ActionNet: Vision-based Workflow Action Recognition From Programming Screencasts

Dehai Zhao, Zhenchang Xing
Research School of Computer Science
Australian National University
Australia
{dehai.zhao, zhenchang.xing}@anu.edu.au

*Chunyang Chen, *Xin Xia
Faculty of Information Technology
Monash University
Australia
{chunyang.chen, Xin.Xia}@monash.edu

*Guoqiang Li
School of Software
Shanghai Jiao Tong
University, Shanghai,China
li.g@sjtu.edu.cn

***Abstract*—Programming screencasts have two important applications in software engineering context: study developer behaviors, information needs and disseminate software engineering knowledge. Although programming screencasts are easy to produce, they are not easy to analyze or index due to the image nature of the data. Existing techniques extract only content from screencasts, but ignore workflow actions by which developers accomplish programming tasks. This significantly limits the effective use of programming screencasts in downstream applications. In this paper, we are the first to present a novel technique for recognizing workflow actions in programming screencasts. Our technique exploits image differencing and Convolutional Neural Network (CNN) to analyze the correspondence and change of consecutive frames, based on which nine classes of frequent developer actions can be recognized from programming screencasts. Using programming screencasts from Youtube, we evaluate different configurations of our CNN model and the performance of our technique for developer action recognition across developers, working environments and programming languages. Using screencasts of developers' real work, we demonstrate the usefulness of our technique in a practical application for action-aware extraction of key-code frames in developers' work.**

***Keywords*—Programming Screencast; Action Recognition; Deep learning;**

I. Introduction

Screencasting is a technique to record computer screen output at a specific time interval (e.g., 1/15 second). In the context of software engineering, programming screencasts provide a direct record of both a developer's workflow actions and the application content involved in programming tasks (e.g., typing code, scrolling content, switching windows) . They not only provide the data basis to study developer behaviors and information needs in software engineering research [1]–[6], but they are also a common content carrier for disseminating software engineering knowledge [7], [8].

In the application of studying developers, programming screencasts provide direct observational data, as opposed to survey and interview data that rely on self-reporting [9], [10]. According to a recent survey of data collection methods used in the 26 papers that study developer behaviors [11], 21 of these papers rely on programming screencasts to study a wide range of software engineering activities, such as feature location [1], debugging [12], [13], program comprehension [14]–[16], tool design [6], [17], and distributed programming [18]. In the application of disseminating software engineering knowledge, programming screencasts offer live-coding experience which is absent in text-based tutorials. Millions of programming tutorials are published on Youtube and are watched by millions of developers. Seeing a developer's coding in action, for example, how changes are made to source code step by step and how errors occur and are fixed, can be more valuable than text-based tutorials [8].

As all operating systems provide a simple API to record computer screen, screencasting does not need application-specific support which makes it very easy to deploy, compared with software instrumentation that requires sophisticated accessibility or UI automation APIs [5], [19], [20]. However, this easy-to-deploy convenience comes with a high-barrier of video analytics. As a screencast is a sequence of screen-captured images, one cannot study the developer behaviors or harness the programming knowledge in the screencast until the workflow actions and the application content captured in the screencast can be effectively extracted [4], [8], [11], [21].

It is very time-consuming to manually identify the workflow actions and the application content in a screencast [11]. This limits the scalability of behavioral research on software developers. Existing automatic techniques [11], [21] focus on only the content extraction from screencasts using Optical Character Recognition (OCR) techniques [22], but ignore the workflow actions, i.e., actions that the developer takes to accomplish programming tasks. Ignoring the workflow actions makes programming screencasts less valuable in studying developer behaviors, because we lose the dynamic aspects of the developer's coding practice. For example, we cannot see what actions lead to program errors and how the developers fix the errors, or what is the bottleneck for code search. Ignoring the workflow actions in programming video tutorials also limits the ways that developers can search and navigate the video tutorials, resulting in less effective learning experience [8].

In fact, extracting content without considering workflow actions itself is problematic, resulting in noisy extracted content that will negatively affect the effective use of programming screencasts in studying developer behaviors or learning programming knowledge. For example, as shown in the frames

*Corresponding authors

f_x and f_{x+1} in Fig. 1, while the developer is typing the code, a popup window appears to suggest APIs. This popup window makes the current screenshot very different from the previous screenshot and existing frame-similarity based methods [11], [21] will select f_{x+1} for content extraction. First, the popup window likely contains APIs irrelevant to the developer's current code. Second, it blocks the actual code in the main window. Third, the content in the popup window mixed together with the main window content will degrade the OCR quality. As another example, the developer selects a piece of code (see the frames f_y and f_{y+1} in Fig. 1). This also results in enough screen changes which again triggers the content extraction by frame-similarity based methods. The selected code has different background and foreground color from other code, which will degrade the OCR quality. However, the code does not actually change which means that the content extraction for f_{y+1} is completely unnecessary.

To overcome the limitation of the content-centric analysis of programming screencasts, this paper presents a deep learning based computer vision technique to automatically recognize developer actions from programming screencasts. In this work, we focus on three categories of nine actions (see Table I) frequently observed in programming work. We do not limit the action occurrences in only IDEs, as programming work may involve many other software tools (e.g., web browser, interactive shell). Our approach first uses image differencing techniques to detect the change regions between the two consecutive frames, resulting from developer actions. The detected change regions are then fed into a Convolutional Neural Network (CNN) model to extract abstract image features, which will then be fed into a softmax classifier to predict the action that most likely causes the screen changes.

We collect 50 programming screencasts (25 for Python and 25 for Java) from the 10 popular programming playlists on Youtube. These 10 playlists are produced by 10 different developers and use different development tools. Through intra-playlist, inter-playlist and inter-programming-language experiments, we show that our approach can be effectively trained and deployed in very diverse working environment and programming language settings (F1-score>0.7), on par with the accuracy of popular human action recognition techniques [23]–[25]. We further collect 10 hours of screencasts of two developers' real work and ask the developers to identify key-code frames in the screencasts. We demonstrate that action-aware extraction of key-code frames identifies key-code frames that correspond to the developers' annotations much better than existing action-agnostic methods.

We make the following contribution in this paper:

- To the best of our knowledge, this is the first work to recognize workflow actions from programming screencasts. We extract finer-grained workflow actions than previous work on the manual analysis of developer behaviors.
- To extract workflow actions, we propose a two-stage deep learning based method. It first detects screen changes by image differencing, and then recognizes developer actions from screen changes with a CNN-based model.

TABLE I
THE CATEGORY OF ACTIONS TO BE RECOGNIZED IN THIS WORK

General Category	ID	Description
Control cursor/mouse	C1	Move cursor by keyboard
	C2	Move mouse over text region
	C3	Move mouse over non-text region
Edit content	C4	Enter text (e.g., char, word, paragraph)
	C5	Delete text (e.g., char, word, paragraph)
Interact with app	C6	Trigger popups (e.g., menu, tooltip)
	C7	Scroll text (e.g., code, console output)
	C8	Select text (e.g, code, console output)
	C9	Switch window (within or across app)
	C10	Others (e.g., resize window, click button)

- Through extensive experiments, we not only confirm the effectiveness and generality of our method, but also demonstrate its usefulness for action-aware extraction of key-code frames in programming screencasts which can enable more accurate code extraction or video search.

II. PROBLEM STATEMENT

A programming screencast is a sequence of time-stamped screenshots (i.e., computer screen outputs) recorded at a specific time interval while the developer is working on a computer. Each screenshot is a screen image and is referred to as a frame in the screencast. The interaction between the developer and the development tools during screencasting results in the visual changes on the computer screen over time, for example, typing a char results in the typed char appearing on the screen, selecting a word results in the change of the foreground and background color of the selected word.

When people watch the screencast, they can manually recognize a sequence of developer actions from the screen changes in consecutive frames. The goal of this work is to develop a computer-vision based technique to automate the recognition of developer actions in programming screencasts. As illustrated in Fig. 1, our technique (ActionNet) takes as input a sequence of frames in a programming screencast, and automatically produces as output a sequence of actions that the developer performs on the computer during screencasting.

