UINav: A Practical Approach to Train On-Device Automation Agents

Wei Li[†] **Fu-Lin Hsu**[‡*] **Will Bishop**[†] **Folawiyo Campbell-Ajala**[†] **Max Lin**[†] **Oriana Riva**[†]

[†] Google Research
[‡] University of Pennsylvania

Abstract

Automation systems that can autonomously drive application user interfaces to complete user tasks are of great benefit, especially when users are situationally or permanently impaired. Prior automation systems do not produce generalizable models while AI-based automation agents work reliably only in simple, hand-crafted applications or incur high computation costs. We propose *UINav*, a demonstration-based approach to train automation agents that fit mobile devices, yet achieving high success rates with modest numbers of demonstrations. To reduce the demonstration overhead, UINav uses a referee model that provides users with immediate feedback on tasks where the agent fails, and automatically augments human demonstrations to increase diversity in training data. Our evaluation shows that with only 10 demonstrations UINav can achieve 70% accuracy, and that with enough demonstrations it can surpass 90% accuracy.

1 Introduction

The next frontier in artificial intelligence is agents that autonomously operate computers as humans do. Instructed by users in natural language, these agents are especially valuable when their users have visual or motor disabilities or when they are situationally impaired (e.g., driving, cooking). We are particularly interested in agents that can execute human tasks by interacting directly with the user interface (UI) of a running application. These so-called *UI automation agents* (Liu et al., 2018; Li et al., 2020; Humphreys et al., 2022) can scale well to support a myriad of tasks because they do not depend on third-party APIs.

Existing approaches to UI automation range from UI scripting to AI-based agents. UI scripts can work reliably, but they involve coding or manual demonstrations (Kundra, 2020; Barman et al., 2016; Riva and Kace, 2021; Li et al., 2017) and they cannot tolerate well changes in the UI and workflows, thus leading to high maintenance costs – this is, however, what enterprises use to automate business workflows (UIPath, 2023). AI-based approaches can scale better. Using imitation learning and reinforcement learning (Liu et al., 2018; Gur et al., 2018), agents are trained to navigate the web autonomously. However, their synthetic and simplified test environments (Shi et al., 2017) and their dependency on large amounts of demonstrations (Humphreys et al., 2022) make them hard to deploy. Recent work leverages Transformers (Li et al., 2020; Li and Li, 2023; Venkatesh et al., 2022; Wang et al., 2023) and pre-trained large language models (LLMs) (Yan et al., 2023; Venkatesh et al., 2022; Zheng et al., 2024). Despite the performance improvement, these solutions come with large resource costs (multiple days of training on hundreds of GPUs/TPUs and high inference costs).

A practical approach to UI automation requires trading between accuracy, generalizability and computational costs. We find a sweet spot between these three properties, and propose *UINav*, a demonstration-based system designed to produce lightweight neural agents that can run on mobile devices while yielding good success rates.

As in prior work, UINav needs to address the challenge of how to achieve good success rates with fewer demonstrations. We observe that the demonstrations required to achieve good performance differs widely across tasks and environments. If the environment is relatively static even a handful of demonstrations is sufficient; for tasks that must work across many different UIs more demonstrations are needed. When collecting demonstrations, UINav provide users with immediate feedback on which tasks are failing and may benefit from additional demonstrations, and which are satisfactory. It does so through a *referee* model which is trained with the same set of demonstrations used to train

[*] Work done as an intern at Google Research.

the automation agent, but with a different goal: predicting whether a task is successfully completed (rather than predicting which UI action to perform).

Another challenge UINav addresses is how to increase the robustness of automation agents against system delays and changes in the UI. It does so through three key techniques. First, every UI action is executed as a small program composed of lower-level operations with status checks. These programs, referred to as *macro action*s, abstract the system-specific details thus greatly reducing the agent's state space and therefore the number of required demonstrations. Second, UINav adopts *demonstration augmentation* where human demonstrations are augmented by randomizing non-critical UI elements to increase their diversity. Finally, through *utterance masking* variable sub-strings in utterances are abstracted out.

We develop UINav using an internal dataset of 40+ tasks and test is on actual Android phones. We also evaluate it on a public dataset, where UINav outperforms various baselines and demonstrates generalizability. Overall, we make the following contributions: (i) a practical system to build UI automation agents that achieve near perfect success rates on previously seen tasks and that can be deployed to mobile devices; (ii) an error-driven process to collect demonstrations paired with augmentation techniques and macro actions; and (iii) a comprehensive evaluation demonstrating UINav's advantages over state-of-the-art systems.

2 Related work

UI automation scripts. Record-and-replay tools like Selenium (Kundra, 2020) can be used to facilitate the generation of UI automation scripts. These scripts can also be integrated with robotic process automation tools (UIPath, 2023; Automation Anywhere, 2023; Blue Prism, 2023). Programming by demonstration tools (Sugiura and Koseki, 1998; Leshed et al., 2008; Lin et al., 2009; Li et al., 2010; Barman et al., 2016; Li et al., 2017; Chasins et al., 2018) are advanced record-and-replay tools that can generate fully functional UI scripts and even action graphs (Riva and Kace, 2021) from recordings of task interactions (demonstrations), which could also be provided in the format of video recordings (Bernal-Cárdenas et al., 2020; Chen et al., 2022). Overall, a major downside of this line of work is that these systems do not produce models that generalize to new task workflows and UIs.

AI-based automation. Transformer-based architectures (Li et al., 2020; Bai et al., 2021; He et al., 2021; Banerjee et al., 2022; Li and Li, 2023) and reinforcement learning approaches (Liu et al., 2018; Gur et al., 2018; Li and Riva, 2021) have been proposed to train agents capable of navigating apps and websites when provided with natural language instructions. Yet, it is unclear how well these systems perform in a variety of real-world environments and scale across task categories because either they have been tested in synthetic webpages of 10–50 UI elements (Shi et al., 2017) or on limited datasets (Li et al., 2020; Burns et al., 2022). Recent work leverages LLMs to ground natural language instructions in UIs (Venkatesh et al., 2022; Wang et al., 2023; Yan et al., 2023; Zheng et al., 2024; Rawles et al., 2023). These approaches come with a large training overhead (e.g., multiple days of training on hundreds of GPUs/TPUs) and a high inference cost which prevents them from running on mobile devices.

In this paper, we extend our previous work (Li, 2021) where macro actions were introduced but was limited to work with OCR and icon recognition, into a full system, that bridges the gap between programming by demonstrations and AI-based systems by providing an easy-to-learn system to train robust, multi-task agents for UI navigation in real-world applications. While the system requires manual demonstrations for training, it provides an error-driven collection of demonstrations where testing scenarios are automatically generated and evaluated by the system, thus reducing the number of redundant demonstrations. The error driven demo collection of UINav is inspired by the DAGGER (Ross et al., 2011) algorithm and we show that it is effective in reducing the number of demonstrations for both sequential (referee) and non-sequential (agent) models.

3 Why is UI automation hard?

We study the problem of how a UI automation system can generalize to new execution environments, including different apps and different tasks, *without* requiring an excessive number of demonstrations. To illustrate the challenges we use an apparently simple task, *search*, i.e., operating the search bar of an app. Two aspects make this task challenging.

Search is a universal task that must work across a myriad of apps where search bars can take many different formats. Some search bars require the user

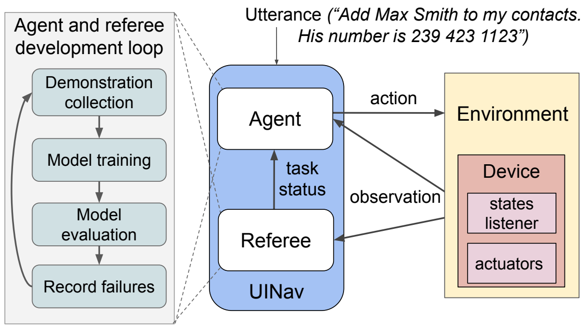


Figure 1: High-level architecture of UINav.

