# CC2Vec: Distributed Representations of Code Changes

Thong Hoang, Hong Jin Kang, David Lo
Singapore Management University, Singapore
{vdthoang.2016,hjkang.2018,davidlo}@smu.edu.sg

Julia Lawall
Sorbonne University/Inria/LIP6, France
Julia.Lawall@inria.fr

## ABSTRACT

Existing work on software patches often use features specific to a single task. These works often rely on manually identified features, and human effort is required to identify these features for each task. In this work, we propose CC2Vec, a neural network model that learns a representation of code changes guided by their accompanying log messages, which represent the semantic intent of the code changes. CC2Vec models the hierarchical structure of a code change with the help of the attention mechanism and uses multiple comparison functions to identify the differences between the removed and added code.

To evaluate if CC2Vec can produce a distributed representation of code changes that is general and useful for multiple tasks on software patches, we use the vectors produced by CC2Vec for three tasks: log message generation, bug fixing patch identification, and just-in-time defect prediction. In all tasks, the models using CC2Vec outperform the state-of-the-art techniques.

## 1 INTRODUCTION

Patches, used to edit source code, are often created by developers to describe new features, fix bugs, or maintain existing functionality (e.g., API updates, refactoring, etc.). Patches contain two main pieces of information, a log message and a code change. The log message, used to describe the semantics of the code changes, is written in natural language by the developers. The code change indicates the lines of code to remove or add across one or multiple files. Research has shown that the study of historical patches can be employed to solve software engineering problems, such as just-in-time defect prediction [21, 28], identification of bug fixing patches [22, 57], tangled change prediction [34], recommendation of a code reviewer for a patch [50], and many more.

Exploring patches to solve software engineering problems requires choosing a representation of the patch data. Most prior work involves manually crafting a set of features to represent a patch and using these features for further processing [28–30, 44, 57, 60]. These features have mostly been extracted from properties of patches, such as the modifications to source code (e.g., number of removed and added lines, the number of files modified), the history of changes (e.g., the number of prior or recent changes to the updated files), the record of patch authors and reviewers (e.g., the number of developers or reviewers who contributed to the patch), etc. These

features can be used as an input to a machine learning classifier (e.g., Support Vector Machine, Logistic Regression, Random Forest, etc.) to address various software engineering tasks [28, 34, 50, 57]. Extracting a suitable vector representation to represent the "meaning" of a patch is certainly crucial. Intuitively, the quality of a patch representation plays a major role in determining the eventual learning outcome.

In this paper, to boost the effectiveness of existing solutions that employ the properties of patches, we wish to learn vector representations of the code changes in patches that can be used for a number of tasks. We propose a new deep learning architecture named CC2Vec that can effectively embed a code change into a vector space where similar changes are close to each other. As log messages, written by developers, are used to describe the semantics of the code change, we use them to supervise the learning of code changes' representations from patches. Specifically, CC2Vec optimizes the vector representation of a code change in a patch to predict appropriate words, extracted from the first line of the log message. We consider only the first line, as it is the focus of many prior works [39, 49], and is considered to carry the most semantic meaning with the least noise.[1]

CC2Vec analyzes the code change, i.e., scattered fragments of removed and added code across multiple files. Code removed or added from a file follows a hierarchical structure (words form line, lines form hunks). Recent work has suggested that the attention mechanism can help in modelling structural dependencies [3, 32], thus, we hypothesize that the attention mechanism may be effective for modelling the structure of a code change. We propose a specialized hierarchical attention network (HAN) to construct a vector representation of the removed code (and another for the added code) of each affected file in a given patch. Our HAN first builds vector representations of lines; these vectors are then used to construct vector representations of hunks; and we then aggregate these vectors to construct the embedding vector of the removed or added code. Next, we employ multiple comparison functions to capture the difference between two embedding vectors representing removed and added code. This produces features representing the relationship between the removed and added code. Each comparison function produces a vector and these vectors are then concatenated to form an embedding vector for the affected file. Finally, the embedding vectors of all the affected files are concatenated to build a vector representation of the code change in a patch. After training is completed, CC2Vec can be used to extract representations of code changes even from patches with empty or meaningless log messages (which are common in practice [26, 38, 39]). CC2Vec is also programming-language agnostic; one can use it to learn vector representations of code changes for any language.

---

[1]https://chris.beams.io/posts/git-commit/

The code change representation enables us to employ the power of (potentially a large number of) unlabeled patch data to improve the effectiveness of supervised learning tasks (also known as semi-supervised learning [12]). We can use the code change representation to boost the effectiveness of many supervised learning tasks (e.g., identification of bug fixing patches, just-in-time defect prediction, etc.), especially on those tasks for which only a limited set of labeled data may be available.

CC2Vec converts code changes into their distributed representations by learning from a large collection of patches. The distributed representation captures pertinent features of the code changes by considering the characteristics of the whole collection of patches. Such distributed representations can be used as additional features for other tasks. Past studies have demonstrated the value of distributed representations to improve text classification [43], action recognition [40], image classification [13], etc. Unfortunately, prior to our work, there is no existing solution that can produce a distributed representation of a code change.

To evaluate the effectiveness of CC2Vec, we employ the representation learned by CC2Vec in three software engineering tasks: 1) log message generation [39] 2) bug fixing patch identification [22] and 3) just-in-time defect prediction [21]. In the first task of log message generation, we generate the first line of a log message given a code change. CC2Vec can be used to improve over the best baseline by 24.73% in terms of BLEU score (an accuracy measure that is widely used to evaluate machine translation systems). For the task of identifying bug fixing patches, CC2Vec helps to improve the best performing baseline by 5.22%, 9.18%, 4.36%, and 6.51% in terms of accuracy, precision, F1, and Area Under the Curve (AUC). For just-in-time defect prediction, CC2Vec helps to improve the AUC metric by 7.03% and 7.72% on the QT and OPENSTACK datasets [42] as compared to the best baseline.

The main contributions of this work are as follows:

- We propose a deep learning architecture, namely CC2Vec, that learns distributed representations of code changes guided by the semantic meaning contained in log messages. To the best of our knowledge, our work is the first work in this direction.
- We empirically investigate the value of integrating the code change vectors generated by CC2Vec and feature vectors used by state-of-the-art approaches on three tasks (i.e., log message generation, bug fixing patch identification, and just-in-time defect prediction) and demonstrate improvements.

The rest of this paper is organized as follows. Section 2 elaborates the design of CC2Vec. Section 3 describes the experiments that demonstrate the value of our learned code change representations to aid in the three different tasks. Section 4 presents an ablation study and some threats to validity. Section 5 describes related prior studies. We conclude and mention future work in Section 6.

## 2 APPROACH

In this section, we first present an overview of our framework. We then describe the details of each part of the framework. Finally, we present an algorithm for learning effective settings of our model's parameters.
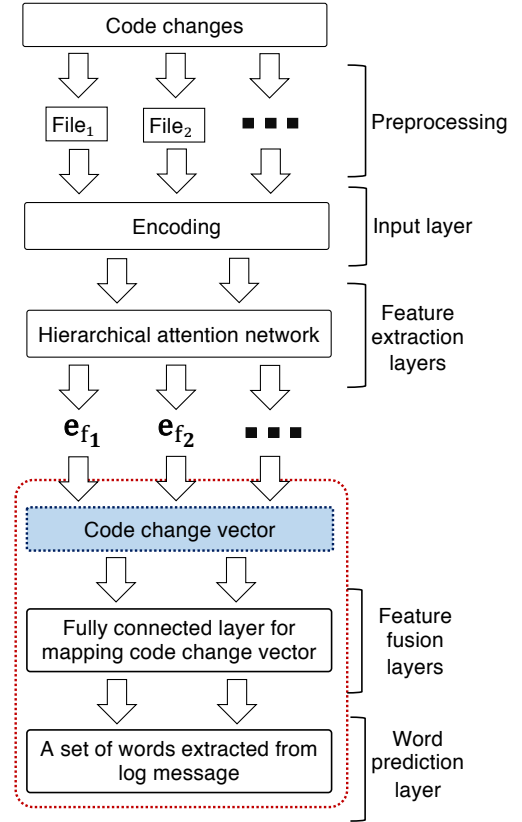


Figure 1: The overall framework of CC2Vec. Feature extraction layers are used to construct the embedding vectors for each affected file from a given patch (i.e., $e_{f_1}$, $e_{f_2}$, etc). The embedding vectors are then concatenated to build a vector representation for the code change in the patch (code change vector). The code change vector is connected to the fully connected layer and is learned by minimizing an objective function of the word prediction layer.

