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

Anonymous Author(s)*

## 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.

## 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 embeddgings are critical for RNN model accuracy. 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 (**TRA**ce **D**ivergence analysis and **E**mbedding **R**egulation). 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

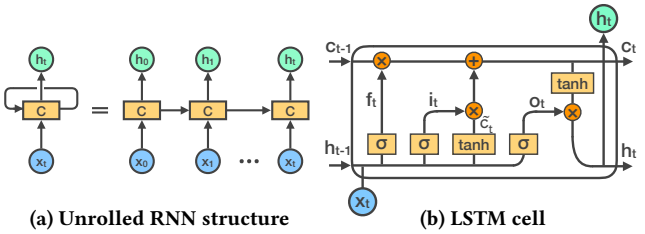
Recurrent neural networks (RNNs) are a family of neural networks designed to tackle problems with sequential inputs. Different from traditional neural networks that require fixed-length inputs, RNNs can handle sequential inputs of arbitrary length. That is, input is received continuously through many 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.

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

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

$$prediction = W \cdot h_n + b, \tag{2}$$

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 arg max() on the output prediction vector.




(a) Unrolled RNN structure (b) LSTM cell

**Figure 1: Architecture of recurrent neural networks. (a) An unrolled representation of RNN architecture. (b) The internal structure of Long Short Term Memory networks.**

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.

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \tag{3}$$
$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \tag{4}$$
$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \tag{5}$$
$$\tilde{c}_t = \tanh(W_c \cdot [h_{t-1}, x_t] + b_c) \tag{6}$$
$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \tag{7}$$
$$h_t = o_t \odot \tanh(c_t), \tag{8}$$

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 $\tilde{c}_t$; 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 Schnabel 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

| error | | great | | reinstall | |
|---|---|---|---|---|---|
| Original | Regulated | Original | Regulated | Original | Regulated |
| errors | errors | good | good | reinstalling | reinstalling |
| correct | mistake | well | well | uninstall | install |
| mistake | difference | little | much | refresh | uninstall |
| difference | correct | much | experience | install | refresh |

**Figure 2: Nearest neighbors of words according to their embeddings. The top row denotes the target words, and the following rows are their nearest words measured by cosine similarity. Column Original denotes words using GloVe word embeddings and column Regulated denotes word embeddings after regulation.**

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 $[1, 1, 2, 3]$. By

```
1  def count(data):
2      output = 0
3      for i in range(len(data)):
4          embed = embedding(data[i]) # map to integer
5          if embed >= 1 and embed < 2:
6              output += 1
7      return output
8
9  def main():
10     data = ["1.1", "0.3", "1.4", "1.8"]
11     print count(data)
```

```
1  def RNN(text):
2      outputs = []
3      hidden = zeros(size)
4      for i in range(len(text)):
5          # map to word embedding
6          embed  = embedding(text[i])
7          input  = concatenate(hidden, embed)
8          hidden = matmul(input, W_h) + b_h
9          output = matmul(hidden, W_out) + b_out
10         outputs.append(output)
11     return outputs[len(text) - 1]
```

**Figure 3: Comparison between code snippets of a simple counting program and an RNN model. The left code snippet shows a simple program for counting the number of data points within range [1, 2). The right code snippet gives a simplified RNN model for predicting the label of an input sentence.**

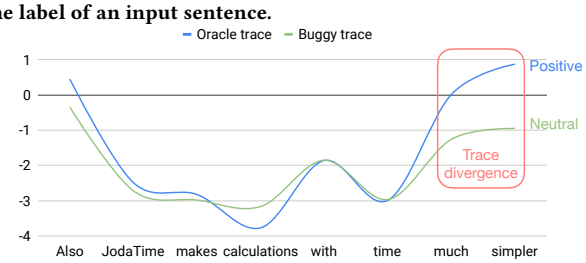



**Figure 4: The trace of a sample text from Stack overflow dataset [49] predicted by a real-world LSTM model [19]. The blue line denotes the oracle trace with correct prediction of positive sentiment. The green line denotes the buggy trace produced by the LSTM model with incorrect prediction of neutral sentiment.**

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`[1] `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 ② is in the middle. The blue arrow to ⓑ denotes the model

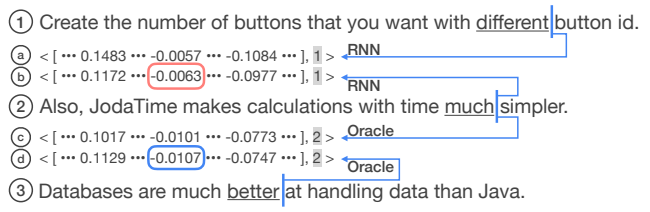



**Figure 5: Analysis of an example text (shown in ②). Vectors ⓐ-ⓓ denote the model states (with the prediction result at the end, 1 for neutral and 2 for positive) right after the underlined words in various sentences: ⓑ and ⓒ are states for the subject sentence by the buggy RNN model and the oracle, respectively; ⓐ is a state by the buggy model closest to ⓑ; ⓓ is a state by the oracle closest to ⓒ. Sentences ② is not in either the training set or the validation set; ① is from the training set and ③ from the validation set.**

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 positive). As sentence ② is not in the training data set, the model has to generalize based on what it has seen during training. Further inspection shows that sentence ① is in the training set, and its state ⓐ after word "`different`" is very close to ⓑ and the corresponding output is neutral, which explains why the model predicts neutral sentiment. Below sentence ②, we show the state after word "much" by the oracle model, which approximates model output when given the concatenation of the input, cell state, and hidden state vectors $[x_t, c_{t-1}, h_{t-1}]$ *for all the correctly classified sentences in the validation set.* Its construction will be discussed in Section 3.1. When we provide the concatenated vector of sentence ② to the oracle model, it predicts positive as shown in ⓒ. This is because ⓒ is close to state ⓓ after word "`better`" in sentence ③ from the validation set, which has the positive sentiment.

The state vector values heavily depend on the word embeddings. If we consider ⓓ denotes the state derived from the ideal embeddings, the root cause lies in that the current problematic embeddings lead to the state divergence of ⓑ and ⓓ, which are highlighted by the red and blue rectangles. *The essence of our technique is hence to harden the word embeddings to minimize such differences.*

Note that although we use a single input sentence to intuitively explain the idea, our technique essentially has to minimize such differences for all misclassified sentences in the training set to achieve the effect of improving overall accuracy.

