Information and Software Technology 104 (2018) 60–71

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Improving code readability classification using convolutional neural networks

Qing Mi[*,a], Jacky Keung[a], Yan Xiao[a], Solomon Mensah[a], Yujin Gao[b]

[a] Department of Computer Science, City University of Hong Kong, Kowloon, Hong Kong
[b] School of Computer Science and Technology, Beijing Institute of Technology, Beijing, China

ARTICLE INFO

Keywords:
Code readability
Convolutional Neural Network
Deep learning
Program comprehension
Empirical software engineering
Open source software

ABSTRACT

Context: Code readability classification (which refers to classification of a piece of source code as either readable or unreadable) has attracted increasing concern in academia and industry. To construct accurate classification models, previous studies depended mainly upon handcrafted features. However, the manual feature engineering process is usually labor-intensive and can capture only partial information about the source code, which is likely to limit the model performance.

Objective: To improve code readability classification, we propose the use of Convolutional Neural Networks (ConvNets).

Method: We first introduce a representation strategy (with different granularities) to transform source codes into integer matrices as the input to ConvNets. We then propose DeepCRM, a deep learning-based model for code readability classification. DeepCRM consists of three separate ConvNets with identical architectures that are trained on data preprocessed in different ways. We evaluate our approach against five state-of-the-art code readability models.

Results: The experimental results show that DeepCRM can outperform previous approaches. The improvement in accuracy ranges from 2.4% to 17.2%.

Conclusions: By eliminating the need for manual feature engineering, DeepCRM provides a relatively improved performance, confirming the efficacy of deep learning techniques in the task of code readability classification.

1. Introduction

About 70% of the lifecycle cost of a software project is in the maintenance phase [1]. Estimates show that this high maintenance cost correlates strongly with the difficulty of reading and understanding source code, particularly that written by others [2–4], which highlights the importance of a good measurement of code readability.

Code readability refers to a human judgment of how easily a program's source code can be read and understood [5]. In a survey of the information needs of 110 developers and managers at Microsoft, 90% of participants recorded that they would use readability metrics if they were available [6], but little relevant research has been reported. For instance, Buse and Weimer [5] constructed a code readability model based on a simple set of local code features and showed that it could be 75%–80% effective at predicting human readability judgments. Making use of the well-known size and Halstead metrics, Posnett et al. [7] presented another model that was simpler and performed better than that of Buse and Weimer.

Despite the encouraging results, several problems limit the performance and generality of the existing approaches. The most significant threat is likely the widely used feature engineering process. As discussed above, previous studies (including but not limited to [5,7]) usually handcrafted surface-level features such as operator counts or line lengths [8] to represent code readability. The process is admittedly labor-intensive, and it is possible that some important factors will be missed. Thus, instead of manual design and extraction of features, we introduce a deep learning-based approach that can learn complicated underlying features automatically from the source code. In particular, we use Convolutional Neural Networks (ConvNets). The approach is not another model based on feature engineering, but a constructive method of gaining a deep understanding of the input data, which makes our study novel in this regard.

* Corresponding author.
E-mail addresses: qing.mi@my.cityu.edu.hk (Q. Mi), jacky.keung@cityu.edu.hk (J. Keung), yanxiao6-c@my.cityu.edu.hk (Y. Xiao), smensah2-c@my.cityu.edu.hk (S. Mensah), paulgyj@bit.edu.cn (Y. Gao).

https://doi.org/10.1016/j.infsof.2018.07.006
Received 10 November 2017; Received in revised form 4 July 2018; Accepted 7 July 2018
Available online 10 July 2018
0950-5849/

The contributions of this paper are threefold:

- A simple and customizable representation strategy (with three different granularities) is introduced to convert source codes into integer matrices as the input to ConvNets. The method enables deep learning-based program analyses, which can be easily applied to other software engineering activities.
- DeepCRM, a deep learning-based framework that integrates three separate ConvNets (corresponding to the three granularities), is proposed for code readability classification. Our approach eliminates the need for manual feature engineering and can learn various levels of features automatically from the source code.
- A new dataset, which contains more than 25,000 code snippets retrieved from open source Java projects, is released to support future code readability studies.[1]

