

Learning to Automate GUI Tasks from Demonstration

Thanapong Intharah

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
of
University College London.

Department of Computer Science
University College London

2nd October, 2018

I, Thanapong Intharah, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the work.

Abstract

This thesis explores and extends Computer Vision applications in the context of Graphical User Interface (GUI) environments to address the challenges of Programming by Demonstration (PbD). The challenges are explored in PbD which could be addressed through innovations in Computer Vision, when GUIs are treated as an application domain, analogous to automotive or factory settings. Existing PbD systems were restricted by domain applications or special application interfaces. Although they use the term *Demonstration*, the systems did not actually see what the user performs. Rather they listen to the demonstrations through internal communications via operating system.

Machine Vision and Human in the Loop Machine Learning are used to circumvent many restrictions, allowing the PbD system to *watch* the demonstration like another human observer would. This thesis will demonstrate that our prototype PbD systems allow non-programmer users to easily create their own automation scripts for their repetitive and looping tasks. Our PbD systems take their input from sequences of screenshots, and sometimes from easily available keyboard and mouse sniffer software. It will also be shown that the problem of inconsistent human demonstration can be remedied with our proposed Human in the Loop Computer Vision techniques.

Lastly, the problem is extended to learn from demonstration videos. Due to the sheer complexity of computer desktop GUI manipulation videos, attention is focused on the domain of video game environments. The initial studies illustrate that it is possible to teach a computer to watch gameplay videos and to estimate what buttons the user pressed.

Contents

1	Introduction	20
1.1	Research Questions	24
1.2	Contributions of This Thesis	25
1.3	Scope of the Thesis	27
1.4	Research Papers	27
2	Literature Review	29
2.1	Traditional Approaches to End-user Program Synthesis	29
2.1.1	Programming by Demonstration	30
2.1.2	Other End-user Program Synthesis Approaches	33
2.2	Other Approaches to Comparable Domains	35
2.2.1	Digital Game Domain	35
2.2.2	Web Search Engine Domain	37
2.3	Background Knowledge	38
2.3.1	GUI Analysis	38
2.3.2	Joint Segmentation and Classification of Action	40
2.3.3	Discovering Looping Pattern	41
3	Visual-based Programming by Demonstration	43
3.1	Overview of Challenges	44
3.2	Learning to Perform Tasks from Demonstration	46
3.2.1	Demonstration Phase	46
3.2.2	Teaching Phase	49

3.2.3	Running Phase	57
3.3	Evaluation and Results	58
3.3.1	User Study Scenarios	61
3.4	Discussion and Future Work	68
4	Looping GUI Action Automation	71
4.1	Overview	74
4.2	Looping Action Recognition	74
4.2.1	Basic Motif Finding	76
4.2.2	Distance Between Two Sequences $\text{Dist}(\mathbf{S}_i, \mathbf{S}_j)$	76
4.2.3	The Proposed Method for Multiple Motif Finding	78
4.2.4	Artificial Subsequences for Robustness	79
	Looping Action Recognition	80
4.3	Prediction of Future Actions	81
	Generating Artificial Subsequences	82
4.3.1	Human-in-the-loop	84
4.4	Datasets	85
4.4.1	Demonstration Dataset	85
4.4.2	Looping GUI Automation Dataset	85
4.5	Validation of the Algorithms	86
4.6	User Feedback	92
4.7	Conclusions and Future Work	93
5	Generating Log-file from Video	95
5.1	Baselines and Challenges They Face	97
5.1.1	Baseline: Human Performance	97
5.1.2	Challenge: Class Imbalance	98
5.1.3	Challenge: Multiple Control Buttons Per Record	100
5.1.4	Challenge: Many-to-One	101
5.2	Architecture	101
5.2.1	The Network	103

5.2.2	3D convolution layers	103
5.2.3	Training	104
5.2.4	Losses	104
5.3	Experiments and Results	105
5.3.1	Data	105
5.3.2	Performance Metrics	106
5.3.3	Result: Overall Results	107
5.3.4	Result: Class Imbalance	111
5.3.5	Result: 2D VS 3D Filters	111
5.3.6	Result: Many-to-One	111
5.3.7	Result: Generalization	112
5.4	Discussion and Conclusion	112
6	Conclusions	116
6.1	Future Research Directions	117
	Appendices	120
A	Pseudo-codes	120
	The Looping GUI Automation Dataset Evaluation Protocol	121
B	Datasets	123
B.1	Demonstration Dataset	123
B.2	Looping GUI Automation Dataset	124
C	Looping GUI Automation Evaluation and Datasets	126
C.1	Automating SMS sending	127
C.2	Adding contacts on phone from spreadsheet via 3rd party app	130
C.3	Saving area chairs' homepage as PDFs (icons clicked in regular order)	133
C.4	Saving area chairs' homepage as PDF's (icons clicked in random order)	136
C.5	Renaming files on Google Drive	139
C.6	Deleting specific files on a cluttered desktop	142

C.7 Deleting files in folder (smaller icon) 144

C.8 Creating list of filenames from a folder (files selected in regular order) 146

C.9 Creating list of filenames from a folder of remote computer (regular ordered) 148

C.10 Creating list of filenames from a folder of remote computer (random ordered) 150

C.11 Creating Slides of images from folder of images 153

C.12 Zipping every file in a folder 156

C.13 Unzipping every file in a folder and renaming the files to the names of zip files 158

C.14 Taking screenshots of list of websites 161

C.15 Taking screenshots of list of websites on mobile 163

Bibliography **165**

List of Figures

1.1	Diagram of conventional Programming by Demonstration systems. The system has two limitations as follows. First, the PbD system listens to the demonstration, instead of watching, so they need an Accessibility API to perceive messages about the demonstration. Second, The PbD system needs an application's Open API to operate the application widget.	22
1.2	Diagram of our proposed Visual-based Programming by Demonstration system. The system watches the demonstration from the same screen as the user and operates directly at a specific location navigated by visual features.	26
3.1	Flow of the system. Details for each phase are described in the text.	45
3.2	Example of a log-file which merges the inputs from both video and sniffer data.	47
3.3	The system workflow for the teaching phase. The yellow boxes indicate where the system poses questions to the teacher.	50
3.4	Examples of a segment, a time interval, a key frame, and an input to SVMs. 'L' indicates the left mouse button is pressed, 'R' indicates the right mouse button is pressed, and '-' indicates none of the mouse buttons are pressed.	52

- 3.5 Target patterns in (a) & (b) are distinguishable on their own. The spreadsheet cells in (c) need row and column names to differentiate between one another. Text fields in registration forms in (d) can be distinguished by the text field labels. Supporter helps distinguish locally ambiguous patterns. 53

- 3.6 An example of a supporter for a looping task. The table shows names of characters and actors/actresses of a popular TV show. The names in each column have similar appearance, so, if the user intended to loop through one of the columns, marking the column name as a supporter will help the system to distinguish between columns. 54

- 3.7 An example of a spatial supporter. Blue boxes are user provided positive examples, red are user provided negative examples, yellow are target detections, and a green box indicates a user-provided supporter. (a) and (b) demonstrate detection performance without and with a spatial supporter. Red color in the heatmaps means a high detection score. In (a), the left image shows the heatmap of target detection scores, and the right image shows detected targets. In (b), the left image shows the heatmap of target detection when combined with the spatial supporter scores, and the right image shows detected targets. The spatial supporter successfully suppressed all similar looking patterns under the Character column in heatmap (b) so that the system is able to detect only desired targets under the Actor/Actress column. 54

3.8 Example screenshots of HILC when it asks users for supporters. (a) HILC asks users to add supporters when there are confusing patterns (red boxes) which look similar to the user intended patterns (green box). In this case, NCC scores of the confusing patterns are higher than a threshold. The system uses only the NCC detector for the intended pattern at the Running Phase, unless the users provide supporter(s). (b) HILC asks users to add supporters when a confusing pattern (red box) has the same NCC score to the intended pattern (green box). Unless the users provide at least one supporter, HILC trains RF detector for the intended pattern. 56

3.9 An example screenshot of HILC when it asks users for additional information during the Teaching Phase of looping tasks. 57

3.10 An example screenshot of HILC when it requests a user to provide the visual cue (yellow box) which is the visual pattern that invokes the system to run the rest of the script whenever the system finds it. 57

3.11 The system workflow for the running phase. The yellow box indicates user interaction. 59

3.12 The instructor clicked on the speaker in the green box, but the system also detected a similar pattern - the speaker in the red box. In this situation, the system asks the teacher for a supporter(s), the yellow box, to help with detecting the intended pattern. 62

3.13 High-Contrast-Mode comparing with Normal Mode and (in the red box) the transcribed steps of the task demonstrated by HILC. 62

3.14 Steps to complete remote access via TeamViewer. Red lines link related patterns on the screen with the pattern in the transcript. It is noteworthy that performing the basic action DragTo from and to the same pattern has a similar effect as performing the basic action Click on that pattern. Participants often unintentionally perform the basic action Dragto instead of the basic action Click. HILC is robust to this type of different-but-interchangeable action. 63

- 3.15 YouTube Skip Ad. These advertisements show before or during a playing video for varying periods of times, and HILC successfully closes them in Scenario 4, as soon as the text appears. The visual cue is highlighted by the magenta rectangle. 64

- 3.16 Close-ups of YouTube Ads. These ads appear at the bottom of a playing video, and HILC detects and successfully closes them in Scenario 5. The visual cues are pointed by magenta arrows. 65

- 3.17 Scenario 6: create slides from folder full of images. The generated script is shown in the red frame. HILC starts by building a list of locations that will be the starting points for each iteration. The list is formed by the Trained RF, which trained and refined in the teaching phase with a few examples stemming from the demonstration phase. The system then iteratively executes a sequence of actions from line three to five (DragTo, Click, Click). In this scenario, the two applications are displayed side-by-side. 66

- 3.18 Two application screens from Scenario 7, where file names are being collected into a spreadsheet. The script of the task, in the red frame, involves switching back and forth between the two applications, and pasting the text into similar-looking cells. 67

- 3.19 A synthesized script of Scenario 8, where a BibTex file is automatically constructed from a list of paper titles. Three different desktop GUI's were involved. The user was able to train the system quite easily, and can just run the task without further instructions when writing her next research paper. 68

- 3.20 HILC successfully use videos of a screencast software as input of the system, instead of generating the input log-file from the sniffer, to create a working script. It is noteworthy that the system failed to remove mouse pointer from the target patterns in the first and the seventh lines. 69

- 4.1 A looping GUI task, where the user is renaming files on Google Drive to match names in an Excel spreadsheet. This type of task is long and tedious, and hard for a typical computer user to automate. Our system is designed to learn to complete such tasks by watching a user performs a few demonstration iterations. 73
- 4.2 An example problem setup. This diagram shows that the proposed system allows detour actions, Windows' popup asks for restarting the system, that can accidentally happen during the demonstration process. Without that mechanism the user need to re-record everything again from scratch. It is noteworthy that the diagram shows extracted objects, PDF file icons, buttons, *etc.*, which linked with actions instead of in reality the system only has access to whole screen screenshots and their corresponding mouse pointer locations. 73
- 4.3 The proposed future action prediction algorithm chooses *where* to apply action events, based on the small number of user demonstrations. (a) If the demonstration events follow a linear pattern, the spatial prior uses linear regression to predict action locations. (b) If the demonstration events do not exhibit spatial correlation, then the spatial prior becomes uniform and the pattern matching likelihood becomes dominant. Here, it learns to locate the '.pdf' files on a cluttered desktop. 83

4.4 Quantitative results of the proposed looping action recognition algorithm on the Demonstration Dataset. The proposed algorithm, NCC+Noisy+Missing, is compared against three baselines: Division, greedMotif, and Motif algorithms. GT in the name indicates that the algorithm has access to the ground truth distance matrix instead of using Eq 4.2, while NCC indicates that the algorithm use the proposed Normalized Cross Correlation as the distance function between two visual objects. GT Motif, NCC Motif and GT greedMotif share the same graph at {GT,NCC}-Motif, GT greedMotif. GT+Noisy+Missing demonstrates the accuracy of the proposed algorithm when it has access to the ground truth distance matrix. NCC+Noisy is the ablation study, showing the result when the artificially appended actions is removed. 88

- 4.5 Prediction / execution / improvement of future actions. **Top:** An illustrated event history in 14 steps. In this unseen test task the user demonstrated three loops, each starting with a left-click on a hyper-link (shown as red boxes). Boxes in blue show the bot’s subsequent predictions. For each such prediction (*e.g.*, bot 4), the user presses the spacebar to confirm they are happy for the system to proceed automatically. If the prediction’s score falls below the stopping threshold, the system asks the user to approve, correct, or additionally, to terminate — these events are labeled as green boxes followed by a question mark. “hum X” shows when the human terminated the loop. Here, “Bot 5 ?” asked the user to confirm whether the pattern in the box is the next target; the user instead corrected the system by specifying the next correct target as “hum 6”. **Bottom:** The figure shows what the user sees when autocompleting the task. The system visualizes the action it will take next, rendered as a virtual mouse-arrow. It asks the user for approval through large rendered messages, though two-way audio interfaces could be easier for other users to access. 89
- 4.6 An example output of the proposed looping action recognition algorithm. The task here is to make a list of filenames from a folder of files. The system outputs a rendering showing discovered loops in rows and matched actions across loops in the same column. Numbers before the actions indicate the order in the input sequence. Here, there are three missing actions and one noisy action (not shown), all of which were detected by the algorithm. The images are extracted from the screenshots at the position each action was performed. It can be seen here that the variation in spatial location of user interaction; images within each column are shifted, or worse. It is this variation that makes it hard to parse a user’s demonstration. 90

4.7 Another example that can be easily completed by RecurBot: Adding mobile contacts from a spreadsheet program via an Android remote access program, Vysor [28] 90

5.1 Classical titles: Tetris from Nintendo Entertainment System (NES) and Mega Man X from Super Nintendo Entertainment System (SNES) are used in the experiments. Cover art © Nintendo Co., Ltd. 97

5.2 Example of a question in the questionnaires: an example video clip of Tetris gameplay, with check-boxes for a user to indicate what buttons they think were pressed at the middle frame. While this is a demanding and time-consuming task, users were fairly successful when “transcribing” Tetris. 98

5.3 Button-combination Frequencies for Tetris (a) and Mega Man X (b). For Tetris, the dominant input is Idle (nothing pressed), followed by three main buttons: Down, Left, and Right. For Mega Man X, the two main classes are Idle and Shoot, followed by the “Right and Shoot” combination. Y-axis represents button-combination (Class) and X-axis represents frequency of the combination. 99

5.4 Functionally equivalent button-combinations for Tetris, shown in (a), and Mega Man X shown in (b), are visualized through confusion matrices. Each numerical entry indicates how many times the button-combination (of that row) produces the same visual output as another button-combination (column). 102

5.5 Detailed analysis of DeepLogger network’s performance on Tetris for per button and per class prediction. (a) shows the performance of per button prediction and (b) shows the performance of per class prediction. In both cases, rotation is the hardest to recognize. Small red X’s on the per button accuracy chart indicate there is no ground truth label for the buttons (Up and Select). 108

- 5.6 Detailed analysis of DeepLogger network’s performance on Mega Man X for per button and per class prediction. (a) shows the performance of per button prediction and (b) shows the performance of per class prediction. Binary labels in (b) along the horizontal axis represent *pressing (1)* and *not pressing (0)* that game controller button and The Binary codes preserve the button-order from (a). Red X’s on the per button accuracy chart indicate there is no ground truth label for the buttons (R and Select). X-axis represents button-combination (class). 109
- 5.7 Comparing loss functions. This bar-chart compares two DeepLogger networks using four performance metrics. For the red bars, the network was trained with normal multi-label loss, and for the blue bars, the network was first trained with normal multi-label loss on 90% of the training data, and then fine-tuned with multi-label-multi-choice loss on the remaining 10% of the training data. This bar-chart is generated from the Tetris gameplay dataset. Under each measure, fine-tuning with multi-label-multi-choice loss yielded better scores. Y-axis represents prediction accuracy. 112
- 5.8 The graph visualizes Euclidean distances between embedding features, produced by the ”Dense9” layer of the network, for the query clip and other clips from different videos. Min1, Min2, and Min3 clips are examples of the clips which have the smallest distances to the query clip. Y-axis represents distance and X-axis represents time steps in the video of the retrieved gameplay. 115

List of Tables

3.1	User study on HILC compared to Sikuli Slides. Scenario 3.2 is an alternative way to perform Scenario 3, without pressing shortcut key combinations that Sikuli Slides is known to be missing. Nevertheless, we eventually realized that Sikuli Slides is not detecting the right click actions either. (✓= successful, ✓* = partially successful, ✓** = can be successful with guidance from the operator, ✗= can not succeed at the task at all). x represents the number of repeated loops needed to complete the task. Please note that 90% of the refining time for Task 9 is offline - devoted to the time spent on processing video to produce the log-file.	60
4.1	Average statistics of the proposed Demonstration Dataset. These data serve for training and testing of just the analysis part of the visual motif-finding, analogous to typical (non-visual) motif-finding challenges. Here, each sequence is a unique task, made of basic GUI interactions (Actions), performed by 7 different computer users. Test users performed the first 3 or 4 loops of each task, and these were labeled to quantify experimental performance. Loops within the same sequence naturally differ from each other by having extra, missing, or iteratively changing actions. <i>% Variation Seq</i> is the % of sequences that have at least one user variation, either noisy or missing.	75

4.2 Quantitative evaluation of the task completion system on the Looping GUI Automation Dataset. After training each task with 3 demonstrated loops, how many actions were correctly predicted and automatically run (Correct (Auto)). How many action were correctly predicted but the score fell below threshold, so need user approval (Correct (Approval)). If the system make mistake on the prediction and the confident score is below threshold, the system wait for user approval or modification (Incorrect (Approval)). If the system make incorrect prediction with high confident, it is counted as “Incorrect”. Qualitative views of action prediction are shown in Figure 4.5 and more detail on the tasks can be found in appendix C. 91

4.3 Results of the user study on the prototype system. Each question was scored on a Likert scale out of a maximum of 7, and the questions in each of the four categories were averaged for display in this table. 93

5.1 The performance of both human experts and the proposed DeepLogger system, on estimating a gamer’s controller inputs (the log) from gameplay videos only: Tetris and Mega Man X. The criteria (rows) are explained in the text, but higher accuracies and F1 scores are better. Humans are better with Tetris videos, while DeepLogger does better with Mega Man X, possibly due to the UI complexity. 100

5.2 Diagram of the proposed DeepLogger Network where D is the number of frames per clip, which are 21 frames and 11 frames for Tetris and Mega Man X respectively. $W \times H$ are the image dimensions of the gameplay videos, which are the default screen dimensions of NES, 256×224 for Tetris, and SNES, 586×448 for Mega Man X. C is the number of buttons, with 8 buttons for Tetris and 12 buttons for Mega Man X. 103

- 5.3 Performance of 3 CNN's: DeepLogger, DeepLogger2D, and VGG on Tetris and Mega Man X. Figures in parentheses indicate performance of the networks when they were trained without oversampling the non-majority classes. 110
- 5.4 Generalization evaluation. *Base* is the performance of the system when testing on gameplay video from the same level, gamer, and encoder. *Diff 1 level* shows the performance of the system when testing with gameplay videos from a different level than the training data, by 1 level, *Diff 2 levels* tests the same thing, but where the difference is 2 levels. *Diff Gamers* level shows the performance of the system when testing with gameplay videos from different gamers. *Diff Encoder* level shows the performance of the system when testing gameplay videos downloaded from video-sharing site YouTube, where compression and frame-rate changes can occur. . . 113

Chapter 1

Introduction

The Graphical User Interface (GUI) is one of the most important innovations that helped make personal computers (PCs) prevalent in modern society. It allows a wider range of users to access and interact with the complex algorithms behind the user-friendly surface of the computers. Prior to the wide-availability of GUIs, text-based interfaces were most commonly used. Using text interfaces typically requires users to have knowledge of commands specific to each operating system (OS), thus limiting its use to expert users only. GUIs make it easier for non-expert users to use computers by alleviating the need for knowledge of OS-specific commands. Instead, GUIs are often intuitive and provide special tools for simple and commonly used tasks.

However, one of the major disadvantages of GUIs is that they are not as flexible as text-based interfaces. One specific instance where text-based interfaces are still preferred to GUIs is for the execution of repetitive or looping tasks. Repetitive tasks are those where a user repeatedly performs the same set of steps every time he/she executes those tasks, *e.g.*, turning on/off high contrast mode. The tasks might be easy but require several steps to complete which can be troublesome for people who have certain disabilities or lack of knowledge. Looping tasks, on the other hand, are tasks which require users to apply a similar set of steps to several different objects, which have a similar pattern, in one execution, *e.g.*, sending different messages with a similar pattern to each individual persons listed in a spreadsheet. Although each iteration of looping tasks usually takes fewer steps compared to repetitive tasks, one

of its challenges is to generalize across all iterators which are slight different among one another from a few examples.

A task discussed throughout this thesis are defined as a sequence of basic desktop actions, e.g. left click, right click, and key press, which are needed to be performed to achieve a user's goal. Examples of the tasks are turning on high contrast mode (Linear task), removing all pdf files in a specific folder (Looping task), and closing an Ads. on YouTube when it appears (Stand by task).

To automate a task, users need special mechanisms to interpret their intention into scripts. The mechanisms range from a programming language to a special recording program which can record steps of the task and playback. Currently, GUI task automation is achieved through the use of special APIs provided by the OS or applications. Thus, the automation power of the computer is limited not only by programming skills but also available tools. Accordingly, most PC users, who need to rely on GUIs, have not used their machines to their full capabilities. Countless day-to-day tasks are tedious and repetitive. It is the fact that computers are good at executing repetitive and looping tasks. So why are we, humans, the ones who have to perform all of those tedious tasks, instead of the computer?

Many attempts have been made to make GUI task automation possible, e.g., Sikuli [95], OSX's Automator [4], and CoScriptor [55]. Programming by Demonstration (PbD) is one of those paths to achieve the full automation machine goal. PbD takes natural human demonstration process(es) as an input, and produces a script which performs the automation, instead of asking users to write cryptic computer codes, or explicitly list steps by following a predefined set of rules. In other words, PbD systems synthesize automation scripts by observing users' demonstrated examples. The advantage of this approach is that users of PbD systems do not need to have any programming skill in order to create an automation script and the script creation process takes no more than the time taken to perform the task manually.

Although many applications include features that enable task automation by user demonstration, the tasks are restricted within a group of applications which

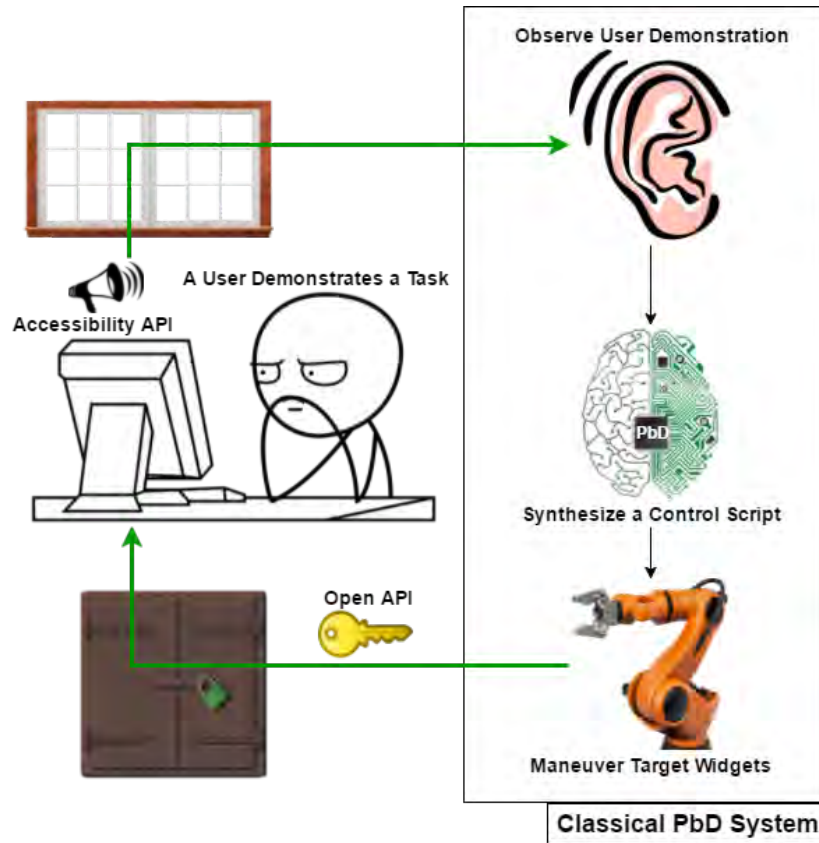


Figure 1.1: Diagram of conventional Programming by Demonstration systems. The system has two limitations as follows. First, the PbD system listens to the demonstration, instead of watching, so they need an Accessibility API to perceive messages about the demonstration. Second, The PbD system needs an application's Open API to operate the application widget.

were implemented with the accessibility API functionalities. A conventional PbD system is demonstrated in figure 1.1.

To enable cross-application communication, all involved applications have to conform to special APIs. First of all, an application has to be developed under an Accessibility API framework to allow other applications, in this case the PbD systems, to *perceive* which of its GUI widgets are activated by users. Moreover, the application has to provide an Open API to let other applications *interact* with its internal components. Developing software/applications with those special APIs supported incurs considerably more cost and effort. Additionally, it is likely that only a fraction of applications were developed under these special API frameworks. Furthermore, some applications such as remote desktop applications stream videos

of the server-side screen to the client-side so it is almost impossible for the client-side to access the application's APIs of the host directly. This is also a limitation when working with mobile phones whose screens (only) can be operated from a Desktop PC.

The main motivation of this research is to get rid of the APIs completely to allow PBD systems to work across applications regardless of how the applications were developed. The research is divided into three phases.

First, the APIs are replaced with Computer Vision and follow-up questions in a prototype PBD system, HILC, which is described in Chapter 3. HILC is the first attempt which leverages Computer Vision to observe user demonstration instead of relying on the Accessibility API and also use it to guide actions at the execution time instead of relying on the Open API. In additions, follow-up questions are used to refine the computer vision model. However, demonstrating looping tasks in HILC requires special steps which are burdens for casual users. HILC needs the users to exactly indicate where are the start and the end of the loop and explicitly state what are examples of the iterators.

Thus, in the second phase, the research is focused on simplifying the demonstration of the looping task to allow more casual users to use the PBD system. In the research, a special motif discovery algorithm is devised to extract different parts demonstrated loops as well as example iterators. This novel algorithm is robust to user inconsistencies. As a result, RecurBot, a Computer Vision based PbD system which can learn more complex programming paradigms from simple demonstrations, is proposed. Chapter 4 presents detail of the algorithms, the datasets and results of the studies.

It has been shown in Chapter 3 and Chapter 4 that the APIs, which are heavily relied by existing PbD systems, can be replaced with Computer Vision and carefully designed human in the loop schemes. Although the APIs can be completely eliminated from the prerequisites of PbD system, the demonstration still has to be done on a system which equipped with a customized sniffer program.

The aim of the last phase is to learn user demonstration from a recorded video.

Hence, the demonstration can be done on any machine. In other words, the sniffer program requirements are lifted. In this phase, however, the study is limited to GUIs of two 2D game environments, NES and SNES, because input actions of the game environments are tractable comparing to the input actions of the computer desktop environment. This initial study illustrates challenges which are faced when designing a system to observe interactions between the users and the GUIs from a video. These interactions are deemed as user demonstration in this context. The study is reported in Chapter 5. Solutions to the challenges are also discussed.

1.1 Research Questions

The core research question which is posed and attempted to validate in this thesis is “*Does Human in the Loop Machine Learning help casual users train a computer to perform general GUI tasks?*”. *Human in the Loop* describes a machine learning technique which benefits from human interaction with the system or human feedback during the training process. *Machine Learning*, which includes Computer Vision, leverages data, mathematical models, and optimization to achieve the prescribed goal. *Casual users* are computer users, whether inexperienced or experienced, who attempt to demonstrate the GUI tasks to train the computer. In this research, it was found that while the users are performing the GUI tasks, the demonstrations are frequently, though unintentionally, inconsistent. For example, users perform two identical tasks differently at different moments depending on states of the computer. Users occasionally click on different locations of the same object when they perform a task. The goal is to build a system which is robust to these inconsistencies. *Train* is defined as the methods used to feed inputs to the system. *A computer* is a separate system, which learns from the user demonstration data to automate the GUI task. *General GUI tasks* in the statement emphasizes that the system can automate GUI tasks independently of OS and Application, regardless of special API restrictions or on-screen content. This very point makes this system unique because existing PbD systems relied on either Accessibility APIs or Open APIs, or both and the systems ignore non-widget screen elements.

This thesis touches the main research question in three aspects. First, how can the special API restrictions of PbD systems be lifted? This question is answered in Chapter 3 by proposing HILC, the first API-less PbD system. Second, what make the demonstration hard for casual users and how that can be remedied? The answers to these questions are discussed in user studies and the development of RecurBot in Chapter 4. Third, is it possible to remove all instrumented tools needed to observe the demonstration? The detailed study of this question is presented in Chapter 5 where DeepLogger is proposed.

The next section lists and discusses contributions of this thesis to different computer science areas.

1.2 Contributions of This Thesis

The main research question is explored on three different aspects which lead to the following contributions.

The first contribution of the thesis is to propose a solution for the special API restrictions of existing PbD system by building a prototype visual-based PbD system that automates users' simple GUI tasks. This prototype takes as input GUI screenshot images and basic mouse-keyboard events. One assumption is that the mouse-keyboard events, which can be easily retrieved from the OS, are available. Machine Vision techniques are used to compensate the needs of the Accessibility API, and then directly inject real mouse-keyboard-action signal back to the system to avoid the need of Open API. Figure 1.2 demonstrates the proposed Visual-based Programming by Demonstration system, which allows the proposed PbD system to effortlessly work across applications regardless of how demonstrated applications were developed. In other words, the system is not restricted only to the applications which were developed with a special API framework in mind like existing PbD systems discussed in detail in Chapter 2.

Secondly, in practice, users trust a system when they know what the system is going to do and they have full control over it. By integrating the proposed Human in the Loop framework to the system, it builds a user's trust and also benefits from

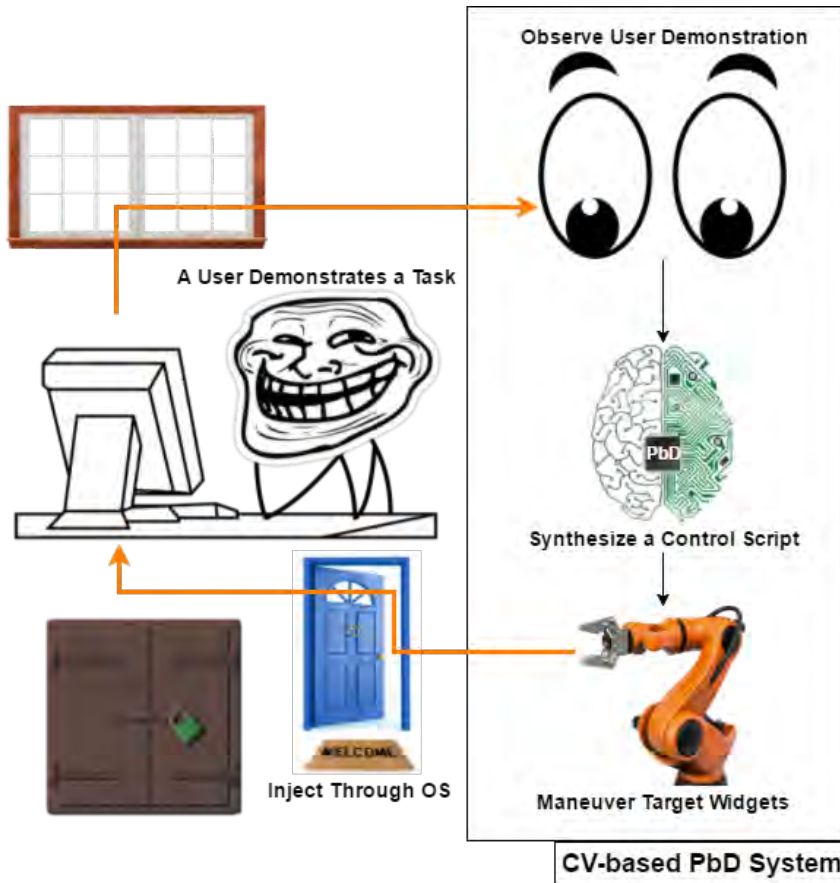


Figure 1.2: Diagram of our proposed Visual-based Programming by Demonstration system. The system watches the demonstration from the same screen as the user and operates directly at a specific location navigated by visual features.

the user's feedback. Further, two kinds of tasks users wish to automate, *Repetitive Tasks* and *Looping Tasks* are studied in detail. In response to a user study in Chapter 4, a novel motif discovery algorithm which is robust to user inconsistencies in the demonstration phase is invented. The new PbD system simplifies a user's demonstration process by mirroring how a human teaches another to do GUI tasks. To make this research reproducible and pave the way for further research, novel GUI task automation datasets are also proposed, along with their companion toolkits.

Lastly, the use of demonstration videos as the only input to a PbD system are explored. All of the works which have been done so far in Chapter 3 and Chapter 4 takes as input a screen-captured images and a log-file, produced by tailor-made sniffer software that captures sequences of screenshot images and corresponding

mouse-keyboard input signals. To extend an ability of PbD systems to learn from richer sources, such as online instructional videos, the possibility of producing the same log-file by analyzing just a demonstration video is studied.

In this research, the study commences with less complex environments of the two systems of two different game genres, the Super Nintendo Entertainment System (SNES) for the game Mega Man X and the Nintendo Entertainment System (NES) for the game Tetris. The aim is to predict a button-press combination for each frame of an input gameplay video for two different game genres. The study addresses explicit and implicit challenges which hurt the prediction quality of both human and the computer. Convolutional Neural Networks (CNNs) and their training strategy specially designed to address the challenges are proposed and evaluated. Additionally, human experts prediction quality is also discussed in Chapter 5.

1.3 Scope of the Thesis

1. Due to very complex environment and subtle visual signals of desktop environments, Chapter 3 and Chapter 4 only focus on perceiving user interaction via a simple custom made sniffer.
2. For proposed Programming by Demonstration systems, only linear task, looping task and standby task are programming paradigms which were addressed in this thesis.
3. Pure computer vision approach, without using sniffer software, is only applied to two game environments: Mega Man X (SNES) and Tetris (NES).

1.4 Research Papers

Since the start of PhD journey of the author, three publications had been produced and published in three premier international conferences on Human-Computer Interaction.

1. “DeepLogger: Extracting User Input Logs From 2D Gameplay Videos” was accepted to present in CHI PLAY 2018 conference. This paper explains ex-

tensive experiments on training a CNN to produce a log-file of a video of gameplay. Chapter 5 presents detail of the work.

2. “*RecurBot: Learn to Auto-complete GUI Tasks from Human Demonstrations*” was presented at CHI 2018 conference as a Late Breaking Work poster presentation. In this work, The focus is on making looping tasks of the PbD system robust to users inconsistencies during the demonstration phase. More detail of the work can be found in Chapter 4.
3. “*Help, It Looks Confusing: GUI Task Automation through Demonstration and Follow-up Questions*” was presented in the ACM IUI 2017 conference, and was awarded the best student paper honorable mention. In the paper a prototype PbD system which analyzes sequences of GUI screenshot images and corresponding mouse-keyboard events to synthesize automation scripts is presented. This work can be mapped to Chapter 3.

Chapter 2

Literature Review

In this chapter, classical approaches to the end-user program synthesis are presented. The end-user program synthesis is defined as a set of approaches which allow end-user of applications to create automation scripts to complete the application related tasks. Moreover, other works related to understanding user intent from interaction log are reviewed. Furthermore, backgrounds of useful concepts which are used through out this report are described. In section 2.1, there are extensive reviews of existing Programming by Demonstration (PbD) systems, and explorations of other alternative end-user program synthesis approaches. In section 2.2, Other approaches on comparable domains are discussed. Although these works are not closely related to the Desktop GUI automation, they are worth exploring because many approaches can be applied to different domains. In section 2.3, all useful concepts upon which the proposed systems are built are explained. The important concepts are GUI image analysis, sequence decoding, looping action recognition, and time series motif discovery.

2.1 Traditional Approaches to End-user Program Synthesis

The goal of end-user program synthesis is to give tools for the end-users to create their custom-made computer scripts to perform personal tasks on the related applications without modifying the applications internal source code. For example, users might always want to use specific printer setup when they print documents

from Microsoft Excel. This setup contains a sequence of settings, which is different from the default setting for other applications, the users need to apply before printing from Microsoft Excel. With the end-user program synthesis, they could create a script which, when invoked, the script selects printing menu in Microsoft Excel and it then sets everything up for the users. The end-user program synthesis can be categorized by their means of script creation *e.g.*, writing applications specific codes, providing input-output examples, and demonstrating how to complete the tasks. Programming by Demonstration (PbD) is one of the end-user program synthesis approaches which allows the users to create a script by demonstrating steps to complete the intended task.

Generally, a PbD system observes a user performing a task, sequentially manipulating GUI widgets of target applications. The system then synthesizes the script which will manipulate elements of target applications at the execution time by following the observed steps. Many programs, for examples Microsoft Excel and Adobe Photoshop, provide users functions to record macros of their tasks. Nevertheless, one common drawback of existing PbD systems is that the systems require target applications to conform to special APIs in order for the PbD systems to perceive the demonstration and to manipulate the widgets at execution time. Hence, they rarely work across applications. In 2.1.1, existing PbD systems are reviewed in terms of their functionalities and what distinguishes them from other systems. Furthermore, other approaches to the end-user program synthesis are discussed in 2.1.2.

2.1.1 Programming by Demonstration

Dated back to as early as the 90s, Cypher published the book, *Watch What I Do* [25], which compiles early Programming by Example (PbE) and Programming by Demonstration (PbD) systems until 1993. Works from that book paved ways for PbE and PbD system design and implementation. Examples of notable systems in the book include Pygmalion, the first PbD system which introduced the concept of creating a script by observing a user demonstrating steps of the task instead of having the user write abstract logic in a programming language; TELS, a system which predicts the next text editing action similar to what Microsoft Excel's Autofill does

nowadays; Eager, a PbD system that learns from a set of user demonstrated examples to produce the more generalized scripts; and Triggers, a system that uses visual information of the display screen to trigger keyboard and mouse macros. These system build concepts some of which are still used in current PbD systems. Current well known PbD systems are discussed below.