### 2.1 Framework Overview

Figure 1 illustrates the overall framework of CC2Vec. CC2Vec takes the code change of a patch as input and generates its distributed representation. CC2Vec uses the first line of the log message of the patch to supervise learning the code change representation. Specifically, the framework of CC2Vec includes five parts:

- *Preprocessing*: This part takes information from the code change of the given patch as an input and outputs a list of files. Each file includes a set of removed code lines and added code lines.
- *Input layer*: This part encodes each changed file as a three-dimensional matrix to be given as input to the hierarchical attention network (HAN) for extracting features.
- *Feature extraction layers*: This part extracts the embedding vector (a.k.a. features) of each changed file. The resulting embedding vectors are then concatenated to form the vector representation of the code change in a given patch.

• *Feature fusion layers and word prediction layer*: This part maps the vector representation of the code change to a word vector extracted from the first line of log message; the word vector indicates the probabilities that various words describe the patch.

CC2Vec employs the first line of the log message of a patch to guide the learning of a suitable vector that represents the code change. Words, extracted from the first line of log message, can be viewed as semantic labels provided by developers. Specifically, we define a learning task to construct a prediction function $f : P \rightarrow \mathcal{Y}$, where $y_i \in \mathcal{Y}$ indicates the set of words extracted from the first line of the log message of the patch $p_i \in P$. The prediction function $f$ is learned by minimizing the differences between the predicted and actual words chosen to describe the patch. After the prediction function $f$ is learned, for each patch, we can obtain its code change vector from the intermediate output between the feature extraction and feature fusion layers (see Figure 1). We explain the details of each part in the following subsections.

## 2.2 Preprocessing

The code change of the given patch includes changes made to one or more files. Each changed file contains a set of lines of removed code and added code. We process the code change of each patch by the following steps:

• **Split the code change based on the affected files.** We first separate the information about the code change to each changed file into a separate code document (i.e., File$_1$, File$_2$, etc., see Figure 1).
• **Tokenize the removed code and added code lines.** For the changes affecting each changed file, we employ the NLTK library [9] for natural language processing (NLP) to parse its removed code lines or added code lines into a sequence of words. We ignore blank lines in the changed file.
• **Construct a code vocabulary.** Based on the code changes of the patches in the training data, we build a vocabulary $\mathcal{V}^C$. This vocabulary contains the set of code tokens that appear in the code changes of the collection of patches.

At the end of this step, all the changed files of the given patch are extracted from the code changes and they are fed to the input layer of our framework for further processing.

## 2.3 Input Layer

A code change may include changes to multiple files; the changes to each file may contain changes to different hunks; and each hunk contains a list of removed and/or added code lines. To preserve this structural information, in each changed file, we represent the removed (added) code as a three-dimensional matrix, i.e., $\mathcal{B} \in \mathbb{R}^{\mathcal{H} \times \mathcal{L} \times W}$, where $\mathcal{H}$ is the number of hunks, $\mathcal{L}$ is the number of removed (added) code lines for each hunk, and $W$ is the number of words in each removed (added) code line in the affected file. We use $\mathcal{B}_r$ and $\mathcal{B}_a$ to denote the three-dimensional matrix of the removed and added code respectively.

Note that each patch may contain a different number of affected files ($\mathcal{F}$), each file may contain a different number of hunks ($\mathcal{H}$), each hunk may contain a different number of lines ($\mathcal{L}$), and each
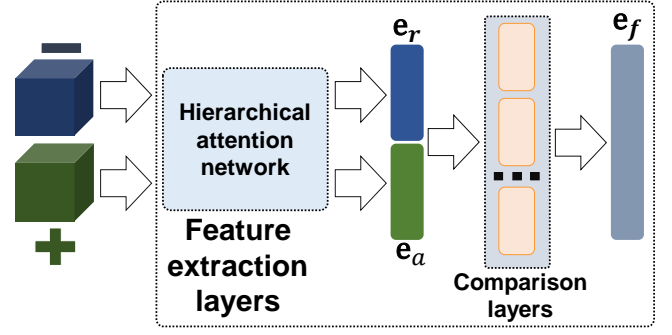


**Figure 2: Architecture of the feature extraction layers for mapping the code change of the affected file in a given patch to an embedding vector. The input of the module is the removed code and added code of the affected file, denoted by "-" and "+", respectively.**

line may contain a different number of words ($W$). For parallelization [21, 31], each input instance is padded or truncated to the same $\mathcal{F}, \mathcal{H}, \mathcal{L},$ and $W$.

## 2.4 Feature Extraction Layers

The feature extraction layers are used to automatically build an embedding vector representing the code change made to a given file in the patch. The embedding vectors of code changes to multiple files are then concatenated into a single vector representing the code change made by the patch.

As shown in Figure 2, for each affected file, the feature extraction layers take as input two matrices (denoted by "-" and "+" in Figure 2) representing the removed code and added code, respectively. These two matrices are passed to the *hierarchical attention network* to construct corresponding embedding vectors: $e_r$ representing the removed code and $e_a$ representing the added code (see Figure 2). These two embedding vectors are fed to the *comparison layers* to produce the vectors representing the difference between the removed code and the added code. These vectors are then concatenated to represent the code changes in each affected file. We present the hierarchical attention network and the comparison layers in the following sections.

*2.4.1 Hierarchical Attention Network.* The architecture of our hierarchical attention network (HAN) is shown in Figure 3. A HAN takes the removed (added) code of an affected file of a given patch as an input and outputs the embedding vector representing the removed (added) code. Our HAN consists of several parts: a word sequence encoder, a word-level attention layer, a line encoder, a line-level attention layer, a hunk sequence encoder, and a hunk attention layer.

Suppose that the removed (added) code of the affected file contains a sequence of hunks $\mathbf{H} = [t_1, t_2, \ldots, t_{\mathcal{H}}]$, each hunk $t_i$ includes a sequence of lines $[s_{i1}, s_{i2}, \ldots, s_{i\mathcal{L}}]$, and each line $s_{ij}$ contains a sequence of words $[w_{ij1}, w_{ij2}, \ldots, w_{ijW}]$. $w_{ijk}$ with $k \in [1, W]$ represents the word in the $j$−th line in the $i$−th hunk. Now, we describe how the embedding vector of the removed (added) code is built using the hierarchical structure.
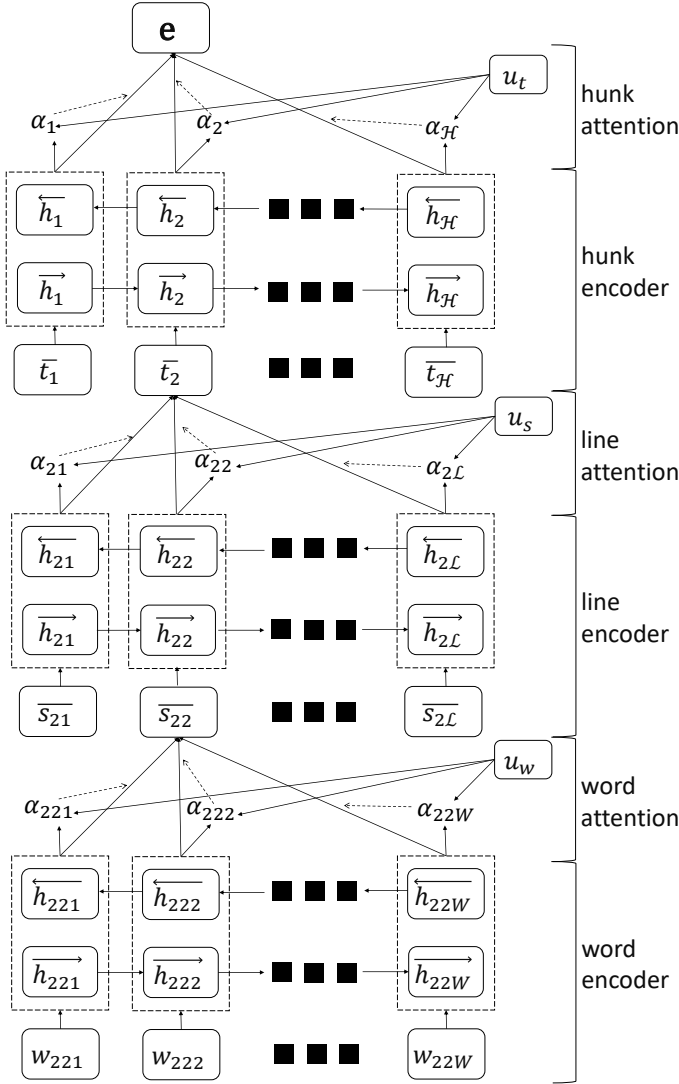
**Figure 3: The overall framework of our hierarchical attention network (HAN). The HAN takes as input the removed (added) code of the affected file of a given patch and outputs the embedding vector (denoted by e) of the removed (added) code.**

