TRADER: Trace Divergence Analysis and Embedding Regulation for Debugging Recurrent Neural Networks

Guanhong Tao
tao@purdue.edu
Purdue University

Shiqing Ma
shiqing.ma@rutgers.edu
Rutgers University

Yingqi Liu
liu1751@purdue.edu
Purdue University

Qiuling Xu
xu1230@purdue.edu
Purdue University

Xiangyu Zhang
xyzhang@cs.purdue.edu
Purdue University

ABSTRACT

Recurrent Neural Networks (RNN) can deal with (textual) input with various length and hence have a lot of applications in software systems and software engineering applications. RNNs depend on word embeddings that are usually pre-trained by third parties to encode textual inputs to numerical values. It is well known that problematic word embeddings can lead to low model accuracy. In this paper, we propose a new technique to automatically diagnose how problematic embeddings impact model performance, by comparing model execution traces from correctly and incorrectly executed samples. We then leverage the diagnosis results as guidance to harden/repair the embeddings. Our experiments show that TRADER can consistently and effectively improve accuracy for real world models and datasets by 5.37% on average, which represents substantial improvement in the literature of RNN models.

ACM Reference Format:

1 INTRODUCTION

Deep learning (DL) models are becoming an integral part of many modern computing systems. For example, a self-driving car system often makes use of DL models to recognize objects and even maneuver vehicles; online advertisement leverages DL models to identify potential customers and deliver the corresponding ads; latest mobile/wearable devices use various DL techniques to authenticate users, detect and monitor user behaviors. Engineering DL models is becoming a critical step of engineering such intelligent computing systems. Among the various kinds of DL models, Recurrent Neural Networks (RNNs) are particularly useful in software related applications as they are designed to deal with textual inputs (of arbitrary length) and inputs in sequence. Note that many software artifacts are in the form of text or sequences. For example, a program can be considered as a piece of text in some special language; program comments are essentially texts in natural language with specific semantics; and execution traces are a sequence of values of arbitrary length. As such, RNN models find their way to many software engineering applications.

Wang et al. [88] leveraged RNNs to construct semantic program embeddings, which are a way to encode textual programs to numerical values. The embeddings were further employed in a search-based program repair system to correct errors in programs [88]. Henkel et al. [29] used a DL model called GloVe [65] to construct word embeddings for abstract symbolic traces of programs. The learned embeddings were used to find bugs, which were further confirmed by traditional static analysis. Panichella et al. [61] employed a model [72] to assess the polarity of app reviews, which can provide developers with informative feedback to improve application quality and facilitate software maintenance. Du et al. [21] modeled RNN as an abstract state transition system for test generation and adversarial sample detection. Tian et al. [83] proposed a testing tool for automatically detecting erroneous behaviors of DNN-driven vehicles with CNN or RNN as the internal model. RNN models [36, 49, 72] are also widely used in processing textual software artifacts, such as code comments [33, 60, 81], developer commits [75], and app reviews [27, 61].

DL model reliability is hence a critical part of the overall reliability of many software systems. Just like software, DL models may have undesirable behaviors, such as exceptionally low test accuracy. They are called model bugs in the literature [52]. Such model bugs may lead to undesirable system-wide behaviors. For example, SentiStrength [82] is a state-of-the-art tool that can predict sentiment of developer comments. Such sentiment information is further used to extract problematic API features [102]. However, a recent study [49] showed that its underlying model achieved recall and precision lower than 40% on negative sentences. Since negative sentences are critical indicators for problematic APIs, the low model accuracy will cause many problems for the downstream analyses.

Different from normal programs, DL models are difficult to debug due to their "black box" (unexplainable) nature [3, 6, 68]. Most existing works focused on providing more data to improve model performance [16, 25]. Generative adversarial networks (GANs) [25] are also widely used to generate additional data for further training [41]. However, these methods can hardly be considered as debugging techniques as they lack the diagnosis step that identifies the root cause. MODE [52] is a recent model debugging technique for...
convolutional neural networks (CNNs) that often deal with fixed size inputs such as images. It leverages differential analysis to identify faulty neurons and selects additional training inputs to "correct" such neurons' behavior. However, MODE cannot handle RNNs.

In this paper, we focus on debugging RNN models for textual inputs (e.g., sentiment analysis for developer comments), especially for a type of bugs in which problematic word embeddings lead to suboptimal model accuracy. Many studies [5, 73, 86, 96] have shown that word embeddings are critical for RNN model accuracy. For those textual tasks, input words/tokens are always converted to embeddings first before training/testing. Although embeddings seem external to the model, the model itself cannot be studied/debugged separately without considering embeddings. Inspired by software debugging [10, 18, 23], we view an RNN model as a program with very specific semantics. Therefore, the traditional trace analysis in software debugging can be adopted to debug RNN models, as long as the specific semantics are properly modeled. In particular, given a buggy model, our technique performs trace divergence analysis that identifies the problematic internal model state vector dimensions responsible for the misclassification, called the faulty state dimensions. Intuitively, the state vectors are a number of buffers that are updated after the model processes each word or input element. A dimension is essentially a buffer element at a specific index, which semantically encodes some feature of the current input word and its preceding context. Hence, the faulty dimensions represent the features (of words and their contexts) that the model has confusing/buggy behavior. Then an embedding regulation algorithm is proposed to mitigate the problem by hardening the model behaviors for those dimensions. Intuitively, it applies small mutations to those dimensions and then forces the model to learn to disambiguate the perturbations. The more the model can disambiguate, the better accuracy it can achieve (as it is less confused on those dimensions).

Our contributions are summarized in the following.

