}$ into $y(t) \in \mathbb{R}$ through an intermediate hidden state $h(t) \in \mathbb{R}^N$. In this framework, $A \in \mathbb{R}^{N \times N}$ serves as the evolution parameter, while $B \in \mathbb{R}^{N \times 1}$ and $C \in \mathbb{R}^{1 \times N}$ act as projection parameters.

$$\begin{aligned} h'(t) &= Ah(t) + Bx(t), \\ y(t) &= Ch(t). \end{aligned} \tag{1}$$

S4 and Mamba represent the discrete adaptations of the continuous system, incorporating a timescale parameter Δ to convert the continuous parameters A and B into their discrete counterparts $\overline{A}$ and $\overline{B}$. A commonly employed transformation technique for this purpose is the zero-order hold (ZOH), which is defined as follows:

$$\begin{aligned} \overline{A} &= \exp(\Delta A), \\ \overline{B} &= (\Delta A)^{-1} \left(\exp(\Delta A) - I\right) \cdot \Delta B. \end{aligned} \tag{2}$$

After discretizing $\overline{A}$ and $\overline{B}$, the discretized version of Equation (1) with a step size Δ can be rewritten as:

$$\begin{aligned} h_t &= \overline{A} h_{t-1} + \overline{B} x_t, \\ y_t &= C h_t. \end{aligned} \tag{3}$$

At last, the models compute output through a global convolution.

$$\begin{aligned} \overline{K} &= (C\overline{B}, C\overline{AB}, \ldots, C\overline{A}^{M-1}\overline{B}), \\ y &= x * \overline{K}, \end{aligned} \tag{4}$$

where M is the length of the input sequence x, and $\overline{K} \in \mathbb{R}^M$ is a structured convolutional kernel.

3.4.1.2 Vision Mamba

Figure 10 provides an overview of the proposed Vision Mamba (Vim) model. The standard Mamba is optimized for one-dimensional sequences. To adapt it for vision tasks, a two-dimensional

image $t \in \mathbb{R}^{H \times W \times C}$ is first transformed into flattened 2-D patches $x_p \in \mathbb{R}^{J \times (P^2 \cdot C)}$, where H and W represent the image dimensions, C is the number of channels, and P denotes the size of the image patches. Subsequently, x_p is linearly projected into a vector of size D, and position embeddings $E_{\text{pos}} \in \mathbb{R}^{(J+1) \times D}$ are added as defined below:

$$T_0 = \left[t_{\text{cls}}; t_p^1 W; t_p^2 W; \cdots; t_p^J W\right] + E_{\text{pos}}, \tag{5}$$

where t_p^j is the j-th patch of t, and $W \in \mathbb{R}^{(P^2 \cdot C) \times D}$ is the learnable projection matrix. Drawing inspiration from Vision Transformers (ViT) [12] and BERT [24], the proposed Vim model incorporates a class token, denoted as t_{cls}, to represent the entire patch sequence. The token sequence T_{l-1} is then passed to the l-th layer of the Vim encoder, producing the output T_l. Finally, the output class token T_{L}^0 is normalized and passed through a multi-layer perceptron (MLP) head to generate the final prediction $\hat{p}$, as outlined below:

$$\begin{aligned} T_l &= \text{Vim}(T_{l-1}) + T_{l-1}, \\ f &= \text{Norm}(T_L^0), \\ \hat{p} &= \text{MLP}(f). \end{aligned} \tag{6}$$

where Vim is the proposed vision mamba block, L is the number of layers, and Norm is the normalization layer.

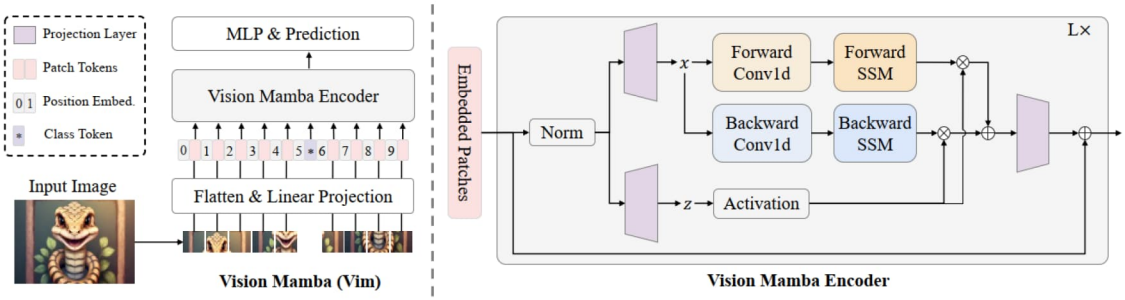


Figure 10: Vision Mamba model overview. The process starts by breaking an image into patches of relatively small size. Patches will be subsequently turned into patch tokens. In sequence, those tokens undergo the process imposed by the Vim encoder. For the task of classifying the image classes into ImageNet categories, on top of these tokens a learnable classification token was placed. While Mamba is strictly developed with support for text sequences only for its modeling, the Vim encoder processes token sequences in both forward and backward directions. This figure is taken from Figure 2 of [54].

The proposed Vision Mamba (Vim) model in Figure 10 operates as follows: the input image

is first divided into patches, which are subsequently projected into patch tokens. The resulting sequence of tokens is then fed into the Vim encoder. For ImageNet classification, an additional learnable classification token is appended to the sequence of patch tokens. Unlike Mamba, which is designed for text sequence modeling, the Vim encoder processes the token sequence bidirectionally, incorporating both forward and backward directions.

3.4.1.3 Vim Block

The original Mamba block was designed for one-dimensional sequences, making it unsuitable for vision tasks that demand spatial awareness. To address this limitation, the Vim block is introduced, integrating bidirectional sequence modeling tailored for vision applications. A detailed depiction of the Vim block is provided in Figure 10.

Algorithm 1 Vim Block Process

Require: token sequence $T_{l-1} : (B, M, D)$

Ensure: token sequence $T_l : (B, M, D)$

/* *normalize the input sequence* T_{l-1} */

$T'_{l-1} : (B, M, D) \leftarrow \text{Norm}(T_{l-1})$

$x : (B, M, E) \leftarrow \text{Linear}^x(T'_{l-1})$

$z : (B, M, E) \leftarrow \text{Linear}^z(T'_{l-1})$

/* *process with different direction* */

for o **in** {*forward*, *backward*} **do**

  $x'_o : (B, M, E) \leftarrow \text{SiLU}(\text{Conv1d}_o(x))$

  $B_o : (B, M, N) \leftarrow \text{Linear}^B_o(x'_o)$

  $C_o : (B, M, N) \leftarrow \text{Linear}^C_o(x'_o)$

  /* *softplus ensures positive* Δ_o */

  $\Delta_o : (B, M, E) \leftarrow \log(1 + \exp(\text{Linear}^\Delta_o(x'_o) + \text{Parameter}^\Delta_o))$

  /* *shape of* Parameter^A_o *is* (E, N) */

  $\overline{A_o} : (B, M, E, N) \leftarrow \Delta_o \otimes \text{Parameter}^A_o$

  $\overline{B_o} : (B, M, E, N) \leftarrow \Delta_o \otimes B_o$

  $y_o : (B, M, E) \leftarrow \text{SSM}(\overline{A_o}, \overline{B_o}, C_o)(x'_o)$

end for

/* *get gated* y_o */

$y'_{\text{forward}} : (B, M, E) \leftarrow y_{\text{forward}} \odot \text{SiLU}(z)$

$y'_{\text{backward}} : (B, M, E) \leftarrow y_{\text{backward}} \odot \text{SiLU}(z)$

/* *residual connection* */

$T_l : (B, M, D) \leftarrow \text{Linear}^T(y'_{\text{forward}} + y'_{\text{backward}}) + T_{l-1}$

Return: T_l

The operations of the Vim block are detailed in Algorithm 21. Initially, the input token sequence T_{l-1} is normalized using a normalization layer. Following this, the normalized sequence is linearly

projected to x and z, each with a dimension size of E. The x sequence is then processed in both forward and backward directions. For each direction, a one-dimensional convolution is applied to x, producing x'_o. Subsequently, x'_o is linearly projected to B_o, C_o, and Δ_o. The parameter Δ_o is then utilized to transform $\overline{A_o}$ and $\overline{B_o}$, respectively. Finally, the y_{forward} and y_{backward} components are computed using the state space model (SSM). These components are then modulated through a gating mechanism with z and subsequently combined to produce the output token sequence T_l.

