DeepDebugger: An Interactive Time-Travelling Debugging Approach for Deep Classifiers

Xianglin Yang*
xianglin@u.nus.edu
Shanghai Jiao Tong University
Shanghai, China

Yun Lin†
lin_yun@sjtu.edu.cn
Shanghai Jiao Tong University
Shanghai, China

Yifan Zhang‡
zyifan828@gmail.com
Shanghai Jiao Tong University
Shanghai, China

Linpeng Huang
huang-lp@cs.sjtu.edu.cn
Shanghai Jiao Tong University
Shanghai, China

Jin Song Dong
dcsdjs@nus.edu.sg
National University of Singapore
Singapore, Singapore

Hong Mei
meih@pku.edu.cn
Shanghai Jiao Tong University
Shanghai, China

ABSTRACT

A deep classifier is usually trained to (i) learn the numeric *representation vector* of samples and (ii) classify sample representations with learned *classification boundaries*. Time-travelling visualization, as an explainable AI technique, is designed to transform the model training dynamics into an animation of canvas with colorful dots and territories. Despite that the training dynamics of the high-level concepts such as sample representations and classification boundaries are now observable, the model developers can still be overwhelmed by tens of thousands of moving dots across hundreds of training epochs (i.e., frames in the animation), which makes them miss important training events.

In this work, we make the first attempt to develop the model time-travelling visualizers to the model time-travelling debuggers, for its practical use in model debugging tasks. Specifically, given an animation of model training dynamics of sample representation and classification landscape, we propose DeepDebugger solution to recommend the samples of user interest in a human-in-the-loop manner. On one hand, DeepDebugger monitors the training dynamics of samples and recommends suspicious samples based on their abnormality. On the other hand, our recommendation is interactive and fault-resilient for the model developers to explore the training process. By learning users' feedback, DeepDebugger refines its recommendation to fit their intention. Our extensive experiments on applying DeepDebugger on the known time-travelling visualizers show that DeepDebugger can (1) detect the majority of the abnormal movement of the training samples on canvas; (2) significantly boost the recommendation performance of samples of interest (5-10X more accurate than the baselines) with the runtime overhead of 0.015s per feedback; (3) be resilient under the 3%, 5%, 10% mistaken user feedback. Our user study of the tool shows that the interactive recommendation of DeepDebugger can help the participants accomplish the debugging tasks by saving 18.1% completion time and boosting the performance by 20.3%.

CCS CONCEPTS

• **Software and its engineering → Software notations and tools**; **Software notations and tools**.

KEYWORDS

debugging, visualization, deep classifier, user study

ACM Reference Format:
Xianglin Yang, Yun Lin, Yifan Zhang, Linpeng Huang, Jin Song Dong, and Hong Mei. 2023. DeepDebugger: An Interactive Time-Travelling Debugging Approach for Deep Classifiers. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3611643.3616252

1 INTRODUCTION

Deep Learning based classifiers (or, deep classifiers) are learnable functions mapping a sample to a predefined class, which have been widely used in software systems in areas such as computer vision [15, 31, 34, 39], finance [33, 44, 52], education [6, 24], and transportation [36, 43, 47]. As showed in Figure 1, typical deep classifiers consist of two components:

- **Representation Learning:** A sample (e.g., an image, a sentence, a voice clip) is transformed into a representation vector in a high-dimensional space. The transformation can be implemented as convolutional layers [25], LSTM layers [16], or transformers [41].
- **Representation Fitting:** The learned representation is further fed into layers for fitting its label as prediction, which is typically implemented as fully connected layers.

A well-functioning deep classifier can learn the *classification landscape* consisting of (1) representations of the samples distributed in the space and (2) classification boundaries to distinguish them well. Mathematically, the sample representations are in the form of high-dimensional vectors, and the classification boundaries are in the form of composite formulas on the high-dimensional vectors.

*Also with National University of Singapore.
†Corresponding author
‡Also with National University of Singapore.

ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA

ACM ISBN 979-8-4007-0327-0/23/12…$15.00
https://doi.org/10.1145/3611643.3616252

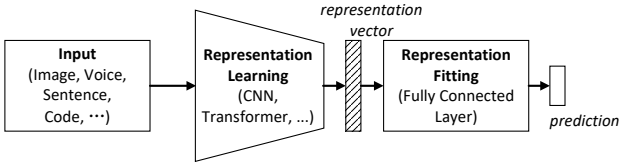


Figure 1: General architecture of deep classifiers, consisting of representation learning and representation fitting

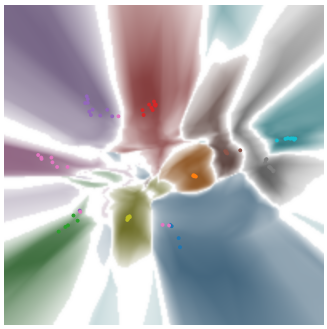


Figure 2: A visualized classification landscape of one training epoch by a time-travelling visualizer TimeVis [48]. A pink dot located in the red region indicates a correct prediction. A pink dot located in the blue region indicates a mis-prediction. The white region indicates classification boundaries.

Recently, time-travelling visualization techniques [48, 49] are emerging to help observe the training dynamics of the high-dimensional representations and classification boundaries in a low-dimensional canvas. Figure 2 shows one example of such visualization. The canvas stands for the universal high-dimensional space, where each dot stands for a training/testing sample, and each colored region stands for a classification region, and the white boundaries stand for the classification boundaries. By this means, the process of learning a deep classifier is transformed into a visualized animation of an evolving canvas. The visualization is useful for the model developers to generally observe how the model predictions are formed. However, the techniques still suffer from the following challenges to serve as a practical model debugging solution:

- **Overwhelming Training Samples and Events:** When visualizing the dynamics of tens of thousands of training samples as dots on the canvas, it is challenging for the model developers to locate which samples and what patterns of sample movements need more attention. Even worse, some movements are hard to be explicitly expressed, as shown in Figure 5.
- **Lack of Intention Inference:** In the model debugging tasks, the developers can raise different hypotheses (e.g., "*whether the unexpectedly low accuracy is due to noisy training samples?*" or "*whether the model is trained with sufficient training samples?*") as they have different intentions for different tasks. Unfortunately, the presentation of animation still lacks the support to detect different task-relevant intentions and validate the aforementioned hypotheses in a debugging task.

In this work, we propose DeepDebugger to further develop the model time-travelling visualizers to the model time-travelling debuggers, with the support of validating diverse hypotheses from the model developers. To this end, we design DeepDebugger as an interactive solution for developers to explore samples of interest in the training animation. Specifically, we design DeepDebugger to (1) recommend abnormal and suspicious samples (i.e., samples with potential issues such as data corruption, and mislabeling), (2) detect task-relevant samples by taking the developers' feedback on the recommendation, and (3) tolerate mistaken feedback for resilient recommendation results. Technically, we design an anomaly detection algorithm based on the location, velocity, and acceleration of the training samples on the canvas. After the developers give feedback (i.e., accept or reject) on the recommendation, we formulate the feedback adaption problem as a 0-1 regression model learning problem. By this means, our data-driven feedback learning model can (1) perform well even with a small set of feedback and (2) resilient to mistaken feedback (by ignoring inconsistent samples). The tool-supported debugging process is iterative and interactive. The recommendation keeps being refined until the developers have validated their hypotheses or located the root cause.

We implement DeepDebugger as a TensorBoard [3] plugin. We extensively evaluate our approach by integrating DeepDebugger with two known time-travelling visualizers. Our experiment on anomaly detection shows that DeepDebugger can detect the samples with abnormal movement with the precision of 82.4% and the recall of 71.9%. Our feedback simulation experiment shows that our recommendation can identify 5-10X more samples of interest than the random baseline, with strong resilience to mistaken feedback. Our user study consists of 16 participants in two model debugging tasks, showing that our feedback-based design in DeepDebugger can help the participants to be more efficient to accomplish the debugging tasks by either saving 18.1% accomplishment time or boosting 20.3% performance, compared to the DeepDebugger version without feedback-based recommendation.

In summary, we make the following contributions:

- **Interaction over Observation:** We enhance the time-travelling visualizer into time-travelling debugger, by introducing the capabilities of detecting abnormal movement, user interaction, and learning feedback. The human-in-the-loop recommendation can be task-adaptive and resilient to mistaken feedback. Through interactive exploration, the users can have a more intuitive understanding of the model behaviors.
- **Tool Support:** We build a TensorBoard plugin, DeepDebugger, based on our approach, supporting various functionalities such as visualization, recommendation, sample query, and interaction. Screenshots, demos, and source code are available at [2].
- **Extensive Experiments:** We design systematic experiments and user studies to evaluate (1) the effectiveness of the designed recommendation and interaction, and (2) the practicability and usability of our tool. The results show that DeepDebugger can significantly improve the model debugging efficiencies in tasks such as identifying suspicious samples and selecting informative unlabelled samples to retrain the model.