Sheepdog [19, 53] is a PbD system specially designed for IT support tasks. After observing IT experts demonstrate variations of procedures to complete a specific task, it then uses an Input/Output Hidden Markov Model to learn how to complete the task from the set of demonstrated sequences of actions. At the execution time, *Sheepdog* couples user interaction with the inference to guide the system on each step. Moreover, *Familiar* [77] was developed as a PbD system to automate iterative (looping) GUI tasks by learning from a few performed examples of the task. Nevertheless, target applications of both systems need to conform to OS accessibility APIs which restricted them to the closed environments while HILC proposed in Chapter 3 and *RecurBot* proposed in Chapter 4 are not restricted by the APIs.

CHINLE [22] allows applications which were developed under the SUPPLE framework [29] to automatically generate its own PbD functionality. The system takes as input SUPPLE's functional specification of the application's interface. Although the applications of the system are restricted to the SUPPLE framework, the paper studied extensively on an important problem which is recovering from user's mistakes at the demonstration phase. *CHINLE* allows users to modify the generated script before the execution phase. It is also confirmed in Chapter 4 that user demonstration inputs are brittle and users are willing to help preventing the script to run into disaster. In contrast to *CHINLE*, *RecurBot* solves the issue automatically with the proposed motif discovery algorithm.

Sikuli Slides [5] was developed as an extended work of *Sikuli* [95], described in detail in section 2.1.2, to simplify the GUI automation script creation process which allows less programming-skilled users to generate such scripts. Instead of coding with *Sikuli* script, *Sikuli Slides* represents a process as a PowerPoint slide presentation, listing each sequential step on a separate slide. This generated slide

can be executed as a script later by the system. Additionally, *Sikuli Slides* provides an action recorder, so a user can record the process by demonstrating and saving it as a starting draft of the *Sikuli Slides* script. Although the system process GUI screen-captured images instead of requiring access to the Accessibility API, similar to HILC in Chapter 3 and RecurBot in Chapter 4, the ability of the action recording feature is still very limited. For example, it can only interpret simple user events and linear tasks. The scripts generated by the action recorder are often incomplete and need modification from the users to work.

Nowadays, smartphones have become a part of human everyday life. There have been many attempts to allow users to create automation scripts which operate their smartphone [9, 2, 3, 1, 67, 58, 10]. Out of those works, *Keep Doing It* [67] and *Sugilite* [58] were developed under the Programming by Demonstration concept, both of them relying on Android Accessibility API to listen to users demonstrated events. *Keep Doing It* [67] analyzes users' mobile interaction logs to generate automation script for the task users intended to do. Toby et al. [58] recently proposed *Sugilite*. The system gives users many flexibilities such as users can modify the generated scripts later when the GUI is changed, creating forks for the conditional tasks. The system learns to generalize tasks well thank to the Android Accessibility API which not only allows the system to access the demonstrated events but also the software hierarchy of the involved applications. Unfortunately, These two systems rely on the APIs like existing PbD systems, so they also suffer from tasks involving web-based applications and poorly labeled alternative text applications.

It is a truism to say that most of the existing PbD systems hugely rely on the Accessibility APIs, except the *Sikuli Slides*'s action recorder which is still at the primitive stage. In this thesis, two PbD systems which successfully analyze visual data to observe user demonstrations instead of relying on the Accessibility APIs are proposed. This allows the systems to work across applications and to be independent of domain applications.

2.1.2 Other End-user Program Synthesis Approaches

In this section, the rest of the end-user program synthesis approaches are grouped by their means of script creation. The script creation ranges from providing examples of input-output pairs of the desired scripts, combining operations from a set of pre-defined operations, and writing a program with the more human-friendly language.

One of the most well-known examples of end-user program synthesis is Microsoft Excel's FlashFill [34, 36], which automatically creates generalized Microsoft Excel scripts from user-provided input-output examples. *FlashFill* produces an output for an unseen inputs according to the pattern discovered in the provided input-output examples. Similar works generate the script by learning the input-output pattern using a deep learning architecture approach [12, 30, 35]. These works are categorized as Programming by Examples (PbE) [59]. As its name implies, Programming by Examples takes as input a set of examples input-output pairs and then synthesizes the generalized script that generates user intended outputs from unseen inputs. These systems needs internal API in order to manipulate elements of the applications.

Another well-known example of the end-user program synthesis is OSX *Automator* [4]. The *Automator* allows users to create simple scripts for user's GUI tasks by constructing procedures from a pre-defined set of operations, which the *Automator* can perform, step by step. Those scripts can be saved and played later when the users want to execute the tasks. OSX Accessibility API is used for both script generation and execution phase.

Koala [61] and *CoScriptor* [55] introduced platforms to generate automation scripts and business processes with a loose programming approach. The loose programming is a programming language which is more similar to human language and has more flexible syntaxes compared to the traditional programming languages. These scripts can be shared and read by humans in the form of an organization Wiki page. Those system focused only on web-based processes; the special APIs are not required since a web-page source code is readily accessible and machine-readable via The Document Object Model (DOM). Furthermore, *CoScriptor* pro-

vides a macro recording functionality which records user demonstrating the tasks and transforming the recorded tasks into the loose programming scripts. Nevertheless, these two systems cannot be extend to general applications outside the web browsers.

Sikuli [95] was the first to provide programmers a semi-visual programming language to allow direct interaction with Graphical User Interface (GUI). the script identify GUI elements and interact with them by relying on their visual information, pixel values, instead of via the accessibility API. Hence the system can interact across applications and can be run on multiple platforms. *Sikuli* was used in many applications *e.g.*, software testing [20, 96]. However, *Sikuli* scripts use traditional programming syntax where it allows variables to be an image of a GUI element. The scripts need programmers or users who have programming skills to write them.

LIA (Learning by Instruction Agent) [10] combines natural language with the pre-defined set of operations to generate automation scripts from natural language instruction. The work focuses on email reading and sending tasks. It use Natural Language Processing techniques to parse and transform human language instructions to the automation scripts which comprise of pre-defined sensor and event procedures. When the system does not understand the instruction, it prompts the user to manually construct the script from the pre-defined sensors and events and then update itself. This approach is the first attempt to use human natural language to synthesize the scripts but it application is still limited to the emailing tasks.

It is important to note that different means of script creation have their own weakness and strength; some approaches are preferable for certain applications than others. Programming by Examples work well with spreadsheet and data wrangling tasks where there are finite sets of functions which map the given input examples to the corresponding output examples. On the other hand, GUI tasks have intractably large search space of operations which map between an input desktop setup and it corresponding output desktop setup. Observing instructors performing the tasks is then a preferable approach for tasks involving GUI operations. Construction scripts from pre-defined operations seems to be easy for end-users but it is almost impos-

sible for developers to provide enough pre-defined operations which can cover all end-user needs. Finally although providing GUI specific programming language is the most robust approach to end-user program synthesis, this approach requires the end-users to at least have a basic knowledge of computer programming which are not easily acquirable skill sets.

2.2 Other Approaches to Comparable Domains

In this section, other domains which share common goals, *e.g.*, understanding user log, are discussed. Two domain environments which are related to the GUI desktop environment listed here are Digital Game Domain and Web Search Engine Domain.

2.2.1 Digital Game Domain

Digital game environments are closely related to the GUI desktop environment. It can be considered as a simplify version of the GUI desktop environment due to the sets of possible inputs are typically smaller than the GUI desktop environment. This subsection summarizes two different branches that relate to understanding interactions between users and graphical user interface of game environments. First, projects where different gaming log-files served as the main input data of the user behavior and gameplay experience analysis are studied. Second, machine learning projects which treat computer games as experimental platforms are highlighted. Where feasible, these establish a connection between machine learning algorithms and computer game analytics research.

2.2.1.1 Understanding Game Users Through Log-files

A log-file records information as a stream of events or messages. Log files have been used extensively in game user research to extract important or hidden information of gamers' emotions or behaviors. Here, important research projects that utilize gaming log-files (with a variety of content) as initial sources of information are explained.

Shute et al. [86, 85, 87] analyze user interaction logs in their stealth assessment of digital games for education. Stealth assessment is an assessment made from the interaction data collected during gameplay sessions. During the gameplay sessions,

users/learners do not notice that they were assessed. [23] uses customized game log data to generate video summarizations of what happened during gameplay. In [91], Tveit et al. proposed to use a player's action log-files from Massive Multi-player games to mine for user behavioral patterns. Zaman and MacKenzie [97] use user input logs and game logs to evaluate different input devices for touchscreen phones. Nacke et al. [72, 73] study the use of game session logs and game event logs to assess game players' experience, and to better understand the gameplay sessions. Smith and Nayar [89] successfully model user's play style from the user's raw controller inputs using Latent Dirichlet Allocation (LDA).

While aforementioned works utilize log information as a core input component, There does not exist a practical tool that allows researchers to retrieve such information from publicly shared gameplay videos. The closest research which aims to extract game information log from gameplay video is Marczak et al. work [66]. This work extracts gameplay metrics such as health bar and in-game items by simple hand-coded image processing techniques. Studies in Chapter 5 propose a system that allows researchers to automatically extract raw control input information from existing gameplay videos. This should boost up both Human Computer Interaction and Computer Vision research communities' ability to work with very large datasets, without conducting extremely laborious and costly data collection processes.

2.2.1.2 Using Machine Learning in Computer Game Research

Several attempts to probe Artificial Intelligence (AI) agents in the context of computer games played by humans are made recently through advancements in Machine Learning (ML) research, and through newly available platforms. To allow the AI and ML research communities to validate their algorithms, a variety of games has been used. Several of these well known platforms are highlighted in this section.

OpenAI Gym [17] and ALE [15] provide environments for computer agents to play Atari 2600 games and many flash games. VizDoom [50] and DeepMind Lab [14] provide API's which allow AI and ML agents to learn to play classical first person shooter (FPS) games, e.g. Doom and Quake III respectively. Re-

cently, RLE [16] was proposed to be an environment for SNES games, and Project Malmö [46] is an environment for the game Minecraft.

These encourage ML researchers to develop new agents, e.g. through Reinforcement Learning: [32] for Tetris and [69] for Mario; Deep Reinforcement Learning agents: [51, 21, 80] for Doom, [68] for Atari games, and NeuralKart [39] for Mario Kart 64; and Supervised Learning agent: TensorKart [41] for Mario Kart 64.

DeepLogger, proposed in Chapter 5, is a tool that intends to complement the works mentioned in this section, by allowing researchers to train their Machine Learning algorithms on any human gameplay videos, in addition or in contrast to having these agents learn only by self-play.

2.2.2 Web Search Engine Domain

When users interact with web search engines, they often input more than one query to reach their goals. These sequential traces are logged by the search engines as “query sessions”. The query sessions is important for the web search engines because the sessions can reveal users’ behaviors which can be used to improve web search engine services such as keyword suggestion and advertisement. There are many works on this domain which attempt to infer users’ goals [81, 54, 44] and users’ tasks [62, 45, 63, 57] from the query session logs.

There are several characteristics, which shared between web search engine log analysis and GUI log analysis, such as users perform sequences of steps to reach their goals, there are more than one path to complete a task, and users’ goals are often inferred to users’ intentions. However, important details which make these two domains different are as follows,

1. In web search engine, the number of possible queries are unbound comparing to the number of possible GUI actions.
2. In web search engine, relationships between queries when composed together to form a task does not have complex concepts, such as looping, conditioning, and standing by, as in the GUI automation.

2.3 Background Knowledge

In this section, important concepts which are applied throughout this thesis are presented. First, the proposed visual-based PbD systems perceive user demonstration through a sequence of screen-captured images. The literature on analyzing GUI screen-captured images can be found in 2.3.1. Additionally, the system then needs to encode a sequence of low-level user demonstration signal into the higher level of abstraction, mouse-keyboard actions, in order to understand the demonstration. The encoding process is closely related to the joint action segmentation and recognition process in Computer Vision literature; brief overview of related work on segmentation and recognition sequence of input images/signals is provided in section 2.3.2. Finally, in section 2.3.3, looping action recognition and the time series motif discovery algorithms which are used for recognizing looping tasks and circumventing inconsistent human demonstrations are discussed.

2.3.1 GUI Analysis

GUI analysis is the first important step of Visual-based Programming by Demonstration. It allows the PbD systems to recognize components on the screen which will then be used to perceive user demonstration. Many attempts have been made on the topic and they are reviewed below.

Dixon et al. [26] studied models of GUI widgets, *e.g.*, buttons, tick boxes, text boxes, *etc.*, in *PreFab*. They associate parts into hierarchical models from visual information. This work and its extension [27] aim to reverse-engineer GUI widgets to augment new user interactions which can enhance existing widgets without modifying internal codes of the applications. Although widgets are identified by their pixel values, instead of relying on the accessibility APIs, the system uses pre-defined heuristics to identify different widgets. Thus the system is restricted by the set of pre-defined widgets and cannot apply the technique to non-widget elements such as images and spreadsheet cells. Likewise, the systems proposed in this thesis do not rely on a pre-defined set of widgets and can work on image and spreadsheet cells.

Waken [13] processes video tutorials to allow users to directly interact with widgets in the video tutorials. The system uses consecutive frame differences to locate the mouse cursor and the application widgets on which the video tutorial is focusing. They applied the system to video tutorials to allow user to directly interact with the widgets in the video player. However, similar to *PreFab*, this system mainly relies on pre-designed heuristic rules to detect cursors and widgets. This make the system require substantial engineering effort to add a new widget to the system when the developers want the system to work with new widgets.

EverTutor [93] processes low-level touch events and screen-captured images on a smartphone to automatically generate a tutorial from a user demonstration. Due to it perceiving user demonstration via screen-captured images and low-level touch events, it is not restricted to a particular set of applications. A tutorial generated by the system can interactively guide the user through the task. Although problem is approached in many similar ways to HILC, in Chapter 3, they use pattern matching algorithm to extract user demonstration process, their algorithm details are omitted.

Grabler et al. [31] presented a system that generates a photo manipulation tutorial from an instructor's demonstration in GIMP, GNU Image Manipulation Program. The system generates each step of the tutorial by accessing changes in the interface and the internal application states. The source code of GIMP was modified to allow such access. In addition, the authors proposed preliminary study on transferring user demonstrations into automation scripts which can be executed later to perform user intended tasks, similar to PbD system. Nevertheless, this work relies heavily on privileged access to the application source codes.

Chronicle [33] allows users to explore and modify the history of a graphical document (image editing), as it was created through sequence of interactions. The system captures the relation between time, regions of interest in the document, tools, and actions. This rich information allows users to playback the video for a specific time or region of interest, replicate the result, change parameters for specific operations, and understand the workflow. This system also heavily rely on accessing to the application source code.

The *Pause-and-Play* system [79] helps users better learn to perform a task from video tutorials. The system finds important events in the video tutorial, and links them with the target applications. It allows the system to automatically pause the video by detecting events in the applications, while the user is following the steps of the tutorial. However, the detection of events in an application is implemented through the accessibility APIs of those applications, so the system can only be used with limited applications.

Karpathy's *Mini World Of Bits* challenge [48], part of OpenAI's Universe platform, is another initiative idea that aims to be a proving-ground for bots. The task of the challenge is to exploit reward functions to perform reinforcement learning on GUI widget images. The main aim of the project is to provide ground truth for learning about GUI widgets and their interaction through their pixel values. Although this project is still in an early stage, the database of the project can be used as a knowledge base to train the ultimate system, learn to automate desktop GUI task from video demonstration, aimed by this thesis.

2.3.2 Joint Segmentation and Classification of Action

A crucial problem faced in HILC and RecurBot is to interpret user demonstration from low level signal log file. In order to transcribe the user interaction log-file into a sequence of basic actions, the log is needed to be segmented into parts. The parts are then classified into basic actions. This problem can be framed as human action segmentation and classification. The techniques applied in this thesis are modified from the following works.

Shi et al. [84, 83] jointly segment and classify human actions in long videos, unlike other action-recognition literature, which only do recognition on pre-segmented clips. They use their Viterbi-like algorithm for inference, and use Hamming distance to measure the loss between labels of two consecutive frames. HILC and RecurBot also use the same dynamic programming algorithm but the distance is modified to accommodate the characteristic of the problem which is discrete action and continuous appearance. Hoai et al. [40] also proposed similar jointly segmentation and classification algorithm with slight modification which can be applied

to segment honeybee motion. Other action-detection methods tend to be slow and ill-suited for GUI problems.

2.3.3 Discovering Looping Pattern

A study from chapter 4 shows that users are often inconsistent when they have to demonstrate the same procedure a few times. This leads to a need for a looping recognition algorithm that is robust to the user inconsistency.

Levy and Wolf [56] train a Convolutional Neural Network with synthetic videos to detect loops in videos of real-world actions. They heavily rely on a pre-processing step to enable motion detection. Unfortunately, GUI task demonstration videos have very subtle movements and changes, so their algorithm fails to detect these actions. Moreover, their algorithm is designed to detect loops which are temporally periodic, whereas for GUI tasks, the same sub-task often takes differing amounts of time, and also have user inconsistency, accidental and intentional variations. *Familiar* [77] was specially designed to detect and automate looping of GUI tasks. They segment a sequence into loops by treating the last action of the full demonstrated sequence as the universal loop-terminator. The best loop is then determined by finding the actions at the intersection of all the subsequences. Finally, the algorithm uses a set of rules to determine whether an action is a noisy representation or a looped representation. *RecurBot*, in chapter 4 does not assume that subsequences are free of noise or omissions, or that specific actions serve as loop-delimiters. Moreover, the proposed *visual-based* system does not have access to an exact comparison metric between two similar basic actions. This makes the problem more complex, but it also allows the system to be domain-independent.

Given a long time series sequence, *motifs* are sub-sequences which are very similar to each other [60, 11]. Motif discovery problem was extensively studied and explored in many applications including medicine [7], biology [8], audio [38], shape [94], and motion [71]. These works are all interested in finding a pair of motifs rather than sets of motifs, and many of the contributions in the field are to speed up existing algorithms. This thesis frames the looping action recognition as a motif discovery problem for an input sequence of actions; tries to find a set of motifs

which vary in length due to extra and missing actions. To the best of our knowledge, The work proposed in Chapter 4 is the first time that the recurrent action recognition problem has been framed as time series motif discovery.

It is noteworthy to mention here that there are works [75, 82, 78] in Natural Language Processing (NLP) which attempt to deal with noise and missing steps in narrative script domain. However, the techniques cannot directly be applied to the problem in this thesis because NLP problems require background knowledge from very large language corpus while the systems proposed in the thesis only have access to examples from the demonstration phase.

Chapter 3

Visual-based Programming by Demonstration

Creating automation scripts for tasks involving GUI interactions is hard. It is challenging because not all software applications allow access to a programs internal state, nor do they all have accessibility APIs. Although much of the internal state is exposed to the user through the GUI, it is hard to programmatically operate GUIs widgets. Accordingly, existing PbD systems suffers from these limitations.

The aim of this chapter is to an answer an important question, “How can the special API restrictions of PbD systems be lifted?”, by developing a system prototype which learns-by-demonstration, called HILC (Help, It Looks Confusing). HILC utilizes Computer vision as a delicate tool needed to replace the APIs. Users, both programmers and non-programmers, train HILC to synthesize a task script by demonstrating the task. A demonstration produces the needed screenshots and their corresponding mouse-keyboard signals. After the demonstration, the user answers follow-up questions.

This chapter examines many small and non-obvious challenges to learning-by-demonstration in a GUI world. While template-matching of icons and scripting of macros and bots are low-tech by modern vision standards, these technologies are effective as an initial solution. The overall contributions of the proposed approach are that:

- Non-programming users can teach a task to the system, simply by demon-

strating it, either on a computer with a sniffer program or through screencast-video.

- The system asks the human for help if parts of the demonstrated task were ambiguous.
- The same or other users can run the task repeatedly, giving them functionality that was previously hard to achieve or was missing entirely, *e.g.*, looping.

Tasks range from short single-click operations to inter-application chains-of-events. The informal surveys revealed that each user had different tasks they wished to automate, but they agreed universally that the teaching of a task should not take much longer than performing the task itself.

With the long-term aim of improving assistive technology, the role played by the *instructor* is separated from that of the *end-user*. For example, one person could use a mouse and keyboard (or an eye-tracker) to demonstrate a task and answer follow-on questions. Then the same or another person could run that learned task, *e.g.*, using speech. Algorithms to make the instructor effective are the main focus of this chapter. Moreover, the teacher role is split off from the instructor role to support diverse demonstration inputs, *e.g.*, video tutorials *vs.* sniffer programs.

Through the designated scenarios, three interesting issues of general Programming by Demonstration systems are explored. For the linear tasks, script generation is explored; for the looping tasks, generalization of the generated scripts is explored; and for the monitoring tasks, invocation of scripts is explored.

3.1 Overview of Challenges

At a high level, HILC collects observations while the instructor performs a task, then it finds confusing looking patterns that call for the instructor to give more input. Once the task is learned and saved as a script, it can be called up by the end-user to run one or more times, or to monitor for some visual trigger before running.

A number of challenges make this an interesting technical problem that relates to object recognition, action recognition, and one-shot learning. The instructor's

computer can be instrumented with a sniffer program, that records mouse/keyboard events, and screen-appearance. However, for example, a click-drag and a slow click are still hard to distinguish, and hugely varying time-scales make sniffed observations surprisingly hard to classify. Learning of tasks from pre-recorded screencast videos, which display noisy details of key/mouse events, is also explored. Further, clicking someplace like a File-menu usually means the task calls for that *relative* location to be a target, but what if other parts of the screen have similar looking menus? The teacher can help find visual cues that serve as supporters.

Moreover, while demonstrating, the instructor runs through a linear version of the task, and can indicate if some part of the chain of actions should actually become a loop. Template matching may reveal that the to-be-looped segment was applied to one unique target (an icon, a line of text, etc.,), but HILC can request further input to correctly detect the other valid targets on the screen. This functionality is especially useful when a looping task must “step” each time, operating on subsequent rows or columns, instead of repeatedly visiting the same part of the screen.

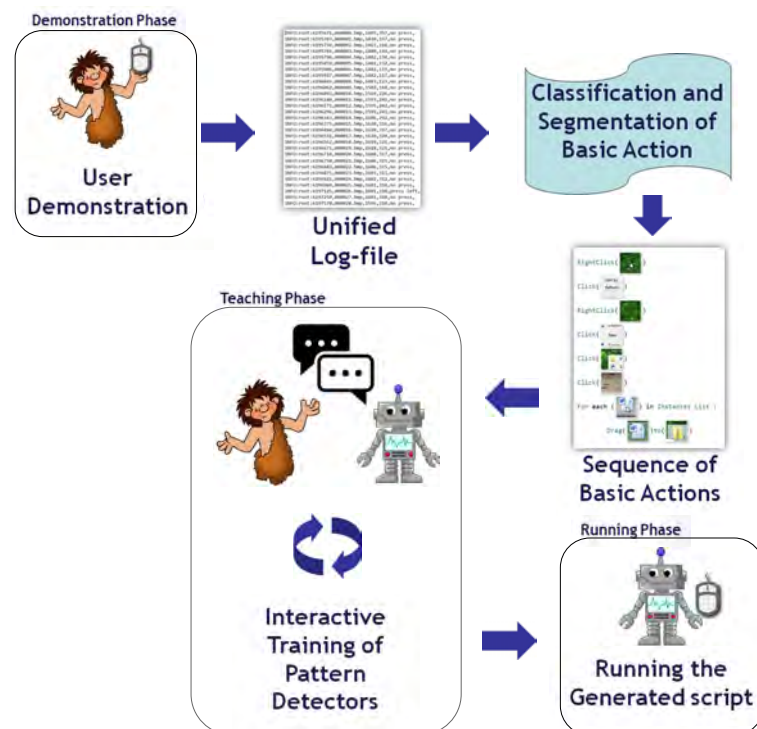


Figure 3.1: Flow of the system. Details for each phase are described in the text.

3.2 Learning to Perform Tasks from Demonstration

The proposed system, HILC¹, has three phases: demonstration, teaching, and running. First, an instructor can choose to record their demonstration as either a screen-cast video (so highlighting mouse and key presses) or through custom-made sniffer software. Both methods have pros and cons, which will be discussed further in the Demonstration Phase section. Next, during the teaching phase, HILC performs joint segmentation and classification of basic actions. To generalize the observed actions, a human teacher (can be the same person as the instructor) who can help the system refine its pattern detectors is introduced. The challenges and proposed solutions of joint segmentation and classification, and for humans in the loop training, are in the Teaching Phase section. Finally in the Running phase, HILC performs actions according to the transcript generated by the instructor and improved by the teacher. The system also generates a Sikuli-like script, see *Sequence of Basic Actions* in Figure 3.1, for visualization purposes. A stand-alone runtime script in pure python is made using the PyAutoGUI package. Overview of HILC is displayed in Figure 3.1.

To start, a set of basic actions that the system can perform is defined, and the system needs to jointly segment and recognize these actions from the input demonstration. The set of basic actions is composed of Left Click, Right Click, Double Click and Click Drag.

3.2.1 Demonstration Phase

In this phase, an instructor who knows how to complete the task demonstrates the task while only recording a video or while running the sniffer program. If the user chooses to record the task as a video, HILC will pre-process the video and create the unified format log-file from every frame of the video. On the other hand, if the user chooses to record the task via a sniffer, the system will record every signal produced by the user to the unified format log-file.

¹Project page: <http://visual.cs.ucl.ac.uk/pubs/HILC/>

3.2.1.1 Recording the Task

For both input methods (video-only or sniffer) a unified sensor-data format is introduced. Functioning like a log-file (see Figure 3.2), each entry records the screenshot image and the low-level status of the machine: its mouse button status, mouse cursor location, and the keyboard's key press status. A sniffer can access useful information directly from the OS, but cannot determine the class of the basic action. It does naturally generate a log-file from the demonstration. However, one must take into account time intervals between each log entry, inconsistent machine lag, and storage space of screenshots. Details are presented in the Implementation section.

To help HILC recognize whether a demonstrated segment is a linear, looping, or monitoring/standby task, special key combinations that the user was briefed on before using the system are used. The terms monitoring and standby are used in this thesis interchangeably and they both refer to the same kind of task. The details of the recording step and the special key combination are presented in the User Study Scenarios section.

Although processing a video-only demonstration is slower, it is more versatile because end-users can leverage pre-recorded and internet-shared videos as an input demonstration. However, processing of demonstration videos has many challenges, such as locating the mouse cursor, retrieving low-level mouse/keyboard events from screencast messages, removing the mouse pointer from the video's screenshot images (*i.e.*, capturing a template of the button without the cursor's occlusion), and noise/compression in the video recording.

The implementation details of the approach for coping with the aforementioned issues for both sniffer and video inputs are described in the next section.

```
INFO:root:3483502,000087.bmp,1265,285,no press,  
INFO:root:3483502,000088.bmp,1263,286,no press,  
INFO:root:3483642,000089.bmp,1262,287,no press,  
INFO:root:3483829,000090.bmp,1262,287,press left,  
INFO:root:3483876,000091.bmp,1266,287,press left,  
INFO:root:3483892,000092.bmp,1274,287,press left,  
INFO:root:3483892,000093.bmp,1288,284,press left,
```

Figure 3.2: Example of a log-file which merges the inputs from both video and sniffer data.

3.2.1.2 Implementation

HILC was implemented in Python 2.7 and it was deployed on Microsoft Windows 7 64-bit machines with Intel Core i5-3317u @1.70GHz CPU and 8GB of RAM.

Implementing a custom sniffer which records low-level mouse and keyboard events is carried out. Log-file entries and screenshots were only saved before and after each of the mouse and keyboard status changes, to keep the hard drive i/o from slowing down the machine. Records are re-sampled to have log-files where the time difference between records is 1000/30 milliseconds, to make sniffer log-files versions agree with video versions that sample screenshots at 30 frames per second.

Using an existing video as a demonstration requires the addressing of two important issues: how can the system retrieve low-level mouse and keyboard status information, and how can the system remove the mouse cursor from video frames for clean pattern extraction?

Video tutorials are commonly recorded with specialized screencasting software that renders visual indicators of mouse events, left-right button pressing, and keystrokes. Hence, it is assumed that the demonstration video was recorded with key-casting software. Each key-caster is different, but HILC was trained for Key-CastOW for Windows and Key-mon for Unix. Others could easily be added. Thus, key-cast display locations are extracted from the video and the information for each frame is recognized via the OCR software Tessarract. The mouse status information displayed by the key-cast software is still low-level, similar to sniffer output. The mouse cursor location is detected by a Normalized Cross Correlation detector of the mouse pointer template. Videos of demonstrations have an inherent problem of the mouse cursor occluding a target pattern during a basic action. To overcome this, significant appearance changes of the screen are detected and a temporal median filter is used to remove the cursor to create a clean mouse-free screenshot.

In the Demonstration Phase, the instructor's basic actions are recorded by HILC. Hence, to interact with HILC, for example, to start or stop the recording, the instructor triggers explicit signals. In the implementation, these are special key combinations. The detection of loops or standby patterns is not automatic, and has

to be flagged by the instructor. Therefore, three special key combinations are defined for the different signals: End of Recording, Looping, and Standby, which are discussed in detail before each designated scenario is explained in the Evaluation and Results section.

3.2.2 Teaching Phase

At this phase, HILC takes as input the log-file from the Demonstration Phase, and then produces a transcribed script that consists of a sequence of basic actions. HILC workflow of the teaching phase is in Figure 3.3.

3.2.2.1 Classification and Segmentation of Basic Actions

Although the log-file contains all of the low-level information, identifying basic actions is not straightforward. For example, the “left mouse button pressed” status can be presented in the log-file at multiple consecutive entries for a single “Left Click” basic action. Further, it is ambiguous if a left click is a single “Left Click” basic action, or a part of the “Double Click” basic action. Moreover, variability based on how individual users interact with the computer via peripheral devices makes it hard for deterministic rules to distinguish between different basic actions. This drives the need for training data, though each user provides only very little.

To create a system that can cope with ambiguities in recognizing basic actions, the problem is treated as a Viterbi path decoding problem. With dynamic programming, the proposed algorithm segments and classifies the basic actions concurrently.

Let $Y = \{y_1, \dots, y_z\}$ be the set of all possible states, a label for each temporal segment. x is the observation of a segment, a feature vector representing part of a basic action. The unary terms $U(y|x)$ are probability distribution functions over parts of basic actions, learned from pre-collected training data, The pairwise terms $P(\tilde{y}, y)$ are the constraints that force consecutive parts of actions to be assigned to the same basic action.

A_k is defined as a basic action made from a sequence of parts of the action k , $A_k = \{y_1^k, y_2^k, \dots, y_l^k\}$; $|A_k| = l$ and y_l^k is the last part of A_k . y_n^k is a part of the basic action k where n indicates a status change frame (key frame) in the log-file, e.g., a

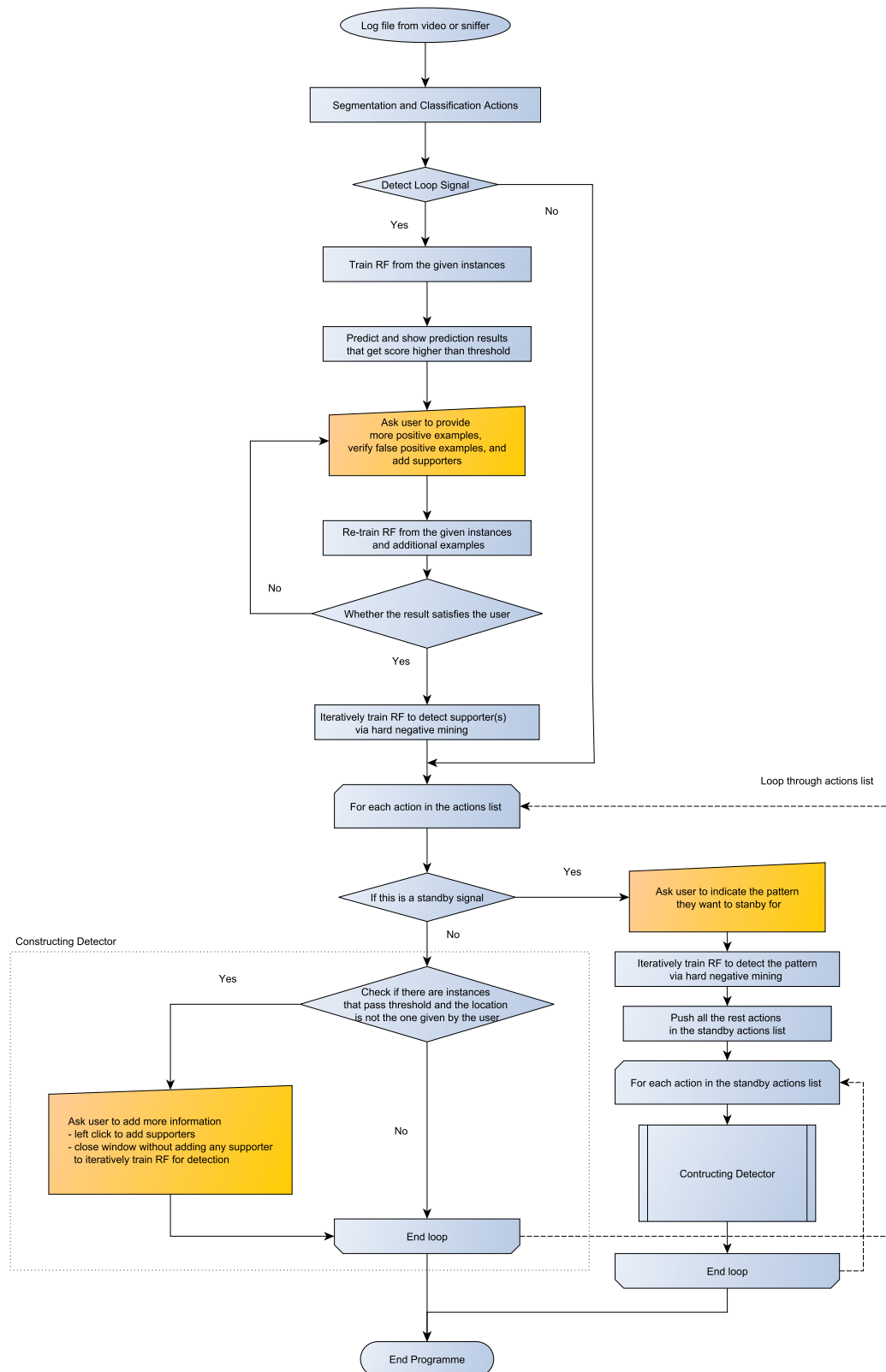


Figure 3.3: The system workflow for the teaching phase. The yellow boxes indicate where the system poses questions to the teacher.

frame where the mouse button status is changed, from press to release or vice versa. Pairwise term is defined as in Eq 3.1:

$$P(y_{v-1}^i, y_v^j) = \begin{cases} +1 & \text{if } i = j \text{ and } y_{v-1}^i \text{ follows } y_v^j \\ 0 & \text{if } i \neq j \text{ and } y_v^i \text{ is the last} \\ & \text{part of } A_i \\ -1 & \text{otherwise.} \end{cases} \quad (3.1)$$

Learning Probability Distribution Functions

One of the challenges of basic action classification is that each basic action has a different duration. For example, a “Left Click” usually spans a few milliseconds in the log-file, but “Click Drag” may span seconds. Moreover, each basic action also has its inter-variability in terms of action duration.

The proposed approach is to train a Hinge-loss Linear SVM for each basic action using the hard negative mining method, where each SVM may make a prediction on a different time interval. For an input time interval, a feature vector is constructed by computing histograms of frequencies of different encoded records from the log-file. Each record is encoded by 3 binary low-level states: the status of each of the mouse buttons, pressing the left button/ the right button, and whether the mouse is moving. The feature vector also encodes context information of each action by adding a histogram of a small time interval after an action is done. Finally, the feature vector for the Linear SVM has 4×2^3 dimensions, the number 4 is from 3 histograms of equally divided time intervals of an action and plus 1 context histogram. It is noteworthy that this is different from a part of an action y_n^k . Figure 3.4 depict the input part to the SVM.

The prediction scores of SVMs are not proportional to one another, so they must be scaled before use within the Viterbi algorithm. Hence, for each basic action, a Random Forest (RF), which takes as input a vector of the prediction scores from SVMs and outputs the probability of each basic action class, is trained.

For any unknown time interval, a unary matrix which maps between a part

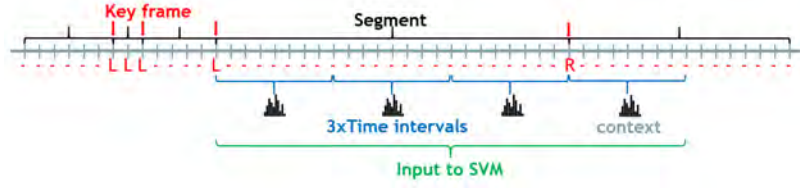


Figure 3.4: Examples of a segment, a time interval, a key frame, and an input to SVMs. ‘L’ indicates the left mouse button is pressed, ‘R’ indicates the right mouse button is pressed, and ‘-’ indicates none of the mouse buttons are pressed.

of action to its probability distribution can be constructed from trained RF for that action. The unary matrix $U_{s \times c}$ has the shape (number of states \times number key frames in the input sequences). The number of states is defined as Eq 3.2

$$\text{number of states} = \sum_{A_k \in B} |A_k|, \quad (3.2)$$

It should be noted that the same probability distribution is assigned to all parts of each basic action.

Annotation of Log-files for Basic Actions

To annotate an unknown log-file, the system detects status change frames in the log-file and use them as key frames. Records in the log-file are then grouped into N different time intervals indexed by the key frames and passed to the trained Random Forests to construct the unary matrix U . Lastly, the Viterbi dynamic programming algorithm is used to infer about \mathbb{Y}^* , which is the sequence of y_n^k that maximizes Eq 3.3, where

$$\mathbb{Y}^* = \arg \max_{\mathbb{Y}^*} \sum_{v=1}^N (P(y_{v-1}^i, y_v^j) + U(y_v^j | x_v)) \quad (3.3)$$

3.2.2.2 Interactive Training of Pattern Detectors with Few Examples

To perform any basic action, HILC needs to learn the appearance of the target of the basic action. For linear tasks, the system needs to find a single correct location of the target pattern. However, for looping tasks, the system has to find multiple

correct locations of the targets of the looping task, and thus the system needs to generalize about the target pattern appearance. In both cases, the target pattern may have appearance variations. For example the icon of the file may have moved on the desktop and has a different section of the wallpaper as its background.

Target patterns can also be categorized into two groups: patterns that are discriminative on their own and patterns that need extra information to be distinguishable, see Figure 3.5. Hence, the system needs to treat each of these possibilities differently and a concept of supporters is also introduced here.