**Word encoder.** Given a line $s_{ij}$ with a sequence of words $w_{ijk}$ and a word embedding matrix $\mathbf{W} \in \mathbb{R}^{|\mathcal{V}^C| \times d}$, where $\mathcal{V}^C$ is the vocabulary containing all words extracted from the code changes and $d$ is the dimension of the representation of word, we first build the matrix representation of each word in the sequence as follows:

$$\overline{w_{ijk}} = \mathbf{W}[w_{ijk}] \tag{1}$$

where $\overline{w_{ijk}} \in \mathbb{R}^d$ indicates the vector representation of word $w_{ijk}$ in the word embedding matrix $\mathbf{W}$. We employ a bidirectional GRU to summarize information from the context of a word in both directions [6]. To capture this contextual information, the bidirectional

GRU includes a forward GRU that reads the line $s_{ij}$ from $w_{ij1}$ to $w_{ijW}$ and a backward GRU that reads the line $s_{ij}$ from $w_{ijW}$ to $w_{ij1}$.

$$\overrightarrow{h_{ijk}} = \overrightarrow{GRU}(\overline{w_{ijk}}), k \in [1, W]$$
$$\overleftarrow{h_{ijk}} = \overleftarrow{GRU}(\overline{w_{ijk}}), k \in [W, 1] \tag{2}$$

We obtain an annotation of a given word $w_{ijk}$ by concatenating the forward hidden state $\overrightarrow{h_{ijk}}$ and the backward hidden state $\overleftarrow{h_{ijk}}$ of this word, i.e., $h_{ijk} = [\overrightarrow{h_{ijk}} \oplus \overleftarrow{h_{ijk}}]$ ($\oplus$ is the concatenation operator). $h_{ijk}$ summarizes the word $w_{ijk}$ considering its neighboring words.
**Word attention.** Based on the intuition that not all words contribute equally to extract the "meaning" of the line, we use the attention mechanism to highlight words important for predicting the content of the log message. The attention mechanism was previously used in source code summarization and was shown to be effective for encoding source code sequences [26, 36]. We also use the attention mechanism to form an embedding vector of the line. We first feed an annotation of a given word $w_{ijk}$ (i.e., $h_{ijk}$) through a fully connected layer (i.e., $\mathcal{W}_w$) to get a hidden representation (i.e., $u_{ijk}$) of $h_{ijk}$ as follows:

$$u_{ijk} = ReLU(\mathcal{W}_w h_{ijk} + b_w) \tag{3}$$

where ReLU is the rectified linear unit activation function [45], as it generally provides better performance in various deep learning tasks [4, 14]. Similar to Yang et al. [61], we define a word context vector ($u_w$) that can be seen as a high level representation of the answer to the fixed query "what is the most informative word" over the words. The word context vector $u_w$ is randomly initialized and learned during the training process. We then measure the importance of the word as the similarity of $u_{ijk}$ with the word context vector $u_w$ and get a normalized importance weight $\alpha_{ijk}$ through a softmax function [10]:

$$\alpha_{ijk} = \frac{\exp(u_{ijk}^T u_w)}{\sum_k \exp(u_{ijk}^T u_w)} \tag{4}$$

For each line $s_{ij}$, its vector is computed as a weighted sum of the embedding vectors of the words based on their importance as follows:

$$\overline{s_{ij}} = \sum_k \alpha_{ijk} h_{ijk} \tag{5}$$

**Line encoder.** Given a line vector (i.e., $\overline{s_{ij}}$), we also use a bidirectional GRU to encode the line as follows:

$$\overrightarrow{h_{ij}} = \overrightarrow{GRU}(\overline{s_{ij}}), j \in [1, \mathcal{L}]$$
$$\overleftarrow{h_{ij}} = \overleftarrow{GRU}(\overline{s_{ij}}), j \in [\mathcal{L}, 1] \tag{6}$$

Similar to the word encoder, we obtain an annotation of the line $s_{ij}$ by concatenating the forward hidden state $\overrightarrow{h_{ij}}$ and backward hidden state $\overleftarrow{h_{ij}}$ of this line. The annotation of the line $s_{ij}$ is denoted as $h_{ij} = [\overrightarrow{h_{ij}} \oplus \overleftarrow{h_{ij}}]$, which summarizes the line $s_{ij}$ considering its neighboring lines.

**Line attention.** We use an attention mechanism to learn the important lines to be used to form a hunk vector as follows:

$$u_{ij} = \text{ReLU}(\mathcal{W}_s h_{ij} + b_s) \tag{7}$$

$$\alpha_{ij} = \frac{\exp(u_{ij}^{\mathrm{T}} u_s)}{\sum_j \exp(u_{ij}^{\mathrm{T}} u_s)} \tag{8}$$

$$\overline{t_i} = \sum_j \alpha_{ij} h_{ij} \tag{9}$$

$\mathcal{W}_s$ is the fully connected layer to which we need to feed an annotation of the given line (i.e., $s_{ij}$). We define $u_s$ as the line context vector that can be seen as a high level representation of the answer to the fixed query "what is the informative line" over the lines. $u_s$ is randomly initialized and learned during the training process. $\overline{t_i}$ is the hunk vector of the $i$-th hunk in the removed (added) code.

**Hunk encoder.** Given a hunk vector $\overline{t_i}$, we again use a bidirectional GRU to encode the hunk as follows:

$$\begin{aligned} \overrightarrow{h_i} &= \overrightarrow{GRU}(\overline{t_i}), t \in [1, \mathcal{H}] \\ \overleftarrow{h_i} &= \overleftarrow{GRU}(\overline{t_i}), t \in [\mathcal{H}, 1] \end{aligned} \tag{10}$$

An annotation of the hunk $t_i$ is then obtained by concatenating the forward hidden state $\overrightarrow{h_i}$ and the backward hidden state $\overleftarrow{h_i}$, i.e., $h_i = [\overrightarrow{h_i}, \overleftarrow{h_i}]$. $h_i$ summarizes the hunk $t_i$ considering the other hunks around it.

**Hunk attention.** We again use an attention mechanism to learn important hunks used to form an embedding vector of the removed (added) code as follows:

$$u_i = \text{ReLU}(\mathcal{W}_h h_i + b_h) \tag{11}$$

$$\alpha_i = \frac{\exp(u_i^{\mathrm{T}} u_t)}{\sum_i \exp(u_i^{\mathrm{T}} u_t)} \tag{12}$$

$$\mathbf{e} = \sum_i \alpha_i h_i \tag{13}$$

$\mathcal{W}_h$ is the fully connected layer used to feed an annotation of a given hunk (i.e., $h_i$). $u_t$ is the hunk context vector that can be seen as a high level representation of the answer to the fixed query "what is the informative hunk" over the hunks. Similar to $u_w$ and $u_s$, $u_t$ is randomly initialized and learned during the training process. $\mathbf{e}$, collected at the end of this part, is the embedding vector of the removed (added) code. For convenience, we denote $\mathbf{e}_r$ and $\mathbf{e}_a$ as the embedding vectors of the removed code and added code, respectively.