2 BACKGROUND

2.1 Deep Classifier

Consider a classification problem with input samples $S = \{s_1, s_2, ..., s_n\}$ and a predefined set of classes $C = \{c_1, c_2, \ldots, c_p\}$, a learned

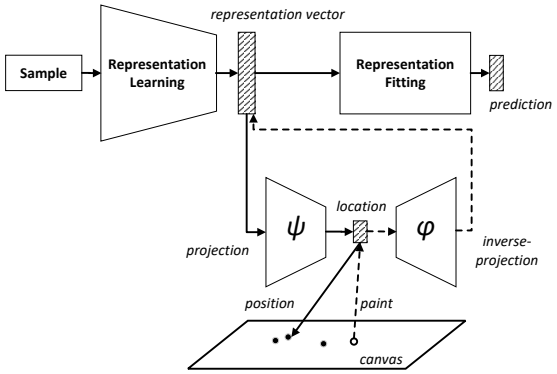


Figure 3: An illustration of how a time-travelling visualizer visualizes an individual snapshot of a classifier, i.e., the classifier trained at the i-th epoch.

classifier $m : S \rightarrow C$ is a function which maps s to one of the predefined classes $c_i \in C$, denoted as $m(s) = c_i \in C$. Typically, a deep learning based classifier m can be decomposed into a representation learning function $f(\cdot)$ and a representation fitting function $g(\cdot)$, i.e., $m = f \circ g$, as showed in Figure 1. Therefore, given a dataset $\mathcal{S} = \{s_1, s_2, ..., s_n\}$, we can have the set of representation vectors $\mathcal{R} = \{r_1, r_2, ..., r_n\}$ where each $r_i = f(s_i)$ is a high-dimensional vector. Moreover, we also have $m(s_i) = g(r_i) = g(f(s_i))$.

2.2 Time-Travelling Visualization

2.2.1 Definition of Time-Travelling Visualization. A time-travelling visualizer takes as input a dataset $\mathcal{S} = \{s_1, s_2, ..., s_n\}$ and a sequence of partially trained classifiers $\mathcal{M} = \langle m_1, m_2, ..., m_k \rangle$ where m_i is the classifier learned in i-th epoch in the training process. Then, it generates as output a sequence of frames $\mathcal{F} = \langle frm_1, frm_2, ..., frm_k \rangle$ where frm_i is a two-dimensional canvas, for reflecting the classification landscape of i-th representation space $\mathcal{R}$. Intuitively, the classification landscape describes (1) which region in $\mathcal{R}$ belongs to what class and (2) how the prediction confidence distributes over each region. Thus, for each pixel in a frame frm_i, the visualizer can describe its predicted class and confidence by the corresponding color and its depth. Specifically, the visualizer has a coloring function $\mathbb{R}^2 \rightarrow \{0, 1, \ldots, 255\}^3$ where the input is a pixel and the output is a depth-aware color in the form of the RGB value. The colors represent classes, and the color depth represents the confidence. The smaller the confidence, the lighter the color. As a result, each frame is visualized as the canvas showed in Figure 2.

2.2.2 Visualizing One Frame. A time-travelling visualizer is implemented as a visualization model $v = \{\phi, \psi\}$ with auto-encoder architecture [30] where ϕ is the encoder and ψ is the decoder. Figure 3 shows how a general time-travelling visualizer works, which consists of two branches, i.e., the encoder branch (the solid lines in Figure 3) and the decoder branch (the dashed lines in Figure 3).

The encoder $\phi : \mathcal{R} \rightarrow \mathbb{R}^2$ is for *positioning* the sample representation. A sample representation $r \in \mathcal{R}$ will be projected to low-dimensional space at $loc = \phi(r) \in \mathbb{R}^2$ where loc is a two-dimensional location on the frame. By this means, any samples, training or testing, can find its position on the canvas.

The decoder $\psi : \mathbb{R}^2 \rightarrow \mathcal{R}$ is for *painting* classification landscapes. Given an arbitrary location loc on the canvas, the decoder inverse-projects it to the representation space to have $r' = \psi(loc)$ so that we can paint loc with the color of $c = g(r')$. Note that, many representation fitting layers in deep learning model provides a measurement of confidence, e.g., by sigmoid or softmax function. Hence, the color of depth can be calculated with confidence.

Generally, different time-travelling visualizers learn the encoder ϕ and decoder ψ for different visualization properties [48, 49].

2.2.3 Visualizing Training Process. Given a training process as a sequence of classifiers $\mathcal{M} = \langle m_1, m_2, ..., m_k \rangle$, we can generate one frame frm_i from each $m_i \in \mathcal{M}$. Generally, different time-travelling visualizers have different strategies to make sure that two adjacent frames frm_i and frm_{i+1} are continuous [48, 49].

2.2.4 State-of-the-art Visualizers. The time-traveling visualizers, to the best of our knowledge, are DVI [49] and TimeVis [48]. Both can transform the model training progress (i.e., a sequence of recorded models) into an animation in the two-dimensional canvas. DVI has its advantage over TimeVis in preserving the spatial topology between the low and the high dimensional space. On the other hand, TimeVis prioritizes efficient learning of the visualization model, albeit at the cost of some visualization effectiveness. Given the space limit, the interested audience can refer to [49] and [48] for more details. In the following, we call the model being visualized as *subject model* and the autoencoder model implementing the visualization as *visualizer*.

3 PROBLEM DEFINITION

Despite time-travelling visualizers providing a useful abstraction for intuitive inspection, there are enormous training events such as representation movement and boundary evolution in the training dynamics. In this work, we aim to address the following research problems to equip the visualization with debugging functionalities:

- **Problem 1 (Training Anomaly Detection):** Given a large number of training events such as representation movement, how could we recommend the events (significantly) different from the majority to be inspected first?
- **Problem 2 (Intention Detection):** Assuming a model developer has an implicit intention for certain tasks, how could the time-travelling debugger capture its intention and recommend samples of interest accordingly?

It is important to note that Problem 2 does not define a specific task, such as detecting noisy training data or addressing vanishing gradients. Instead, we require a solution that can infer the user's intention in an open-ended setting. Analogous to a traditional debugger for software developers, a model debugger should facilitate open-ended exploration of changes in classification boundaries and representation embeddings during model training, adapting to various scenarios and intentions.

4 APPROACH OVERVIEW

Figure 4 shows an overall design of the DeepDebugger framework, as a time-travelling debugger for model behavior investigation. DeepDebugger is built upon time-travelling visualizers which transform the training process into an animation. DeepDebugger takes

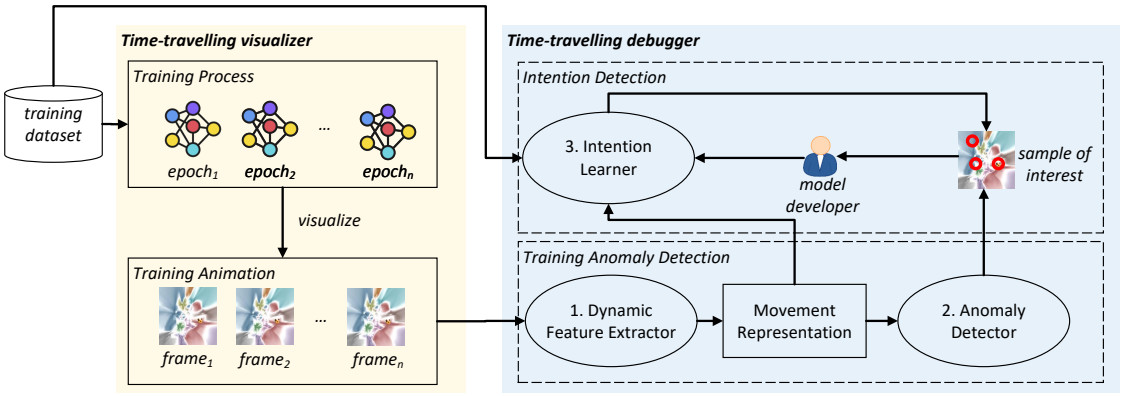


Figure 4: Overview of the designed DeepDebugger framework

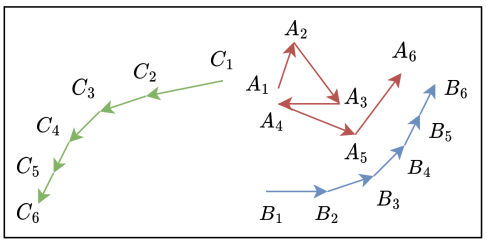


Figure 5: Three trajectories of three learned representation vectors. A and B are similar in terms of the locations; A and B can be distinguishable in terms of velocity; and B and C are similar in terms of acceleration.

the training animation as input and (1) reports the samples with abnormal training samples and (2) interactively learns and refines the samples of user interest from the feedback of the model developers.

To this end, we will first extract the features of each sample's training dynamics (Dynamic Feature Extractor in Figure 4). Then, we design an anomaly detector to report the abnormal movement, as the initial recommendation of samples of interest (Anomaly Detector in Figure 4). The model developers can investigate recommended samples and provide feedback on their relevance, with options to accept, reject, or ignore based on their current tasks. We implement an intention-learning solution to (1) efficiently learn the feedback on-the-fly and (2) generalize the small-scale feedback to more samples without feedback. This interactive, iterative process continues until users can identify root causes (e.g., mislabeled training samples) or validate their debugging hypotheses. validate their debugging hypothesis (Intention Learner in Figure 4).