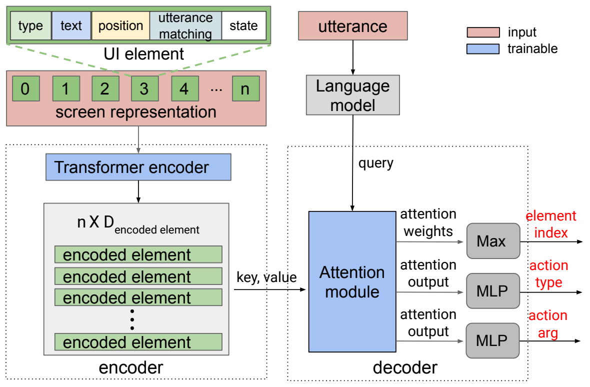


Figure 2: The neural network of the agent model.

to type some keywords and then click an icon (typically on the right hand-side); others, as the user types, automatically display search results which can be directly opened; some others have an additional field (e.g., a category) which must be set beforehand; there are also search bars that are hidden and reveal only upon clicking on an icon; etc.

The second axis of complexity regards the agent's start state. When an agent is requested to search in a specific app, the user's device screen may not display the target app or may display it in a page (state) without any search functionality. The agent must first understand how to navigate to the state offering the search function, which may involve navigating back, launching a different app, or dismissing welcome screens and ads. Even when the environment already shows the desired search widget, its state may need to be reset, e.g., by deleting search terms previously entered (see the YouTube example in Fig. 6 in the Appendix).

In general, in a real environment, an agent is exposed to many different screen conditions caused by a combination of factors: different apps, dynamic app content, previous interactions, layout variance across devices, UI changes across app/OS versions, ads and notifications, etc. An agent needs to ignore irrelevant UI elements and navigate to relevant states. One way to tackle this variability is through more demonstrations, but with obvious overheads. UINav's first contribution is to adopt an error-driven process to collect demonstrations (§4.2). Its second contribution is to amplify the learning brought by each demonstration by automated augmentation (§5). Finally, to address variability issues due to system delays, rather than relying on demonstrations UINav takes a programmatic approach by introducing macro actions (§5).

4 System design

Fig. 1 shows the high-level architecture of UINav. Given a task represented by a natural language utterance and an observation of the device state (i.e., a representation of what is currently displayed on the screen), a neural network-backed agent responds with its choice of action to complete the task. The predicted action is executed by the environment by interacting with a device's system (an emulator or a real phone). Then, the agent is provided with a new observation describing the new state and a new action is predicted. This setup is similar to that of a reinforcement learning agent, but UINav also includes a second agent called *referee*, which is responsible for judging the completion status of a task (episode) at each time step.

The development of UINav agents (left of Fig. 1) involves first collecting human demonstrations for some target tasks, then training the neural networks of the agent (§4.1) and referee (§4.2), and finally evaluating them on the device. Failures of either the agent or the referee are recorded and used to guide the collection of new demonstrations to be used in the next round of training. The development loops over these steps until no more errors of either the agent or the referee are found.

4.1 Agent's neural network architecture

The UINav agent consists of an encoder-decoder architecture (Fig. 2). It perceives the state of the device through observations of what is currently displayed on the screen, represented by the set of UI elements composing it. Each UI element is described by a set of attributes: type (button, icon, etc.), text (visible text, content description, resource identifier, etc.), on-screen position, utterance matching (whether on-screen text matches the utterance[1]), and state (e.g., whether a checkbox is selected). The screen representation can be generated from raw pixels processed by screen understanding techniques (Chen et al., 2020; Wu et al., 2021; Zhang et al., 2021), which also include icon detection and text recognition, or from a tree-structured representation of the UI, such as the Android accessibility tree. Our implementation dynamically switches between the two sources of screen representation based on simple heuristics, such as whether the target app is known to provide poor accessibility support or whether the number of accessibility nodes is extremely small.

Then, the input to the neural network of the agent is a set of UI elements and an utterance. Each UI element is represented by a vector concatenated from the feature vectors of its attributes. Text labels of UI elements are encoded by a language model (Devlin et al., 2019). The feature vectors of the UI elements are fed into a Transformer encoder. The output of the encoder is a function of the encoding of each UI element plus its attention over all other UI elements on the screen, including itself.

The decoder predicts which action to perform. This involves predicting *(i)* the UI element on which to perform the action, *(ii)* the type of action (click, type, etc.), and *(iii)* any argument for the action. Actions (summarized in Table 3, §A.2) can be of two types. Element actions (click, focus_and_type, dismiss) manipulate a specific element, while global actions (wait, back, scroll, open_app) are general operations or platform-specific functions.

The decoder uses a single cross-attention module, with the utterance embedding serving as the query vector and element encodings serving as keys and values. The largest attention weight is used to select the element to act upon, while the vector output of the cross-attention module is passed through two independent multi-layer perceptrons (MLP) to predict action type and argument.

In its essence, the agent's neural network implements a scoring system. For any given screen, all its elements are scored, and the highest-scored one is selected. Due to the attention in the encoder, for any UI element, its relationship with all the other elements can be encoded. The Transformer model learns how different combinations of UI elements and utterances map to actions, and uses this knowledge to rank elements to act on. It is essential that the model learns to evaluate single UI elements in the *context* of others because the meaning of UI elements is often context sensitive (Banerjee et al., 2022) – elements of similar appearance (color, size and shape) can have different functions but neighboring elements like text labels can help resolve the ambiguity. For specific examples on how UINav contextually evaluates UI elements see §A.8.

[1]Similarly to previous work (Liu et al., 2018), we compute utterance matching as the average of the similarity scores of all words in the UI element's text with the utterance.

4.2 Referee model

In the agent's action space there is no "done" action. This means that the agent does not stop on its own but instead relies on the environment to terminate a task. This is common practice in reinforcement learning. Instead of building task-specific termination logic, we train a *referee* model to predict whether a task is completed at each step and what its outcome is. The referee is trained using the exactly same set of demonstrations as the agent, hence it does not incur extra effort in data collection. However, it also serves a second purpose.

A well-known challenge in demonstration-based systems is that they can require excessive developer time to collect a sufficient number of demonstrations (Lau, 2009) and that it may be difficult to provide samples that are sufficiently different from each other (Myers and McDaniel, 2001; Lee et al., 2017). By automatically evaluating the execution of a currently-trained agent and identifying failing tasks, the referee guides users towards collecting new demonstrations *only* for critical scenarios. Failed executions are saved along with all their parameters and passed to the demonstrator.

The neural architecture of the referee model is similar to that of the agent except that it is wrapped in a recurrent neural network to consider the history of actions (see §A.3 for more details). The referee predicts one out of 4 labels: (1) SUCCESSFUL: the task is completed successfully; (2) FAILED: the task has failed or has reached the maximum number of allowed steps; (3) PENDING: the task is ongoing; or (4) INFEASIBLE: the task cannot be executed.

4.3 Utterance masking

UINav's focus is on generalizing to different execution environments without requiring an excessive number of demonstrations. However, another large source of variability is the input instruction provided in natural language. To address this problem, we design UINav agents to learn general task workflows rather than specific task instances. We do so by pre-processing utterances to identify sub-string that represent the variables of a task. For example, in *Search for tiktok in Google*, *tiktok* is the phrase to search for and can be replaced by other keywords. The variable sub-strings are masked and replaced by *placeholders* before being encoded, so that the utterance embedding is independent on the specific instances. As a result, there is no need to train with different utterances covering the distribution of variables.

For any utterance, all the replaced sub-strings are included in the list of entities associated with the task. A matching vector is computed for each UI element on the screen and is included in the element attributes passed as input to the agent. In the matching vector, each scalar is in the range of $[0, 1]$ and computed as the cosine similarity between the text label of the UI element and the corresponding entity string.

Variable sub-strings can be identified by either following pre-defined patterns, through the use of explicit delimiters, or semantic parsers (Kamath and Das, 2019). LLMs can also be employed (Shin and Van Durme, 2022; Drozdov et al., 2022; Mekala et al., 2022). UINav still works without utterance masking but may require more demonstrations to reach similar accuracy (see ablation analysis in Table 2).