To validate our approach, we compare the performance of DeepCRM with five state-of-the-art code readability models, namely, those of *Buse and Weimer* [5], *Posnett et al.* [7], *Dorn* [8], and *Scalabrino et al.* [9], and *A Comprehensive Model* [9]. The experimental results show that DeepCRM outperforms the best competitor, reaching 83.8% accuracy and 83.5% f-measure. Although deep learning has achieved remarkable success in other areas, to the best of our knowledge, we are the first to observe its effectiveness in code readability classification.

The rest of this paper is organized as follows. Section 2 introduces the background and related work. Section 3 explains the motivation for this study. In Section 4, we describe the proposed approach. Section 5 presents the detailed design of our experiments and Section 6 summarizes the results. In Section 7, we discuss our findings, followed by Section 8 that investigates the threats to validity. We conclude with suggestions for future work in Section 9.

2. Background and related work

In this section, we discuss related work on code readability research and deep learning applications in the context of software engineering. We also introduce background knowledge regarding Convolutional Neural Networks (ConvNets).

2.1. Code readability research

Prior work related to code readability can be broadly divided into two categories: (1) investigation of factors that can affect code readability; (2) derivation of general models to represent code readability. We describe the two directions in detail below.

Many earlier studies focused on exploration of the influential factors of code readability. Binkley et al. [10] conducted an empirical study to analyze the effects of identifier naming conventions (i.e., camelCase and under_score) on code readability.[2] Sasaki et al. [12] proposed a reordering technique to improve code readability by shortening the distance between the definition of a variable and its reference. Lee et al. [13] tested whether significant violations of coding conventions affected the readability of developed codes. Wang et al. [14] proposed an automatic blank line insertion algorithm to separate meaningful blocks to increase code readability.

Alternatively, a few studies attempted to model code readability using combinations of surface-level features. Aggarwal et al. [15] estimated the readability of source code by the ratio of all lines to comment lines in a study of software maintainability. Analogous to the Flesch Reading Ease Score (a widely used readability metric for English text) [16], Börstler et al. [17] proposed SRES (Software Readability Ease Score) to judge code readability according to the average length of lexemes and the average number of words per statement/block. Buse and Weimer [5] used data collected from 120 human annotators to construct the first model of code readability based on a set of local code features (e.g., the number of identifiers). Building on this study, Dorn [8] derived a generalizable formal model of code readability with additional visual, spatial, and linguistic features. Using the same dataset from [5], Posnett et al. [7] presented a simple, intuitive theory of code readability that relied on two main measures: size and code entropy. More recently, Scalabrino et al. [9] proposed a set of textual features based on source code lexicon analysis to complement the work of Buse and Weimer.

This study belongs to the latter category. As discussed above, most studies use generic, handcrafted features for code readability classification. The manual feature engineering process is neither effective nor efficient. To address this problem, we introduce a deep learning-based approach that can learn complicated features automatically from the source code.

2.2. Deep learning in software engineering

Deep learning has been proven to be a very powerful method in a variety of fields such as image recognition [18,19] and natural language processing [20,21]. It has also attracted considerable attention from researchers and industrial practitioners in the software engineering community. Successful applications include effort estimation [22], defect prediction [23], bug localization [24], code clone detection [25,26], and API recommendation [27].

Dam et al. [28] presented a vision for DeepSoft, a generic framework built upon Long Short-Term Memory for modeling software and its development process. Wang et al. [23] leveraged Deep Belief Networks to automatically learn semantic features of programs to improve defect prediction. Choetkiertikul et al. [22] developed a prediction system for estimation of story points based on a novel combination of Long Short-Term Memory and Recurrent Highway Network. Gu et al. [27] proposed DeepAPI, a deep learning-based method to generate relevant API usage sequences for a given natural language query. White et al. [25,26] introduced learning-based detection techniques to represent code fragments for code clone detection using Recurrent Neural Networks and Recursive Neural Networks. Lam et al. [24] constructed HyLoc, a model that combines the revised Vector Space Model (an advanced IR technique) with Deep Neural Networks to recommend potentially buggy files for a bug report.