4.1 Dynamic Feature Extraction

Given a sample, its dynamic features consist of both its trajectory on the canvas and the change of its prediction during the training process, namely trajectory features and prediction features. They track the training dynamics of the representation learning layer and representation fitting layer respectively, offering valuable insights into the underlying patterns of movement (see Figure 1).

4.1.1 Trajectory Features. For the visual trajectory features, we capture the movement by location, velocity, and acceleration.

Location. The location of a sample represents regional information on the classification landscape, which can capture regional patterns such as samples sharing similar predictions. Specifically, given a sample s, we can have all its historical locations on the canvas, $L = \langle l_1, l_2, ..., l_n \rangle$, where l_i indicates its location on i-th frame (or epoch). Each location l is a two-dimensional vector (x, y) representing a coordinate on the canvas. Thus, we encode the location feature $r_{loc} = (x_1, y_1, x_2, y_2, ..., x_n, y_n)$. Note that the location features allow us to detect anomalies based on the embedding of the inputs. Figure 5 shows an example with three trajectories A, B, and C. The location features make A and B share more similarity than that of A and C and that of B and C.

Velocity. The velocity is the first-order derivative of the location feature, which quantifies the rate of change in position from its source to its destination. Specifically, given a location sequence L, we can have its velocity sequence $V = \langle v_1, v_2, ..., v_{n-1} \rangle$ where $v_i = l_{i+1} - l_i = (x'_i, y'_i)$. Thus, we encode the velocity feature $r_v = (x'_1, y'_1, x'_2, y'_2, ..., x'_{n-1}, y'_{n-1})$. As in Figure 5, the velocity features enable differentiation between trajectories A and B, as A exhibits a more complex, zigzag movement pattern, indicative of greater struggle compared to the relatively smoother path of trajectory B.

Acceleration. The acceleration is the first-order derivative of the velocity, and the second-order derivative of the location, which quantifies the rate of change in velocity as a sample representation transitions from its origin to its destination. Specifically, given the velocity sequence V of a sample, we have its acceleration sequence $A = \langle ac_1, ac_2, ..., ac_{n-1} \rangle$ where $ac_i = v_{i+1} - v_i = (x''_i, y''_i)$. Thus, we encode the acceleration feature $r_{ac} = (x''_1, y''_1, x''_2, y''_2, ..., x''_{n-2}, y''_{n-2})$. As in Figure 5, the acceleration feature identifies similarities between the trajectories B and C. Both trajectories exhibit smooth progression towards their respective destinations.

We concatenate three dynamic features as $r_t = (r_{loc}|r_v|r_{ac})$.

4.1.2 Prediction Feature. In this work, we assume that the target classifier can provide us with a continuous value to indicate its confidence. For example, the softmax function [8] predicts a probability p for each class, the higher the p, the more confident the prediction is. Similarly, the sigmoid function [13] predicts a value p between 0 and 1 for binary classification, the closer p is to 0 or 1, the more confident the prediction. Therefore, given a sample s, its confidence $conf$ is the score of the classifier on s at an epoch e:

$$conf(s)_e = \begin{cases} p & \text{softmax function} \\ 2 \times |p - 0.5| & \text{sigmoid function} \end{cases} \quad (1)$$

Thus, we have prediction feature $r_{conf} = \langle conf_1, conf_2, ..., conf_n \rangle$. Finally, the overall concatenated dynamic feature is $r_{mv} = (r_t | r_{conf})$.

4.2 Anomaly Detector

In this work, we detect the anomalies by learning the concept of *normality* and the *abnormality* from the training dataset. We denote a training sample for learning the (ab)normality as a "seen" sample and generalize it to "unseen" samples. To this end, we adopt a clustering algorithm for anomaly detection. We assess sample similarity using the cosine similarity of the dynamic features extracted in Section 4.1. Subsequently, we assign an anomaly score to each sample based on its cluster size. Ultimately, we identify and report samples belonging to smaller clusters as anomalies.

4.2.1 Learning Normality and Abnormality. Let any dynamic feature (e.g., location, velocity) be $R = \{r_1, r_2, ..., r_n\}$, we use Birch hierarchical clustering algorithm [53] to split R into k clusters. We choose the Birch [54] clustering algorithm as its leading performance in time and space complexity among other alternatives such as K-means [32]. Then, for any sample in cluster cls, we quantify its movement abnormality as $1 - \frac{|cls|}{n}$. The smaller the size of cluster cls, the more abnormal the trajectory in the cluster. For example, assume that we have 3 clusters cls_1, cls_2, cls_3 where $|cls_1| = 45,000$, $|cls_2| = 4,500$, and $|cls_3| = 500$. Therefore, the abnormality of cls_1 is $1 - \frac{45000}{50000} = 0.1$. In contrast, that of cls_3 is $1 - \frac{500}{50000} = 0.99$, which is much higher than that of cls_1. We let the samples within the same cluster share the same abnormality score.

4.2.2 Generalizing Normality and Abnormality. As for an unseen sample s_{unk}, we can have its dynamic feature r_{unk} as introduced in Section 4.1. First, we denote the centroid of a cluster cls as

$$center(cls) = \frac{1}{|cls|} \times \sum_{i=1}^{|cls|} r_i \quad (2)$$

Also, we denote the radius of a cluster C as

$$radius(cls) = \arg\max_{r_i \in cls} dist(center(cls), r_i) \quad (3)$$

Then we determine if an unseen sample s_{unk} belongs to a cluster cls_i by comparing the Euclidean distance between r_{unk} and the cluster center $center(cls_i)$ to the cluster's radius $radius(cls_i)$. If the distance is smaller than the radius, we consider the sample to be part of the cluster. The sample s_{unk} is considered normal if it falls into a normal cluster, and abnormal if it falls into an abnormal cluster. If r_{unk} does not belong to any of the existing clusters, we create a new cluster cls_{k+1}. In this case, the sample's abnormality score is close to 1, indicating a high likelihood of it being an anomaly.

We consider samples with scores exceeding a user-defined threshold th_{ab} as anomalies. For instance, we report all samples in cls_3 (the last cluster) and r_{unk}, which belong to the new cluster cls_{k+1}, as anomalies in both cases respectively.

Ultimately, for a given sample s, if any of its features is identified as an anomaly, we report the sample itself as an anomaly.

4.3 Intention Detector

We design the intention detection scheme with the following consideration. (i) It should perform well on limited and imbalanced feedback without overfitting, as human effort is valuable and often scarce. (ii) it must be efficient enough for ensuring timely responses to user input. (iii) it should be robust against noisy feedback, considering that humans can make mistakes.

To this end, we design the recommendation system as follows. Initially, we recommend the most abnormal training samples (see Section 4.2). Then we take as input the feedback from the users in the form of accepting or rejecting these samples and recommend relevant samples as output. Specifically, we regard each feedback as a label for a sample and use an efficient, robust intention learning algorithm to fit and generalize the feedback based on abnormal scores extracted in Section 4.2. This algorithm assigns an interest potential score to other samples without feedback. We recommend samples with high scores and iteratively refine our recommendations based on user feedback.

4.3.1 Interest Potential Estimation. Given a sample s, we estimate its interest potential by an interest estimation function $IE(a_1, a_2, ..., a_n)$ with a_i being its attributes. We utilize the scalar anomaly score to represent each dynamic feature, i.e., a_i being the anomaly score for i-th dynamic feature. This enables faster processing and reduces noise of raw dynamic features, alleviating concerns (i) and (ii). Then, we regard each user feedback for a sample as a label $l \in \{0, 1\}$, with acceptance being 1 and rejection being 0. Further, we can transform the interest-estimation problem as a 0-1 regression [42] problem on the attributes $a_1, a_2, ..., a_n$. Generally, we can use different kernel functions to fit $IE(\cdot)$. In this work, we opt for linear ridge regression due to its learning efficiency and robustness, which effectively address concerns (i), (ii), and (iii). The linear ridge regression predictor can be expressed as:

$$IE(s) = IE(a_1, a_2, ..., a_n) = \sum_{i=1}^{n} c_i \cdot a_i + c_0 \quad (4)$$

In Equation 4, the coefficients (e.g., c_i) are learnable variables regarding users' implicit intention.

4.3.2 The Feedback Framework. We first initialize c_i by $\sum_{i=1}^{n} c_i = 1$ and $c_1 = c_2 = ... = c_n$, reflecting an equal importance score for all dynamic features at the start. Each time users provide feedback by accepting or rejecting a sample, we label it as interested (positive sample) or uninterested (negative sample). Then, we can refine our interest estimation function with the new training data point, continuously improving our recommendation with the evolving user feedback. Practitioners can enhance our work by incorporating additional attributes or employing a different kernel for special needs and use cases. For the example in Figure 5, if the model developer is looking for samples with similar predictions, his or her feedback is expected to lead to a high similarity between A and B. In contrast, if the model developer is to check the learning smoothness of each sample, the feedback can lead to a high similarity between B and C.