5 Increasing robustness and efficiency

We have described how UINav helps developers balance accuracy and number of demonstrations. Next, we describe the techniques that increase the agent robustness in the face of system delays, UI changes, and variations in task descriptions.

Action validation and macro actions. Controlling UIs of an actual device involves dealing with various system issues. There are unavoidable delays between the time a state is collected from a device and when a predicted action is ready to be performed. Screens can also be slow at loading or updating, hence an agent needs to wait for them to stabilize. These delays are particularly noticeable on a mobile device. To deal with these issues, rather than modeling this variability through more demonstrations, we take various programmatic measures.

First, before executing an action, UINav validates it by checking whether a referenced UI element is still on the current screen and if so, whether it has changed. If the action is not applicable anymore, it requests a new prediction.

Table 1: Inference time (msec) on high/low-end phones. None of the models utilize any accelerators.

Device	Agent	Referee	SmallBERT	Total
High-end	1.98	2.21	262.79	267.00
Low-end	4.40	5.24	427.63	437.27

Second, every action is executed as a small program that is composed of lower level operations with status checks. Such a program is referred to as *macro*. Each macro is implemented following a state transition graph and it is atomic so that while a macro is running the agent stays idle and changes to the screen are not visible to it. An example of macro action is focus_and_type which comprises 4 low-level actions: clicking the input field to obtain focus, waiting for the blinking cursor to appear, typing the text in the field, and (optionally) pressing Enter. See §A.4 for more details.

Demonstration augmentation. To further limit the number of required demonstrations and amplify the learning brought by each one, UINav also augments the collected demonstrations by randomizing the attributes of randomly-selected, non-critical UI elements. This teaches the agent which elements may be safely ignored, and ultimately makes it more tolerant to UI changes. Non-critical UI elements have their attributes modified with a predefined probability by either (i) replacing the embedding of their text labels with random vectors, or (ii) by adding random offsets to the four scalars of their bounding boxes, which is equivalent to randomizing both the element's position and size. Despite its simplicity, demo augmentation is highly effective at improving UINav's performance (see Table 2).

6 System evaluation

We built UINav for Android. Both the agent and referee are implemented in TensorFlow. The agent model has 320k parameters and its tflite version occupies 1.3MB, while the referee has 430k parameters and it is 1.8MB large. For text encoding we use SmallBERT (Turc et al., 2019) and convert it to a 17.6MB tflite model. No quantization is applied during the conversion (More implementation details in §A.5). As shown in Table 1), both the agent and referee take only a couple of milliseconds to execute on a high-end phone (e.g., Pixel6pro) and around 5 milliseconds on a low-end phone (Pixel 3a). BERT dominates the total time.

Table 2: Task and step accuracy on MoTIF.

Model	App seen task unseen		App unseen task seen	
	task acc	step acc	task acc	step acc
Seq2Seq	22.5%	40.4%	18.0%	31.3%
MOCA	21.3%	40.0%	17.0%	32.7%
Seq2Act	32.4%	66.4%	28.3%	67.7%
UINav	37.9%	73.7%	36.8%	66.8%
UINav+aug	39.4%	74.9%	39.7%	68.4%
UINav+aug+utt	**68.3%**	**89.7%**	**59.6%**	**81.9%**

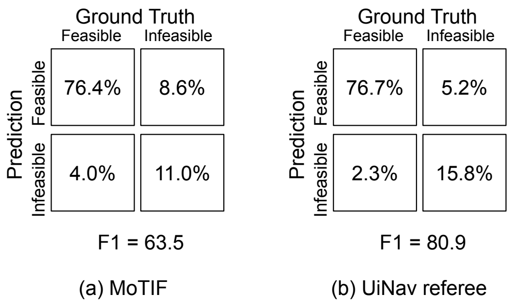


Figure 3: Referee model compared to the MoTIF system (Burns et al., 2022) using the MoTIF dataset.

6.1 Agent and referee accuracy

We evaluate UINav on the MoTIF dataset (Burns et al., 2022). MoTIF includes two splits: (i) *app seen task unseen* which tests whether a model can generalize to new tasks, and (ii) *app unseen task seen* which tests whether a model can generalize to new apps. As in the evaluation of the MoTIF system, we train UINav using low-level instructions, and compare against three baselines: Seq2Seq (Shridhar et al., 2019), MOCA (Singh et al., 2020), and Seq2Act (Li et al., 2020). More training details in §A.7. We measure *(i) step accuracy*, the percentage of task steps where the model and the dataset have matching outputs, and *(ii) task accuracy*, the percentage of tasks with all steps matching.

Table 2 reports the results. For ablation purposes, we consider three variants of UINav, depending on whether demonstration augmentation (+aug) and utterance masking (+utt) are enabled. UINav+aug surpasses all baselines by 7 and 11 percentage points in task accuracy and 8.5 and 0.7 points in step accuracy. Without demo augmentation UINav outperforms all three baselines, in all except one case (step accuracy in app unseen and task seen). This demonstrates the effectiveness of the UINav design and how demo augmentation effectively exposes the model to a larger variety of training conditions thus improving generalizability. In this dataset, generalizing to new apps appears to be harder than generalizing to new tasks. With the addition of utterance matching, on unseen apps, UINav still achieves 59.6% in task accuracy and 81.9% in step accuracy, well above all baselines.

To evaluate the referee model we use again the MoTIF dataset as its traces are labeled as "feasible" or "infeasible", depending on whether the task was successfully completed. We compare against the MoTIF system, specifically designed to predict task feasibility/infeasibility. As the UINav referee predicts 4 states, we map `SUCCESSFUL`/ `PENDING` to "feasible" and `FAILED`/`INFEASIBLE` to "infeasible". As Fig. 3 shows, our referee model produces a significantly better F1 score, 80.9% vs. 63.5%, and it is especially better in identifying infeasible tasks.

6.2 Demonstration effort

To evaluate the effectiveness of the error-driven demo collection approach of UINav we cannot use static datasets. Hence, we quantify the demonstration effort of UINav by using it to train high-accuracy agents for 43 different tasks across 128 Android apps and websites, selected based on popularity (e.g., Gmail, Contacts, Amazon, Airbnb, linkedin.com, target.com, etc.). Please see §A.9 for the full list. For demo collection we build a dedicated GUI which can be connected to Android phones or emulators (see §A.6). The GUI supports macro actions and error-driven data collection. During data collection and testing, the environment automatically performs a few random operations at the beginning of each task, including randomly changing pixel densities, font scales, device orientation, and issuing a sequence of random number of clicks on randomly selected UI elements. The purpose is to start a task from a random state and to diversify data coverage.

We collect demonstrations with the goal to achieve near perfect success rates. With the exception of the search task we collect from 10 to 106 demonstrations (on average 32.7) per task, 3661 in total (Fig. 4). Collecting 10 demonstrations takes less than 10 minutes. The search task must work across 100+ apps hence requiring 1700+ samples. To verify this data is sufficient to train accurate agents, in a second phase we collect additional 596 test samples. Because of the random initialization of the environment, and the dynamic characteristics of a live system, it is unlikely that the models see a training sample that is identical to a test one. The UINav agent achieves 90.6% task accuracy and 95.8% step accuracy; the referee is 99.5% accurate.

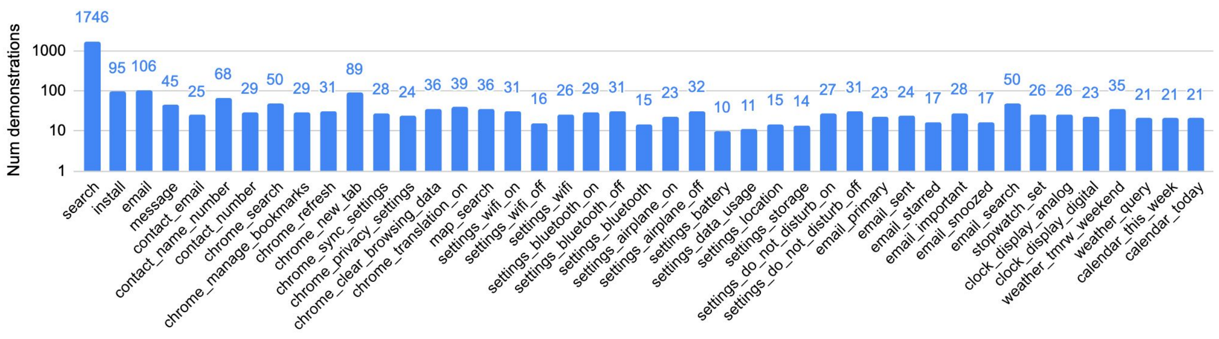


Figure 4: Number of demonstrations in the training set collected for 43 tasks across 128 apps/websites.