[1] A data and time library for Java. https://www.joda.org/joda-time/

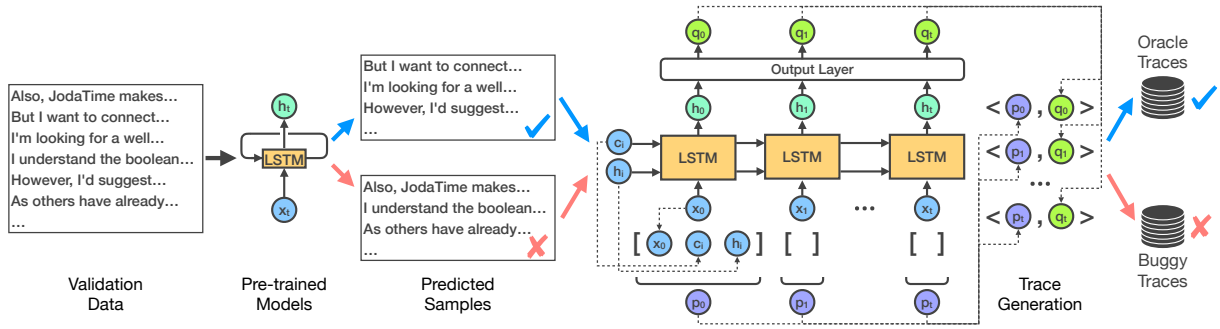



**Figure 6: Trace generation for samples in the validation set. Words in input sentences are mapped to their corresponding word embeddings $x_t$. Internal states $c_i$ and $h_i$ are the initial internal state vectors for LSTM models. The output layer is used to predict the final output label as discussed in Equation 2. Here, it has been extended to the whole sequence for acquiring the internal output at each step.**


**Figure 7: Overview of TRADER.**

## 3 DESIGN

Given an RNN model to debug, we leverage the validation dataset to inspect the problematic behaviors. Figure 7 illustrates the overall design of our approach. Data in the validation set can be first processed by the model to identify the correctly classified and misclassified samples. We consider that traces of the correctly classified samples to-some-extent denote the desired behaviors of the model, while traces of the misclassified samples represent undesired behaviors. A trace divergence analysis is then performed on the traces from these two sets of samples. That is, we utilize the traces to construct two models, called the *oracle machine* and the *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 *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 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 *training set* and then tested on a *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 *validation set*. Following a similar philosophy, our technique leverages the validation set for model debugging.

As shown in Figure 6, 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 *oracle traces* (generated from the correctly classified samples) and the *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:

$$q_t = W_r \cdot p_t + b_r, \tag{9}$$

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

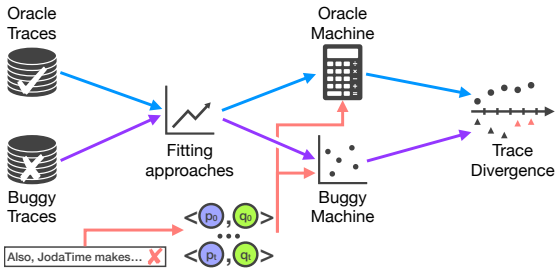



**Figure 8: Construction of *oracle machine* and *buggy machine*, and trace divergence analysis. Trace divergence analysis is conducted on a sample text by feeding its traces to the two machines and comparing the output values.**

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.

**Algorithm 1** Aggregated Divergence Analysis

```
1:  function DIVERGENCE(dataset S)
2:      d ← zero vector with size of step input
3:      for sample in S do
4:          T ← generate_trace(sample)
5:          for p in T do
6:              q_o ← arg max(OM.predict(p))
7:              q_b ← arg max(BM.predict(p))
8:              if not q_o == q_b then
9:                  d ← add_vector(d, p)
10:     w_o, w_b ← get_weight(OM, BM)
11:     w_d ← sub_vector(w_o, w_b)
12:     d = abs(mul_vector(d, w_d))
13:     return d
```

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 9 shows an example of vector $d$ for an LSTM

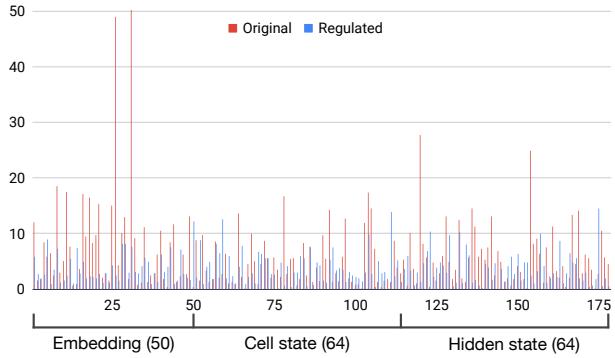



**Figure 9: An example of vector $d$ for an LSTM model.**

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 values of state vectors are largely determined by word embeddings. That is, words close to each other in the embedding space tend to lead to state vector values close to each other. Our fixing strategy is hence to change word embeddings so that the model becomes less sensitive along the faulty dimensions, by enlarging the embedding distances of the words that could lead to substantial output variations. This is done by applying perturbations to the faulty dimensions and then searching for minimal output variations by tuning word embeddings. Since RNN type of models have a loopy structure, the perturbations are applied to each iteration.

Figure 10 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 output differences to the input embeddings. The divergence loss is formalized as follows.

$$\mathcal{L}_{div} = \mathcal{L}_{ce} + \mathcal{L}_{l2}, \tag{10}$$

where $\mathcal{L}_{ce}$ is cross entropy loss [24] and $\mathcal{L}_{l2}$ 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

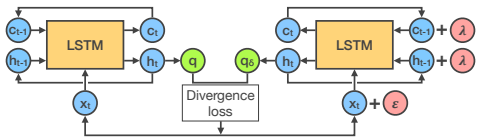



**Figure 10: Structure of embedding regulation. The left LSTM has the original model structure. The right LSTM is extended with perturbations for input embeddings $x_t$ as well as internal states $c_{t-1}$ and $h_{t-1}$; $q$ and $q_\delta$ are the final outputs from the original model and the perturbed model, respectively.**

defines its essence.

$$\mathcal{L}_{ce} = -\frac{1}{N}\sum_{n=1}^{N}\sum_{c=1}^{C} y_n^{o,c}\ln(p_n^{o,c}), \tag{11}$$