For each intention, users can start one *session* of interactions for locating the relevant samples of interest. If users change their intentions, they can start a new session. Technically, all the learned

coefficients of the regression model are reset and new feedback and interactions can derive a new model capturing their new intention.

5 GUI OF DEEPDEBUGGER

We implement DeepDebugger as a TensorBoard [3] plugin as Figure 6. More screenshots and videos are available at [2]. DeepDebugger is designed with four features:

A: Time-travelling Visualization *(TT Vis)*. The model developers can investigate the evolving visualized classification landscape frame by frame (i.e., epoch by epoch) on the timeline at the bottom. They can either play the animation or switch between different epochs by clicking the index on the timeline.

B: Canvas Investigation *(Canvas Inv)*. At each frame, they can observe the training and testing accuracy. The corresponding canvas of each frame can be zoomed in and zoomed out. The users can query the samples and highlight them on the canvas, observe the sample (e.g., image), and select a subset of samples by dragging and dropping a bounding box on the canvas.

C: Sample of Interest Recommendation *(SoI Rec)*. DeepDebugger can recommend samples of interest based on the technique described in Section 4.2 and Section 4.3. The recommended samples will be listed in section D1 of Figure 6. Hovering on any sample, a detailed description of the sample such as its appearance, prediction, and label will be shown.

D: Interactive Feedback *(Interaction)*. Once the users accept or reject individual recommendations (in section D2 of Figure 6), the runtime interest detector in DeepDebugger will be triggered to provide a new recommendation. The process is interactive, explorative, and iterative, for the model developers to have a more informed investigation to understand the model training behaviors.

6 EVALUATION

We evaluate DeepDebugger with the following research questions (RQ). Readers can check [2] for the replication details.

- **RQ1 (Abnormality Detection)**: Can DeepDebugger effectively detect abnormal training movement on the canvas?
- **RQ2 (Intention Detection)**: Can DeepDebugger recommend relevant samples based on user feedback with acceptable feedback efforts and runtime overhead? And what is the importance scores of different features in different scenarios?
- **RQ3 (Error Resistance)**: What is the performance of DeepDebugger if the user provides incorrect feedback?
- **RQ4 (User Study)**: Whether the interaction design in DeepDebugger is useful to address the model debugging tasks in practice? How do they use the tool?

To answer RQ1, we generate ground-truth normal and abnormal trajectories to evaluate their precision and recall to detect the abnormal ones. To answer RQ2, we design two types of intentions and simulate user feedback in relation to those intentions. To answer RQ3, we inject noisy feedback on the tasks designed in RQ2, evaluating how the performance of DeepDebugger is impacted. To answer RQ4, we conduct a user study involving two model debugging tasks, assessing the effectiveness of DeepDebugger when used by human participants.

Table 1: Performance of detecting out-of-distribution Abnormal Movements on 6 visualized training processes.

Dataset	MNIST		FMNIST		CIFAR-10	
Visualizer	DVI	TimeVis	DVI	TimeVis	DVI	TimeVis
Precision	92.6%	94.7%	80.2%	85.5%	80.1%	72.4%
Recall	67.4%	78.4%	89.0%	83.8%	56.4%	56.6%
F1 score	78.0%	85.8%	84.4%	84.6%	66.2%	63.5%

Table 2: Selection of out-of-distribution datasets

Dataset	CIFAR-10	MNIST	FMNIST
OOD dataset	FMNIST	CIFAR-10	MNIST

Experiment Settings. In the evaluation, we empirically chose $k = 30$ with the following consideration: (1) to ensure all the clusters are not excessively small, we opted for a small value of k, and (2) to prevent a few anomalies from being merged into a single large cluster, we selected a relatively larger value of k. Additionally, we set $th_{ab} = 0.98$. Such settings are used throughout all the experiments.

6.1 Anomaly Detection (RQ1)

6.1.1 Datasets, Subject models, and Time-travelling visualizers. We first design some training processes to be visualized. In this experiment, we use MNIST [10], FMNIST [45], and CIFAR-10 [23] as the datasets. We train the model architecture of ResNet18 as the subject model on each dataset until the training accuracy converges. The subject model takes 15 epochs to converge on MNIST, 50 epochs to converge on FMNIST, and 200 epochs to converge on CIFAR-10. We use all known time-travelling visualizers to the best of our knowledge, i.e., DVI [49] and TimeVis [48], to visualize the training processes of three datasets. Thus, we have 2 (visualizers) × 3 (processes) = 6 visualized processes. For convenience, we use $P(dataset, visualizer)$ to denote the visualized process derived by the classifier trained by $dataset$ with $visualizer$. For example, $P(cifar10,dvi)$ indicates the visualized training process for the model trained by CIFAR10 with DVI.

6.1.2 Setup. We define ground-truth of unseen normal and abnormal sample movements as follows.

- **Ground-truth Normal Movement**: We take the trajectories of testing samples as the ground-truth normal movements considering they are not used by the visualizers and share the same distribution with training data.
- **Synthesized Abnormal Movement**: To generate abnormal movements, we take into account both trajectory features and prediction features. For trajectory features, we create a random trajectory step-by-step that has the same shape as normal samples on the canvas but with random accelerations. For prediction features, we randomly sample a confidence sequence with each of its elements ranging from 0 to 1.
- **Natural Abnormal Movement**: For a given training process, we use the samples in another dataset as the out-of-distribution data and record their movements as abnormal movements. We assume that out-of-distribution samples behave differently from the training data. Specifically, we choose the out-of-distribution datasets for our datasets as Table 2.

For each visualization $P(dataset, visualizer)$, DeepDebugger identifies m abnormal movements. Suppose c out of these m movements

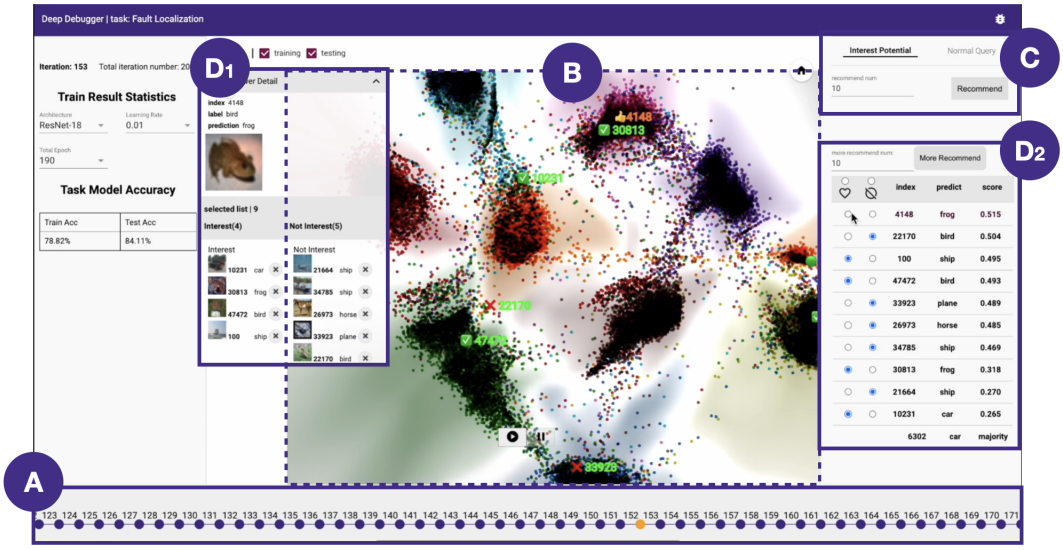


Figure 6: Screenshot of DeepDebugger tool. The interface includes four sections: A as Time-Travelling Visualization *(TT Vis)*, B as Canvas Investigation *(Canvas Inv)*, C as Sample of Interest Recommendation *(SoI Rec)*, and D as Interactive Feedback *(Interaction)*.

are correct. we calculate precision as $\frac{c}{m}$ and recall as $\frac{c}{n}$. We assessed our anomaly detector using three distinct sets: 500 testing samples, 500 synthesized samples, and 500 out-of-distribution data points.

6.1.3 Results. The recall of synthesized abnormal movement is 100% showing DeepDebugger's strong ability in detecting synthesized anomalies. Table 1 shows the good performance of DeepDebugger on the precision and the recall of the natural abnormal movements. Overall, the average precision is 82.4% and the average recall is 71.9%. Moreover, the performance over different time-travelling visualizers are comparable.

Further, we qualitatively analyze the false positive and false negative as follows. The "`testing`" sample in Figure 7 is an example of a false positive in the MNIST dataset, caused by the distribution shift of the testing samples. Compared to the majority of normal pictures of digital "9" (as shown in "`normal`" 9), the sample renders a difference in the inclining angle, which might incur unexpected anomaly. In contrast, the "`OOD`" sample in Figure 7 is an example of false negative for the FMNIST dataset. In this case, the movement of the "`OOD`" sample is normal despite it being regarded as an out-of-distribution. This observation aligns with the reported findings in explaining out-of-distribution data [5] that out-of-distribution could behave in an in-distribution manner.