Please note that the numbers of demonstrations in Fig. 4 are most likely more than the minimum required to reach the same accuracy, as we prioritize improving accuracy over reducing the number of training samples. It is less effort to add new demonstrations as a batch than finding out whether a specific demonstration improves model accuracy.

In an informal user study, a few software engineers with no prior experience using UINav utilized it to build agents for a few tasks. They started from scratch, without using any existing demonstrations. The time spent on collecting data for each task was between 10 to 20 minutes while all participants claimed their resulting agents performed perfectly.

6.3 Multi-task vs. single-task agents

To reduce the resource overhead on mobile devices, we train a single multi-task agent. We show this choice is preferable also for small numbers of demonstrations. From our in-house dataset, we select the 10 tasks with the largest number of demonstrations. We then train one multi-task UINav agent using demonstrations across all 10 tasks and 10 single-task UINav agents using demonstrations from individual tasks. We repeat the training for an increasing number of demonstrations. As Fig. 5 shows, the multi-task agent reaches 51% accuracy even with just one demonstration, demonstrating transfer learning across tasks is happening. The average accuracy for both multi-task and single-task agents surpasses 80% with 40 demonstrations.

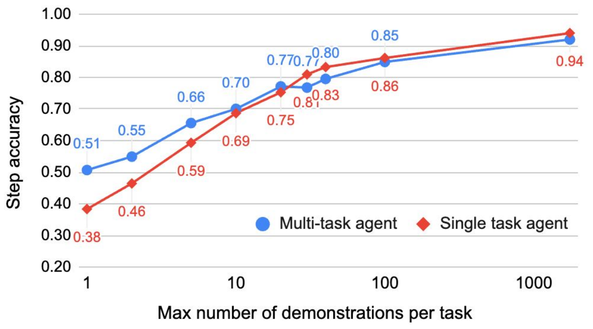


Figure 5: Comparison between multi- and single-task agents with an increasing number of demonstrations.

7 Limitations

To limit the number of required demonstrations, the UINav agent makes decisions based only on the contents of the current screen and does not utilize information from previous screens. However, if a task truly requires an agent to remember previous states or actions, then the current architecture of the agent model will fail. Our assumption is that a well-designed UI often presents all the information that is needed for successful human interaction on the current screen. The accuracy of our memory-less agents proves that this is the case for the tasks tested so far. For tasks or UIs that require memory, the UINav agent model can be enhanced with memory through either a recurrent neural network or by padding previous states in its input.

Our approach depends on UI elements for both representing features of screens as well as defining actions. It will not work if a screen representation fails to capture critical UI elements. This can happen also when accessibility trees miss critical nodes because content embedded in WebViews and Canvas is generally not captured.

8 Conclusions

We presented a demonstration-based system for building small and fast UI automation agents that are suitable for mobile devices. Our approach requires small human effort and no coding skills. With modest numbers of demonstrations UINav agents achieve near perfect success rate on previously-seen tasks and with more effort they can generalize well to new tasks and applications.

References

Automation Anywhere. 2023. https://www.automationanywhere.com.

Chongyang Bai, Xiaoxue Zang, Ying Xu, Srinivas Sunkara, Abhinav Rastogi, Jindong Chen, and Blaise Agüera y Arcas. 2021. UIBert: Learning generic multimodal representations for UI understanding. In *Proc. of the 30th International Joint Conference on Artificial Intelligence, IJCAI 2021*, pages 1705–1712. ijcai.org.

Pratyay Banerjee, Shweti Mahajan, Kushal Arora, Chitta Baral, and Oriana Riva. 2022. Lexi: Self-supervised learning of the UI language. In *Proc. of the 2022 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics.

Shaon Barman, Sarah Chasins, Rastislav Bodik, and Sumit Gulwani. 2016. Ringer: Web Automation by Demonstration. In *Proc. f the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 748–764. ACM.

Carlos Bernal-Cárdenas, Nathan Cooper, Kevin Moran, Oscar Chaparro, Andrian Marcus, and Denys Poshyvanyk. 2020. Translating video recordings of mobile app usages into replayable scenarios. In *Proc. of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, pages 309–321.

Blue Prism. 2023. https://www.blueprism.com.

Andrea Burns, Deniz Arsan, Sanjna Agrawal, Ranjitha Kumar, Kate Saenko, and Bryan A. Plummer. 2022. A dataset for interactive vision language navigation with unknown command feasibility. In *European Conference on Computer Vision (ECCV)*.

Sarah E. Chasins, Maria Mueller, and Rastislav Bodik. 2018. Rousillon: Scraping Distributed Hierarchical Web Data. In *Proc. of the 31st Annual ACM Symposium on User Interface Software and Technology*, UIST '18, pages 963–975. ACM.

Jieshan Chen, Amanda Swearngin, Jason Wu, Titus Barik, Jeffrey Nichols, and Xiaoyi Zhang. 2022. Extracting replayable interactions from videos of mobile app usage.

Jieshan Chen, Mulong Xie, Zhenchang Xing, Chunyang Chen, Xiwei Xu, Liming Zhu, and Guoqiang Li. 2020. Object detection for graphical user interface: Old fashioned or deep learning or a combination? In *Proc. of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, pages 1202–1214.

Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *EMNLP*.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proc. of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics.

Andrew Drozdov, Nathanael Schärli, Ekin Akyürek, Nathan Scales, Xinying Song, Xinyun Chen, Olivier Bousquet, and Denny Zhou. 2022. Compositional semantic parsing with large language models.

Izzeddin Gur, Ulrich Rückert, Aleksandra Faust, and Dilek Hakkani-Tür. 2018. Learning to navigate the web. *CoRR*, abs/1812.09195.

Zecheng He, Srinivas Sunkara, Xiaoxue Zang, Ying Xu, Lijuan Liu, Nevan Wichers, Gabriel Schubiner, Ruby B. Lee, and Jindong Chen. 2021. ActionBert: Leveraging User Actions for Semantic Understanding of User Interfaces. In *35th AAAI Conference on Artificial Intelligence, AAAI 2021*, pages 5931–5938.

Peter C Humphreys, David Raposo, Toby Pohlen, Gregory Thornton, Rachita Chhaparia, Alistair Muldal, Josh Abramson, Petko Georgiev, Alex Goldin, Adam Santoro, and Timothy Lillicrap. 2022. A data-driven approach for learning to control computers. *ICML*.

Aishwarya Kamath and Rajarshi Das. 2019. A survey on semantic parsing.

Manav Kundra. 2020. Selenium - a trending automation testing tool. *International Journal of Trend in Scientific Research and Development*, 4(4):1321–1324.

Tessa Lau. 2009. Why Programming-By-Demonstration Systems Fail: Lessons Learned for Usable AI. *AI Mag.*, 30(4):65–67.

Tak Yeon Lee, Casey Dugan, and Benjamin B. Bederson. 2017. Towards understanding human mistakes of programming by example: An online user study. In *Proc. of the 22nd International Conference on Intelligent User Interfaces*, IUI '17, pages 257–261.

Gilly Leshed, Eben M. Haber, Tara Matthews, and Tessa Lau. 2008. CoScripter: Automating & Sharing How-to Knowledge in the Enterprise. In *Proc. of CHI '08*, pages 1719–1728.

Gang Li and Yang Li. 2023. Spotlight: Mobile UI understanding using vision-language models with a focus.

Ian Li, Jeffrey Nichols, Tessa Lau, Clemens Drews, and Allen Cypher. 2010. Here's What I Did: Sharing and Reusing Web Activity with ActionShot. In *Proc. of CHI '10*, pages 723–732.