Developer actions can be at various levels of abstraction [4]. For example, an "edit file" activity can comprise primitive actions such as "enter text", "select text", "delete text", "scroll text". A "browse web" activity can comprise primitive actions such as "enter query", "scroll web page", "select web page content". A "debug code" activity can comprise primitive actions such as "scroll code", "switch file". Although high-level activities differ greatly in goals and software involved, they share primitive actions in the process of human-computer interaction. In this work, we decide to recognize primitive actions in programming screencasts. The primitive actions can be aggregated into high-level activities by rule-based or machine learning techniques [4], [8].

We define nine classes of developer actions to be recognized in programming screencasts, based on our own programming experiences, the survey of HCI literature [5], [26], [27], and the empirical coding of frequent primitive actions in the 10 randomly selected programming screencasts on Youtube. These nine classes of primitive actions are frequent actions in programming work and they fall into three general categories

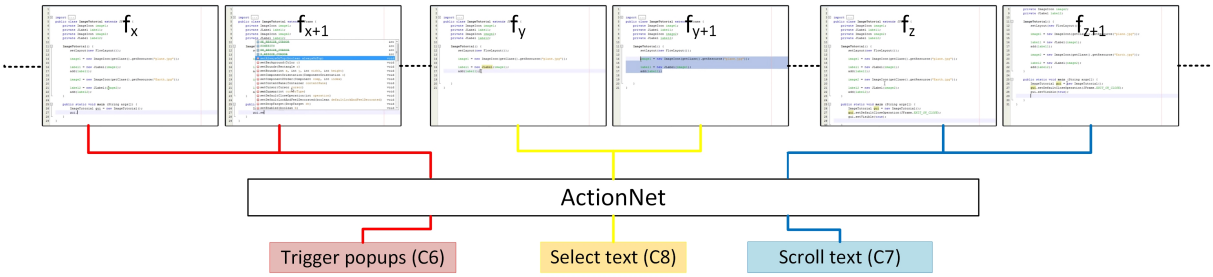

Fig. 1. An Illustration of Workflow Action Recognition in Programming Screencasts

(see Table I). These primitive actions can occur in IDEs, interactive shell, web browsers and text editors that are commonly used in software development. Except for mouse and cursor actions that can be instrumented using simple operating-system level APIs, instrumenting other classes of actions will require accessibility or UI automation APIs [5], [11]

We classify all other less frequent HCI actions (e.g., resize window, click button) as an "others" class (C10). Note that we do not consider "no action" class, because "no action" can be easily determined when the consecutive frames remain unchanged by image differencing (see Section III-A). As a programming screencast often contains many "no action" periods, considering "no action" class will superficially inflate the model performance but has no practical meaning.

Recognizing developer actions on computer in a programming screencast differs significantly from recognizing human actions in a natural scene video. Human actions in a natural scene, such as press key, move mouse, have a physical duration, and they cause changes in at least several frames. In contrast, the computer screen changes resulting from the developer actions on computer are computer rendered, and they happen instantly from one frame to next. Therefore, we must be able to recognize developer actions with only the information in two consecutive frames.

We formulate our task as a multi-class classification problem which predicts the probability of the above 10 classes primitive actions (including "others") given the two consecutive frames in a programming screencast.

III. Approach

Fig. 2 presents the main steps of our approach. Given a screencast with N frames, our approach analyzes the two consecutive frames f_i and f_{i+1} $(1 \leq i < N)$ sequentially from the beginning to the end of the screencast. First, it uses image differencing technique to detect the largest change region $diff_i^{i+1}$ in between the two frames f_i and f_{i+1}. Then, it crops the screen region $R_i@diff_i^{i+1}$ and $R_{i+1}@diff_i^{i+1}$ on f_i and f_{i+1} respectively with respect to $diff_i^{i+1}$. Next, the cropped screen regions are fed into a CNN-based feature extractor that is trained to extract image features of the correspondence and change between the two cropped screen regions. Finally, based on the extracted image features, a softmax classifier is trained to predict the probability of the 10 classes of primitive actions (including "others") that most likely cause the screen changes from f_i to f_{i+1}.

A. Change Region Detection by Image Differencing

A naive solution to our problem would be to predict developer actions directly from the two consecutive frames. However, this solution will not be effective, because many developer actions, such as typing a char or moving the mouse pointer, result in very small screen changes, compared with the size of the whole screen. If we take the whole frames as input, the important features in small screen changes would be too weak to recognize the corresponding developer actions. This is especially the case when extracting image features using deep neural network [28]. Therefore, we decide to detect the change regions between the two frames and then use these change regions for action recognition.

We adopt the mature computer vision technique that is widely used to detect change regions between two frames [29]–[31]. Specifically, we use scikit-image APIs [32] to detect change regions in the two frames f_i and f_{i+1}. As illustrated in Fig. 3, we first compute the structural similarity index between the same-position pixels of the two frames. Structural similarity compares local patterns of pixel intensities that have been normalized for luminance and contrast. Based on the pixel structural similarities, a black and white image can be obtained in which white means the same pixels and black means the different pixels between the two frames. We find the bounding boxes of the black pixels which identify the change regions between the two frames. We then crop the screen regions on f_i and f_{i+1} with respect to the identified change regions.

One technical challenge in using change regions for action recognition is that there can be more than one change regions between two frames. For example, the screencast may record system clock update in addition to the screen changes resulting from developer actions. Furthermore, one developer action may result in several screen changes. For example, entering a word results in a direct screen change (the word appears), but may also result in indirect screen changes (e.g., a code assist icon appears on the editor ruler).

To identify change regions directly related to developer actions, we use two simple filters observed during our manual labelling of developer actions in programming screencasts (see Section IV-B). First, we observe that the minimum change regions related to developer actions resulting from cursor movement are always of 5 by 16 pixels. Therefore, we discard any change regions smaller than 5 by 16 pixels. Second, we observe that as a response to human actions, screen

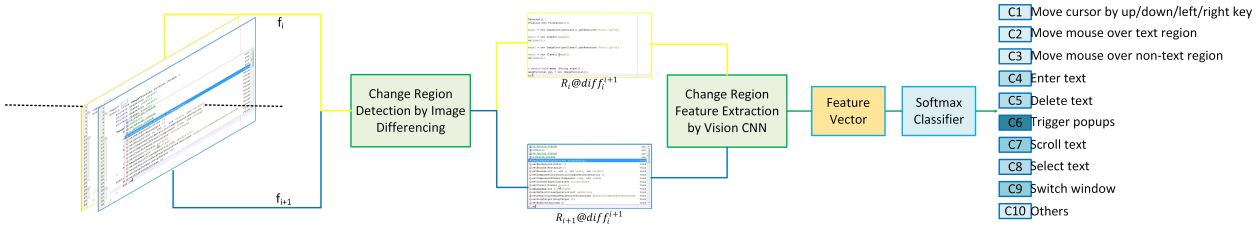

Fig. 2. An Overview of the Main Steps of Our ActionNet

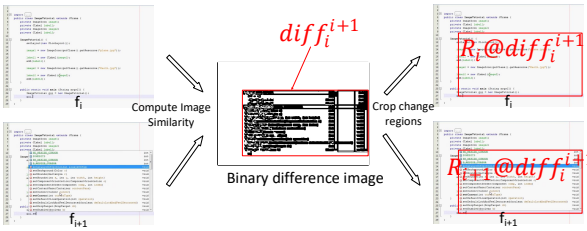

Fig. 3. Steps to Detect Changes Regions in Between f_i and f_{i+1}

changes directly related to developer actions will be the largest changes. Therefore, we keep only the largest change region between f_i and f_{i+1}. We refer to this largest change region as $diff_i^{i+1}$, and the screen regions on f_i and f_{i+1} with respect to $diff_i^{i+1}$ as $R_i@diff_i^{i+1}$ and $R_{i+1}@diff_i^{i+1}$, respectively.

B. Change Region Feature Extraction by Vision CNN

To predict actions from screen changes, we must be able to identify the correspondence and contrast features from screen changes. In this work, we consider 10 classes of developer actions. The screen changes resulting from these actions have intra-class variations. For example, entering text can be either typing a char or pasting a paragraph of text. The background and foreground color changes resulting from text selection may differ from one application to another. Meanwhile, we have inter-class similarities. For example, switching window, scrolling text and triggering popups may look alike in term of large screen content changes. Finally, we have working environment variations across developers. Some developers use IDEs, some use text editors, and others use the interactive console. Even for the same tool, different developers may use different color themes, font size, etc.