To the best of our knowledge, deep learning has never been used to improve code readability studies. Inspired by the aforementioned work, we explore the possibility and feasibility of applying Convolutional Neural Networks to the field of code readability classification.

2.3. Convolutional neural network

Convolutional Neural Networks (ConvNets) [29,30] are special kinds of Neural Networks with biologically inspired structures. As shown in Fig. 1, a typical ConvNet is composed of a succession of convolutional, pooling, and fully-connected layers:[3]

2.3.1. Convolutional layer

A convolution is often interpreted as a filter. By applying a convolution operation followed by a nonlinearity to the input, we can obtain a feature map (the plane in Fig. 1). Specifically, the output y at lth layer is given by:

[1] https://github.com/CityU-QingMi/DeepCRM.

[2] The results showed that camelCase was easier to read than under_score variables [10]. However, an eye-tracking replication of Binkley et al.'s study indicated otherwise [11].

[3] The ConvNet in Fig. 1 is actually LeNet-5 [19,31], which is designed for recognition of handwritten and machine-printed characters.

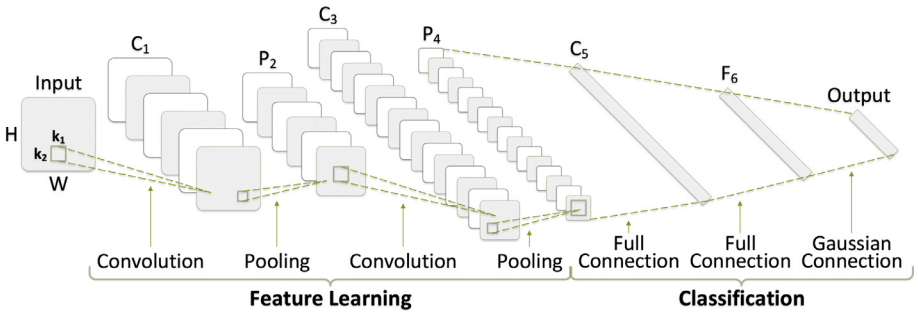

Fig. 1. Architecture of a typical ConvNet.

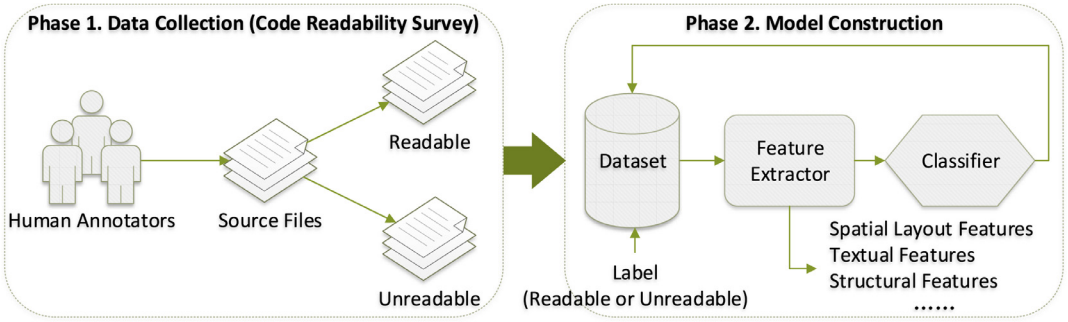

Fig. 2. Workflow of code readability research.

$$x_{i,j}^{l} = \sum_{m}\sum_{n} w_{m,n}^{l} y_{i+m,j+n}^{l-1} + b^{l} \tag{1}$$

$$y_{i,j}^{l} = \sigma(x_{i,j}^{l}) \tag{2}$$

where input x is of dimension $W \times H$ (i.e., *width* $\times$ *height* of input x, see Fig. 1) and has i by j as the iterators; filter w is of dimension $k_1 \times k_2$ (i.e., *width* $\times$ *height* of filter w, see Fig. 1) and has m by n as the iterators; b^l denotes the bias unit at l^{th} layer; and $\sigma(\cdot)$ denotes a nonlinear activation function such as Sigmoid ($\frac{1}{1+e^{-x}}$), Tanh ($\frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$), or ReLU ($max(0, x)$). The output feature maps serve as the input to the next layer.