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 $y_n^{o,c}$ is 1; otherwise 0. $p_n^{o,c}$ 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.

$$\mathcal{L}_{l2} = \frac{1}{2}\sum_{n=1}^{N}\sum_{c=1}^{C}(p_n^c - \hat{p}_n^c)^2, \tag{12}$$

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

**Algorithm 2** Embedding Regulation

```
1:  function REGULATION(model M, dataset S, embedding E, divergence D)
2:      u ← zero vector with dimension size of embedding
3:      v ← zero vector with dimension size of hidden neurons
4:      for d in D do
5:          if d > θ then
6:              idx ← D.index(d)
7:              if idx belongs to E then
8:                  u[idx] ← random_normal(0, ϵ)
9:              else
10:                 v[idx] ← random_normal(0, λ)
11:     E ← get_variable(E)
12:     E ← add_vector(E, u)
13:     c_t, h_t ← initialized with dimension size of hidden neurons
14:     for i in range(max_step) do
15:         c_t ← add_vector(c_t, v)
16:         h_t ← add_vector(h_t, v)
17:         c_t, h_t ← LSTM(c_t, h_t)
18:     S ← embedding(S, E)
19:     M.freeze()
20:     LSTM.train(M, S, L_div)
21:     return E
```

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 $N(0, \epsilon)$ (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 deviation of original word embeddings as it should not overshadow the original values. For dimensions denoting the hidden 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 9 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

**Table 1: Statistics of datasets.**

| Dataset | # samples | # negative | # neutral | # positive | Max length |
|---|---|---|---|---|---|
| App reviews | 341 | 130 | 25 | 186 | 231 |
| IMDB | 50,000 | 25,000 | 0 | 25,000 | 2,506 |
| JIRA issues | 926 | 636 | 0 | 290 | 49 |
| Stack overflow | 1,500 | 131 | 1,191 | 178 | 52 |
| Yelp | 5,946,620 | 1,544,553 | 0 | 4,402,067 | 3,929 |

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 co-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 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. A batch size of 24 samples is used for each training iteration, except for Yelp (which has the size of 512, its default setting). We use the Adam optimizer [40] with the learning rate of 0.001. Note that we train these models by ourselves, which is consistent with existing works on RNN [19, 77]. The accuracy of the trained models align well with the literature [49].

**Table 2: Word embeddings.**

| Embedding | # dimensions | Mean | Standard deviation |
|---|---|---|---|
| GloVe | 50 | 0.020941 | 0.644104 |
| Word2vec | 300 | -0.004118 | 0.107278 |
| Adversarial | 256 | -0.053817 | 0.767211 |

**Hyper-parameters.** Three hyper-parameters (i.e., $\theta$, $\epsilon$, $\lambda$) are used for embedding regulation. Parameter $\theta$ is used for selecting faulty dimensions. The value of $\theta$ can vary from the minimum value to the maximum of the divergence vector. Parameter $\epsilon$ and $\lambda$ are used to perturb embedding vectors and internal states, with $\epsilon$ ranging from (0, 1) and $\lambda = 10^{-4}$. The values of $\theta$ and $\epsilon$ 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., $\mathcal{L}_{new} = \mathcal{L}_{old} + ||W||^2$ 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 likes ours (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 mostly around a few MBs. Thus, it is negligible compared to the millions of weights and hours of training. For the large datasets IMDB and Yelp, the analysis time is around a few minutes 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.

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 approach, we demonstrate the results in Table 4. The first column denotes the

**Table 3: Trace divergence analysis overhead.**

| Overhead | Dataset | Vanilla RNN | | | | | | | | | LSTM | | | | | | | | | GRU | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | GloVe | | | Word2vec | | | Adversarial | | | GloVe | | | Word2vec | | | Adversarial | | | GloVe | | | Word2vec | | | Adversarial | | |
| | | 64 | 128 | 256 | 64 | 128 | 256 | 64 | 128 | 256 | 64 | 128 | 256 | 64 | 128 | 256 | 64 | 128 | 256 | 64 | 128 | 256 | 64 | 128 | 256 | 64 | 128 | 256 |
| Time (s) | App reviews | 1.737 | 1.779 | 1.823 | 1.927 | 1.833 | 1.958 | 1.679 | 1.798 | 2.085 | 2.950 | 2.841 | 3.029 | 3.151 | 3.108 | 3.216 | 2.903 | 2.720 | 2.526 | 3.454 | 4.063 | 4.223 | 3.856 | 5.137 | 3.511 | 4.823 | 6.064 | 3.706 |
| | IMDB (×10) | 0.742 | 0.967 | 0.912 | 0.911 | 1.279 | 1.174 | 0.939 | 1.096 | 1.097 | 1.523 | 1.276 | 1.550 | 1.994 | 1.822 | 2.024 | 1.672 | 1.665 | 1.946 | 1.483 | 1.346 | 1.633 | 1.678 | 1.690 | 2.053 | 1.694 | 1.651 | 1.906 |
| | JIRA issues | 1.070 | 1.034 | 1.057 | 1.272 | 1.164 | 1.409 | 1.142 | 1.275 | 1.161 | 1.499 | 1.313 | 1.339 | 1.430 | 1.276 | 1.383 | 1.397 | 1.435 | 1.463 | 1.415 | 1.374 | 1.527 | 1.481 | 1.716 | 1.331 | 1.505 | 1.483 | 1.525 |
| | Stack overflow | 1.277 | 1.242 | 1.220 | 1.223 | 1.109 | 1.266 | 1.107 | 1.243 | 1.442 | 1.366 | 1.553 | 1.616 | 1.486 | 1.634 | 1.609 | 1.500 | 1.503 | 1.508 | 1.638 | 1.611 | 1.502 | 1.706 | 1.435 | 1.594 | 1.664 | 1.555 | 1.767 |
| | Yelp (×$10^3$) | 0.206 | 0.305 | 0.466 | 0.536 | 0.638 | 0.842 | 0.491 | 0.613 | 0.775 | 0.278 | 0.404 | 0.707 | 0.553 | 0.717 | 1.117 | 0.494 | 0.625 | 0.970 | 0.502 | 0.605 | 0.820 | 1.725 | 2.149 | 1.836 | 1.182 | 1.097 | 1.315 |
| Space (M) | App reviews | 0.399 | 0.616 | 1.052 | 1.249 | 1.467 | 1.903 | 1.100 | 1.317 | 1.753 | 0.616 | 1.052 | 1.923 | 1.467 | 1.903 | 2.774 | 1.317 | 1.753 | 2.624 | 0.399 | 0.616 | 1.052 | 1.249 | 1.467 | 1.903 | 1.100 | 1.317 | 1.753 |
| | IMDB (×$10^2$) | 1.943 | 3.015 | 5.160 | 6.131 | 7.203 | 9.348 | 5.394 | 6.466 | 8.610 | 1.943 | 3.015 | 5.160 | 6.131 | 7.203 | 9.348 | 5.394 | 6.466 | 8.610 | 1.943 | 3.015 | 5.160 | 6.131 | 7.203 | 9.348 | 5.394 | 6.466 | 8.610 |
| | JIRA issues | 0.327 | 0.507 | 0.866 | 1.029 | 1.209 | 1.569 | 0.906 | 1.086 | 1.446 | 0.507 | 0.866 | 1.586 | 1.209 | 1.569 | 2.289 | 1.086 | 1.446 | 2.165 | 0.327 | 0.507 | 0.866 | 1.029 | 1.209 | 1.569 | 0.906 | 1.086 | 1.446 |
| | Stack overflow | 0.742 | 1.147 | 1.958 | 2.326 | 2.731 | 3.542 | 2.047 | 2.453 | 3.264 | 1.147 | 1.958 | 3.580 | 2.731 | 3.542 | 5.165 | 2.453 | 3.264 | 4.886 | 0.742 | 1.147 | 1.958 | 2.326 | 2.731 | 3.542 | 2.047 | 2.453 | 3.264 |
| | Yelp (×$10^4$) | 0.483 | 0.766 | 1.290 | 1.560 | 1.822 | 2.382 | 1.407 | 1.660 | 2.191 | 6.933 | 1.124 | 2.000 | 1.515 | 1.931 | 2.787 | 1.339 | 1.756 | 2.592 | 4.561 | 6.879 | 1.154 | 1.319 | 1.521 | 1.955 | 1.118 | 1.319 | 1.730 |