- We identify buggy behaviors of RNN models through a trace divergence analysis, and locate faulty state dimensions responsible for misclassification.
- We propose an RNN repair technique, a new training procedure that freezes model parameters and regulates word embeddings according to observed trace divergences.
- We develop a prototype TRADER (TRAce Divergence analysis and Embedding Regulation). Experimental evaluations are conducted on five public datasets, three word embeddings, and three model structures, with a total of 135 models. TRADER can consistently and effectively improve the performance by 5.37% on average, substantially outperforming a state-of-the-art embedding regulation technique based on four regularization strategies [63], which improves model accuracy by 0.6% on average. Note that due to the need of dealing with inputs of arbitrary length, it is challenging to improve RNN accuracy in general. Most reported improvement in the literature (not using debugging techniques, but rather new model architecture or new optimizers) range from 0.05%-3.76% with a median of 0.7% [17, 38, 39, 51, 64, 94].
- Our implementation, datasets, configurations, and model checkpoints are publicly available at [84].

2 BACKGROUND

2.1 Recurrent Neural Networks

Recurrent neural networks (RNNs) are a family of neural networks designed to tackle problems with sequential inputs. Different from traditional neural networks that require fixed-length inputs, RNNs can handle sequential inputs of arbitrary length. That is, input is received continuously through many steps, via a loop structure as shown in Figure 1a. The sequential inputs are fed into the RNN model one by one. At each time step, the model leverages the previous hidden value and the current input so as to update the hidden value. For specific tasks, the prediction outputs are obtained by adding an output layer using the hidden value, such as in machine translation [4, 79], speech recognition [30], and image caption generation [95]. More specifically, $x_t$ (blue circle) is the input at step $t$ and $h_t$ (green circle) is the hidden value. Intuitively, $h_t$ encodes the history/context in the previous $t−1$ steps. At each time step, an input $x_t$ and the previous hidden value $h_{t−1}$ are fed into the RNN cell (yellow rectangle), and the hidden value for the next step $h_t$ is computed. A vanilla RNN cell contains a regular feed-forward neural network with layers of neurons. We can obtain the hidden value $h_t$ at step $t$ using the following formula.

$$h_t = \sigma(W_h \cdot [h_{t−1}, x_t] + b_h),$$

(1)

where $\sigma$ is the activation function. $W_h$ denotes the weight matrix and $b_h$ the bias. The operation $[\cdot]$ concatenates two vectors. For a specific task (e.g., sentiment analysis), the final prediction is computed using the last hidden value $h_n$:

$$\text{prediction} = W \cdot h_n + b,$$

(2)

where $W$ and $b$ are the weight matrix and bias of the output layer, respectively; and $n$ denotes the length of an input sequence which can be arbitrarily large. The output prediction is normally a vector of logits, and the final predicted class can be obtained by applying function $\text{arg max}(\cdot)$ on the output prediction vector.

Vanilla RNNs are not able to "remember" temporal context of long sequences [8, 31]. In order to deal with long-term dependencies, a new type of RNN model, called Long Short Term Memory (LSTM) networks, was proposed by Hochreiter et al. [32]. LSTMs inherit the same loop structure to deal with arbitrary input length. For the cell structure, instead of using regular feed-forward neural networks, LSTMs are designed with multiple gates to control how much information from the previous and current contexts is being passed on to later computation. Figure 1b illustrates the internal
structure of an LSTM cell. The leftmost two inputs are hidden values from the previous step, where the top $c_{t-1}$ is called cell state and the bottom $h_{t-1}$ is called hidden state. Cell state encodes the contextual information through the entire sequence (a kind of long term memory). Hidden state is similar to the hidden state in a vanilla RNN, representing recent historic information. An LSTM cell can be formalized as follows.

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

(3)

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

(4)

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$

(5)

$$\hat{c}_t = \tanh(W_c \cdot [h_{t-1}, x_t] + b_c)$$

(6)

$$c_t = f_t \odot c_{t-1} + i_t \odot \hat{c}_t$$

(7)

$$h_t = o_t \odot \tanh(c_t)$$

(8)

where operators $\cdot$ and $\odot$ denote matrix multiplication and element-wise vector multiplication, respectively; $\sigma()$ and $\tanh()$ denote sigmoid and tanh activation functions that crop/normalize activation values; $f_t$ denotes the forget gate that controls how much information from previous steps needs to be forgotten/remembered; $i_t$ acts as the input gate that determines how information needs to be added to cell state; $o_t$ is the output gate that decides the degree of output information being accumulated to hidden state; cell state $c_t$ is updated according to the preceding cell state $c_{t-1}$ and the current context $\hat{c}_t$; and finally the hidden state $h_t$ is updated based on current cell state $c_t$ and output $o_t$.

### 2.2 Word Embeddings

In Natural Language Processing (NLP) tasks, the inputs are normally texts containing various numbers of words. Existing machine learning (ML) models (e.g., RNNs) require numerical inputs so as to do mathematical computation. To integrate NLP tasks into ML models, word embeddings are adopted to address this issue. That is, each word is represented as a numerical vector. For instance, in the sentence “I like movies”, word “I” is represented as [1, 0, 0], word “like” as [0, 1, 0], and word “movies” as [0, 0, 1]. This type of word embeddings is called one-hot embeddings, where the length of each embedding is the size of the dictionary and only one dimension has value 1. It is straightforward to encode words into one-hot embeddings. Such word embeddings, however, are too sparse for storage and computation. A more concise way of representing words is to leverage all the dimensions of word embeddings with continuous values, which is called distributed representations. For instance, word “I” will be represented as [0.9, 0.3], word “like” as [0.2, 0.4], and word “movies” as [0.5, 0.6]. Researchers have been exploring different approaches to achieve such a dense form of word embeddings. A typical approach is to train a neural network model with a large corpus (e.g., Wikipedia pages). The task of the neural network is to predict the center word given a sequence of (usually 5) words in a sentence. The learned weight of the neural network is regarded as word embeddings [55]. Such learned word embeddings have a nice property that words with similar meanings have small embedding distances (e.g., Euclidean distance between two word embeddings). This allows the model to generalize. Intuitively, even though a model may not have seen some words/sentences during training, it can still perform prediction based on their embedding neighbors that appear during training.