2.3.2. Pooling layer

After a convolutional layer, a pooling layer is usually added to perform down-sampling by concentrating the input over a certain area into a single value. The process can help relieve the computational load because it reduces the number of parameters in the network. Various techniques can be used in pooling layers, including max-pooling and mean-pooling.

2.3.3. Fully-connected layer

Fully-connected layers are allocated after a couple of convolutional and pooling layers. Each neuron in fully-connected layers connects to all activations in the previous layer.

From another perspective, a typical ConvNet consists of a feature learning network (including one or more convolutional and pooling layer pairs) and a classification network (including one or more fully-connected layers) [32]. The former network extracts features automatically from the input data, whereas the latter network generates output based on the extracted features. The system parameters (i.e., weights w and biases b) are adjusted via the training process.

3. Motivation

In this section, we describe the motivation for this study and the rationale behind the use of Convolutional Neural Networks (ConvNets).

According to the definition, code readability is essentially an intuitive concept that is claimed to connect with various factors such as the structural pattern (e.g., nested loops and recursive functions) and the spatial layout (e.g., blank lines and indentations) [33,34]. Because a source code that is considered readable by one person may not be considered so by another, a large-scale survey involving multiple human annotators is necessary to determine whether a source code is readable or unreadable. Therefore, the first phase of modern code readability studies is usually a survey process, followed by a second phase that aims to construct a classification model based on the dataset gathered during the first phase as the ground truth. The entire workflow is illustrated in Fig. 2.

As shown in Fig. 2, a common practice in the second phase is to handcraft readability-related features and then apply them as predictors to differentiate between readable and unreadable codes. For instance, Buse and Weimer [5] designed a set of layout features (e.g., the number of blank lines[4]), whereas Scalabrino et al. [9] attached greater importance to textual aspects (e.g., the readability of comments). However, this widely used feature engineering method has the following problems:

- **Ineffective**: First, the process can capture only partial information about the source code, which may not be adequate. Second, features that are usable for one dataset may not be usable for another. Third, redundancies and overlaps may exist among the manually designed features, which is likely to limit the model's performance.
- **Inefficient**: Handcrafting good features is difficult and time-consuming because it requires strong domain-specific knowledge [28]. Moreover, researchers and practitioners must examine the validity of all promising features separately or as a combination, which makes the manual approach expensive or even impossible.

Although programs usually contain statistical properties, they are difficult for humans to capture [35,36]. To address this issue, we turn to ConvNets. The reasons behind using ConvNets are as follows: (1) ConvNets include the feature extractor in the training process rather

[4] In fact, several recent studies suggested that simple blank lines are more important than comments to local judgments of code readability [5,14].