*2.4.2 Comparison Layers.* The goal of the comparison layers is to build the vectors that capture the differences between the removed code and added code of the affected file in a given patch. We use multiple comparison functions [59] to represent different angles of comparison. These comparison functions were previously used in a question answering task. The comparison layers take as input the embedding vectors of the removed code and added code (denoted by $\mathbf{e}_r$ and $\mathbf{e}_a$, respectively) and output the vectors representing the difference between the removed code and the added code. These vectors are then concatenated to represent an embedding vector
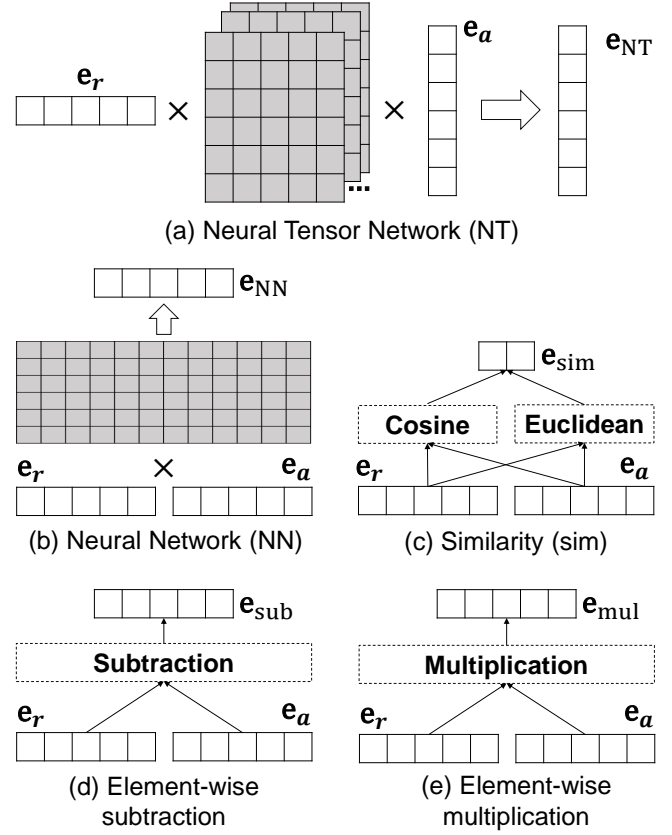


(a) Neural Tensor Network (NT)

(b) Neural Network (NN)

(c) Similarity (sim)

(d) Element-wise subtraction

(e) Element-wise multiplication

**Figure 4: A list of comparison functions in the comparison layers.**

of the affected file in the given patch. Figure 4 shows the five comparison functions used in the comparison layers to capture the difference between the removed code and added code. We briefly explain these comparison functions in the following paragraphs.

**(a) Neural Tensor Network.** Inspired by previous works in visual question answering [7], we employ a neural tensor network [52] as follows:

$$\mathbf{e}_{\text{NT}} = \text{ReLU}(\mathbf{e}_r^{\mathrm{T}} \mathbf{T}^{[1, \dots, n]} \mathbf{e}_a + b_{\text{NT}}) \tag{14}$$

$\mathbf{T}^i \in \mathbb{R}^{n \times n}$ is a tensor and $b_{\text{NT}}$ is the bias value. These parameters are learned during the training process. Note that both the removed code and added code have the same dimension (i.e., $\mathbf{e}_r \in \mathbb{R}^n$, $\mathbf{e}_a$).

**(b) Neural Network.** We consider a simple feed forward neural network [54]. The output is computed as follows:

$$\mathbf{e}_{\text{NN}} = \text{ReLU}(\mathbf{W}[\mathbf{e}_a \oplus \mathbf{e}_r] + b_{\text{NN}}) \tag{15}$$

$\oplus$ is the concatenation operator, the matrix $\mathbf{W} \in \mathbb{R}^{n \times 2n}$, and the bias value $b_{\text{NN}}$ are parameters to be learned.

**(c) Similarity.** We employ two different similarity measures, euclidean distance and cosine similarity, to capture the similarity
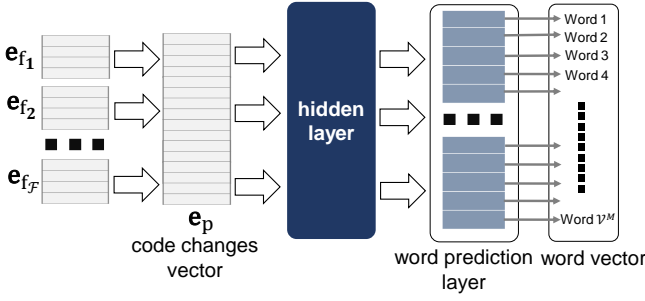
**Figure 5: The details of the red dashed box in Figure 1. It takes as input a list of embedding vectors of the affected files of a given patch (i.e., $e_{f_1}$, $e_{f_2}$, ..., $e_{f_{\mathcal{F}}}$). $e_p$ is the vector representation of the code change and is fed to a hidden layer to produce the word vector (i.e., the probability distribution over words). $\mathcal{V}^M$ is a set of words extracted from the first line of the log messages.**

between the removed code and added code as follows:

$$e_{\text{sim}} = \text{EUC}(e_r, e_a) \oplus \text{COS}(e_r, e_a)$$
$$\text{EUC}(e_r, e_a) = ||e_r - e_a||_2 \qquad (16)$$
$$\text{COS}(e_r, e_a) = \frac{e_r e_a}{||e_r|| ||e_a||}$$

$\text{EUC}(\cdot)$ and $\text{COS}(\cdot)$ are the euclidean distance and cosine similarity, respectively. Note that $e_{\text{sim}}$ is a two-dimensional vector.
**(d) Element-wise subtraction.** We simply perform a subtraction between the embedding vector of the removed code and the embedding vector of the added code.

$$e_{\text{sub}} = e_r - e_a \qquad (17)$$

**(e) Element-wise multiplication.** We perform element-wise multiplication for the embedding vectors of the removed code and added code.

$$e_{\text{mul}} = e_r \odot e_a \qquad (18)$$

where $\odot$ is the element-wise multiplication operator.

The vectors resulting from applying these five different comparison functions are then concatenated to represent the embedding vector of the affected file (denoted by $e_{f_i}$) in the given patch as follows:

$$e_{f_i} = e_{\text{NT}} \oplus e_{\text{NN}} \oplus e_{\text{sim}} \oplus e_{\text{sub}} \oplus e_{\text{mul}} \qquad (19)$$

where $f_i$ is the i-th file of the code change in the given patch.

### 2.5 Feature Fusion and Word Prediction Layers

Figure 5 shows the details of the part of the architecture shown inside the red (dashed) box in Figure 1. The inputs of this part are the list of embedding vectors (i.e., $e_{f_1}$, $e_{f_2}$, ..., $e_{f_{\mathcal{F}}}$) representing the features extracted from the list of affected files of a given patch. These embedding vectors are concatenated to construct a new embedding vector ($e_p$) representing the code change in a given patch as follows:

$$e_p = e_{f_1} \oplus e_{f_2} \oplus \cdots \oplus e_{f_{\mathcal{F}}} \qquad (20)$$

We pass the embedding vector ($e_p$) into a hidden layer (a fully connected layer) to produce a vector $h$:

$$h = \alpha(w_h e_p + b_h) \qquad (21)$$

where $w_h$ is the weight matrix used to connect the embedding vector $e_p$ with the hidden layer and $b_h$ is the bias value. Finally, the vector $h$ is passed to a word prediction layer to produce the following:

$$o = -h w_o \qquad (22)$$

where $w_o$ is the weight matrix between the hidden layer and the word prediction layer, and $o \in \mathbb{R}^{|\mathcal{V}^M| \times 1}$ ($\mathcal{V}^M$ is a set of words extracted from the first line of log messages). We then apply the sigmoid function [10] to get the probability distribution over words as follows:

$$p(o_i|p_i) = \frac{1}{1 + \exp(o_i)} \qquad (23)$$

where $o_i \in o$ is the probability score of the $i^{th}$ word and $p_i$ is the patch that we want to assign words to.