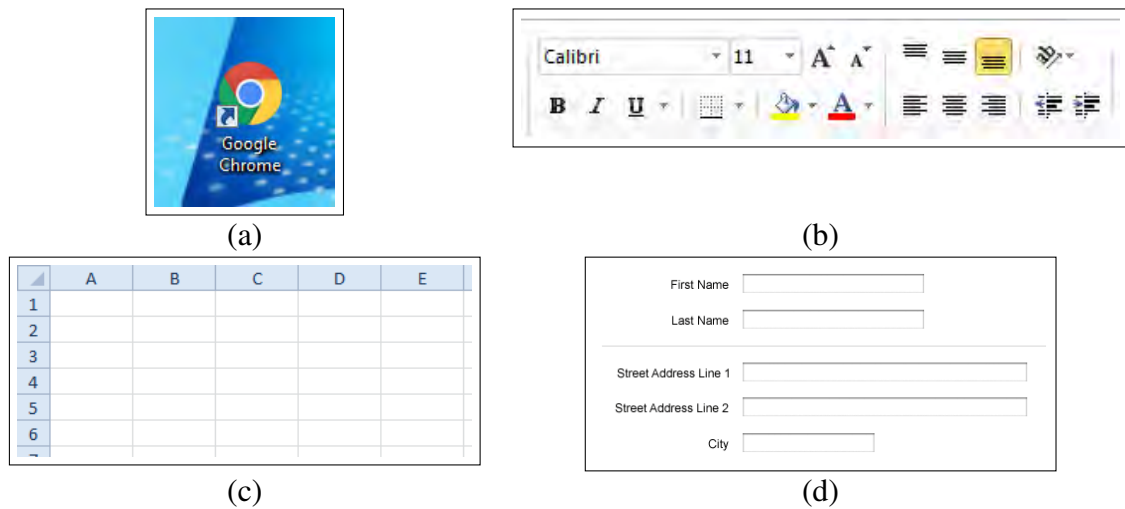


Figure 3.5: Target patterns in (a) & (b) are distinguishable on their own. The spreadsheet cells in (c) need row and column names to differentiate between one another. Text fields in registration forms in (d) can be distinguished by the text field labels. Supporter helps distinguish locally ambiguous patterns.

Supporters The supporters are salience patterns that have certain offsets to the target pattern. In linear tasks, a spreadsheet program's row and column names distinguish similar-looking cells, Figure 3.5(c); or field names distinguish similar looking textboxes, Figure 3.5(d). In the Running Phase, HILC uses the same technique that is used for detecting the target pattern, to detect supporters. The target patterns give votes to each detection location, but supporters give votes to the offset locations.

Supporters for looping tasks work differently from the supporters for linear tasks, as a fixed offset is not informative for multiple looping targets. Hence, the supporters for looping provide x-axis and y-axis offsets to the targets, see Figure 3.6.

The final result is the average of target pattern detectors and spatial supporters, see Figure 3.7.

Actor/Actress	Character
Peter Dinklage	Tyrion Lannister
Nikolaj Coster-Waldau	Jaime Lannister
Lena Headey ¹	Cersei Lannister
Emilia Clarke	Daenerys Targaryen
Kit Harington	Jon Snow
Iain Glen	Jorah Mormont
Aidan Gillen	Petyr Baelish
Sophie Turner	Sansa Stark
Maisie Williams	Arya Stark

Figure 3.6: An example of a supporter for a looping task. The table shows names of characters and actors/actresses of a popular TV show. The names in each column have similar appearance, so, if the user intended to loop through one of the columns, marking the column name as a supporter will help the system to distinguish between columns.

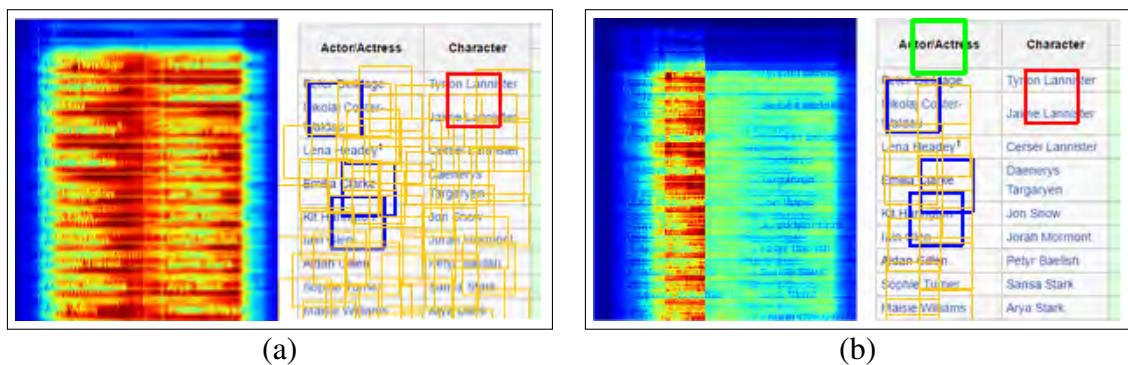


Figure 3.7: An example of a spatial supporter. Blue boxes are user provided positive examples, red are user provided negative examples, yellow are target detections, and a green box indicates a user-provided supporter. (a) and (b) demonstrate detection performance without and with a spatial supporter. Red color in the heatmaps means a high detection score. In (a), the left image shows the heatmap of target detection scores, and the right image shows detected targets. In (b), the left image shows the heatmap of target detection when combined with the spatial supporter scores, and the right image shows detected targets. The spatial supporter successfully suppressed all similar looking patterns under the Character column in heatmap (b) so that the system is able to detect only desired targets under the Actor/Actress column.

Follow-up Questions:

For different kinds of tasks, the system asks for help from the teacher differently.

Linear tasks

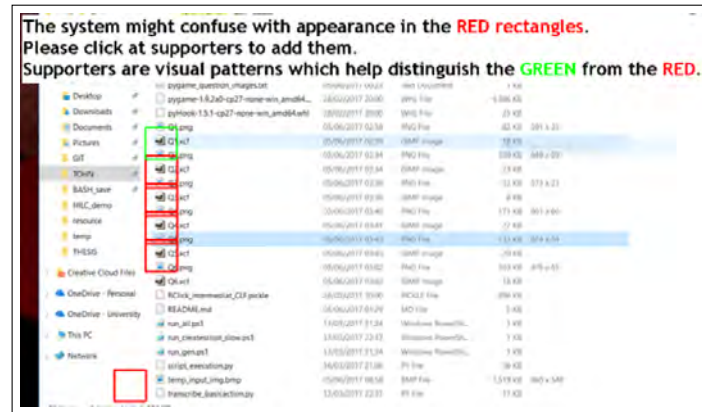
Every basic action of linear tasks has a unique target. The linear tasks can be executed multiple times, but each run performs the same task on the same unique targets. For each basic action of the linear task, the system has to learn the corresponding target pattern from only one positive example. To train the detector for the target pattern, the system initially performs Normalized Cross Correlation (NCC) matching with the given positive example on the screenshot image when the basic action was about to be executed by the instructor, to prevent the pattern from changing appearance after the action is executed. If there are false-positive locations with high NCC score, the system asks the teacher to help the system to distinguish between true and false positive examples by providing a supporter(s). Figure 3.8 shows screenshots when HILC queries for help from users.

After teachers provide supporters, the system uses NCC as the detector for both the target pattern and the supporters. If the teacher does not provide any supporters, the system assumes that this pattern is distinguishable on its own. More false positive patches are mined to retrain RF until the system correctly detects the target location using raw RGB pixel values of every position in the patch as features.

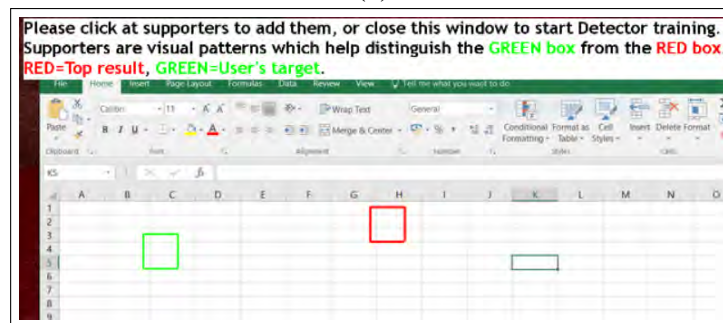
Looping tasks

Looping tasks are a generalization of linear tasks, so that in the running phase, each run of the task iterates over a set of targets. For example, a linear task always prints a unique PDF file in a folder, but a looping task prints all PDF files in a folder by looping over each PDF file icon.

For looping tasks, the instructor shows the task once on a single looping target, but the system needs to repeatedly perform the task on all looping targets that are similar to the pattern the instructor, or the teacher, or the end-user had specified. In the Demonstration Phase, the instructor is asked to show more than one example of a looping target after demonstrating one complete iteration of a task. In the Teaching Phase, the system trains an RF with the provided positive examples, and uses



(a)



(b)

Figure 3.8: Example screenshots of HILC when it asks users for supporters. (a) HILC asks users to add supporters when there are confusing patterns (red boxes) which look similar to the user intended patterns (green box). In this case, NCC scores of the confusing patterns are higher than a threshold. The system uses only the NCC detector for the intended pattern at the Running Phase, unless the users provide supporter(s). (b) HILC asks users to add supporters when a confusing pattern (red box) has the same NCC score to the intended pattern (green box). Unless the users provide at least one supporter, HILC trains RF detector for the intended pattern.

random patches as negative examples. Next, the system validates the RF predictions by asking the teacher to verify the predicted positive and negative examples, and/or add supporters. Figure 3.9 illustrates an example screenshot when HILC queries users in a looping task.

Monitoring tasks

In monitoring tasks, the system in the Running Phase perpetually runs in standby, looking for a specified visual pattern, to invoke the rest of the script. In the Demonstration Phase, the instructor indicates when the invocation pattern appears, then demonstrates the task itself. In the Teaching Phase, the system asks the

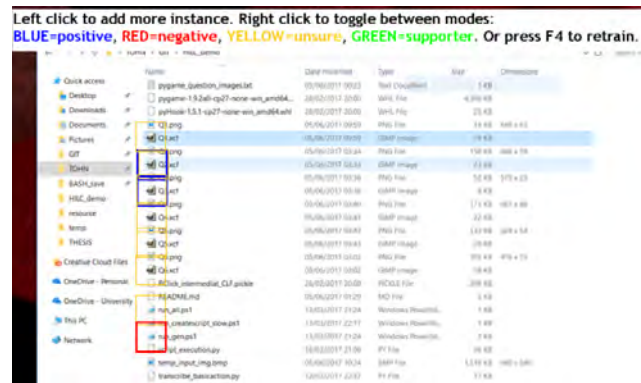


Figure 3.9: An example screenshot of HILC when it asks users for additional information during the Teaching Phase of looping tasks.

teacher to indicate which pattern needs to be detected.

Figure 3.10 displays an example screenshot while HILC is in the Teaching Phase, and asks for the visual cue. Examples of visual cues are illustrated in Figure 3.15 and 3.16.



Figure 3.10: An example screenshot of HILC when it requests a user to provide the visual cue (yellow box) which is the visual pattern that invokes the system to run the rest of the script whenever the system finds it.

3.2.3 Running Phase

The main reason the running phase is separated from the demonstration phase and the teaching phase is that the ultimate goal of HILC is to help end-users who are non-regular computer users, and disabled users, to complete tasks that might be hard for them but easy for others. The separated system is easy to execute by voice command or any other kind of triggering methods.

In the running phase, HILC sequentially executes each action of the interpreted sequence of actions from the teaching phase. When a special signal like Looping or Standby is found, the system executes the specific module for each signal. The Running phase's system flow is shown in Figure 3.11. For each normal basic action, the system starts by taking a screenshot of the current desktop and then looks for the target pattern and supporters (if available) using the trained detector(s). For the looping part, after taking a screenshot of the current desktop, the system evaluates every position on the screen with the trained RF detector, and applies the spatial voting from the supporter(s). After that, the non-maxima suppression and thresholding are applied respectively to the result, to get the list of positions to loop over. For the standby task, the system continuously takes a screenshot of a current desktop and evaluates the specified positions whether the target pattern may appear. When the target pattern is found, the system triggers the sequence of actions that the instructor designed, and then proceeds to the standby loop again.

3.3 Evaluation and Results

The algorithm is evaluated quantitatively through a small user study, and qualitatively to probe the system's functionality. The only viable baseline PbD system available for comparison is Sikuli Slides[5], because it too assumes users are non-programmers, and it too has sniffer-like access to user events. Nine use cases are reported in this chapter. To construct the set of use cases, a large list of scenarios were populated through a survey which inquires about tasks for which the users would like a virtual personal assistant to complete. The scenarios from the list were then prioritized. The scenarios that spanned the different basic actions, different lengths, different applications, and different programming paradigms were picked. Here the three scenarios tested in the seven-person user-study are listed, and the results are showed in Table 3.1. Just these three linear tasks were picked because Sikuli Slides can not handle looping or standby tasks. Further scenarios are discussed in the qualitative evaluation.

The task in each scenario was assessed in terms of 1) transcription accuracy

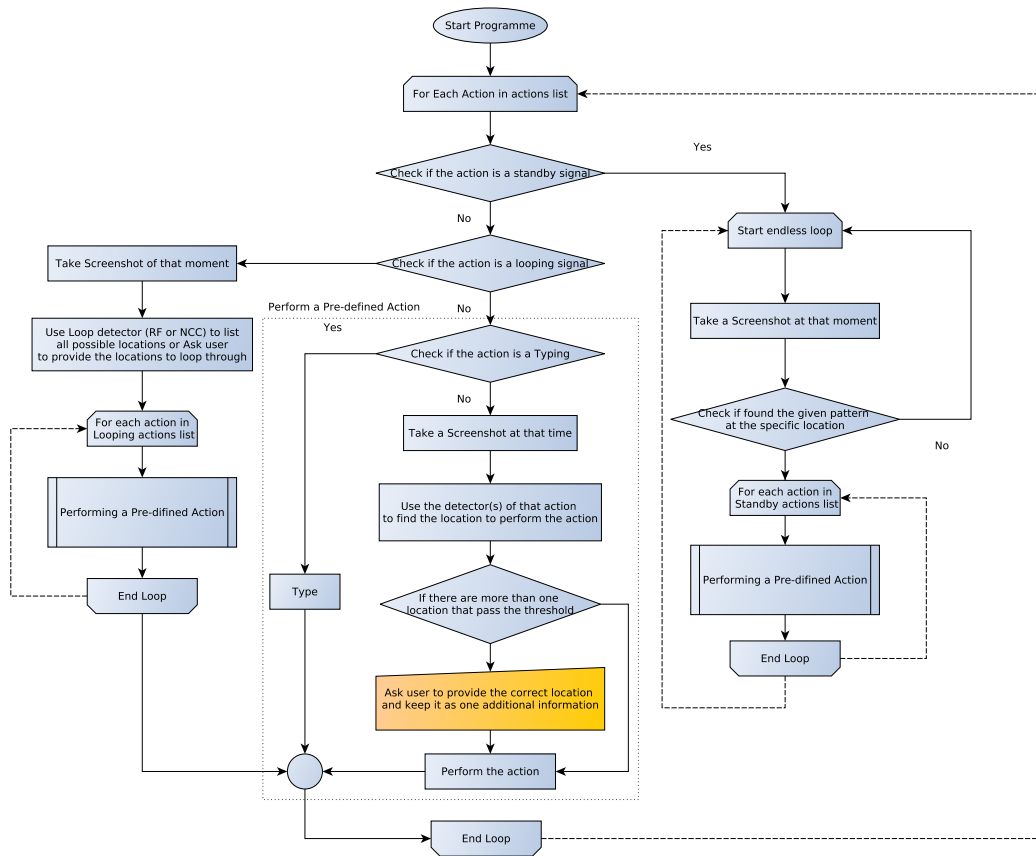


Figure 3.11: The system workflow for the running phase. The yellow box indicates user interaction.

Scenario	Basic Actions + Typing					Transcription		Reproduction		Demonstration Time VS Refining Time (average)	
	Click	Click Drag	Double Click	Right Click	Typing	Sikuli Slides	HILC	Sikuli Slides	HILC	Sikuli Slides	HILC
1 Mute Audio playback (Linear)	2	0	0	0	0	✓	✓	✓*	✓	10s/49s	10s/27s
2 Turn on High-Contrast-Mode (Linear)	6	1	0	0	0	✓*	✓	✓**	✓	27s/10m	27s/170s
3 Remote access with Team Viewer (Linear)	11	0	0	0	4	✓*	✓	✗	✓	40s/∞	40s/4m
3.2 Remote access with Team Viewer (Linear)	13	0	0	4	0	✓*	✓	✗	✓	37s/∞	37s/7m
4. Skip YouTube ads (Monitoring)	1	0	0	0	0	✗	✓	✗	✓	N/A	10s/5.5m
5. Close YouTube ads (Monitoring)	1	0	0	0	0	✗	✓	✗	✓	N/A	12s/6.9m
6. Create slides out of jpgs folder (Looping)	2x	1x	0	0	0	✗	✓	✗	✓	N/A	35s/10m
7. Create spreadsheet of filenames (Looping)	4x	2x	0	0	4x	✗	✓	✗	✓	N/A	60s/6.6m
8. Create BibTex from spreadsheet (Looping)	9x	0	0	0	8x	✗	✓	✗	✓	N/A	86s/12.5m
9. Move MSWord files to a folder (Looping-Video)	4	1x	0	2	0	✗	✓	✗	✓	N/A	25s/ 22m

Table 3.1: User study on HILC compared to Sikuli Slides. Scenario 3.2 is an alternative way to perform Scenario 3, without pressing shortcut key combinations that Sikuli Slides is known to be missing. Nevertheless, we eventually realized that Sikuli Slides is not detecting the right click actions either. (✓= successful, ✓* = partially successful, ✓** = can be successful with guidance from the operator, ✗= can not succeed at the task at all). x represents the number of repeated loops needed to complete the task. Please note that 90% of the refining time for Task 9 is offline - devoted to the time spent on processing video to produce the log-file.

(evaluating the classification and segmentation algorithm), 2) task reproduction, *i.e.*, measuring pattern detection generalization, and 3) time users took to demonstrate and then refine the task model. In the user studies, four of the participants had no programming exposure, two had taken a school-level course, and one was a trained programmer. Participants needed 1.5 - 2 hours because each completed all three tasks under both systems: they randomly started with either HILC or Sikuli Slides, and then repeated the same task with the other system before proceeding to the next task. Before using both systems the users were briefed about the goal of the study as well as how to use both systems for 20 minutes. In addition, the users were shown the videos of the instruction phase for each task before performing the task to ensure the users understand what are the tasks.

3.3.1 User Study Scenarios

The first three basic scenarios (linear tasks) are evaluated quantitatively against Sikuli Slides. Monitoring and looping tasks are evaluated qualitatively.

3.3.1.1 Linear Task

Linear tasks are simply linear sequences of actions. They are the most basic type of task, and run only once. To record and edit all kinds of tasks, two common steps need to be done: First, at the end of a task-demonstration, the user presses the special key combination `Shift+Esc`, to indicate the end of the sequence. Second, also in the teaching phase, if HILC cannot clearly distinguish between an input pattern and the other on-screen content, the system asks the teacher to click on supporter(s) near that pattern.

1. Mute audio playback (Linear) This simple and short task was actually non-trivial because the speaker icon in some Windows installations is not unique (Figure 3.12). Half the users had to refine the model, which for the system meant adding a supporter. All users produced a working model of this task using HILC, and some users were able to produce a working model of the task using Sikuli Slides (the right speaker icon was correctly selected by chance).

2. Turn on High-Contrast-Mode (Linear) Some visually impaired *end-users* may



Figure 3.12: The instructor clicked on the speaker in the green box, but the system also detected a similar pattern - the speaker in the red box. In this situation, the system asks the teacher for a supporter(s), the yellow box, to help with detecting the intended pattern.

want to trigger this task through a speech recognition system. Here, HILC's sighted *instructors* were ultimately successful using both systems, but the study-supervisor had to walk users of Sikuli Slides through the extra steps of re-demonstrating the task and making and editing of screen-shots to refine that model. This task involves the Click Drag action, which Sikuli Slides was never able to recognize when transcribing. Figure 3.13 illustrates the High-Contrast vs default desktop modes, and HILC synthesized script for switching. It is noteworthy that High-Contrast-Mode also modifies the scale of objects on the screen.

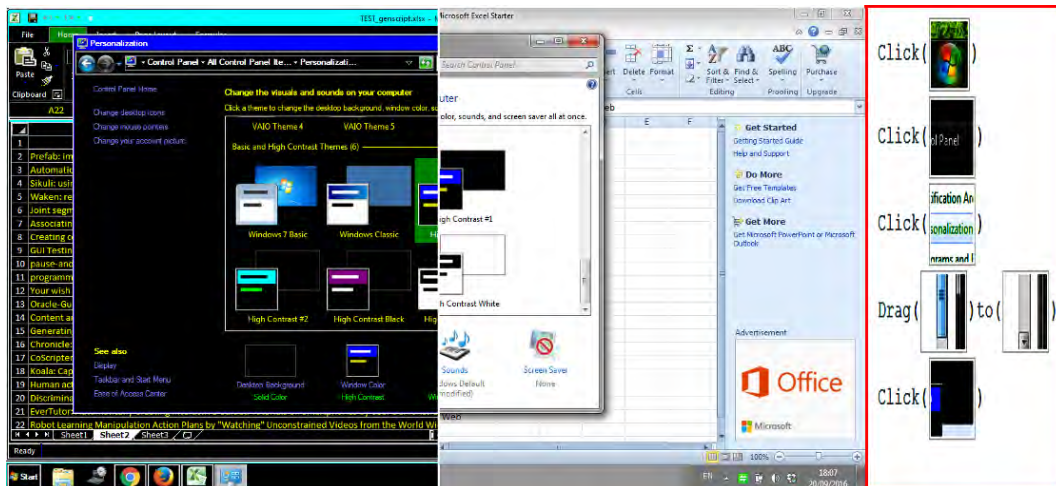


Figure 3.13: High-Contrast-Mode comparing with Normal Mode and (in the red box) the transcribed steps of the task demonstrated by HILC.

3. Remote access with TeamViewer (Linear) This sysadmin (or mobile-phone testing) task consists of running the TeamViewer application and logging into an-

other device, using an ID and password provided in a spreadsheet file. The teaching phase of the task involves helping the system clarify ambiguous patterns by adding supporters. The task and the transcribed steps are illustrated in Figure. 3.14. Inadvertently, this task proved impossible for Sikuli Slides users because it involves copy-pasting text, which that system is not able to capture the key combination shortcut. One user invented an alternate version of this scenario (3.2) where she tried to right-click and use a context menu to copy and paste, but it was then realized that right-clicks are also not captured by Sikuli Slides.

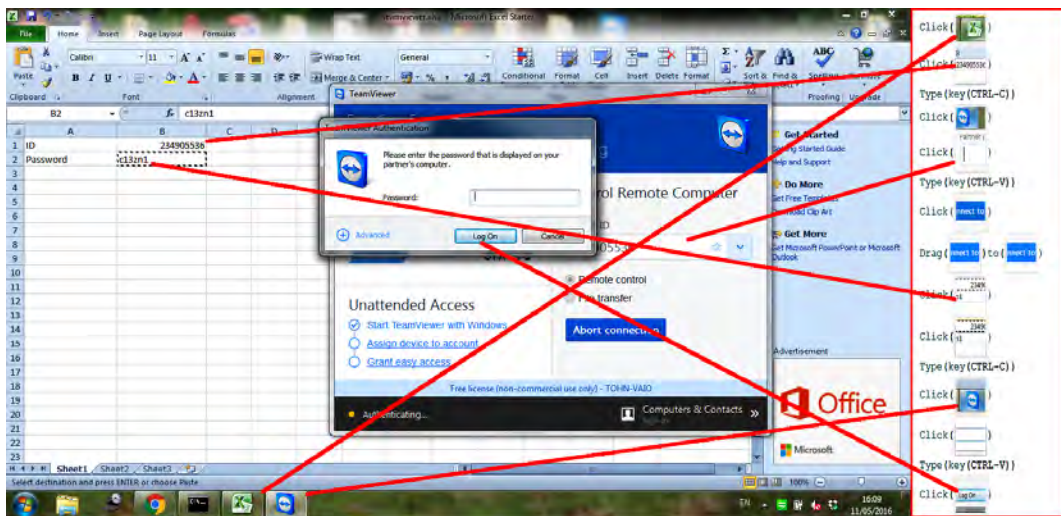


Figure 3.14: Steps to complete remote access via TeamViewer. Red lines link related patterns on the screen with the pattern in the transcript. It is noteworthy that performing the basic action DragTo from and to the same pattern has a similar effect as performing the basic action Click on that pattern. Participants often unintentionally perform the basic action Dragto instead of the basic action Click. HILC is robust to this type of different-but-interchangeable action.

Qualitative Evaluation

The remaining scenarios can not be addressed using Sikuli Slides because they entail monitoring or looping tasks. HILC's new capabilities are outlined here, along with qualitative findings, and task illustrations, to better gauge success.

3.3.1.2 Monitoring

Monitoring tasks run perpetually and then respond to specific patterns. When the specified pattern is detected, the script triggers the sequence of predefined actions.

In the Demonstration Phase, instructors press the special key combination (standby signal), `Ctrl+Shift+w` or `Ctrl+Shift+PrtScr`, to indicate that the pattern, which the system is programmed to detect, has appeared. The instructor then performs a desired sequence of actions, such as a linear task. There is an extra step in the Teaching Phase, where HILC asks the teacher to indicate where the invocation pattern *can* occur (e.g., anywhere, or in the taskbar).

4. Skip YouTube ads (Monitoring) is a standby task that clicks the text *Skip Ad* if/when it appears during a YouTube video. This task illustrates the need for spontaneous responses, because the *Skip Ad* advertisement banners appear randomly, for varying periods of time, ranging from 10 seconds to a few minutes, during playback of the requested content. Figure 3.15 demonstrates an example of the *Skip ad* task.

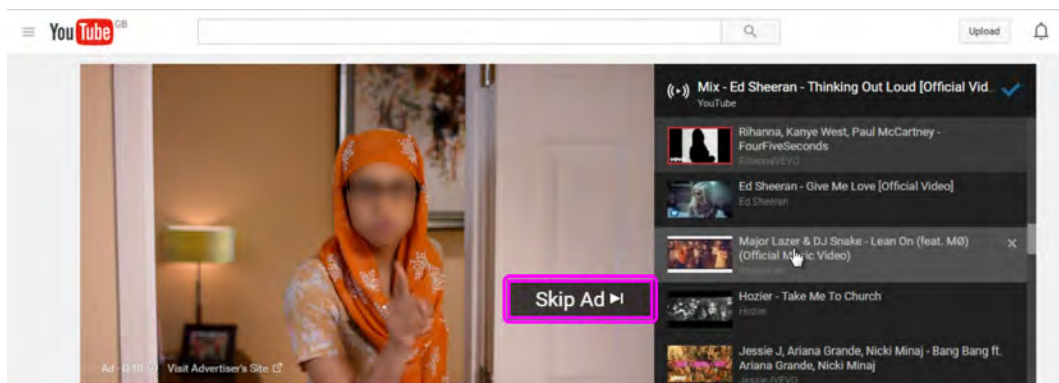


Figure 3.15: YouTube Skip Ad. These advertisements show before or during a playing video for varying periods of times, and HILC successfully closes them in Scenario 4, as soon as the text appears. The visual cue is highlighted by the magenta rectangle.

5. Close YouTube ads (Monitoring) creates a standby script to close advertisements that may appear, despite various changing backgrounds, as shown in Figure 3.16. The first line of the script directs the system to monitor an area where the given pattern can appear. When the system detects the pattern, the system triggers a script, in the second line, to click on that pattern.

3.3.1.3 Looping

HILC allows a loop to be a step in a linear action, or to be a stand-alone script. Looping tasks are the tasks that execute the same sequences of linear actions multi-



Figure 3.16: Close-ups of YouTube Ads. These ads appear at the bottom of a playing video, and HILC detects and successfully closes them in Scenario 5. The visual cues are pointed by magenta arrows.

ple times on similar looking yet different objects. To indicate the start and stop of a loop, the demonstrator inputs the looping signal key combination `Ctrl+Shift+l` or `Ctrl+Shift+Break`, before and after performing one sequence of actions that need to be repeated. Thereafter, the instructor gives examples of patterns that needed to be a starting point of the loop by pressing a `Ctrl` key while clicking on an example pattern. When the instructor is happy with the examples, they then input the looping signal, `Ctrl+Shift+l`, once more. The script can be followed by linear actions or can finish right after the third looping signal.

The teaching phase of a looping task has one additional step. HILC displays the result of the trained Random Forest, and lets the teacher add positive examples, remove false positive results and provide supporters. This triggers the re-training process.

6. Create slides out of jpgs folder (Looping) The task is to create a presentation where each slide features one image from a given folder. To create the script, a demonstrator only shows how to create one slide from one jpg, and gives a few examples of what the jpg file-icon looks like. In the running phase, HILC steps through all the jpg files in a given folder, making each one into a separate slide of the LibreOffice Impress presentation. Not only does this show that the system can loop, the task also demonstrates that the system can help the user complete repeated steps across different applications (LibreOffice Impress and Windows Explorer). An example screen of the task and the generated script are shown in Figure 3.17.

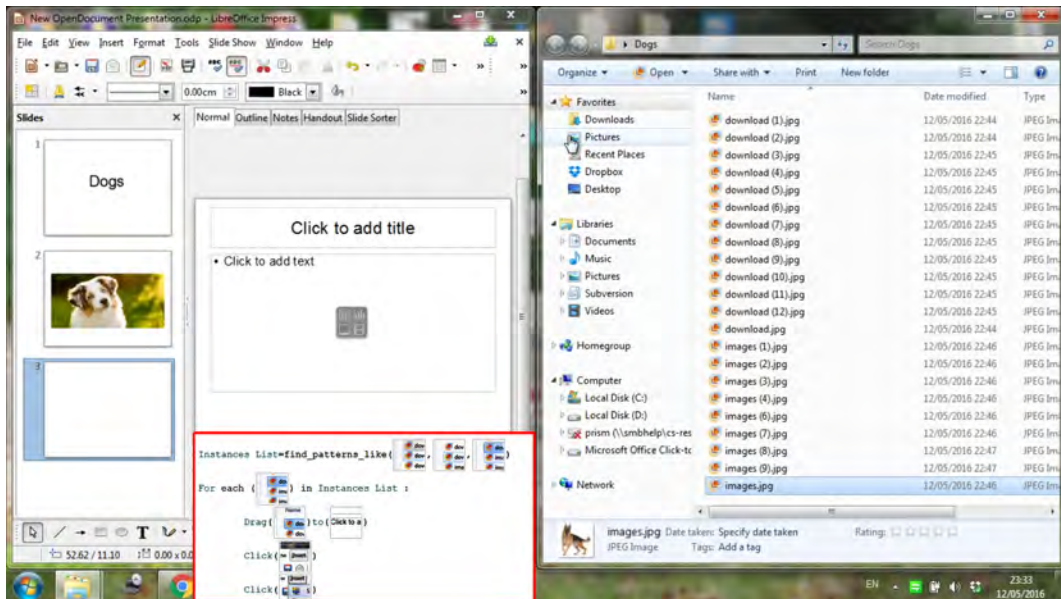


Figure 3.17: Scenario 6: create slides from folder full of images. The generated script is shown in the red frame. HILC starts by building a list of locations that will be the starting points for each iteration. The list is formed by the Trained RF, which trained and refined in the teaching phase with a few examples stemming from the demonstration phase. The system then iteratively executes a sequence of actions from line three to five (DragTo, Click, Click). In this scenario, the two applications are displayed side-by-side.

7. Create spreadsheet of filenames (Looping) The purpose of this scenario is to create a list of filenames in a spreadsheet program, see Figure 3.18. In the running phase, HILC copies the filenames from within a given folder into successive Microsoft Excel cells. While repeatedly successful, the paste operation targeted the cell below the previously-selected (dark outline) cell on the spreadsheet. So the first entry will always be pasted below the initially selected cell.

8. Create BibTex from spreadsheet (Looping) This is the most complex of all the scenarios listed here. The task involves switching between three different applications (eight different screens). An instructor needs to plan out the task's steps, to ensure each application is in a state that is ready for the same action of the next loop to be executed. In the running phase, HILC works through a Microsoft Excel file that lists titles of papers that should be cited. The system then uses Google Scholar website to search for the BibTex of each paper, and produces a single BibTex file listing all the citations using the Notepad program. Figure 3.19 illustrates

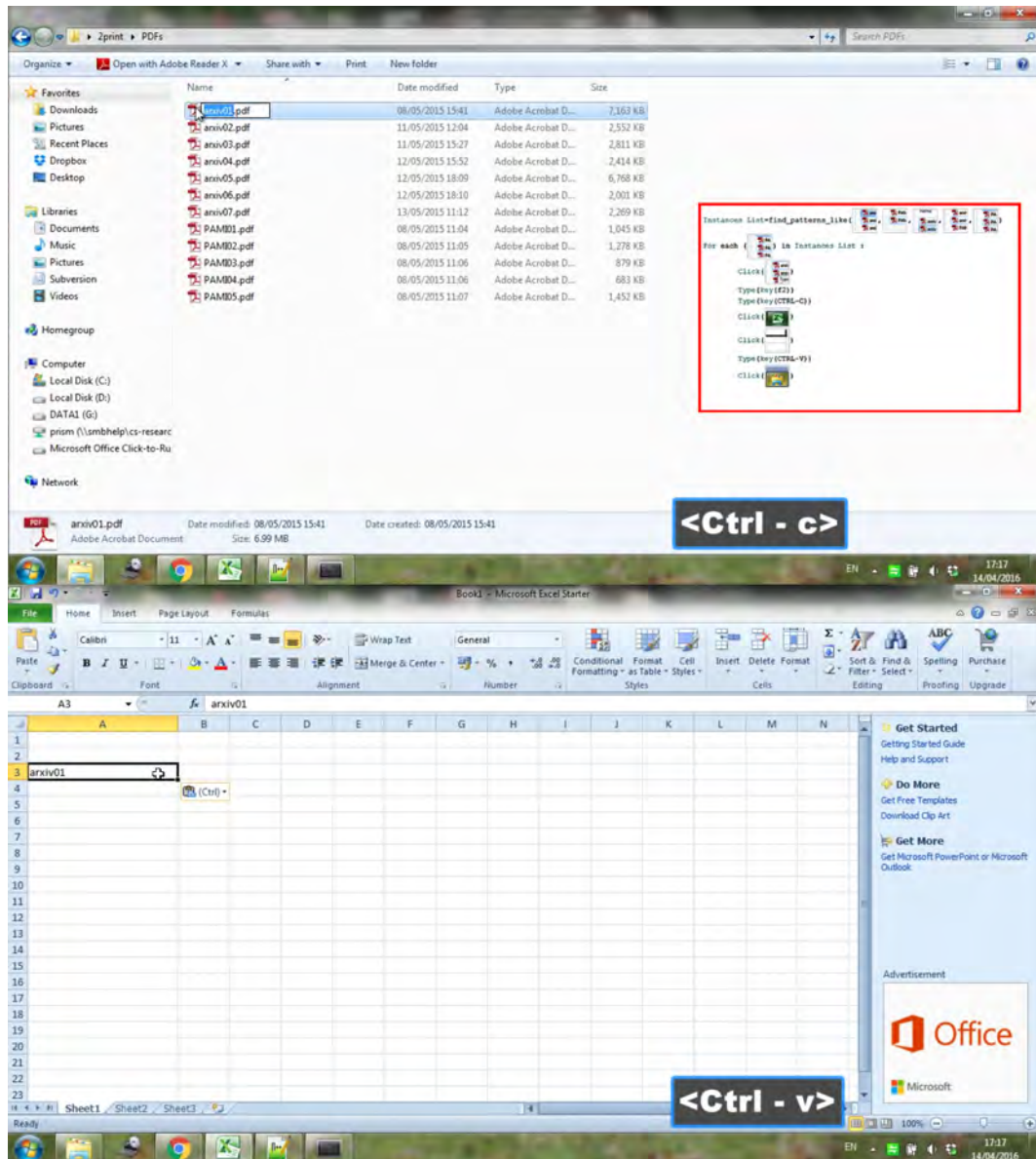


Figure 3.18: Two application screens from Scenario 7, where file names are being collected into a spreadsheet. The script of the task, in the red frame, involves switching back and forth between the two applications, and pasting the text into similar-looking cells.

the generated script.

9. Move MSWord files to a folder (Looping-Video) In this scenario, a further proof of concept of HILC was tested. The system successfully uses *only* video from a screencast software as input, instead of data from the sniffer, illustrating that instructors could post how-to-videos online, which can then easily be refined into a working script.

```

Instances List=find_patterns_like (
  [ig context
  isting Using],
  [gmenatic
  ting the vi
  g context],
  [lically ider
  ang GUI sc
  reverse er]
)
For each (
  [lically ider
  ang GUI sc
  reverse er]
) in Instances List :
  Click (
    [menting a
    identify]
  )
  Click (
    [ade
    ]
  )
  Type (key (SHIFT-Home))
  Type (key (CTRL-C))
  Click (
    [
    ]
  )
  Type (key (CTRL-V))
  Type (key (enter))
  Click (
    [progress a
    is Cite]
  )
  Click (
    [BibTeX]
  )
  Type (key (CTRL-A))
  Type (key (CTRL-C))
  Click (
    [
    ]
  )
  Click (
    [Scholar]
  )
  Click (
    [
    ]
  )
  Type (key (CTRL-V))
  Type (key (enter))
  Click (
    [
    ]
  )

```

Figure 3.19: A synthesized script of Scenario 8, where a BibTeX file is automatically constructed from a list of paper titles. Three different desktop GUI’s were involved. The user was able to train the system quite easily, and can just run the task without further instructions when writing her next research paper.

The scenario starts by executing a sequence of linear basic actions to create a new folder. It then continues to iteratively Drag and Drop each Microsoft Word file into the newly created folder. The script is shown in Figure 3.20.

3.4 Discussion and Future Work

The two sets of evaluation scenarios showed that the proposed approach substantially extends the Programming by Demonstration functionality that was available to non-programming users of desktop-automation tools. The main innovation is the sanity-check performed when the instructor demonstrates their task: given a cooperative human, it allows the system to transition from a winner-takes-all template-matching view of targets and actions, into a supervised-classification interpretation of the instructor’s intentions.

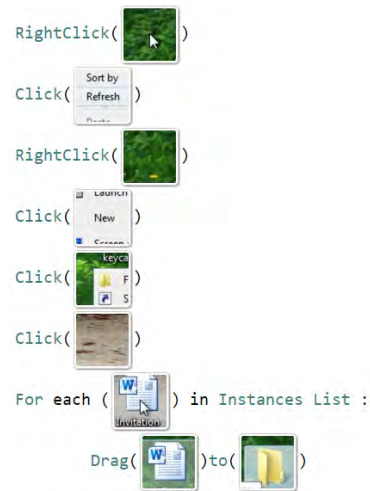


Figure 3.20: HILC successfully use videos of a screencast software as input of the system, instead of generating the input log-file from the sniffer, to create a working script. It is noteworthy that the system failed to remove mouse pointer from the target patterns in the first and the seventh lines.