All these variations make manual feature engineering for action recognition in programming screencasts infeasible. Inspired by the success of CNN for computer vision tasks, we design CNN-based feature extractors that automatically learn to extract effective image features from training data without the need for manual feature engineering.

1) Input Change Regions to CNN: For the CNN to extract effective features for action recognition, we must provide it sufficient information about screen changes resulting from developer actions. We adopt three strategies to produce the input screen change regions to CNN.

The first strategy (change-contrast) uses $R_i@diff_i^{i+1}$ on f_i and $R_{i+1}@diff_i^{i+1}$ on f_{i+1} with respect to the largest change region $diff_i^{i+1}$. This strategy contrasts the corresponding largest change region $R_i@diff_i^{i+1}$ and $R_{i+1}@diff_i^{i+1}$ on f_i and f_{i+1} respectively to recognize developer actions.

The second strategy (action-continuity) uses $R_i@diff_{i-1}^{i}$ on f_i with respect to $diff_{i-1}^{i}$ between f_{i-1} and f_i and $R_{i+1}@diff_i^{i+1}$ on f_{i+1} with respect to $diff_i^{i+1}$ between f_i and f_{i+1}. This strategy leverages the fact that developer actions have continuity, for example, typing a sequence of chars, scrolling text continually. Therefore, it considers both the screen change $R_i@diff_{i-1}^{i}$ on f_i resulting from the previous action and the screen change $R_{i+1}@diff_i^{i+1}$ on f_{i+1} resulting from the current action. However, the second strategy does not consider $R_i@diff_i^{i+1}$ (the contrast of $R_{i+1}@diff_i^{i+1}$) on f_i.

The third strategy (change-contrast$\oplus$action-continuity) is a combination of the first and the second strategies. First, based on the corner positions of $R_i@diff_{i-1}^{i}$ on f_i and $R_{i+1}@diff_i^{i+1}$ on f_{i+1}, we determine the least screen region BR_i^{i+1} on f_i and f_{i+1} respectively that can include both $R_i@diff_{i-1}^{i}$ on f_i and $R_{i+1}@diff_i^{i+1}$ on f_{i+1}. This screen region BR_i^{i+1} on f_i will also include $R_i@diff_i^{i+1}$ on f_i (the contrast of $R_{i+1}@diff_i^{i+1}$). In addition, it may include some screen regions that are the same between f_i and f_{i+1}, which may provide additional context for action recognition.

Fig. 4 illustrates these three strategies. Assume the developer starts with the code in f_1, she types a "." which is recorded in f_2, and this action further triggers a "code completion popup window" recorded in f_3. We box $R_1@diff_1^2$ on f_1, $R_2@diff_1^2$ on f_2, $R_2@diff_2^3$ on f_2, $R_3@diff_2^3$ on f_3, BR_2^3 on f_2 and f_3. To recognize the "trigger popup" action from f_2 to f_3, the strategy-1 uses $R_2@diff_2^3$ on f_2 and $R_3@diff_2^3$ on f_3 as input change regions. The strategy-2 uses $R_2@diff_1^2$ on f_2 and $R_3@diff_2^3$ on f_3 as input. Note that although $R_2@diff_1^2$ is determined by the change region $diff_1^2$ between f_1 and f_2, we do not use any screen information from f_1. The strategy-3 uses BR_2^3 on f_2 and f_3 as input.

Note that the strategy-2 may take two screen regions of very different size. As CNN requires the input images to be of the same size, we have to resize the input change regions to the same size. However, in the situation illustrated in the strategy-2, resizing the two input change regions to the same size will distort the small-size image. In contrast, the strategy-3 does not suffer from this issue. As the two input change regions are of the same size, resizing will result in the same level of scaling of images.

2) CNN Architectures: Our model is based on Inception ResNet V2 [34], which combines the advantages of Microsoft's ResNet [35] and Inception architecture [36]. Each input change region is an image $I \in \mathcal{R}^{WHD}$ where W and H are the width and height of the image, and $D = 3$ for RGB color image (i.e., the red, green, blue channel respectively). Given the two change regions, we develop two architectures to extract image features. As shown in Fig. 5, early fusion architecture first concatenates the two change regions into a 6-channel input volume, which is fed into a single CNN to

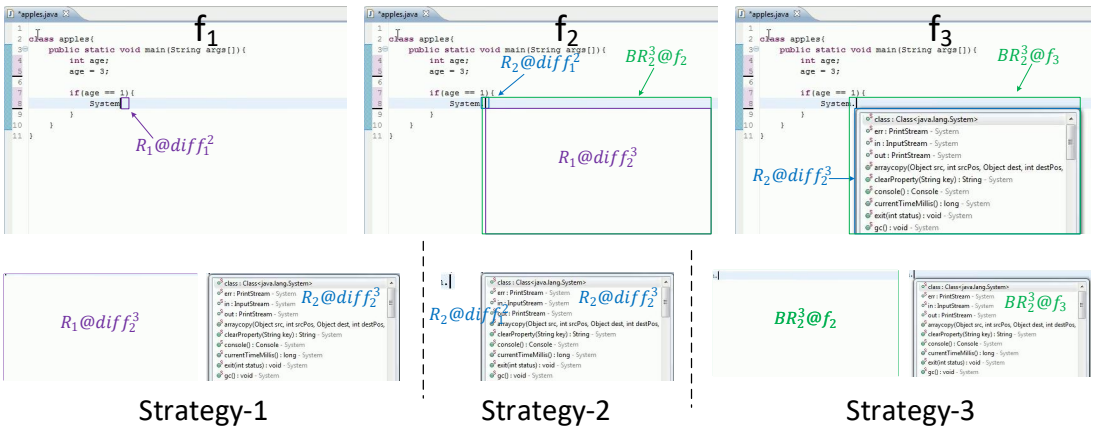

Fig. 4. Illustration of Three Strategies for Input Change Regions

extract image features. In contrast, late fusion architecture feeds each input region into a CNN separately and then concatenate the output feature vector of the two CNNs. It adopts a Siamese network architecture [33] in which the two CNNs share the weights.

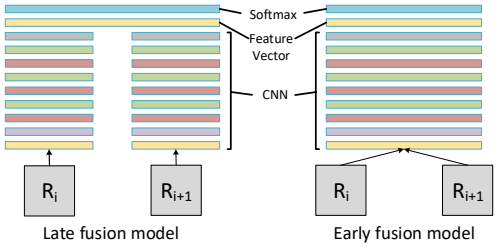

Fig. 5. Early Fusion versus Late Fusion Architecture

C. Action Recognition by Multi-Class Classification

Given the image feature vector S extracted by the CNN, we train a softmax classifier to predict the developer action that most likely causes the screen change from f_i to f_{i+1}. Specifically, the softmax classifier predicts the probability distribution $\hat{y}$ over the 10 class labels as defined in Section II, i.e., $\hat{y} = softmax(WS + b)$, where $\hat{y} \in \mathcal{R}^{10}$ is the vector of prediction probabilities over the 10 class labels, and W and b are the learnable parameters of the classifier. The CNN model and the softmax classifier is training by minimizing the cross-entropy loss of the predicted labels and the ground-truth labels: $L(\hat{y}, y) = -\sum_{1 \le i \le M} \sum_{1 \le j \le 10} (y_{ij} log(\hat{y}_{ij})$ where M is the number of training samples, y_{ij} is the ground-truth label for the jth class (1 for the ground-truth class, 0 otherwise) for the ith training example, and $\hat{y}_{ij}$ is the predicted probability of the jth class for the ith training example.

IV. Evaluation of Model Performance

We collect programming screencasts from Youtube to investigate the following three research questions:

- **RQ1**: How do the alternative designs of the vision CNN affect the model performance for action recognition?
- **RQ2**: How well do our action recognition model perform when training and testing across different developers, working environments and programming languages?
- **RQ3**: What is the runtime performance of our model?

A. The Dataset of Programming Screencasts on Youtube

In this study, we consider two programming languages: Python and Java. To prepare data, we use Youtube Data API [37] to search "java tutorial" and "python tutorial" on Youtube. We retrieve the top 50 returned playlists for Python and Java respectively. From these candidate playlists, we select 5 playlists for each language in which the video authors are programming in the screencasts. The authors of the selected 10 playlists are all different. We use Youtube-dl API [38] to download all the screencasts at High Definition resolution in these 10 selected playlists. Finally, we randomly select 5 screencasts from each playlist and obtain a collection of 50 screencasts (25 for Python and 25 for Java).