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

| Dataset | Trace | Vanilla RNN | | | | | | | | | LSTM | | | | | | | | | GRU | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | GloVe | | | Word2vec | | | Adversarial | | | GloVe | | | Word2vec | | | Adversarial | | | GloVe | | | Word2vec | | | Adversarial | | |
| | | 64 | 128 | 256 | 64 | 128 | 256 | 64 | 128 | 256 | 64 | 128 | 256 | 64 | 128 | 256 | 64 | 128 | 256 | 64 | 128 | 256 | 64 | 128 | 256 | 64 | 128 | 256 |
| App Reviews | Oracle | 0.974 | 0.971 | 0.679 | 0.971 | 0.955 | 0.967 | 0.980 | 0.983 | 0.989 | 0.992 | 0.981 | 0.999 | 0.997 | 0.993 | 0.998 | 0.986 | 0.993 | 1.000 | 0.978 | 0.984 | 0.989 | 0.989 | 0.990 | 0.994 | 0.983 | 0.975 | 0.987 |
| | Buggy | 0.995 | 0.970 | 0.466 | 0.991 | 0.988 | 0.993 | 0.978 | 1.000 | 1.000 | 0.994 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 0.996 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 0.998 | 1.000 | 1.000 |
| IMDB | Oracle | 0.933 | 0.965 | 0.966 | 0.986 | 0.985 | 0.985 | 0.884 | 0.858 | 0.830 | 0.975 | 0.984 | 0.964 | 0.990 | 0.989 | 0.985 | 0.912 | 0.917 | 0.963 | 0.992 | 0.993 | 0.992 | 0.996 | 0.995 | 0.996 | 0.992 | 0.991 | 0.985 |
| | Buggy | 0.932 | 0.968 | 0.983 | 0.985 | 0.986 | 0.986 | 0.868 | 0.851 | 0.816 | 0.949 | 0.967 | 0.932 | 0.985 | 0.980 | 0.977 | 0.852 | 0.853 | 0.932 | 0.978 | 0.982 | 0.980 | 0.991 | 0.988 | 0.990 | 0.981 | 0.979 | 0.965 |
| JIRA issues | Oracle | 0.943 | 0.949 | 0.828 | 0.977 | 0.987 | 0.993 | 0.973 | 0.974 | 0.989 | 0.986 | 0.984 | 0.994 | 0.999 | 0.999 | 0.999 | 0.987 | 0.991 | 0.999 | 0.960 | 0.941 | 0.950 | 0.993 | 0.996 | 0.998 | 0.974 | 0.986 | 0.995 |
| | Buggy | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 | 1.000 |
| Stack overflow | Oracle | 0.862 | 0.858 | 0.899 | 0.868 | 0.865 | 0.973 | 0.904 | 0.817 | 0.916 | 0.994 | 0.902 | 0.891 | 0.994 | 0.954 | 0.975 | 0.950 | 0.917 | 0.933 | 0.948 | 0.948 | 0.944 | 0.967 | 0.987 | 0.881 | 0.880 | 0.908 | 0.953 |
| | Buggy | 0.875 | 0.882 | 0.970 | 0.944 | 0.944 | 0.988 | 0.944 | 0.934 | 1.000 | 0.997 | 0.950 | 0.893 | 0.999 | 0.998 | 0.997 | 0.993 | 0.998 | 0.997 | 0.967 | 0.969 | 0.977 | 0.991 | 0.993 | 0.987 | 0.966 | 0.990 | 0.999 |
| Yelp | Oracle | 0.946 | 0.958 | 0.970 | 0.986 | 0.990 | 0.919 | 0.925 | 0.915 | 0.915 | 0.987 | 0.988 | 0.988 | 0.991 | 0.992 | 0.991 | 0.979 | 0.969 | 0.971 | 0.990 | 0.990 | 0.989 | 0.993 | 0.993 | 0.994 | 0.990 | 0.990 | 0.990 |
| | Buggy | 0.947 | 0.962 | 0.978 | 0.987 | 0.989 | 0.950 | 0.930 | 0.913 | 0.918 | 0.967 | 0.970 | 0.969 | 0.978 | 0.979 | 0.981 | 0.948 | 0.925 | 0.928 | 0.978 | 0.976 | 0.975 | 0.983 | 0.981 | 0.984 | 0.974 | 0.971 | 0.971 |