This prototype has important opportunities for improvements. Basic actions are occasionally misclassified, when none of them has a high probability. Tests showed the joint segmentation and classification algorithm has an average accuracy of 95.4% for classifying each basic action. HILC allows users to fix misclassified actions instead of requiring a user to re-record the instruction again. Users were more successful and could do more with the system, but found the concept of supporters somewhat foreign, at least as presented in the provided instructions.

Currently, the system works without the awareness of states of the computer. For example, if a task expects to work with a pre-opened folder (or to open a closed one), the end-user must prepare their desktop appropriately. In addition, short fixed-length *sleep()* after each action to account for loading time of the computer is inserted because the system cannot know if the OS task has finished/ web-page has loaded. Therefore, shorter sleeps would make automated tasks go faster, but could ask for actions before the GUI is ready. This could be addressed in the future by training the system to recognize computer states from visual signals.

Furthermore, the current appearance models have fixed size and aspect ratio, which can hurt accuracy when items in a list are short and wide. Learned appearance features, even spanning across devices, could emerge, given enough training

footage.

This chapter successfully proves that Computer Vision and user feedbacks can replace the need for the special APIs of the PbD systems. Although using key combinations to distinguish between different programming paradigms and to indicate different parts of the demonstration are effective, it restricts usability of the system to users who understand basic programming concept such as looping. In the next chapter, a new demonstration procedure is introduced to improve usability and robustness of the visual-based PbD system.

Chapter 4

Looping GUI Action Automation

After it has been shown in the previous chapter that visual-based programming by demonstration is viable in learning tasks from various programming paradigms, a question has been raised that “What make the demonstration hard for casual users and how that can be remedied?” In this chapter, a new demonstration procedure for looping task, which is easier for casual users to perform, is designed and a preliminary user study had been carried out to identify all issues. The study showed that while users demonstrated loops, they usually included both intentional and accidental variations. To answer the research question, this chapter focuses on improving demonstration and learning of looping tasks.

Looping tasks are tasks which require performing a similar set of steps at all semantically related objects. Teaching HILC to generate a script for a looping task involves executing special key combination at the right moments and require users to have basic notions of looping iterators. The proposed system in this chapter, RecurBot, on the other hand, only requires users to demonstrate a few example loops similar to when human teaches a task to other humans. From that demonstration, the system automatically discovers where the start and the end of each loop are, and finally performs, auto-complete, the remaining actions in the task. For example, a user can send customized SMS messages to the first three contacts in a school’s spreadsheet of parents; then the proposed system loops the process, iterating through the remaining parents.

Software agents, known as bots, are still very far from fulfilling the seamless

learning and skill-transfer dreams of Maes [65], Negroponte [74], and Kay [49]. Karpathy's Mini World Of Bits [48], part of OpenAI's Universe platform, will be a proving-ground for bots, especially ones that exploit reward functions to perform reinforcement learning. It is a truism that even computer-literate users over-estimate modern bot capabilities until they are asked to automate a repetitive task themselves. There are two main misconceptions. **Myth 1:** "Modern bots can see, but just lack good human interfaces." Bots can indeed grab screenshots, and they can attempt local Optical Character Recognition. However, they can not systematically understand the GUI elements in terms of grouping widgets [42] or identifying interactive buttons. The diversity in mobile app GUIs makes this harder than ever. Without training data, scene-understanding of GUIs is not especially easier than scene-understanding of satellite images. **Myth 2:** "The bot can just ask the operating system (OS) without computer vision of the GUI." Even Open Source OS's like Android restrict developers (within an app) to the narrow parameters of accessibility API's, and those cannot retrieve the complete visual information of a GUI [58]. Even though OS-specific sniffer software can detect that a mouse button was pressed twice at some position, it cannot even be sure if the particular application interpreted it as two clicks or one double-click.

RecurBot, proposed in this chapter, addresses a desktop version of the Programming by Demonstration problem. It lies at the intersection of intention-inference, software usability, and action recognition and prediction. It lets a user teach a bot, much like they would teach a human, to perform a repetitive task. Consider the example shown in Figure 4.1. Here, the user is renaming each file in Google Drive to match a list of names given in an Excel spreadsheet. This type of looping task is common to most computer users, and it is only the experts, who have access to the scripting tools required, to automate them. The proposed algorithm takes the user's mouse/keyboard events as inputs, and from that initial user-demonstration of the task, it segments and extrapolates what was different about each loop, to complete the task automatically. Like Microsoft Excel's AutoFill, users want to extrapolate from these first few inputs, rather than cloning them.

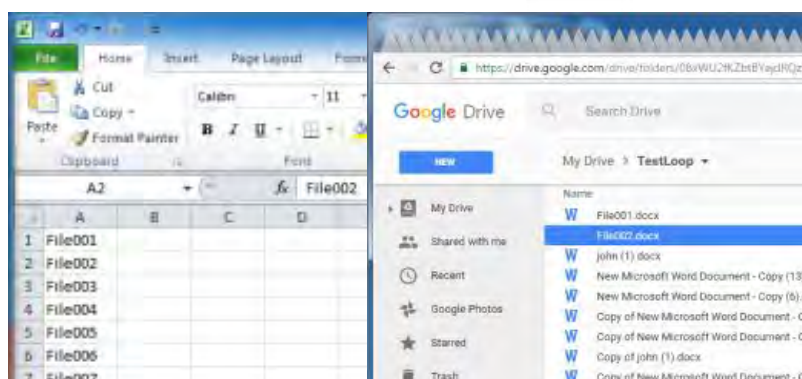


Figure 4.1: A looping GUI task, where the user is renaming files on Google Drive to match names in an Excel spreadsheet. This type of task is long and tedious, and hard for a typical computer user to automate. Our system is designed to learn to complete such tasks by watching a user performs a few demonstration iterations.

Aptly, Excel’s FlashFill is touted as an important milestone [37] for practical inductive programming, because it extrapolates non-sequential patterns, *e.g.*, parsing of initials from people’s names. Unlike Microsoft’s applications, RecurBot’s input comes from many diverse apps, bitmaps of heterogeneous content, and *noisy time-series human demonstrations*, where order matters.

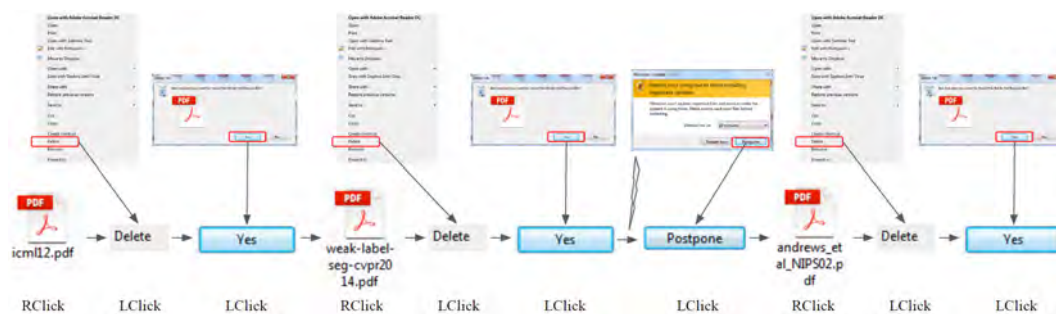


Figure 4.2: An example problem setup. This diagram shows that the proposed system allows detour actions, Windows’ popup asks for restarting the system, that can accidentally happen during the demonstration process. Without that mechanism the user need to re-record everything again from scratch. It is noteworthy that the diagram shows extracted objects, PDF file icons, buttons, *etc.*, which linked with actions instead of in reality the system only has access to whole screen screenshots and their corresponding mouse pointer locations.

4.1 Overview

The main contribution of this chapter is an algorithm for programming by demonstration of looping GUI tasks. The proposed visual motif analysis overcomes three main challenges: (1) The user-demonstrated loops are non-identical. The community working on set-based Motif-finding has avoided visual problems, and seeks to identify a perfect subsequence that was repeated K times, given N actions and the number of motifs K . (2) The demonstrated actions are typically iterating through the initial loops of a lengthy task. To automatically predict and execute the remaining loops, one must detect the one or more iterators implied by the demonstration. (3) Compared to passive action-recognition, recognizing and then predicting GUI tasks requires a video dataset annotated with acceptable-interaction meta-information.

Motif-finding and action-prediction are formidable challenges because human-computer interactions are highly variable. When a user does the first three loops of a task, each loop may have *both* extra and missing actions. Actions may be missing because a window already had focus or a text-entry was already filled in. Extra actions can occur when the user performs extra clicks without a specific purpose (*e.g.*, double-clicking a hyperlink) or gets interrupted by a tangential task, like suppressing an update message (see Figure 4.2). This variability is captured in the proposed new dataset (see Table 4.1), which motivates this work and establishes metrics for progress toward the goal of interactive software agents. The supplemental video, in the project page, further illustrates the algorithm’s prototype, where predicted action-previews are shown to the user for approval before being performed.

4.2 Looping Action Recognition

The goal of this work sets out to recover and predict looping actions from input data. The algorithm takes as input a sequence of basic actions, $\mathbf{A} = (A_0, A_1, A_2, \dots, A_N)$, obtained using [43], which is otherwise unhelpful here. Figure 4.2 shows the input sequence for a simple example task, where the user has used click actions in different locations to sequentially delete one type of file from a folder. Each basic action A_n is a tuple containing the *action type* a_n , together with, where appropri-

No. of Sequences	Actions/Sequence	Actions/Loop
55	16.36	4.33
Missing Actions /Seq	Noisy Actions/Seq	% Variation Seq
0.29	0.65	38%

Table 4.1: Average statistics of the proposed Demonstration Dataset. These data serve for training and testing of just the analysis part of the visual motif-finding, analogous to typical (non-visual) motif-finding challenges. Here, each sequence is a unique task, made of basic GUI interactions (Actions), performed by 7 different computer users. Test users performed the first 3 or 4 loops of each task, and these were labeled to quantify experimental performance. Loops within the same sequence naturally differ from each other by having extra, missing, or iteratively changing actions. *% Variation Seq* is the % of sequences that have at least one user variation, either noisy or missing.

ate, a screenshot and mouse cursor location. The action types which are used are: LeftClick, RightClick, DoubleClick, ClickDrag, and Typing.

Assuming that there is a sequence of actions \mathbf{T} which, in the user’s mind, is the ‘true’ sequence. Template \mathbf{T} contains the ground truth sequence of events that the user wishes to be repeated to complete their long chain of iterative tasks. If, in demonstrating the task, the user just perfectly performed \mathbf{T} once, the problem would *still* be difficult, as:

1. The computer vision system seeks to find visual similarity between elements, and if the loop is only performed once, then the system only has a single training example for each future loop prediction.
2. The prediction of future actions relies on knowing about *iterative changes* between user actions in different loops. For example, loop two might require a click below the corresponding click in the first loop. This cannot be learned from a single loop.
3. In reality, even when a user tries to complete a sequence correctly, they typically deviate from the true sequence. Multiple repetitions help the algorithm to discover the ‘true’ intention of the user.

RecurBot therefore asks that the user performs several loops. The proposed *motif-finding algorithm* is then used to recover a set of divided subsequences, given \mathbf{A} .

The only assumption which is made is that the input full sequence \mathbf{A} contains at least one ‘good enough’ sequence $\tilde{\mathbf{T}}$ which is functionally equivalent to the true sequence \mathbf{T} . A sequence $\tilde{\mathbf{T}}$ is a sequence of actions which performs the same task as \mathbf{T} , despite having some minor additions or deviations from the ideal. Other instantiations of \mathbf{T} in \mathbf{A} may be subject to extra, unwanted actions being performed, or missing actions from the sequence. This proposed motif-finding algorithm is usually able to deal with these issues.

4.2.1 Basic Motif Finding

Given a sequence \mathbf{A} , the purpose of basic motif finding algorithms is to identify just a pair of subsequences $(\mathbf{S}_i, \mathbf{S}_j)$, each of which is composed of equivalent actions executed in the same order [70, 71]. Because only a single pair is identified, this is not suitable for the purpose of this work where all loops in \mathbf{A} are desired to be discovered.

Bagnall et al. [11] propose two greedy algorithms for finding a *set* of motifs from an input sequence. They find the best matching pair of subsequences from all the existing subsequence pairs in \mathbf{A} first, then greedily match other non-overlapping subsequences to this bootstrap pair. There are, however, fundamental limitations to this greedy style algorithm. The largest problem is that it can be trapped in a local minimum — if a bad initial pairwise match is found, then the algorithm can not recover. These issues are demonstrated experimentally in Section 4.5. Further, they require the motif length to be specified by the user. In comparison to these basic methods, the proposed algorithm in this chapter maintain a set of candidate matches while iterating over different length subsequences.

4.2.2 Distance Between Two Sequences $\text{Dist}(\mathbf{S}_i, \mathbf{S}_j)$

Crucial to the motif finding algorithm is a method for determining the similarity between a pair of candidate subsequences $(\mathbf{S}_i, \mathbf{S}_j)$. The algorithm requires a distance measure $\text{Dist}()$ which is small when the two subsequences perform the same task,

and is large otherwise. For example, given the following three sequences:

$$\begin{aligned}
 A &= \left(\begin{array}{c} \text{PDF} \\ \text{icml12.pdf} \end{array}, \begin{array}{c} \text{Create sh} \\ \text{Delete} \\ \text{Rename} \end{array}, \begin{array}{c} \text{Yes} \end{array} \right), \\
 B &= \left(\begin{array}{c} \text{PDF} \\ \text{weak-label-} \\ \text{seg-cvpr20} \\ \text{14.pdf} \end{array}, \begin{array}{c} \text{Create sh} \\ \text{Delete} \\ \text{Rename} \end{array}, \begin{array}{c} \text{Yes} \end{array} \right), \\
 C &= \left(\begin{array}{c} \text{PDF} \\ \text{andrews_et} \\ \text{al_kuroki} \end{array}, \begin{array}{c} \text{Postpone} \\ \text{Delete} \\ \text{Rename} \end{array}, \begin{array}{c} \text{Create sh} \\ \text{Delete} \\ \text{Rename} \end{array} \right),
 \end{aligned} \tag{4.1}$$

It is expected A to have a smaller distance to B than it does to C . Each action is a tuple containing the basic action a and, where applicable, screenshots at the start and end of the basic action, and the respective mouse locations. For clarity, in this example only the extracted screenshots are depicted. Dist is defined as the sum of individual differences between corresponding pairs of actions, *i.e.*,

$$\begin{aligned}
 \text{Dist}(\mathbf{S}_i, \mathbf{S}_j) &= \sum_{z=0}^L d(S_{iz}, S_{jz}), \text{ where} \\
 d(S_{iz}, S_{jz}) &= d_{Action}(S_{iz}, S_{jz}) + d_{Obj}(S_{iz}, S_{jz}).
 \end{aligned} \tag{4.2}$$

The difference between actions, d_{Action} , is a simple binary indicator function, applying infinite penalty if the basic actions (*e.g.*, `LeftClick`, `ClickDrag`) performed did not match, and a fixed cost of ε where they do. ε is a small constant which makes the distance between different subsequence length different. d_{Obj} is a visual matching penalty, evaluated by comparing the Normalized Cross Correlation between shifted sub-images of the screenshots extracted from the region around the cursor position when the action took place.

The difference between actions, d_{Action} , is defined as a simple binary indicator function, applying infinite penalty if the basic actions performed did not match:

$$d_{Action}(S_i, S_j) = \begin{cases} \varepsilon & \text{if } a_i = a_j \\ \infty & \text{otherwise.} \end{cases} \tag{4.3}$$

While d_{Action} compares two actions performed by the user, d_{Obj} captures differences between the visual data captured along with the two actions:

$$d_{Obj}(S_i, S_j) = 1 - \frac{NCC_{\max}(S_i^l, S_j^s) + NCC_{\max}(S_i^s, S_j^l)}{2} \quad (4.4)$$

$NCC_{\max}(S_i, S_j)$ is the maximum value from the result of the Normalized Cross Correlation between the screenshot images of S_i and S_j . The cursor position during the interaction is used to extract a sub-image from the screenshot. Superscript s indicates that a *small* sub-image is extracted and used for matching, while superscript l indicates that a *larger* sub-image is used. The small and large image sizes in our experiments are 61×61 and 101×101 respectively.

Only using Dist strongly favors trivial pairwise matches of length 1. Therefore, a *normalized* distance, which favors longer subsequences, defined as

$$\text{NormDist}(\mathbf{S}_i, \mathbf{S}_j) = \alpha^{-|\mathbf{S}_i|} \text{Dist}(\mathbf{S}_i, \mathbf{S}_j), \quad (4.5)$$

is used where α is a small constant.

4.2.3 The Proposed Method for Multiple Motif Finding

The algorithm extends the exact time series motif discovery [70, 11] by jointly adding to a *set* of motifs instead of just a single pair. By assumption, the user has provided K , the number of times they have performed the loop. The algorithm starts by finding a set \mathcal{C} of candidate pairs of *single actions*. This set includes all pairs which have a NormDist smaller than a threshold r . Each of these pairs is one possible candidate seed (ideally, two-of-a-kind), which could grow into a full solution. For the example in Figure 4.2, this might be:

$$\mathcal{C} = \left\{ \left(\begin{array}{c} \text{PDF} \\ \text{icml12.pdf} \end{array}, \begin{array}{c} \text{PDF} \\ \text{andrews_et} \\ \text{at_kuroon?} \end{array} \right), \left(\begin{array}{c} \text{PDF} \\ \text{icml12.pdf} \end{array}, \begin{array}{c} \text{Yes} \end{array} \right), \left(\begin{array}{c} \text{weak-label-} \\ \text{seg-cvpr20} \\ \text{14.pdf} \end{array}, \begin{array}{c} \text{PDF} \\ \text{andrews_et} \\ \text{at_kuroon?} \end{array} \right), \dots \right\}.$$

Each pair is then grown by finding the closest matching action from \mathcal{A} not yet in the pair. This creates a set of 3-tuples:

$$\mathcal{C} = \left\{ \left(\begin{array}{c} \text{PDF} \\ \text{icml12.pdf} \end{array}, \begin{array}{c} \text{PDF} \\ \text{andrews_et} \\ \text{at_kuroon?} \end{array}, \begin{array}{c} \text{weak-label-} \\ \text{seg-cvpr20} \\ \text{14.pdf} \end{array} \right), \left(\begin{array}{c} \text{PDF} \\ \text{icml12.pdf} \end{array}, \begin{array}{c} \text{Yes} \end{array}, \begin{array}{c} \text{Yes} \end{array} \right), \right. \\ \left. \left(\begin{array}{c} \text{weak-label-} \\ \text{seg-cvpr20} \\ \text{14.pdf} \end{array}, \begin{array}{c} \text{PDF} \\ \text{andrews_et} \\ \text{at_kuroon?} \end{array}, \begin{array}{c} \text{PDF} \\ \text{icml12.pdf} \end{array} \right), \dots \right\}. \quad (4.6)$$

Some of these lists correctly contain multiple occurrences of the same action, while others do not. A second pruning stage shrinks this set to a manageable size. The algorithm then extends each 3-tuple of single items in \mathcal{C} into 3-tuples of two-actions, by adding subsequent actions from \mathbf{A} :

$$\mathcal{C} = \left\{ \left(\left(\begin{array}{c} \text{PDF} \\ \text{icml12.pdf} \end{array}, \begin{array}{c} \text{Create shor} \\ \text{Delete} \\ \text{Rename} \end{array} \right), \left(\begin{array}{c} \text{PDF} \\ \text{andrews_et} \\ \text{at_kubcon?} \end{array}, \begin{array}{c} \text{reate shortc} \\ \text{elete} \\ \text{ename} \end{array} \right), \left(\begin{array}{c} \text{weak-label-} \\ \text{seg-cvpr20} \\ \text{14.pdf} \end{array}, \begin{array}{c} \text{Create sho} \\ \text{Delete} \\ \text{Rename} \end{array} \right) \right), \\ \left(\left(\begin{array}{c} \text{PDF} \\ \text{icml12.pdf} \end{array}, \begin{array}{c} \text{Create shor} \\ \text{Delete} \\ \text{Rename} \end{array} \right), \left(\text{Yes} \right), \left(\begin{array}{c} \text{weak-label-} \\ \text{seg-cvpr20} \\ \text{14.pdf} \end{array}, \begin{array}{c} \text{Create sho} \\ \text{Delete} \\ \text{Rename} \end{array} \right), \left(\begin{array}{c} \text{PDF} \\ \text{andrews_et} \\ \text{at_kubcon?} \end{array}, \begin{array}{c} \text{reate shortc} \\ \text{elete} \\ \text{ename} \end{array} \right) \right), \dots \left. \right\}.$$

This process continues until there are no more items to add to each list. A score of each candidate in \mathcal{C} is assigned by measuring the average normalized distance between each pair of subsequences. The list in \mathcal{C} with the lowest score is returned as the discovered set of motifs \mathbf{R} . In this example, the expected output is

$$\left(\left(\left(\begin{array}{c} \text{PDF} \\ \text{icml12.pdf} \end{array}, \begin{array}{c} \text{Create shor} \\ \text{Delete} \\ \text{Rename} \end{array}, \begin{array}{c} \text{Yes} \end{array} \right), \left(\begin{array}{c} \text{weak-label-} \\ \text{seg-cvpr20} \\ \text{14.pdf} \end{array}, \begin{array}{c} \text{Create sho} \\ \text{Delete} \\ \text{Rename} \end{array}, \begin{array}{c} \text{Yes} \end{array} \right), \left(\begin{array}{c} \text{PDF} \\ \text{andrews_et} \\ \text{at_kubcon?} \end{array}, \begin{array}{c} \text{reate shortc} \\ \text{elete} \\ \text{ename} \end{array}, \begin{array}{c} \text{Yes} \end{array} \right) \right) \right). \quad (4.7)$$

It is noteworthy that the three separate loops have been correctly identified and segmented, and the equivalent actions in each loop have been aligned. Also, the extra action (of dismissing the popup window) has been correctly identified as noise, and is therefore not shown in this final result. In practice, several strategies are incorporated to make this process tractable, including ‘early abandon’ as the lists grow — full details are given in Algorithm 1.

4.2.4 Artificial Subsequences for Robustness

The algorithm so far has been assumed that each version of \mathbf{R} contained in \mathbf{A} is a perfect, unmodified copy. However, as has been discussed, many users will miss out actions or include extra unneeded actions. The user variations can be categorized into three classes: (a) Missing actions, where the user omits a single step; (b) Noisy actions between two subsequences, and (c) Noisy actions within a subsequence.

Algorithm 1 Looping Action Recognition**Input:** a sequence of actions \mathbf{A} and the number of demonstrated loops K .**Output:** a ranked list of identified looping subsequence alternatives ($\mathbf{R}_1, \dots, \mathbf{R}_r$)

```

1:  $\mathbf{R} \leftarrow \emptyset$ 
2:  $\mathbf{avgDists} \leftarrow \emptyset$ 
3:  $\text{Dist} \leftarrow \text{buildDistanceMatrix}()$   $\triangleright$  Cache distances to save computation
4:  $\mathbf{C} \leftarrow \{(A_x, A_y) \mid \forall A_x \in \mathbf{A}, \forall A_y \in \mathbf{A}; A_x \neq A_y\}$   $\triangleright L = 1$  candidates
5: for each  $(A_x, A_y)$  in  $\mathbf{C}$  do
6:    $\text{thisAns} \leftarrow \{A_x, A_y\}$ 
7:    $\text{thisDist} \leftarrow \text{Dist}(A_x, A_y)$ 
8:   for  $nn = 1 : K - 2$  do
9:      $\text{NN} \leftarrow \text{getNN}(A_x, A_y)$   $\triangleright \text{getNN}$  finds closest example to both  $A_x$  and  $A_y$ 
10:     $\text{thisAns.append}(\text{NN})$ 
11:     $\text{thisDist} += (\text{Dist}(A_x, \text{NN}) + \text{Dist}(A_y, \text{NN}))$ 
12:    $\mathbf{R.append}(\text{thisAns})$ 
13:    $\mathbf{avgDists.append}(\frac{\text{thisDist}}{K\alpha})$ 
14:  $\mathbf{R} \leftarrow \text{getBestTop}(\mathbf{R}, r)$   $\triangleright$  Sort all the candidates by  $\mathbf{avgDist}$ , and retain only
    the best  $r$ 
15:  $\text{minDist} \leftarrow \min(\mathbf{avgDists})$ 
16: for each  $L$  in  $(2, 3, \dots, \frac{N}{2})$  do
17:    $\mathbf{C}, \mathbf{D} \leftarrow \text{GenArtificialSubsequences}(\mathbf{C}, \mathbf{D})$   $\triangleright$  Create artificial subsequences
    (Algorithm 2)
18:   for each  $(\mathbf{S}_{ref}, \mathbf{S}_{gen}), D$  in  $\mathbf{C}, \mathbf{D}$  do
19:      $\text{thisAns} \leftarrow \{\mathbf{S}_{ref}, \mathbf{S}_{gen}\}$ 
20:      $\text{thisDist} \leftarrow \frac{D}{K\alpha^L}$ 
21:     for  $nn = 1 : K - 2$  do
22:       if  $\text{thisDist} > \text{minDist}$  then
23:         go to 18  $\triangleright$  Early abandon
24:        $\mathbf{S}_{NN} \leftarrow \text{getNN}(\mathbf{S}_{ref}, \mathbf{S}_{gen})$ 
25:        $\text{thisDist} += \frac{1}{K\alpha^L} (\text{Dist}(\mathbf{S}_{ref}, \mathbf{S}_{NN}) + \text{Dist}(\mathbf{S}_{gen}, \mathbf{S}_{NN}))$ 
26:        $\text{thisAns.append}(\mathbf{S}_{NN})$ 
27:       if  $\text{thisDist} < \text{minDist}$  then
28:          $\mathbf{R.append}(\text{thisAns})$ 
29:          $\mathbf{avgDist.append}(\text{thisDist})$ 
30:    $\mathbf{R} \leftarrow \text{getBestTop}(\mathbf{R}, r)$   $\triangleright$  Sort all the candidates by  $\mathbf{avgDist}$ , and retain
    only the best  $r$ 
31:    $\text{minDist} \leftarrow \mathbf{R}[\text{last}].\text{distance}()$ 

```

Figure 4.2 shows an example of a noisy action within a subsequence, where the user has dismissed a system dialogue with a click while demonstrating the loops.

To cope with these user variations, the algorithm generates *artificial subsequences* during the solving process. It extends \mathcal{C} with copies of items within \mathcal{C} , each of which has some user actions removed, or extra ones appended. These simulate user variations, helping to improve the matching between noisy input subsequences. Distance measures computed from or to any of these generated subsequences has an additional penalty Γ added, defined as

$$\Gamma(\mathbf{S}_j) = \eta^a + s\beta, \quad (4.8)$$

where a is the number of appended actions and s is the number of skipped actions; η and β are penalty costs added for each appended actions and skipped actions respectively. Full details are given in Algorithm 2.

The parameters α , β , and η are learned using ground truth data to create distance matrices where perfectly matched actions have distances 0.1, otherwise ∞ . The optimization is

$$\alpha, \beta, \eta = \arg \min_{\alpha, \beta, \eta} \left\{ - \sum_{\tilde{t}_i \in \tilde{T}} (F(\mathbf{A}_i | \alpha, \beta, \eta) = \tilde{t}_i) \right\}, \quad (4.9)$$

where \tilde{T} is the training set and $F(x)$ is the proposed looping action recognition algorithm, which takes as input sequence of actions \mathbf{A} and outputs a list of subsequences \mathbf{R} .

4.3 Prediction of Future Actions

The discovered sets of loops are used to predict the user's intended actions. The best discovered set of subsequences \mathbf{R} effectively forms a training set to enable this inference. See, for example, the columns in Figure 4.6, or the corresponding elements in each tuple in Eq 4.7.

For each discovered action A_i , the set of corresponding actions is constructed using each of the K subsequences in \mathbf{R} . This gives up to K cropped *training images*

Algorithm 2 Generating Artificial Subsequences

Input: a list of length L candidate pairs $\mathbf{C} = \{(\mathbf{S}_{\text{ref}0}, \mathbf{S}_{\text{gen}0}), \dots, (\mathbf{S}_{\text{ref}C}, \mathbf{S}_{\text{gen}C})\}$, the list of distances between subsequences of the pairs $\mathbf{D} = \{D_0, \dots, D_C\}$, the sequence of actions \mathbf{A} , and the best so far minimum distance min .

Output: a list of length $L + 1$ candidate pairs $\tilde{\mathbf{C}} = \{(\mathbf{S}_{\text{ref}0}, \mathbf{S}_{\text{gen}0}), \dots, (\mathbf{S}_{\text{ref}\tilde{C}}, \mathbf{S}_{\text{gen}\tilde{C}j})\}$ and the list of distances between subsequences of the pairs $\tilde{\mathbf{D}} = \{D_0, \dots, D_{\tilde{C}}\}$.

```

1: function GenArtificialSubsequences( $\mathbf{C}, \mathbf{D}$ )
2:    $\tilde{\mathbf{C}} \leftarrow \emptyset$ 
3:    $\tilde{\mathbf{D}} \leftarrow \emptyset$ 
4:   for each  $(\mathbf{S}_{\text{ref}}(x), \mathbf{S}_{\text{gen}}(y)), D$  in  $\mathbf{C}, \mathbf{D}$  do
5:     if  $A_{x+L+1} \notin \mathbf{S}_{\text{gen}}$  then
6:        $\mathbf{S}_{\text{ref}}.\text{append}(A_{x+L+1})$ 
7:        $\mathbf{S}_{\text{gen}}.\text{append}(A_{x+L+1})$ 
8:       if  $\frac{D+\eta^{a+1}}{\alpha^L} < \text{minDist}$  then ▷ add an appended sequence
9:          $\tilde{\mathbf{C}}.\text{append}((\mathbf{S}_{\text{ref}}, \mathbf{S}_{\text{gen}}))$ 
10:         $\tilde{\mathbf{D}}.\text{append}(D + \eta^{a+1})$ 
11:       for each  $s$  in  $\{0, \dots, n\}$  do ▷ add a normal sequence ( $s = 0$ ) and
skipped sequences  $s = 0, \dots, n$ 
12:         if  $A_{y+L+1+s} \notin \mathbf{S}_{\text{ref}}$  then
13:            $i_{\text{ref}} \leftarrow x + L + 1$ 
14:            $i_{\text{gen}} \leftarrow y + L + 1 + s$ 
15:            $n\text{Dist} \leftarrow D + s\beta + \text{Dist}(A_{i_{\text{ref}}}, A_{i_{\text{gen}}})$ 
16:           if  $\frac{n\text{Dist}}{\alpha^L} < \text{minDist}$  then
17:              $\mathbf{S}_{\text{ref}}.\text{append}(A_{i_{\text{ref}}})$ 
18:              $\mathbf{S}_{\text{gen}}.\text{append}(A_{i_{\text{gen}}})$ 
19:              $\tilde{\mathbf{C}}.\text{append}((\mathbf{S}_{\text{ref}}, \mathbf{S}_{\text{gen}}))$ 
20:              $\tilde{\mathbf{D}}.\text{append}(n\text{Dist})$ 
return  $\tilde{\mathbf{C}}, \tilde{\mathbf{D}}$ 

```

for each action in the loop. This set of crops associated with action A_i is denoted as \mathcal{H}_i . The system iteratively plays back each action A_i at time t at a predicted screen location (x^*, y^*) , computed using Bayes' Theorem as

$$\begin{aligned}
x^*, y^* &= \arg \max_{x, y} P(x, y | \mathcal{H}_i, I) \\
&= \arg \max_{x, y} P(\mathcal{H}_i, I | x, y) P(x, y), \tag{4.10}
\end{aligned}$$

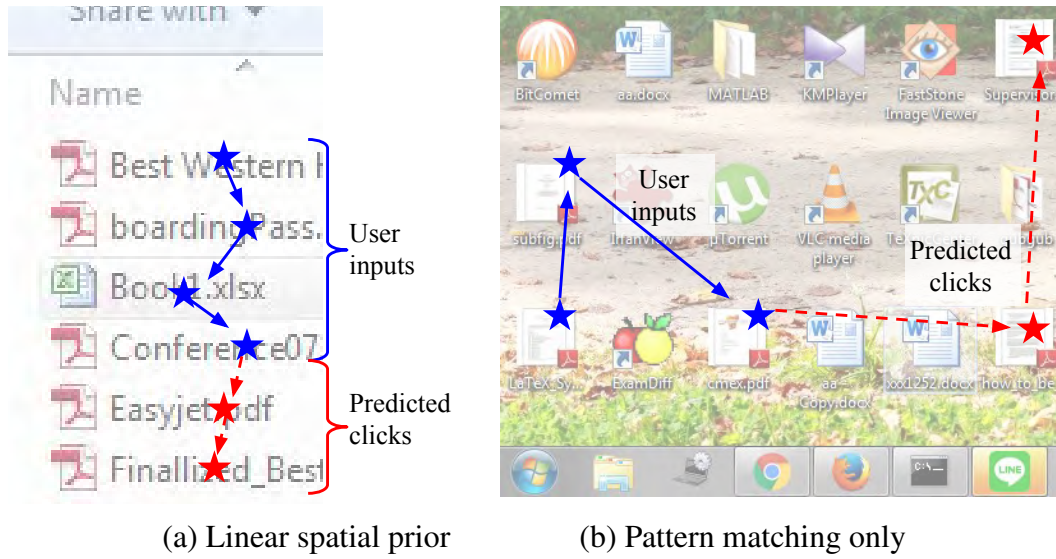


Figure 4.3: The proposed future action prediction algorithm chooses *where* to apply action events, based on the small number of user demonstrations. (a) If the demonstration events follow a linear pattern, the spatial prior uses linear regression to predict action locations. (b) If the demonstration events do not exhibit spatial correlation, then the spatial prior becomes uniform and the pattern matching likelihood becomes dominant. Here, it learns to locate the ‘.pdf’ files on a cluttered desktop.

where I is the current screenshot. The evidence is computed as

$$P(\mathcal{H}_i, I | x, y) = \prod_{H \in \mathcal{H}_i} \text{NCC}_{x,y}(I, H), \quad (4.11)$$

where $\text{NCC}_{x,y}$ computes the normalized cross-correlation when the crop H is overlaid on the screenshot I at location x, y . Eq 4.11 effectively finds the x, y location in the current screenshot which best agrees with the *visual appearance* of the user-demonstrated actions.

Eq 4.10 includes a location prior $P(x, y)$, which is based on the relative offset of user clicks over time. While some actions always occur at the same x, y location, others occur at a different location each time, for example when sequentially clicking on each checkbox on a web page or each row in a spreadsheet. These behaviors are modeled by assuming a linear relationship between locations being clicked on the same action of each loop (Figure 4.3(a)). Due to the low number of exemplars for each action, $P(x, y)$ is modeled using a linear regression model, assumed to be

independent for each dimension:

$$P(x, y) = P(x)P(y) \quad (4.12)$$

$$= \text{Norm}_x[\phi_x t, \sigma_x^2] \text{Norm}_y[\phi_y t, \sigma_y^2], \quad (4.13)$$

where σ_x^2 , σ_y^2 , ϕ_x and ϕ_y are inferred from the training examples using maximum likelihood learning. Where σ_x^2 and σ_y^2 are above a threshold, it is assumed that there is no spatial dependency between iterations of the loop, and the *visual* matching should dominate (Figure 4.3(b)). To ensure this, the prior probabilities which overlap with previously clicked locations is set to zero to prevent the trivial solution of previously selected locations being re-suggested. Here, additive smoothing is applied to the probabilities in Eq 4.10 to prevent issues arising from zero probability areas.

4.3.1 Human-in-the-loop

A good PbD system should let users know the next action that the system is going to execute, and allow users to approve or, if necessary, modify the action [52]. After predicting the most likely action location x^*, y^* at time step t , RecurBot shows the user an animation of the proposed action and asks them to approve or correct the action. The user's response is added to the information 'bucket' for this action, used for recomputing ϕ_x , ϕ_y , σ_x^2 , and σ_y^2 for the action at the next iteration. This means that as the human interacts with the system, it learns more about relative offsets between action locations.

The stopping condition for looping is set to 70% of the first detected maximum posterior, $\max(P(x, y | \mathcal{C}_i, I))$. When the maximum posterior falls under the threshold, the system asks the user whether to stop or continue. If the user elects to continue, the stopping threshold is then updated to 90% of the current maximum posterior.

4.4 Datasets

For experimental validation of the algorithm two datasets were created. Colleagues were informally surveyed to identify GUI tasks that they found repetitive. From those, tasks that span different lengths, input modalities, apps and GUI interfaces, complexities, and repetitions were distilled. The proposed algorithm were tested with the two datasets in aspects of looping action recognition and the complete pipeline. These fully annotated datasets are made available with the proposed algorithm. The datasets can be found in the project page: <http://visual.cs.ucl.ac.uk/pubs/RecurBot/>

4.4.1 Demonstration Dataset

The dataset comprises 55 tasks for quantitative evaluation of the motif-finding algorithm. Each sequence was recorded by asking 7 experienced computer users to perform the first four or so loops of specific repetitive GUI tasks. While working with their knowledge, they were recorded by the sniffer-software that captured both mouse/key events, and screenshots throughout each task. The mouse/key events in this dataset, and all sniffer-events observed at test-time, are converted into actions (*e.g.*, single-click, double-click, click-drag, *etc.*,) using the basic version of [43]. Each task's action-transcript was then annotated, identifying the boundaries between loops and tagging the parts of each loop that included either extra actions or were missing actions, as compared to the other loops in the task.

Table 4.1 demonstrated statistics of this dataset. On average, each loop in this dataset has 4.33 actions. A test user performed the first 3 or 4 loops of each task, and these were labeled to quantify experimental performance. Because real users are not perfect, loops within the same sequence naturally differ from each other by having extra, missing, or iteratively changing actions. On average, 65% of sequences have noisy actions, and 29% have missing actions.

4.4.2 Looping GUI Automation Dataset

This dataset is used to benchmark the complete pipeline of the algorithm, the Looping Action Recognition and the Action Prediction. The systems evaluated on this

dataset are challenged with correctly predicting the type and location of future actions, given the user demonstrations as training data. The dataset comprises 15 tasks of fully annotated basic actions, including a user’s demonstration, all subsequent interactions required to complete the task, and the mask for every basic action. This action mask enables future action predictions to be evaluated. Any prediction, of the correct action type, which falls within the marked area in the mask is deemed to be correct. The testing protocol is outlined in Appendix A and B, which is itself a contribution, necessary for making reproducible GUI-action evaluations.