6.2 Intention Detection (RQ2)

In this experiment, we consider two types of samples of interest. **Noise Samples Detection.** The first type is noisy samples (i.e., mislabeled samples). Model developers can assess if poor training results are due to noisy training samples by examining the data. **Mispredicted Samples Detection.** The second type is mis-predicted samples in the wild (i.e., mis-predicted samples that are neither part of the training nor testing datasets). Given the high cost of labeling samples, some developers might intend to have a partially trained

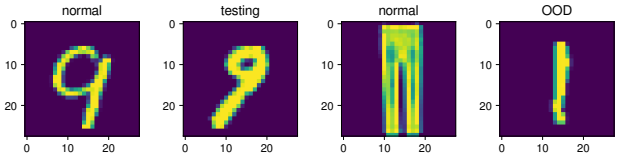


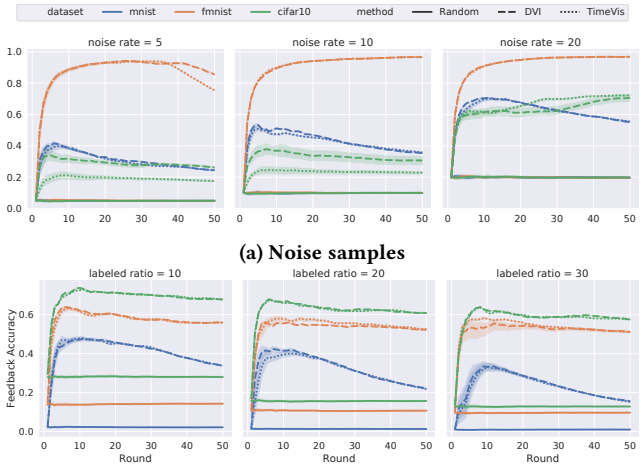
Figure 7: Examples of true negative, false negative, and false positive. The first and third figures are two true negatives from MNIST and FMNIST respectively. The second figure is a testing sample from MNIST which is reported as false positive. The fourth figure is an OOD sample for FMNIST, reported as a false negative.

model and then look for unlabelled samples with more potential to improve the model. Thus, they would be interested in finding unlabelled samples on which the preliminary model mis-predict.

6.2.1 Setup. We use the configurations and hyperparameter settings (i.e., datasets, model architecture, and the use of visualizers) as described in Section 6.1.1, except that the training dataset is changed for the designated intention:

- **Noisy Samples**: Before training, we randomly change 5%, 10%, and 20% labels to other classes in the training dataset.
- **Mis-predicted Samples in the wild**: We only train the model using 10%, 20%, and 30% of samples in the training datasets. The remaining samples are viewed as unlabeled data from the wild. Note that, those samples are not used for training and we don't have their label information.

Following the notion defined in Section 6.1, we define $P(dataset, visualizer, type, ratio)$ as the visualization process for the model trained on $dataset$, with an intention type of $type$, an intention ratio of $ratio$, and the $visualizer$ element. We then perform a feedback-simulation experiment for each configuration (e.g., $P(cifar10, dvi, n,$



(a) Noise samples

(b) Misprediction samples in the wild

Figure 8: Results on the accuracy of recommendation. (a): The accuracy of recommending noise samples on MNIST, FMNIST, and CIFAR-10 in 50 rounds of feedback when noise rate is 5% (left), 10% (mid), and 20% (right). (b): The accuracy of recommending mispredicted samples in the wild on MNIST, FMNIST, and CIFAR-10 in 50 rounds of feedback when training data is 10% (left), 20% (mid), and 30% (right).

5%)) 50 times, documenting the time taken in each feedback round under two scenarios. In our analysis, we benchmark DeepDebugger against a random selection baseline. In addition, the feedback is automatically generated based on the target intention, providing an acceptance (i.e., 1) if the sample is of interest and a rejection (i.e., 0) otherwise. This feedback emulates user input within the context of the tested intention.

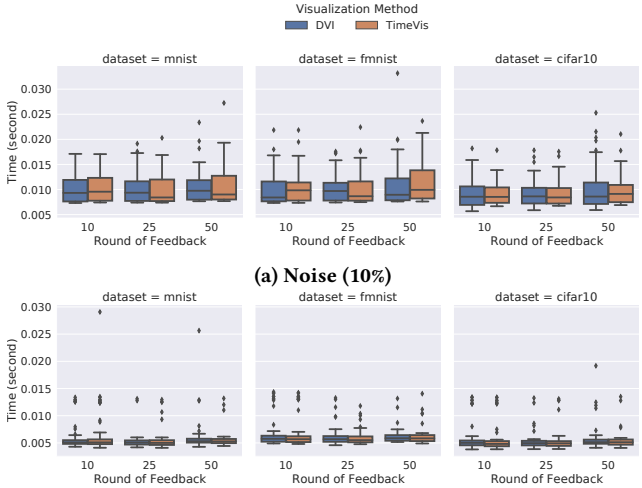
Finally, we evaluate DeepDebugger's feedback performance by the number of accurate recommendations within k ($k = 50$) feedback iterations. 50 samples are recommended in each iteration.

6.2.2 Results. Figure 8 shows how feedback can help recommend noise samples and mis-predicted samples in the wild. Line colors and styles denote different datasets and visualization methods, while the shaded area indicates the standard deviation. Overall, DeepDebugger is effective when paired with both DVI and TimeVis visualization methods. There is a significant improvement in accuracy in aligning with user intention within just 2 to 3 rounds of feedback, which substantially outperforms the random baseline. Furthermore, the marginal benefits of feedback increase as there are fewer samples of interest available in the pool. In Figure 8a, the gap between DeepDebugger with DVI and DeepDebugger with TimeVis on CIFAR-10 dataset demonstrates that DVI produces higher-quality visualization when the target training process is much longer (i.e., 200 epochs for CIFAR-10, 20 epochs for MNIST), resulting in improved feedback accuracy.

Table 3 shows the importance scores extracted from Equation 4 of various dynamic features in situations with a noise sample rate of 20% and a labeled ratio of 10% for mis-predicted samples. More results in other settings are in [2]. The results highlight the importance of different features in varied scenarios, emphasizing the

Table 3: The important scores of four dynamic features. FT stands for the feature type.

Scenarios	FT	MNIST		FMNIST		CIFAR10	
		DVI	TimeVis	DVI	TimeVis	DVI	TimeVis
Noise samples	conf	0.402	0.467	0.063	0.067	0.051	0.811
	pos	0.203	0.180	0.341	0.185	0.407	0.003
	vel	0.145	0.182	0.121	0.212	0.153	0.078
	acc	0.251	0.171	0.474	0.536	0.389	0.108
mispredicted samples in the wild	conf	0.795	0.830	0.399	0.265	0.344	0.299
	pos	0.107	0.037	0.469	0.247	0.348	0.144
	vel	0.051	0.071	0.051	0.251	0.181	0.204
	acc	0.048	0.063	0.081	0.237	0.127	0.354



(a) Noise (10%)

(b) Misprediction (10%)

Figure 9: Runtime Efficiency. Box plot of runtime efficiency of our feedback algorithm on two visualization methods: *DVI and TimeVis*, and two scenarios tested.

robustness of our method across diverse situations. For CIFAR-10 dataset's noise sample scenario, DeepDebugger's importance scores differ noticeably between DVI and TimeVis. DeepDebugger with DVI favours trajectory features as dominant features, while that with TimeVis depends on the confidence features. This aligns with the discrepancy in feedback accuracy for the CIFAR-10 dataset observed in Figure 8 between DVI and TimeVis, given TimeVis's predominant reliance on the confidence feature.

Figure 9 shows the runtime overhead for generating feedback in the 10-th, 25-th, and 50-th rounds in scenarios where the noise sample ratio is 10% and the labeled sample ratio is 20%. Full results are available in [2]. The time overhead to generate feedback is less than 0.015s, which makes DeepDebugger very practical. In addition, the cost is almost constant regarding different configurations.

6.3 Error Resistance (RQ3)

6.3.1 Setup. We use the same configuration as described in Section 6.2. Nevertheless, we inject 3%, 5%, and 10% errors in the simulated feedback and observe the accuracy of recommending samples of interest to simulate human error.

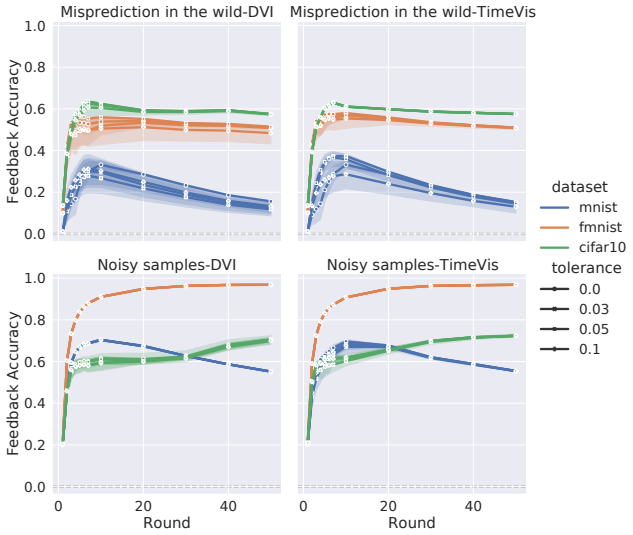


Figure 10: Feedback Tolerance. Line plot of feedback tolerance on two methods: *DVI* and *TimeVis*, and two scenarios tested.