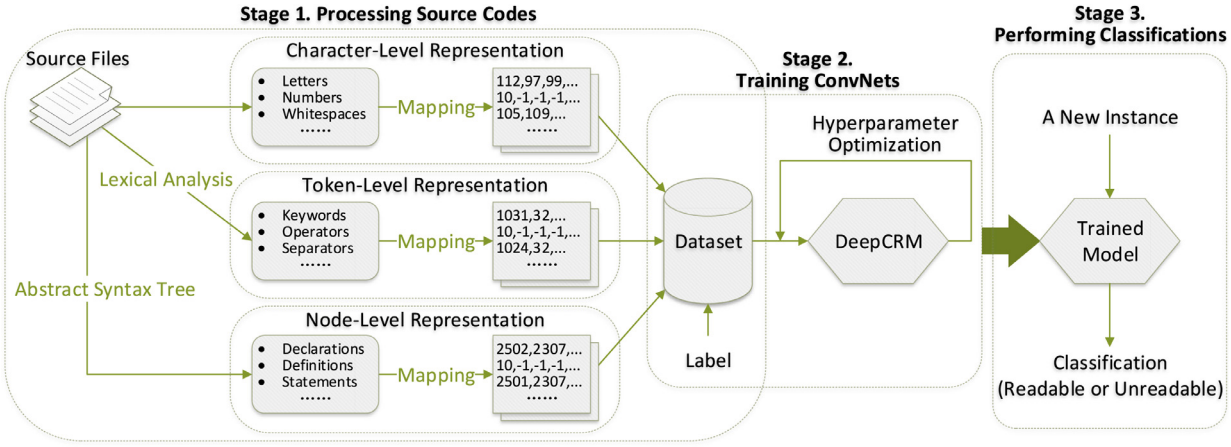

Fig. 3. General overview of the proposed approach.

than requiring it to be manually designed (as shown in Fig. 1) [32]. (2) ConvNets achieve remarkable results for tasks that deal with inputs in the form of multiple arrays (e.g., image recognition). Inspired by this, we treat a source code as a matrix (a two-dimensional array) of symbols. We expect that ConvNets can automatically form a deep understanding of what constitutes a readable code.

4. Proposed approach

We regard the code readability classification problem as a binary classification problem: given a piece of source code $x = \begin{pmatrix} x_{11} & \dots & x_{1n} \\ \vdots & \ddots & \vdots \\ x_{m1} & \dots & x_{mm} \end{pmatrix}$, where m denotes lines of code, n denotes the maximum line length, and x_{ij} denotes a code symbol.[5] We classify x into a *Readable* or *Unreadable* class.

As shown in Fig. 3, our approach consists of three major stages: (1) transformation of source codes into integer matrices as the input to ConvNets; (2) construction of multiple ConvNets and training with respect to learnable parameters; (3) determination of whether a new instance is readable or unreadable using the trained model. In this section, we describe the first two stages in detail.

4.1. Source code representation

To enable deep learning-based program analyses, we must formulate an appropriate strategy for source code representation. Because there is no definitive answer as to which factor(s) can affect code readability, our basic principle is to preserve as much as possible the original information about the source code. Specifically, we first parse the source codes into a set of symbols and then convert the symbols into integer matrices, which is the universal format that ConvNets can easily analyze and manipulate. Possible granularities of the code symbol (x_{ij}) are enumerated as follows [28,36]:[6]

- **Character-level representation**: Each character (e.g., a-z, A–Z, and 0–9) is treated as a symbol.
- **Token-level representation**: Each token (e.g., keywords and operators) is treated as a symbol.
- **Node-level representation**: Each node in the Abstract Syntax Tree (e.g., declarations and definitions) is treated as a symbol.

In this section, we detail our representation strategy specific to each granularity. Note that we focus solely on Java language because most previous studies constructed their code readability models based on Java programs (including but not limited to [5,7–9]). We remain consistent with their choice for comparison purposes.

4.1.1. Character-level representation

Like the pixels of an image, we treat a source code as a matrix of characters. Specifically, we transform letters (i.e., a–z and A–Z), numbers (i.e., 0–9), and marks (e.g., parentheses and braces) into their ASCII values. Given that the spatial layout of the source code may well affect its readability, we also preserve whitespaces (i.e., spaces, horizontal tabs, and line terminators) that have generally been ignored in prior deep learning studies. The final output takes the form of two-dimensional arrays. A simplified example is presented in Fig. 3. Note that we pad the arrays with a special integer (here, -1) because ConvNets require all inputs have the same length.

4.1.2. Token-level representation

Referring to the tokenization mechanism proposed by Basit et al. [37],[7] we too assign a unique integer to each token, including keywords (e.g., public and int), separators (e.g., semicolons and brackets), operators (e.g., subtraction and decrement), literals (e.g., true and null), and whitespaces. The advantage of this method is its simplicity and customizability. For instance, if we consider the difference among data types *byte, short, int*, and *long* to be insignificant for code readability classification, we can specify an integer to uniformly represent them.