Table II shows the details of the programming screencasts we crawl. The selected playlists cover beginner, intermediate and advanced level of programming knowledge. From *Video Topic*, we can see that nine playlists cover the fundamental programming concepts and knowledge, and one playlist (P9) covers Java GUI. The tools used in screencasts are diverse. For Python, P1, P3 and P5 use the interactive shell, P4 uses IDE (PyCharm), and P2 uses both interactive shell and PyCharm. For Java, all five playlists use IDE (four use Eclipse and one (P9) uses NetBeans). Even for the same tool, different developers may use different color themes, font size, etc.

The selected screencasts have a duration between four to 15 minutes (median=7 minutes). Python and Java screencasts have almost the same total duration. The durations of the selected screencasts are appropriate for our study because they are long enough to contain adequate and diverse developer actions, but they do not contain much repetitive work which may inflate model performance superficially. The programming screencasts in a playlist as a whole can be regarded as about 30-40 minutes of programming work by a developer.

B. Manual Labeling of Developer Actions in Screencasts

In this work, we decode programming screencasts into frames at the rate of 15 frames per second by OpenCV [39]. As our model takes two consecutive frames as input, the model training and testing datasets are organized in the form of frame pairs. As explained in our problem statement (see Section II), we discard frame pairs with no screen changes (i.e., no action) by image differencing. This removes about 60% of frame

TABLE II
THE DATASET OF PROGRAMMING SCREENCASTS CRAWLED FROM YOUTUBE

Python						Java					
PL ID	**PL Name**	**Tools**	**Video ID**	**Video Topic**	**Dur(s)**	**PL ID**	**PL Name**	**Tools**	**Video ID**	**Video Topic**	**Dur(s)**
P1	Python Programming Tutorials	Interactive Shell	V1	bitwise operation	420	P6	Java Tutorial for Beginners	Eclipse	V1	variables	597
			V2	variables	259				V2	input	730
			V3	lists	450				V3	switch case	577
			V4	dictionaries	382				V4	while	408
			V5	arithmetic	323				V5	string	534
P2	Python 3.4 Programming Tutorials	Interactive Shell & PyCharm	V1	numbers	329	P7	Java (Beginner) Programming Tutorials	Eclipse	V1	variables	445
			V2	string	505				V2	input	331
			V3	lists	465				V3	if	362
			V4	if else	552				V4	switch	407
			V5	for	429				V5	classes	394
P3	Python Programming Tutorials	Interactive Shell	V1	numbers	340	P8	Java (Intermediate) Tutorials	Eclipse	V1	array	360
			V2	variables	385				V2	stack	342
			V3	strings	383				V3	queue	337
			V4	dictionaries	373				V4	hashset	287
			V5	for & while	337				V5	return	365
P4	Python Programming Tutorials	PyCharm	V1	while	399	P9	Java GUI Tutorials	NetBeans	V1	image	465
			V2	functions	394				V2	event	496
			V3	dictionaries	778				V3	numbers	445
			V4	bitwise operation	588				V4	beeper	527
			V5	if else	378				V5	grid layout	295
P5	Python Tutorial for Beginners	Interactive Shell	V1	numbers	542	P10	Java Tutorial for Beginners 2018	Eclipse	V1	variables	516
			V2	variables	608				V2	if else	418
			V3	models functions	641				V3	while	486
			V4	string	756				V4	arithmetic	545
			V5	lists	756				V5	class	559

pairs in the initial dataset. We then manually label each frame pair with screen changes by one of the 10 classes of actions defined in Section II. Take Fig. 1 as example. The frame pair (f_x, f_{x+1}) is labeled as "trigger popup (C6)", (f_y, f_{y+1}) as "select text (C8)", and (f_z, f_{z+1}) as "scroll text (C7)".

To ensure the quality of data labeling, the two authors and another developer participate in the data labeling. The annotators have at least 3 years programming experience on Python and Java. For efficient and consistent labeling, we develop a Python application by which the annotators can view frame pairs with screen changes in a screencast one by one and select a class label for each frame pair. Each annotator first labels the whole dataset independently. The Fleiss' kappa of the three annotators' labeling results is 0.76 which indicates substantial agreement. If two or three annotators assign the same label to a frame pair, that label is the final label. In the cases when the three annotators give three different labels for a frame pair, the annotators discuss to decide the final label.

Table III summarizes the distribution of different classes of developer actions out of our manual labeling process. This labeled dataset consists of 73725 frame pairs in total, which requires significant human efforts (about 3 man-months). We can see that "scroll text" and "switch window" have fewer instances than other classes. This is because programming screencasts on Youtube usually do not involve very long code to scroll or many files to switch. Furthermore, we observe that the action distributions for Python and Java are largely similar. But Java has relatively more "trigger popups" and "switch window". This is because all five Java screencasts use IDEs while 4 of 5 Python screencasts use interactive shell (with no popup support or fewer windows to switch).

C. Evaluation Metrics

We evaluate and compare model performance for action recognition by four metrics: Accuracy, Precision, Recall and F1-score. The correctness of the predicted action for a frame pair is determined against the human label of developer action for that frame pair (i.e., ground truth). Precision for an action class C is the proportion of frame pairs that are correctly predicted as C among all frame pairs predicted as C. Recall for an action class C the proportion of frame pairs that are correctly predicted as C among all ground-truth frame pairs labelled as C. F1-score for an action class C combines the precision and recall as $2 \times (Precision_C \times Recall_C)/(Precision_C + Recall_C)$. As multiple action labels are predicted by our model, we compute the weighted average of precision, recall and F1-score for all action classes as a whole (i.e., $\sum_{c=1}^{10} (metric * count_c) \,/\, \sum_{c=1}^{10} count_c$), which gives a view on the general prediction performance. We also calculate accuracy to evaluate the overall performance, i.e., the number of frame pairs correctly predicted by the model over the total number of frame pairs.

TABLE III
STATISTICS OF DEVELOPER ACTIONS BY MANUAL LABELING

Action Class	**Python**	**Java**	**All**
Move cursor by keyboard (C1)	10281	9714	19995
Move mouse over text region(C2)	11589	12321	23910
Move mouse over non-text region (C3)	4098	3723	7821
Enter text (C4)	3642	3264	6906
Delete text (C5)	1890	1671	3561
Trigger popups (C6)	1059	3831	4890
Scroll text (C7)	990	1122	2112
Select text (C8)	1539	1488	3027
Switch window (C9)	558	945	1503
Total	**35646**	**38079**	**73725**

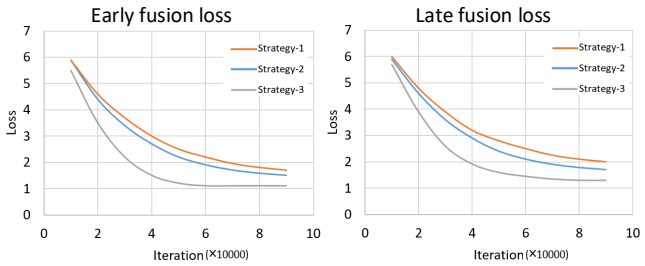

Fig. 6. Loss Convergence for Different Model Configurations

D. Impact of Alternative Model Design (RQ1)

Motivation: Our approach relies on the CNN model to extract image features for action recognition. The effectiveness of the CNN for feature extraction directly affect the recognition performance. In this work, we have three alternative strategies for preparing input change regions to the CNN: change-contrast (Strategy-1), action-continuity (Strategy-2), and change-contrast⊕action-continuity (Strategy-3), which exploit different properties of developer actions and screen changes. In addition, we extract image features by the two different CNN architectures: early fusion versus late fusion. We want to comparatively investigate the impact of these alternative model designs on the performance of action recognition.

Method: In this experiment, we combine Python and Java data in Table III. We use 80% of frame pairs and their corresponding action labels for model training and the rest 20% for testing. We combine each strategy for input change regions and each CNN architecture. As such, we have six different models. We train each model separately using the same training data and compare the model performance on the same testing data.