3.4.1.4 Architecture Details

In summary, the hyperparameters of the proposed architecture are as follows:

L: The number of blocks.

D: The hidden state dimension.

E: The expanded state dimension.

N: The dimension of the state space model (SSM).

Following the methodologies of Vision Transformers (ViT) [12] and DeiT [42], a projection layer with a 16×16 kernel size is initially employed to generate a one-dimensional sequence of non-overlapping patch embeddings. This is followed by directly stacking L Vim blocks. By default, the architecture is configured with 24 blocks ($L = 24$) and an SSM dimension (N) of 16. To maintain alignment with the model sizes of the DeiT series, the hidden state dimension (D) and expanded state dimension (E) are set to 192 and 384, respectively, for the tiny-size variant. For the small-size variant, these dimensions are increased to $D = 384$ and $E = 768$.

3.4.1.5 Efficiency Analysis

Traditional SSM-based approaches, such as those in Equation (4), leverage the FFT to accelerate the convolution operations. However, the SSM operation in Line 11 of Algorithm 1 is no longer equivalent to convolution due to the data dependency in Mamba and other purely data-dependent approaches. The two approaches, Mamba and the proposed Vim, share a hardware-efficient strategy towards optimizing performance on modern hardware accelerators such as GPUs. The basic principle for this optimization is to avoid IO and memory bottlenecks, which usually occur with such hardware.

IO-Efficiency. High Bandwidth Memory (HBM) and Static Random-Access Memory (SRAM) are two critical components of GPU architecture. SRAM offers higher bandwidth, while HBM provides greater memory capacity. In the standard implementation of Vim's state space model (SSM) operation using HBM, the required memory input/output (IO) operations scale on the order of $\mathcal{O}(BMEN)$, where B, M, E, and N represent the respective dimensions of the computation. Drawing inspiration from Mamba, Vim implements an efficient memory management strategy by

first transferring $\mathcal{O}(BME + EN)$ bytes of memory (Δ_o, A_o, B_o, C_o) from the slower High Bandwidth Memory (HBM) to the faster Static Random-Access Memory (SRAM). Once in SRAM, Vim computes the discrete $\overline{A_o}$ and $\overline{B_o}$, with dimensions (B, M, E, N). Subsequently, Vim executes the state space model (SSM) operations within SRAM and writes the resulting output, sized (B, M, E), back to HBM. This approach effectively reduces the memory input/output (IO) operations from $\mathcal{O}(BMEN)$ to $\mathcal{O}(BME + EN)$, significantly improving efficiency.

Memory-Efficiency. To address out-of-memory issues and reduce memory usage when processing long sequences, the Vision Mamba (Vim) adopts the same recomputation strategy as Mamba. For intermediate states of size (B, M, E, N), required for gradient calculation, Vim recomputes these states during the network's backward pass. Similarly, intermediate activations, such as outputs of activation functions and convolution operations, are recomputed to optimize GPU memory usage. This approach is effective as activation values, though memory-intensive, can be recomputed quickly without significant computational overhead.

Computation-Efficiency. The state space model (SSM) in the Vim block (Line 11 in Algorithm 1) and the self-attention mechanism in Transformers are both crucial for adaptively capturing global context. For a visual sequence $T \in \mathbb{R}^{1\times M\times D}$ with the default setting $E = 2D$, the computational complexities of global self-attention and SSM are as follows:

$$\Omega(\text{self-attention}) = 4MD^2 + 2M^2D, \tag{7}$$

$$\Omega(\text{SSM}) = 3M(2D)N + M(2D)N, \tag{8}$$

In this context, the computational complexity of self-attention is quadratic with respect to the sequence length M, while the complexity of the state space model (SSM) is linear with M, where N is a fixed parameter set to 16 by default. This computational efficiency allows Vim to scale effectively for gigapixel applications involving large sequence lengths.

3.4.2 Vision Transformers

ViT is a breakthrough in computer vision, considering it applies the transformer architecture from NLP to image recognition tasks. Dosovitskiy et al. [12] introduced the model in their work "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale", where they challenged the long dominance of CNNs in computer vision.

Transformers were first introduced in the NLP domain by Vaswani et al. [43] in "Attention is All You Need." Transformers are built around self-attention, a mechanism that enables the model

to dynamically weigh the importance of different input elements relative to one another. This self-attention mechanism replaces the fixed convolutional kernels of CNNs with global attention, allowing the model to capture long-range dependencies and relationships in the input data. ViT extends this architecture to image data by reimagining images as sequences of smaller units analogous to words in a sentence. This framework's critical innovation lies in how it tokenizes and processes images. Figure 11 depicts the overview of the vision transformer model.

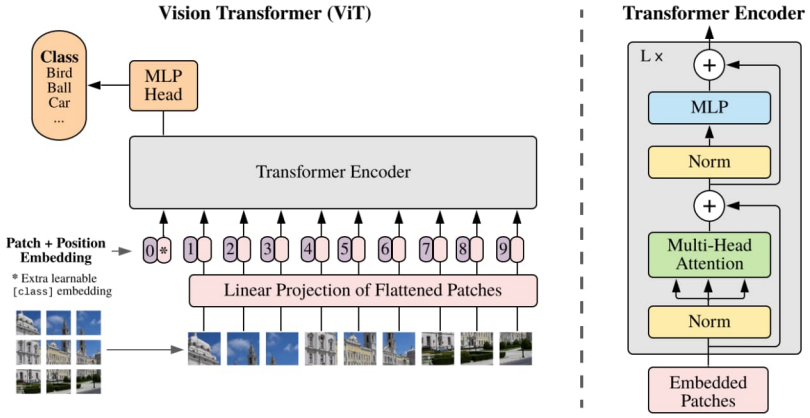


Figure 11: Vision Transformer model overview. The process involves dividing an image into fixed-size patches, linearly embedding each patch, and adding positional embeddings to the resulting vectors. These vectors are then input into a standard Transformer encoder. For classification purposes, the method incorporates a standard technique of appending an additional learnable "classification token" to the sequence. This figure is taken from Figure 1 of [12].

The first challenge in applying transformers to images is that unlike textual data, which is naturally sequential, image data is inherently spatial. ViT addresses this by splitting an image into non-overlapping fixed-size patches (e.g., 16x16 pixels). Each patch is flattened into a vector and linearly projected into a fixed-dimensional embedding space. These embeddings serve as "tokens" that the transformer can process.. For an image of size $H \times W$ (height $\times$ width) with C color channels, dividing it into patches of size $P \times P$ results in:

$$N = \frac{HW}{P^2}$$

patches. Each patch is represented as a vector of size $C \cdot P^2$, and these vectors are linearly transformed into embeddings of dimension D. This transformation ensures that the input to the transformer matches the dimensions expected by the model.

Positional embeddings are added to patch embeddings to preserve the spatial relationships between patches. Since transformers are naturally permutation-invariant, they would lose track of

position without these embeddings. Once the image is tokenized and embedded, the sequence of tokens is passed through a standard transformer encoder. This encoder consists of alternating layers of:

- **Multi-Head Self-Attention (MHSA):** Allows each token to dynamically focus on other tokens in the sequence based on their learned relevance.
- **Feed-Forward Neural Networks (FFNNs):** Applied independently to each token, transforming its representation.

These layers are combined with residual connections and layer normalization to ensure stable and efficient training. Stacking multiple encoder layers allows the model to learn more abstract and complex representations of the input patches, enabling it to better understand patterns and relationships within the image.

For classification tasks, a special class token is added to the beginning of the patch embedding sequence at the input layer. This token, which is randomly initialized, serves as a global representation of the entire image. As the sequence passes through the transformer layers, the class token gathers information from all patches. Its final embedding is then fed into a classification head—usually a linear layer—to generate the output logits for each class.

A key challenge with Vision Transformers (ViTs) is data efficiency. Unlike CNNs, which generalize well with limited training data due to built-in inductive biases like locality and translation invariance, ViTs lack these inherent properties. As a result, ViTs require large-scale datasets to learn meaningful representations, making them more data-hungry compared to CNNs.

To overcome this challenge, ViTs are often pre-trained on massive datasets like JFT-300M, which contains 300 million labeled images. This pretraining helps the model learn general representations that can be transferred to smaller datasets, such as ImageNet, during fine-tuning. This approach significantly improves data efficiency and boosts performance on downstream tasks.

Despite their success, Vision Transformers (ViTs) face several challenges:

- **Computational Cost:** Self-attention scales quadratically with the number of tokens, making it computationally expensive for high-resolution images.
- **Data Hunger:** ViTs require large-scale datasets to outperform CNNs.
- **Interpretability:** Like other deep learning models, transformers can be challenging to interpret, especially when applied to vision tasks.