6.3.2 Results. Figure 10 shows the accuracy of recommending samples of interest under the noise feedback of 3%, 5%, and 10%. The colors of the line represent the dataset, and the styles of the line represent the noise rate of feedback. We only show mispredicted samples in the wild with the labeled ratio of 30% and noise samples with a noise rate being 20%. Please refer to [2] for results in other settings. As shown in Figure 10, minimal performance degradation is observed when noise is added to feedback. Therefore, DeepDebugger is robust against noise in diverse settings and the performance is still $5-10X$ better than the random baseline. In detecting mis-predicted samples in the wild scenario, we observe a decreasing performance in the MNIST dataset when the noise rate of feedback increases. This is due to there being fewer and fewer remaining available mis-predicted samples in the pool.

6.4 User Study (RQ4)

To answer RQ4, we design a user study on two practical model debugging tasks to observe the effectiveness of DeepDebugger and user's behavior in practice.

6.4.1 Participants. We recruit 16 participants having computer science backgrounds from XXX in YYY (anonymous university and country). The participants have on average 2.4 years of experience in model training. 12 out of 16 participants have experience using TensorBoard. None of them have used DeepDebugger before. A more detailed demographical analysis can be found in [2]. We provide each participant with a tutorial on DeepDebugger using a toy example a day prior to the study, followed by a quiz consisting of five questions. Participants were then paired and split into two equivalent groups based on their quiz performance: the experimental group (**EG**) which used DeepDebugger with all functionalities, and the control group (**CG**) which used DeepDebugger with feedback-based recommendations replaced by default recommendations (i.e., user feedback was not utilized for refining recommendations). Both groups interacted with the same user interface and were unaware of their group assignment.

6.4.2 Tasks. We design two model debugging tasks:

- **Fault Localization (Task 1):** In the one-shot model training setting, given that the training and testing accuracies do not meet the expectation, how do we infer the root cause?
- **Informative Sample Selection (Task 2):** In the continuous model training setting, given that we have (1) a model m trained based on a limited set of training dataset and (2) a set of unlabelled samples $\mathcal{D}_u$, how can we know what samples selected from $\mathcal{D}_u$ can be more useful to boost the performance of m?

Task 1 Setup. We inject 20% mislabeled samples on CIFAR-10 dataset and train a model with limited training and testing accuracy. We present the participants with the training/testing accuracy and 7 choices of the root causes, then ask them to choose the root cause with justification. The choices include (1) noisy training samples, (2) noisy testing samples, (3) inexpressive model architecture, (4) inappropriate learning rate, (5) insufficient training epochs, (6) gradient disappear, and (7) unbalanced training dataset. Note that, participants selecting the noisy samples option must identify at least 20 such samples. All of the root causes can be pinpointed by our tool via direct canvas examination, feedback recommendations, observing accuracy and dynamics in alternate settings using provided visualization tabs, or checking training metadata. Detailed elaborations of these aspects can be found in [2].

Task 2 Setup. We start by training a model on 4% of the CIFAR-10 dataset. The participants are shown the visualization of the model's landscape and the distribution of training, testing, and unlabelled samples on this landscape Participants are then asked to answer the following question, "*Given this model, if we aim to label an additional 500 unlabelled samples for retraining, would using mispredicted unlabelled samples prove more beneficial for boosting performance than using uncertain samples?*". In this task, the uncertainty-based selection is no less effective than the misprediction-based solution. After selecting and labeling the relevant samples, the participants can use DeepDebugger to retrain the model to validate their hypotheses.

Both tasks require participants to identify relevant samples to affirm their hypothesis, a process akin to programmers debugging a program. Each task has a one-hour time budget, but early submissions are acceptable. Participants are instructed to record their sessions using a video recorder (i.e., Zoom) for post-analysis. Moreover, we have instrumented all of DeepDebugger's functionalities to record user behaviors quantitatively. After the study, participants are invited to provide feedback on the tool through a survey.

Performance Evaluation. We evaluate the performance of Task 1 based on (1) whether the participants make the correct choice, (2) if so, whether they can find enough noisy samples to support their claim, and (3) task completion efficiency. Task 2 performance is assessed based on (1) whether participants reach the correct conclusion and (2) the efficiency of drawing the conclusion.

6.4.3 Results. Table 4 and Table 5 display the performance of the EG and CG in Tasks 1 and 2, based on their decisions, the quantity of correct selected **s**amples **o**f **i**nterest (SoI), and time spent. Both groups made the correct decisions for both tasks, yet the EG selected more correct SoI (17.5 vs. 17.4 in Task 1 and 432.9 vs. 359.9 in Task 2), and took less time (25'8" vs 31'52" in Task 1 and 30'15" vs. 33.53" in Task 2). Table 6 shows a Wilcoxon signed rank test on four variables, i.e., *SoI* and *Time* in both tasks. The p-values of

Table 4: Comparison of groups based on correct sample selection and time spent on Task 1.

EG	Decision	SoI	Time	CG	Decision	SoI	Time
P1	Correct	20	0:13:00	P9	Correct	20	0:17:00
P2	Correct	20	0:16:00	P10	Correct	20	0:19:00
P3	Correct	20	0:36:00	P11	Correct	19	0:40:00
P4	Correct	19	0:19:00	P12	Correct	19	0:40:00
P5	Correct	17	0:20:00	P13	Correct	18	0:21:00
P6	Correct	18	0:32:00	P14	Correct	18	0:47:00
P7	Correct	16	0:28:00	P15	Correct	15	0:30:00
P8	Correct	10	0:37:00	P16	Correct	10	0:41:00
Avg	/	17.5	0:25:08	/		17.4	0:31:53

Table 5: Comparison of groups based on correct sample selection and time spent on Task 2.

EG	Decision	SoI	Time	CG	Decision	SoI	Time
P1	Correct	406	0:27:00	P9	Correct	388	0:34:00
P2	Correct	469	0:40:00	P10	Correct	466	0:39:00
P3	Correct	432	0:23:00	P11	Correct	321	0:39:00
P4	Correct	479	0:35:00	P12	Correct	388	0:26:00
P5	Correct	469	0:37:00	P13	Correct	344	0:46:00
P6	Correct	449	0:36:00	P14	Correct	329	0:25:00
P7	Correct	322	0:15:00	P15	Correct	351	0:36:00
P8	Correct	437	0:29:00	P16	Correct	292	0:26:00
Avg	/	432.9	0:30:15	/		359.9	0:33:53

Table 6: Wilcoxon signed rank test of correct *SoI* and *Time* both in Task 1 and Task 2. The subscript denotes different tasks.

Var	SoI_{T1}		$Time_{T1}$		SoI_{T2}		$Time_{T2}$	
Group	EG	CG	EG	CG	EG	CG	EG	CG
Avg	17.5	17.4	0:25:08	0:31:53	432.9	359.9	0:30:15	0:33:53
p-value	0.95		0.02		0.04		0.55	

$Time_{T1}$ and SoI_{T2} fall below 0.05, suggesting statistically significant differences. Specifically, in Task 1, EG spent less time than CG with statistical differences, while their SoI selection performance was comparable. In Task 2, EG selected more SoI with statistical significance, while time usage was comparable. In summary, these findings substantiate the efficacy of DeepDebugger in improving feedback quality across both debugging tasks.

We gathered data on the frequency of each function's usage (see Section 5) by individual users in Task 1 and Task 2 as Table 7, with the following observation: **How do the participants find the noisy samples as the root cause in Task 1?** Given the seven options provided, participants from both groups interacted with the visual canvas and employed the animation feature to identify the correct one. As in Table 7, the function of *TT Vis* and *Canvas Inv* are used frequently. For example, P3, P7, P9, and P10 root out gradient disappear by observing whether the visualized canvas starts being "frozen" at certain checkpoints. Some participants (e.g., P3 and P13) query the number of samples by labels to investigate training sample imbalance. Considering the dataset contains 20% noisy samples, it's unavoidable for every participant to come across some of these noisy instances. However, when it pertains to locating sufficient noisy samples to validate their hypothesis, two groups exhibit divergent strategies.

Experimental Group. Typically, the EG employs the *SoI Rec* function to locate potential samples of interest. Upon encountering several noisy samples, most participants express their interest by offering positive feedback on these samples. With an average of 5 rounds of interaction, nearly all the participants in the EG gather sufficient samples and observe that these samples are evenly dispersed across the canvas. Consequently, they verify the hypothesis concerning the noisy dataset and proceed to make their decision.