Results: Fig. 6 shows the loss convergence during the model training process. The horizontal axis is the number of training iterations and the vertical axis shows the loss value after each training iteration. We can see that the same input strategy has similar loss convergence rate in early-fusion and late-fusion architecture, but early-fusion architecture converges a bit faster than late-fusion architecture. In both CNN architectures, the change-contrast⊕action-continuity (InputStrategy-3) converges faster than the other two input strategies.

Table IV and Table V show the performance results of the six model configurations. We can see that using the input Strategy-3 in both early-fusion and late-fusion architecture achieves the much better performance in all evaluation metrics, compared with the other two input strategies. The average F1-score of the 10 action classes for the input Strategy-3 is 0.70 and 0.73 in early-fusion and late-fusion architecture, respectively. The average F1-score for the other two input strategies is only about 0.5. The input Strategy-1 and Strategy-2 have very similar performance. They achieve acceptable performance only for "move cursor" and "move mouse over text" (F1-score around 0.7) which have the most number of training samples. For the input Strategy-1 and Strategy-2, the F1-scores for most other action classes are around or below 0.5. In contrast, for the input Strategy-3, the F1-scores for most action classes are around or above 0.7.

Comparing the same input strategy in different CNN architectures, we can see that the average F1-score of the 10 classes is very close for the input Strategy-2 and Strategy-3. The performance gap for input Strategy-1 is relatively larger. In general, late-fusion architecture performs better than early-fusion architecture, especially for those action classes with small numbers of data instances. However, late-fusion has to execute the CNN twice for the two input change regions, which doubles the computing time (see Section IV-F), compared with early fusion that concatenates the two change regions and feeds them as a whole through the CNN once.

Considering both change contrast and action continuity in the input change regions is beneficial for action recognition, compared with considering change contrast or action continuity alone. Early fusion and late fusion have very close performance for action recognition, but late fusion requires double computing time.

E. Model Performance Across Different Settings (RQ2)

Motivation: An effective action recognition model should be able to generalize over variations within one class and variations across developers, working environments (e.g., tools, color themes) and programming languages. This RQ is set to evaluate the performance of our model across different developers, working environments and programming languages.

Approach: Considering the experiment results of RQ1, our model in RQ2 uses change-contrast⊕action-continuity as input strategy and early-fusion as CNN architecture. First, we conduct 10 intra-playlist experiments. We randomly divide the human-labeled frame pairs of each playlist into 80% for training and 20% for testing. As the screencasts in a playlist are produced by the same developer in the same working environment, the intra-playlist experiments set the performance upper bound to compare and understand the inter-playlist and inter-programming-language performance.

For inter-playlist experiments, we use the human-labeled frame pairs of four playlists in a programming language as training data and the frame pairs of the left one playlist as testing data. So we have 10 inter-playlist experiments. Considering the data characteristics of our crawled playlists (see Section IV-A), these inter-playlist experiments can test our model performance across different developers and/or working environments. For inter-language experiments, we train the model using the human-labeled frame pairs of the five playlists of one language and test the model using the data of the other language. So we have two inter-language experiments (denoted as Python→Java and Java→Python). In the inter-language setting, both developers and working environments are also different between model training and testing.

Results: Next, we report our intra-playlist, inter-playlist and inter-language experiment results:

1) Intra-playlist: Table VI shows our model's performance in the 10 intra-playlist experiments. We show the accuracy and the average precision, recall and F1-score for each playlist.

TABLE IV
PERFORMANCE OF THREE INPUT STRATEGIES WITH EARLY FUSION ARCHITECTURE

	Strategy-1			Strategy-2			Strategy-3		
Action Class	**Precision**	**Recall**	**F1-score**	**Precision**	**Recall**	**F1-score**	**Precision**	**Recall**	**F1-score**
Move cursor by keyboard (C1)	0.65	0.78	0.71	0.68	0.73	0.70	0.88	0.86	0.87
Move mouse over text region(C2)	0.79	0.59	0.67	0.81	0.63	0.71	0.84	0.84	0.84
Move mouse over non-text region (C3)	0.31	0.63	0.41	0.33	0.70	0.45	0.71	0.78	0.74
Enter text (C4)	0.73	0.42	0.53	0.54	0.50	0.52	0.77	0.86	0.81
Delete text (C5)	0.45	0.24	0.31	0.41	0.33	0.36	0.67	0.71	0.69
Trigger popups (C6)	0.43	0.31	0.36	0.50	0.50	0.50	0.71	0.54	0.61
Scroll text (C7)	0.18	0.24	0.20	0.40	0.18	0.25	0.66	0.40	0.50
Select text (C8)	0.55	0.38	0.45	0.49	0.30	0.37	0.77	0.50	0.60
Switch window (C9)	0.17	0.61	0.26	0.41	0.27	0.32	0.53	0.61	0.56
Others (C10)	0.34	0.51	0.41	0.53	0.47	0.50	0.69	0.66	0.67
Average	**0.39**	**0.52**	**0.44**	**0.54**	**0.49**	**0.51**	**0.71**	**0.68**	**0.70**
Accuracy	**0.59**			**0.63**			**0.81**		

TABLE V
PERFORMANCE OF THREE INPUT STRATEGIES WITH LATE FUSION ARCHITECTURE

	Strategy-1			Strategy-2			Strategy-3		
Action Class	**Precision**	**Recall**	**F1-score**	**Precision**	**Recall**	**F1-score**	**Precision**	**Recall**	**F1-score**
Move cursor by keyboard (C1)	0.67	0.71	0.69	0.71	0.72	0.71	0.85	0.83	0.84
Move mouse over text region(C2)	0.74	0.60	0.66	0.72	0.61	0.66	0.87	0.85	0.86
Move mouse over non-text region (C3)	0.49	0.52	0.50	0.46	0.45	0.45	0.81	0.83	0.82
Enter text (C4)	0.49	0.40	0.44	0.53	0.51	0.52	0.81	0.84	0.82
Delete text (C5)	0.66	0.61	0.63	0.61	0.54	0.57	0.65	0.78	0.71
Trigger popups (C6)	0.50	0.46	0.48	0.57	0.38	0.45	0.75	0.83	0.79
Scroll text (C7)	0.45	0.40	0.42	0.47	0.41	0.44	0.75	0.61	0.67
Select text (C8)	0.65	0.47	0.54	0.65	0.46	0.54	0.67	0.80	0.78
Switch window (C9)	0.50	0.38	0.43	0.52	0.38	0.44	0.67	0.69	0.68
Others (C10)	0.41	0.63	0.49	0.43	0.58	0.49	0.74	0.68	0.70
Average	**0.45**	**0.62**	**0.52**	**0.47**	**0.58**	**0.51**	**0.75**	**0.70**	**0.73**
Accuracy	**0.60**			**0.62**			**0.82**		

Our model performs very well for intra-playlist prediction, with the mean F1-score 0.88 ± 0.016. We observe very small performance differences between the playlists.

2) Inter-playlist: Table VII shows our model's performance on each testing playlist in the 10 inter-playlist experiments. We can see that our model still has very good performance for inter-playlist prediction, with the mean F1-score 0.79 ± 0.084. Compared with the intra-playlist performance, the inter-playlist performance is inevitably lower, which is unsurprising considering the variations across developers and/or working environments. The performance is especially low for the testing playlist *P1* and *P9*. *P1* and *P9* have very different working environments from other playlists. For example, the mouse pointer over non-text region in *P1* is a very unique blue circle, which never appears in the training playlist (P2/P3/P4/P5). This results in the very poor precision and recall for "move mouse over nontext" class. $P9$ is recorded with unusual dark IDE theme which leads to very different screen features from the IDEs used in the training playlists (P6/P7/P8/P10). This affects the prediction of most action classes.

3) Inter-language: Table VIII shows the model performance in the two inter-language experiments. The inter-language setting is even more challenging as it involves not only language variations but also developer and working environment variations. In this challenging setting, our model has reasonably good performance (average F1-score 0.59 for Python→Java and 0.68 for Java→Python). However, the model does not have equally-good performance on all action classes. For example, the model trained by Java screencasts cannot recognize "select text" well in Python screencasts, but the model trained by Python data performs reasonably well for recognizing "select text" in Java data. By analyzing the data, we find that "select text" in Python video is more variable than that in Java video. "select text" in Java video only has blue background, but "select text" in Python video has both blue and gray background. Furthermore, neither model performs well for "switch window" and "scroll text", because these two action classes have much fewer train samples than other classes.