Distributed representations of word embeddings are widely used in NLP tasks [76, 87, 93] as well as software engineering (SE) tasks [1, 28, 34, 91, 92]. According to many studies [5, 73, 86, 96], word embeddings are the dominating factor in model accuracy in RNN applications. For example in sentiment analysis, according to Schneabel et al. [73], the same ML model using different word embeddings as features can have divergent prediction accuracy ranging from 62.95% to 88.90%, indicating its importance. Figure 2 demonstrates examples of words represented using the GloVe word embeddings [65]. The top row denotes the target words and the following rows are the nearest neighbors of target words measured by cosine similarity. Column Original lists the nearest words based on the original GloVe word embeddings, while column Regulated are based on our regulated word embeddings. It can be observed that for target word “error”, word “correct” is the second nearest word using GloVe while it is moved down the list after embedding regulation. Word “little” is the third nearest word for “great” using GloVe and it is not in the top list after regulation. It is similar for the case “reinstall”. One can easily tell that models generalize better with the regulated embeddings.

#### 2.3 Model Debugging

Just like software inevitably contains bugs and software debugging is a key step in software development, DL models may have undesirable behaviors, called model bugs [52]. Model debugging is becoming an essential step in intelligent software engineering. Model bugs are different from traditional coding bugs. They are misconducts in the model engineering process, such as biased training data [52] and problematic model structure, which lead to undesirable consequences such as low model accuracy and vulnerabilities to adversarial sample attacks [26, 80], in which normal inputs are mutated (e.g., by perturbations not human perceptible) to induce mis-classification. In our context, model debugging is a procedure to study model internals to understand the root cause of mis-classification and then conduct counter-measure to “fix” the root cause.

Model debugging is difficult as DL models are not interpretable [6, 68]. There are techniques that use data augmentation (e.g., image reflection, cropping, and rotations for CNN models) to provide additional data to improve model performance [16, 25, 43]. Another method is to use GANs [25] to generate additional training data [89, 103]. However, these methods are not feedback driven, meaning that they do not intend to understand what causes the low accuracy before trying to fix the problem, which limits their effectiveness.
MODE [52] is a recent feed-back driven technique for CNNs. It analyzes model internals to identify "faulty" neurons and then selects additional training inputs to "correct" such neuron behaviors. These existing approaches, including MODE, are mostly designed to improve image-related models (e.g., handwritten digit recognition and object classification). For text-related models (e.g., sentiment analysis), they are hardly applicable. For example, using GANs to generate text inputs often suffers from low quality and lack of diversity [89]. According to Wang et al. [89], most generated texts have fewer than 15 words, whereas sentences in real-world training datasets have more than a few hundreds words.

In this paper, we focus on debugging text input oriented RNN models that have a lot of software engineering applications [9, 29, 35, 36, 47, 49, 88, 90]. While there are many possible kinds of bugs for these models, such as biased training inputs [52], sub-optimal model structures, and incorrect hyper parameter settings. We focus on problematic word embeddings as the literature has indicated that embeddings are critical for RNN model accuracy [5, 73, 86, 96].

In our view, an RNN model is essentially a program with special semantics. Therefore, our overarching idea is to adapt existing software debugging techniques (e.g., [10, 18, 23]), especially execution trace analysis, to debug RNN models, by properly modeling RNN models’ special semantics. Next, we will use a program example and an RNN example side-by-side to intuitively illustrate our idea.

The left part of Figure 3 shows a simple program for counting the number of data points within range [1, 2]. The functionality of method `embedding()` at line 4 is to convert string values to numerical values and then map them to discrete values. The mapping can be implemented as a rounding operation, where values are converted to their closest discrete numbers. For instance, value 1.1 becomes 1 while 1.8 becomes 2. When the developer executes this piece of code, value 2 is printed as the output, which is incorrect. To locate the bug, the developer prints out the value of variable `output` at each iteration of the loop, and obtains the value trace of [1, 1, 2, 2]. For this simple program, the developer has the oracle that the value trace of variable output should be [1, 1, 2, 2, 3]. By comparing the actual trace to the oracle, the developer can easily locate the trace diverged at the fourth iteration and finally identify the buggy implementation of method `embedding()`.

Interestingly, RNNs have a very similar loop structure as shown on the right of Figure 3. Method `embedding()` at line 6 maps each word to its corresponding embedding. Method `concatenate()` at line 7 concatenates two vectors and `matmul()` is the matrix multiplication method. Analogous to the simple program on the left, we can use the same trace divergence analysis to inspect the state at each iteration if undesired behaviors happen. Figure 4 demonstrates a sample text from the Stack overflow dataset [49]: "Also, JodaTime makes calculations with time much simpler". It is labeled as having positive sentiment. An LSTM model [19], however, predicts a neutral sentiment. We record the state values at each iteration (i.e., after each word) and also query the oracle for the same step. The green line in Figure 4 shows the trace of the LSTM model, while the blue line is the trace acquired from the oracle. As we will discuss in Section 3.1, having an oracle model that always produces the correct intermediate model states is infeasible, just like having a correct reference program for regular program debugging is infeasible in general. In the literature, various techniques were proposed to approximate the reference (e.g., using a similar but correct execution as in Delta Debugging [56, 78, 97]). Similarly, in our context of RNN model debugging, we train a model from validation data set to approximate the reference model (see Section 3.1). From Figure 4, it can be observed that at the step with word "much", the two traces start to diverge, which finally leads to different output labels.

We hence further inspect the state differences at the divergence step. Figure 5 presents the comparison. In the figure, the input sentence ¶ is in the middle. The blue arrow to ✄ denotes the model state after the word "much". Here, a model state is the concatenation of the input, cell state, and hidden state vectors [x_t, c_{t-1}, h_{t-1}], followed by the output (1 denotes neutral sentiment and 2 denotes

---

1 A data and time library for Java. https://www.joda.org/joda-time/
construct two models, called the \textit{oracle machine} and the \textit{buggy machine}, which approximate the distributions of state values from the correctly classified samples and from misclassified samples, respectively. These two machines are the reference models for identifying diverged steps (Section 3.1). We aggregate all the diverged steps of misclassified samples in the validation dataset, and inspect the difference of their state vectors to identify the critical dimensions, which have large aggregated differences. We consider them the \textit{faulty state dimensions}. Intuitively, they denote the sub-space that the model gets “confused”. In the fixing step, we target on further training these faulty dimensions to alleviate the confusion. Specifically, we add small perturbations to these dimensions and then retrain the word embeddings using the \textit{original training set} so that the perturbations only cause minimal output variations. Intuitively, we are tuning the embeddings so that the model becomes more affirmative and have stable prediction even when confusion (perturbation) is intentionally injected in the faulty dimensions. As we observed during experiments, identifying the faulty dimensions is critical as perturbing all dimensions leads to accuracy degradation.