Control Group. Similar to the EG, the CG initially utilizes the *SoI Rec* function. However, participants quickly discern that while the tools are able to recommend a few noisy samples, their feedback does not prompt DeepDebugger to recommend more. The quantity recommended is insufficient to draw conclusive results. After engaging with approximately 11.6 rounds of recommendations on average, many of them decide to take matters into their own hands by exploring the visualized canvas directly. This exploration involves a detailed investigation of various regions using the **S**ample **S**election with bounding box *(SS)* function. Once they discover that certain regions contain a higher proportion of noisy samples, they strategically focus their efforts on these areas to uncover more noisy samples. These behaviors align with Table 7, which indicates that the *Interaction* function was utilized more than twice as frequently by the CG compared to the EG (11.6 vs. 5). Similarly, the *SS* function was employed four times as often by the CG (2 vs. 0.5).

Consequently, both groups exhibit comparable proficiency in identifying the root cause, although the EG achieves this in less time. This indicates the potential time-saving benefits of the tool's feedback mechanisms.

How do the participants find the mis-predicted samples in the wild in Task 2? Compared to Task 1, Task 2 requires participants to collect and label a larger number of samples (i.e., 500). In Task 1, the EG had developed trust in the *Interaction* function, leading them to primarily rely on this function. In contrast, more participants in the CG abandoned the *Interaction* function and adopted the *SS* function with a bounding box to support their exploration-and-exploitation strategy. As evidenced in Table 7, there is a fivefold difference in the use of the *SS* function, and a threefold difference in the use of both *Interaction* and *SoI Rec* functions between the two groups. Although CG's strategy is effective, it lacks the precision offered by the *Interaction* function. As a result, despite spending roughly the same amount of time on identifying samples, the EG excels over the CG in terms of selecting a greater number of correct samples.

Generally, the *SS* function enables participants to devise an "explore-and-exploit" strategy as a mitigative measure once the recommendations lose effectiveness. As per the post-study survey and interviews, the majority of participants from both groups agree that the *Interaction* function is both intuitive and beneficial. Participants from the EG in particular found the *Interaction* function useful in selecting samples of interest and validating their debugging hypotheses.

6.4.4 Threats to Validity. Our study has two main threats. Internally, participants' varied expertise could affect comparison fairness. To mitigate this, we conduct a pre-study survey and a quiz to evaluate their expertise. Externally, we limit ourselves to two model debugging tasks. To mitigate this, we select the most popular datasets and common debugging practices. Nevertheless, more

Table 7: The frequency of each function's usage by individual users in Task 1 and Task 2. SI represents **S**ample **I**nvestigation, SS represents **S**ample **S**election with bounding box, SQ represents **S**ample **Q**uery. Section 5 describes the function of TT Vis/Canvas Inv/SoI Rec/Interaction. For instance, row 1 column 12 indicates that participant P1 used the "Interaction" function 11 times under Task 2.

Group	Participants	Task 1						Task 2					
		TT Vis	Canvas Inv			SoI Rec	Interaction	TT Vis	Canvas Inv			SoI Rec	Interaction
			SI	SS	SQ				SI	SS	SQ		
EG	P1	3	224	0	0	0	0	1	2094	0	0	2	11
	P2	0	673	0	0	4	6	1	4282	2	0	2	13
	P3	43	2054	0	20	4	4	0	2971	0	0	2	30
	P4	2	581	1	0	2	5	0	2732	10	0	0	33
	P5	5	862	1	1	1	11	0	2831	1	0	1	46
	P6	11	1943	1	3	1	7	0	3865	6	7	3	48
	P7	13	408	0	5	4	5	0	497	0	2	2	10
	P8	59	1721	0	25	7	2	0	2750	0	4	1	24
Avg		**17**	**1058.2**	**0.3**	**6.7**	**2.8**	**5**	**0.2**	**2752.7**	**2.3**	**1.6**	**1.6**	**26.8**
Std		**21.8**	**731.6**	**0.5**	**9.9**	**2.2**	**3.2**	**0.4**	**1143.4**	**3.7**	**2.6**	**0.9**	**15.1**
CG	P9	43	1061	1	1	5	12	2	1160	2	1	0	12
	P10	33	1341	1	4	5	12	0	2428	7	4	0	1
	P11	16	929	0	1	1	13	2	2942	5	1	3	17
	P12	2	283	0	2	1	7	0	1409	22	2	0	0
	P13	1	2220	2	10	4	9	0	4084	3	10	0	22
	P14	8	2243	5	0	12	12	1	2820	21	0	0	1
	P15	0	996	0	1	1	14	3	1755	15	1	1	0
	P16	24	1192	7	0	1	14	0	4718	11	0	0	11
Avg		**15.8**	**1283.1**	**2**	**2.3**	**3.7**	**11.6**	**1**	**2664.5**	**10.7**	**2.3**	**0.5**	**8**
Std		**16.1**	**662**	**2.6**	**3.3**	**3.8**	**2.4**	**1.1**	**1256.7**	**7.8**	**3.3**	**1**	**8.6**

configurations such as more architectures are still needed to be investigated in future work.

7 RELATED WORK

Visual Explanation of Deep Learning Models. Many researchers have proposed visualization techniques to understand AI models [9, 11, 37, 38, 40, 48, 49]. GradCAM [38] and its variants [9, 11] are designed to highlight a region of an image to explain the prediction. Seq2seq-Vis is designed to visualize the most influential words in an input sentence to its translation [40]. Our approach focuses on improving the debugging capability of the time-travelling visualization techniques such as DVI [49] and TimeVis [48].

Interactive Debugging and Fault Localization. Comparing to the one-shot fault localization techniques such as spectrum-based fault localization [4, 51] and delta-debugging [7, 17, 50], many debugging approaches are designed in an interactive way, on program execution trace [19–22, 28, 29, 46] and source code [12, 14, 18, 26, 27, 35]. Trace-based interaction can enhance the time-travelling debugging techniques [19–22, 28, 29, 46]. Ko et al. [19–22] propose Whyline, to generate why and why-not questions about the program behaviors to the users. Their follow-up works are further proposed to recommend suspicious steps on the trace to expedite the recommendation and trace exploration, by pattern summarization [29], probability inference [46], and machine learning model [28]. Source code based interaction is also designed to enhance existing approaches such as spectrum-based fault localization [12, 18, 26, 27] and breakpoint recommendation [14, 35]. Different from those approaches, DeepDebugger focus on the model debugging functionalities, by leveraging the time-travelling visualizers [48, 49] for debugging deep classifiers.

8 CONCLUSION

We introduce DeepDebugger, the first interactive debugging tool that can cooperate with any time-travelling visualization solution for debugging deep classifiers. DeepDebugger recommends potential samples of interest, and iteratively and interactively refine the recommendation with the user feedback. Through extensive quantitative experiments and a case study, we also show the effectiveness, efficiency, and robustness of DeepDebugger.

9 DATA AVAILABILITY

Our code and full results can be downloaded on an anonymous website [1, 2]. A tutorial on DeepDebugger is also available.

ACKNOWLEDGMENTS

This research is supported in part by National Natural Science Foundation of China (62172099, 23Z990203011), the Minister of Education, Singapore (T2EP20120-0019, MOET32020-0004), NUS-NCS Joint Laboratory for Cyber Security, Singapore, the National Research Foundation, Singapore, and Cyber Security Agency of Singapore under its National Cybersecurity Research and Development Programme (Award No. NRF-NCR_TAU_2021-0002) and A*STAR, CISCO Systems (USA) Pte. Ltd and National University of Singapore under its Cisco-NUS Accelerated Digital Economy Corporate Laboratory (Award I21001E0002), National Research Foundation, Singapore, and the Cyber Security Agency under its National Cybersecurity R&D Programme (NCRP25-P04-TAICeN). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore and Cyber Security Agency of Singapore.

REFERENCES

[1] [n. d.]. Github Repository of DeepDebugger. https://github.com/code-philia/deep-debugger. Accessed: 2023-09-01.

[2] [n. d.]. Website for DeepDebugger. https://sites.google.com/view/deep-debugger/home. Accessed: 2023-09-01.

[3] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.

[4] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. 2007. On the accuracy of spectrum-based fault localization. In *Testing: Academic and industrial conference practice and research techniques-MUTATION (TAICPART-MUTATION 2007)*. IEEE, 89–98.

[5] Julius Adebayo, Michael Muelly, Ilaria Liccardi, and Been Kim. 2020. Debugging tests for model explanations. *arXiv preprint arXiv:2011.05429* (2020).

[6] Muhammad Adnan, Duaa H AlSaeed, Heyam H Al-Baity, and Abdur Rehman. 2021. Leveraging the Power of Deep Learning Technique for Creating an Intelligent, Context-Aware, and Adaptive M-Learning Model. *Complexity* 2021 (2021).

[7] Cyrille Artho. 2011. Iterative delta debugging. *International Journal on Software Tools for Technology Transfer* 13, 3 (2011), 223–246.

[8] Guillaume Bouchard. 2007. Efficient bounds for the softmax function, applications to inference in hybrid models.