### 2.6 Parameter Learning

Our model involves the following parameters: the word embedding matrix of code changes, the hidden states in the different encoders (i.e., the word encoder, line encoder, and hunk encoder), the context vectors of words, lines, and hunks, the weight matrices and the bias values of the neural tensor network and the neural net in the comparison layers, and the weight matrices and the bias values of the hidden layer and the word prediction layer. After these parameters are learned, the vector representation of the code change of each patch can be determined. These parameters are learned by minimizing the following objective function:

$$O = \sum_{y_i \in y} (y_i \times -\log(p(o_i|p_i)) + (1 - y_i)$$
$$\times -\log(1 - p(o_i|p_i))) + \frac{\lambda}{2}||\theta||_2^2 \qquad (24)$$

where $p(o_i|p_i)$ is the predicted word probability defined in Equation 23, $y_i = \{0, 1\}$ indicates whether the $i$-th word is part of the log message of the patch $p_i$, and $\theta$ are all parameters of our model. The regularization term, $\frac{\lambda}{2}||\theta||_2^2$, is used to prevent overfitting in the training process [11]. We employ the dropout technique [53] to improve the robustness of CC2Vec. Since Adam [33] has been shown to be computationally efficient and require low memory consumption, we use it to minimize the objective function (i.e., Equation 24). We also use backpropagation [19], a simple implementation of the chain rule of partial derivatives, to efficiently update the parameters during the training process.

### 3 EXPERIMENTS

The goal of this work is to build a representation of code changes that can be applied to multiple tasks. To evaluate the effectiveness of this representation, we employ our framework, namely CC2Vec, on three different tasks, i.e., log message generation [39], bug fixing patch identification [22] and just-in-time defect prediction [21].

In the first task of log message generation, we use the vector representation of code changes, extracted by CC2Vec, to find a patch that is most similar to another. For the other two tasks, CC2Vec is used to extract additional features that are input to the models of bug fixing patch identification and just-in-time defect prediction. We compare the resulting performance with and without using our code change vector. We next elaborate on the three tasks, the baselines, and results.

## 3.1 Task 1: Log Message Generation

*3.1.1 Problem Formulation.* While we learn representations of code changes with the aid of log messages, we also study the task of generating log messages from code changes. Developers do not always write high-quality log messages. Dyer et al. [16] reported that around 14% of log messages in 23,000 Java projects on SourceForge[2] were empty. Log messages are important for program comprehension and understanding the evolution of software, therefore this motivates the need for the automatic generation of log messages. In this task, given the code change of a given patch, we aim to produce a brief log message summarizing it.

*3.1.2 State-of-the-art Approach.* The state-of-the-art approach is *NNGen* [39], which takes as input a new code change with an unknown log message and a training dataset (patches), and outputs a log message for the new code change. NNGen first extracts code changes from the training set. Each code change in the training set and the new code change are represented as vectors in the form of a "bag-of-words" [41]. NNGen then calculates the cosine similarity between the vector of the new code change and the vector of each code change in the training set, and selects the top-k nearest neighbouring code changes in the training dataset. From these k nearest neighbours, the BLEU-4 score [48] is computed between each of the code changes in the top-k and the new code change with an unknown log message. A log message of the code change in the top-k with the highest BLEU-4 score is reused as the log message of the new code change.

The BLEU-4 score is a measure used to evaluate the quality of machine translation systems, measuring the closeness of a translation to a human translation. It is computed as follows:

$$BLEU = BP \cdot \exp\left(\sum_{n=1}^{N} \frac{1}{N} \log(p_n)\right)$$

$$BP = \begin{cases} 1 & \text{if } c > r \\ e^{(1-r/c)} & \text{if } c \leq r \end{cases}$$

$N$ is the maximum number of N-grams. Following the previous work [39], we select $N = 4$. $p_n$ is the ratio of length $n$ subsequences that are present in both the output and reference translation. $BP$ is a brevity penalty to penalize short output sentences. Finally, $c$ is the length of the output translation and $r$ is the length of the reference translation.

A deep learning approach was previously proposed for this task by Jiang et al. [26], however, it underperformed the simpler baseline NNGen. In this study, we refer to their work as *NMT*. Their approach modelled this task as a neural machine translation task, translating the code change to a target log message. Like our work, they proposed an attention-based model, however, our work differs

**Table 1: Performance of each approach on the original and cleaned dataset reported in BLEU-4**

|          | LogGen | NNGen | NMT |
|----------|--------|-------|-----|
| *Original* | **43.20** | 38.55 | 31.92 |
| *Clean*    | **20.48** | 16.42 | 14.19 |

from theirs as ours incorporates the structure of code changes. Liu et al. [39] investigated the performance of Jiang et al.'s attention model; they found that once they remove trivial and automatically-generated messages, the performance of the model decreased significantly, suggesting that this model does not generalize.

*3.1.3 Our Approach.* To use CC2Vec for this task, we propose *LogGen*. Similar to the nearest neighbours approach used by Liu et al. [39], LogGen reuses and outputs a log message from the training set. However, instead of treating each code change as a bag of words, LogGen uses code change vectors produced by CC2Vec. CC2Vec is first trained over the training dataset. Given a new code change from the test dataset with an unknown log message, we find the code changes with a known log message that have the closest CC2Vec vector. Like Liu et al. [39], after identifying the closest code changes, we reuse the log message as the output.

*3.1.4 Experimental Setting.* The purpose of evaluating CC2Vec on this task is to determine if the code change representations received from CC2Vec outperform the naive representation used by Liu et al. [39]. Jiang et al. [26] originally collected and filtered the commits to construct the original dataset. Another version of the dataset was used by Liu et al. [39], who modified the original dataset.

Jiang et al. extracted a total of 2 million patches from the 1K most starred Java projects. They collected the first line of each log message. To normalize the dataset, patch ids and issue ids were removed from the code changes and log messages. Patches were filtered to remove merges, rollbacks, and patches that were too long. The log messages that do not conform to verb-direct-object pattern, e.g. "delete a method", are also removed. After filtering, the dataset contains 32K patches.

Still, even with all this cleaning, Liu et al. [39] investigated the dataset and found that there were many patches with bot messages and trivial messages. Bot messages refer to messages produced automatically by other development tools, such as continuous integration bots. Trivial messages refer to messages containing only information that can be obtained by looking at the names of the changed files (e.g. "modify dockerfile"). Such messages are of low quality and Liu et al. used regular expressions to locate and remove these patches.

We used the original dataset of Jiang et al. [26] and the cleaned dataset of Liu et al. [39] for evaluation. While the original dataset consists of a training dataset of 30K patches and a testing dataset of 3K patches, the cleaned dataset consists of a training dataset of 22K patches and a testing dataset of 2.5K patches. To compare the different approaches, we use BLEU-4 to evaluate each approach since this was used in both previous works.

*3.1.5 Results.* We report the performance of LogGen, NNGen and NMT in Table 1. LogGen outperforms both NNGen and NMT. The

---

*Clean* dataset refers to the dataset which Liu et al. filtered out patches with bot and trivial log messages. On this dataset, LogGen outperforms NNGen and NMT by a BLEU-4 score of 4.06 and 6.29 respectively. LogGen *improves* over the performance of NNGen by 24.75%, a greater improvement than NNGen's improvement over NMT of 15.70% . On the original dataset collected by Jiang et al., LogGen outperforms NNGen and NMT by a BLEU-4 score of 4.65 and 11.28. These results indicate that LogGen can improve over the performance of NNGen and NMT by 12.06% and 2.07% in terms of the BLEU-4 score respectively.

Thus, we conclude that the log messages retrieved by LogGen are closer in quality to a human translation than those retrieved by NNGen and the log messages generated by NMT. This suggests that CC2Vec produces vector representations of patches that correlate to the meaning of the patch more strongly than a bag-of-words.

## 3.2 Task 2: Bug Fixing Patch Identification

*3.2.1 Problem Formulation.* Software requires continuous evolution to keep up with new requirements, but this also introduces new bugs. Backporting bugfixes to older versions of a project may be required when a legacy code base is supported. For example, Linux kernel developers regularly backport bugfixes from the latest version to older versions that are still under support. However, the maintainers of older versions may overlook relevant patches in the latest version. Thus, an automated method to identify bug fixing patches may be helpful. We treat the problem as a binary classification problem, in which each patch is labelled as a bug-fixing patch or not. Given the code change and log message, we produce one of the two labels as the output.