One may wonder why not simply train the model using the validation set or even both the training set and validation set. Note that the essence of our technique is not to leverage the additional samples in the validation set to train. Instead, we utilize the validation set just to locate the dimensions that do not generalize well to new data and then further harden these dimensions. In fact, we will show in Section 4 that training the model using both the training set and the validation set cannot achieve the same level of improvement as TRADER.

3.1 Trace Divergence Analysis

DL models are normally trained on a \textit{training set} and then tested on a \textit{test set}, which is unseen to DL models during training. It is essential to have another set (also unseen during training) for debugging models and avoiding over-fitting on the training set, which is referred to as the \textit{validation set}. Following a similar philosophy, our technique leverages the validation set for model debugging.

As shown in Figure 7, text samples in the validation set are fed to the model. By comparing to the ground truth labels, we can acquire two sets of samples: the correctly classified samples (in the top box) and the misclassified samples (in the bottom box). These text samples can be further processed by the same model separately to record their traces. More specifically, given a text sample, for each step an input word is fed into the model, we record the input embedding $x_t$, and the previous contexts $c_{t-1}$ and $h_{t-1}$ as the state vector $p_t$. The output vector $q_t$ produced by the output layer is also recorded. The state vector $p_t$ and the output vector $q_t$ are regarded as a trace entry. Hence, a sequence of trace entries can be generated for each input sample. We use the same procedure to generate two separate trace sets, called the \textit{oracle traces} (generated from the correctly classified samples) and the

![Figure 6: Overview of TRADER.](image-url)
Also, JodaTime makes… But I want to connect… I’m looking for a well… However, I’d suggest… As others have already…

Figure 7: Trace generation for samples in the validation set. Words in input sentences are mapped to their corresponding word embeddings \( x_t \). Internal states \( c_t \) and \( h_t \) are the initial internal state vectors for LSTM models. The output layer is used to predict the final output label as discussed in Equation 2. Here, it has been extended to the whole sequence for acquiring the internal output at each step.

Figure 8: Construction of oracle machine and buggy machine, and trace divergence analysis. Trace divergence analysis is conducted on a sample text by feeding its traces to the two machines and comparing the output values.

buggy traces (from the misclassified samples). These two set of traces are crucial for debugging RNN models as they represent models’ internal behaviors on unseen data.

We utilize the two sets of traces acquired from the validation set to learn the distributions of model’s proper and buggy behaviors. Figure 8 illustrates the procedure of trace divergence analysis. To model the distribution of oracle/buggy traces, we employ the linear regression approach to approximate the relation between state vector \( p_t \) and output value \( q_t \) using the following equation:

\[
q_t = W_p \cdot p_t + b_r, \tag{9}
\]

where \( W_r \) and \( b_r \) are weight and bias of linear regression, respectively. These parameters will be updated based on all the traces, and each dimension of weight \( W_r \) denotes the importance of the corresponding state vector dimensions with respect to the output value. The fitted models for oracle (buggy) traces are called oracle (buggy) machine. In other words, these machines predict output \( q_t \) from given state \( p_t \). Note that they are not RNN models but rather simple classifiers to predict one step of model behavior. The two machines approximate the desired and undesired behaviors of the model, respectively, in the presence of unseen data.

For a given misclassified sample from the validation set (boxed text in the bottom), traces are extracted from the RNN model, and fed to both the oracle machine and the buggy machine. We then compare the outputs from the two machines to identify the diverged steps. Note that such divergence cannot be directly identified by monitoring the original model operations.

As we aim to identify the root cause of trace divergence, i.e., the faulty dimensions, we aggregate the state vectors from the diverged steps. Algorithm 1 details the aggregation procedure. The algorithm loops over all the misclassified samples in the validation set. For each sample, it first generates the corresponding traces from the RNN model (line 4). The trace divergence analysis is conducted on each step. The Oracle machine (OM) is provided with the state vector \( p \) and outputs an oracle value \( q_o \) (line 6). Similarly, the Buggy machine (BM) is also provided with the same state vector \( p \), and outputs a buggy value \( q_b \) (line 7). These two output values \( q_o \) and \( q_b \) are compared to identify the diverged steps. Those diverged steps are aggregated in \( d \) (line 9). The comparison is needed because not all the steps in a buggy trace are wrong. In line 10, we acquire the weights from both the oracle machine and the buggy machine, which indicate the importance of state vector for oracle traces and buggy traces. Intuitively, the difference between these two weight vectors denotes the importance divergence of the individual dimensions of state vector (line 11). At the end, we multiple \( d \) with \( w_d \) to compute the weighted differences for individual dimensions (line 12). The faulty dimensions are the ones with exceptionally large values. Figure 10 shows an example of vector \( d \) for an LSTM model (the red bars). Observe that there are a number of dimensions that have much larger values than the others. They denote the split/confusing behaviors of the subject model in the presence of
unseen data. While these dimensions are not human interpretable, they provide sufficient guidance for embedding tuning as shown in the next section.

3.2 Embedding Regulation

After identifying the faulty state vector dimensions, which intuitively are the places the model is very unstable and has diverging behaviors for correctly and incorrectly classified samples, we next aim to mitigate the problem by regulating word embeddings. The essence of state divergence lies in that the model is so sensitive for these dimensions that small changes can lead to substantial output changes. A key feature of RNN type of models is that the encoded state vectors are largely determined by word embeddings. We propose a divergence loss to propagate error information from the two outputs is then propagated to the input word embeddings. Since RNN type of models have a loopy structure, the perturbations are applied to each iteration.