Vision Transformers (ViTs) represent a major shift in computer vision, challenging the long-standing dominance of CNNs. By leveraging global self-attention, ViTs achieve state-of-the-art

results in image recognition tasks. Their ability to capture global relationships and adapt to various downstream tasks makes them a versatile and powerful tool in deep learning.

3.4.3 ResNet

Residual Networks or ResNets are a breakthrough in deep neural network architecture, especially within computer vision. Proposed by Kaiming He et al. [21], ResNets had been developed to solve some crucial problems while training very deep neural networks: the degradation problem where increased network depth results in higher training errors.

While a few deep networks existed before ResNets, many of them suffered from this degradation problem: a suitably deep model with added more layers often leads to higher training error, which is not expected. Apart from this, the problem differed from the vanishing/exploding gradient problem, indicating that just placing additional layers prohibits effective training.

A ResNet consists of several residual blocks, each consisting of a few layers with a short-cut connection-a direct connection from the block input to its output. This acts like a skip: the network can decide whether it wants to go through the layer or use the shortcut to bypass one or more layers, allowing the gradient to flow directly through the network in the backward pass. Such architecture prevents vanishing gradients and therefore allows the training of networks consisting of hundreds or even thousands of layers.

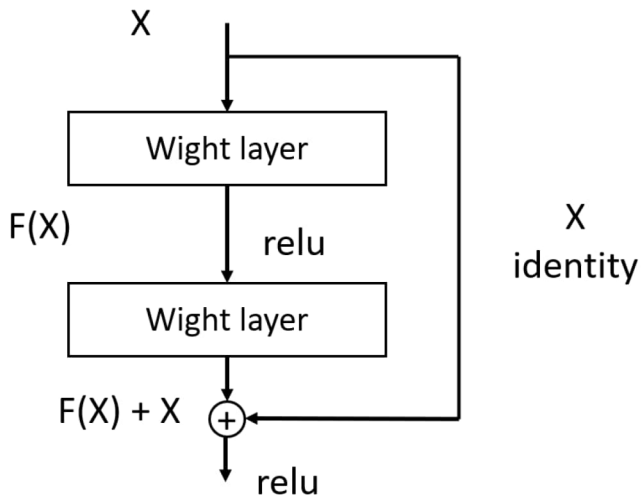


Figure 12: Residual learning: A building block.

A ResNet is composed of residual blocks, each containing a series of layers and a shortcut connection that adds the input of the block to its output. This shortcut, or skip connection, enables the network to bypass one or more layers, allowing the gradient to flow directly through the network

during backpropagation. This design mitigates the vanishing gradient problem and enables the training of networks with hundreds or even thousands of layers.

ResNets have evolved to include various types of residual blocks:

- **Basic Block:** Consists of two sequential 3×3 convolutional layers with a residual connection. The input and output dimensions of both layers are equal.
- **Bottleneck Block:** Comprises three sequential convolutional layers: a 1×1 convolution for dimension reduction, a 3×3 convolution, and another 1×1 convolution for dimension restoration. This design is used in deeper ResNet architectures like ResNet-50, ResNet-101, and ResNet-152.
- **Pre-activation Block:** Applies activation functions before the residual function, reducing the number of non-identity mappings between residual blocks. This design has been used to train models with over 1000 layers.

ResNets have had a huge impact on deep learning, especially in computer vision for image classification, object detection, and segmentation. Residual connections also influenced the design of other architectures, including transformer models such as BERT and GPT, and systems like AlphaGo Zero and AlphaFold.

The success of ResNets has triggered a number of theoretical analyses that try to explain their behavior. It has been observed that residual connections help to preserve norms, which means that backpropagation is stable and the training dynamics are far better. This norm-preserving property explains why ResNets can be efficiently trained even if they are very deep.

ResNets revolutionized the training of deep neural networks by introducing residual connections to avoid the degradation problem, hence allowing the construction of very deep architectures. Their impact goes beyond computer vision into a number of other domains and has inspired many new network designs. The theoretical understanding of their properties is constantly improving, giving deeper insights into their effectiveness while guiding future developments in deep learning architectures.

3.4.4 Workflow of the Research

In the figure 13 the workflow of this research is depicted. Initially, some application transitions are selected from the RICO dataset, and SSIM is applied to them to create an automatically labeled dataset. This dataset requires manual refinement to ensure 100% accuracy. Once refined, it is used to train large deep learning models. Ultimately, the models' results are compared to determine the most accurate model.

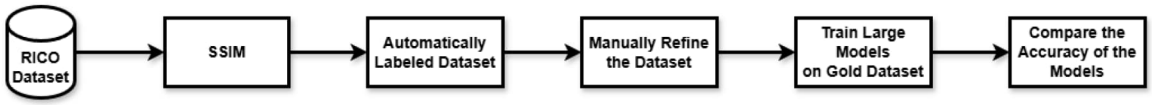


Figure 13: The workflow of the research.

3.5 Conclusion

This chapter introduced a methodology that would classify fully and partially rendered GUIs in mobile applications as an important preliminary step toward optimizing accuracy and efficiency for automated testing frameworks. First to be discussed was the AdaT approach based on SSIM. While SSIM provides a basic structural similarity measure, our analysis has shown critical limitations in capturing partial renderings, especially for dynamic or transitional GUI states. It thus opened the way for more sophisticated approaches beyond mere pixel-level comparisons.

To overcome these shortcomings, we presented an approach that integrated fine-tuning on large-scale pre-trained models such as Vision Mamba, Vision Transformers, and ResNet. We explained in detail the architecture, intuition behind, and role of every model in our system. Among them, Vision Mamba was identified as the most suitable candidate for challenging visual tasks such as GUI classification with its unique bidirectional compression and position-aware mechanism. Visual Transformers were added, with their self-attention mechanisms, and ResNet for powerful residual connections to offer a wider view of model performance.

This chapter aims to serve as a bridge between conceptual design and actual implementation. The following chapter will introduce the implementation of the proposed methodology, presenting experimental results, performance comparisons, and lessons learned while testing the classification system on the Rico dataset. Results of this type will offer a critical look at our approach's effectiveness and generalization capability.

Chapter 4

Results of the Study

4.1 Introduction

The basis of differentiation between fully and partially rendered images is the core focus of this study. This study, in its pursuit to meet its objectives, addresses three important questions to achieve the following:

- RQ1: How accurate is SSIM in labeling the Rico dataset?
- RQ2: What dataset is needed to fine-tune large models effectively?
- RQ3: How effective and efficient are the large models for classifying GUI images?

For **RQ1**, we manually experimented by assessing the accuracy of SSIM labeling through the correctly labeled percentage of images. In **RQ2**, for selecting and precisely labeling the images to prepare the fine-tuning of large models, we analyze several app GUIs from the Rico dataset. Finally, three large models are fine-tuned for **RQ3**, giving promising results to determine the effectiveness of classifying GUI images as fully or partially rendered. The classification logic acts as a trigger for running testing tools that guarantee proper test execution.

4.2 RQ1: SSIM's Labeling Accuracy

Motivation. In this research question, we aim to check the reliability of the Structural Similarity Index in annotation on the Rico dataset, specifically its accuracy for such a large dataset as RICO. Although holding a wide-spread application for image comparison tasks, SSIM sometimes gives unexpected results because of its relatively simple approach. Given the size of the RICO dataset,

quality analysis of the SSIM annotation process is crucial since any wrong labeling would greatly affect the outcome of later processes, such as model training.

Approach. To answer **RQ1**, we conducted an accuracy assessment of SSIM's labeling on a subset of the Rico dataset. We randomly selected 89 apps, consisting of 2,217 images from the unlabeled portion of the RICO dataset. We ran SSIM to label those images and manually checked each image to decide whether the labeling was accurate. Then, we counted the number of correctly and incorrectly labeled images to objectively measure the actual performance of SSIM in this respect.

Results. From our analysis, it came out that approximately 30% of the images were wrongly labeled by SSIM. This is a serious fact because these wrong labels will then give unreliable training data, and hence, the deep learning models trained on such data will make wrong predictions. Further insight into the quality of labeling in the dataset is given in Table 1, which provides the number of correctly and incorrectly labeled images.

Additionally, we identified two major issues with the labeled dataset:

- **Repetitive Images:** A significant number of images were near duplicates, coming from frames that differ very little from their neighbors. These duplicate images can introduce bias to model training by overrepresenting certain visual elements and may affect the model's generalization.
- **Corrupted Images:** Some images were completely corrupted, consisting of white and black pixels, for reasons such as being ineligible for classification. These corrupted images introduce more noise into the dataset, further compromising label quality.