Our model can be effectively trained and deployed in very diverse developer, working environment and programming language settings. Exposing the model to diverse training data is crucial for good model performance.

TABLE VI
INTRA PLAYLIST RESULTS

Playlist ID	**Precision**	**Recall**	**F1-score**	**Accuracy**
P1	0.88	0.90	0.89	0.88
P2	0.90	0.87	0.88	0.89
P3	0.88	0.90	0.89	0.90
P4	0.90	0.88	0.89	0.89
P5	0.90	0.85	0.87	0.87
P6	0.87	0.85	0.86	0.87
P7	0.85	0.83	0.84	0.86
P8	0.90	0.90	0.90	0.90
P9	0.86	0.90	0.88	0.89
P10	0.90	0.87	0.88	0.90
mean±stddev	**0.89±0.018**	**0.88±0.024**	**0.88±0.016**	**0.88±0.013**

F. Runtime Performance (RQ3)

We test our tool for action recognition on a PC with 64G RAM, i9-7900x CPU and Titan Xp GPU. The CNN model is implemented in TensorFlow [40]. Using early fusion architecture, our tool can process about 8-10 frame pairs per second. Using late fusion architecture, our tool can process

TABLE VII
INTER PLAYLIST RESULTS

Playlist ID	Precision	Recall	F1-score	Accuracy
P1	0.71	0.73	0.72	0.73
P2	0.83	0.81	0.82	0.83
P3	0.85	0.81	0.83	0.85
P4	0.86	0.83	0.84	0.85
P5	0.85	0.86	0.85	0.85
P6	0.87	0.84	0.85	0.86
P7	0.81	0.85	0.83	0.84
P8	0.83	0.79	0.81	0.83
P9	0.66	0.50	0.57	0.67
P10	0.83	0.85	0.84	0.84
mean±stddev	**0.80±0.065**	**0.79±0.102**	**0.79±0.084**	**0.82±0.059**

TABLE VIII
INTER PROGRAMMING LANGUAGE RESULTS

	Python→Java			Java→Python		
Action	**Precision**	**Recall**	**F1**	**Precision**	**Recall**	**F1**
C1	0.88	0.90	0.89	0.86	0.84	0.85
C2	0.78	0.85	0.81	0.85	0.81	0.83
C3	0.62	0.70	0.66	0.58	0.80	0.67
C4	0.68	0.89	0.77	0.77	0.70	0.73
C5	0.48	0.73	0.58	0.56	0.50	0.53
C6	0.80	0.51	0.62	0.67	0.73	0.70
C7	0.43	0.52	0.47	0.31	0.62	0.41
C8	0.88	0.58	0.70	0.54	0.60	0.57
C9	0.52	0.44	0.47	0.46	0.55	0.50
C10	0.68	0.46	0.54	0.58	0.78	0.67
Average	**0.69**	**0.52**	**0.59**	**0.61**	**0.78**	**0.68**
Accuracy	**0.74**			**0.78**		

about 4-5 frame pairs per second. Image differencing accounts for about 90% of processing time. Our results show that our tool is fast enough for real-time developer action recognition.

V. PRACTICAL APPLICATION: ACTION-AWARE KEY-CODE FRAME EXTRACTION

Having evaluated the performance of our model for action recognition, we want to demonstrate a practical application that our approach enables for programming screencast analysis. A key challenge in programming screencast analysis is to determine the key frames from which important code should be extracted. In this study, we compare action-aware extraction of key-code frames based on recognized developer actions in a screencast with action-agnostic extraction of key-code frames commonly used in existing work.

TABLE IX
COMPARISON OF KEY-CODE FRAME EXTRACTION METHODS

Method	TP	FP	TN	FN	Pre.	Rec.	F1
Fixed time interval	24	36769	514790	312	0.0006	0.07	0.0013
Image similarity	336	2143	549516	0	0.13	1.00	0.24
ActionNet-based	316	91	551468	20	0.77	0.94	0.85

A. Screencast Dataset of Real Developer Work

We use screencast software to record about 10 hours of real programming work of the two software developers. One developer works on a Java project in Eclipse and the other developer works on a Python project in Jupyter (an online Python development environment). The screencasts are decoded at the rate of 15 frames per second. We ask the two developers to identify the key-code frames in their screencasts. The two developers define key-code frames as the frames that contain code fragments before or after significant code changes, or the frames that contain code fragments that may not be visible again. They follow three behavioral patterns to identify key-code frames: 1) Selects a block of code and deletes it, which means that an old version of the code in the frame before code deletion should be extracted. 2) Edits code and then switches to another window, which means that code editing is temporarily finished and the latest version of the code in the frame before window switching should be extracted. 3) Scrolls code and both the disappearing code in the frame before scrolling and the appearing code in the frame after scrolling should be extracted. In total, 336 key-code frames have been identified which are used as ground-truth to compare different key-code frame extraction methods.

B. Methods for Extracting Key-Code Frames

We train our model (change-contrast⊕action-continuity with early-fusion architecture) using the dataset of Python and Java programming screencasts from Youtube (see Section IV-A). Then, we use the trained model to recognize the actions in the 10 hours of screencasts of real developer work. Based on the recognized actions, we identify the key-code frames in the screencasts of real developer work by searching certain sequential patterns of developer actions. We compare our method with two action-agonistic methods commonly used in existing work on programming screencast analysis [11], [21], [30]. The first method extracts the first frame at a fix time interval. Following the work [30], we set the time interval at 1 second. The second method extracts the next frame that is different enough (below a user-specified similarity threshold) from the previous frame. Following the work [26], we set the similarity threshold at 0.95.

C. Results

Table IX presents the comparison results. We present True Positive (TP), True Negative (TN), False Positive (FP) and False Negative (FN) for the three methods. The fixed-time-interval based method is completely unaware of the workflow and content of the screencasts. It outputs too many frames (TP+FP=36793), among which only 24 frames accidentally hit the ground-truth key-code frames. So the fixed-time-interval based method has very low precision and recall. It will waste so much computing resource to try to extract code from so many unnecessary frames.

Frame-similarity based method significantly decreases the number of extracted frames, but it still extracts 2479 (TP+FP) frames. As the ground-truth key-code frames identified by the developers all involve substantial screen changes, frame-similarity based method actually identifies all ground-truth frames at the similarity threshold 0.95 (i.e., recall=1 in our experiment). However, this result may be sensitive to the scale of screen changes and the similarity threshold. Furthermore, frame-similarity based method still identifies many false positive frames (FP=2143), for example, frames with popups and text selection. Although these frames have substantial screen changes, they do not change the code and do not need to be

repeatedly extracted if the key-code frames have already been extracted. Furthermore, screen content in such frames is noisy which will negatively affect the quality of OCR.

ActionNet-based method extracts the least amount of frames (TP+FP=407), only about 16% of the number of frames extracted by frame-similarity based method. But ActionNet-based method has only a minor decrease in recall, compared with the recall of frame-similarity based method (0.94 versus 1.00). ActionNet-based method still extracts some unnecessary frames (FP=91), but its precision is about six times higher than that of frame-similarity based method (0.77 versus 0.13). The main reason for the false positive frames by ActionNet-based method is the coarse-grained definition of action classes. For example, both "enter or delete a word" and "enter or delete several lines of code" are now recognized as "enter or delete text". So some frames with minor code changes are extracted, but the developers do not consider such changes significant enough so that they do not label such frames as key-code frames. Such false positive frames could be filtered out by distinguishing finer-grained actions.

VI. Related Work

Developer behavioral research: In behavioral research, data collection methods include observation, survey and interview. Observational data can be collected by using eyetracker and fMRI [41]–[43], software instrumentation [5], [44], [45], or screencasting [11], [26]. Among these observational data collection methods, screencasting is the easiest one to deploy. However, due to the image nature of screencasts, the workflow actions and application content in screencasts must be extracted before any meaningful behavioral research can be conducted. Our work develops the first tool to automatically extract workflow actions from programming screencasts. Our tool can lower the barrier for action-centric analysis of programming screencasts [4], enabling large-scale behavioral research in software engineering.