Figure 9 demonstrates the detailed procedure of our embedding regulation. The left LSTM model is the original model, which takes an input embedding \( x_t \) at each time and outputs the final result \( q \). The right LSTM model is the perturbed LSTM model, where a perturbation vector \( \epsilon \) is added to the input embedding \( x_t \) and a perturbation vector \( \lambda \) is added to both cell state \( c_{t-1} \) and hidden state \( h_{t-1} \) at each step. The output \( q_\delta \) from the perturbed model is compared to the output \( q \) from the original model. The difference between the two outputs is then propagated to the input word embeddings. We propose a divergence loss to propagate error information from the two differences to the input embeddings. The divergence loss is formalized as follows.

\[
L_{div} = L_{ce} + L_{L2},
\]

where \( L_{ce} \) is cross entropy loss [24] and \( L_{L2} \) is L2 (squared error) loss. The loss functions are used to express our objective to minimize output variations in the presence of perturbations. Cross entropy loss is widely used in classification tasks. The following formula defines its essence.

\[
L_{ce} = \frac{1}{N} \sum_{n=1}^{N} \sum_{c=1}^{C} y_n^{c} \ln(p_n^{c}),
\]

where \( N \) is the number of training samples, and \( C \) is the number of classes. For each sample \( n \), if the final output \( o \) is different from ground truth label \( c \), then \( y_n^{c} = 1 \); otherwise \( y_n^{c} = 0 \). \( p_n^{c} \) is the predicted probability of output \( o \) at class \( c \). Intuitively, cross entropy loss gauges the scale of output differences. Note that in our method, perturbations are added to faulty dimensions for all the steps. Such perturbations may accumulate over time and have inappropriate impact on the final output. We hence employ L2 (squared error) loss to reduce the influence from perturbations. It is commonly used for regression tasks [46, 59]. In our scenario, it is formalized as follows.

\[
L_{L2} = \frac{1}{2} \sum_{n=1}^{N} \sum_{c=1}^{C} (\hat{p}_n^{c} - p_n^{c})^2,
\]

where \( N \) is the number of training samples, and \( C \) is the number of classes. \( \hat{p}_n^{c} \) is the original prediction, and \( p_n^{c} \) is the predicted probability. Intuitively, it minimizes the difference between the original prediction and the perturbed prediction.

DL models are usually trained by minimizing the loss function, during which model parameters are updated through backpropagation [45]. Different from general DL training, we introduce a new training procedure, where model parameters are frozen and only input embeddings are updated during training, as our purpose is to regulate input embeddings according to the divergence loss. That is, the gradients calculated from our divergence loss is backpropagated to only the embedding variables, which are updated during training. Algorithm 2 illustrates the procedure of embedding regulation. In Section 3.1, aggregated divergence analysis is conducted on all the misclassified samples in the validation set. A vector of dimension divergence is generated for identifying faulty state vector dimensions. Here, we utilize this divergence vector to only apply perturbations to the faulty dimensions, which are essentially the most influential dimensions. More specifically, we traverse over all the dimensions and find the ones that are faulty (line 4-5). For dimensions denoting input \( x_t \), a random value sampled from a normal distribution \( \mathcal{N}(0, \epsilon) \) (i.e., mean of 0, variance of \( \epsilon \)) is added to those dimensions (line 7-8). The variance value \( \epsilon \) is chosen based on the standard

---

Algorithm 2 Embedding Regulation

1. function Regulation(model \( M \), dataset \( S \), embedding \( E \), divergence \( D \))
2. \( u \leftarrow \) zero vector with dimension size of embedding
3. \( v \leftarrow \) zero vector with dimension size of hidden neurons
4. for \( d \in D \) do
5. if \( d > 0 \) then
6. \( idx \leftarrow D.index(d) \)
7. if \( idx \) belongs to \( E \) then
8. \( u[idx] \leftarrow random.normal(0, \epsilon) \)
9. else
10. \( v[idx] \leftarrow random.normal(0, \lambda) \)
11. \( E \leftarrow get\_variable(E) \)
12. \( E \leftarrow add\_vector(E, u) \)
13. \( c_t, h_t \leftarrow \) initialized with dimension size of hidden neurons
14. for \( i \in range(max\_step) \) do
15. \( c_t \leftarrow add\_vector(c_t, v) \)
16. \( h_t \leftarrow add\_vector(h_t, v) \)
17. \( c_t, h_t \leftarrow LSTM(c_t, h_t) \)
18. \( S \leftarrow embedding(S, E) \)
19. \( M.freezef() \)
20. \( LSTM.train(M, S, L_{div}) \)
21. return \( E \)
deviation of original word embeddings as it should not overshadow the original values. For dimensions denoting the internal states, a random value sampled from a normal distribution with mean of 0 and variance of $\lambda$ is added to those dimensions. Different from $\epsilon$ that is only added to the input vector, the variance value $\lambda$ is added to internal states at each step. A large value of $\lambda$ will accumulate over time and can significantly affect normal behaviors of models. Thus, $\lambda$ is much smaller than $\epsilon$. Lines 11-12 apply the input perturbations. RNN kind of models have a loopy structure where internal states are computed through multiple steps. Thus, for internal hidden states, perturbations are added for every step (line 14-17). Training inputs of text tasks are sequences of words, which are mapped to the corresponding word embeddings before training (line 18). Model parameters are frozen during training as we aim to regulate embeddings (line 19). Finally, we leverage the divergence loss to tune word embeddings (line 20). The blue bars in Figure 10 presents the trace divergence analysis results for the same model after embedding regulation. Observe that the significance of the faulty dimensions are substantially reduced.

After we acquire the new embeddings, we freeze the embeddings and retrain the model (by updating model parameters). This is a typical procedure for training RNN type of models when embeddings are changed.

4 EVALUATION

We evaluate TRADER on various datasets, word embeddings, and RNN model structures. Most experiments were conducted on a server equipped with two Xeon E5-2667 3.20GHz 8-core processors, 128 GB of RAM, 2 Tesla K40c GPU, 2 GeForce GTX TITAN X GPU and 4 TITAN Xp GPU cards.