Dataset Category	**Total number of images**	**Wrong Labeled images**	**Accuracy**
Test	429	127	70.39%
Validation	406	121	70.19%
Train	1382	415	69.97%
Total	2217	663	**70.09%**

Table 1: The table provides a summary of dataset statistics of 89 apps, including the distribution across training, validation, and test sets. It highlights the occurrence of mislabeling within each category and presents the overall accuracy.

Discussion. That points to the fact that SSIM, although effective in simple image comparisons, falters for larger, more diverse data like RICO. The very 30% mislabeling rate indicated issues with relying on SSIM when doing large-scale labeling without methods for refinement or validation. Bad labeling can snowball the performance of a machine learning model, leading it to generalize to poor degrees, hence making the model unreliable.

Moreover, the presence of repetitive and corrupted images points to the need for much more robust data curation in training dataset preparation. Ensuring that datasets are clean, diverse, and representative is key to achieving the best results from deep learning tasks.

4.3 RQ2: How to prepare a high quality dataset?

Motivation. In **RQ1**, we saw that SSIM generated a poorly labeled dataset. As a result, a large portion of images were no more than mislabeled. The fact just gave rise to doubts about the usage of SSIM. Thus, we decided to take a small dataset from Rico, call it **GOLD**, and fine-tune large models on it. The proposed research question seeks to establish dataset characteristics that can help fine-tune large models effectively. A well-labeled and high-quality dataset will make a model perform better; thus, a model can generalize well for new data and classify GUI images.

Approach. For the answer to **RQ2**, we first required a clean dataset manually labeled to fine-tune our models. Given that SSIM wrongly labeled some of these, we had to create a new dataset. In particular, those images that SSIM mislabeled were revised, and their labels were corrected. Also, 548 more screenshots from RICO were added, categorizing each image as Fully or Partially rendered. The whole process ensured the quality of the dataset and devoid it of errors identified in **RQ1**.

In particular, we must take special care while training and testing split to serve the dataset well for model generalization. A simple random shuffle of the dataset would result in a data leakage problem [23] since GUIs in the same app would share very similar visual features. To avoid this, we stratified based on the applications; thus, images from a specific app would not be represented in the training and the test sets. This yielded a split of 7:1.5:1.5 across training, validation, and test set splits.

Results. We created GOLD dataset with two distinct labels and divided it into three sets: training, validation, and testing. The dataset statistics, including the number of images per label and set, are summarized in Table 2:

Label	**Train**	**Validation**	**Test**
Full	996	213	215
Partial	938	201	202
Total	1934	414	417

Table 2: The table provides an overview of GOLD dataset, detailing the distribution of fully and partially labeled instances across the training, validation, and test sets.

Discussion. Since there are identified issues about accuracy, as revealed in **RQ1**, this was handled

by the manual curation of the dataset; only the correctly labeled images were used for fine-tuning the large models. A well-balanced dataset of full and partial would be quite important for training a model with good performance in classifying GUI images according to their rendered state.

This careful preparation of the data, especially splitting by app, was necessary to avoid overfitting and ensure the models generalized well. By not allowing any app data to leak into both the training and testing sets, we did not run the risk of the model learning patterns specific to any particular app that may not hold for other apps. This refined dataset allowed for fine-tuning large models with better performance, discussed in **RQ3**. This serves as a starting point for further experiments with different models.

4.4 RQ3: Accuracy of Large Models on Classifying GUI Images

Motivation. The pretraining of many large models on vast amounts of image datasets, including Vision Mamba, Vision Transformers and ResNet to MobileNetV2, may be done so that all these large models are representative of complex features. Alternatively, we can train a model from scratch on a large RICO dataset, but We prefer fine-tuning each large model on a curated small subset. This will save computational resources but yield very strong performance in classifying the rendering state of GUIs. In this regard, we will test the performances of three such models: Vision Mamba, Vision Transformers, and ResNet, all trained on large datasets of images and fine-tuned on our refined dataset for classifying the rendering states of GUIs in **RQ3**.

Experimental Setup. For the experiments, we have utilized vision mamba, Google ViT model, and ResNet-152, and MobileNetV2. We trained the models on a 1934 input image, with the size 224*224, carefully selected from the Rico dataset as described in **RQ2**. We used the AdamW optimizer with a 3×10^{-5} learning rate and set the batch size to 16. All three models have been trained for 20 epochs. Experiments were done with an NVIDIA RTX A4500 with 20 GB.

Metrics. Since the task is a frame of an image classification problem, we used three common metrics in the model's performance evaluation: precision, recall, and F1-score. These metrics provide a comprehensive view of model accuracy, considering both the ability to identify positive instances correctly and the model's sensitivity to detecting all true positives.

Precision is the proportion of GUIs correctly predicted as fully rendered among all GUIs predicted as fully rendered.

$$\text{Precision} = \frac{\#\text{GUIs correctly predicted as fully rendered}}{\#\text{All GUIs predicted as fully rendered}} \tag{9}$$

Recall is the proportion of GUIs correctly predicted as fully rendered among all fully rendered GUIs.

$$\text{Recall} = \frac{\#\text{GUIs correctly predicted as fully rendered}}{\#\text{All fully rendered GUIs}} \tag{10}$$

F1-score is the harmonic mean of precision and recall, which combines both of the two metrics above.

$$\text{F1-score} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{11}$$

Higher values are better for all three metrics. We have also measured the inference time, which denotes the time required by the model to classify an image after it has been trained. The lower the inference time, the faster the model performs, which is crucial for real-time applications.

4.4.1 Analysis of Models

Analysis of Vision Mamba

This section analyzes Vision Mamba's fine-tuning process on our small dataset. First, we load both the model architecture and its pre-trained weights. Then, we change the classification head to output two classes to align the network with our binary classification task. Unlike methods that freeze certain layers, we decided to keep all the layers trainable to fine-tune the whole model.

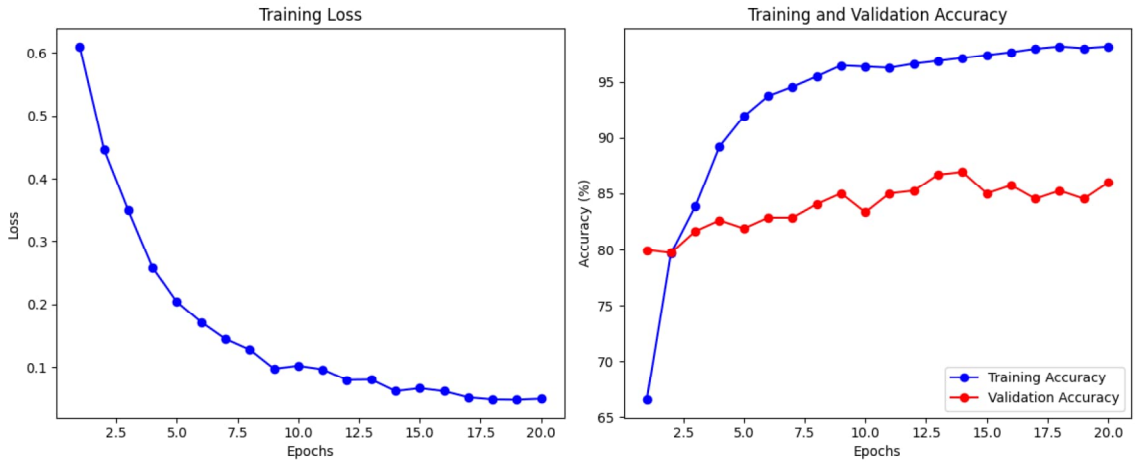


Figure 14: The figure illustrates the training process of Vision Mamba model, showing the reduction in training loss over epochs and the progression of accuracy for both training and validation.

Training Loss. Figure 14 on the left, this plot shows the training loss in several epochs. Much error remains in the model's predictions from the higher value of the training loss in the first epoch during training. The increase in the number of epochs reduces the loss quickly, especially for the

initial epochs - from 1 to 5 - revealing that the model effectively learns this pattern. After about ten epochs, the reduction in loss starts to slow down and converges at a low value close to zero by epoch 20. The consistent decrease in training loss shows the optimization process is going well. The small final training loss suggests a good fit of the model for the training data.