*3.2.2 State-of-the-art Approach.* The state-of-the-art approach is *PatchNet* [22], which represents the removed (added) code as a three dimensional matrix. The dimensions of the matrix are the number of hunks, the number of lines in each hunk, and the number of words in each line. PatchNet employs a 3D-CNN [25] that automatically extracts features from this matrix. Unfortunately, the 3D-CNN lacks a mechanism to identify important words, lines, and hunks. To address this limitation, we propose a specialized hierarchical attention neural network to quantify the importance of words, lines, and hunks in our model (CC2Vec).

Another approach was proposed by Tian et al. [57] that combines Learning from Positive and Unlabelled examples (LPU) [37] and Support Vector Machine (SVM) [27] to build a patch classification model. Unlike CC2Vec, this approach requires the use of manually selected features. These features include word features, which is a "bag-of-words" extracted from log messages, and 52 features, manually extracted from the code change (e.g., the number of loops added in a patch and if certain words appear in the log message).

*3.2.3 Our Approach.* CC2Vec is first used to learn a distributed representation of code changes on the whole dataset. All patches from the training and test dataset are used since the log messages of the test dataset are not the target of the task. Next, we integrate these vector representations of the code changes with the two existing approaches. To use CC2Vec in PatchNet, we concatenate the vector representation of the code change extracted by CC2Vec with the two embedding vectors extracted from the log message and

**Table 2: Evaluation of the approaches on the bug-fixing patch identification task**

|                  | Acc.  | Prec. | Recall | F1    | AUC   |
|------------------|-------|-------|--------|-------|-------|
| LPU-SVM          | 73.1  | 75.1  | 71.6   | 73.3  | 73.1  |
| LPU-SVM + CC2Vec | **77.1** | **77.2** | **79.8** | **78.5** | **76.2** |
| PatchNet         | 86.2  | 83.9  | 90.1   | 87.1  | 86.0  |
| PatchNet + CC2Vec | **90.7** | **91.6** | **90.1** | **90.9** | **91.6** |

code change by PatchNet to form a new embedding vector. The new embedding vector is fed into PatchNet's classification module to predict whether a given patch is a bug fixing patch. For the approach proposed by Tian et al. [57] which uses an SVM as the classifier, we pass the vectors produced by CC2Vec from the code change into the SVM as features.

*3.2.4 Experimental Setting.* The goal of this task is to investigate if CC2Vec helps existing approaches to effectively classify bug-fixing patches. We use the dataset of Linux kernel bug-fixing patches used in the PatchNet paper. This dataset consists of 42K bug-fixing patches and 40K non-bug-fixing patches collected from the Linux kernel versions v3.0 to v4.12, released in July 2011 and July 2017 respectively. Patches in this dataset are limited to 100 lines of changed code, in line with the Linux kernel stable patch guidelines. The non-bug-fixing patches are selected such that they have a similar size, in terms of the number of files and the number of modified lines, as the bug-fixing patches. Following the PatchNet paper, we use 5-fold cross-validation for the evaluation.

To compare the performance of the approaches, we employ the following metrics:

- *Accuracy*: The ratio of correct predictions to the total number of predictions.
- *Precision*: The ratio of correct predictions of bug-fixing patches to the total number of bug-fixing patch predictions
- *Recall*: The ratio of correct predictions of bug-fixing patches to the total number of bug-fixing patches.
- *F1*: Harmonic mean between precision and recall.
- *AUC*: Area under the curve plotting the true positive rate against the false positive rate. AUC values range from 0 to 1, with a value of 1 indicating perfect discrimination.

These metrics were also used in previous studies on this task.

*3.2.5 Results.* We report the performance of the different approaches in Table 2. We observe that the best performing approach is PatchNet augmented with CC2Vec. For both Tian et al.'s model (LPU-SVM) and PatchNet, the versions augmented with CC2Vec outperform the original versions. Specifically, CC2Vec helps to improve the best performing baseline (i.e, PatchNet) by 5.22%, 9.18%, 4.37%, and 6.51% in terms of accuracy, precision, F1, and AUC. CC2Vec also helps to improve the performance of LPU-SVM by 5.47%, 2.80%, 11.45%, 7.09%, and 4.24% in accuracy, precision, recall, F1, and AUC. This suggests that CC2Vec can learn patch representations that are general and useful beyond the task it was trained on.

**Table 3: The AUC results of the various approaches**

|              | QT   | OPENSTACK |
| ------------ | ---- | --------- |
| DeepJIT      | 76.8 | 75.1      |
| DeepJIT + CC2Vec | **82.2** | **80.9** |

## 3.3 Task 3: Just-in-Time Defect Prediction

*3.3.1 Problem Formulation.* The task of just-in-time (JIT) defect prediction refers to the identification of defective patches. JIT defect prediction tools provide early feedback to software developers to optimize their effort for inspection, and have been used at large software companies [44, 51, 55]. We model the task as a binary classification task, in which each patch is labelled as a patch containing a defect or not. Given a patch containing a code change and a log message with unknown label, we label the patch with one of the two labels.

*3.3.2 State-of-the-art Approach.* The state-of-the-art approach is *DeepJIT*, proposed by Hoang et al. [21]. DeepJIT takes as input the log message and code change of a given patch and outputs a probability score to predict whether the patch is buggy. DeepJIT employs a Convolutional Neural Network (CNN) [31] to automatically extract features from the code change and log message of the given patch. However, DeepJIT ignores information about the structure of the removed code or added code, instead relying on CNN to automatically extract such information.

*3.3.3 Our Approach.* Similar to the previous task (i.e., bug fixing patch identification), CC2Vec is first used to learn distributed representations of the code changes in the whole dataset. All patches from the training and test dataset are used since the log messages of the test dataset are not part of the predictions of the task. We then integrate CC2Vec with DeepJIT. To use CC2Vec with DeepJIT, for each patch, we concatenate the vector representation of the code change extracted by CC2Vec with two embedding vectors extracted from the log message and code change of the given patch extracted by DeepJIT to form a new embedding vector. The new embedding vector is fed into DeepJIT's feature combination layers, to predict whether the given patch is defective.

*3.3.4 Experimental Setting.* The purpose of this task is to evaluate if CC2Vec can be used to augment existing approaches in effectively classifying defective patches. Our evaluation is performed on two datasets, the *QT* and *OPENSTACK* datasets, which contain patches collected from the QT and OPENSTACK software projects respectively by McIntosh and Kamei [42]. The QT dataset contains 25K patches over 2 years and 9 months while the OPENSTACK dataset contains 12K patches over 2 years and 3 months. 8% and 13% of the patches are defective in the QT dataset and the OPENSTACK datasets respectively. Like Hoang et al. [21], we use 5-fold cross validation for the evaluation.

To compute the effectiveness of the approaches, we use the Area Under the receiver operator characteristics Curve (AUC), similar to the previous studies.

*3.3.5 Results.* The evaluation results for this task are reported in Table 3. The use of CC2Vec with DeepJIT improves the AUCS score of DeepJIT, from 76.8 and 75.1 to 82.2 and 80.9 on the QT

and OPENSTACK datasets respectively. Specifically, CC2Vec helps to improve the AUC metric by 7.03% and 7.72% for the QT and OPENSTACK datasets, respectively, as compared to DeepJIT. This indicates that CC2Vec is effective in learning a useful representation of patches that an existing state-of-the-art technique can utilize.

## 4 DISCUSSION

### 4.1 Ablation Study

Our approach involves five comparison functions for calculating the difference between the removed code and added code. To estimate the usefulness of comparison functions (see Section 2.4.2), we conduct an ablation study on the three tasks: log message generation, bug fixing patch identification, and just-in-time defect prediction. Specifically, we first remove the comparison functions entirely and then remove these functions one-by-one. For each task, we compare the CC2Vec model and its six reduced variants: All−all (omit all comparison functions), All−NT (omit the neural network tensor comparison function), All−NN (omit the neural network comparison function), All−sim (omit the similarity comparison function), All−sub (omit the subtraction comparison function), and All−mul (omit the multiplication comparison function).

Table 4 summarizes the results of our ablation test on three different tasks. We see that CC2Vec model always performs better than the reduced variants for all three tasks. This suggests that each comparison function plays an important role and omitting these comparison functions may greatly affect the overall performance. All−all (CC2Vec model without using any comparison functions) performs the worst. Among the five remaining variants (i.e., All−NT, All−NN, All−sim, All−sub, and All−mul), All-NT performs the worst. This suggests that the *neural network tensor* comparison function is more important the other comparison functions (i.e., *neural network*, *similarity*, *subtraction*, and *multiplication*).