Programming screencast analysis: Existing methods falls into two categories: content extraction [11], [46]–[51] and video search and navigation [8], [21], [27], [29]–[31]. Content extraction in existing work is simply based on fix time interval or frame similarity. Some tools like Waken [26] use image differencing technique to identify UI elements (e.g., mouse pointer) but not actions (e.g., move the mouse over text). VT-Revolution by Bao et al. [8] shows that workflow actions in a programming screencast, if available, can significantly improve video search and navigation efficiency and enhance the learning experience. However, it uses software instrumentation to collect workflow actions during screencasting. It envisions to support the interactive video watching experience for Youtube programming screencasts that are not accompanied with workflow actions. Our tool is the first step towards making this vision closer to reality.

General human action recognition: Our work recognizes developer actions in the virtual world, while general human action recognition recognizes human actions in the physical world. Early techniques for human action recognition include hidden Markov model [52] and discriminative SVM models [53]. Recent work has used deep learning techniques, such as two stream CNNs [23], C3D (3D convolutional neural network) [54]. Considering the duration of physical human actions, these techniques usually analyze multiple frames (e.g. 16 frames in C3D). However, developer actions on computer cause instant screen changes. As such, developer actions must be recognized from the screen change between two frames. The accuracy of our technique is on par with that of human action recognition [23]–[25]. Human action recognition enables many applications, such as action-centric video search [55], [56], automatic surveillance [57], [58], smart homes [59], [60]. These applications inspire downstream applications based on the recognition of developer actions in screencasts using our technique, such as developer risk behavior surveillance. Existing tools (e.g., CheckStyle, FindBugs) are code-centric. None of them can prevent programming mistakes from behavioral perspectives.

Deep learning for software data: Recently, deep learning techniques have been successfully applied to many forms of software data, such as source code and software text [61]–[66], and user interface images [67]–[69]. Different from these works, our work is the first to apply deep learning to recognize workflow actions in programming screencasts. A recent work [50] uses CNN-based techniques to predict programming languages used in screencasts, which is a much easier task than our developer action recognition.

VII. Conclusions and Future Work

This paper fills in an important missing technique in the tool set for programming screencast analysis. The core component of our technique is a CNN model. This design is driven by the CNN's ability to automatically learn to extract image features from the screen changes resulting from developer actions, thus removing the need for manual feature engineering which is a challenging task due to the diversity of developer actions, working environments and programming languages. Our experiments show that our technique can generalize over variations within action classes and variations across developers, working environments and programming languages.

This work develops an enabling technique for action-aware analysis of programming screencasts (e.g., key-code frame extraction). In this future, we will build a big database of developers' workflow actions, considering millions of programming screencasts on Youtube. Such a database will enable much downstream research work which we will investigate, such as large-scale behavioral research in software engineering, action-aware search and navigation of Youtube programming screencasts, or developer risk behavior surveillance for proactively avoiding programming mistakes.

VIII. Acknowledgment

We gratefully acknowledge the support of NVIDIA Corporation with the donation of the Titan Xp GPU used for this research.

REFERENCES

[1] J. Wang, X. Peng, Z. Xing, and W. Zhao, "An exploratory study of feature location process: Distinct phases, recurring patterns, and elementary actions," in *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*. IEEE, 2011, pp. 213–222.

[2] H. Li, Z. Xing, X. Peng, and W. Zhao, "What help do developers seek, when and how?" in *Reverse Engineering (WCRE), 2013 20th Working Conference on*. IEEE, 2013, pp. 142–151.

[3] X. Xia, L. Bao, D. Lo, Z. Xing, A. E. Hassan, and S. Li, "Measuring program comprehension: A large-scale field study with professionals," *IEEE Transactions on Software Engineering*, 2017.

[4] L. Bao, Z. Xing, X. Xia, D. Lo, and A. E. Hassan, "Inference of development activities from interaction with uninstrumented applications," *Empirical Software Engineering*, vol. 23, no. 3, pp. 1313–1351, 2018.

[5] L. Bao, D. Ye, Z. Xing, X. Xia, and X. Wang, "Activityspace: a remembrance framework to support interapplication information needs," in *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 2015, pp. 864–869.

[6] A. J. Ko, H. Aung, and B. A. Myers, "Eliciting design requirements for maintenance-oriented ides: a detailed study of corrective and perfective maintenance tasks," in *Proceedings of the 27th international conference on Software engineering*. ACM, 2005, pp. 126–135.

[7] L. Ponzanelli, G. Bavota, A. Mocci, R. Oliveto, M. Di Penta, S. C. Haiduc, B. Russo, and M. Lanza, "Automatic identification and classification of software development video tutorial fragments," *IEEE Transactions on Software Engineering*, no. 1, pp. 1–1, 2017.

[8] L. Bao, Z. Xing, X. Xia, and D. Lo, "Vt-revolution: Interactive programming video tutorial authoring and watching system," *IEEE Transactions on Software Engineering*, 2018.

[9] M. Hilton and A. Begel, "A study of the organizational dynamics of software teams," in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*. ACM, 2018, pp. 191–200.

[10] X. Xia, L. Bao, D. Lo, P. S. Kochhar, A. E. Hassan, and Z. Xing, "What do developers search for on the web?" *Empirical Software Engineering*, vol. 22, no. 6, pp. 3149–3185, 2017.

[11] L. Bao, J. Li, Z. Xing, X. Wang, X. Xia, and B. Zhou, "Extracting and analyzing time-series hci data from screen-captured task videos," *Empirical Software Engineering*, vol. 22, no. 1, pp. 134–174, 2017.

[12] J. Lawrance, C. Bogart, M. Burnett, R. Bellamy, K. Rector, and S. D. Fleming, "How programmers debug, revisited: An information foraging theory perspective," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 197–215, 2013.

[13] J. Sillito, K. De Voider, B. Fisher, and G. Murphy, "Managing software change tasks: An exploratory study," in *Empirical Software Engineering, 2005. 2005 International Symposium on*. IEEE, 2005, pp. 10–pp.

[14] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks," *IEEE Transactions on software engineering*, vol. 32, no. 12, pp. 971–987, 2006.

[15] D. Piorkowski, S. D. Fleming, C. Scaffidi, L. John, C. Bogart, B. E. John, M. Burnett, and R. Bellamy, "Modeling programmer navigation: A head-to-head empirical evaluation of predictive models," in *Visual Languages and Human-Centric Computing (VL/HCC), 2011 IEEE Symposium on*. IEEE, 2011, pp. 109–116.

[16] T. Fritz, D. C. Shepherd, K. Kevic, W. Snipes, and C. Bräunlich, "Developers' code context models for change tasks," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 7–18.

[17] A. J. Ko, H. H. Aung, and B. A. Myers, "Design requirements for more flexible structured editors from a study of programmers' text editing," in *CHI'05 extended abstracts on human factors in computing systems*. ACM, 2005, pp. 1557–1560.

[18] P. Dewan, P. Agarwal, G. Shroff, and R. Hegde, "Distributed side-by-side programming," in *Proceedings of the 2009 ICSE workshop on cooperative and human aspects on software engineering*. IEEE Computer Society, 2009, pp. 48–55.

[19] A. Hurst, S. E. Hudson, and J. Mankoff, "Automatically identifying targets users interact with during real world tasks," in *Proceedings of the 15th international conference on Intelligent user interfaces*. ACM, 2010, pp. 11–20.

[20] T.-H. Chang, T. Yeh, and R. Miller, "Associating the visual representation of user interfaces with their internal structures and metadata," in *Proceedings of the 24th annual ACM symposium on User interface software and technology*. ACM, 2011, pp. 245–256.

[21] L. Ponzanelli, G. Bavota, A. Mocci, M. Di Penta, R. Oliveto, B. Russo, S. Haiduc, and M. Lanza, "Codetube: extracting relevant fragments from software development video tutorials," in *Proceedings of the 38th International Conference on Software Engineering Companion*. ACM, 2016, pp. 645–648.

[22] S. Mori, H. Nishida, and H. Yamada, *Optical character recognition*. John Wiley & Sons, Inc., 1999.

[23] K. Simonyan and A. Zisserman, "Two-stream convolutional networks for action recognition in videos," in *Advances in neural information processing systems*, 2014, pp. 568–576.

[24] J. Donahue, L. Anne Hendricks, S. Guadarrama, M. Rohrbach, S. Venugopalan, K. Saenko, and T. Darrell, "Long-term recurrent convolutional networks for visual recognition and description," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 2625–2634.