4.5 Validation of the Algorithms

First, the system’s ability to recognize looping actions in the user’s demonstrated loops is measured using the **Demonstration Dataset**. The proposed looping action recognition algorithm is able to produce a ranked list of possible answers, sorted by the average value of normalized distances between every pair of motifs inside the answer. The fraction of tasks, where the correct answer is within the top k answers returned by the algorithm, is counted up and this success rate is plotted against different values of k . Figure 4.4 demonstrates the performance of the algorithm compared to three baselines: *Division*, *Motif* and *Greedy motif*.

Division is the simplest baseline, which assumes that there is no human variation in the demonstration, thus the input sequence of actions can be segmented into K equal-length subsequences at every $\frac{N}{K}$ actions. This algorithm therefore does not rely on the distance measures. *greedMotif* [11] is a standard motif-finding algorithm which first finds the best pair of motifs, then grows a set of motifs from that. *Motif* is an improved version which is modified from the standard motif-finding algorithm to grow a set of motifs from all possible subsequence pairs instead of growing only from the best pair of each length. Both *Motif* and *greedMotif* perform equally well, gaining around 10% improvement over *Division*. This improvement is due to the robustness that these motif discovery algorithms have when presented with noisy actions between looping subsequences. It is noteworthy that the greedy algorithm is very sensitive to the distance function between two visual patterns. When *greed-*

Motif does not have access to the ground truth distance matrix, *NCC-greedMotif*, the performance is far worse than the *Division* algorithm.

NCC+Noisy is the proposed algorithm without the missing actions solver. It gains about 15% improvement from *Motif*. This shows that there are two types of noisy actions. The first are actions between looping subsequences, which can be solved using the ordinary motif discovery algorithm. The second are the noisy actions within a looping subsequence, which can be addressed by including artificial skipping actions to the list of candidates. Lastly, *GT+Noisy+Missing* shows the performance of the proposed looping action recognition algorithm when it has access to the ground truth distance matrix between actions. So what happens to the prediction when motif-finding fails? The 4% of sequences which failed in this case did not have even one good enough looping subsequence in the input. Hence, they violate the only assumption of the algorithm. By disapproving and overriding predictions, the system gets extra chances to improve. Figure 4.6 demonstrates an example output of our looping action recognition algorithm.

Next the action prediction algorithm is evaluated on the **Looping GUI Automation Dataset**. The prediction is masked as correct if the user simply accepts the default suggestion using the space bar. If the human was required to *modify* a predicted action, this is counted as an Incorrect prediction. The correctly predicted actions, the number of incorrectly predicted actions, and the number of human interventions needed when the prediction scores are below the stopping threshold are counted. The system's success in assessing when to stop iterating is also measured. When the system prematurely prompts the user to halt iterating, this failure is counted as an 'Early-stopping' failure.

After training each task with 3 demonstrated loops, how many actions were correctly predicted (so the user simply approved the default: either an action or the decision to terminate) are measured. 85.78% of sequences had all their actions correctly predicted. In 6.90% of tasks the score fell below the stopping criterion and so the system recommended exiting; this included tasks where the next prediction would have been correct. The final 7.33% of tasks were confidently predicted,

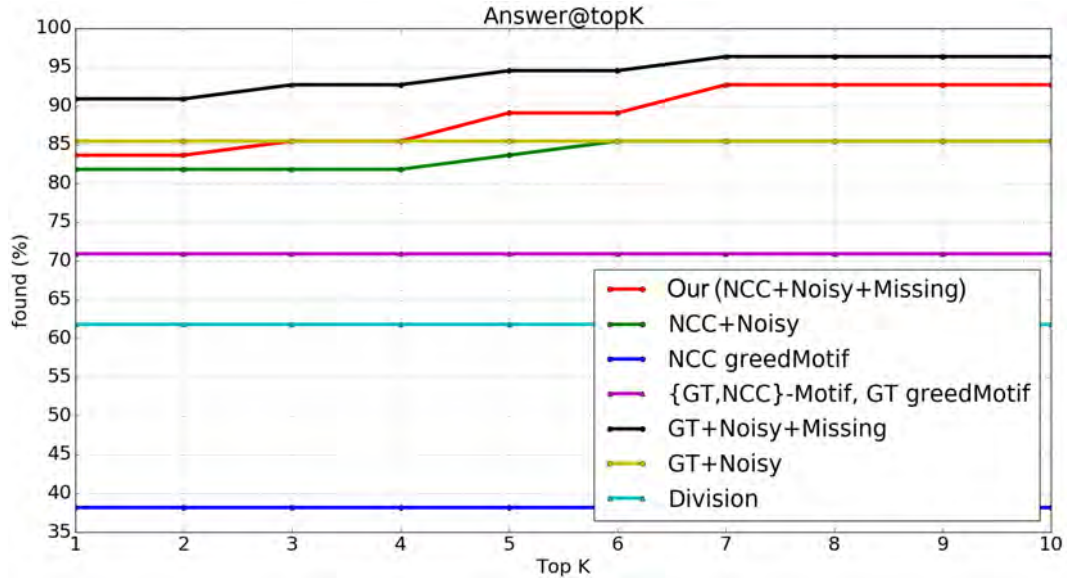


Figure 4.4: Quantitative results of the proposed looping action recognition algorithm on the Demonstration Dataset. The proposed algorithm, NCC+Noisy+Missing, is compared against three baselines: Division, greedMotif, and Motif algorithms. GT in the name indicates that the algorithm has access to the ground truth distance matrix instead of using Eq 4.2, while NCC indicates that the algorithm use the proposed Normalized Cross Correlation as the distance function between two visual objects. GT Motif, NCC Motif and GT greedMotif share the same graph at {GT,NCC}-Motif, GT greedMotif. GT+Noisy+Missing demonstrates the accuracy of the proposed algorithm when it has access to the ground truth distance matrix. NCC+Noisy is the ablation study, showing the result when the artificially appended actions is removed.

but not approved by the human user. Quantitative detail of each task is shown in Table 4.2.

Although simple computer vision techniques are used for appearance matching, the results show that they suffice when paired with action-analysis in the user-in-the-loop scheme. Figure 4.5 shows the ability to generalize to different test patterns. Here, the user is presented with a web page which links to many different users' homepages. The task is to follow each link, and save each homepage as a pdf file. RecurBot successfully iterates through the links, prompting the user for help when required. More qualitative views of action prediction are shown in Appendix C.

Figures 4.1, 4.5 and 4.7 show difficult examples that can be completed by the system. Figure 4.7 shows the user adding mobile contacts from a desktop spread-

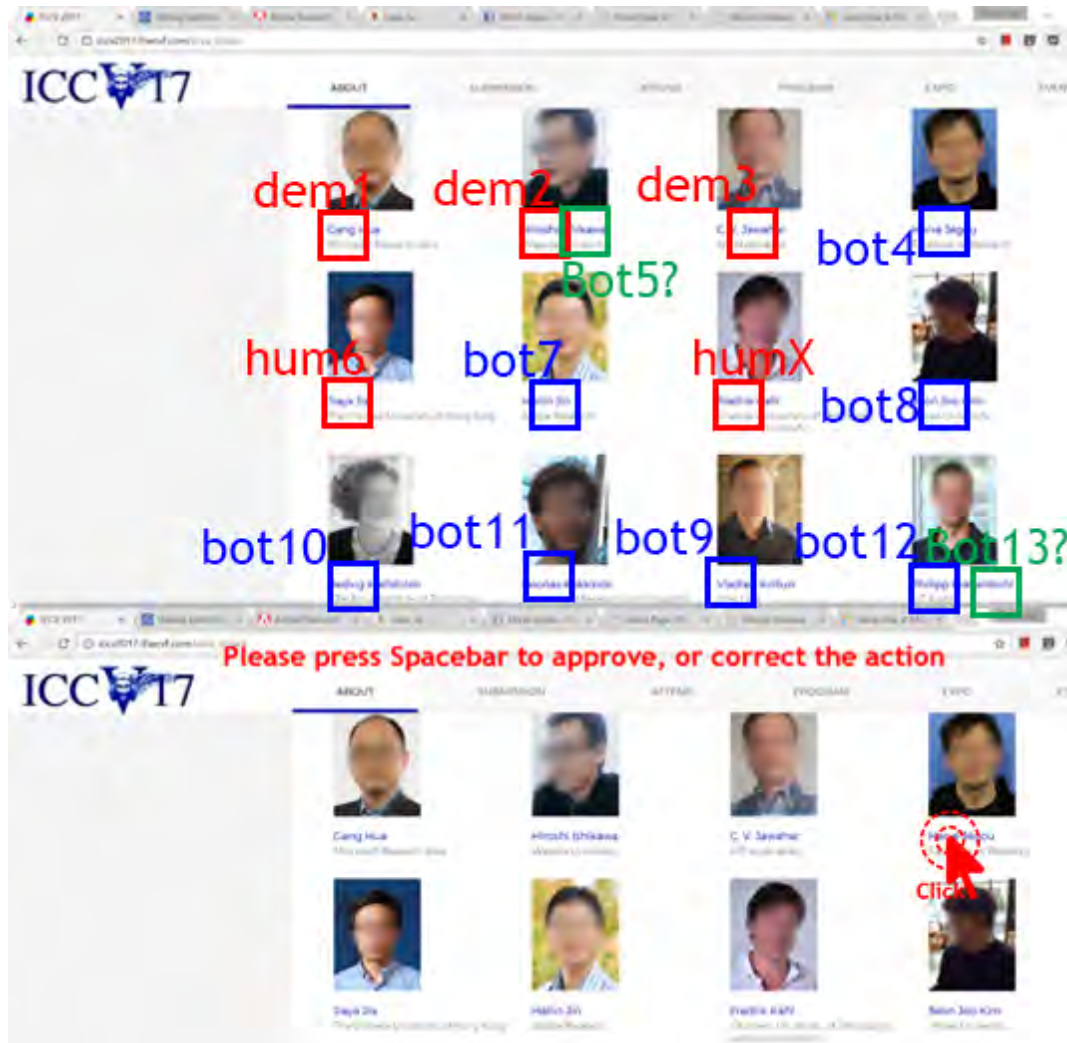


Figure 4.5: Prediction / execution / improvement of future actions. **Top:** An illustrated event history in 14 steps. In this unseen test task the user demonstrated three loops, each starting with a left-click on a hyperlink (shown as red boxes). Boxes in blue show the bot’s subsequent predictions. For each such prediction (e.g., bot 4), the user presses the spacebar to confirm they are happy for the system to proceed automatically. If the prediction’s score falls below the stopping threshold, the system asks the user to approve, correct, or additionally, to terminate — these events are labeled as green boxes followed by a question mark. “hum X” shows when the human terminated the loop. Here, “Bot 5 ?” asked the user to confirm whether the pattern in the box is the next target; the user instead corrected the system by specifying the next correct target as “hum 6”. **Bottom:** The figure shows what the user sees when autocompleting the task. The system visualizes the action it will take next, rendered as a virtual mouse-arrow. It asks the user for approval through large rendered messages, though two-way audio interfaces could be easier for other users to access.

Loop 0	***Missing***	[0] : RClick()	[1] : Click()	[2] : Type (CTRL-C)	***Missing***	[4] : Type (CTRL-V)
Loop 1	[5] : Click()	[6] : RClick()	[7] : Click()	***Missing***	[8] : DoubleClick()	[9] : Type (CTRL-V)
Loop 2	[10] : Click()	[11] : RClick()	[12] : Click()	[13] : Type (CTRL-C)	[14] : DoubleClick()	[15] : Type (CTRL-V)
Loop 3	[16] : Click()	[17] : RClick()	[18] : Click()	[19] : Type (CTRL-C)	[20] : DoubleClick()	[21] : Type (CTRL-V)

Figure 4.6: An example output of the proposed looping action recognition algorithm. The task here is to make a list of filenames from a folder of files. The system outputs a rendering showing discovered loops in rows and matched actions across loops in the same column. Numbers before the actions indicate the order in the input sequence. Here, there are three missing actions and one noisy action (not shown), all of which were detected by the algorithm. The images are extracted from the screenshots at the position each action was performed. It can be seen here that the variation in spatial location of user interaction; images within each column are shifted, or worse. It is this variation that makes it hard to parse a user’s demonstration.

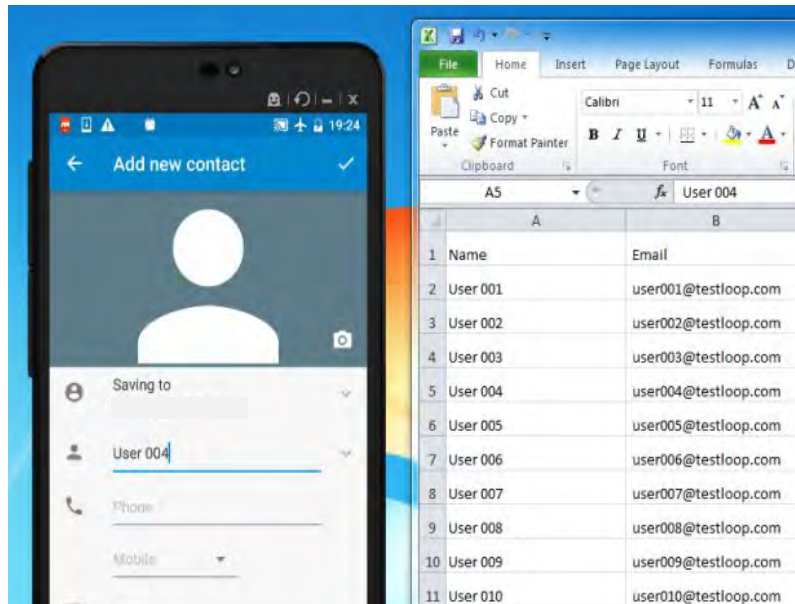


Figure 4.7: Another example that can be easily completed by RecurBot: Adding mobile contacts from a spreadsheet program via an Android remote access program, Vysor [28]

Task	Correct (Auto)	Correct (Approval)	Incorrect (Approval)	Incorrect	Total actions
C.1 Automating SMS sending	64.7%	9.7%	0.3%	0.3%	344
C.2 Adding contacts on phone from spreadsheet via 3rd party app	87.8%	12.2%	0.0%	0.0%	224
C.3 Saving area chairs' homepage as PDFs (icons clicked in regular order)	78.3%	18.8%	0.0%	2.9%	69
C.4 Saving area chairs' homepage as PDFs (icons clicked in random order)	81.2%	14.5%	1.4%	2.9%	69
C.5 Renaming files on Google Drive	76.4%	8.3%	6.3%	9.0%	144
C.6 Deleting specific files on a cluttered desktop	68.0%	28.0%	0.0%	4.0%	25
C.7 Deleting files in folder (smaller icon)	68.0%	12%	0.0%	20%	25
C.8 Creating list of filenames from a folder (files selected in regular order)	82.0%	13.1%	4.9%	0.0%	61
C.9 Creating list of filenames from a folder of remote computer (regular)	87.7%	11.3%	0.0%	1.0%	97
C.10 Creating list of filenames from a folder of remote computer (random)	82.7%	12.4%	3.9%	1.0%	97
C.11 Creating Slides of images from folder of images	78.2%	13.5%	8.3%	0.0%	96
C.12 Zipping every file in a folder	70.6%	17.6%	5.9%	5.9%	34
C.13 Unzipping every file in a folder and renaming the files	75.7%	15.3%	0.0%	9.0%	111
C.14 Taking screenshots of list of websites	79.4%	6.5%	13.0%	1.1%	92
C.15 Taking screenshots of list of websites on mobile	87.5%	12.5%	0.0%	0.0%	106

Table 4.2: Quantitative evaluation of the task completion system on the Looping GUI Automation Dataset. After training each task with 3 demonstrated loops, how many actions were correctly predicted and automatically run (Correct (Auto)). How many action were correctly predicted but the score fell below threshold, so need user approval (Correct (Approval)). If the system make mistake on the prediction and the confident score is below threshold, the system wait for user approval or modification (Incorrect (Approval)). If the system make incorrect prediction with high confident, it is counted as “Incorrect”. Qualitative views of action prediction are shown in Figure 4.5 and more detail on the tasks can be found in appendix C.

sheet program to an Android phone via Vysor remote access [28]. This is difficult because it involves transferring back and forth between a spreadsheet program on a PC and a third party program that allows users to control the mobile device via GUI. This is an example where visual data is crucial; accessibility APIs cannot capture information from both operating systems. Figure 4.1 shows renaming files on Google Drive. Computer-literate users can perform this easily on files on their own computer. However, for novice users or web-based storage, programmatic solutions are not always possible. RecurBot successfully completes the task after watching a user demonstrates using names from a spreadsheet program to rename Google Drive files. More details of these tasks are given Appendix C.

Timings

An average user can complete the 7 steps in one loop of the ‘Google Drive files’ task in 12 seconds, ‘printing people’s homepages’ in 36 seconds (5 steps per loop), and ‘adding mobile contacts remotely’, a 13-step loop, in 37 seconds. The shortest task comprises 13 loops, and takes the average user 2 minutes to complete manually, taking their full attention. In contrast, once the prototype is trained, the user spends only a second to check and approve each action, and about half a second to correct the action if a wrong target is predicted (so 13-20 seconds instead of 2 min., after the one-time cost of recognition). Presently, the looping task recognition part takes most of the processing time. While it averages 18 minutes, this is trivially parallelizable unoptimized code, and runs unattended.

4.6 User Feedback

A user study to evaluate the user feedbacks to the prototype system had been conducted. Ten users, who are white-collar workers, participated in this study. The group consisted of seven females and three males, their ages varying between 24 to 40 years. One of the participants works as a programmer, and the rest have little to no programming skills (the average score of ten participants on the question “I can do programming” is 2.4 on 7-point Likert scale). All of the participants reported

that they used computers in their daily work.

The participants were introduced to the RecurBot system and each of them was asked to complete two looping tasks in rounds: the first round has them complete a task manually (without the system), while they complete the task using the system in the second round. The tasks set to the users were “creating a list of filenames from the files in a folder”, and “creating slides of images from a folder of images”. After completing the tasks, participants were asked to fill out the USE questionnaire [64], which consisted of 30 items measuring the usability of the system in terms of its usefulness, ease of use, ease of learning, and satisfaction. Each item was rated on a 7-point Likert scale, with values from “Strongly Disagree:1” to “Strongly Agree:7”.

The overall averaged score of the system was 5.70, and the breakdown of scores across categories is shown in Table 4.3. Participants agreed in the open ended comments, that the system reduced tedium in completing looping work, is easy to learn and use, and gains users’ trust by asking the users when a prediction is uncertain. Users expressed that they would like the initial analysis to run faster, and for the pattern matching to be more robust (*e.g.*, to occlusions).

Category	Score / 7
Usefulness	5.65 ± 0.36
Ease of use	5.51 ± 0.30
Ease of learning	6.05 ± 0.06
User satisfaction	5.86 ± 0.14
Overall average	5.70 ± 0.32

Table 4.3: Results of the user study on the prototype system. Each question was scored on a Likert scale out of a maximum of 7, and the questions in each of the four categories were averaged for display in this table.

4.7 Conclusions and Future Work

This chapter has shown how to recognize repeated actions in hybrid visual-sniffer data, when a user interacts with a GUI. The system recognizes and predicts further actions, showing improvements over the baselines of existing motif finding algorithms.

For future work, it is challenging to deal with situations where the scrollbar

or navigation buttons are required to access GUI elements required for interaction. Extending the matching algorithm using OCR is also a sound improvement. This would help to find matches in cases where the text is important. Even without these enhancements, RecurBot demonstrates a new problem within action-recognition, and shows strong potential for enabling repetitive GUI-based tasks to be performed quickly, which has special benefit for motor-impaired and hands-free computer users.

This chapter shows that with RecurBot casual users can create automation script by simply demonstrating the task. To improve further on usability and robustness of the system, the need for sniffer program has to be eliminated. The research on learning the user demonstrations from videos will greatly improve usability of the visual-based PbD system by allowing the system to learn from richer sources such as existing instruction videos on YouTube. The next chapter focuses on using machine learning to predict the sniffer-like output from the demonstration video.

Chapter 5

Generating Log-file from Video

In the last two chapters, a visual-based Programming by Demonstration system which generates automation scripts for basic computer tasks from user demonstration screenshots and an algorithm which allows the system to work with more challenging looping tasks are presented. Although both of them are less intrusive compared to traditional PbD systems, *i.e.*, work without using special APIs of Operating Systems or the applications, they still rely on a sniffer or key-cast program to log user demonstrations and save related screenshots.

This leads to the third aspect of the main research question, “Is it possible to remove all instrumented tools needed to observe the demonstration?” This chapter attempt to answer the question by developing a computer vision system which can do the same job as an existing sniffer program. In other words, a system which learn to predict log-file solely from demonstration video is proposed. Due to actions on Computer Desktop environments are obscure and the screenshot images can vary drastically, this study starts with simpler environments: console game environments.

Game and player analysis would be much easier if user interactions were electronically logged and shared with game researchers. Understandably, sniffing software is perceived as invasive and a risk to privacy. To collect player analytics from large populations, the millions of users who already publicly share videos of their game playing are examined. Though labor-intensive, it is a truism that someone with experience of playing a specific game can watch a screen-cast of someone

else playing, and can then infer approximately what buttons and controls the player pressed, and when. This chapter seeks to automatically convert video into such game-play transcripts, or logs.

To capture a player's experience means watching them, probing them with different scenarios, and interviewing them to understand how they felt in the game and afterward. Game analytics and large scale game evaluations are also important, to supplement such careful analysis of individual players. But practical constraints influence the balance between these kinds of depth vs breadth analyses. For example, when studying how players experience different game levels, important insights about play-tuning and interfaces come from both small focus-groups, and global-scale cohorts of players. When available, just the log files themselves provide invaluable insights [97, 86, 85, 87]. Critically, the sampled population size for each study is partly a question of time and cost.

DeepLogger, a Convolutional Neural Network (CNN) which is custom-built to generate player-computer interaction log-files from gameplay videos, is proposed in this chapter. An example of its intended use would be to collect information about a specific game, and how players do better/worse depending on whether they play using a keyboard, game controller, or a particular mobile phone model. A DeepLogger CNN would first be trained by a cooperative game-player. She would record video and key/button logs on a computer with a sniffer program installed. The trained DeepLogger could then be run on each of the $10^3 \dots 10^6$ relevant gameplay videos of that game. The resulting log files could reveal trends and advantage-giving interfaces. Additionally, for interface-researchers who don't own the game's copyright or code, they can avoid the copyright infringement risks that sometimes go along with building emulators [24].

The proposed network is evaluated on a spectrum of quantitative metrics, through different scenarios: training and testing on different players' videos, different levels, and different video encoders. Two classic games from two popular console systems: Tetris (NES) [76] and Mega Man X (SNES) [18] were picked as the test cases, shown in Figure 5.1.



Figure 5.1: Classical titles: Tetris from Nintendo Entertainment System (NES) and Mega Man X from Super Nintendo Entertainment System (SNES) are used in the experiments. Cover art © Nintendo Co., Ltd.

Both obvious and unexpected challenges are set out in the next section. These are challenges faced by both humans and baseline networks, as they try to convert a video into a log file. The rest of the chapter works through the proposed solutions, and evaluation criteria for validating the DeepLogger approach.

5.1 Baselines and Challenges They Face

Initially, the performance of human gamers when asked to estimate what buttons were pressed during the middle frame of a short video clip of someone else’s game-play is illustrated. The challenges faced by both a human and the automated systems, as they attempt to generate interaction logs from videos, are then discussed.

5.1.1 Baseline: Human Performance

In this section, how people fare, when estimating user input logs from gameplay videos, is analyzed. The small user study was conducted online, with 8 gamers recruited to participate. The gamers were first screened, selecting only those gamers who had experience playing Tetris and Mega Man X. In questionnaires, the gamers had to answer 50 questions for each of the two games. Each question displays a short video clip (Figure 5.2), randomly extracted from the gameplay videos. The users were asked to select all the check-boxes for game-control buttons that they

think were pressed in the short video clip.



Figure 5.2: Example of a question in the questionnaires: an example video clip of Tetris gameplay, with check-boxes for a user to indicate what buttons they think were pressed at the middle frame. While this is a demanding and time-consuming task, users were fairly successful when “transcribing” Tetris.

Table 5.1 summarizes the human subjects’ and previews DeepLogger’s performance on the task of predicting input logs from the video. It can be seen that the system performs better than human experts on a harder game like Mega Man X where there are many possible button-combinations, but it performs worse than human experts on Tetris, a game that is visually easier to decipher, with fewer possible button-combinations.

5.1.2 Challenge: Class Imbalance

Class imbalance happens when some controls or button-combinations are used more often than others. See button-combination statistics for Tetris and Mega Man X in Figure 5.3. It is quite common for game logs to have substantial imbalance. For instance, in a side-scrolling game such as Mega Man X, the character progresses by moving steadily right, so other direction-controls are pressed less often.

Substantial class imbalance, as found here, severely impacts the training of machine learning systems, and colors the performance metrics. For example, if one

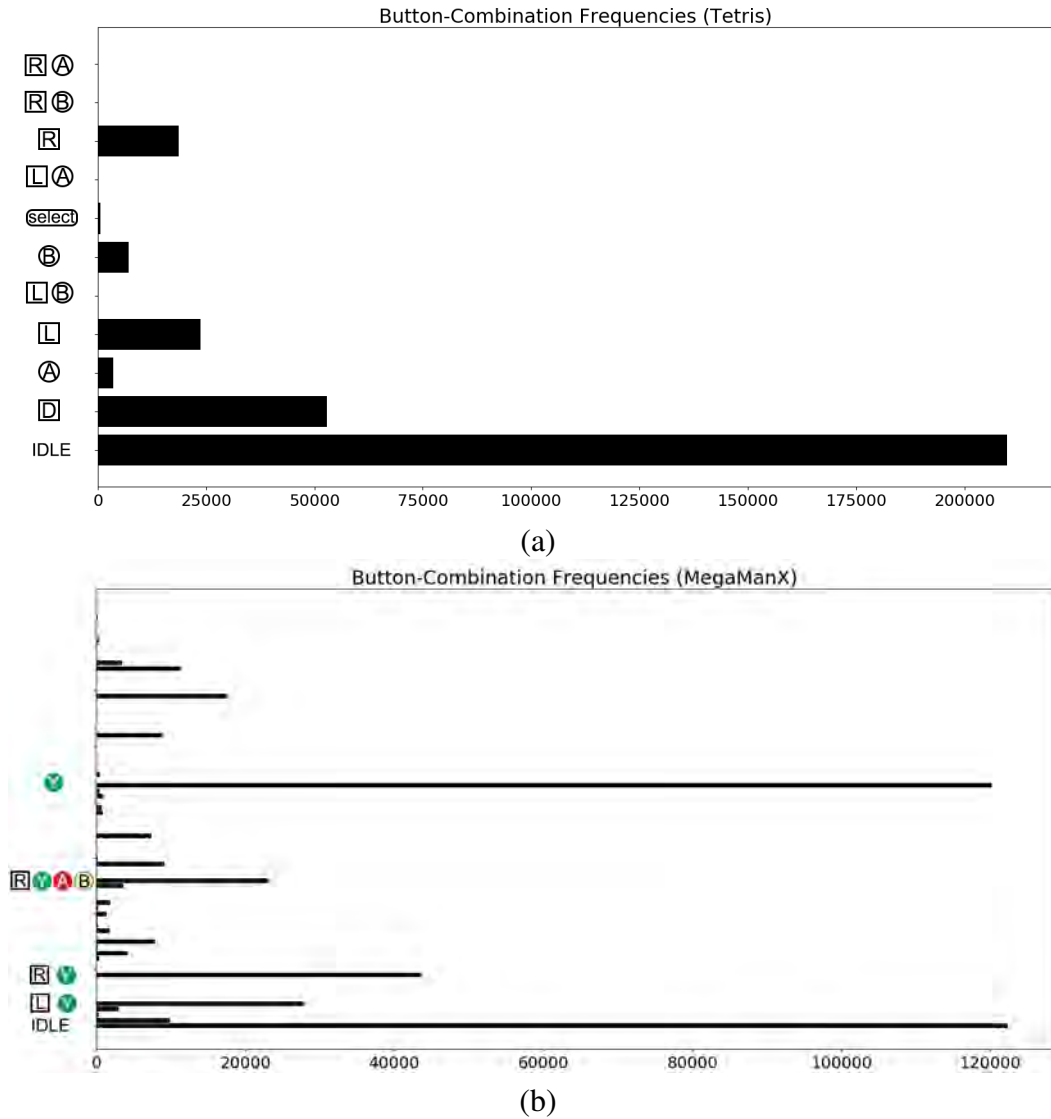


Figure 5.3: Button-combination Frequencies for Tetris (a) and Mega Man X (b). For Tetris, the dominant input is Idle (nothing pressed), followed by three main buttons: Down, Left, and Right. For Mega Man X, the two main classes are Idle and Shoot, followed by the “Right and Shoot” combination. Y-axis represents button-combination (Class) and X-axis represents frequency of the combination.

button-combination occurs 90% of the time, and the training optimization focuses on subset accuracy (no partial credit for imperfect key combinations), then regardless of input, the network will simply always predict that one combination.

In the Architecture Section, the details of training DeepLogger CNN while accounting for imbalanced data is given.

Tetris	Human	DeepLogger
Single-label Accuracy	0.8400±0.00	0.7885
Multi-Label Accuracy	0.8400±0.00	0.7911
F1-score (Example-based)	0.8644±0.00	0.8882
F1-score (Label-based)	0.7794±0.02	0.5691
Mega Man X	Human	DeepLogger
Single-label Accuracy	0.2250±0.18	0.5356
Multi-Label Accuracy	0.4420±0.20	0.7060
F1-score (Example-based)	0.5936±0.19	0.8325
F1-score (Label-based)	0.4722±0.18	0.5363

Table 5.1: The performance of both human experts and the proposed DeepLogger system, on estimating a gamer’s controller inputs (the log) from gameplay videos only: Tetris and Mega Man X. The criteria (rows) are explained in the text, but higher accuracies and F1 scores are better. Humans are better with Tetris videos, while DeepLogger does better with Mega Man X, possibly due to the UI complexity.

5.1.3 Challenge: Multiple Control Buttons Per Record

For each time step, gamers can and tend to press multiple control keys at once. This leads to many possible unique button-combinations. For NES, eight control buttons can generate 8^2 combinations. For SNES, twelve control buttons can generate 12^2 combinations, though “select” and “start” are rarely pressed, leaving 10^2 most of the time.

Notation: Q is the number of input buttons. x is the input of the classifier, so in practice, a short gameplay video clip. Instead of a single output y , Y is the classifier’s output vector, representing the state of all the controller buttons. Y' is the ground truth label associated with x .

The proposed CNN would normally be trained using a standard multi-class loss function H ,

$$H_{Y'}(Y) := - \sum_q^Q Y'_q \log Y_q^*, \quad (5.1)$$

that compares the ground truth output vector Y' to the softmax output tensor of the network Y^* . Y^* is

$$\text{softmax}(Y_q) := \frac{e^{Y_q}}{\sum_q e^{Y_q}}. \quad (5.2)$$

Training the CNN by minimizing the normal multi-class loss (5.1) will not

work here for two reasons. First, if each button is a disjoint class, then the loss will encourage the CNN to treat each button as mutually exclusive of others, discouraging chords. Second, if each button-combination is deemed as a class, the network will not be able to predict classes that weren't present in the training set.

The Losses section in the Architecture description details how the multi-label loss and the proposed multi-label-multi-choice loss are designed to cope with these problems.

5.1.4 Challenge: Many-to-One

This is a problem that was not anticipated, and it may surprise readers who have not analyzed interaction logs. In most games, there are times when pressing two different buttons (or button-combinations) produces the same in-game result. These situations is referred here as functionally equivalent, or as many-to-one situations.

Figure 5.4 depicts the confusion statistics of the two sample games. This challenge is the obstacle that most affects human performance on the video-to-log task. For simplicity, a multi-button-combination is considered as a single class label.

Many-to-one happens when many classes generate one output. For example, when the game is unresponsive, no matter which keys are pressed, the subsequent frames are the same. To account for video examples with more than one associated label, the multi-label-multi-choice loss is proposed. The result is analyzed in section 5.3.6.

5.2 Architecture

Convolutional Neural Networks (CNN's) have proved successful in a wide range of applications, especially on computer vision tasks such as image classification. In this research a new CNN architecture for transcribing user input logs from videos of gameplay is devised, by combining existing CNN components and carefully choosing appropriate training procedures to tackle gameplay video challenges. Moreover, a new loss function, multi-label-multi-choice loss, is proposed to tackle the many-to-one challenges, described in the section 5.1.4. The network is implemented in

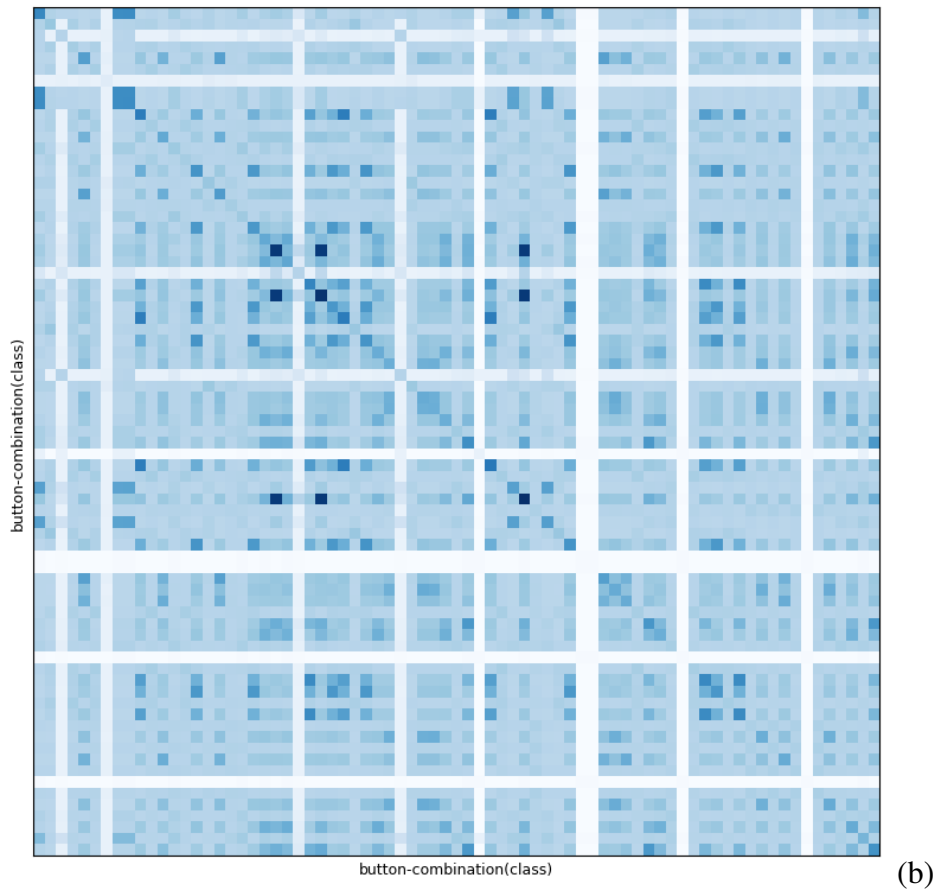
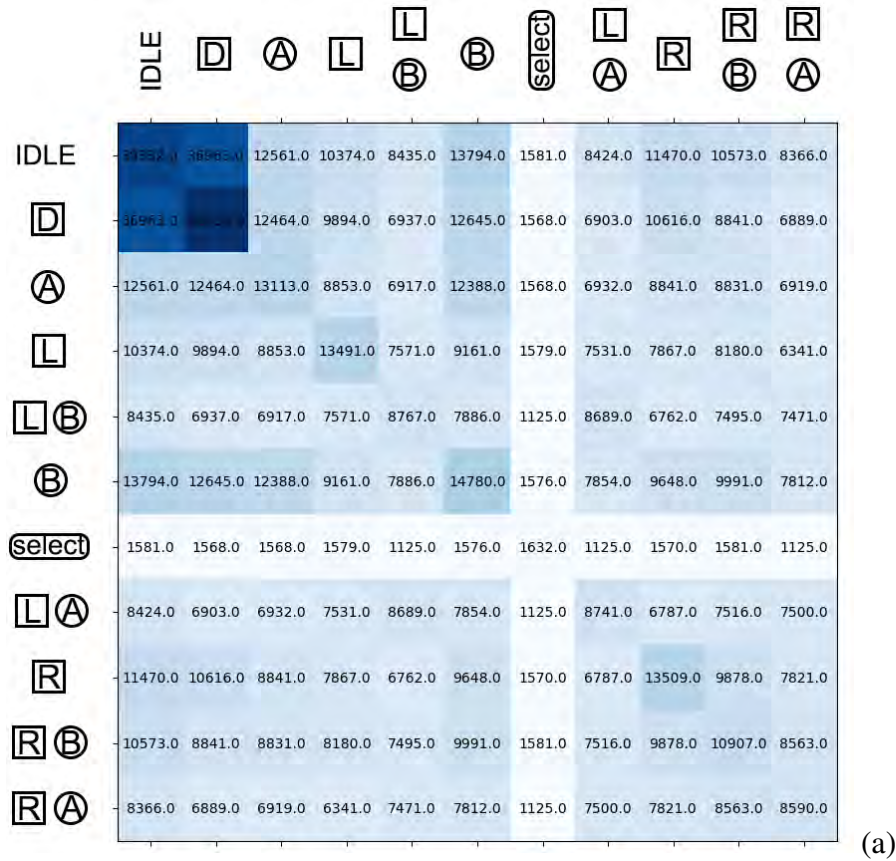


Figure 5.4: Functionally equivalent button-combinations for Tetris, shown in (a), and Mega Man X shown in (b), are visualized through confusion matrices. Each numerical entry indicates how many times the button-combination (of that row) produces the same visual output as another button-combination (column).

Tensorflow [6]. The dataset and the codes can be found at the project page.¹ It is worth to note here that all the network setting: the number of convolution and fully connected layers as well as dropout ratio and learning rate were achieved empirically through a number of experiments which are omitted in this report.

5.2.1 The Network

The DeepLogger network is composed of five 3D convolutional layers, each of which has rectified linear units (ReLU) as activation functions. The network then has five fully connected layers with dropout between layers. Table 5.2 demonstrates the diagram of the proposed CNN network.