Toby Jia-Jun Li, Amos Azaria, and Brad A. Myers. 2017. SUGILITE: Creating Multimodal Smartphone Automation by Demonstration. In *Proc. of CHI '17*, pages 6038–6049.

Wei Li. 2021. Learning ui navigation through demonstrations composed of macro actions. *arXiv preprint arXiv:2110.08653*.

Yang Li, Jiacong He, Xin Zhou, Yuan Zhang, and Jason Baldridge. 2020. Mapping natural language instructions to mobile UI action sequences. In *Proc. of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, pages 8198–8210. Association for Computational Linguistics.

Yuanchun Li and Oriana Riva. 2021. Glider: A reinforcement learning approach to extract UI scripts from websites. In *Proc. of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2021)*.

James Lin, Jeffrey Wong, Jeffrey Nichols, Allen Cypher, and Tessa A. Lau. 2009. End-user programming of mashups with vegemite. In *Proc. of the 14th International Conference on Intelligent User Interfaces*, IUI '09, pages 97–106. ACM.

Evan Zheran Liu, Kelvin Guu, Panupong Pasupat, and Percy Liang. 2018. Reinforcement learning on web interfaces using workflow-guided exploration. In *6th International Conference on Learning Representations (ICLR '18)*.

Dheeraj Mekala, Jason Wolfe, and Subhro Roy. 2022. Zerotop: Zero-shot task-oriented semantic parsing using large language models.

Brad A. Myers and Richard McDaniel. 2001. Demonstrational interfaces: Sometimes you need a little intelligence, sometimes you need a lot. In Henry Lieberman, editor, *Your Wish is My Command*, Interactive Technologies, pages 45–60. Morgan Kaufmann.

Chris Rawles, Alice Li, Daniel Rodriguez, Oriana Riva, and Timothy Lillicrap. 2023. Android in the wild: A large-scale dataset for android device control. In *NeurIPS 2023 Datasets and Benchmarks Track*.

Oriana Riva and Jason Kace. 2021. Etna: Harvesting action graphs from websites. In *UIST '21: The 34th Annual ACM Symposium on User Interface Software and Technology, Virtual Event, USA, October 10-14, 2021*, pages 312–331. ACM.

Stephane Ross, Geoffrey Gordon, and Drew Bagnell. 2011. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, volume 15 of *Proceedings of Machine Learning Research*, pages 627–635, Fort Lauderdale, FL, USA. PMLR.

Tianlin Shi, Andrej Karpathy, Linxi Fan, Jonathan Hernandez, and Percy Liang. 2017. World of bits: An open-domain platform for web-based agents. In *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 3135–3144. PMLR.

Richard Shin and Benjamin Van Durme. 2022. Few-shot semantic parsing with language models trained on code. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 5417–5425, Seattle, United States. Association for Computational Linguistics.

Mohit Shridhar, Jesse Thomason, Daniel Gordon, Yonatan Bisk, Winson Han, Roozbeh Mottaghi, Luke Zettlemoyer, and Dieter Fox. 2019. ALFRED: A benchmark for interpreting grounded instructions for everyday tasks. *CoRR*, abs/1912.01734.

Kunal Pratap Singh, Suvaansh Bhambri, Byeonghwi Kim, Roozbeh Mottaghi, and Jonghyun Choi. 2020. MOCA: A modular object-centric approach for interactive instruction following. *CoRR*, abs/2012.03208.

Atsushi Sugiura and Yoshiyuki Koseki. 1998. Internet Scrapbook: Automating Web Browsing Tasks by Demonstration. In *Proc. of the 11th Annual ACM Symposium on User Interface Software and Technology*, UIST '98, pages 9–18.

Daniel Toyama, Philippe Hamel, Anita Gergely, Gheorghe Comanici, Amelia Glaese, Zafarali Ahmed, Tyler Jackson, Shibl Mourad, and Doina Precup. 2021. Androidenv: A reinforcement learning platform for android. *CoRR*, abs/2105.13231.

Iulia Turc, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. Well-read students learn better: On the importance of pre-training compact models. *arXiv preprint arXiv:1908.08962v2*.

UIPath. 2023. https://www.uipath.com/.

Sagar Gubbi Venkatesh, Partha Talukdar, and Srini Narayanan. 2022. UGIF: UI grounded instruction following.

Bryan Wang, Gang Li, and Yang Li. 2023. Enabling conversational interaction with mobile ui using large language models. In *Proc. of the 2023 CHI Conference on Human Factors in Computing Systems*, CHI '23. Association for Computing Machinery.

Jason Wu, Xiaoyi Zhang, Jeff Nichols, and Jeffrey P Bigham. 2021. Screen parsing: Towards reverse engineering of UI models from screenshots. In *Proc. of the 34th Annual ACM Symposium on User Interface Software and Technology*, UIST '21, pages 470–483.

An Yan, Zhengyuan Yang, Wanrong Zhu, Kevin Lin, Linjie Li, Jianfeng Wang, Jianwei Yang, Yiwu Zhong, Julian McAuley, Jianfeng Gao, Zicheng Liu, and Lijuan Wang. 2023. GPT-4V in Wonderland: Large multimodal models for zero-shot smartphone GUI navigation.

Xiaoyi Zhang, Lilian de Greef, Amanda Swearngin, Samuel White, Kyle Murray, Lisa Yu, Qi Shan, Jeffrey Nichols, Jason Wu, Chris Fleizach, Aaron Everitt, and Jeffrey P Bigham. 2021. Screen Recognition: Creating Accessibility Metadata for Mobile Applications from Pixels. In *Proc. of the 2021 CHI Conference on Human Factors in Computing Systems*, CHI '21.

Boyuan Zheng, Boyu Gou, Jihyung Kil, Huan Sun, and Yu Su. 2024. Gpt-4v(ision) is a generalist web agent, if grounded. *arXiv preprint arXiv:2401.01614*.

A Appendix

Ethical considerations

A use case that motivates UINav agents include screen readers for visually-impaired users. As accessibility labels are often missing or incomplete in mobile apps, UINav can provide them with access to a much wider range of applications and functionality. Another potential use case of UINav is task automation, which has societal, security and privacy implications. An agent may leak private information or carry out a task in an unacceptable way or produce unwanted side effects. Malicious actors could also use UINav agents for undesired purposes such as overriding anti-fraud mechanisms or manipulating applications to achieve undesirable goals.

To develop UINav we collected a dataset internally. The demonstrators were asked to avoid entering any private information and received fair compensation.

A.1 An example task: search in YouTube

Fig. 6 shows the UINav agent searching in YouTube. The agent dismisses popups twice (a) and (b) to reveal the search bar. It then clicks the "X" button to erase the previous search phrase “something” (c). The system does not reach the desired start state for a search until the screen shown in (d), where the agent sets the focus on the search bar to then enter the search term.

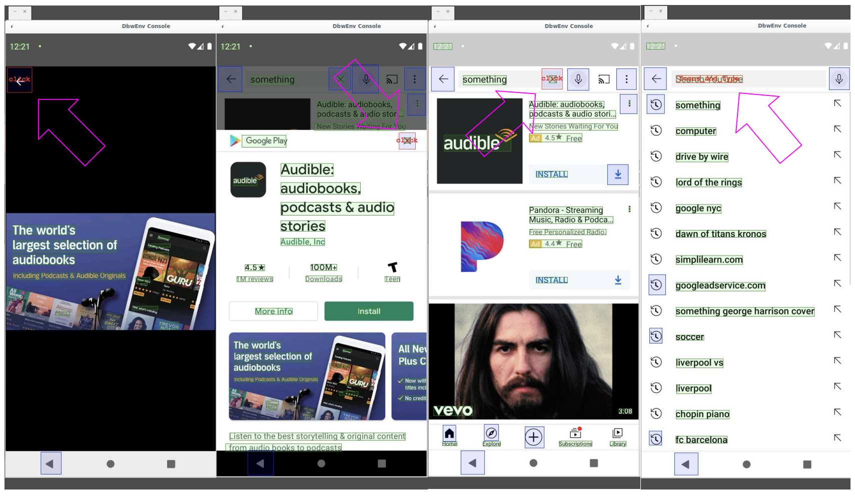
Fig. 4 shows the SEARCH task requires over 1700 task demonstrations because it must work for 100 or more different apps and websites. All other tasks are specific to a single app and thus require fewer samples, 33 on average.