**Table 5: Test accuracy on five applications, three word embeddings, three RNN structures with three different settings.**

| Embedding | Dataset | Vanilla RNN | | | | | | | | | LSTM | | | | | | | | | GRU | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Original | | | RS | | | TRADER | | | Original | | | RS | | | TRADER | | | Original | | | RS | | | TRADER | | |
| | | 64 | 128 | 256 | 64 | 128 | 256 | 64 | 128 | 256 | 64 | 128 | 256 | 64 | 128 | 256 | 64 | 128 | 256 | 64 | 128 | 256 | 64 | 128 | 256 | 64 | 128 | 256 |
| GloVe | App reviews | 67.65 | 67.65 | 58.82 | 67.65 | 67.65 | 58.82 | **79.41** | **73.53** | **73.53** | 73.53 | 70.59 | 70.59 | 67.65 | 70.59 | 70.59 | **79.41** | **82.35** | **82.35** | 76.47 | 76.47 | 76.47 | 76.47 | 76.47 | 76.47 | **79.41** | **79.41** | **82.35** |
| | IMDB | 64.81 | 63.42 | 60.32 | 61.91 | 66.52 | 65.70 | **71.47** | **69.40** | **72.17** | 84.67 | 84.51 | 84.28 | 84.48 | 84.82 | 84.74 | **87.13** | **86.80** | **85.72** | 85.63 | 85.82 | 85.50 | 85.11 | 85.05 | 84.82 | **87.44** | **87.41** | **86.98** |
| | JIRA issues | 88.04 | 91.30 | 84.78 | 89.13 | 89.13 | 81.52 | **96.74** | **96.74** | **97.83** | 94.57 | 94.57 | 93.48 | 93.48 | 92.39 | 91.30 | **97.83** | **97.83** | **97.83** | 91.30 | 92.39 | 94.57 | 94.57 | 93.48 | 93.48 | **97.83** | **96.74** | **96.74** |
| | Stack overflow | 86.00 | 86.00 | 86.00 | 86.00 | 86.00 | 86.00 | **88.00** | **88.00** | **88.00** | 86.00 | 86.00 | 86.00 | 86.67 | 86.67 | 86.00 | **89.33** | **89.33** | **89.33** | 86.67 | 86.67 | 86.00 | 86.67 | 86.67 | 86.67 | **88.67** | **89.33** | **89.33** |
| | Yelp | 79.56 | 78.51 | 77.70 | 79.33 | 80.67 | 77.67 | **83.11** | **83.17** | **89.90** | 92.02 | 92.45 | 92.09 | 92.79 | 92.94 | 92.93 | **94.07** | **94.67** | **94.81** | 91.45 | 91.34 | 90.89 | 92.14 | 92.06 | 91.90 | **94.19** | **94.36** | **94.50** |
| Word2vec | App reviews | 61.76 | 55.88 | 67.65 | 67.65 | 58.82 | 58.82 | **79.41** | **85.29** | **79.41** | 79.41 | 73.53 | 67.65 | 82.35 | 79.41 | 76.47 | **88.24** | **88.24** | **88.24** | 79.41 | 76.47 | 76.47 | 82.35 | 79.41 | 79.41 | **82.35** | **88.24** | **85.29** |
| | IMDB | 68.86 | 63.62 | 70.62 | 72.09 | 73.34 | 71.01 | **77.27** | **78.73** | **77.32** | 87.42 | 87.88 | 87.33 | 87.18 | 87.28 | 87.18 | **88.26** | **88.23** | **87.75** | 88.02 | 88.45 | 88.38 | 87.75 | 87.56 | 87.41 | **88.67** | **88.60** | **88.34** |
| | JIRA issues | 92.39 | 91.30 | 88.04 | 93.48 | 94.57 | 88.04 | **98.91** | **97.83** | **97.83** | 93.48 | 94.57 | 91.30 | 94.57 | 96.74 | 95.65 | **96.74** | **96.74** | **97.83** | 93.48 | 96.74 | 96.74 | 94.57 | 96.74 | 96.74 | **97.83** | **97.83** | **97.83** |
| | Stack overflow | 86.67 | 86.67 | 86.00 | 86.67 | 86.67 | 86.00 | **88.00** | **88.00** | **88.00** | 86.67 | 86.00 | 86.00 | 86.67 | 86.00 | 86.67 | **89.33** | **89.33** | **89.33** | 86.67 | 86.67 | 86.67 | 87.33 | 86.00 | 86.00 | **88.00** | **88.67** | **88.00** |
| | Yelp | 82.91 | 82.10 | 88.86 | 84.02 | 89.76 | 86.64 | **87.00** | **90.85** | **89.93** | 92.50 | 93.10 | 93.32 | 93.40 | 93.57 | 93.49 | **94.62** | **95.02** | **95.16** | 92.82 | 92.70 | 92.80 | 92.90 | 92.84 | 92.76 | **94.62** | **94.73** | **94.87** |
| Adversarial | App reviews | 55.88 | 58.82 | 52.94 | 61.76 | 61.76 | 58.82 | **76.47** | **79.41** | **82.35** | 76.47 | 61.76 | 64.71 | 67.65 | 61.76 | 67.65 | **85.29** | **82.35** | **82.35** | 73.53 | 70.59 | 73.53 | 76.47 | 76.47 | 76.47 | **82.35** | **85.29** | **85.29** |
| | IMDB | 76.22 | 74.20 | 67.75 | 76.62 | 73.09 | 70.35 | **78.78** | **79.33** | **79.68** | 87.94 | 88.39 | 88.51 | 87.13 | 88.44 | 89.07 | **88.62** | **89.07** | **89.20** | 88.96 | 88.52 | 88.65 | 88.94 | 89.12 | 88.79 | **89.25** | **89.42** | **89.70** |
| | JIRA issues | 89.13 | 90.22 | 89.13 | 91.30 | 89.13 | 91.30 | **96.74** | **96.74** | **97.83** | 93.48 | 93.48 | 94.57 | 94.57 | 93.48 | 96.74 | **96.74** | **96.74** | **97.83** | 95.65 | 96.74 | 96.74 | 96.74 | 96.74 | 95.65 | **98.91** | **98.91** | **98.91** |
| | Stack overflow | 86.00 | 84.67 | 86.67 | 82.67 | 85.33 | 84.00 | **88.00** | **88.00** | **88.00** | 86.67 | 85.33 | 86.67 | 86.67 | 86.00 | 86.00 | **88.67** | **88.67** | **90.00** | 86.67 | 86.67 | 86.00 | 86.67 | 86.67 | 86.67 | **88.00** | **89.33** | **89.33** |
| | Yelp | 85.19 | 86.30 | 85.27 | 85.33 | 86.29 | 79.35 | **87.92** | **88.54** | **91.67** | 93.11 | 93.49 | 93.14 | 94.04 | 94.26 | 94.38 | **94.52** | **94.97** | **95.10** | 93.69 | 93.84 | 93.80 | 94.07 | 94.15 | 94.12 | **94.60** | **94.77** | **94.96** |