[25] S. Ji, W. Xu, M. Yang, and K. Yu, "3d convolutional neural networks for human action recognition," *IEEE transactions on pattern analysis and machine intelligence*, vol. 35, no. 1, pp. 221–231, 2013.

[26] N. Banovic, T. Grossman, J. Matejka, and G. Fitzmaurice, "Waken: reverse engineering usage information and interface structure from software videos," in *Proceedings of the 25th annual ACM symposium on User interface software and technology*. ACM, 2012, pp. 83–92.

[27] C. Nguyen and F. Liu, "Making software tutorial video responsive," in *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*. ACM, 2015, pp. 1565–1568.

[28] A. Mahendran and A. Vedaldi, "Understanding deep image representations by inverting them," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 5188–5196.

[29] T.-J. K. P. Monserrat, S. Zhao, K. McGee, and A. V. Pandey, "Notevideo: facilitating navigation of blackboard-style lecture videos," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2013, pp. 1139–1148.

[30] K. Khandwala and P. J. Guo, "Codemotion: expanding the design space of learner interactions with computer programming tutorial videos," in *Proceedings of the Fifth Annual ACM Conference on Learning at Scale*. ACM, 2018, p. 57.

[31] S. Pongnumkul, M. Dontcheva, W. Li, J. Wang, L. Bourdev, S. Avidan, and M. F. Cohen, "Pause-and-play: automatically linking screencast video tutorials with applications," in *Proceedings of the 24th annual ACM symposium on User interface software and technology*. ACM, 2011, pp. 135–144.

[32] https://scikit-image.org, august 24, 2018.

[33] G. Koch, R. Zemel, and R. Salakhutdinov, "Siamese neural networks for one-shot image recognition," in *ICML Deep Learning Workshop*, vol. 2, 2015.

[34] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, "Inception-v4, inception-resnet and the impact of residual connections on learning." in *AAAI*, vol. 4, 2017, p. 12.

[35] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[36] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 2818–2826.

[37] https://developers.google.com/youtube/v3/, august 24, 2018.

[38] https://rg3.github.io/youtube-dl/, august 24, 2018.

[39] https://opencv.org, august 24, 2018.

[40] https://www.tensorflow.org, august 24, 2018.

[41] P. Rodeghero, C. McMillan, P. W. McBurney, N. Bosch, and S. D'Mello, "Improving automated source code summarization via an eye-tracking study of programmers," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 390–401.

[42] T. R. Shaffer, J. L. Wise, B. M. Walters, S. C. Müller, M. Falcone, and B. Sharif, "itrace: Enabling eye tracking on software artifacts within the ide to support software engineering tasks," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 954–957.

[43] J. Siegmund, C. Kästner, S. Apel, C. Parnin, A. Bethmann, T. Leich, G. Saake, and A. Brechmann, "Understanding understanding source

code with functional magnetic resonance imaging," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 378–389.

[44] D. M. Hilbert and D. F. Redmiles, "Extracting usability information from user interface events," *ACM Computing Surveys (CSUR)*, vol. 32, no. 4, pp. 384–421, 2000.

[45] J. H. Kim, D. V. Gunn, E. Schuh, B. Phillips, R. J. Pagulayan, and D. Wixon, "Tracking real-time user experience (true): a comprehensive instrumentation solution for complex systems," in *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*. ACM, 2008, pp. 443–452.

[46] J. Kim, P. J. Guo, C. J. Cai, S.-W. D. Li, K. Z. Gajos, and R. C. Miller, "Data-driven interaction techniques for improving navigation of educational videos," in *Proceedings of the 27th annual ACM symposium on User interface software and technology*. ACM, 2014, pp. 563–572.

[47] L. Ponzanelli, G. Bavota, A. Mocci, M. Di Penta, R. Oliveto, M. Hasan, B. Russo, S. Haiduc, and M. Lanza, "Too long; didn't watch!: extracting relevant fragments from software development video tutorials," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 261–272.

[48] J. Escobar-Avila, E. Parra, and S. Haiduc, "Text retrieval-based tagging of software engineering video tutorials," in *Software Engineering Companion (ICSE-C), 2017 IEEE/ACM 39th International Conference on*. IEEE, 2017, pp. 341–343.

[49] S. Yadid and E. Yahav, "Extracting code from programming tutorial videos," in *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. ACM, 2016, pp. 98–111.

[50] J. Ott, A. Atchison, P. Harnack, A. Bergh, and E. Linstead, "A deep learning approach to identifying source code in images and video," 2018.

[51] P. Moslehi, B. Adams, and J. Rilling, "Feature location using crowd-based screencasts," 2018.

[52] J. Yamato, J. Ohya, and K. Ishii, "Recognizing human action in time-sequential images using hidden markov model," in *Computer Vision and Pattern Recognition, 1992. Proceedings CVPR'92., 1992 IEEE Computer Society Conference on*. IEEE, 1992, pp. 379–385.

[53] J. C. Niebles, C.-W. Chen, and L. Fei-Fei, "Modeling temporal structure of decomposable motion segments for activity classification," in *European conference on computer vision*. Springer, 2010, pp. 392–405.

[54] D. Tran, L. Bourdev, R. Fergus, L. Torresani, and M. Paluri, "Learning spatiotemporal features with 3d convolutional networks," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 4489–4497.

[55] D. Gerónimo and H. Kjellström, "Unsupervised surveillance video retrieval based on human action and appearance," in *Pattern Recognition (ICPR), 2014 22nd International Conference on*. IEEE, 2014, pp. 4630–4635.

[56] L. Shao, S. Jones, and X. Li, "Efficient search and localization of human actions in video databases," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 24, no. 3, pp. 504–512, 2014.

[57] A. Meidan and R. B. Sella, "Automatic video surveillance system and method," Feb. 2 2016, uS Patent 9,253,453.

[58] S. Pushparaj and S. Arumugam, "Using 3d convolutional neural network in surveillance videos for recognizing human actions." *Int. Arab J. Inf. Technol.*, vol. 15, no. 4, pp. 693–700, 2018.

[59] B. M. H. Alhafidh, A. I. Daood, and W. H. Allen, "Comparison of classifiers for prediction of human actions in a smart home," in *Internet-of-Things Design and Implementation (IoTDI), 2018 IEEE/ACM Third International Conference on*. IEEE, 2018, pp. 287–288.

[60] B. Alhafidh and W. Allen, "Design and simulation of a smart home managed by an intelligent self-adaptive system," *International Journal of Engineering Research and Applications*, vol. 6, no. 8, pp. 64–90, 2016.

[61] B. Xu, D. Ye, Z. Xing, X. Xia, G. Chen, and S. Li, "Predicting semantically linkable knowledge in developer online forums via convolutional neural network," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2016, pp. 51–62.

[62] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2016, pp. 87–98.

[63] Q. Luo, D. Poshyvanyk, and M. Grechanik, "Mining performance regression inducing code changes in evolving software," in *Mining Software Repositories (MSR), 2016 IEEE/ACM 13th Working Conference on*. IEEE, 2016, pp. 25–36.

[64] J. Li, A. Sun, and Z. Xing, "Learning to answer programming questions with software documentation through social context embedding," *Information Sciences*, vol. 448, pp. 36–52, 2018.

[65] M. Choetkiertikul, H. K. Dam, T. Tran, T. Pham, and A. Ghose, "Predicting components for issue reports using deep learning with information retrieval," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ACM, 2018, pp. 244–245.

[66] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing." in *AAAI*, vol. 2, no. 3, 2016, p. 4.

[67] C. Chen, T. Su, G. Meng, Z. Xing, and Y. Liu, "From ui design image to gui skeleton: a neural machine translator to bootstrap mobile gui implementation," in *Proceedings of the 40th International Conference on Software Engineering*. ACM, 2018, pp. 665–676.

[68] K. Moran, B. Li, C. Bernal-Cárdenas, D. Jelf, and D. Poshyvanyk, "Automated reporting of gui design violations for mobile apps," *arXiv preprint arXiv:1802.04732*, 2018.

[69] K. Moran, C. Watson, J. Hoskins, G. Purnell, and D. Poshyvanyk, "Detecting and summarizing gui changes in evolving mobile apps," *arXiv preprint arXiv:1807.09440*, 2018.