Layers	Kernel Dimensions	Number of kernels
Input	input dimensions = $D \times 1 \times W \times H$	
Conv1	$3 \times 5 \times 5$	24
Conv2	$3 \times 5 \times 5$	36
Conv3	$3 \times 5 \times 5$	48
Conv4	$3 \times 3 \times 3$	64
Conv5	$3 \times 3 \times 3$	64
Dense6	output dimensions = 1164	
Dropout	keep = 0.8	
Dense7	output dimensions = 100	
Dropout	keep = 0.8	
Dense8	output dimensions = 50	
Dropout	keep = 0.8	
Dense9	output dimensions = 30	
Dropout	keep = 0.8	
Output	output dimensions = C	

Table 5.2: Diagram of the proposed DeepLogger Network where D is the number of frames per clip, which are 21 frames and 11 frames for Tetris and Mega Man X respectively. $W \times H$ are the image dimensions of the gameplay videos, which are the default screen dimensions of NES, 256×224 for Tetris, and SNES, 586×448 for Mega Man X. C is the number of buttons, with 8 buttons for Tetris and 12 buttons for Mega Man X.

5.2.2 3D convolution layers

In this work, a video is deemed as a stack of temporal 2D images (frames). Before passing each frame to the network, the RGB frame is transformed into a grayscale

¹<http://visual.cs.ucl.ac.uk/pubs/DeepLogger/>

image. While 2D convolution layers look at a whole temporal block at a time which does not consider relations between consecutive frames and the frames order, 3D convolution layers look at smaller temporal blocks. In Table 5.2, “Conv” are 3D convolution layers. The temporal dimension of each layer is set to 3 which means that the network is restricted to consider temporal relations of every three consecutive frames of the given temporal block.

5.2.3 Training

Since user input log data is extremely imbalanced, as shown in Figure 5.3, the network is trained by over-sampling all other classes. The effect is that in each mini-batch, the non-majority classes (button-combinations) have comparable numbers of samples to the majority class.

One data point for Mega Man X and Tetris are set to ± 5 and ± 10 consecutive frames per short clip respectively. The network is trained with the mini-batch scheme using batch size 16 for 50 epochs. The learning rate for both games are fixed to 1E-5.

The imbalanced training procedure is compared with the normal training procedure on both games in the Experiments section.

5.2.4 Losses

As discussed in section 5.1.3, each input button can be pressed at the same time and pressing one button is independent of pressing another button. This problem is framed as the multi-label problem where each class can occur independently. To train a multi-label NN classifier, sigmoid cross entropy loss is used:

$$H_{Y'}(Y) := - \sum Y'_q \log(\sigma(Y_q)), \quad (5.3)$$

where $\sigma(\dagger) = \frac{1}{1+e^{\dagger}}$ is the result of applying the sigmoid function to the output of the network.

However, an extremely challenging characteristic of gameplay video is the many-to-one issues are not yet addressed; and from an inspection of the training data, these situations happen a lot, as shown in figure 5.4.

To tackle the issues, The sigmoid cross entropy loss is modified to consider multiple label choices. When there are more than one class (button-combination) which generates the same visual output as another class, functionally equivalent class, the loss should not penalize those classes even if they are not the ground truth label.

To make the network aware of that, a new ground truth is generated by re-playing the emulator with the ground truth input logs and checking which button-combination is functionally equivalent to the ground truth label. By doing that, the new ground truth label for each example become a set of labels. It is called multi-choice labels.

The modified loss is designed to consider each ground truth label as a set. It looks for the best label from those label set, and computes multi-label loss against that label. The loss is named multi-label-multi-choice loss. The multi-label-multi-choice loss is mathematically defined as

$$H_{\mathbf{Y}'_i}(Y) := \min_{Y'_q \in \mathbf{Y}'_i} (-\sum Y'_q \log(\sigma(Y_q))), \quad (5.4)$$

where \mathbf{Y}'_i is a multi-choice ground truth label of the example i^{th} .

5.3 Experiments and Results

In this section, experiments which were used to validate different components of the network as well as the whole system’s performance are discussed. Table 5.3 demonstrates the key figures from the CNN experiments. DeepLogger is the proposed network; DeepLogger2D is the modified version of the proposed network, using 2D convolutional layers instead of 3D convolutional layers, to validate performance of using 3D filters in the convolutional layers; the VGGNet [88] is a baseline CNN due to its broad acceptance in the computer vision community.

5.3.1 Data

The datasets of both Tetris and Mega Man X were collected using the BizHawk Emulator version 2.1.1. Gamers for each game were recruited to play the game

multiple times. For Mega Man X, one playthrough recorded from one gamer is used as the training data and another recorded playthrough from the same gamer as the test data. For Tetris, the network was trained on twenty gameplay recordings of one level from one gamer; and the test data comprises of gameplay recordings of three different levels and two additional gamers. BizHawk records user control inputs at each time step for each gameplay, in a log-file which can be played back later. This user input log, Y_i , and the associated screenshot image, im_i , are used to construct the training data for the model. For the training data, one data point is a clip of $2n+1$ frames, $x_i = \{im_{i-n}, im_{i-n+1}, \dots, im_i, \dots, im_{i+n-1}, im_{i+n}\}$, where $n = 5$ for Mega Man X and $n = 10$ for Tetris.

The data is split into 80% training set, 5% for the validation set, and 15% test set, which amounts to 316,848 training examples for Tetris and 436,203 training examples for Mega Man X.

5.3.2 Performance Metrics

The performance of the proposed system are benchmarked over a range of different metrics [98]. All metrics and their characteristics are listed in this section.

Single-label Accuracy or Subset Accuracy is a performance metric that captures the fraction of perfectly correct predictions. This performance metric only counts as a correct predictions when Y is identical to the ground truth for all buttons/controls. This metric is similar to the accuracy of the single-label problem.

Multi-label Accuracy evaluates the fraction of correctly classified labels in the multi-label setting. It returns 1 if the predicted set of input buttons is identical to the ground truth, similar to the subset accuracy above, but it returns non-zero numbers if the predicted result is partly correct, so some of the input buttons match. This metric can analyze more fine-grained performance measurements of the classifiers, because it gives partial credit.

Example-based F1-Score measures the harmonic mean of Precision and Recall, where it first evaluates the performance of each example separately, and then returns the average value across the test set. Precision and Recall are complimentary performance measurements to Accuracy, because they take into account class

imbalance, via false positive and false negative statistics.

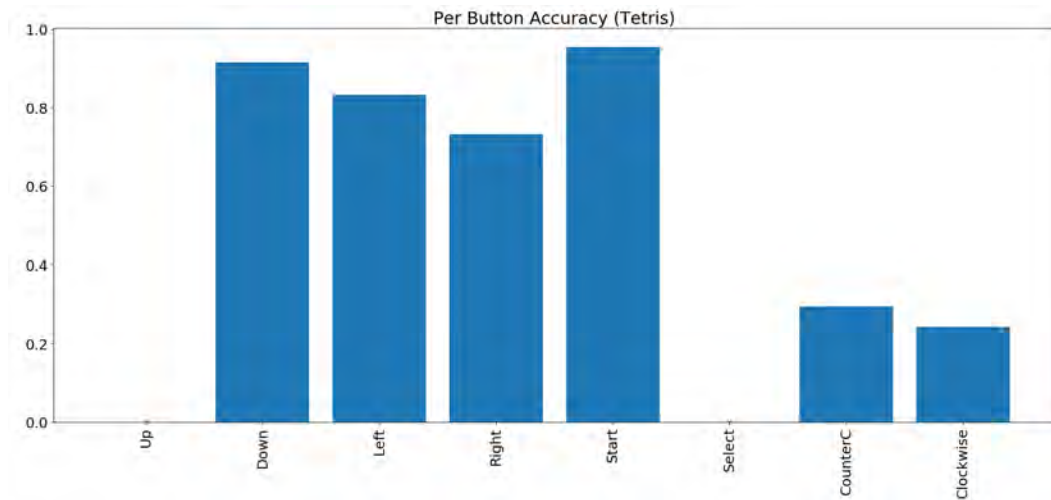
Label-based F1-Score computes an F1-score by first evaluating the performance of each class label (input button) separately, and then returning the average value across all class labels, where every class label is given equal weight (Macro average).

Although the Example-based F1-Score is good for being sensitive to imbalanced data, it can be misled by very large numbers of test examples. Label-based F1-Score, instead, focuses on measuring the performance of predicting individual class labels for each input button.

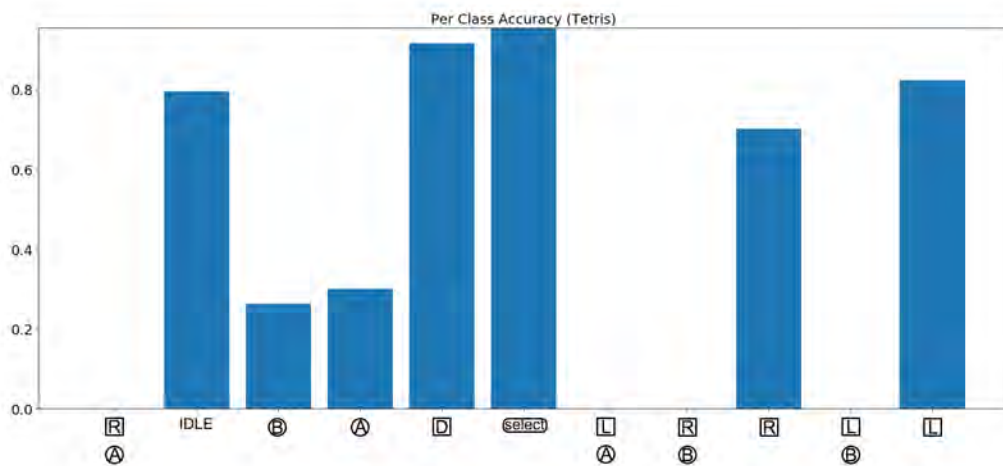
5.3.3 Result: Overall Results

Table 5.3 presents the performance of DeepLogger network on Tetris and Mega Man X. It is clear that DeepLogger has better performance across all four criteria, at least as compared to the 2D alternate version of DeepLogger, and against VGGNet [88], which is a standard architecture used in thousands of computer vision tasks. The numerical results are discussed below, with respect to the different challenges identified at the outset.

Figure 5.5 and Figure 5.6 demonstrate detailed analysis of how the network performs on per button and per class (button-combination) predictions. For Tetris, the per button accuracy chart (a) suggests that the class *rotate block clockwise* and class *rotate block counter-clockwise* are harder for the network to distinguish than the classes which translate the block. This leads to poor performance on the button-combination prediction, where the classes have both direction and rotation buttons being pressed. For Mega Man X, the per button accuracy chart (a) indicates that the network always made mistakes at predicting the button *X* because this button does not map to any action, Also, the network got relatively low scores on the button *L* because *L* and *R* are sliding the inventory window left and right respectively, and there is no visual distinction between the two.

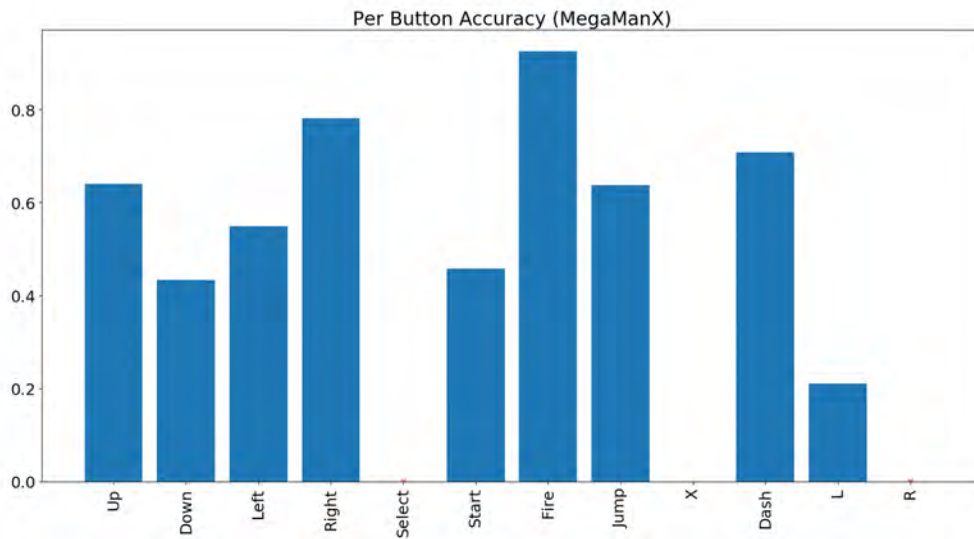


(a)

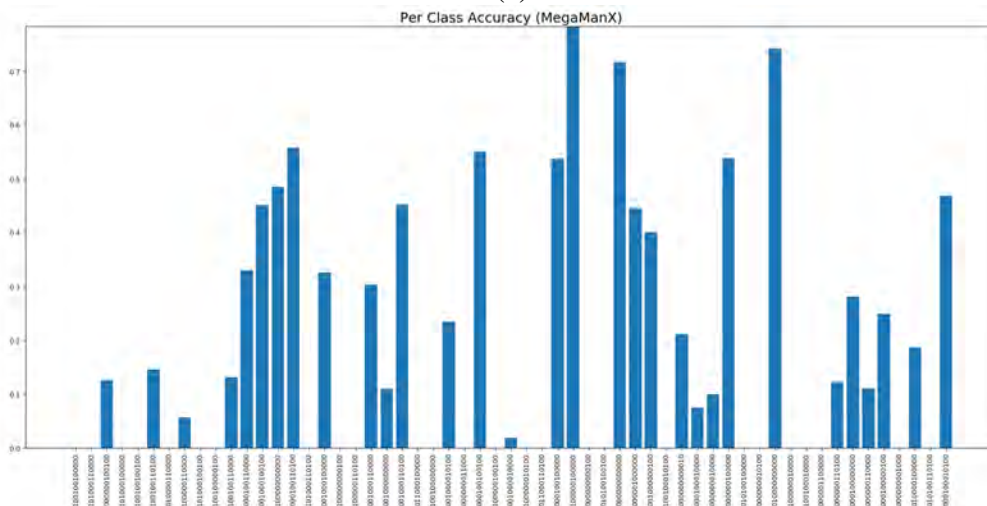


(b)

Figure 5.5: Detailed analysis of DeepLogger network’s performance on Tetris for per button and per class prediction. (a) shows the performance of per button prediction and (b) shows the performance of per class prediction. In both cases, rotation is the hardest to recognize. Small red X’s on the per button accuracy chart indicate there is no ground truth label for the buttons (Up and Select).



(a)



(b)

Figure 5.6: Detailed analysis of DeepLogger network’s performance on Mega Man X for per button and per class prediction. (a) shows the performance of per button prediction and (b) shows the performance of per class prediction. Binary labels in (b) along the horizontal axis represent *pressing* (1) and *not pressing* (0) that game controller button and The Binary codes preserve the button-order from (a). Red X’s on the per button accuracy chart indicate there is no ground truth label for the buttons (R and Select). X-axis represents button-combination (class).

Tetris	DeepLogger (-balanced)	DeepLogger2D (-balanced)	VGGNet [88] (-balanced)
Single-label Accuracy	0.7885 (0.7735)	0.5712 (0.6558)	0.6386 (0.6558)
Multi-Label Accuracy	0.7911 (0.7735)	0.5734 (0.6558)	0.6298 (0.6553)
F1-score (Example-based)	0.8882 (0.8754)	0.7675 (0.7921)	0.7841 (0.7921)
F1-score (Label-based)	0.5691 (0.1865)	0.0874 (0.0000)	0.03907 (0.0000)
Mega Man X	DeepLogger (-balanced)	DeepLogger2D (-balanced)	VGGNet [88] (-balanced)
Subset Accuracy	0.5356 (0.6014)	0.4126 (0.5088)	0.2730 (0.4480)
Multi-Label Accuracy	0.7060 (0.7459)	0.6135 (0.6901)	0.5151 (0.6400)
F1-score (Example-based)	0.8325 (0.8587)	0.7740 (0.8278)	0.7179 (0.7970)
F1-score (Label-based)	0.5363 (0.4352)	0.3578 (0.3149)	0.2866 (0.2616)

Table 5.3: Performance of 3 CNN's: DeepLogger, DeepLogger2D, and VGG on Tetris and Mega Man X. Figures in parentheses indicate performance of the networks when they were trained without over-sampling the non-majority classes.

5.3.4 Result: Class Imbalance

Three different CNN architectures with/without over-sampling the non-majority classes were trained, to balance the number of examples from each class of button-combinations. The results are listed in Table 5.3 where training with the over-sampling scheme is shown first, and results after training without the over-sampling scheme are shown in parentheses.

From the table, it can be seen that for Mega Man X, training with the over-sampling scheme can cause a slight performance drop on three metrics: single-label accuracy, multi-label accuracy, and example-based F1-score. However, the over-sampling scheme always improves the label-based F1-score.

5.3.5 Result: 2D VS 3D Filters

This experiment demonstrates the benefit of using 3D filters for predicting an action from a sequence of images. The performance of two identical network where one uses 3D convolutional layers (DeepLogger) and another that uses 2D convolutional layers (DeepLogger2D) are shown in Table 5.3. Please note that, VGGnet [88] also uses 2D convolutional layers.

For both games 3D convolutions significantly improve the performance of the CNN's across all metrics.

5.3.6 Result: Many-to-One

Figure 5.7 shows the improvement in score when training with multi-label-multi-choice loss, compared to training with ordinary sigmoid cross entropy loss.

The emulator is used here to generate multi-choice labels from a normal single-choice input log. The recorded user input logs are replayed through the emulator, where at each subsequent time step, the script tries all possible button-combinations, looking for ones that generate the same visual output as the original single-label. These functionally equivalent (at least in the short-term) extra combinations are added to the list of multi-choice labels, supplementing the original ground truth.

This process takes substantial time per gameplay log-file, so figure 5.7 only reports the performance when the network is fine-tuned with multi-choice-multi-

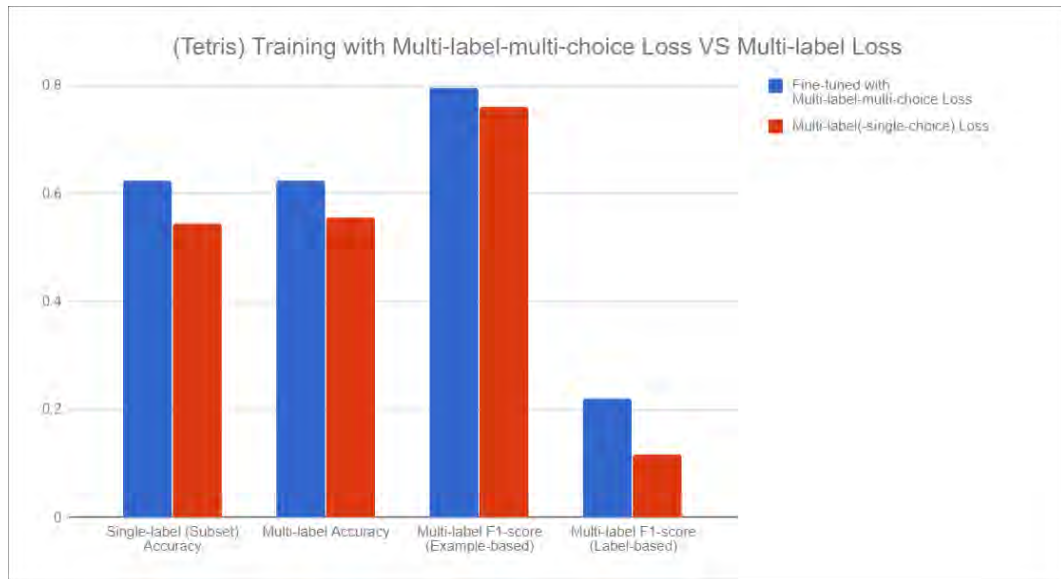


Figure 5.7: Comparing loss functions. This bar-chart compares two DeepLogger networks using four performance metrics. For the red bars, the network was trained with normal multi-label loss, and for the blue bars, the network was first trained with normal multi-label loss on 90% of the training data, and then fine-tuned with multi-label-multi-choice loss on the remaining 10% of the training data. This bar-chart is generated from the Tetris gameplay dataset. Under each measure, fine-tuning with multi-label-multi-choice loss yielded better scores. Y-axis represents prediction accuracy.

label loss on 10% of the training data. This process is trivially parallelizable if needed.

5.3.7 Result: Generalization

Table 5.4 shows experiments where the network is validated on further challenging tasks: training on one person’s gameplay videos and then testing on another person’s, training on one level and then testing on different levels, and training with locally collected video, and then testing on different videos that were encoded by YouTube and downloaded back again, as if scraped.

5.4 Discussion and Conclusion

It has been shown in this chapter that it is possible to extract UI log information from gameplay videos. While the accuracy of the proposed system still has room for improvement, compared with the 100% one would get from using sniffer software,

Tetris	Base	Diff 1 level	Diff 2 levels	Diff Gamers	Diff Encoder
Single-label Accuracy	0.7885	0.7251	0.7988	0.7228	0.7633
Multi-Label Accuracy	0.7911	0.7269	0.8017	0.7235	0.7649
F1-score Example-based	0.8882	0.8489	0.8870	0.8441	0.8743
F1-score Label-based	0.5691	0.4529	0.5504	0.4137	0.6690

Table 5.4: Generalization evaluation. *Base* is the performance of the system when testing on gameplay video from the same level, gamer, and encoder. *Diff 1 level* shows the performance of the system when testing with gameplay videos from a different level than the training data, by 1 level, *Diff 2 levels* tests the same thing, but where the difference is 2 levels. *Diff Gamers* level shows the performance of the system when testing with gameplay videos from different gamers. *Diff Encoder* level shows the performance of the system when testing gameplay videos downloaded from video-sharing site YouTube, where compression and frame-rate changes can occur.

the system could be applied to the millions of users' videos that are generated each month. Preserving users' privacy while collecting broad-scale usage statistics has a value that is presently hard to measure.

The proposed network, along with the learning algorithm that finds functionally equivalent user commands, outperforms the baseline networks in Table 5.3. Presently, it performs on par with what humans can do when focusing their attention, as shown in Table 5.1. It seems from the Mega Man X experiments that human performance is not a ceiling on accuracy or F1 scores (people are slightly worse on Mega Man X), but further machine learning developments are needed to improve on those metrics.

In addition, the secondary benefit of the system has some interesting opportunities. Figure 5.8 visualizes what is learned by the network. The figure shows that the network learns to ignore or at least tolerate the background of the screenshots, and focuses on the main character action. While the estimated logs are the main focus of this research, the neural network's ability to embed "similar" frames near each other in feature space is valuable for other kinds of analytics. For example, a researcher could search, across all recorded gameplay of one or many users, for situations where the game takes a certain turn, or the player performs a certain chain of actions. These would correspond to points in the CNN's abstract feature space, but could be found more reliably than, say, using the absolute time passed from the start of a level - different players move at different speeds. Also, the same game played across different devices may have different logs, but the appearance part of interesting in-game situations will be comparatively consistent. The CNN-embedding gives researchers new opportunities to compare gameplay across game levels and across players. This is similar to the opportunity DeepLogger may present for AI researchers who want to train game-playing AI's, without relying purely on self-play.

For limitations, the current network can only predict user input logs for a game where training data is available. In further work, it would be attractive to learn from many games, to try to generalize across games, increasing the pool of available

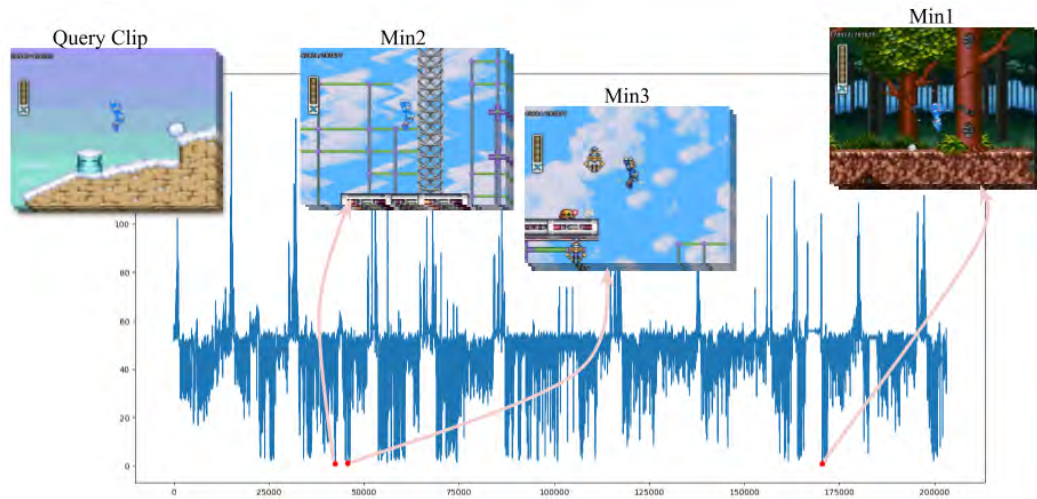


Figure 5.8: The graph visualizes Euclidean distances between embedding features, produced by the "Dense9" layer of the network, for the query clip and other clips from different videos. Min1, Min2, and Min3 clips are examples of the clips which have the smallest distances to the query clip. Y-axis represents distance and X-axis represents time steps in the video of the retrieved gameplay.

training data.

Due to the problem, in this research, is framed as a classification problem, the system is limited to games with discrete control inputs. Further work on transcribing games with continuous inputs is worth exploring because modern games tend to use continuous inputs such as mouse trajectory, gyroscope, or gestures.

Furthermore, it would be interesting to extend the network to work with multi-player games. For 2-player games which each of their two controllers is fixed to the certain part of the screen, such as Super Mario Kart [90], or fixed to certain character appearances, such as Contra [92], the network might need only a little tweak to work. However, the more challenging research topic is to transcribe the games of which the controllers are not fixed to a certain aspect.

Last but not least, in future work an interesting research to be pursued is to build a visual-based PbD system for desktop environments which can learn from demonstrated videos.

Chapter 6

Conclusions

Three key aspects of the main research question have been thoroughly explored in Chapter 3-5. This thesis proposes three separate systems: HILC, RecurBot, and DeepLogger. Each system helps answering the main research question in different angles that “Human in the Loop Machine Learning helps casual users train a computer to perform general GUI task”.

For the first aspect, HILC, the first visual-based Programming by Demonstration, was built to prove that Computer Vision techniques coupled with user feedback scheme can replace the APIs, which is required by existing PbD systems. The study in Chapter 3 has shown that users with no programming skills can use HILC to create scripts for GUI tasks of three different programming concepts. The system bypasses special API restrictions of other existing PbD systems with simple pattern matching techniques and, when necessary, the system asks for help from the user, by posing questions to the user, to clarify confusing patterns.

For the second aspect, RecurBot was developed to solve one weakness of HILC which is the demonstration procedure. Following up research in Chapter 4 attempted to simplify the user demonstration process of the visual-based PbD system in Chapter 3. In this chapter, looping tasks, which are tasks that users have to apply the same sets of actions to several targets to complete the tasks, are the main focus. From the user study, user demonstrations are often inconsistent. The users unintentionally demonstrate extra actions or skip some actions when they were asked to perform the same task more than one time. This chapter established another

evidence that Computer Vision, Data Mining, and User Interaction can solve the problem of sloppy users and help simplify the demonstration process of HILC. In the study, a new motif discovery algorithm, which works better with user inconsistency than existing motif discovery algorithms, was proposed. Furthermore, two datasets and their evaluation frameworks for the looping GUI tasks were proposed for future research.

Lastly, an attempt to go full un-intrusive had been explored. It is noteworthy that the systems in Chapter 3 and Chapter 4 rely on a sniffer program to perceive the interactions. In Chapter 5, Computer Vision approaches which allow computers to perceive interactions between users and application GUIs by watching videos of users perform GUI tasks are studied. Although the study started with more tractable environments, console game environments, the environments shared many problems of GUI in general. From the study, obvious and obscure problems faced by both human and computer were identified. Solutions for each problem were discussed and DeepLogger the network which produce a sniffer-like log file from a demonstration video was proposed.

6.1 Future Research Directions

The research in Chapter 3 and Chapter 4 has introduced a new research direction which will bring together the Computer Vision community and the Human Computer Interaction community to work on a challenging problem, which is “how to allow an end-user who does not have programming expertise to create an automation script without domain-application restrictions”. This is important because, first, non-programming end-users which are a majority of computer users should be able to create their own automation scripts without struggling with a steep learning curve of learning programming concepts. Second, without any prior knowledge, end-users should be able to create automation scripts for any applications and across application as they can perform the tasks manually.

At the end of Chapter 3 and Chapter 4, ideas to improve visual-based PbD systems are discussed. First, the system can be improved by having the ability to

understand states of the application, such as whether the application is still in the processing state or the application has already finished the execution of the previous action. Research on understanding state of applications from GUI manipulation videos has its own challenges and is a very interesting topic to study further for both the Computer Vision community and the Human Computer Interaction community. The more robust system which can deal with an off-screen object by analyzing from the GUI screenshots could be another direction to explore in future research. Understanding behavior of some widgets such as *scroll bar* might lead to the awareness of the system about off-screen object. Additionally, including the Optical Character Recognition ability extends the application of the visual-based PbD to work with a much wider range of tasks. Although there are many attempts to recognize born-digital text [47], this is still far from perfect. On the PbD side, the system still lacks the abilities to learn and synthesize more sophisticated programming concepts such as nested loop tasks and conditional tasks.

In chapter 5, a tool that allows game user researchers to work with large scale data were introduced. These millions of gameplay videos published on the Internet are important for the game user research because these large scale data are almost impossible to be created in a lab setting and the gameplays were collected in-the-Wild. This in-the-Wild data will allow the researchers to study the real-world experienced of the game users. Although it had been demonstrated in the chapter that the proposed network can extract user interaction log information from gameplay videos with comparable performance to human experts, the system still has rooms for improvement. Logging different GUI environments are the whole new challenges to be solved in the future research. Furthermore, CNN-embedding video clip which is the by-product of the network can be applied to a range of application to facilitate scene and action retrievals.

Although the main research question has been answered in Chapter 3-5, there are still more research to be done to achieve a perfect Visual-based Programming by Demonstration system. This thesis reports initial attempts to build a more flexible PbD system by bridging Human Computer Interaction with Machine Vision. The

research in Chapter 3 was presented in ACM IUI 2017 conference and received very positive responses. The authors were invited to submit the extended version of the paper to the TiiS Journal. The research in Chapter 4 was presented in CHI 2018 conference as a late breaking work poster. Lastly, the work in Chapter 5 was accepted to present in CHI PLAY 2018 conference.

Appendix A

Pseudo-codes

Here the details of the algorithms used in the evaluation protocol for our Looping GUI Automation Dataset, in Chapter 4, are presented in Algorithm 3 where Algorithm 4 and 5 are the sub-functions of the Algorithm 3.

Algorithm 3 The Looping GUI Automation Dataset Evaluation Protocol

Input: the ground truth sequence of actions **GT** and the evaluated algorithm **A**.

Output: the list of evaluation results $\mathbf{R} = \{\mathbf{r}_1, \dots, \mathbf{r}_R\}$

```

1:  $\mathbf{R} \leftarrow \emptyset$ 
2:  $\mathbf{A}.\text{observeDemo}(\mathbf{GT}.\text{Demo})$   $\triangleright$  provide the user's demonstration to the
   algorithm
3:  $\text{sharedMasks} \leftarrow \mathbf{GT}.\text{sharedMasks}$ 
4: for each  $\text{loop}$  in  $\mathbf{GT}.\text{automation}$  do
5:   if not  $\text{loop}.\text{terminate}$  then
6:     for each  $\text{stepIndex}$  in  $\text{loop}.\text{steps}.\text{len}$  do
7:        $\text{action}, \text{position} \leftarrow \mathbf{A}.\text{observeAutomation}(\text{loop}[\text{stepIndex}].\text{screenshot})$ 
       $\triangleright$  the algorithm makes a prediction given the screenshot image
8:       if  $\text{loop}[\text{stepIndex}].\text{isSharedMask}$  then
9:          $\text{Mask} \leftarrow \text{sharedMasks}[\text{stepIndex}]$ 
10:      else
11:         $\text{loop}[\text{stepIndex}].\text{gtMask}$ 
12:      if action is "terminate" then
13:        if position in  $\text{Mask}$  then
14:           $\mathbf{r} \leftarrow \underline{\text{WRONG\_TERMINATE}}$ 
15:          if  $\text{loop}[\text{stepIndex}].\text{isSharedMask}$  then
16:             $\text{RemoveRegion}(\text{Mask}, \text{position})$   $\triangleright$  update the shared
            mask
17:          else
18:             $\mathbf{r} \leftarrow \underline{\text{INCORRECT}}$ 
19:          if  $\text{loop}[\text{stepIndex}].\text{isSharedMask}$  then
20:             $\text{RemoveRegion}(\text{Mask})$   $\triangleright$  update the shared mask
21:          else if action is  $\text{loop}[\text{stepIndex}].\text{gtPosition}$  then
22:            if  $\text{loop}[\text{stepIndex}].\text{isSharedMask}$  then
23:               $\mathbf{r} \leftarrow \text{EvaluateUpdateRegion}(\text{Mask}, \text{position})$   $\triangleright$  update the
              shared mask
24:            else
25:               $\mathbf{r} \leftarrow \text{EvaluateRegion}(\text{Mask}, \text{position})$ 
26:             $\mathbf{R}.\text{append}(\mathbf{r})$ 
27:          else
28:             $\text{action}, \text{position} \leftarrow \mathbf{A}.\text{observeAutomation}(\text{loop}[\text{stepIndex}].\text{screenshot})$ 
             $\triangleright$  the algorithm makes a prediction given the screenshot image
29:            if action is "terminate" then
30:               $\mathbf{r} \leftarrow \underline{\text{CORRECT}}$ 
31:            else
32:               $\mathbf{r} \leftarrow \underline{\text{INCORRECT}}$ 
33:             $\mathbf{R}.\text{append}(\mathbf{r})$ 
return  $\mathbf{R}$ 

```

Algorithm 4 Function for evaluating position and updating the mask

Input: the ground truth mask $Mask$ and the evaluated position $position$

Output: the evaluated result r

```

1: function EVALUATEUPDATEREGION( $Mask, position$ )
2:   if  $position$  in  $Mask$  then
3:      $r \leftarrow$  CORRECT
4:     RemoveRegion( $Mask, position$ )
5:   else
6:      $r \leftarrow$  INCORRECT
7:     RemoveRegion( $Mask$ )
return  $r$ 

```

Algorithm 5 Function for updating the mask by removing a region from the mask

Input: the ground truth mask $Mask$ and the position to be removed $position$

```

1: function REMOVEREGION( $Mask, position = \emptyset$ )
2:   if  $position$  is not  $\emptyset$  then
3:     for  $eachRegion$  in  $Mask.regions$  do
4:       if  $position$  in  $eachRegion$  then
5:          $eachRegion.remove()$ 
6:   else
7:      $Mask.regions.random.remove()$ 

```

Appendix B

Datasets

For experimental validation of our algorithm, we informally surveyed colleagues to identify GUI tasks that they found repetitive. From those, we distilled tasks that span different lengths, input modalities, apps and GUI interfaces, complexities, and repetitions. To test our algorithm in aspects of looping action recognition and the complete pipeline, we created two datasets. These fully annotated datasets will be made available with the proposed algorithm.

B.1 Demonstration Dataset

We collected and labeled one dataset of 55 tasks for quantitative evaluation of our motif-finding. These are summarized in Table 4.1 of Chapter 4. They were recorded by asking 7 experienced computer users to perform the first four or so loops of specific repetitive GUI tasks. While working and with their knowledge, they were recorded by sniffer-software that captured both mouse/key events, and screenshots throughout each task. The mouse/key events in this dataset, and all sniffer-events observed at test-time, are converted into actions (*e.g.*, single-click, double-click, click-drag, *etc.*,) using the basic version of Intharah et al.’s system. We then annotated each task’s action-transcript, identifying the boundaries between loops, and tagging the parts of each look that included either extra actions or were missing actions, as compared to the other loops in the task.

One limitation of GUI-task data is that a computer’s exact state (including installed software/hardware/network, layout of open windows, and currently-running

programs) is hard to store or replicate. For this reason, motif-finding generally works with fixed datasets (similar to our 55 tasks). User interactions are harder to replicate, but still possible, as explained next.

B.2 Looping GUI Automation Dataset

For benchmarking the complete pipeline of the algorithm, the Looping Action Recognition and the Action Prediction, we constructed this dataset and its corresponding evaluation protocol. The dataset comprises 15 tasks of fully annotated basic actions, including a user’s demonstration, all subsequence interactions required to complete the task, and the mask for every basic action.

The dataset needs a protocol to drive the evaluation of the sequence of interactions between the evaluated system and the dataset ground truth. The protocol requires the evaluated algorithm to have two functions to interact with the ground truth which are the *observeDemo()* to provide the algorithm a user’s demonstration of the task, and the *observeAutomation()* to progressively provide screenshot of each remaining step to the algorithm and take the prediction result from the algorithm. The user’s demonstration is in the form of a long sequence of basic actions, their locations, and their corresponding screenshot images. For the *observeDemo()*, the algorithm need to figure out what is the next basic action and where to perform, given a screenshot image. The image is shown to the algorithm to the *observeAutomation()* where the algorithm returns the prediction result. The algorithm may output the prediction result *action* and *position* where the *action* $\in \{LeftClick, RightClick, ClickDrag, DoubleClick, Terminate\}$ and the *position* is the most probable target location of the screen to execute the *action*.

The pseudocode of the main evaluation protocol is illustrated in Algorithm 3 in Appendix A. For each basic action, the protocol evaluates the type of basic action, target location of the action, and the termination step. The protocol assumes user interception when the algorithm provides incorrect prediction. The protocol uses the ground truth mask in a basic action to evaluate the algorithm’s predicted location. Due to basic actions of the looping tasks can be roughly categorized into

normal basic actions and the basic actions which are iterators, the masks are also categorized into two types: normal masks and shared masks accordingly. For a basic action which is an iterator, the protocol only evaluates task completion, not the order of completion, so after an algorithm performs an action the mask is updated.

Although the dataset and the protocol were carefully constructed for reproducibility of this paper, and they captured most aspects of the GUI automation, we leave the loading time between actions of the dataset as future work because understanding system states from screenshot images holds its own challenges. Hence this dataset assumes that processing time between basic actions is instant.

We tested our system with this dataset and recorded our algorithm interactions as the bot attempted to complete the task. The recorded information includes the sniffer logs, raw action-lists, data files and spreadsheets employed by the users. All 15 tasks, and our results on them, are detailed in Appendix C.

Appendix C

Looping GUI Automation Evaluation and Datasets

In this section, we show an overview of each of our 15 test tasks. For each task, we give:

- The description of the task
- A representative screen shot from each task
- A grid showing each action performed. Different symbols indicate whether the action was performed by a human or a computer, and if the user had to correct the computer's action.
- The output of our motif-finding algorithm. We show cropped screenshots from the original input video, as grouped together by our algorithm.