Training Accuracy. Figure 14 the right plot compares the training versus validation accuracy across the epochs. Training accuracy increases rapidly during the first few epochs and reaches about 95% at epoch 10. Beyond this point, the training accuracy continues improving a little and stabilizes at almost 98.19% at around epoch 20. This indicates that the model efficiently learns the training data as it increases rapidly and stabilizes.

Validation Accuracy. The accuracy on the validation set improves within the first few epochs, then is stabilized around 80% after epoch 5. Beyond the fifth epoch, the behavior of the validation accuracy does not improve, and only minor fluctuations can be seen in the later epochs. Another trend that it reflects is the gap between training and validation accuracy. That indicates the model has overfitted the training data because it performs significantly better on the training set than on the validation set.

Test Results. Table 3 reveals that in class Full, this model was a bit better off than in the case of Partial, as demonstrated by the higher recall for class Full. About balanced precision, recall, and F1-score, an overall excellent performance without high bias can be noticed towards one class from another. From an accuracy perspective, it can be seen that an overall accuracy of about 84% indicates quite a very good generalization of this model for the test dataset, whereas there is indeed scope for improvement. The Vision Mamba model takes an average time of 12.47 ms per GUI inference.

Class	**Precision**	**Recall**	**F1-score**	**Support**
Full	0.82	0.89	0.85	215
Partial	0.87	0.79	0.83	202
Accuracy	-	-	0.84	417
Macro Avg	0.84	0.84	0.84	417
Weighted Avg	0.84	0.84	0.84	417

Table 3: The table presents the classification performance of the Vision Mamba model on the test set.

Analysis of Vision Transformers

In this section, we will fine-tune a pre-trained ViT model on ImageNet-21k for our binary classification problem. We start by loading the model and updating its classification layer to output two classes. Unlike the other approaches where some parameters are frozen, we do not explicitly lock any layers here. It enables all the parameters of the ViT model. It changes their status to a trainable state, indicating that the whole architecture can adjust the learned representations according to the features of our target dataset.

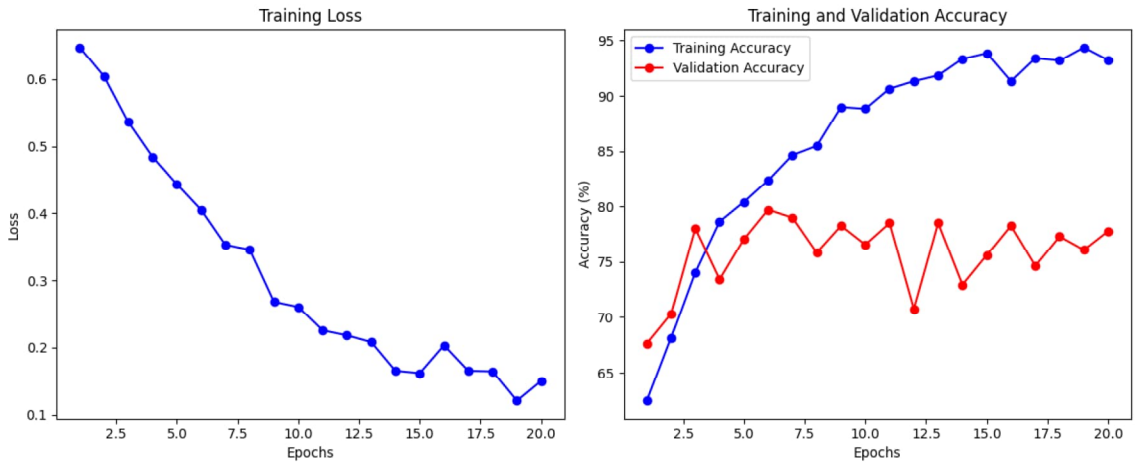


Figure 15: The figure illustrates the training process of the vision transformers model, showing the reduction in training loss over epochs and the progression of accuracy for both training and validation.

Training Loss. In figure 15, the left plot shows the decrease in training loss over 20 epochs. First, the training losses are rather high and above 0.6 during the initial rounds of epochs. One can see a gradual decline due to training, most strongly during the first ten epochs. After the tenth epoch, further reduction decelerates toward stability within the range of 0.1 to 0.2 toward the end. The consistent decrease in training loss supports that model optimization is going pretty well. In contrast, stabilizing the loss on a low value tells us that this model generalized well from the training data.

Training Accuracy. The right plot compares training and validation accuracy over 20 epochs. The training accuracy increases steadily, reaching above 90% by epoch 10. It stabilizes between 93% and 95% during the later epochs, indicating strong performance on the training data.

Validation Accuracy. First, the validation accuracy improves in the first five or so epochs, reaching its peak around epoch 5, then with large fluctuations between 70% and 80%. Large fluctuation in validation accuracy is an indication of instability in the generalization performance of the model.

While the training accuracy increases with more epochs, the gap between the training and validation accuracy becomes large after epoch five, which might indicate overfitting.

Test Results. Table 4 shows that the model has performed better in class Full compared to class Partial. Low recall of class Partial means difficulty finding the positive samples, resulting in many false negatives. With the overall accuracy at 77.94%, this is a reasonable performance but with scope for more improvements. On average, it takes 6.71 ms for a single GUI inference using the Vision Transformers model.

Class	**Precision**	**Recall**	**F1-score**	**Support**
Full	0.74	0.87	0.80	215
Partial	0.84	0.68	0.75	202
Accuracy	-	-	77.94	417
Macro Avg	0.79	0.78	0.78	417
Weighted Avg	0.79	0.78	0.78	417

Table 4: The table presents the classification performance of the Vision Transformers model on the test set.

Analysis of ResNet

This section will adapt a pre-trained ResNet152 model for our binary classification task. We load the model and replace its last fully connected layer with a new one that outputs two classes. We keep all parameters trainable, allowing the entire model to adjust its representations to the specifics of our target dataset. This approach leverages ResNet's powerful, pre-trained feature extraction capabilities while retaining the flexibility to refine high-level and low-level representations, potentially improving overall performance in our binary classification setting.

Training Loss. In figure 16, the left plot shows how the training loss changes over 20 epochs. The loss is relatively high at the start, around 0.66 in the first epoch. The loss drops quickly during the first few epochs, with the most noticeable decrease occurring before epoch 5. After this point, the rate of improvement slows, and the loss stabilizes, fluctuating slightly around 0.56. By epoch 20, the loss reaches its lowest value, though the improvement is minimal compared to earlier epochs. The sharp drop in training loss during the initial stages indicates that the model quickly picks up on important patterns in the data. The stabilization and small fluctuations after epoch 5 suggest that the model has reached a point where additional training provides limited benefit.

Training Accuracy. The right plot compares the training and validation accuracy over the epochs. The training accuracy starts at around 60% in the first epoch and gradually rises to approximately

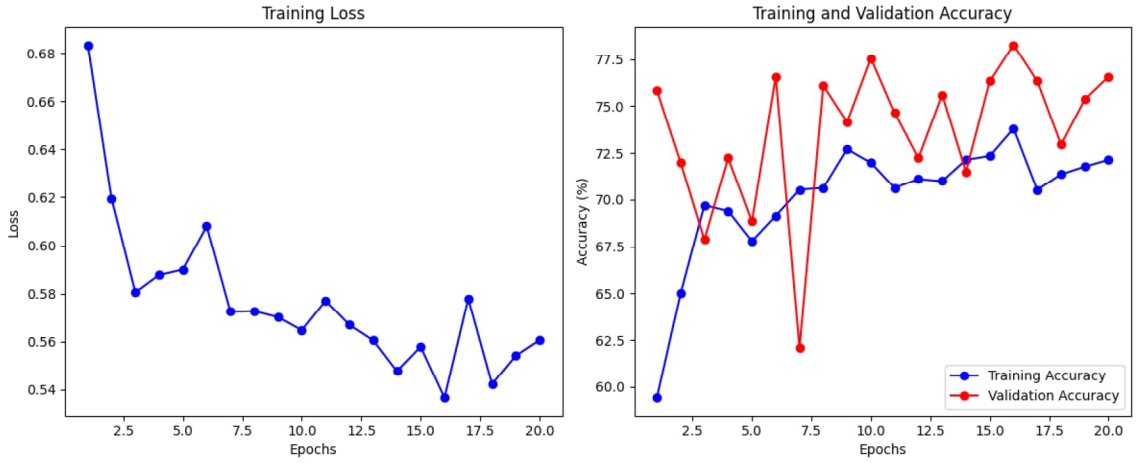


Figure 16: The figure illustrates the training process of the ResNet model, showing the reduction in training loss over epochs and the progression of accuracy for both training and validation.