4.1 Setup

We use five datasets, three well-known word embeddings, and three widely used RNN model structures (each having three different settings), with a total of 135 models. The scale of our experiments is much larger than similar works on RNN models [17, 38, 39, 51, 63, 94], which use 2-13 models.

Datasets. Five datasets are employed in the evaluation. Three of them, stack overflow discussions, mobile app reviews, JIRA issue comments, are from the software engineering (SE) community provided by Lin et al. [49]. IMDB dataset [53] is a large dataset for movie reviews. Another dataset, Yelp reviews [100], is one of the largest datasets for sentiment analysis. Table 1 shows the statistics of these five datasets. The app reviews dataset has 341 samples with 3 sentiment classes. The longest sentence in this dataset has 231 words. The IMDB dataset has 50,000 samples with two sentiment classes. The max length of sentences in IMDB is 2,506. The JIRA issues dataset has 926 samples with two sentiment classes. The longest sentence in this dataset has 49 words. The stack overflow dataset has 1,500 samples with three sentiment classes. Most of the samples are from the neutral class. The Yelp reviews dataset is obtained from the Yelp Dataset Challenge in 2015 containing around 6 million samples. The max length of sentences in Yelp is 3,929. For the three SE datasets and the Yelp dataset, we partition them into three disjoint sets: training set (80%), validation set (10%), testing set (10%), which is consistent with the setting in [17, 58, 94].

The original IMDB dataset has already been split into two sets with 25,000 samples for training and 25,000 samples for testing. We follow the convention by preserving the test set, and further partition the training set to two parts: 22,500 samples in training and 2,500 in validation. Thus, the IMDB dataset is split into three parts: training (45%), validation (5%), and testing (50%).

Word Embeddings. Three kinds of word embeddings are studied in our experiments. The GloVe word embedding was proposed by Jeffrey et al. [65]. It leverages statistical information in a large corpus and only trains on the nonzero elements in a word-word occurrence matrix. We employ a pre-trained GloVe embedding from a real-word application [19]. The Word2vec embedding was introduced by Mikolov et al. [54]. It is one of the most widely employed embeddings with many applications [48, 74, 98]. We also obtain a pre-trained word2vec from an existing project [15]. The Adversarial word embedding was especially optimized for text classification [58]. We utilize the original implementation from the authors [57] to train the embedding. Table 2 illustrates the statistics of these three embeddings. GloVe has 50 dimensions for each word, and has the largest mean value compared to the other two embeddings. Word2vec has 300 dimensions, the largest number of dimensions among all three embeddings. The standard deviation of word2vec is smaller than other embeddings, meaning the perturbation variance $\epsilon$ should be small for word2vec. The Adversarial embedding has 256 dimensions. It has the largest standard deviation, meaning a large variance value of perturbation $\epsilon$ should be chosen.

Models. We use three popular RNN model structures, each having three different settings (on the number of hidden neurons, namely, 64, 128, and 256). The vanilla RNN is a basic RNN model with

<table>
<thead>
<tr>
<th>Dataset</th>
<th># samples</th>
<th># negative</th>
<th># neutral</th>
<th># positive</th>
<th>Max length</th>
</tr>
</thead>
<tbody>
<tr>
<td>App reviews</td>
<td>341</td>
<td>138</td>
<td>25</td>
<td>186</td>
<td>231</td>
</tr>
<tr>
<td>IMDB</td>
<td>50,000</td>
<td>25,000</td>
<td>0</td>
<td>25,000</td>
<td>2,506</td>
</tr>
<tr>
<td>JIRA issues</td>
<td>926</td>
<td>836</td>
<td>0</td>
<td>290</td>
<td>49</td>
</tr>
<tr>
<td>Stack overflow</td>
<td>1,500</td>
<td>131</td>
<td>1,191</td>
<td>178</td>
<td>52</td>
</tr>
<tr>
<td>Yelp</td>
<td>5,946,620</td>
<td>1,544,553</td>
<td>0</td>
<td>4,402,067</td>
<td>3,929</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Embedding</th>
<th># dimensions</th>
<th>Mean</th>
<th>Standard deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>GloVe</td>
<td>50</td>
<td>0.020941</td>
<td>0.644104</td>
</tr>
<tr>
<td>Word2vec</td>
<td>300</td>
<td>-0.004118</td>
<td>0.107278</td>
</tr>
<tr>
<td>Adversarial</td>
<td>256</td>
<td>-0.035817</td>
<td>0.767211</td>
</tr>
</tbody>
</table>
works on RNN [19, 77]. The accuracy of the trained models aligns with the Adam optimizer [40] with the learning rate of 0.001. Note that we use a batch size of 24 samples for each training iteration, except for the validation set.

In most cases, a baseline is used for selecting faulty model weights), re-embedding words which minimizes difference of the pre-trained embeddings and the embeddings fine-tuned for embedding regulation. Parameter \( \epsilon \) for embedding regulation. Parameter \( \theta \) for the maximum of the divergence vector. Parameter \( \lambda \) for the norm of weights/embeddings, with \( \epsilon \) ranging from (0, 1) and \( \lambda = 10^{-4} \). The values of \( \epsilon \) and \( \lambda \) are chosen using the validation set. Specifically, we uniformly sample ten values from their range and select the one that produces the best result on the validation set. In most cases, \( \theta \) is close to the mean and \( \epsilon \) is in (0.1, 0.3) depending on the model. Concrete settings can be found in [84]. Note that such parameter tuning is typical in deep learning.

Baseline. We compare our technique with a state-of-the-art RNN hardening technique [63] that does not use debugging feedback, but rather standard model hardening strategies, including penalizing weights/embeddings which adds \( L_2 \)-norm of weights/embeddings to the cost function (e.g., \( L_{new} = L_{old} + ||W||^2 \) where \( W \) is the model weights), re-embedding words which minimizes difference between pre-trained embeddings and the embeddings fine-tuned during supervised training, and dropout which sets each neuron to 0 with a probability \( p \) during training. Since it is a general technique it also works on RNN models (without requiring any model structure enhancement) and reports state-of-the-art results, we use it as the baseline. Note that we cannot use GANs as a baseline like in MODE [52] because high quality GANs for RNN models are still an open challenge, as pointed out in [89].