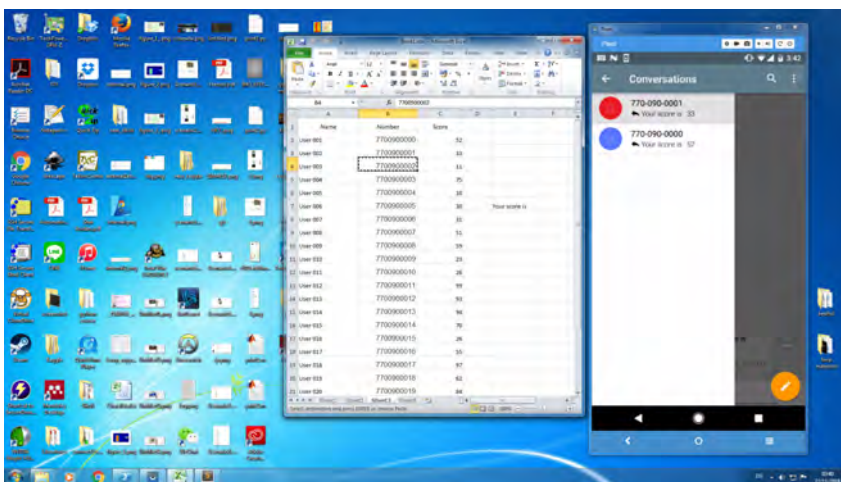
Some tasks are harder for the bot than others. In general, the automatic task completion by the bot reduces the human effort by a factor of 3.

Per-action evaluation charts in the following sections indicate where human effort was required.

C.1 Automating SMS sending

Here, the task is to send a sequence of SMS messages. Given a spreadsheet of telephone numbers and associated test scores, the user is required to send a fixed-format message to each telephone number containing their test score. The mobile telephone is accessed via VPN software.

Illustrative screenshot of task



Per-action evaluation

- Here we show each action performed in the entire task as a symbol, to communicate both the input data and the successes and failures of our algorithm.
- Each row depicts one loop, and each column a group of equivalent actions.
- After the human demonstration ('H'), our system predicts future events. An ideal system would produce a screen of checkmarks ('✓✓'), while crosses (X) indicate failures.

Loop	Actions																
1	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H
2	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H
3	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H
4	✓	✓	✓✓	✓	✓✓	✓	✓	✓✓	✓	✓✓	✓	✓	✓✓	✓	✓✓	✓	✓
5	✓	✓	✓✓	✓	✓✓	✓	✓	✓✓	✓	✓✓	✓	✓	✓✓	✓	✓✓	✓	✓
6	✓	✓	✓✓	✓	✓✓	✓	✓	✓✓	✓	✓✓	✓	✓	✓✓	✓	✓✓	✓	✓
7	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
8	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
9	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
10	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
11	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
12	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
13	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
14	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
15	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
16	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
17	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
18	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
19	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
20	✓✓	✗	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✗✗	✓✓	✓✓	✓✓	✓✓	✓✓
21	✓✓	E															

Legend

- H Actions demonstrated by user
- H_n Extra, unneeded action provided by user
- ∄ Missing action, omitted by user
- ✓ Correct prediction, approved by user
- ✓✓ Correct prediction, automatic mode
- ✗ Incorrect prediction, but score below threshold so user is prompted to confirm
- ✗✗ Incorrect prediction, but score above threshold so user must intervene
- E System correctly predicts end of task

Human demonstrated actions, as grouped by our motif-finding system

- Each row depicts one loop, as performed by the human user.
- Each column contains a group of equivalent actions, as grouped by our motif-finding algorithm.

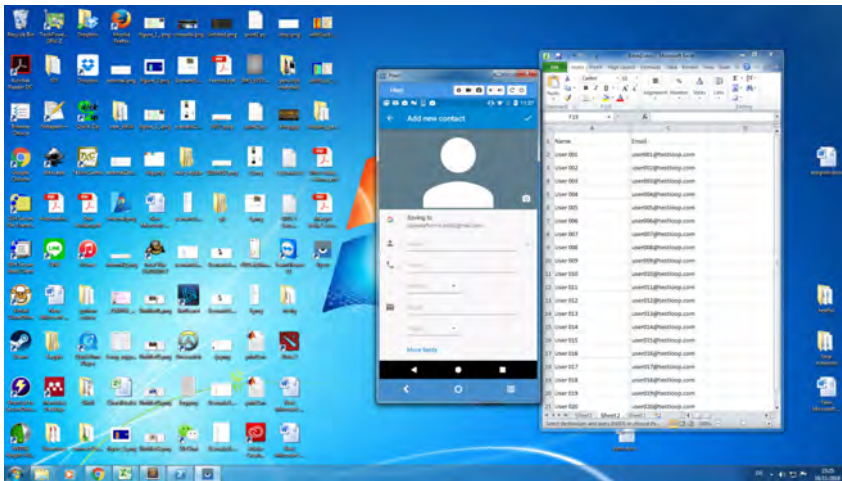
- Where *extra* actions have been identified, these are displayed below the main grid as ‘noise’.

Loop 1	[40] : C11a(C	[41] : C11a(C 770000)	[42] : Type(CTRL-C)	[43] : C11a(C	[44] : Type(CTRL-V)	[45] : C11a(C	[46] : Type(CTRL-V)	[47] : Type(CTRL-C)	[48] : C11a(C	[49] : Type(CTRL-V)	[50] : C11a(C	[51] : Type(CTRL-V)	[52] : C11a(C	[53] : Type(CTRL-C)	[54] : C11a(C	[55] : Type(CTRL-V)	[56] : C11a(C
Loop 2	[37] : C11a(C	[38] : C11a(C 770000)	[39] : Type(CTRL-C)	[40] : C11a(C	[41] : Type(CTRL-V)	[42] : C11a(C	[43] : Type(CTRL-V)	[44] : Type(CTRL-C)	[45] : C11a(C	[46] : Type(CTRL-V)	[47] : C11a(C	[48] : Type(CTRL-V)	[49] : Type(CTRL-C)	[50] : C11a(C	[51] : Type(CTRL-V)	[52] : C11a(C	[53] : Type(CTRL-V)
Loop 2	[34] : C11a(C	[35] : C11a(C 770000)	[36] : Type(CTRL-C)	[37] : C11a(C	[38] : Type(CTRL-V)	[39] : C11a(C	[40] : Type(CTRL-V)	[41] : Type(CTRL-C)	[42] : C11a(C	[43] : Type(CTRL-V)	[44] : C11a(C	[45] : Type(CTRL-V)	[46] : Type(CTRL-C)	[47] : C11a(C	[48] : Type(CTRL-V)	[49] : C11a(C	[50] : Type(CTRL-V)

C.2 Adding contacts on phone from spreadsheet via 3rd party app

Here, the task is to add multiple entries of contact information to a mobile phone. On the computer, there is a database of contact names, each of which has an email address. Using VPN software, each contact is added to the mobile phone via the contacts app.

Illustrative screenshot of task



Per-action evaluation

- Here we show each action performed in the entire task as a symbol, to communicate both the input data and the successes and failures of our algorithm.
- Each row depicts one loop, and each column a group of equivalent actions.
- After the human demonstration ('H'), our system predicts future events. An ideal system would produce a screen of checkmarks ('✓✓'), while crosses (X) indicate failures.

Loop	Actions												
1	H	H	H	H	H	H	H	H	H	H	H	H	H
2	H	H	H	H	H	H	H	H	H	H	H	H	H
3	H	H	H	H	H	H	H	H	H	H	H	H	H
4	✓	✓	✓	✓✓	✓	✓✓	✓	✓	✓✓	✓	✓✓	✓	✓
5	✓	✓	✓	✓✓	✓	✓✓	✓	✓	✓✓	✓	✓✓	✓	✓
6	✓	✓	✓	✓✓	✓	✓✓	✓	✓	✓✓	✓	✓✓	✓	✓
7	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
8	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
9	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
10	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
11	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
12	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
13	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
14	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
15	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
16	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
17	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
18	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
19	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓	✓✓	✓✓	✓✓	✓✓	✓✓
20	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
21	✓✓	✓✓	E										






























Legend

- H Actions demonstrated by user
- H_n Extra, unneeded action provided by user
- ∄ Missing action, omitted by user
- ✓ Correct prediction, approved by user
- ✓✓ Correct prediction, automatic mode
- ✗ Incorrect prediction, but score below threshold so user is prompted to confirm
- ✗✗ Incorrect prediction, but score above threshold so user must intervene
- E System correctly predicts end of task

Human demonstrated actions, as grouped by our motif-finding system

- Each row depicts one loop, as performed by the human user.
- Each column contains a group of equivalent actions, as grouped by our motif-finding algorithm.

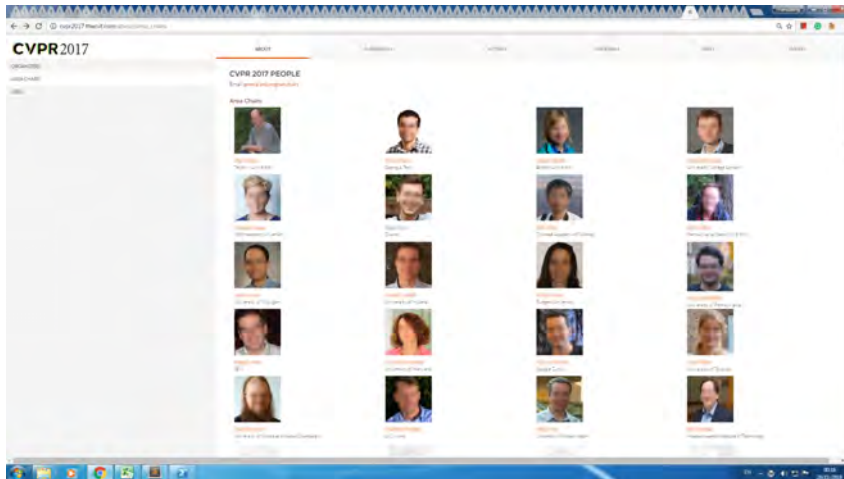
- Where *extra* actions have been identified, these are displayed below the main grid as ‘noise’.

Loop 0	[9] : Click()	[11] : Click()	[12] : Click()	[13] : Type(CTRL-C)	[14] : Click()	[15] : Type(CTRL-V)	[16] : Click()	[17] : Click()	[18] : Type(CTRL-C)	[19] : Click()	[20] : Click()	[21] : Type(CTRL-C)	[22] : Click()	[23] : Type(CTRL-V)	[24] : Click()	[25] : Click()
Loop 1	[13] : Click()	[14] : Click()	[15] : Click()	[16] : Type(CTRL-C)	[17] : Click()	[18] : Type(CTRL-V)	[19] : Click()	[20] : Click()	[21] : Type(CTRL-C)	[22] : Click()	[23] : Type(CTRL-C)	[24] : Click()	[25] : Type(CTRL-V)	[26] : Click()	[27] : Click()	[28] : Click()
Loop 2	[26] : Click()	[27] : Click()	[28] : Click()	[29] : Type(CTRL-C)	[30] : Click()	[31] : Type(CTRL-V)	[32] : Click()	[33] : Click()	[34] : Type(CTRL-C)	[35] : Click()	[36] : Type(CTRL-V)	[37] : Click()	[38] : Click()	[39] : Click()	[40] : Click()	[41] : Click()

C.3 Saving area chairs' homepage as PDFs (icons clicked in regular order)

This task is to save each of the CVPR area chair's home pages as a PDF file. This task is hard, as the images on the website are in a grid format rather than a list, and because the task involves some one-time settings changes such as setting the printer to save as pdf. In this version of the task, the user chose to select each item from the grid in turn, starting with the image in the top-left before proceeding across the top row. This gives a *low variance* in the linear regression term (Equation (11) in the main paper), meaning that the spatial prior is used to predict future events based on previous click locations.

Illustrative screenshot of task



Per-action evaluation

- Here we show each action performed in the entire task as a symbol, to communicate both the input data and the successes and failures of our algorithm.
- Each row depicts one loop, and each column a group of equivalent actions.
- After the human demonstration ('H'), our system predicts future events. An ideal system would produce a screen of checkmarks ('✓✓'), while crosses (✗) indicate failures.

C.3. Saving area chairs' homepage as PDFs (icons clicked in regular order) 134

Loop	Actions				
1	H	H	H _n H _n H	H	H
2	H	H	H	H	H
3	H	H	H	H	H
4	✓	✓✓	✓✓	✓	✓
5	✗✗	✓✓	✓✓	✓	✓
6	✓	✓✓	✓✓	✓	✓
7	✓	✓✓	✓✓	✓✓	✓✓
8	✓	✓✓	✓✓	✓✓	✓✓
9	✓	✓✓	✓✓	✓✓	✓✓
10	✓✓	✓✓	✓✓	✓✓	✓✓
11	✓✓	✓✓	✓✓	✓✓	✓✓
12	✓✓	✓✓	✓✓	✓	✓✓
13	✓✓	✓✓	✓✓	✓✓	✓✓
14	✓✓	✓✓	✓✓	✓✓	✓✓
15	✓✓	✓✓	✓✓	✓✓	✓✓
16	✓✓	✓✓	✓✓	✓✓	✓✓
17	✓✓	✓✓	✓✓	✓✓	✓✓
18	✓	✓✓	✓✓	✓✓	✓✓
19	✓✓	✓✓	✓✓	✓✓	✓✓
20	✓✓	✓✓	✓✓	✓✓	✓✓
21	✗✗				










Legend

- H Actions demonstrated by user
- H_n Extra, unneeded action provided by user
- ∅ Missing action, omitted by user
- ✓ Correct prediction, approved by user
- ✓✓ Correct prediction, automatic mode
- ✗ Incorrect prediction, but score below threshold so user is prompted to confirm
- ✗✗ Incorrect prediction, but score above threshold so user must intervene
- E System correctly predicts end of task



Human demonstrated actions, as grouped by our motif-finding system

- Each row depicts one loop, as performed by the human user.
- Each column contains a group of equivalent actions, as grouped by our motif-finding algorithm.
- Where *extra* actions have been identified, these are displayed below the main grid as 'noise'.

C.3. Saving area chairs' homepage as PDFs (icons clicked in regular order) 135

Loop 0	[0] : Click()	[1] : Type(CTRL-P)	[4] : Type(enter)	[5] : Click()	[6] : Click()
Loop 1	[7] : Click()	[8] : Type(CTRL-P)	[9] : Type(enter)	[10] : Click()	[11] : Click()
Loop 2	[12] : Click()	[13] : Type(CTRL-P)	[14] : Type(enter)	[15] : Click()	[16] : Click()

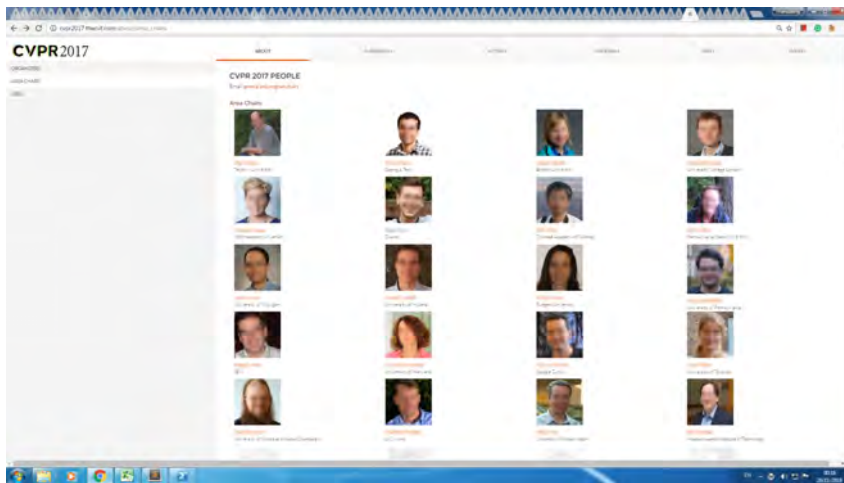
Noise:

[2] : Click()	[3] : Click()
--	--

C.4 Saving area chairs' homepage as PDF's (icons clicked in random order)

This task is equivalent to that described in Section C.3, but this time the user clicks on the images in a *random* order. Now, the variance in Equation (11) of the main paper is high. This means that the spatial prior is not used; instead, the algorithm uses the method described in Section 4 to correctly select new locations.

Illustrative screenshot of task



Per-action evaluation

- Here we show each action performed in the entire task as a symbol, to communicate both the input data and the successes and failures of our algorithm.
- Each row depicts one loop, and each column a group of equivalent actions.
- After the human demonstration ('H'), our system predicts future events. An ideal system would produce a screen of checkmarks ('✓✓'), while crosses (✗) indicate failures.

C.4. Saving area chairs' homepage as PDF's (icons clicked in random order) 137

Loop	Actions				
1	H	H	H _n H _n H	H	H
2	H	H	H	H	H
3	H	H	H	H	H
4	✓	✓✓	✓✓	✓	✓
5	✓	✓✓	✓✓	✓	✓
6	✓	✓✓	✓✓	✓	✓
7	✓✓	✓✓	✓✓	✓✓	✓✓
8	✓✓	✓✓	✓✓	✓✓	✓✓
9	✓✓	✓✓	✓✓	✓✓	✓✓
10	✓✓	✓✓	✓✓	✓✓	✓✓
11	✓✓	✓✓	✓✓	✓✓	✓✓
12	✓✓	✓✓	✓✓	✓✓	✓✓
13	✓✓	✓✓	✓✓	✓✓	✓✓
14	✓✓	✓✓	✓✓	✓✓	✓✓
15	✓✓	✓✓	✓✓	✓✓	✓
16	✓✓	✓✓	✓✓	✓✓	✓✓
17	✓✓	✓✓	✓✓	✓✓	✓✓
18	✓✓	✓✓	✓✓	✓✓	✓✓
19	XX	✓✓	✓✓	✓✓	✓✓
20	X	✓✓	✓✓	✓	✓✓
21	XX				










Legend

- H Actions demonstrated by user
- H_n Extra, unneeded action provided by user
- ∅ Missing action, omitted by user
- ✓ Correct prediction, approved by user
- ✓✓ Correct prediction, automatic mode
- X Incorrect prediction, but score below threshold so user is prompted to confirm
- XX Incorrect prediction, but score above threshold so user must intervene
- E System correctly predicts end of task



Human demonstrated actions, as grouped by our motif-finding system

- Each row depicts one loop, as performed by the human user.
- Each column contains a group of equivalent actions, as grouped by our motif-finding algorithm.
- Where *extra* actions have been identified, these are displayed below the main grid as 'noise'.

C.4. Saving area chairs' homepage as PDF's (icons clicked in random order) 138

Loop 0	[0] : Click()	[1] : Type(CTRL-P)	[4] : Type(enter)	[5] : Click()	[6] : Click()
Loop 1	[7] : Click()	[8] : Type(CTRL-P)	[9] : Type(enter)	[10] : Click()	[11] : Click()
Loop 2	[12] : Click()	[13] : Type(CTRL-P)	[14] : Type(enter)	[15] : Click()	[16] : Click()

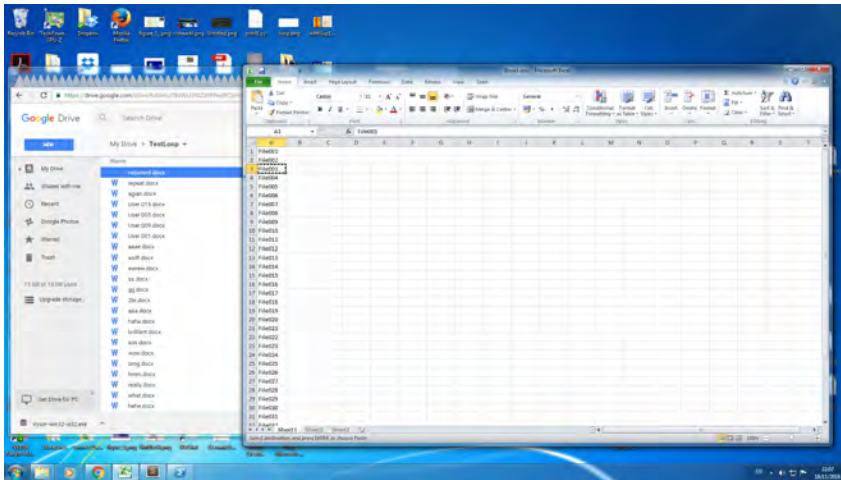
Noise:

[2] : Click()	[3] : Click()
--	--

C.5 Renaming files on Google Drive

The task here is to rename each file in a folder on Google Drive, to match a list of pre-defined names in a spreadsheet application.

Illustrative screenshot of task



Per-action evaluation

- Here we show each action performed in the entire task as a symbol, to communicate both the input data and the successes and failures of our algorithm.
- Each row depicts one loop, and each column a group of equivalent actions.
- After the human demonstration ('H'), our system predicts future events. An ideal system would produce a screen of checkmarks ('✓✓'), while crosses (X) indicate failures.













Loop	Actions						
1	H	H	H	H	H	H	H
2	H	H	H	H	H	H	H
3	H	H	H	H	H	H	H
4	XX	✓	✓✓	✓	✓	✓✓	✓✓
5	✓	✓	✓✓	✓	x	✓✓	✓✓
6	✓	✓	✓✓	✓	x	✓✓	✓✓
7	✓	✓	✓✓	✓✓	x	✓✓	✓✓
8	✓✓	✓✓	✓✓	✓✓	x	✓✓	✓✓
9	✓✓	✓✓	✓✓	✓✓	x	✓✓	✓✓
10	✓✓	✓✓	✓✓	✓✓	x	✓✓	✓✓
11	✓✓	✓✓	✓✓	✓✓	x	✓✓	✓✓
12	✓✓	✓✓	✓✓	✓✓	XX	✓✓	✓✓
13	✓✓	✓✓	✓✓	✓✓	x	✓✓	✓✓
14	✓✓	✓✓	✓✓	✓✓	XX	✓✓	✓✓
15	✓✓	✓✓	✓✓	✓✓	XX	✓✓	✓✓
16	✓✓	✓✓	✓✓	✓✓	XX	✓✓	✓✓
17	✓✓	✓✓	✓✓	✓✓	XX	✓✓	✓✓
18	✓✓	✓✓	✓✓	✓✓	XX	✓✓	✓✓
19	✓✓	✓✓	✓✓	✓✓	XX	✓✓	✓✓
20	✓✓	✓✓	✓✓	✓✓	XX	✓✓	✓✓
21	✓✓	✓✓	✓✓	✓✓	XX	✓✓	✓✓
22	✓✓	✓✓	✓✓	✓	XX	✓✓	✓✓
23	✓✓	✓✓	✓✓	x	XX	✓✓	✓✓
24	✓✓	✓✓	✓✓	XX			

Legend

- H Actions demonstrated by user
- H_n Extra, unneeded action provided by user
- ⊘ Missing action, omitted by user
- ✓ Correct prediction, approved by user
- ✓✓ Correct prediction, automatic mode
- x Incorrect prediction, but score below threshold so user is prompted to confirm
- XX Incorrect prediction, but score above threshold so user must intervene
- E System correctly predicts end of task

Human demonstrated actions, as grouped by our motif-finding system

- Each row depicts one loop, as performed by the human user.
- Each column contains a group of equivalent actions, as grouped by our motif-finding algorithm.
- Where *extra* actions have been identified, these are displayed below the main grid as ‘noise’.

Loop 0	[0] : Click()	[1] : Click()	[2] : Type(CTRL-C)	[3] : RClick()	[4] : Click()	[5] : Type(CTRL-V)	[6] : Type(enter)
Loop 1	[7] : Click()	[8] : Click()	[9] : Type(CTRL-C)	[10] : RClick()	[11] : Click()	[12] : Type(CTRL-V)	[13] : Type(enter)
Loop 2	[14] : Click()	[15] : Click()	[16] : Type(CTRL-C)	[17] : RClick()	[18] : Click()	[19] : Type(CTRL-V)	[20] : Type(enter)

C.6 Deleting specific files on a cluttered desktop

This task is to delete all the files of a certain type from a cluttered desktop.

Illustrative screenshot of task



Per-action evaluation

- Here we show each action performed in the entire task as a symbol, to communicate both the input data and the successes and failures of our algorithm.
- Each row depicts one loop, and each column a group of equivalent actions.
- After the human demonstration ('H'), our system predicts future events. An ideal system would produce a screen of checkmarks ('✓✓'), while crosses (X) indicate failures.







Loop	Actions		
1	H _n H	H	H
2	H	H	H
3	H _n H	H	H
4	✓	✓	✓✓
5	✓	✓	✓✓
6	✓	✓	✓✓
7	✓✓	✓✓	✓✓
8	✓✓	✓✓	✓✓
9	✓✓	✓✓	✓✓
10	✓✓	✓✓	✓✓
11	✓	✓✓	✓✓
12	XX		

Legend



- H Actions demonstrated by user
- H_n Extra, unneeded action provided by user
- ∅ Missing action, omitted by user
- ✓ Correct prediction, approved by user
- ✓✓ Correct prediction, automatic mode
- X Incorrect prediction, but score below threshold so user is prompted to confirm
- XX Incorrect prediction, but score above threshold so user must intervene
- E System correctly predicts end of task

Human demonstrated actions, as grouped by our motif-finding system

- Each row depicts one loop, as performed by the human user.
- Each column contains a group of equivalent actions, as grouped by our motif-finding algorithm.
- Where *extra* actions have been identified, these are displayed below the main grid as ‘noise’.

Loop 0	[1] : RClick()	[2] : Click()	[3] : Type(enter)
Loop 1	[4] : RClick()	[5] : Click()	[6] : Type(enter)
Loop 2	[8] : RClick()	[9] : Click()	[10] : Type(enter)

Noise:

[0] : Click()	[7] : Click()
--	--

C.7 Deleting files in folder (smaller icon)

Like the task in Section C.6, here we aim to delete files of a certain type. However, now the files are small icons in a folder.

Illustrative screenshot of task



Per-action evaluation

- Here we show each action performed in the entire task as a symbol, to communicate both the input data and the successes and failures of our algorithm.
- Each row depicts one loop, and each column a group of equivalent actions.
- After the human demonstration ('H'), our system predicts future events. An ideal system would produce a screen of checkmarks ('✓✓'), while crosses (X) indicate failures.




Loop	Actions		
1	H	H	H
2	H	H	H
3	H	H	H
4	✓	✓✓	✓✓
5	✓	✓✓	✓✓
6	✓	✓✓	✓✓
7	XX	✓✓	✓✓
8	✓✓	✓✓	✓✓
9	XX	✓✓	✓✓
10	XX	✓✓	✓✓
11	XX	✓✓	✓✓
12	XX		

Legend

- H Actions demonstrated by user
- H_n Extra, unneeded action provided by user
- ∅ Missing action, omitted by user
- ✓ Correct prediction, approved by user
- ✓✓ Correct prediction, automatic mode
- X Incorrect prediction, but score below threshold so user is prompted to confirm
- XX Incorrect prediction, but score above threshold so user must intervene
- E System correctly predicts end of task

Human demonstrated actions, as grouped by our motif-finding system

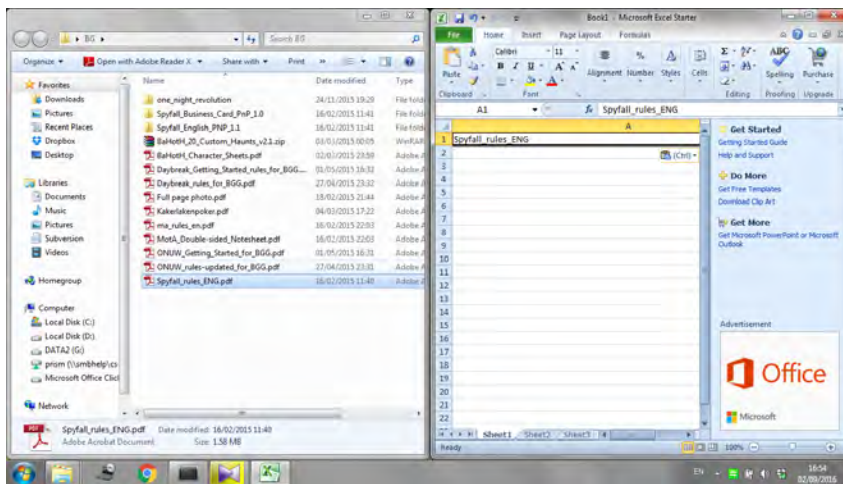
- Each row depicts one loop, as performed by the human user.
- Each column contains a group of equivalent actions, as grouped by our motif-finding algorithm.
- Where *extra* actions have been identified, these are displayed below the main grid as ‘noise’.

Loop 0	[0] : Click()	[1] : Type(delete)	[2] : Type(enter)
Loop 1	[3] : Click()	[4] : Type(delete)	[5] : Type(enter)
Loop 2	[6] : Click()	[7] : Type(delete)	[8] : Type(enter)

C.8 Creating list of filenames from a folder (files selected in regular order)

The aim of this task is to iterate over each file in a folder, and create a list of names of each of the files. This list is entered into a spreadsheet program.

Illustrative screenshot of task



Per-action evaluation

- Here we show each action performed in the entire task as a symbol, to communicate both the input data and the successes and failures of our algorithm.
- Each row depicts one loop, and each column a group of equivalent actions.
- After the human demonstration ('H'), our system predicts future events. An ideal system would produce a screen of checkmarks ('✓'), while crosses (X) indicate failures.












Loop	Actions					
1	H	H	H	H	∅	H
2	H	H	H	H	H	H
3	H	H	H	H	H	H
4	H	H	H	H	H	H
5	✓	✓✓	✓✓	✓	✓	✓✓
6	✓	✓✓	✓✓	✗	✓	✓✓
7	✓	✓✓	✓✓	✓	✓	✓✓
8	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
9	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
10	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
11	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
12	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
13	✗	✓✓	✓✓	✓✓	✓✓	✓✓
14	✗	✓✓	✓✓	✓✓	✓✓	✓✓
15	E					

Legend

- H Actions demonstrated by user
- H_n Extra, unneeded action provided by user
- ∅ Missing action, omitted by user
- ✓ Correct prediction, approved by user
- ✓✓ Correct prediction, automatic mode
- ✗ Incorrect prediction, but score below threshold so user is prompted to confirm
- ✗✗ Incorrect prediction, but score above threshold so user must intervene
- E System correctly predicts end of task

Human demonstrated actions, as grouped by our motif-finding system

- Each row depicts one loop, as performed by the human user.
- Each column contains a group of equivalent actions, as grouped by our motif-finding algorithm.
- Where *extra* actions have been identified, these are displayed below the main grid as ‘noise’.

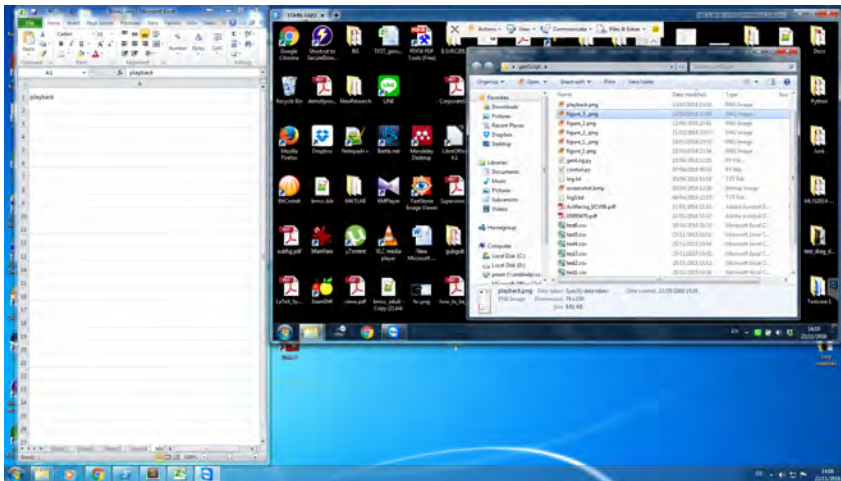
Loop 0	[0] : Click()	[1] : Type(f2)	[2] : Type(CTRL-C)	[3] : Click()	***Missing***	[4] : Type(CTRL-V)
Loop 1	[5] : Click()	[6] : Type(f2)	[7] : Type(CTRL-C)	[8] : Click()	[9] : Click()	[10] : Type(CTRL-V)
Loop 2	[11] : Click()	[12] : Type(f2)	[13] : Type(CTRL-C)	[14] : Click()	[15] : Click()	[16] : Type(CTRL-V)
Loop 3	[17] : Click()	[18] : Type(f2)	[19] : Type(CTRL-C)	[20] : Click()	[21] : Click()	[22] : Type(CTRL-V)

C.9 Creating list of filenames from a folder of remote computer (regular ordered)

This task is equivalent to the task in Section C.8. However, this time we access the files on a remote computer, rather than on the local machine.

This task demonstrates our ability to operate across networks, as we use the visual on-screen data as input.

Illustrative screenshot of task



Per-action evaluation

- Here we show each action performed in the entire task as a symbol, to communicate both the input data and the successes and failures of our algorithm.
- Each row depicts one loop, and each column a group of equivalent actions.
- After the human demonstration ('H'), our system predicts future events. An ideal system would produce a screen of checkmarks ('✓✓'), while crosses (X) indicate failures.

C.9. Creating list of filenames from a folder of remote computer (regular ordered)149













Loop	Actions					
1	H	H	H	H	H	H
2	H	H	H	H	H	H
3	H	H	H	H	H	H
4	✓	✓✓	✓✓	✓	✓	✓✓
5	✓	✓✓	✓✓	✓	✓	✓✓
6	✓	✓✓	✓✓	✓	✓	✓✓
7	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
8	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
9	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
10	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
11	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
12	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
13	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
14	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
15	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
16	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
17	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
18	✓	✓✓	✓✓	✓✓	✓✓	✓✓
19	✓	✓✓	✓✓	✓✓	✓✓	✓✓
20	XX					

Legend

- H Actions demonstrated by user
- H_n Extra, unneeded action provided by user
- ∅ Missing action, omitted by user
- ✓ Correct prediction, approved by user
- ✓✓ Correct prediction, automatic mode
- ✗ Incorrect prediction, but score below threshold so user is prompted to confirm
- ✗✗ Incorrect prediction, but score above threshold so user must intervene
- E System correctly predicts end of task

Human demonstrated actions, as grouped by our motif-finding system

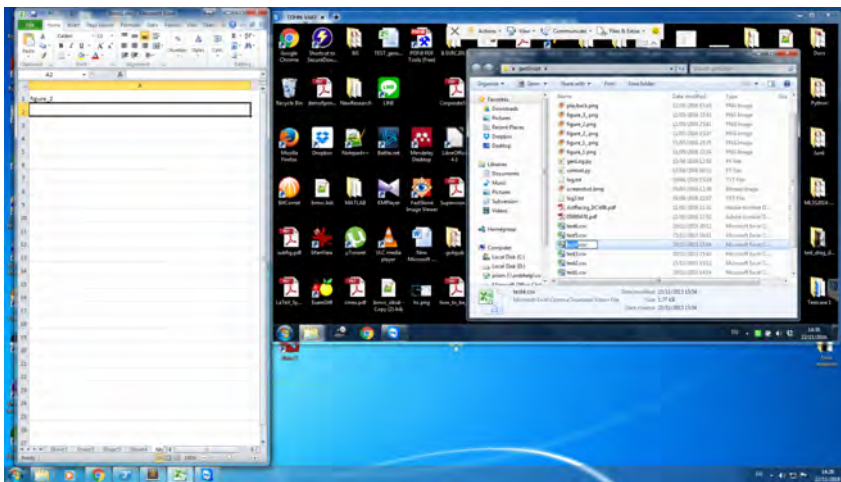
- Each row depicts one loop, as performed by the human user.
- Each column contains a group of equivalent actions, as grouped by our motif-finding algorithm.
- Where *extra* actions have been identified, these are displayed below the main grid as ‘noise’.

Loop 0	[0] : Click() 	[1] : Type(f2)	[2] : Type(CTRL-C)	[3] : Click()	[4] : Click()	[5] : Type(CTRL-V)
Loop 1	[6] : Click() 	[7] : Type(f2)	[8] : Type(CTRL-C)	[9] : Click()	[10] : Click()	[11] : Type(CTRL-V)
Loop 2	[12] : Click() 	[13] : Type(f2)	[14] : Type(CTRL-C)	[15] : Click()	[16] : Click()	[17] : Type(CTRL-V)

C.10 Creating list of filenames from a folder of remote computer (random ordered)

This task is equivalent to that in Section C.9. but in this example the files are chosen by the user in a *random* (rather than *regular*) ordering.

Illustrative screenshot of task



Per-action evaluation

- Here we show each action performed in the entire task as a symbol, to communicate both the input data and the successes and failures of our algorithm.
- Each row depicts one loop, and each column a group of equivalent actions.
- After the human demonstration ('H'), our system predicts future events. An ideal system would produce a screen of checkmarks ('✓'), while crosses (X) indicate failures.

C.10. Creating list of filenames from a folder of remote computer (random ordered)151

Loop	Actions					
1	H	H	H	H	H	HH _n
2	H	H	H	H	H	H
3	H	H	H	H	H	H
4	XX	✓✓	✓✓	✓	✓	✓✓
5	✓	✓✓	✓✓	✓	✓	✓✓
6	✓	✓✓	✓✓	✓	✓	✓✓
7	✓	✓✓	✓✓	✓✓	✓✓	✓✓
8	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
9	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
10	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
11	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
12	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
13	✓	✓✓	✓✓	✓✓	✓✓	✓✓
14	✓	✓✓	✓✓	✓✓	✓✓	✓✓
15	✓	✓✓	✓✓	✓✓	✓✓	✓✓
16	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
17	X	✓✓	✓✓	✓✓	✓✓	✓✓
18	X	✓✓	✓✓	✓✓	✓✓	✓✓
19	X	✓✓	✓✓	✓✓	✓✓	✓✓
20	E					










Legend

- H Actions demonstrated by user
- H_n Extra, unneeded action provided by user
- ∅ Missing action, omitted by user
- ✓ Correct prediction, approved by user
- ✓✓ Correct prediction, automatic mode
- X Incorrect prediction, but score below threshold so user is prompted to confirm
- XX Incorrect prediction, but score above threshold so user must intervene
- E System correctly predicts end of task

Human demonstrated actions, as grouped by our motif-finding system

- Each row depicts one loop, as performed by the human user.
- Each column contains a group of equivalent actions, as grouped by our motif-finding algorithm.
- Where *extra* actions have been identified, these are displayed below the main grid as ‘noise’.