72% by epoch 5. Beyond epoch 5, the training accuracy fluctuates and largely stabilizes at about 70%, with minimal further improvement toward the later epochs.

Validation Accuracy. The validation accuracy shows significant variation across the epochs, peaking at around 80% near epoch five but frequently dropping to as low as 60%. This inconsistency suggests the model's performance on the validation set is unstable. While the training accuracy follows a gradual upward trend and stabilizes over time, the validation accuracy fluctuates unpredictably. These large swings in validation accuracy indicate potential overfitting or poor generalization, as the model may be overly tuned to the training data and struggling to maintain performance on unseen data.

The reasons why validation accuracy is considerably higher than training accuracy could be attributed to many things. Regularization methods such as dropout or batch normalization usually alter the parameters in training and evaluation phases separately, which results in under performance on the the training accuracy. Augmented data seems to be harder to grasp compared to the unmodified data, but can be easier for validation. A training set that is small or has noise can be more challenging to learn from, but the validation set can be easier and more defined. Differences in batch sizes while training and validating may influence the batch normalization statistics. A lack of time to train, bad shuffling of training data, and an easy validation Set are also possible reasons for this discrepancy.

Test Results. Table 5 indicates that the model returns a higher recall for class Full, with a value of 0.85, than class Partial, with a recall of 0.62. Precision is equilibrated between both classes: 0.71 for Full and 0.79 for Partial, giving a good classification capability. An accuracy of 73.86% reflects

a good generalization of the test set but leaves room for improvement. The ResNet model takes, on average, 31.89 ms per GUI inference

Class	Precision	Recall	F1-score	Support
Full	0.71	0.85	0.77	215
Partial	0.79	0.62	0.70	202
Accuracy	-	-	73.86	417
Macro Avg	0.75	0.74	0.73	417
Weighted Avg	0.75	0.74	0.73	417

Table 5: The table presents the classification performance of the ResNet model on the test set.

Analysis of MobileNetV2

This section will adapt a pre-trained MobileNetV2 model for our binary classification task. We load the model and replace its last fully connected layer with a new one that outputs two classes. We keep all parameters trainable, allowing the entire model to adjust its representations to the specifics of our target dataset.

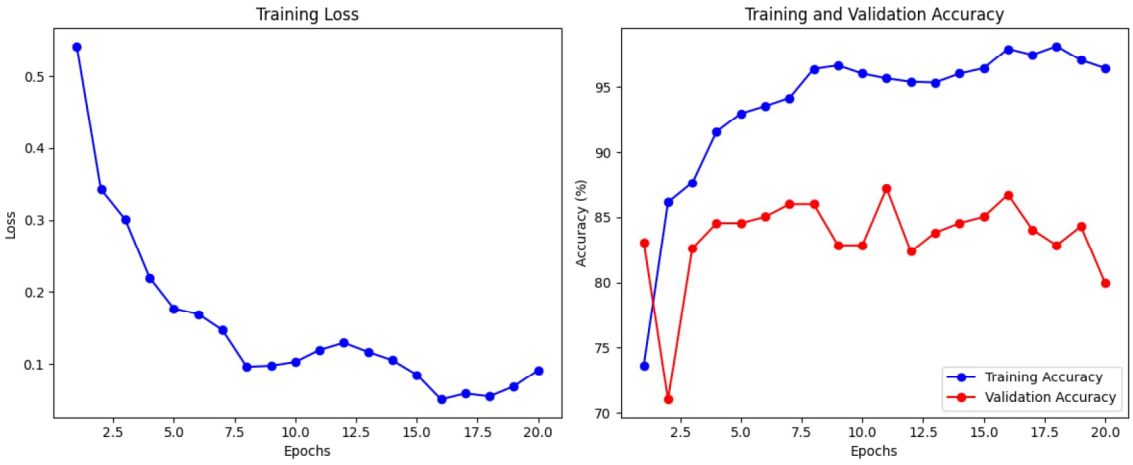


Figure 17: The figure illustrates the training process of the MobileNetV2 model, showing the reduction in training loss over epochs and the progression of accuracy for both training and validation.

Training Loss. In figure 17, the left plot shows how the training loss changes over 20 epochs. The loss is relatively high at the start, around 0.55 in the first epoch. The loss decreases sharply during the first few epochs, indicating that the model is learning effectively. Around epoch 10, the loss stabilizes but shows minor fluctuations. This could suggest slight instability in optimization.

Training Accuracy. The right plot compares the training and validation accuracy over the epochs. The training accuracy starts at around 73% in the first epoch and gradually rises to approximately 92% by epoch 5, indicating that the model is fitting well to the training data. After epoch 10, it plateaus, suggesting that the further training may not significantly improve performance.

Validation Accuracy. The validation accuracy fluctuates more compared to the training accuracy. There is a sharp increase, but the afterward fluctuations show potential overfitting. The final part of the curve shows lower amount than corresponding part in training graph. This depicts that the model may not generalize well to unseen data.

Test Results. Table 6 indicates that the model returns a higher recall for class Full, with a value of 0.82, than class Partial, with a recall of 0.79. Precision is equilibrated between both classes: 0.80 for Full and 0.81 for Partial, giving a good classification capability. An accuracy of 81% reflects a good generalization of the test set but leaves room for improvement. The MobileNetV2 model takes, on average, 4.24 ms per GUI inference

Class	**Precision**	**Recall**	**F1-score**	**Support**
Full	0.82	0.80	0.81	215
Partial	0.79	0.81	0.80	202
Accuracy	-	-	0.81	417
Macro Avg	0.81	0.81	0.81	417
Weighted Avg	0.81	0.81	0.81	417

Table 6: The table presents the classification performance of the MobileNetV2 model on the test set.

4.4.2 Comparative Analysis of Models

Comparison of training loss

Compared to both, Vision Mamba shows very low final training loss with less volatility. In contrast, ViT and MobileNetV2 only perform fairly, with less output stability. ResNet's performance is poor because of the unusually large variability in that a higher final loss is compared to the other methods studied here, thereby establishing a tough case for optimization using these choices of training conditions. This highlights how architectural design and proper mixing and matching for a good training breed can significantly help attain low training volatility.

Comparison of Training and Validation Accuracy

The accuracy trends of the training and validation for the four models represent marked differences in general stability and their propensity for generalization or overfitting. It is observed from the above that Vision Mamba achieves very high accuracy for both training and validation with minimal variance, demonstrating robust generalization. ViT and MobileNetV2 also attain very high training accuracy, but their validation trends have been erratic, reflecting difficulty in translating the learned patterns to new data. Among them, ResNet falls behind in overall accuracy and stability, struggling to improve and maintain performance on either set. These patterns collectively highlight the superior balance and reliability of Vision Mamba, the intermediate stability of ViT and MobileNetV2, and the challenges faced by ResNet in achieving steady, effective learning.

Comparison of Test Results

Model	**Accuracy (%)**	**Precision**	**Recall**	**F1-Score**	**Time (ms)**
Vision Mamba	84.00	0.84	0.84	0.84	12.47
MobileNetV2	80.58	0.81	0.81	0.81	4.24
Vision Transformers	77.94	0.79	0.78	0.78	6.71
ResNet	73.86	0.75	0.74	0.73	31.89

Table 7: Comparison of test results for Vision Mamba, Vision Transformers, and ResNet.

Testing on the test dataset, Vision Mamba, MobileNetV2, Vision Transformers, and ResNet show striking differences in predictive performance and computational efficiency. The best performance is given by Vision Mamba, with the highest accuracy of 84.00% and a balanced precision, recall, and F1-score of 0.84, showing this model generalizes fairly well and isn't biased towards either class. Although its inference time of 12.47 ms is nothing to brag about, it is still reasonably efficient in making Vision Mamba competent for scenarios that require high accuracy and demand moderate speed.

ViT provides strong, though far from the best predictive metrics - 77.94% accuracy at 0.78 F1-score - but very strong inference speed, 6.71 ms. In this regard, ViT achieves a fantastic balance that may be of interest for tasks where time is crucial, but the absolute top-of-the-class accuracy is not as important. On the contrary, ResNet falls behind on most dimensions - starting from accuracy (73.86%) and F1-score (0.73), finishing with the longest time for one inference at 31.89 ms. These results reinforce that ResNet is more challenging to optimize effectively and unsuitable for applications where speed and accuracy are a priority.

Vision Mamba perhaps provides the best overall balance of predictive performance and efficiency.

On the other hand, ViT offers competitive accuracy with much higher inference speed. However, ResNet is poor in both ways, making it thus the least advantageous choice under these conditions.