### 4.2 Threats to Validity

Threats to internal validity refer to errors in our experiments and experimenter bias. For each task, we reuse existing implementations of the baseline approaches whenever available. We have double checked our code and data, but errors may remain.

Threats to external validity concern the generalizability of our work. In our experiments, we have studied only three tasks to evaluate the generality of CC2Vec. This may be a threat to external validity since CC2Vec may not generalize beyond the tasks that we have considered. However, each task involves different software projects and different programming languages. As such, we believe that there is minimal threat to external validity. To minimize threats to construct validity, we have used the same evaluation metrics that were used in previous studies.

## 5 RELATED WORK

There are many studies on the representation of source code, including recent studies proposing distributed representations for identifiers [17], APIs [46, 47], and software libraries [56]. A comprehensive survey of learning the representation of source code has been done by Allamanis et al. [1].

Some studies transform the source code into a different form, such as control-flow graphs [15] and symbolic traces [20], or collect

**Table 4: Results of an ablation study**

| | Log generation (BLEU-4) | | Bug fix identification (F1) | | Just-in-time defect prediction (AUC) | | | |
|---|---|---|---|---|---|---|---|---|
| | *Clean* | Drops by (%) | BFP | Drops by (%) | QT | Drops by (%) | OPENSTACK | Drops by (%) |
| All−all | 18.30 | 10.64 | 87.1 | 4.18 | 77.4 | 5.84 | 76.7 | 5.19 |
| All−NT | 19.36 | 5.47 | 88.7 | 2.42 | 79.8 | 2.92 | 79.2 | 2.10 |
| All−NN | 19.80 | 3.32 | 88.8 | 2.31 | 80.1 | 2.55 | 79.5 | 1.73 |
| All−sim | 20.41 | 0.34 | 90.2 | 0.77 | 81.9 | 0.36 | 80.5 | 0.49 |
| All−sub | 20.13 | 1.71 | 89.6 | 1.43 | 80.7 | 1.82 | 80.1 | 0.99 |
| All−mul | 20.25 | 1.12 | 89.7 | 1.32 | 81.1 | 1.34 | 80.5 | 0.49 |
| All | **20.48** | 0 | **90.9** | 0 | **82.2** | 0 | **80.9** | 0 |

runtime execution traces [58], before learning distributed representations. DeFreez et al. [15] found function synonyms by learning embeddings through random walks of the interprocedural control-flow graph of a program. These embeddings are then used in a single downstream task of mining error-handling specifications. Henkel et al. [20] described a toolchain to produce abstracted intraprocedural symbolic traces for learning word embeddings. They experimented on a downstream task to find and repair bugs related to incorrect error codes. Wang et al. [58] used execution traces to learn embeddings. They integrate their embeddings into a program repair system in order to produce fixes to correct student errors in programming assignments. These studies differ from our work as we leverage natural language data as well as source code.

There have been other studies using deep learning of both source code and natural language data, for example, joint learning of embeddings for both text and source code to improve code search [18]. Other studies proposed approaches to learn distributed representations of source code on prediction tasks with natural language output. Iyer et al. [24] proposed a model using LSTM networks with attention for code summarization, and Yin et al. [62] trained a model to align source code to natural language text from StackOverflow posts. However, unlike our work, these studies do not use structural information of the source code.

Several studies [2, 3, 23, 35] account for structural information but differ from our work. Hu et al. [23] proposed an approach to use Sequence-to-Sequence Neural Machine Translation to generate method-level code comments. By prefixing the AST node type in each token and traversing the AST of methods such that the original AST can be unambiguously reconstructed, they convert the AST of each method into a sequence that preserves structural information. Alon et al. proposed code2vec [3], which represents code as paths in an AST, learning the vector representation of each AST path. They trained their model on the task of predicting a label, such as the method name, of the code snippet. In a later work, they proposed code2seq [2]. Instead of predicting a single label, they generate a sequence of natural language words. Similar to our work, structural information of the input source code is encoded in the model's architecture, however, in these studies, the input code snippet is required to be parseable to build an AST.

As our work focuses on the representation of software patches, we deliberately designed CC2Vec to not require parseable code in its input. This is done for two reasons. Firstly, a small but still significant proportion of patches may have compilation errors. A study by Beller et al. on Travis CI build failures revealed that about

4% of Java project build failures are due to compilation errors [8]. CC2Vec is designed to be usable even for these patches. Secondly, parsing will require the entire file with the changed code. Retrieving this information and parsing the entire file will be time consuming.

All the studies above proposed general representations of source code. The representations they learn, with the exception of DeFreez et al. [15], are of source code contained in a single function. In contrast, we learn representations of code changes, which can contain modifications to multiple different functions, across multiple files.

Several of the models related to code changes' representation were discussed in Section 3. These models often do not model the hierarchical structure of a code change or require handcrafted features that may be specific to a single task [5, 26, 28–30, 39, 44, 57, 60].

Two techniques using deep neural networks, PatchNet [22] and DeepJIT [21], are most similar to our work. However, as discussed earlier, our work differs from theirs in various ways. A fundamental difference is in the generality of the techniques. CC2Vec is not specific to a single task. Rather, CC2Vec can be trained for multiple tasks, including both generative and classification tasks. In fact, CC2Vec is orthogonal to these approaches. The objective of CC2Vec is to produce high quality representations of code changes that can be integrated into PatchNet, DeepJIT, and similar models. We showed in Section 3 that the performance of these models improves when they are augmented with the code change representation learned by CC2Vec.

## 6 CONCLUSION

We propose CC2Vec, which produces distributed representations of code changes through a hierarchical attention network. In CC2Vec, we model the structural information of a code change and use the attention mechanism to identify important aspects of the code change with respect to the log message accompanying it. This allows CC2Vec to learn high-quality vector representations that can be used in existing state-of-the-art models on tasks involving code changes.

We empirically evaluated CC2Vec on three tasks and demonstrated that approaches using or augmented with CC2Vec embeddings outperform existing state-of-the-art approaches that do not use the embeddings. Finally, we performed an ablation study to evaluate the usefulness of comparison functions. The results show that the comparison functions play an important role and omitting them in part or in full affects the overall performance.

As future work, to reduce the threat to external validity, we will integrate of CC2Vec into other tools and experiments on other tasks involving software patches.

**Package.** The replication package is available at https://github.com/CC2Vec/CC2Vec.

## REFERENCES

[1] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)* 51, 4 (2018), 81.

[2] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019. code2seq: Generating Sequences from Structured Representations of Code. In *International Conference on Learning Representations*.

[3] 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.

[4] George A Anastassiou. 2011. Univariate hyperbolic tangent neural network approximation. *Mathematical and Computer Modelling* 53, 5-6 (2011), 1111–1132.

[5] Lerina Aversano, Luigi Cerulo, and Concettina Del Grosso. 2007. Learning from bug-introducing changes to prevent fault prone code. In *Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting*. ACM, 19–26.

[6] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473* (2014).

[7] Yalong Bai, Jianlong Fu, Tiejun Zhao, and Tao Mei. 2018. Deep attention neural tensor network for visual question answering. In *Proceedings of the European Conference on Computer Vision (ECCV)*. 20–35.

[8] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. Oops, my tests broke the build: An explorative analysis of Travis CI with GitHub. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 356–367.

[9] Steven Bird and Edward Loper. 2004. NLTK: the natural language toolkit. In *Proceedings of the ACL 2004 on Interactive poster and demonstration sessions*. Association for Computational Linguistics, 31.

[10] Guillaume Bouchard. 2007. Efficient bounds for the softmax function and applications to approximate inference in hybrid models. In *NIPS 2007 workshop for approximate Bayesian inference in continuous/hybrid systems*.

[11] Rich Caruana, Steve Lawrence, and C Lee Giles. 2001. Overfitting in neural nets: Backpropagation, conjugate gradient, and early stopping. In *Advances in neural information processing systems*. 402–408.