datasets. The second column denotes the trace types. The following columns denote different models (Vanilla RNN, LSTM and GRU), word embeddings (GloVe, Word2vec, Adversarial), and model settings (64, 128, 256) that 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.

## 4.3 Evaluation of Fixing Model Bugs

The results of bug fixing are presented in Table 5. The first column denotes the three word embeddings. The second column denotes the five applications. The following columns denote the test accuracy for different models and settings. Column "original" denotes the results for original models. Column RS detotes 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 set in its debugging procedure.

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 5.37% 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

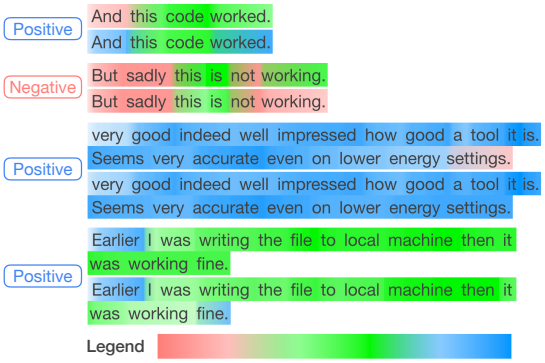



**Figure 11: Prediction by the buggy model and the fixed model. Each pair shows the prediction results by the buggy model (top) and the fixed model (bottom). The color from red to green and then to blue denotes the sentiment from negative to neutral and then to positive. The brightness of colors represents the degree of sentiment values. Brighter the color, larger the degree towards the corresponding sentiment.**

a median of 0.7%. The improvement on the largest dataset Yelp 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.

### 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 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 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.

## 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.

# REFERENCES


[1] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 40.

[2] Dalal Alrajeh and Alessandra Russo. 2018. Logic-Based Learning: Theory and Application. In *Machine Learning for Dynamic Software Analysis: Potentials and Limits*.

[3] Mathieu Aubry and Bryan C Russell. 2015. Understanding Deep Features with Computer-Generated Imagery. In *IEEE International Conference on Computer Vision (ICCV)*. 2875–2883.

[4] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural machine translation by jointly learning to align and translate. In *International Conference on Learning Representations (ICLR)*.

[5] Marco Baroni, Georgiana Dinu, and Germán Kruszewski. 2014. Don't count, predict! A systematic comparison of context-counting vs. context-predicting semantic vectors. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Vol. 1. 238–247.

[6] David Bau, Bolei Zhou, Aditya Khosla, Aude Oliva, and Antonio Torralba. 2017. Network Dissection: Quantifying Interpretability of Deep Visual Representations. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[7] Raja Ben Abdessalem, Shiva Nejati, Lionel C Briand, and Thomas Stifter. 2016. Testing advanced driver assistance systems using multi-objective search and neural networks. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 63–74.

[8] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. 1994. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks* 5, 2 (1994), 157–166.

[9] Sahil Bhatia, Pushmeet Kohli, and Rishabh Singh. 2018. Neuro-symbolic program corrector for introductory programming assignments. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 60–70.

[10] Ivan Bocić and Tevfik Bultan. 2016. Finding access control bugs in web applications with CanCheck. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 155–166.

[11] Gabriel Cadamuro, Ran Gilad-Bachrach, and Xiaojin Zhu. 2016. Debugging machine learning models. In *ICML Workshop on Reliable Machine Learning in the Wild*.

[12] Aleksandar Chakarov, Aditya Nori, Sriram Rajamani, Shayak Sen, and Deepak Vijaykeerthy. 2016. Debugging machine learning tasks. *arXiv preprint arXiv:1603.07292* (2016).

[13] Chunyang Chen, Zhenchang Xing, and Yang Liu. 2017. By the community & for the community: a deep learning approach to assist collaborative editing in q&a sites. *Proceedings of the ACM on Human-Computer Interaction* (2017).

[14] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In *Empirical Methods in Natural Language Processing (EMNLP)*.

[15] Google Code. 2013. *Word2vec*. https://code.google.com/archive/p/word2vec/

[16] Ekin D Cubuk, Barret Zoph, Dandelion Mane, Vijay Vasudevan, and Quoc V Le. 2018. AutoAugment: Learning Augmentation Policies from Data. *arXiv preprint arXiv:1805.09501* (2018).

[17] Andrew M Dai and Quoc V Le. 2015. Semi-supervised sequence learning. In *Advances in neural information processing systems*. 3079–3087.

[18] Loris D'Antoni, Rishabh Singh, and Michael Vaughn. 2017. NoFAQ: synthesizing command repairs from examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 582–592.

[19] Adit Deshpande. 2017. *Sentiment Analysis with LSTMs*. https://github.com/adeshpande3/LSTM-Sentiment-Analysis

[20] Dario Di Nucci, Fabio Palomba, Damian A Tamburri, Alexander Serebrenik, and Andrea De Lucia. 2018. Detecting code smells using machine learning techniques: are we there yet?. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 612–621.

[21] Xiaoning Du, Xiaofei Xie, Yi Li, Lei Ma, Yang Liu, and Jianjun Zhao. 2019. DeepStellar: model-based quantitative analysis of stateful deep learning systems. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 477–487.

[22] Qiang Fu, Jian-Guang Lou, Qing-Wei Lin, Rui Ding, Dongmei Zhang, Zihao Ye, and Tao Xie. 2012. Performance issue diagnosis for online service systems. In *2012 IEEE 31st Symposium on Reliable Distributed Systems*. IEEE, 273–278.