Comparison Between Vision Mamba and MobileNetV2

The studied approach, AdaT, performed its own experiments on the MobileNetV2 model with learning process from scratch. However, we decided to use this model to fine-tune the version that was pre-trained on the ImageNet dataset with more than 1 million images and compare it with Vision Mamba. The results show that MobileNetV2 obtained around 80.58% accuracy with a 4.24 ms processing time per GUI. This shows that this model outperforms ViT but lacks the accuracy to take over the first position among the other models. On the other hand, if the speed is preferred, this model can be utilized rather than vision mamba.

Comparison Between SSIM and Vision Mamba

Consequently, the accuracy and reliability of Vision Mamba outperformed SSIM by a margin. While SSIM records a mislabeling rate of about 30%, the overall accuracy that could be attained for Vision Mamba was 84.00%, showing a much finer job in classifying fully and partially rendered GUIs. Besides this, SSIM relies on comparisons at the level of pixels, which very seldom captures subtle semantic differences in GUI rendering states. On the contrary, Vision Mamba with deep learning analyzes higher-order features, hence making much finer distinctions of the rendering states possible. At the same time, it grants balanced precision and recall, a high F1-score, while the very high rate of misclassifications makes SSIM impractical for effective use. Obviously, the highest scores belong to Vision Mamba and prove the robustness of this method as well as its further scalability for most GUI classification challenges.

4.4.3 How the new result will replace existing component in AdaT

According to the obtained results, Vision Mamba is the most accurate model that can predict the state of the GUI. Therefore, in AdaT architecture we can replace MobileNetV2 with Vision Mamba. On the other hand, the procedure of labeling RICO dataset utilizing SSIM is no longer necessary. Therefore, we only need to train Vision Mamba on GOLD dataset and integrate that in the workflow of AdaT.

Chapter 5

Thesis Contributions and Future Work

5.1 Conclusion

In this study, Vision Mamba emerged as the most effective GUI rendering state classification model, consistently outperforming both ResNet and Vision Transformers across accuracy, precision, recall, and F1-score metrics. Its stable generalization and balanced class-wise performance highlight the model's ability to handle diverse GUI conditions with minimal bias. Although ResNet and Vision Transformers achieved comparable performance on certain metrics, Vision Mamba's predictive quality and robustness advantage are clear despite a modest increase in inference time.

Beyond model comparisons, this chapter critically examined the limitations of the ADAT method, which relies heavily on SSIM-based automated labeling. While computationally efficient, ADAT's approach is prone to a high error rate—30% of labels were found to be incorrect—due to its inability to capture subtle semantic cues or adapt to varied GUI designs. Predefined thresholds and a lack of semantic understanding further constrained ADAT's effectiveness, resulting in inconsistent and unreliable predictions.

A fine-tuning strategy was introduced to address these issues, leveraging a manually curated subset of RICO dataset images and the Vision Mamba model. By grounding the training process in a set of highly accurate annotations, this approach overcame the limitations inherent in SSIM-driven methods. Vision Mamba's deep learning architecture provided a more nuanced understanding of GUI states, adapting seamlessly to complex, dynamic interfaces without relying on hard-coded thresholds. As a result, its labeling accuracy improved substantially, reducing the error rate from 30% to under 5% and producing robust, semantically rich representations of GUI readiness.

This analysis underscores the value of advanced deep learning frameworks in GUI state classification. While the lightweight ADAT method offers computational convenience, its high error rate and inflexible thresholding limit its practical utility. By contrast, Vision Mamba's fine-tuned approach delivers superior accuracy, adaptability, and resilience, establishing it as a more reliable and scalable alternative to the ADAT paradigm.

5.2 Future Work

Building on the insights and results of this thesis, several directions offer opportunities to advance the field of GUI rendering state classification:

- **Refinement of Data Labeling Methods:** Indeed, this manual curation had a payoff- it significantly increases the reliability of the annotations- and the challenge lies in scaling that. Future work can take several directions in semi-supervised or active learning techniques to reduce labeling overhead. These efforts could be complemented further by automatic but highly contextual annotation methods, utilizing, for instance, optical flow or even simple temporal consistency between multiple frames showing a sequence of the same GUI, which have great potential not only for improvements in accuracy but can contribute greatly to the scaling aspect for generating this dataset.

- **Integration of Temporal and Multimodal Information:** While many GUIs involve dynamic features like animations, loading states, and interactively emerging widgets, most current approaches have relied on static screenshots. These could thus be enriched with temporal information such as video frames and/or multimodal cues like textual content, patterns of user interaction, or back-end metadata to provide a complete representation that improves classification for partial or fully rendered states.

- **Domain Adaptation and Robustness Testing:** GUIs can be very different between applications, platforms, and devices. Examples of potential future work include investigating domain adaptation techniques to ensure models like Vision Mamba perform well on new interface styles, themes, or screen resolutions, testing model robustness against adversarial examples, various rendering quirks on devices, and network conditions for real-world environments.

- **Model Explainability and Interpretability:** While Vision Mamba shows very strong performance concerning quantitative metrics, it becomes relevant to understand the logic behind its predictions. Adding explainability frameworks, such as attention heatmaps, feature visualizations, or saliency techniques, would give developers and testers a chance for deeper analysis

of the model's behavior and make sure that at least its classification decisions do not conflict with human logic and domain-specific knowledge.

- **Efficiency and Deployment Considerations:** Although Vision Mamba showed a reasonable inference time, further optimization will be required for some applications, such as mobile testing and low-latency continuous integration pipelines. Model compression techniques, optimizations on edge devices such as quantization and pruning, and cloud deployments would be interesting in bridging the gap from a research prototype to a production-ready solution.

Ultimately, the groundwork laid in this thesis provides a great foundation for further research in this area. Future work can provide more holistic, versatile, and deployable solutions toward GUI rendering state classification by developing superior labeling strategies, incorporating temporal and multimodal data, ensuring domain robustness, explaining, refining the class definition, and focusing on computational efficiency.

Bibliography

[1] David Adamo, Md Khorrom Khan, Sreedevi Koppula, and Renée Bryce. Reinforcement learning for android gui testing. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, pages 2–8, 2018.

[2] Yauhen Leanidavich Arnatovich and Lipo Wang. A systematic literature review of automated techniques for functional gui testing of mobile applications. *arXiv preprint arXiv:1812.11470*, 2018.

[3] Mohammad Bajammal, Andrea Stocco, Davood Mazinanian, and Ali Mesbah. A survey on the use of computer vision to improve software engineering tasks. *IEEE Transactions on Software Engineering*, 48(5):1722–1742, 2020.

[4] Ishan Banerjee, Bao Nguyen, Vahid Garousi, and Atif Memon. Graphical user interface (gui) testing: Systematic mapping and repository. *Information and Software Technology*, 55(10): 1679–1694, 2013.

[5] Ethan Baron, Itamar Zimerman, and Lior Wolf. 2-d ssm: A general spatial layer for visual transformers. *arXiv preprint arXiv:2306.06635*, 2023.

[6] Rohan Bavishi, Erich Elsen, Curtis Hawthorne, Maxwell Nye, Augustus Odena, Arushi Somani, and Sagnak Tasırlar. Introducing our multimodal models, 2023.

[7] Alan F Blackwell. The reification of metaphor as a design tool. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 13(4):490–530, 2006.

[8] Hu Chen, Mingzhe Sun, and Eckehard Steinbach. Compression of bayer-pattern video sequences using adjusted chroma subsampling. *IEEE transactions on circuits and systems for video technology*, 19(12):1891–1896, 2009.

[9] Jieshan Chen, Mulong Xie, Zhenchang Xing, Chunyang Chen, Xiwei Xu, Liming Zhu, and Guoqiang Li. Object detection for graphical user interface: Old fashioned or deep learning or a

combination? In *proceedings of the 28th ACM joint meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1202–1214, 2020.

[10] Wontae Choi, Koushik Sen, George Necula, and Wenyu Wang. Detreduce: minimizing android gui test suites for regression testing. In *Proceedings of the 40th International Conference on Software Engineering*, pages 445–455, 2018.

[11] Biplab Deka, Zifeng Huang, Chad Franzen, Joshua Hibschman, Daniel Afergan, Yang Li, Jeffrey Nichols, and Ranjitha Kumar. Rico: A mobile app dataset for building data-driven design applications. In *Proceedings of the 30th annual ACM symposium on user interface software and technology*, pages 845–854, 2017.

[12] Alexey Dosovitskiy. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

[13] Sidong Feng and Chunyang Chen. Prompting is all you need: Automated android bug replay with large language models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–13, 2024.