[12] Olivier Chapelle, Bernhard Scholkopf, and Alexander Zien. 2009. Semi-supervised learning (chapelle, o. et al., eds.; 2006)[book reviews]. *IEEE Transactions on Neural Networks* 20, 3 (2009), 542–542.

[13] Brian Cheung. 2012. Convolutional neural networks applied to human face classification. In *2012 11th International Conference on Machine Learning and Applications*, Vol. 2. IEEE, 580–583.

[14] George E Dahl, Tara N Sainath, and Geoffrey E Hinton. 2013. Improving deep neural networks for LVCSR using rectified linear units and dropout. In *2013 IEEE international conference on acoustics, speech and signal processing*. IEEE, 8609–8613.

[15] Daniel DeFreez, Aditya V Thakur, and Cindy Rubio-González. 2018. Path-based function embedding and its application to error-handling specification mining. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 423–433.

[16] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N Nguyen. 2013. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 422–431.

[17] Vasiliki Efstathiou and Diomidis Spinellis. 2019. Semantic source code models using identifier embeddings. In *Proceedings of the 16th International Conference on Mining Software Repositories*. IEEE Press, 29–33.

[18] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 933–944.

[19] Martin T Hagan and Mohammad B Menhaj. 1994. Training feedforward networks with the Marquardt algorithm. *IEEE transactions on Neural Networks* 5, 6 (1994), 989–993.

[20] 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*. ACM, 163–174.

[21] Thong Hoang, Hoa Khanh Dam, Yasutaka Kamei, David Lo, and Naoyasu Ubayashi. 2019. DeepJIT: an end-to-end deep learning framework for just-in-time defect prediction. In *Proceedings of the 16th International Conference on Mining Software Repositories*. IEEE Press, 34–45.

[22] Thong Hoang, Julia Lawall, Yuan Tian, Richard J Oentaryo, and David Lo. 2019. PatchNet: Hierarchical Deep Learning-Based Stable Patch Identification for the Linux Kernel. *IEEE Transactions on Software Engineering* (2019).

[23] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension*. ACM, 200–210.

[24] 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 (Volume 1: Long Papers)*. 2073–2083.

[25] Shuiwang Ji, Wei Xu, Ming Yang, and Kai Yu. 2012. 3D convolutional neural networks for human action recognition. *IEEE transactions on pattern analysis and machine intelligence* 35, 1 (2012), 221–231.

[26] Siyuan Jiang, Ameer Armaly, and Collin McMillan. 2017. Automatically generating commit messages from diffs using neural machine translation. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 135–146.

[27] Thorsten Joachims. 1999. Svmlight: Support vector machine. *SVM-Light Support Vector Machine http://svmlight. joachims. org/, University of Dortmund* 19, 4 (1999).

[28] Yasutaka Kamei, Takafumi Fukushima, Shane McIntosh, Kazuhiro Yamashita, Naoyasu Ubayashi, and Ahmed E Hassan. 2016. Studying just-in-time defect prediction using cross-project models. *Empirical Software Engineering* 21, 5 (2016), 2072–2106.

[29] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. 2012. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering* 39, 6 (2012), 757–773.

[30] Sunghun Kim, E James Whitehead Jr, and Yi Zhang. 2008. Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering* 34, 2 (2008), 181–196.

[31] Yoon Kim. 2014. Convolutional Neural Networks for Sentence Classification. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, Doha, Qatar, 1746–1751. https://doi.org/10.3115/v1/D14-1181

[32] Yoon Kim, Carl Denton, Luong Hoang, and Alexander M Rush. 2017. Structured attention networks. *arXiv preprint arXiv:1702.00887* (2017).

[33] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[34] Hiroyuki Kirinuki, Yoshiki Higo, Keisuke Hotta, and Shinji Kusumoto. 2014. Hey! are you committing tangled changes?. In *Proceedings of the 22nd International Conference on Program Comprehension*. ACM, 262–265.

[35] Vladimir Kovalenko, Egor Bogomolov, Timofey Bryksin, and Alberto Bacchelli. 2019. PathMiner: a library for mining of path-based representations of code. In *Proceedings of the 16th International Conference on Mining Software Repositories*. IEEE Press, 13–17.

[36] Alexander LeClair, Siyuan Jiang, and Collin McMillan. 2019. A neural model for generating natural language summaries of program subroutines. In *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, 795–806.

[37] Wee Sun Lee and Bing Liu. 2003. Learning with positive and unlabeled examples using weighted logistic regression. In *ICML*, Vol. 3. 448–455.

[38] Mario Linares-Vásquez, Luis Fernando Cortés-Coy, Jairo Aponte, and Denys Poshyvanyk. 2015. Changescribe: A tool for automatically generating commit messages. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. IEEE, 709–712.

[39] 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.

[40] Subhransu Maji, Lubomir Bourdev, and Jitendra Malik. 2011. Action recognition from a distributed representation of pose and appearance. In *CVPR 2011*. IEEE, 3177–3184.

[41] Christopher Manning, Prabhakar Raghavan, and Hinrich Schütze. 2010. Introduction to information retrieval. *Natural Language Engineering* 16, 1 (2010), 100–103.

[42] Shane McIntosh and Yasutaka Kamei. 2017. Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction. *IEEE Transactions on Software Engineering* 44, 5 (2017), 412–428.

[43] 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*. 3111–3119.

[44] Audris Mockus and David M Weiss. 2000. Predicting risk of software changes. *Bell Labs Technical Journal* 5, 2 (2000), 169–180.

[45] Vinod Nair and Geoffrey E Hinton. 2010. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*. 807–814.

[46] Trong Duc Nguyen, Anh Tuan Nguyen, and Tien N Nguyen. 2016. Mapping API elements for code migration with vector representations. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 756–758.

[47] Trong Duc Nguyen, Anh Tuan Nguyen, Hung Dang Phan, and Tien N Nguyen. 2017. Exploring API embedding for API usages and applications. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 438–449.

[48] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*. Association for Computational Linguistics, 311–318.

[49] Mohammad Masudur Rahman and Chanchal K Roy. 2015. TextRank based search term identification for software change tasks. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 540–544.

[50] Mohammad Masudur Rahman, Chanchal K Roy, and Jason A Collins. 2016. Correct: code reviewer recommendation in github based on cross-project and technology experience. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 222–231.

[51] Emad Shihab, Ahmed E Hassan, Bram Adams, and Zhen Ming Jiang. 2012. An industrial study on the risk of software changes. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 62.

[52] 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*. 1631–1642.

[53] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research* 15, 1 (2014), 1929–1958.

[54] Daniel Svozil, Vladimir Kvasnicka, and Jiri Pospichal. 1997. Introduction to multilayer feed-forward neural networks. *Chemometrics and intelligent laboratory systems* 39, 1 (1997), 43–62.

[55] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, Akinori Ihara, and Kenichi Matsumoto. 2015. The impact of mislabelling on the performance and interpretation of defect prediction models. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 812–823.

[56] Bart Theeten, Frederik Vandeputte, and Tom Van Cutsem. 2019. Import2vec learning embeddings for software libraries. In *Proceedings of the 16th International Conference on Mining Software Repositories*. IEEE Press, 18–28.

[57] Yuan Tian, Julia Lawall, and David Lo. 2012. Identifying Linux bug fixing patches. In *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 386–396.

[58] Ke Wang, Rishabh Singh, and Zhendong Su. 2017. Dynamic neural program embedding for program repair. *arXiv preprint arXiv:1711.07163* (2017).

[59] Shuohang Wang and Jing Jiang. 2017. A Compare-Aggregate Model for Matching Text Sequences. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. https://openreview.net/forum?id=HJTzHtqee

[60] Xinli Yang, David Lo, Xin Xia, Yun Zhang, and Jianling Sun. 2015. Deep learning for just-in-time defect prediction. In *2015 IEEE International Conference on Software Quality, Reliability and Security*. IEEE, 17–26.

[61] Zichao Yang, Diyi Yang, Chris Dyer, Xiaodong He, Alex Smola, and Eduard Hovy. 2016. Hierarchical attention networks for document classification. In *Proceedings of the 2016 conference of the North American chapter of the association for computational linguistics: human language technologies*. 1480–1489.

[62] Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to mine aligned code and natural language pairs from stack overflow. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. IEEE, 476–486.