It is noteworthy that some tokens such as variable names are user-defined and thus method-specific, which cannot be generalized to other code snippets. If we similarly assign every such token with a unique integer, it is likely to cause the undesired data sparseness. One possible approach to handle this situation is to introduce a suitable abstraction, such as replacing all variable names with a particular value.[8] Considering that useful information may be available in user-defined tokens (e.g., naming conventions [10,11]), we put forward an alternative method that transforms these tokens with their ASCII values in an attempt to preserve as much information as possible.

[5] The code symbol acts as the basic unit of the source code, which can be a character, a keyword, a function, etc. Further details are provided in Section 4.1.

[6] It is notable that we do not consider statement-, function-, or even higher-level representations, primarily because they are too abstract and may well lose useful information for code readability classification.

[7] The tokenization mechanism [37] is proposed for code clone detection. Because we are considering a different goal, we have made some necessary changes.

[8] A similar approach was adopted by Wang et al. [23].

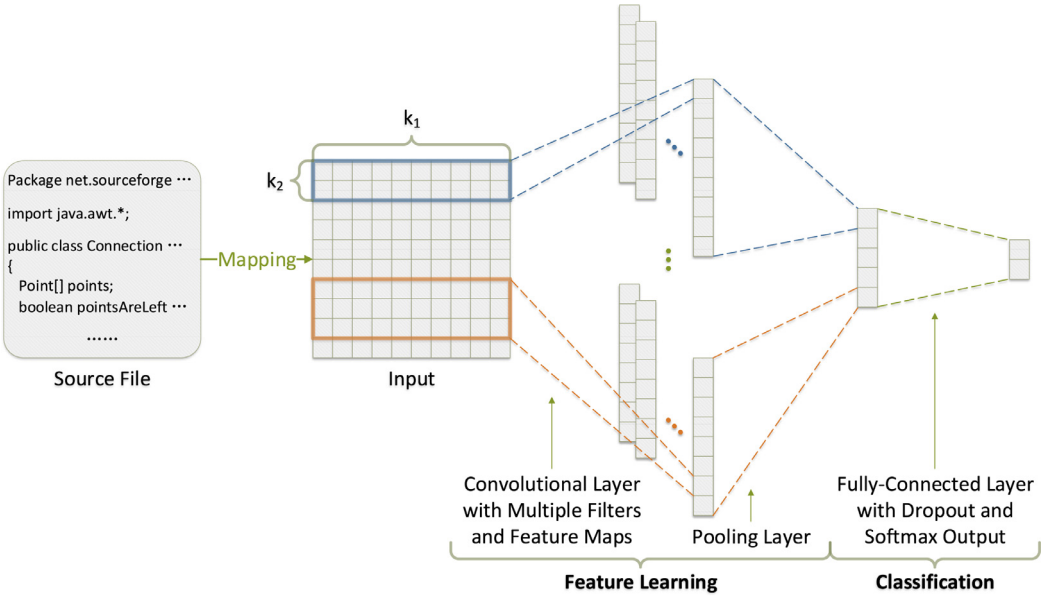

Fig. 4. Architecture of separate ConvNets.

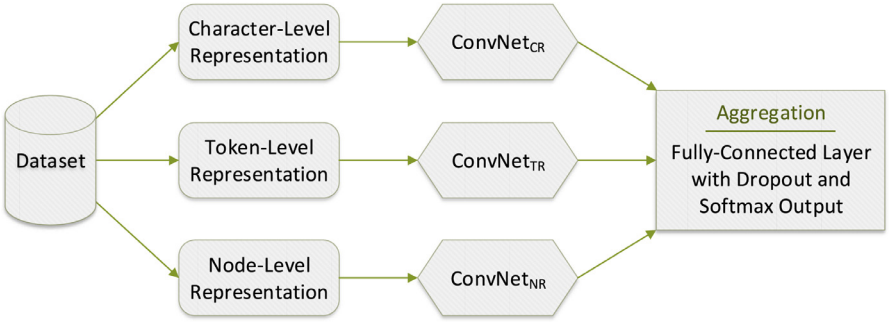

Fig. 5. Ensemble neural network architecture.