[14] Sidong Feng, Mulong Xie, and Chunyang Chen. Efficiency matters: Speeding up automated testing with gui rendering inference. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 906–918. IEEE, 2023.

[15] Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces. *arXiv preprint arXiv:2312.00752*, 2023.

[16] Albert Gu, Karan Goel, and Christopher Ré. Efficiently modeling long sequences with structured state spaces. *arXiv preprint arXiv:2111.00396*, 2021.

[17] Albert Gu, Isys Johnson, Karan Goel, Khaled Saab, Tri Dao, Atri Rudra, and Christopher Ré. Combining recurrent, convolutional, and continuous-time models with linear state space layers. *Advances in neural information processing systems*, 34:572–585, 2021.

[18] Albert Gu, Karan Goel, Ankit Gupta, and Christopher Ré. On the parameterization and initialization of diagonal state space models. *Advances in Neural Information Processing Systems*, 35:35971–35983, 2022.

[19] Ankit Gupta, Albert Gu, and Jonathan Berant. Diagonal state spaces are as effective as structured state spaces. *Advances in Neural Information Processing Systems*, 35:22982–22994, 2022.

[20] Madeleine Havranek, Carlos Bernal-Cárdenas, Nathan Cooper, Oscar Chaparro, Denys Poshyvanyk, and Kevin Moran. V2s: A tool for translating video recordings of mobile app usages into replayable scenarios. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 65–68. IEEE, 2021.

[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[22] Rudolph Emil Kalman. A new approach to linear filtering and prediction problems. 1960.

[23] Shachar Kaufman, Saharon Rosset, Claudia Perlich, and Ori Stitelman. Leakage in data mining: Formulation, detection, and avoidance. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 6(4):1–21, 2012.

[24] Jacob Devlin Ming-Wei Chang Kenton and Lee Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of naacL-HLT*, volume 1, page 2. Minneapolis, Minnesota, 2019.

[25] Junnan Li, Dongxu Li, Silvio Savarese, and Steven Hoi. Blip-2: Bootstrapping language-image pre-training with frozen image encoders and large language models. In *International conference on machine learning*, pages 19730–19742. PMLR, 2023.

[26] Yanghao Li, Hanzi Mao, Ross Girshick, and Kaiming He. Exploring plain vision transformer backbones for object detection. In *European conference on computer vision*, pages 280–296. Springer, 2022.

[27] Yuhong Li, Tianle Cai, Yi Zhang, Deming Chen, and Debadeepta Dey. What makes convolutional models great on long sequence modeling? *arXiv preprint arXiv:2210.09298*, 2022.

[28] Mario Linares-Vásquez, Kevin Moran, and Denys Poshyvanyk. Continuous, evolutionary and large-scale: A new perspective for automated mobile app testing. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 399–410. IEEE, 2017.

[29] Haotian Liu, Chunyuan Li, Qingyang Wu, and Yong Jae Lee. Visual instruction tuning. *Advances in neural information processing systems*, 36, 2024.

[30] Zhe Liu, Chunyang Chen, Junjie Wang, Yuekai Huang, Jun Hu, and Qing Wang. Owl eyes: Spotting ui display issues via visual understanding. In *Proceedings of the 35th IEEE/ACM international conference on automated software engineering*, pages 398–409, 2020.

[31] Zhe Liu, Chunyang Chen, Junjie Wang, Xing Che, Yuekai Huang, Jun Hu, and Qing Wang. Fill in the blank: Context-aware automated text input generation for mobile gui testing. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1355–1367. IEEE, 2023.

[32] Margaret Livingstone. Vision and art: The biology of seeing, 2002.

[33] Atif M Memon, Martha E Pollack, and Mary Lou Soffa. Using a goal-driven approach to generate test cases for guis. In *Proceedings of the 21st international conference on Software engineering*, pages 257–266, 1999.

[34] Stas Negara, Naeem Esfahani, and Raymond Buse. Practical android test recording with espresso test recorder. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 193–202. IEEE, 2019.

[35] Jim Nilsson and Tomas Akenine-Möller. Understanding ssim, 2020. URL `https://arxiv.org/abs/2006.13846`.

[36] Kohei Ohashi, Yukihiro Nagatani, Makoto Yoshigoe, Kyohei Iwai, Keiko Tsuchiya, Atsunobu Hino, Yukako Kida, Asumi Yamazaki, and Takayuki Ishida. Applicability evaluation of full-reference image quality assessment methods for computed tomography images. *Journal of Digital Imaging*, 36(6):2623–2634, 2023.

[37] Dezhi Ran, Zongyang Li, Chenxu Liu, Wenyu Wang, Weizhi Meng, Xionglin Wu, Hui Jin, Jing Cui, Xing Tang, and Tao Xie. Automated visual testing for mobile apps in an industrial seting. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, pages 55–64, 2022.

[38] Kabir S Said, Liming Nie, Adekunle A Ajibode, and Xueyi Zhou. Gui testing for mobile applications: objectives, approaches and challenges. In *Proceedings of the 12th Asia-Pacific Symposium on Internetware*, pages 51–60, 2020.

[39] Jimmy Smith, Shalini De Mello, Jan Kautz, Scott Linderman, and Wonmin Byeon. Convolutional state space models for long-range spatiotemporal modeling. *Advances in Neural Information Processing Systems*, 36, 2024.

[40] Ramadass Sudhir and LDSS Baboo. An efficient cbir technique with yuv color space and texture features. *Computer Engineering and Intelligent Systems*, 2(6):78–85, 2011.

[41] Saghar Talebipour, Yixue Zhao, Luka Dojcilović, Chenggang Li, and Nenad Medvidović. Ui test migration across mobile platforms. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 756–767. IEEE, 2021.

[42] Hugo Touvron, Matthieu Cord, Matthijs Douze, Francisco Massa, Alexandre Sablayrolles, and Hervé Jégou. Training data-efficient image transformers & distillation through attention. In *International conference on machine learning*, pages 10347–10357. PMLR, 2021.

[43] A Vaswani. Attention is all you need. *Advances in Neural Information Processing Systems*, 2017.

[44] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Skyfire: Data-driven seed generation for fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 579–594. IEEE, 2017.

[45] Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing*, 13(4): 600–612, 2004.

[46] Lili Wei, Yepang Liu, and Shing-Chi Cheung. Taming android fragmentation: Characterizing and detecting compatibility issues for android apps. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 226–237, 2016.

[47] Xusheng Xiao, Xiaoyin Wang, Zhihao Cao, Hanlin Wang, and Peng Gao. Iconintent: automatic identification of sensitive ui widgets based on icon classification for android apps. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 257–268. IEEE, 2019.

[48] Shengcheng Yu. Crowdsourced report generation via bug screenshot understanding. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1277–1279. IEEE, 2019.

[49] Shengcheng Yu, Chunrong Fang, Zhenfei Cao, Xu Wang, Tongyu Li, and Zhenyu Chen. Prioritize crowdsourced test reports via deep screenshot understanding. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 946–956. IEEE, 2021.

[50] Shengcheng Yu, Chunrong Fang, Yexiao Yun, and Yang Feng. Layout and image recognition driving cross-platform automated mobile testing. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1561–1571. IEEE, 2021.

[51] Shengcheng Yu, Chunrong Fang, Ziyuan Tuo, Quanjun Zhang, Chunyang Chen, Zhenyu Chen, and Zhendong Su. Vision-based mobile app gui testing: A survey. *arXiv preprint arXiv:2310.13518*, 2023.

[52] Zhengwei Yu, Peng Xiao, Yumei Wu, Bin Liu, and Lijin Wu. A novel automated gui testing echnology based on image recognition. In *2016 IEEE 18th International Conference on*

High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS), pages 144–149. IEEE, 2016.

[53] Xia Zeng, Dengfeng Li, Wujie Zheng, Fan Xia, Yuetang Deng, Wing Lam, Wei Yang, and Tao Xie. Automated test input generation for android: Are we really there yet in an industrial case? In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 987–992, 2016.

[54] Lianghui Zhu, Bencheng Liao, Qian Zhang, Xinlong Wang, Wenyu Liu, and Xinggang Wang. Vision mamba: Efficient visual representation learning with bidirectional state space model. *arXiv preprint arXiv:2401.09417*, 2024.

[55] Penghua Zhu, Ying Li, Tongyu Li, Wei Yang, and Yihan Xu. Gui widget detection and intent generation via image understanding. *IEEE Access*, 9:160697–160707, 2021.