C.10. Creating list of filenames from a folder of remote computer (random ordered)152

Loop 0	[0] : Click()	[1] : Type(f2)	[2] : Type(CTRL-C)	[3] : Click()	[4] : Click()	[5] : Type(CTRL-V)
Loop 1	[7] : Click()	[8] : Type(f2)	[9] : Type(CTRL-C)	[10] : Click()	[11] : Click()	[12] : Type(CTRL-V)
Loop 2	[13] : Click()	[14] : Type(f2)	[15] : Type(CTRL-C)	[16] : Click()	[17] : Click()	[18] : Type(CTRL-V)

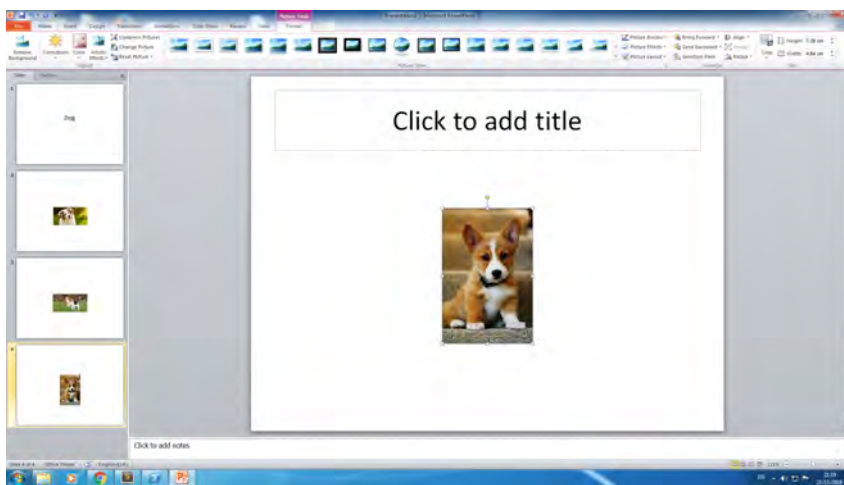
Noise:

[6] : Type(enter)

C.11 Creating Slides of images from folder of images

Here, we assume we have a folder of image files. The aim is to create a slideshow, where each slide displays a different image from the folder. For this we use Microsoft Powerpoint.

Illustrative screenshot of task



Per-action evaluation

- Here we show each action performed in the entire task as a symbol, to communicate both the input data and the successes and failures of our algorithm.
- Each row depicts one loop, and each column a group of equivalent actions.
- After the human demonstration ('H'), our system predicts future events. An ideal system would produce a screen of checkmarks ('✓✓'), while crosses (X) indicate failures.













Loop	Actions				
1	H	H	H	H	H
2	H	H	H	H	H
3	H _n H	H	H	H	H
4	✓	✓	✓	✓✓	✓
5	✓	✓	✓	✓✓	✓
6	✓	✓	✓	✓✓	✓
7	✓✓	✓✓	✓✓	✓✓	✓✓
8	✓✓	✓✓	✓✓	✓✓	✓✓
9	✓✓	✓✓	✓✓	✓✓	✓✓
10	✓✓	✓✓	✓✓	✓✓	✓✓
11	✓✓	✓✓	✓✓	✓✓	✓✓
12	✓✓	✓✓	✓✓	✓✓	✓✓
13	✓✓	✓✓	✓✓	✓✓	✓✓
14	✓✓	✓✓	✓	✓✓	✓✓
15	✓✓	✓✓	✗	✓✓	✓✓
16	✓✓	✓✓	✗	✓✓	✓✓
17	✓✓	✓✓	✗	✓✓	✓✓
18	✓✓	✓✓	✗	✓✓	✓✓
19	✓✓	✓✓	✗	✓✓	✓✓
20	✓✓	✓✓	✗	✓✓	✓✓
21	✓✓	✓✓	✗	✓✓	✓✓
22	✓✓	✓✓	✗	✓✓	✓✓
23	E				

Legend

- H Actions demonstrated by user
- H_n Extra, unneeded action provided by user
- ∅ Missing action, omitted by user
- ✓ Correct prediction, approved by user
- ✓✓ Correct prediction, automatic mode
- ✗ Incorrect prediction, but score below threshold so user is prompted to confirm
- ✗✗ Incorrect prediction, but score above threshold so user must intervene
- E System correctly predicts end of task

Human demonstrated actions, as grouped by our motif-finding system

- Each row depicts one loop, as performed by the human user.
- Each column contains a group of equivalent actions, as grouped by our motif-finding algorithm.
- Where *extra* actions have been identified, these are displayed below the main grid as ‘noise’.

Loop 0	[0] : Click()	[1] : Click()	[2] : Click()	[3] : Type(enter)	[4] : Click()
Loop 1	[5] : Click()	[6] : Click()	[7] : Click()	[8] : Type(enter)	[9] : Click()
Loop 2	[11] : Click()	[12] : Click()	[13] : Click()	[14] : Type(enter)	[15] : Click()

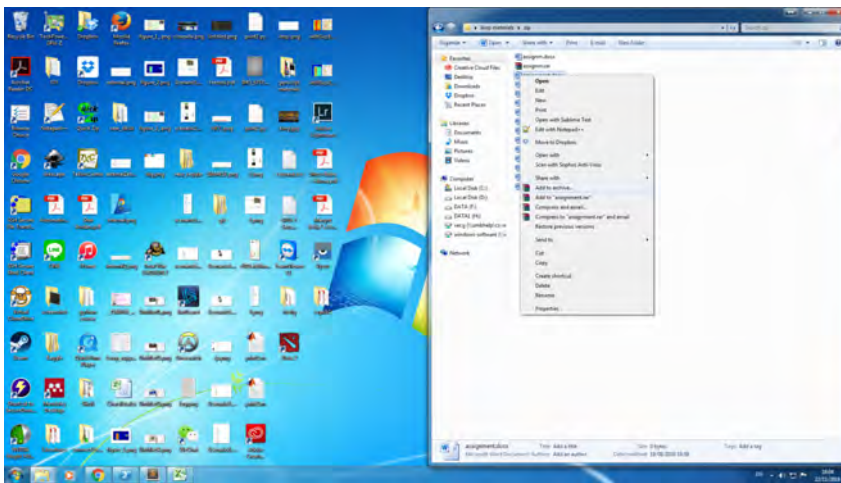
Noise:

[10] : Click()

C.12 Zipping every file in a folder

This task tackles a common operation problem which is easy for skilled computer users with programming knowledge, but difficult for novices. We desire each file in a folder to be zipped into its own zip file. This involves right-clicking on each file, and selecting 'Compress' from the menu.

Illustrative screenshot of task



Per-action evaluation

- Here we show each action performed in the entire task as a symbol, to communicate both the input data and the successes and failures of our algorithm.
- Each row depicts one loop, and each column a group of equivalent actions.
- After the human demonstration ('H'), our system predicts future events. An ideal system would produce a screen of checkmarks ('✓✓'), while crosses (X) indicate failures.







Loop	Actions		
1	H	H	H
2	H	H	H
3	H _n H	H	H
4	✓	✓	✓✓
5	✓	✓	✓✓
6	✓	✓	✓✓
7	✓✓	✓✓	✓✓
8	✓✓	✓✓	✓✓
9	✓✓	✓✓	✓✓
10	✓✓	✓✓	✓✓
11	✓✓	X	✓✓
12	✓✓	✓✓	✓✓
13	✓✓	✓✓	✓✓
14	XX	X	✓✓
15	XX		

Legend

- H Actions demonstrated by user
- H_n Extra, unneeded action provided by user
- ∅ Missing action, omitted by user
- ✓ Correct prediction, approved by user
- ✓✓ Correct prediction, automatic mode
- X Incorrect prediction, but score below threshold so user is prompted to confirm
- XX Incorrect prediction, but score above threshold so user must intervene
- E System correctly predicts end of task

Human demonstrated actions, as grouped by our motif-finding system

- Each row depicts one loop, as performed by the human user.
- Each column contains a group of equivalent actions, as grouped by our motif-finding algorithm.
- Where *extra* actions have been identified, these are displayed below the main grid as ‘noise’.

Loop 0	[0] : RClick()	[1] : Click()	[2] : Type(enter)
Loop 1	[3] : RClick()	[4] : Click()	[5] : Type(enter)
Loop 2	[7] : RClick()	[8] : Click()	[9] : Type(enter)

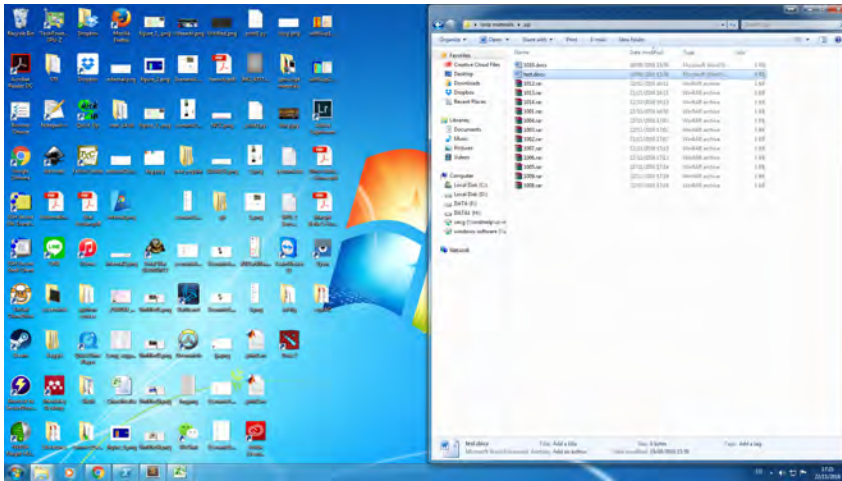
Noise:

[6] : Click()
--

C.13 Unzipping every file in a folder and renaming the files to the names of zip files

When students submit assignments as zip files, normally the zip files are named as student ID but the file inside is sometime named without student's identities. In this task, the user sets out to rename each file with the corresponding zip file name.

Illustrative screenshot of task



Per-action evaluation

- Here we show each action performed in the entire task as a symbol, to communicate both the input data and the successes and failures of our algorithm.
- Each row depicts one loop, and each column a group of equivalent actions.
- After the human demonstration ('H'), our system predicts future events. An ideal system would produce a screen of checkmarks ('✓'), while crosses (X) indicate failures.

C.13. Unzipping every file in a folder and renaming the files to the names of zip files 159

Loop	Actions										
1	H	H	H	H	H	H	H	H	H	H	H
2	H	H	H	H	H	H	H	H	H	H	HH _n H _n
3	H	H	H	H	H	H	H	H	H	H	H
4	H	H	H	∅	H	H	H	H	H	H	H
5	✓	✓✓	✓✓	XX	✓	✓	✓✓	✓✓	✓	✓✓	✓✓
6	✓	✓✓	✓✓	XX	✓	✓	✓✓	✓✓	✓	✓✓	✓✓
7	✓	✓✓	✓✓	XX	✓	✓	✓✓	✓✓	✓	✓✓	✓✓
8	✓✓	✓✓	✓✓	XX	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
9	✓✓	✓✓	✓✓	XX	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
10	✓✓	✓✓	✓✓	XX	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
11	✓✓	✓✓	✓✓	XX	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
12	✓✓	✓✓	✓✓	XX	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
13	✓	✓✓	✓✓	XX	✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
14	✓	✓✓	✓✓	XX	✓	✓✓	✓✓	✓✓	✓	✓✓	✓✓
15	E										














Legend

- H Actions demonstrated by user
- H_n Extra, unneeded action provided by user
- ∅ Missing action, omitted by user
- ✓ Correct prediction, approved by user
- ✓✓ Correct prediction, automatic mode
- X Incorrect prediction, but score below threshold so user is prompted to confirm
- XX Incorrect prediction, but score above threshold so user must intervene
- E System correctly predicts end of task

Human demonstrated actions, as grouped by our motif-finding system

- Each row depicts one loop, as performed by the human user.
- Each column contains a group of equivalent actions, as grouped by our motif-finding algorithm.
- Where *extra* actions have been identified, these are displayed below the main grid as ‘noise’.

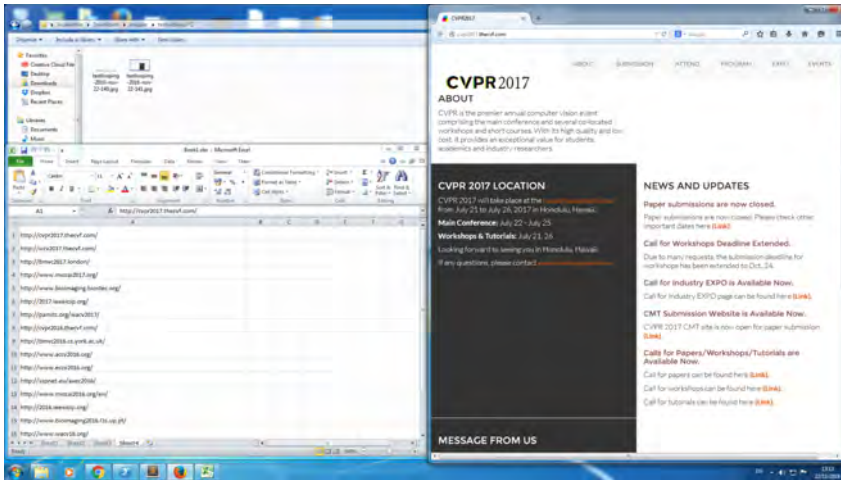
C.13. Unzipping every file in a folder and renaming the files to the names of zip files 160

Loop 0	[0] : Click()	[1] : Type(F2)	[2] : Type(CTRL-C)	[3] : Click()	[4] : RClick()	[5] : Click()	[6] : Type(delete)	[7] : Type(enter)	[8] : Click()	[9] : Type(F2)	[10] : Type(CTRL-V)
Loop 1	[11] : Click()	[12] : Type(F2)	[13] : Type(CTRL-C)	[14] : Click()	[15] : RClick()	[16] : Click()	[17] : Type(delete)	[18] : Type(enter)	[19] : Click()	[20] : Type(F2)	[21] : Type(CTRL-V)
Loop 2	[22] : Click()	[23] : Type(F2)	[24] : Type(CTRL-C)	***Missing***	[25] : RClick()	[26] : Click()	[27] : Type(delete)	[28] : Type(enter)	[29] : Click()	[30] : Type(F2)	[31] : Type(CTRL-V)

C.14 Taking screenshots of list of websites

Given a list of website URLs, the task is to paste each in turn into a browser address bar, and to save a screenshot of the rendered web page.

Illustrative screenshot of task



Per-action evaluation

- Here we show each action performed in the entire task as a symbol, to communicate both the input data and the successes and failures of our algorithm.
- Each row depicts one loop, and each column a group of equivalent actions.
- After the human demonstration ('H'), our system predicts future events. An ideal system would produce a screen of checkmarks ('✓✓'), while crosses (✗) indicate failures.

Loop	Actions						
1	H	H	H	H	H	H	H
2	H	H	H	H	H	H	H
3	H _n H	H	H	H	H	H	H
4	✓	✓	✓✓	XX	✓✓	✓✓	✓✓
5	✓	✓	✓✓	X	✓✓	✓✓	✓✓
6	✓	✓	✓✓	X	✓✓	✓✓	✓✓
7	✓✓	✓✓	✓✓	X	✓✓	✓✓	✓✓
8	✓✓	✓✓	✓✓	X	✓✓	✓✓	✓✓
9	✓✓	✓✓	✓✓	X	✓✓	✓✓	✓✓
10	✓✓	✓✓	✓✓	X	✓✓	✓✓	✓✓
11	✓✓	✓✓	✓✓	X	✓✓	✓✓	✓✓
12	✓✓	✓✓	✓✓	X	✓✓	✓✓	✓✓
13	✓✓	✓✓	✓✓	X	✓✓	✓✓	✓✓
14	✓✓	✓✓	✓✓	X	✓✓	✓✓	✓✓
15	✓✓	✓✓	✓✓	X	✓✓	✓✓	✓✓
16	✓✓	✓✓	✓✓	X	✓✓	✓✓	✓✓
17	E						

Legend

- H Actions demonstrated by user
- H_n Extra, unneeded action provided by user
- ∅ Missing action, omitted by user
- ✓ Correct prediction, approved by user
- ✓✓ Correct prediction, automatic mode
- X Incorrect prediction, but score below threshold so user is prompted to confirm
- XX Incorrect prediction, but score above threshold so user must intervene
- E System correctly predicts end of task

Human demonstrated actions, as grouped by our motif-finding system

- Each row depicts one loop, as performed by the human user.
- Each column contains a group of equivalent actions, as grouped by our motif-finding algorithm.
- Where *extra* actions have been identified, these are displayed below the main grid as ‘noise’.

Loop 0	[0] : Click(Book1.xlsx) Review	[1] : Click(http://cvp)	[2] : Type(CTRL-C)	[3] : Click(Search or)	[4] : Type(CTRL-V)	[5] : Type(enter)	[6] : Type(printscreens)
Loop 1	[7] : Click(Book1.xlsx) Review	[8] : Click(http://iccv)	[9] : Type(CTRL-C)	[10] : Click(.com)	[11] : Type(CTRL-V)	[12] : Type(enter)	[13] : Type(printscreens)
Loop 2	[15] : Click(look1.xlsx)	[16] : Click(http://iccv)	[17] : Type(CTRL-C)	[18] : Click(m)	[19] : Type(CTRL-V)	[20] : Type(enter)	[21] : Type(printscreens)

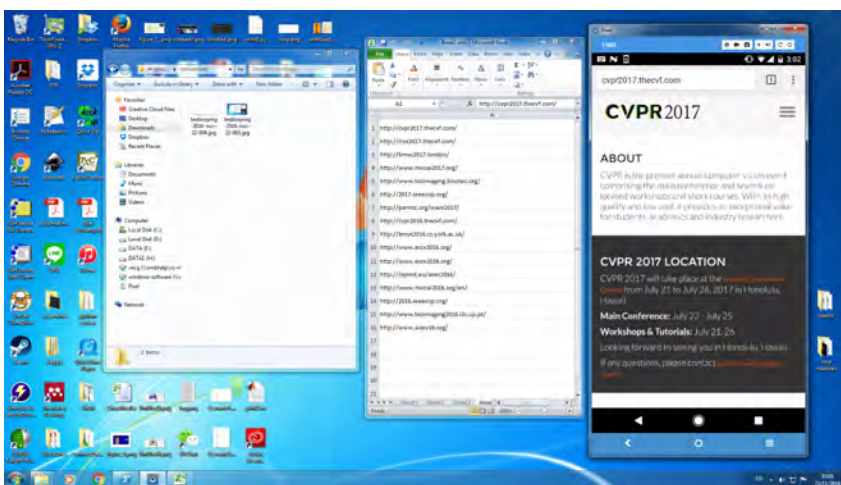
Noise:

[14] : Click(http://bmi)

C.15 Taking screenshots of list of websites on mobile

This task is similar to the screenshot task from Section 3.14, but now we take screenshots from a mobile phone accessed using VPN software. This situation is of particular use for testing how sites look on a mobile device.

Illustrative screenshot of task



Per-action evaluation

- Here we show each action performed in the entire task as a symbol, to communicate both the input data and the successes and failures of our algorithm.
- Each row depicts one loop, and each column a group of equivalent actions.
- After the human demonstration ('H'), our system predicts future events. An ideal system would produce a screen of checkmarks ('✓'), while crosses (X) indicate failures.







Loop	Actions							
1	H	H	H	H	H	H	H	H
2	H	H	H	H	H	H	H	H
3	H	H _n H	H	H	H	H	H	H
4	✓	✓	✓✓	✓	✓✓	✓✓	✓✓	✓
5	✓	✓	✓✓	✓	✓✓	✓✓	✓✓	✓
6	✓	✓	✓✓	✓	✓✓	✓✓	✓✓	✓
7	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
8	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
9	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
10	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
11	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
12	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
13	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
14	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
15	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
16	✓✓	✓	✓✓	✓✓	✓✓	✓✓	✓✓	✓✓
17	✓✓	E						

Legend

- H Actions demonstrated by user
- H_n Extra, unneeded action provided by user
- ∅ Missing action, omitted by user
- ✓ Correct prediction, approved by user
- ✓✓ Correct prediction, automatic mode
- ✗ Incorrect prediction, but score below threshold so user is prompted to confirm
- ✗✗ Incorrect prediction, but score above threshold so user must intervene
- E System correctly predicts end of task

Human demonstrated actions, as grouped by our motif-finding system

- Each row depicts one loop, as performed by the human user.
- Each column contains a group of equivalent actions, as grouped by our motif-finding algorithm.
- Where *extra* actions have been identified, these are displayed below the main grid as ‘noise’.

Loop 0	[0] : Click()	[1] : Click(http://)	[2] : Type(CTRL-C)	[3] : Click(Search)	[4] : Type(CTRL-V)	[5] : Type(enter)	[6] : Type(printscreen)	[7] : Click()
Loop 1	[8] : Click()	[9] : Click(http://)	[10] : Type(CTRL-C)	[11] : Click(Search)	[12] : Type(CTRL-V)	[13] : Type(enter)	[14] : Type(printscreen)	[15] : Click()
Loop 2	[16] : Click()	[18] : Click(http://)	[19] : Type(CTRL-C)	[20] : Click(Search)	[21] : Type(CTRL-V)	[22] : Type(enter)	[23] : Type(printscreen)	[24] : Click()

Noise:

[17] : Click()

Bibliography

- [1] Ifttt. <http://ifttt.com/>.
- [2] Macrodroid. <http://www.macrodroid.co.uk/>.
- [3] Tasker. <http://tasker.dinglis.ch/>.
- [4] Mac OS X Automator. <https://support.apple.com/en-gb/HT2488>, 2014. Accessed: 18th April, 2017.
- [5] Sikuli Slides. <http://slides.sikuli.org/>, 2014. Accessed: 18th April, 2017.
- [6] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.
- [7] Hidenao Abe, Miho Ohsaki, Hideto Yokoi, and Takahira Yamaguchi. Implementing an integrated time-series data mining environment based on temporal pattern extraction methods: A case study of an interferon therapy risk mining for chronic hepatitis. In *New Frontiers in Artificial Intelligence: Joint JSAI Workshop*, 2006.
- [8] I. P. Androulakis, J. Vitolo, and C. Roth. Selecting maximally informative genes to enable temporal expression profiling analysis. In *Proceedings of Foundations of Systems Biology in Engineering*, 2005.

- [9] V. Antila, J. Polet, A. Lämsä, and J. Liikka. Routinemaker: Towards end-user automation of daily routines using smartphones. In *2012 IEEE International Conference on Pervasive Computing and Communications Workshops*, pages 399–402, March 2012.
- [10] Amos Azaria, Jayant Krishnamurthy, and Tom Mitchell. Instructable intelligent personal agent, 2016.
- [11] Anthony Bagnall, Jon Hills, and Jason Lines. Technical Report CMPC14-03: Finding Motif Sets in Time Series. *arXiv:1407.3685v1*, 2014.
- [12] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow. Deepcoder: Learning to write programs. *arXiv:1611.01989*, 2016.
- [13] Nikola Banovic, Tovi Grossman, Justin Matejka, and George Fitzmaurice. Waken : Reverse Engineering Usage Information and Interface Structure from Software Videos. *UIST '12*, pages 83–92, 2012.
- [14] Charles Beattie, Joel Z Leibo, Denis Teplyashin, Tom Ward, Marcus Wainwright, Heinrich Küttler, Andrew Lefrancq, Simon Green, Víctor Valdés, Amir Sadik, et al. Deepmind lab. *arXiv preprint arXiv:1612.03801*, 2016.
- [15] Marc G. Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. In *Proceedings of the 24th International Conference on Artificial Intelligence, IJCAI'15*, pages 4148–4152. AAAI Press, 2015.
- [16] Nadav Bhonker, Shai Rozenberg, and Itay Hubara. Playing snes in the retro learning environment. *arXiv preprint arXiv:1611.02205*, 2016.
- [17] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI Gym. *arXiv:1606.01540*, 2016.
- [18] Capcom. *Mega Man X*. Game [SNES], December 1993.

- [19] Vittorio Castelli, Lawrence Bergman, Tessa Lau, and Daniel Oblinger. Sheep-dog, parallel collaborative programming-by-demonstration. *Knowledge-Based Systems*, 2010.
- [20] Tsung-Hsiang Chang, Tom Yeh, and Robert C Miller. GUI Testing Using Computer Vision. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1535–1544, 2010.
- [21] Devendra Singh Chaplot and Guillaume Lample. Arnold: An autonomous agent to play fps games. In *AAAI*, pages 5085–5086, 2017.
- [22] Jiun-Hung Chen and Daniel S. Weld. Recovering from errors during programming by demonstration. In *Proceedings of the 13th International Conference on Intelligent User Interfaces*, IUI '08, pages 159–168, New York, NY, USA, 2008. ACM.
- [23] Yun-Gyung Cheong, Arnav Jhala, Byung-Chull Bae, and Robert Michael Young. Automatically generating summary visualizations from game logs. In *AIIDE*, pages 167–172, 2008.
- [24] James Conley, Ed Andros, Priti Chinai, and Elise Lipkowitz. Use of a game over: Emulation and the video game industry, a white paper. *Nw. J. Tech. & Intell. Prop.*, 2:261, 2003.
- [25] Allen Cypher and Daniel Conrad Halbert. *Watch What I Do: Programming by Demonstration*. MIT press, 1993.
- [26] Morgan Dixon, Daniel Leventhal, and James Fogarty. Content and Hierarchy in Pixel-Based Methods for Reverse Engineering Interface Structure. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, page 969, 2011.
- [27] Morgan Dixon, A. Conrad Nied, and James Fogarty. Prefab Layers and Prefab Annotations: Extensible Pixel-Based Interpretation of Graphical Interfaces. *UIST '14*, pages 221–230, 2014.

- [28] Koushik Dutta. Vysor. <https://www.vysor.io/>, 2017.
- [29] Krzysztof Gajos and Daniel S. Weld. Supple: Automatically generating user interfaces. In *Proceedings of the 9th International Conference on Intelligent User Interfaces*, IUI '04, pages 93–100, New York, NY, USA, 2004. ACM.
- [30] Alexander L. Gaunt, Marc Brockschmidt, Rishabh Singh, Nate Kushman, Pushmeet Kohli, Jonathan Taylor, and Daniel Tarlow. Terpret: A probabilistic programming language for program induction. *arXiv:1608.04428*, 2016.
- [31] Floraine Grabler, Maneesh Agrawala, Wilmot Li, Mira Dontcheva, and Takeo Igarashi. Generating Photo Manipulation Tutorials by Demonstration. *ACM Transactions on Graphics*, 28(3):1, 2009.
- [32] Alexander Groß, Jan Friedland, and Friedhelm Schwenker. Learning to play tetris applying reinforcement learning methods. In *ESANN*, pages 131–136, 2008.
- [33] Tovi Grossman, Justin Matejka, and George Fitzmaurice. Chronicle: Capture, Exploration, and Playback of Document Workflow Histories. *UIST '10*, pages 143–152, 2010.
- [34] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *ACM SIGPLAN Notices*, 2011.
- [35] Sumit Gulwani. Programming by examples (and its applications in data wrangling). In *Verification and Synthesis of Correct and Secure Systems*. IOS Press, 2016.
- [36] Sumit Gulwani, William R Harris, and Rishabh Singh. Spreadsheet data manipulation using examples. *Communications of the ACM*, 2012.
- [37] Sumit Gulwani, Jose Hernandez-Orallo, Emanuel Kitzelmann, Stephen H Muggleton, Ute Schmid, and Benjamin Zorn. Inductive programming meets the real world. *Communications of the ACM*, 2015.

- [38] Yuan Hao, Mohammad Shokoohi-Yekta, George Papageorgiou, and Eamonn Keogh. Parameter-free audio motif discovery in large data archives. In *International Conference on Data Mining*, 2013.
- [39] Harrison Ho, Varun Ramesh, and Eduardo Torres Montano. Neuralkart: A real-time mario kart 64 ai. 2017.
- [40] Minh Hoai, Zhen-Zhong Lan, and Fernando De la Torre. Joint Segmentation and Classification of Human Actions in Video. *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, pages 3265–3272, 2011.
- [41] Kevin Hughes. Tensorkart: self-driving mariokart with tensorflow. Blog, December 2016. Accessed April 13, 2018.
- [42] Amy Hurst, Scott E Hudson, and Jennifer Mankoff. Automatically Identifying Targets Users Interact with During Real World Tasks. *IUI '10*, pages 11–20, 2010.
- [43] Thanapong Intharah, Daniyar Turmukhambetov, and Gabriel J. Brostow. Help, it looks confusing: Gui task automation through demonstration and follow-up questions. In *Proceedings of the 22nd International Conference on Intelligent User Interfaces*, IUI '17. ACM, 2017.
- [44] Bernard J. Jansen, Danielle L. Booth, and Amanda Spink. Determining the informational, navigational, and transactional intent of web queries. *Information Processing & Management*, 44(3):1251 – 1266, 2008.
- [45] Daxin Jiang, Jian Pei, and Hang Li. Mining search and browse logs for web search: A survey. *ACM Trans. Intell. Syst. Technol.*, 4(4):57:1–57:37, October 2013.
- [46] Matthew Johnson, Katja Hofmann, Tim Hutton, and David Bignell. The malmo platform for artificial intelligence experimentation. In *IJCAI*, pages 4246–4247, 2016.

- [47] Dimosthenis Karatzas, Lluís Gomez-Bigorda, Anguelos Nicolaou, Suman Ghosh, Andrew Bagdanov, Masakazu Iwamura, Jiri Matas, Lukas Neumann, Vijay Ramaseshan Chandrasekhar, Shijian Lu, et al. ICDAR 2015 Competition on Robust Reading. In *Document Analysis and Recognition (ICDAR)*, 2015.
- [48] Andrej Karpathy. Mini world of bits benchmark. <http://alpha.openai.com/miniwob/>, 2017.
- [49] Alan C. Kay. Computer software. *Scientific American*, 1984.
- [50] Michał Kempka, Marek Wydmuch, Grzegorz Runc, Jakub Toczek, and Wojciech Jaśkowski. Vizdoom: A doom-based ai research platform for visual reinforcement learning. In *Computational Intelligence and Games (CIG), 2016 IEEE Conference on*, pages 1–8. IEEE, 2016.
- [51] Guillaume Lample and Devendra Singh Chaplot. Playing fps games with deep reinforcement learning. In *AAAI*, pages 2140–2146, 2017.
- [52] Tessa Lau. Why PBD systems fail: Lessons learned for usable AI. In *CHI 2008 Workshop on Usable AI*, 2008.
- [53] Tessa Lau, Lawrence Bergman, Vittorio Castelli, and Daniel Oblinger. Sheepdog: learning procedures for technical support. In *International conference on Intelligent user interfaces*, 2004.
- [54] Uichin Lee, Zhenyu Liu, and Junghoo Cho. Automatic identification of user goals in web search. In *Proceedings of the 14th International Conference on World Wide Web, WWW '05*, pages 391–400, New York, NY, USA, 2005. ACM.
- [55] Gilly Leshed, Eben M. Haber, Tara Matthews, and Tessa Lau. CoScripter : Automating & Sharing How-To Knowledge in the Enterprise. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1719–1728, 2008.

- [56] O. Levy and L. Wolf. Live repetition counting. In *International Conference on Computer Vision (ICCV)*, 2015.
- [57] Liangda Li, Hongbo Deng, Anlei Dong, Yi Chang, and Hongyuan Zha. Identifying and labeling search tasks via query-based hawkes processes. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '14*, pages 731–740, New York, NY, USA, 2014. ACM.
- [58] Toby Jia-Jun Li, Amos Azaria, and Brad A. Myers. Sugilite: Creating multi-modal smartphone automation by demonstration. CHI '17, 2017.
- [59] Henry Lieberman. *Your Wish is My Command: Programming By Example*. Morgan Kaufmann, 2001.
- [60] Jessica Lin, Eamonn Keogh, Stefano Lonardi, and Pranav Patel. Finding motifs in time series. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining Workshop on Temporal Data Mining*, 2002.
- [61] Greg Little, Tessa A. Lau, Allen Cypher, James Lin, Eben M. Haber, and Eser Kandogan. Koala: Capture, Share, Automate, Personalize Business Processes on the Web. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 943–946, 2007.
- [62] Claudio Lucchese, Salvatore Orlando, Raffaele Perego, Fabrizio Silvestri, and Gabriele Tolomei. Identifying task-based sessions in search engine query logs. In *Proceedings of the fourth ACM international conference on Web search and data mining*, pages 277–286. ACM, 2011.
- [63] Claudio Lucchese, Salvatore Orlando, Raffaele Perego, Fabrizio Silvestri, and Gabriele Tolomei. Discovering tasks from search engine query logs. *ACM Trans. Inf. Syst.*, 31(3):14:1–14:43, August 2013.
- [64] Arnold M Lund. Measuring usability with the use questionnaire. *Usability interface*, 8(2):3–6, 2001.

- [65] Pattie Maes. Agents that reduce work and information overload. *Communications of the ACM*, 1994.
- [66] Raphaël Marczak, Jasper van Vught, Gareth Schott, and Lennart E. Nacke. Feedback-based gameplay metrics: Measuring player experience via automatic visual analysis. In *Proceedings of The 8th Australasian Conference on Interactive Entertainment: Playing the System*, IE '12, pages 6:1–6:10, New York, NY, USA, 2012. ACM.
- [67] Rodrigo de A. Maués and Simone Diniz Junqueira Barbosa. Keep doing what i just did: Automating smartphones by demonstration. In *Proceedings of the 15th International Conference on Human-computer Interaction with Mobile Devices and Services*, MobileHCI '13, pages 295–303, New York, NY, USA, 2013. ACM.
- [68] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.
- [69] Shiwali Mohan and John E Laird. Learning to play mario. *Tech. Rep. CCA-TR-2009-03*, 2009.
- [70] A. Mueen, E. Keogh, Q. Zhu, S. S. Cash, and M. B. Westover. Exact discovery of time series motifs. In *SDM*, 2009.
- [71] Abdullah Mueen, Eamonn Keogh, Qiang Zhu, Sydney S. Cash, M. Brandon Westover, and Nima Bigdely-Shamlo. A disk-aware algorithm for time series motif discovery. *Data Mining and Knowledge Discovery*, 2011.
- [72] Lennart Nacke, Craig Lindley, and Sophie Stellmach. Log whos playing: psychophysiological game analysis made easy through event logging. In *Fun and games*, pages 150–157. Springer, 2008.
- [73] Lennart Nacke, Jonas Schild, and Joerg Niesenhaus. Gameplay experience testing with playability and usability surveys—an experimental pilot study. In

Proceedings of the Fun and Games 2010 Workshop, NHTV Expertise Series, volume 10, 2010.

- [74] Nicholas Negroponte. *The Architecture Machine*. MIT press, 1970.
- [75] John Walker Orr, Prasad Tadepalli, Janardhan Rao Doppa, Xiaoli Fern, and Thomas G Dietterich. Learning scripts as hidden markov models. 2014.
- [76] Alexey Pajitnov and Vladimir Pokhilko. *Tetris*. Game [NES], June 1984.
- [77] Gordon W Paynter. Automating iterative tasks with programming by demonstration. 2000.
- [78] Karl Pichotta and Raymond J Mooney. Learning statistical scripts with lstm recurrent neural networks. In *AAAI*, pages 2800–2806, 2016.
- [79] Suporn Pongnumkul, Mira Dontcheva, Wilmot Li, Jue Wang, Lubomir Bourdev, Shai Avidan, and Michael F. Cohen. Pause-and-Play: Automatically Linking Screencast Video Tutorials with Applications. *UIST '11*, pages 135–144, 2011.
- [80] Dino Ratcliffe, Sam Devlin, Udo Kruschwitz, and Luca Citi. Clyde: A deep reinforcement learning doom playing agent. 2017.
- [81] Daniel E. Rose and Danny Levinson. Understanding user goals in web search. In *Proceedings of the 13th International Conference on World Wide Web*, WWW '04, pages 13–19, New York, NY, USA, 2004. ACM.
- [82] Rachel Rudinger, Vera Demberg, Ashutosh Modi, Benjamin Van Durme, and Manfred Pinkal. Learning to predict script events from domain-specific text. In *Proceedings of the Fourth Joint Conference on Lexical and Computational Semantics*, pages 205–210, 2015.
- [83] Qinfeng Shi, Li Cheng, Li Wang, and Alex Smola. Human Action Segmentation and Recognition Using Discriminative Semi-Markov Models. *International Journal of Computer Vision*, 93(1):22–32, 2011.

- [84] Qinfeng Shi, Li Wang, Li Cheng, and Alex Smola. Discriminative Human Action Segmentation and Recognition Using Semi-Markov Model. *26th IEEE Conference on Computer Vision and Pattern Recognition*, 2008.
- [85] Val Shute. Stealth assessment in video games. 2015.
- [86] Valerie J Shute and Gregory R Moore. Consistency and validity in game-based stealth assessment. *Technology enhanced innovative assessment: Development, modeling, and scoring from an interdisciplinary perspective*, pages 31–51, 2017.
- [87] Valerie J Shute, Matthew Ventura, and Diego Zapata-Rivera. Stealth assessment in digital games, 2013.
- [88] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.
- [89] Brian A. Smith and Shree K. Nayar. Mining controller inputs to understand gameplay. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*, UIST '16, pages 157–168, New York, NY, USA, 2016. ACM.
- [90] Tadashi Sugiyama and Hideki Konno. *Super Mario Kart*. Game [SNES], August 1992.
- [91] Amund Tveit and Gisle B Tveit. Game usage mining: Information gathering for knowledge discovery in massive multiplayer games. In *International Conference on Internet Computing*, pages 636–642, 2002.
- [92] Shigeharu Umezaki and Shinji Kitamoto. *Contra*. Game [NES], February 1987.
- [93] Cheng-Yao Wang, Wei-Chen Chu, Hou-Ren Chen, Chun-Yen Hsu, and Mike Y Chen. EverTutor: Automatically Creating Interactive Guided Tutorials on Smartphones by User Demonstration. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 4027–4036, 2014.

- [94] Xiaopeng Xi, Eamonn Keogh, Li Wei, and Agenor Mafra-Neto. Finding motifs in a database of shapes. In *SIAM International Conference on Data Mining*.
- [95] Tom Yeh, Tsung-Hsiang Chang, and Robert C Miller. Sikuli: using gui screenshots for search and automation. *UIST '09*, 2009.
- [96] Tom Yeh, Tsung-Hsiang Chang, Bo Xie, Greg Walsh, Ivan Watkins, Krist Wongsuphasawat, Man Huang, Larry S. Davis, and Benjamin B. Bederson. Creating Contextual Help for GUIs Using Screenshots. *UIST '11*, page 145, 2011.
- [97] Loutfouz Zaman and I Scott MacKenzie. Evaluation of nano-stick, foam buttons, and other input methods for gameplay on touchscreen phones. In *International Conference on Multimedia and Human-Computer Interaction-MHCI*, pages 69–1, 2013.
- [98] Min-Ling Zhang and Zhi-Hua Zhou. A review on multi-label learning algorithms. *IEEE transactions on knowledge and data engineering*, 26(8):1819–1837, 2014.