To reduce the uncertainty introduced by random perturbation. We ran each experiment 10 times and report the average, except Yelp, which we can only afford running it 3 times due to its extremely large size.

4.2 Evaluation of Trace Divergence Analysis

We leverage traces acquired from the validation set to learn the distributions of model behaviors. We first collect the time and space cost of the trace divergence analysis. We then study the effectiveness of the linear regression approaches in approximating distributions of the oracle traces and the buggy traces.

Table 3 presents the overhead introduced by the trace divergence analysis. From the table, it can be observed that for SE datasets, the analysis time is less than 7 seconds and the space overhead is around thousands of MBs. Note that their results have different scales from the others (indicated by the parentheses in column 2). We argue the analysis cost is still reasonable.

<table>
<thead>
<tr>
<th>Overhead</th>
<th>Dataset</th>
<th>GloVe</th>
<th>Word2Vec</th>
<th>Adversarial</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Vanilla RNN</td>
<td>64</td>
<td>128</td>
<td>256</td>
</tr>
<tr>
<td></td>
<td>LSTM</td>
<td>64</td>
<td>128</td>
<td>256</td>
</tr>
<tr>
<td></td>
<td>GRU</td>
<td>64</td>
<td>128</td>
<td>256</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Time (s)</th>
<th>JIRA issues</th>
<th>GloVe</th>
<th>Word2Vec</th>
<th>Adversarial</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Vanilla RNN</td>
<td>64</td>
<td>128</td>
<td>256</td>
</tr>
<tr>
<td></td>
<td>LSTM</td>
<td>64</td>
<td>128</td>
<td>256</td>
</tr>
<tr>
<td></td>
<td>GRU</td>
<td>64</td>
<td>128</td>
<td>256</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Space (MB)</th>
<th>JIRA issues</th>
<th>GloVe</th>
<th>Word2Vec</th>
<th>Adversarial</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Vanilla RNN</td>
<td>64</td>
<td>128</td>
<td>256</td>
</tr>
<tr>
<td></td>
<td>LSTM</td>
<td>64</td>
<td>128</td>
<td>256</td>
</tr>
<tr>
<td></td>
<td>GRU</td>
<td>64</td>
<td>128</td>
<td>256</td>
</tr>
</tbody>
</table>

Table 4: Fitting scores of oracle traces and buggy traces generated for each model.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Trace</th>
<th>GloVe</th>
<th>Word2Vec</th>
<th>Adversarial</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Vanilla RNN</td>
<td>64</td>
<td>128</td>
<td>256</td>
</tr>
<tr>
<td></td>
<td>LSTM</td>
<td>64</td>
<td>128</td>
<td>256</td>
</tr>
<tr>
<td></td>
<td>GRU</td>
<td>64</td>
<td>128</td>
<td>256</td>
</tr>
</tbody>
</table>

one hidden state (see Section 2.1). LSTM has a complicated model structure with three different types of control gates as discussed in Section 2.1. GRU is a more advanced RNN model introduced by Cho et al. [14]. It has been shown to be one of the state-of-the-arts. It has been shown to be one of the state-of-the-arts.
do that as TRADER essentially makes use of the validation dataset. The fitting score in Table 4 denotes the coefficient of determination $R^2$, which is used to measure the fitting performance. The first column denotes the original model already achieves very high accuracy. When all defective dimension identification in Table 5. Particularly, we chose to do that as TRADER essentially makes use of the validation dataset. We also evaluate the effectiveness of defective dimension identification in Table 5. Particularly, we chose to do that as TRADER essentially makes use of the validation dataset.

### 4.3 Evaluation of Fixing Model Bugs

The results of bug fixing are presented in Table ??, the first column denotes the three word embeddings. The second column denotes the results for original models. Column RS denotes the baseline, a state-of-the-art embedding regulation technique [63]. The original and RS models are trained on both the training and the validation sets. We chose to do that as TRADER essentially makes use of the validation dataset.

<table>
<thead>
<tr>
<th>Embedding Dataset</th>
<th>Vanilla RNN</th>
<th>LSTM</th>
<th>GRU</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>64</td>
<td>128</td>
<td>256</td>
</tr>
<tr>
<td>App reviews</td>
<td>55.88</td>
<td>50.00</td>
<td>55.88</td>
</tr>
<tr>
<td>IMDB</td>
<td>58.32</td>
<td>66.63</td>
<td>59.25</td>
</tr>
<tr>
<td>GloVe</td>
<td>54.13</td>
<td>60.67</td>
<td>56.93</td>
</tr>
<tr>
<td>JIRA issues</td>
<td>54.13</td>
<td>60.67</td>
<td>56.93</td>
</tr>
<tr>
<td>Stack overflow</td>
<td>49.53</td>
<td>56.33</td>
<td>49.53</td>
</tr>
<tr>
<td>Stack overflow</td>
<td>49.53</td>
<td>56.33</td>
<td>49.53</td>
</tr>
<tr>
<td>Yelp</td>
<td>50.66</td>
<td>56.67</td>
<td>50.66</td>
</tr>
</tbody>
</table>

Linear regression is utilized to approximate the distributions of oracle traces and buggy traces to construct the oracle machine and the buggy machine. To evaluate the performance of the models, we demonstrate the results in Table 4. The first column denotes the datasets. The second column denotes the traces. The following columns compare different models (Vanilla RNN, LSTM, and GRU), word embeddings (GloVe, Word2vec, Adversarial, and model settings), which are used for training the original application models. The fitting score in Table 4 denotes the coefficient of determination $R^2$, which is used to measure the fitting performance. It ranges from 0 (worst) to 1 (best). Almost all the scores are over 0.9, which indicates that our oracle (buggy) machine effectively approximates the distribution of oracle (buggy) traces.