A.2 Action space

The types of action the agent can predict define its action space, summarized in Table 3. Actions can be of two categories. Element actions manipulate a specific element. Global actions are general operations or wrappers for platform-specific functions (e.g., for launching an app). All the tasks that we have tested so far are solvable by these two categories of actions. In the future, we expect to expand the action space to incorporate additional functionality including deep-links and APIs.

A.3 Referee model

The referee is a recurrent neural network (RNN)-based model (Fig. 7). The attention over



(a) (b) (c) (d)

Figure 6: UINav agent searches in YouTube. The pink arrows highlight the agent's actions that are also annotated by red boxes and texts. To start using the search bar the agent must first dismiss popups (twice) and clear the search bar. (a) Clicks the back button to dismiss a popup ads; (b) Clicks "X" to dismiss the install page of Audible; (c) Clicks "X" to erase the previously entered search phrase "something"; (d) Focuses on the search bar to enter a new search term.

Table 3: UINav action space.

Element actions	click <elem>	Clicks the center of the specified element.
	focus_and_type <elem,text>	Sets focus on the specified element, types the specified text, and optionally presses Enter.
	dismiss <elem>	Clicks outside of the specified element.
Global actions	wait	Waits until the next observation is received.
	back	Goes back to the previous app screen.
	scroll <left|right|up|down>	Scrolls in the specified direction.
	open_app <app_name>	Launches the specified application.

Transformer-encoded UI elements is similar to that of the agent model, except that the query is the input utterance concatenated with the action history (the action performed in the previous step and its outcome). Although action history could be derived from previous screen representations, feeding it as input directly makes it less challenging as the referee does not have to learn it. The output of the attention module is then fed into a gated recurrent unit (GRU) (Cho et al., 2014). The GRU takes this along with the previous internal hidden state as inputs to predict the current status of the step: (1) SUCCESSFUL: the task is completed and it is successful; (2) FAILED: the task has failed or has reached the maximum number of allowed steps; (3) PENDING: the task is ongoing; or (4) INFEASIBLE: the task cannot be executed (e.g., the task may not be well defined). Failed executions are saved along with all their parameters and passed to the demonstrator.

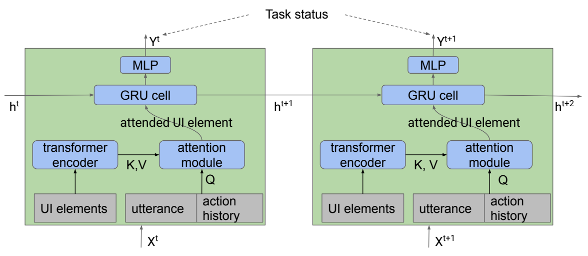
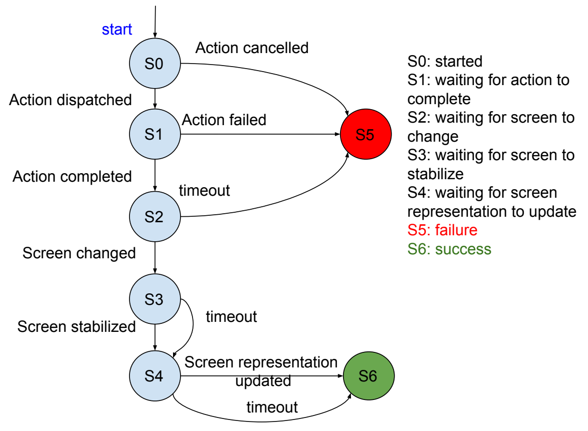


Figure 7: The architecture of the UINav referee model.

A.4 Macro actions

In UINav, every action is executed as a small program that is composed of lower level operations with status checks. Such a program is referred to as *macro*. Macro actions abstract the system-specific details, thus making it possible to build cross-platform agents and simplifying the agent's logic. Each macro action is implemented following a state transition graph. Fig. 8 shows the state transition graph for most macro actions that result in



start
S0
S1
S2
S3
S4
S5
S6
Action cancelled
Action dispatched
Action failed
Action completed
timeout
Screen changed
Screen stabilized
timeout
Screen representation updated
timeout
S0: started
S1: waiting for action to complete
S2: waiting for screen to change
S3: waiting for screen to stabilize
S4: waiting for screen representation to update
S5: failure
S6: success

Figure 8: The state transition graph for macro actions resulting in screen changes.

screen changes, such as `click` and `back`. It starts at S0, and transitions among the other states according to incoming events, such as *Action dispatched* and *Screen changed*, and exits either successfully (S6) or with a failure (S5). The graphs of other macro actions are similar.

Each macro is atomic so that the agent stays idle while a macro is running. During the execution of a macro action, changes to the screen are not visible to the agent, and do not contribute to the state space. In particular, each macro action is designed to encapsulate transitional screens, and finishes when the screen becomes stable or a timeout is reached (required for dynamic screens such as playing a video).

Another advantage of using macro actions is that they package highly dependent, low-level actions. Fig. 9 shows an example. The focus_and_type action (inspired from MiniWoB (Shi et al., 2017)) consists of 4 low-level actions: clicking the input field to obtain focus, waiting for the blinking cursor to appear, typing the text in the field, and (optionally) pressing Enter. (Note that large arrows in purple are drawn to highlight interesting areas.)

As a result, we are able to utilize a memory-less neural network architecture for the agent. In other words, our agent picks an action based only on the information of the current screen. This makes the neural network easier to train. Additionally, a memory-less neural network can be trained using sets of single screenshots, rather than long sequences of screens which can be hard to collect.

A.5 Implementation

We built UINav for the Android platform. However, our design is applicable to other platforms and some of our techniques (e.g., macro-actions and screen representation) are specifically designed to be platform agnostic. Both the agent and the referee models are implemented in TensorFlow. We employ two inference modes, off-device and on-device. During development we use the Python API of TensorFlow to test the models off-device. Once stable, the models are converted to TensorFlow Lite (tflite) for on-device inference. Both agent and referee models utilize the same pre-trained language model to encode utterances and texts appearing on screens. We choose the smallest model, L-2_H-128_A-2, of SmallBERT (Turc et al., 2019), and convert it to a 17.6MB tflite model. Note that no quantization is applied during the tflite conversion of any of the above models. For efficiency, the sentence encoding computation of the agent and referee models are shared.

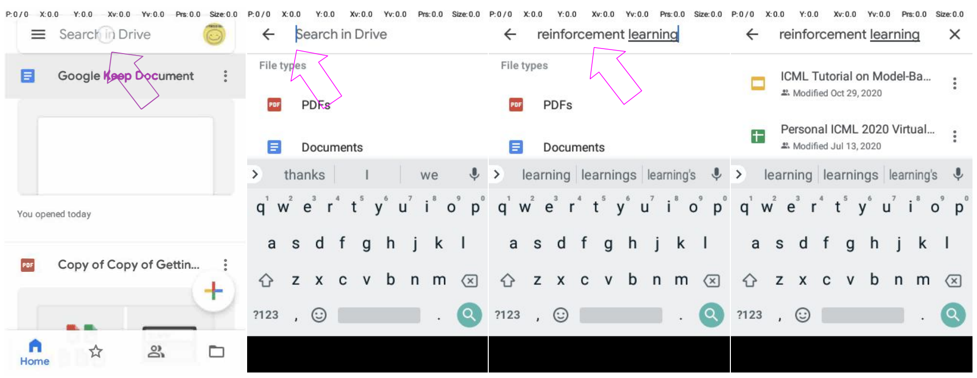
The selection of SmallBERT over a larger language model is mainly for on-device inference. We restrict the input utterances to predefined patterns so that arguments can be parsed through regular expressions. With the help of utterance masking, our models deal with higher data diversity and maintain high-accuracy. If an LLM can be used, such restrictions won't be necessary.