[23] Zheng Gao, Christian Bird, and Earl T Barr. 2017. To type or not to type: quantifying detectable bugs in JavaScript. In *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 758–769.

[24] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

[25] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative Adversarial Nets. In *Advances in Neural Information Processing Systems (NIPS)*. 2672–2680.

[26] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. 2014. Explaining and Harnessing Adversarial Examples. *arXiv preprint arXiv:1412.6572* (2014).

[27] Michael Goul, Olivera Marjanovic, Susan Baxley, and Karen Vizecky. 2012. Managing the enterprise business intelligence app store: Sentiment analysis supported requirements engineering. In *2012 45th Hawaii International Conference on System Sciences*. IEEE, 4168–4177.

[28] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *2018 IEEE/ACM 40th International Conference on Software Engineering*.

[29] Jordan Henkel, Shuvendu K Lahiri, Ben Liblit, and Thomas Reps. 2018. Code vectors: understanding programs through embedded abstracted symbolic traces. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

[30] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. 2012. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal processing* (2012).

[31] Sepp Hochreiter. 1991. Untersuchungen zu dynamischen neuronalen Netzen. *Diploma, Technische Universität München* 91, 1 (1991).

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

[33] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2019. Deep code comment generation with hybrid lexical and syntactical information. *Empirical Software Engineering* (2019), 1–39.

[34] Nasif Imtiaz, Justin Middleton, Peter Girouard, and Emerson Murphy-Hill. 2018. Sentiment and politeness analysis tools on developer discussions are unreliable, but so are people. In *2018 IEEE/ACM 3rd International Workshop on Emotion Awareness in Software Engineering (SEmotion)*. IEEE, 55–61.

[35] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*.

[36] Siyuan Jiang, Ameer Armaly, and Collin McMillan. 2017. Automatically generating commit messages from diffs using neural machine translation. In *Proceedings of IEEE/ACM International Conference on Automated Software Engineering*.

[37] Yuan Jiang and Zhi-Hua Zhou. 2004. Editing training data for kNN classifiers with neural network ensemble. In *International symposium on neural networks*.

[38] Rie Johnson and Tong Zhang. 2015. Semi-supervised convolutional neural networks for text categorization via region embedding. In *Advances in neural information processing systems*. 919–927.

[39] Rie Johnson and Tong Zhang. 2016. Supervised and Semi-Supervised Text Categorization using LSTM for Region Embeddings. In *International Conference on Machine Learning (ICML)*. 526–534.

[40] Diederik P Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. In *International Conference on Learning Representations (ICLR)*.

[41] Diederik P Kingma and Max Welling. 2013. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114* (2013).

[42] Pang Wei Koh and Percy Liang. 2017. Understanding black-box predictions via influence functions. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org, 1885–1894.

[43] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems (NIPS)*. 1097–1105.

[44] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering* 38, 1 (2012), 54–72.

[45] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-Based Learning Applied to Document Recognition. *Proc. IEEE* (1998).

[46] Christopher J Leggetter and Philip C Woodland. 1995. Maximum likelihood linear regression for speaker adaptation of continuous density hidden Markov models. *Computer speech & language* 9, 2 (1995), 171–185.

[47] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. VulDeePecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681* (2018).

[48] Joseph Lilleberg, Yun Zhu, and Yanqing Zhang. 2015. Support vector machines and word2vec for text classification with semantic features. In *2015 IEEE 14th International Conference on Cognitive Informatics & Cognitive Computing*.

[49] Bin Lin, Fiorella Zampetti, Gabriele Bavota, Massimiliano Di Penta, Michele Lanza, and Rocco Oliveto. 2018. Sentiment Analysis for Software Engineering: How Far Can We Go?. In *Proceedings of 40th International Conference on Software Engineering (ICSE)*.

[50] Zhongxin Liu, Xin Xia, Ahmed E Hassan, David Lo, Zhenchang Xing, and Xinyu Wang. 2018. Neural-machine-translation-based commit message generation: how far are we?. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 373–384.

[51] Di Lu, Leonardo Neves, Vitor Carvalho, Ning Zhang, and Heng Ji. 2018. Visual Attention Model for Name Tagging in Multimodal Social Media. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics*.

[52] Shiqing Ma, Yingqi Liu, Wen-Chuan Lee, Xiangyu Zhang, and Ananth Grama. 2018. MODE: automated neural network model debugging via state differential analysis and input selection. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 175–186.

[53] Andrew L Maas, Raymond E Daly, Peter T Pham, Dan Huang, Andrew Y Ng, and Christopher Potts. 2011. Learning word vectors for sentiment analysis. In *Proceedings of annual meeting of the association for computational linguistics*.

[54] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

[55] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*.

[56] Ghassan Misherghi and Zhendong Su. 2006. HDD: hierarchical delta debugging. In *Proceedings of the 28th international conference on Software engineering*.

[57] Takeru Miyato. 2018. *Adversarial Text Classification*. https://github.com/tensorflow/models/tree/master/research/adversarial_text

[58] Takeru Miyato, Andrew M Dai, and Ian Goodfellow. 2017. Adversarial training methods for semi-supervised text classification. In *International Conference on Learning Representations (ICLR)*.

[59] John Neter, William Wasserman, and Michael H Kutner. 1989. Applied linear regression models. (1989).

[60] Yoann Padioleau, Lin Tan, and Yuanyuan Zhou. 2009. Listening to programmers Taxonomies and characteristics of comments in operating system code. In *Proceedings of the 31st International Conference on Software Engineering*.

[61] Sebastiano Panichella, Andrea Di Sorbo, Emitza Guzman, Corrado A Visaggio, Gerardo Canfora, and Harald C Gall. 2015. How can i improve my app? classifying user reviews for software maintenance and evolution. In *2015 IEEE International Conference on Software Maintenance and Evolution*.

[62] Corina S Păsăreanu, Divya Gopinath, and Huafeng Yu. 2019. Compositional Verification for Autonomous Systems with Deep Learning Components. In *Safe, Autonomous and Intelligent Vehicles*. Springer, 187–197.

[63] Hao Peng, Lili Mou, Ge Li, Yunchuan Chen, Yangyang Lu, and Zhi Jin. 2015. A comparative study on regularization strategies for embedding-based neural networks. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 2106–2111.

[64] Minlong Peng, Qi Zhang, Yu-gang Jiang, and Xuanjing Huang. 2018. Cross-Domain Sentiment Classification with Target Domain Specific Information. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Vol. 1. 2505–2513.