### 4.4 Case Study

In this section, we study individual cases to show why the buggy model mis-predicts input samples and how the fixed model performs. Figure 11 shows four text samples from the app reviews dataset and the stack overflow dataset. For each sentence, we present a pair of results, with the first predicted by the buggy model and the second predicted by the fixed model. The color from red to green and then to blue denotes the sentiment from negative to neutral and then to positive. The brightness of colors represents the degree of sentiment values. Brighter the color, larger the degree towards the corresponding sentiment. For the first case, the ground truth label is positive but the buggy model predicts neutral. It can be observed that the sentiment output stays neutral at the step with the red to green and then to blue denotes the sentiment from negative to neutral and then to positive. The brightness of colors represents the degree of sentiment values.

---

The dimensions of embeddings and retrain models based on those embeddings.

We have the following observations. For various applications using different word embeddings and model structures, TRADER can consistently improve the test accuracy, compared to the original models trained on both the training set and the validation set. The baseline RS [63] can improve a subset of models, some with substantial improvement (e.g., 8.82% for App reviews dataset using LSTM with 256 neurons and word2vec embeddings). However, it leads to degradation in a number of models as well (e.g., -5.93% for Yelp dataset using vanilla RNN with 256 neurons and Adversarial embeddings).

The average improvement is 0.6%. In comparison, TRADER achieves 3.7% improvement on average (over the original models), which is substantially larger than RS, and in fact also much larger than the improvement reported in the literature for RNN types of models, which is typically 0.05%-3.76% [17, 38, 39, 51, 64, 94] with a median of 0.7%. The improvement on the largest dataset is relatively smaller than the others, especially for the setting (GRU structure + Adversarial embedding). This is because the original model already achieves very high accuracy.
To reduce the threat, we run our experiments multiple times and compare the results of the buggy model (top) and the fixed model (bottom). The color from red to green and then to blue denotes the sentiment from negative to neutral and then to positive. The brightness of colors represents the degree of sentiment values. Brighter the color, larger the degree towards the corresponding sentiment.

word "worked". In this context, word "worked" has positive sentiment and should significantly contribute to the final prediction. The fixed model acts as expected. In the second case, the buggy model treats word "working" non-negatively. However, in this context, it comes after word "not", which should be considered jointly. The third case shows that in long sentences, the buggy model may focus locally without considering the whole context ("lower energy settings"), which produces wrong prediction. In the fourth case, the buggy model focuses too much on the previous context without considering the local information ("fine"). We think that after regulating the embeddings, the model substructures (e.g., forget gates) have more appropriate behaviors (e.g., remembering the right context and forgetting the undesirable ones) as their behaviors are not perturbed by words that have different meanings but similar embeddings.

5 THREAT TO VALIDITY
Since we use random perturbation during training, which is typical in model hardening [66], the results may have uncertainty. To reduce the threat, we run our experiments multiple times and report the average. The results are achieved on specific settings, such as batch size, optimizer, learning rate, and hyper-parameter values. To achieve fair comparison, we follow the same setting in existing works as much as possible (e.g., regarding how to partition datasets). We also release our settings in [84] for reproduction. Note that although cross-validation is often used to reduce uncertainty in machine learning results, due to the large scale of data, most existing works on RNN, especially those considering datasets like IMDB and Yelp, cannot afford cross-validation [17, 51, 58, 94]. The original models may have bugs other than problematic embeddings (e.g., data bias). The good results we achieve could be partially attributed to the fact that the hardening alleviates some of those bugs. However, the fact that we only perform guided hardening on embedding (instead of on weights like in [52]) indicates that the other bugs, if they exist, have substantial confounding with embeddings. The evaluations are conducted on sentiment analysis task. The proposed TRADER, however, is not application specific. For instance, in sequence-to-sequence tasks (e.g., neural machine translation), two RNN models are usually used: one (Encoder) for encoding input sequences to hidden states and the other one (Decoder) for decoding target sequences from hidden states together with the output of the previous step. The Decoder is similar to RNN models used in sentiment analysis, where hidden states and an input element are fed to the model to obtain an output. We can use TRADER to identify the divergence steps of Decoder and locate faulty dimensions in hidden states and input elements. The embedding regulation can be conducted on both source language and target language embeddings.

6 RELATED WORK
Our technique is inspired by software debugging (e.g., [10, 18, 22, 23, 44, 67, 71, 97, 99]). Many techniques use trace analysis and differential analysis. They locate bugs by tracing program execution and comparing buggy runs with correct runs. Similarly, we trace RNN executions and locate divergence. Unlike traditional software, RNN uses high dimension embeddings and has much more complex data dependences between the embeddings and neuron activation values, so we use embedding regulation and retraining to repair RNN models.

There are many works [2, 7, 13, 20, 28, 50, 62, 85] that employ general machine learning methods and some works [9, 29, 35, 36, 47, 49, 88] specifically use RNN models in software engineering tasks. TRADER can help software engineering researchers debug their RNN models. Researchers have also proposed different methods to debug the machine learning models [11, 12, 52]. However, these works focused on specific machine learning models or feed-forward Neural Networks and are not applicable to RNN models. In the article [70], researchers aim at debugging NLP models by generating adversarial examples as training data. In articles [37, 101], researchers propose methods to debug models by cleaning up the wrongly labeled training data. These approaches debug RNN models by providing better training data and do not analyze model internals. TRADER is orthogonal to these works. There are also works [42, 69] that explain NLP models and use model explanations to help data engineers debug models. These approaches require human efforts while TRADER is fully automated.

7 CONCLUSION
We develop a novel technique to automatically diagnose how problematic word embeddings influence model accuracy, by collecting and comparing model execution traces for correctly and incorrectly classified samples. A new embedding regulation/tuning algorithm is proposed to leverage the diagnosis results to harden the embeddings. Our experiments show that our technique can consistently and effectively improve accuracy for real world models and datasets by 5.37% on average.

ACKNOWLEDGMENTS
We thank the anonymous reviewers for their constructive comments. This research was supported, in part by DARPA FA8650-15-C-7562, NSF 1748764, 1901242 and 1910300, ONR N000141410468 and N000141712947, and Sandia National Lab under award 1701331. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.
REFERENCES