[9] Aditya Chattopadhay, Anirban Sarkar, Prantik Howlader, and Vineeth N Balasubramanian. 2018. Grad-cam++: Generalized gradient-based visual explanations for deep convolutional networks. In *2018 IEEE winter conference on applications of computer vision (WACV)*. IEEE, 839–847.

[10] Li Deng. 2012. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine* 29, 6 (2012), 141–142.

[11] Ruigang Fu, Qingyong Hu, Xiaohu Dong, Yulan Guo, Yinghui Gao, and Biao Li. 2020. Axiom-based grad-cam: Towards accurate visualization and explanation of cnns. *arXiv preprint arXiv:2008.02312* (2020).

[12] Liang Gong, David Lo, Lingxiao Jiang, and Hongyu Zhang. 2012. Interactive fault localization leveraging simple user feedback. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 67–76.

[13] Jun Han and Claudio Moraga. 1995. The influence of the sigmoid function parameters on the speed of backpropagation learning. In *International workshop on artificial neural networks*. Springer, 195–201.

[14] Dan Hao, Lingming Zhang, Lu Zhang, Jiasu Sun, and Hong Mei. 2009. VIDA: Visual interactive debugging. In *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 583–586.

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

[16] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[17] Renáta Hodován and Ákos Kiss. 2016. Modernizing hierarchical delta debugging. In *Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation*. 31–37.

[18] Ferenc Horváth, Árpád Beszédes, Béla Vancsics, Gergő Balogh, László Vidács, and Tibor Gyimóthy. 2022. Using contextual knowledge in interactive fault localization. *Empirical Software Engineering* 27, 6 (2022), 1–69.

[19] Andrew Ko and Brad Myers. 2008. Debugging Reinvented: Asking and Answering Why and Why Not Questions about Program Behavior. In *2008 ACM/IEEE 30th International Conference on Software Engineering*. IEEE, 301–310.

[20] Amy J Ko and Brad A Myers. 2008. Source-level debugging with the whyline. In *Proceedings of the 2008 international workshop on Cooperative and human aspects of software engineering*. 69–72.

[21] Amy J Ko and Brad A Myers. 2009. Finding causes of program output with the Java Whyline. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 1569–1578.

[22] Amy J Ko and Brad A Myers. 2010. Extracting and answering why and why not questions about Java program output. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 20, 2 (2010), 1–36.

[23] Alex Krizhevsky, Geoffrey Hinton, et al. 2009. Learning multiple layers of features from tiny images. (2009).

[24] Henry David Kurzhals. 2022. Challenges and approaches related to AI-driven grading of open exam questions in higher education: human in the loop. (2022).

[25] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *nature* 521, 7553 (2015), 436–444.

[26] Xiangyu Li, Shaowei Zhu, Marcelo d'Amorim, and Alessandro Orso. 2018. Enlightened debugging. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 82–92.

[27] Jingjing Liang, Ruyi Ji, Jiajun Jiang, Shurui Zhou, Yiling Lou, Yingfei Xiong, and Gang Huang. 2021. Interactive Patch Filtering as Debugging Aid. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 239–250.

[28] Yun Lin, Jun Sun, Lyly Tran, Guangdong Bai, Haijun Wang, and Jinsong Dong. 2018. Break the dead end of dynamic slicing: Localizing data and control omission bug. In *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*. 509–519.

[29] Yun Lin, Jun Sun, Yinxing Xue, Yang Liu, and Jinsong Dong. 2017. Feedback-based debugging. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 393–403.

[30] Cheng-Yuan Liou, Wei-Chen Cheng, Jiun-Wei Liou, and Daw-Ran Liou. 2014. Autoencoder for words. *Neurocomputing* 139 (2014), 84–96.

[31] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. 2021. Swin transformer: Hierarchical vision transformer using shifted windows. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 10012–10022.

[32] Stuart Lloyd. 1982. Least squares quantization in PCM. *IEEE transactions on information theory* 28, 2 (1982), 129–137.

[33] Aji Mubarek Mubalaike and Esref Adali. 2018. Deep Learning Approach for Intelligent Financial Fraud Detection System. In *2018 3rd International Conference on Computer Science and Engineering (UBMK)*. 598–603. https://doi.org/10.1109/UBMK.2018.8566574

[34] Daniel Neimark, Omri Bar, Maya Zohar, and Dotan Asselmann. 2021. Video transformer network. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 3163–3172.

[35] Fabio Petrillo, Yann-Gaël Guéhéneuc, Marcelo Pimenta, Carla Dal Sasso Freitas, and Foutse Khomh. 2019. Swarm debugging: The collective intelligence on interactive debugging. *Journal of Systems and Software* 153 (2019), 152–174.

[36] Selim Reza, Marta Campos Ferreira, JJM Machado, and João Manuel RS Tavares. 2022. A multi-head attention-based transformer model for traffic flow forecasting with a comparative analysis to recurrent neural networks. *Expert Systems with Applications* 202 (2022), 117275.

[37] Frank Schneider, Felix Dangel, and Philipp Hennig. 2021. Cockpit: A Practical Debugging Tool for the Training of Deep Neural Networks. *Advances in Neural Information Processing Systems* 34 (2021), 20825–20837.

[38] Ramprasaath R Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. 2017. Grad-cam: Visual explanations from deep networks via gradient-based localization. In *Proceedings of the IEEE international conference on computer vision*. 618–626.

[39] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[40] Hendrik Strobelt, Sebastian Gehrmann, Michael Behrisch, Adam Perer, Hanspeter Pfister, and Alexander M Rush. 2018. Seq2seq-vis: A visual debugging tool for sequence-to-sequence models. *IEEE transactions on visualization and computer graphics* 25, 1 (2018), 353–363.

[41] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

[42] Vladimir Vovk. 2013. Kernel ridge regression. In *Empirical inference*. Springer, 105–116.

[43] Juan José Vázquez, Jamie Arjona, MªPaz Linares, and Josep Casanovas-Garcia. 2020. A Comparison of Deep Learning Methods for Urban Traffic Forecasting using Floating Car Data. *Transportation Research Procedia* 47 (2020), 195–202. https://doi.org/10.1016/j.trpro.2020.03.079 22nd EURO Working Group on Transportation Meeting, EWGT 2019, 18th – 20th September 2019, Barcelona, Spain.

[44] Jia Wang, Tong Sun, Benyuan Liu, Yu Cao, and Hongwei Zhu. 2021. CLVSA: a convolutional LSTM based variational sequence-to-sequence model with attention for predicting trends of financial markets. *arXiv preprint arXiv:2104.04041* (2021).

[45] Han Xiao, Kashif Rasul, and Roland Vollgraf. 2017. *Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms*. arXiv:cs.LG/1708.07747 [cs.LG]

[46] Zhaogui Xu, Shiqing Ma, Xiangyu Zhang, Shuofei Zhu, and Baowen Xu. 2018. Debugging with intelligence via probabilistic inference. In *Proceedings of the 40th International Conference on Software Engineering*. 1171–1181.

[47] Haoyang Yan, Xiaolei Ma, and Ziyuan Pu. 2021. Learning dynamic and hierarchical traffic spatiotemporal features with transformer. *IEEE Transactions on Intelligent Transportation Systems* (2021).

[48] Xianglin Yang, Yun Lin, Ruofan Liu, and Jin Song Dong. 2022. Temporality Spatialization: A Scalable and Faithful Time-Travelling Visualization for Deep

Classifier Training. In *The 31st International Joint Conference on Artificial Intelligence (IJCAI)*.

[49] Xianglin Yang, Yun Lin, Ruofan Liu, Zhenfeng He, Chao Wang, Jin Song Dong, and Hong Mei. 2022. DeepVisualInsight: Time-Travelling Visualization for Spatio-Temporal Causality of Deep Classification Training. In *The Thirty-Sixth AAAI Conference on Artificial Intelligence (AAAI)*.

[50] Andreas Zeller. 1999. Yesterday, my program worked. Today, it does not. Why? *ACM SIGSOFT Software engineering notes* 24, 6 (1999), 253–267.

[51] Mengshi Zhang, Yaoxian Li, Xia Li, Lingchao Chen, Yuqun Zhang, Lingming Zhang, and Sarfraz Khurshid. 2019. An empirical study of boosting spectrum-based fault localization via pagerank. *IEEE Transactions on Software Engineering* 47, 6 (2019), 1089–1113.

[52] Qiuyue Zhang, Chao Qin, Yunfeng Zhang, Fangxun Bao, Caiming Zhang, and Peide Liu. 2022. Transformer-based attention network for stock movement prediction. *Expert Systems with Applications* 202 (2022), 117239.

[53] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. 1996. BIRCH: an efficient data clustering method for very large databases. *ACM sigmod record* 25, 2 (1996), 103–114.

[54] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. 1996. BIRCH: An Efficient Data Clustering Method for Very Large Databases. *SIGMOD Rec.* 25, 2 (jun 1996), 103–114. https://doi.org/10.1145/235968.233324

Received 2023-02-02; accepted 2023-07-27