[65] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. GloVe: Global Vectors for Word Representation. In *Empirical Methods in Natural Language Processing (EMNLP)*. 1532–1543.

[66] Ben Poole, Jascha Sohl-Dickstein, and Surya Ganguli. 2014. Analyzing noise in autoencoders and deep networks. *arXiv preprint arXiv:1406.1831* (2014).

[67] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. 2016. On the" naturalness" of buggy code. In *2016 IEEE/ACM 38th International Conference on Software Engineering*.

[68] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. "Why Should I Trust You?": Explaining the Predictions of Any Classifier. In *Proceedings of the 22nd International Conference on Knowledge Discovery and Data Mining*.

[69] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. 2018. Anchors: High-precision model-agnostic explanations. In *Thirty-Second AAAI Conference on Artificial Intelligence*.

[70] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. 2018. Semantically Equivalent Adversarial Rules for Debugging NLP models. In *Association for Computational Linguistics (ACL)*.

[71] Abhik Roychoudhury and Satish Chandra. 2016. Formula-based software debugging. *Commun. ACM* 59, 7 (2016), 68–77.

[72] Stuart J Russell and Peter Norvig. 2016. *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,.

[73] Tobias Schnabel, Igor Labutov, David Mimno, and Thorsten Joachims. 2015. Evaluation methods for unsupervised word embeddings. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*.

[74] Scharolta Katharina Sienčnik. 2015. Adapting word2vec to named entity recognition. In *Proceedings of the 20th nordic conference of computational linguistics*.

[75] Vinayak Sinha, Alina Lazar, and Bonita Sharif. 2016. Analyzing developer sentiment in commit logs. In *Proceedings of the 13th International Conference on Mining Software Repositories*. ACM, 520–523.

[76] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D Manning, Andrew Ng, and Christopher Potts. 2013. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 conference on empirical methods in natural language processing (EMNLP)*. 1631–1642.

[77] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D. Manning, Andrew Ng, and Christopher Potts. 2013. Recursive Deep Models for Semantic Compositionality Over a Sentiment Treebank. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Seattle, Washington, USA, 1631–1642. https://www.aclweb.org/anthology/D13-1170

[78] William N Sumner and Xiangyu Zhang. 2013. Comparative causality: Explaining the differences between executions. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 272–281.

[79] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*. 3104–3112.

[80] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. 2014. Intriguing Properties of Neural Networks. In *International Conference on Learning Representations (ICLR)*.

[81] Ted Tenny. 1988. Program readability: Procedures versus comments. *IEEE Transactions on Software Engineering* 14, 9 (1988), 1271–1279.

[82] Mike Thelwall, Kevan Buckley, Georgios Paltoglou, Di Cai, and Arvid Kappas. 2010. Sentiment strength detection in short informal text. *Journal of the American Society for Information Science and Technology* (2010).

[83] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 303–314.

[84] trader rnn. 2019. *TRADER*. https://github.com/trader-rnn/TRADER

[85] Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. 2019. On Learning Meaningful Code Changes via Neural Machine Translation. *arXiv preprint arXiv:1901.09102* (2019).

[86] Shyam Upadhyay, Manaal Faruqui, Chris Dyer, and Dan Roth. 2016. Cross-lingual models of word embeddings: An empirical comparison. *arXiv preprint arXiv:1604.00425* (2016).

[87] Alex Wang, Amapreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R Bowman. 2018. GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding. *arXiv preprint arXiv:1804.07461* (2018).

[88] Ke Wang, Rishabh Singh, and Zhendong Su. 2017. Dynamic Neural Program Embedding for Program Repair. *arXiv preprint arXiv:1711.07163* (2017).

[89] Ke Wang and Xiaojun Wan. 2018. SentiGAN: generating sentimental texts via mixture adversarial networks. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*. AAAI Press, 4446–4452.

[90] Huihui Wei and Ming Li. 2017. Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code.. In *IJCAI*. 3034–3040.

[91] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*.

[92] Martin White, Christopher Vendome, Mario Linares-Vásquez, and Denys Poshyvanyk. 2015. Toward deep learning software repositories. In *Proceedings of the 12th Working Conference on Mining Software Repositories*.

[93] Adina Williams, Nikita Nangia, and Samuel Bowman. 2018. A Broad-Coverage Challenge Corpus for Sentence Understanding through Inference. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics*.

[94] Yijun Xiao and Kyunghyun Cho. 2016. Efficient character-level document classification by combining convolution and recurrent layers. *arXiv preprint arXiv:1602.00367* (2016).

[95] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhudinov, Rich Zemel, and Yoshua Bengio. 2015. Show, attend and tell: Neural image caption generation with visual attention. In *International Conference on Machine Learning (ICML)*. 2048–2057.

[96] Liang-Chih Yu, Jin Wang, K Robert Lai, and Xuejie Zhang. 2017. Refining word embeddings for sentiment analysis. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*. 534–539.

[97] Andreas Zeller. 1999. Yesterday, my program worked. Today, it does not. Why?. In *ACM SIGSOFT Software engineering notes*.

[98] Dongwen Zhang, Hua Xu, Zengcai Su, and Yunfeng Xu. 2015. Chinese comments sentiment classification based on word2vec and SVMperf. *Expert Systems with Applications* 42, 4 (2015), 1857–1863.

[99] Xiangyu Zhang, Rajiv Gupta, and Youtao Zhang. 2003. Precise dynamic slicing algorithms. In *25th International Conference on Software Engineering, 2003. Proceedings.* IEEE, 319–329.

[100] Xiang Zhang, Junbo Zhao, and Yann LeCun. 2015. Character-level convolutional networks for text classification. In *Advances in neural information processing systems (NIPS)*. 649–657.

[101] Xuezhou Zhang, Xiaojin Zhu, and Stephen Wright. 2018. Training set debugging using trusted items. In *Thirty-Second AAAI Conference on Artificial Intelligence*.

[102] Yingying Zhang and Daqing Hou. 2013. Extracting problematic API features from forum discussions. In *2013 21st International Conference on Program Comprehension (ICPC)*. IEEE, 142–151.

[103] Zhengli Zhao, Dheeru Dua, and Sameer Singh. 2017. Generating natural adversarial examples. *arXiv preprint arXiv:1710.11342* (2017).