For both off-device and on-device modes, we rely on an in-house built companion Android app to extract screen representations and to perform macro actions. For off-device mode, we utilize AndroidEnv (Toyama et al., 2021) to communicate between the companion app and our learning environment. For on-device mode, all the models interact with the companion app directly.

The neural networks are agnostic to whether the Android accessibility tree or screen understanding techniques are used to produce screen representations. We include demonstrations using both data sources in the same pool of training samples. Both approaches have their limitations. There are icons that are unrecognizable by the icon detectors of screen understanding models and the output of text recognizer may contain errors. On the other hand, visible UI elements may be absent in the corresponding accessibility tree if the app contains Web views, Canvas, etc.

A.6 UINav Console

To collect demonstrations, we have developed a dedicated application, the UINav Console, that can be seen in the right-half side of the screenshots in Fig. 10–12. At each step of a demonstration, a



(a) (b) (c) (d)

Figure 9: The `focus_and_type` macro action consists of four steps: (a) clicking the input field ("Search in Drive") to obtain focus; (b) waiting for the blinking cursor to appear; (c) typing the specified text ("reinforcement learning"); and (d) pressing Enter and wait for the screen to update.

user specifies a macro action, including action type, referenced element, and action argument (if any), and then requests execution of the action.

It is typically less effort to complete a task using the UINav Console than directly manipulating the device. For example, entering text using the console takes at most four clicks (clicking the target element, opening the drop-down list of candidate texts, selecting the text to input, clicking the focus_and_type button), while manipulating a real device requires keying-in individual characters. The UINav Console also exposes system APIs, such as opening an app through intents, that are not available through the actual device. While using the console may encourage users to complete a task in a way that is different than how they might do through a native interface, the main goal of a trained agent is to successfully complete tasks. Whether it behaves like a human is less important.

In the UINav workflow, new human demonstrations are collected only in scenarios where the current version of the agent or the referee make errors. The demonstration collection interface is integrated with the agent and referee. At each step, the agent's choice of an action and its optional argument are assigned to the internal states and are visualized on the GUI. It is not uncommon that an agent produces correct outputs for unseen scenarios due to the neural networks' capability of generalization. In such cases, a demonstrator simply proceeds with a single click to the next step, thus avoiding the effort of manually specifying the action parameters. Error-driven demonstration collection significantly reduces human effort as well as the number of training samples, which ultimately leads to lower training times.

A.7 Model training details

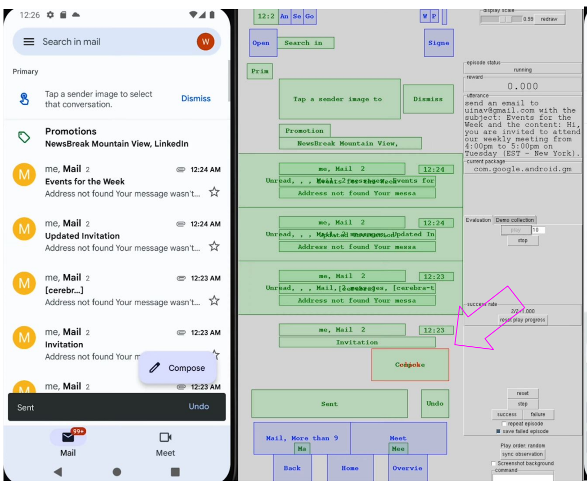
Training the agent model. For the agent model, demo augmentation happens dynamically with a 1% probability for a sample to remain unchanged. The model is optimized by an Adam optimizer with a fixed learning rate of 1e-3. Initially a training runs up to 100,000 samples and can be terminated earlier if the test accuracy stabilizes. If new demonstrations are added, the agent will be trained with additional 20,000 samples. It is trained on CPU or GPU with a batch size of 256.

Training the referee model. For the referee model, each demonstration is augmented to 10 samples at a pre-possessing stage. The model is optimized by an Adam optimizer with a fixed learning rate of 1e-3. A training takes up to 30 epochs and can be terminated earlier if the test accuracy stabilizes. It is trained on CPU or GPU with a batch size of 128.

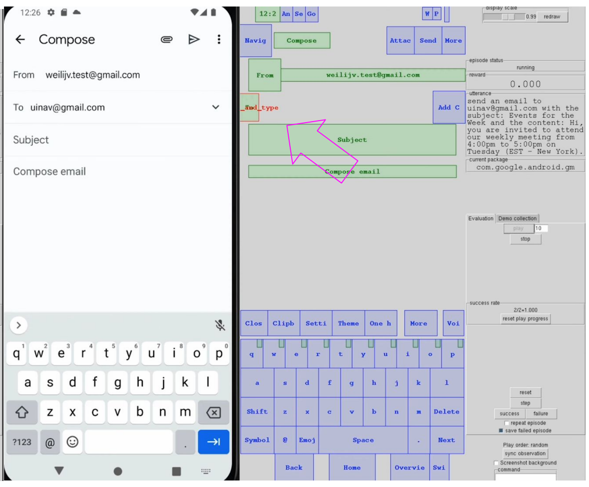
A.8 Case study of agent capabilities

In the following figures we report screenshots and the associated UINav console. The large arrows in purple are drawn on the screenshots to highlight interesting areas. In the console it is the annotated screen, where UI elements are identified using blue and green boxes. An element highlighted by a red box indicates that it is selected to receive the next action.

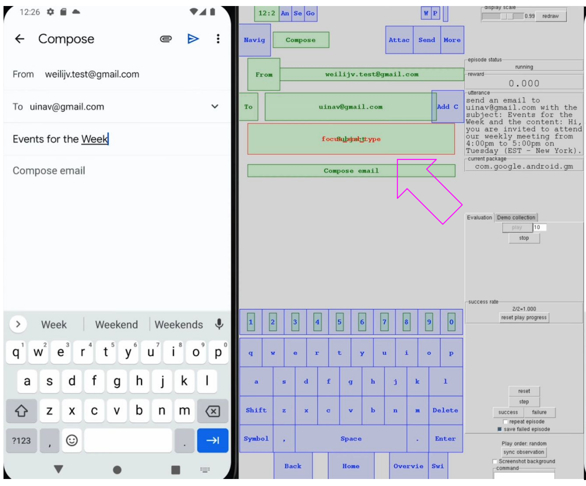
Sending an email with multiple text inputs. Fig. 10 shows the image sequence of a UINav agent completing the "send email" task. The task utterance is "send an email to uinav@gmail.com with



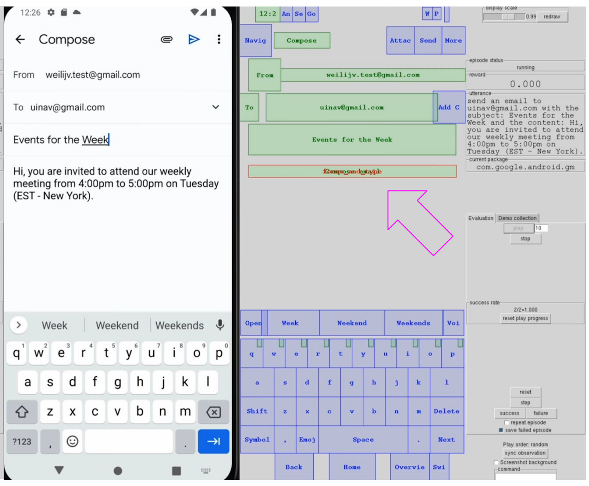
(a)



(b)

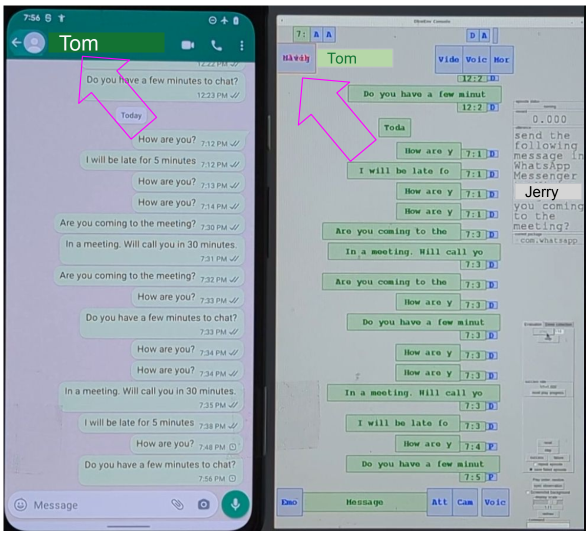


(c)

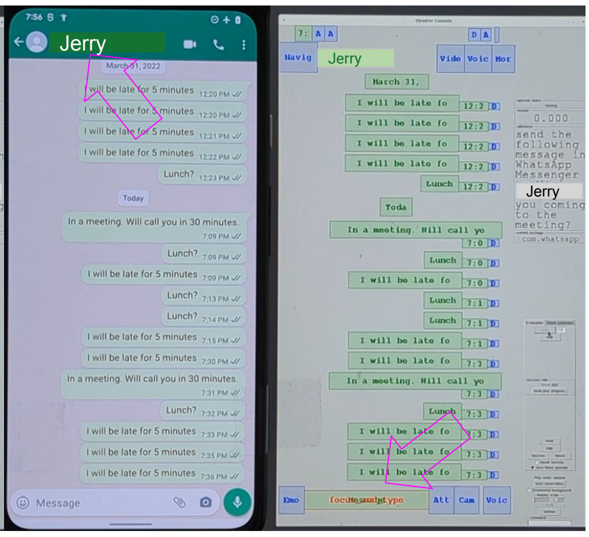


(d)

Figure 10: The UINav agent sends an email: (a) Clicks the compose button; (b) Types the email address; (c) Types the subject; (d) Types the email content. The action of clicking the send button is not shown due to space limitation.

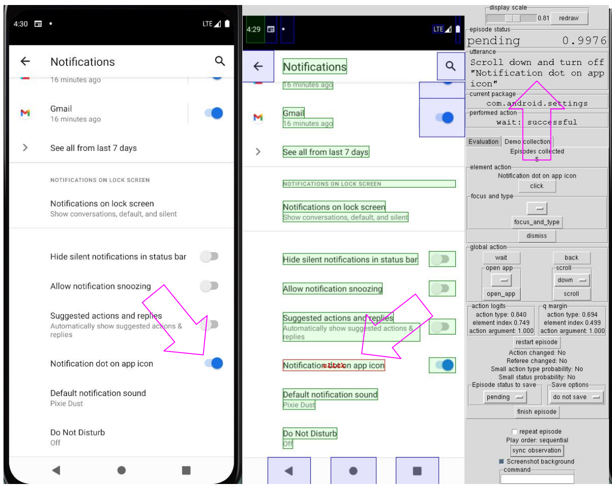


(a)



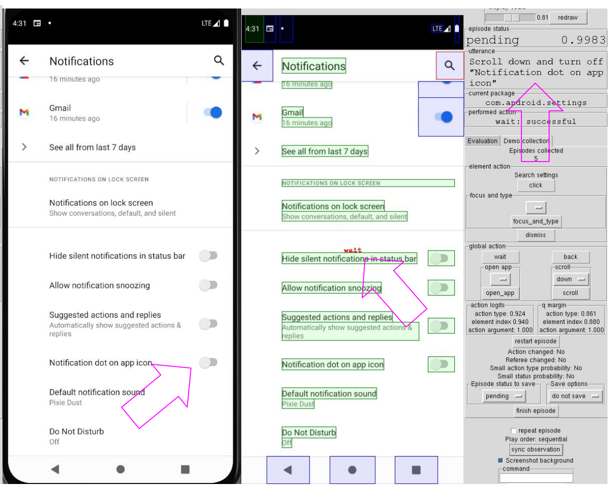
(b)

Figure 11: Two cases of an agent sending a message. The task description is “send the following message in WhatsApp Messenger to Jerry: Are you coming to the meeting?”. (a) In the message view to a different recipient from the one in the utterance; (b) In the message view of the same recipient as the one in the utterance.



Notifications
Gmail
See all from last 7 days
Notifications on lock screen
Hide silent notifications in status bar
Allow notification snoozing
Suggested actions and replies
Notification dot on app icon
Default notification sound
Do Not Disturb
pending
0.9976
Scroll down and turn off "Notification dot on app icon"
com.android.settings
wait: successful
Notification dot on app icon
click

(a)



Notifications
Gmail
See all from last 7 days
Notifications on lock screen
Hide silent notifications in status bar
Allow notification snoozing
Suggested actions and replies
Notification dot on app icon
Default notification sound
Do Not Disturb
wait
pending
0.9983
Scroll down and turn off "Notification dot on app icon"
com.android.settings
wait: successful
Search settings
click

(b)

Figure 12: An agent selects an action to turn off notification dot (a) when the switch is on, and (b) when the switch is already off. The texts in red (click in a) and wait in b)) are the actions selected by the agent.

the subject: Events for the Week and the content: Hi, you are invited to attend our weekly meeting from 4:00pm to 5:00pm on Tuesday (EST - New York)".

Sending a message to the correct recipient. Fig. 11 compares two cases of an agent sending messages. The images are deliberately modified to hide the real names of the recipients. Both (a) and (b) are in the message view of the app but of different recipients, Tom in (a) and Jerry in (b), while the utterance specifies the recipient to be Jerry. The agent correctly recognizes the difference and selects the correct action for both cases: pressing the back button at the top left for (a) and typing the content of the message at the bottom for (b). Note that it is the title bar that contains the information on the current recipient. We believe that it is due to the self-attention of the Transformer encoder that the agent learns whether the text of the title bar matches the recipient is a critical signal in these states.

Understanding the relationship between text label and switch. Fig. 12 shows how the UINav agent selects actions to turn off notification dot in two cases: (a) when the switch is on and the agent selects the action to click the text label of "Notification dot on app icon", and (b) when the switch is already off and the agent chooses to wait for the referee to terminate the task. Note that the text label of "Notification dot on app icon" and its switch are independent UI elements in the screen representation, and there are multiple switches on the screen with identical attributes except for their positions and states. The agent learns their relationship probably by the relative positions (horizontally aligned).

A.9 Apps and websites used in data collection

The full list of Android apps and websites that are used in our data collection is as follows:

Facebook Messenger, TikTok, Instagram, WhatsApp, Amazon Shopping, Facebook, Walmart, Spotify, Pandora, Amazon Prime Video, Google Play Games, Wish, Pinterest, Google Messages, Target, Poshmark, Waze, Twitter, Wayfair, google.com, Google Play Store, Seamless, YouTube, Reddit, Ebay, Etsy, Soundcloud, Tasty, Gmail, Contacts, Android Auto, YouTube Music, Snapchat, Tubi TV, Shop, News Break, Cash App, Pluto TV, Uber, Burger King, Roku, Amazon Alexa, Life 360, HBONow, ESPN, iHeartRadio, Nike, Amazon Photos, Letgo, Walmart Grocery, Weather App, Google News, Files, Home Screen, Google Docs, DoorDash, Google Photos, AirBnB, AliExpress, Amazon Music, Apple Music, Audible, Chewy, Chik Fil A, Costco, Dollar General, Google Drive, Dunkin Donuts, Google Earth, Emoji Home, Family Dollar, wikipedia on firefox, Food Network, GroupMe, Groupon, GrubHub, Instacart, KeepNotes, King James Version, Kroger, Likee, LinkedIn, fb Lite, Lyft, Maps, OfferUp, Phone, Pixaloop, Scanner, SHEIN, Skype, SmartNews, Starbucks, thredUp, Ticket Master, Walgreen's, Yahoo Mail, Yelp, YouTube Kids, Zedge, Zelle, Zillow, wikipedia.org, youtube.com, yahoo.com, facebook.com, live.com, reddit.com, bing.com, linkedin.com, Sam's Club, discord, GoodRx, Outlook, Breaking US News, Lucky Go, CNN, Postmates, Transit, Sephora, target.com, twitter.com, irs.gov, craigslist.org, homedepot.com, Recipes Home, Zillow, and Dialer.