4.1.3. Node-level representation

Programs usually contain well-defined syntax that can assist in detecting certain patterns [23,38]. To capture the syntax information, we use the Abstract Syntax Tree (AST), which is an intermediate tree-like representation that allows us to access the syntactic structure of the source code. Specifically, we traverse each AST and encode every kind of node with a special integer. Note that ASTs omit certain elements in programs and thus enable the suppression of some inessential classes (e.g., the *byte, short, int*, and *long* examples discussed in the previous section). This is actually the most commonly used representation level in deep learning-based program analyses, including defect prediction [23] and code clone detection [26].

4.2. Neural network architecture

Corresponding to each granularity, we construct three separate ConvNets with identical architectures. For simplicity, we denote them as $ConvNet_{CR}$ (CR stands for Character-Level Representation), $ConvNet_{TR}$ (TR stands for Token-Level Representation), and $ConvNet_{NR}$ (NR stands for Node-Level Representation), respectively. The objective is to have multiple ConvNets that are skillful, but from different perspectives. In each ConvNet, the feature learning network contains a convolutional layer with a bank of filters, followed by the ReLU function[9] and a max-pooling layer. The classification network is composed of a fully-connected layer leading to a two-way Softmax classifier. The details are illustrated in Fig. 4.

As we discussed in Section 3, code readability is essentially an intuitive concept. The best way to determine whether a source code is readable is to consult with a number of domain experts rather than depending on one individual's judgment. To simulate this process, we propose the combination of multiple ConvNets into an ensemble model. The intuition is that the collective wisdom often produces a better solution [40]. Accordingly, we aggregate $ConvNet_{CR}$, $ConvNet_{TR}$, and $ConvNet_{NR}$ with adjustable weights. We denote the resulting model as DeepCRM (Deep Learning-Based Code Readability Model). The main novelty of our approach is the inclusion of the ensemble architecture, which is shown in Fig. 5.

4.3. Details of configuration and training

The proposed ConvNets are implemented in Python using TensorFlow.[10] We train them in a supervised manner with the Adam optimization algorithm [41], which is highly recommended as the optimization method for deep learning applications [42]. The learning rate is initialized as 0.001 and adapted during training. The goal is to minimize the network's loss (or misclassification error) with respect to adjustable parameters (i.e., weights[11] and biases). Here we adopt the

[9] The ReLU function (Rectified Linear Unit [39]) is widely used in recent applications because it can yield comparable results but converge several times faster than their equivalents [18].

[10] https://www.tensorflow.org.

[11] Analogous to most ConvNets, we use a weight-sharing strategy to reduce the number of free parameters.

widely used cross-entropy loss [43] for our classification problem:

$$L = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M} t_{i,j}\log y_{i,j} \quad (3)$$

where $N = 50$ is the number of training instances used in one iteration (i.e., the batch size), $M = 2$ is the number of classes (i.e., *Readable* and *Unreadable*), t is the ground truth, and y is the estimated class probability. The training data are randomly shuffled before they are sent to the network. To prevent over-fitting, a dropout with probability 0.5 is applied.

In addition to these configurations, we must fine-tune two hyperparameters: the filter size and the number of feature maps. The details are discussed in Section 6.

5. Experimental setup

To validate the effectiveness and efficiency of our approach, we design several experiments. This section mainly describes the experimental setups. Specifically, we first outline five research questions and the corresponding motivations. We then detail the construction process of the dataset used in this study. Finally, the evaluation metrics are briefly introduced.

5.1. Research questions

The goal of this study is to explore the potentiality of deep learning techniques in the task of code readability classification, with the purpose of improving the state-of-the-art. To evaluate whether deep learning is beneficial, we conduct a series of experimental evaluations. Specifically, we aim to answer the following research questions:

RQ1: Do different parameter settings affect the model performance?

Motivation: We aim to explore the sensitivity of each ConvNet ($ConvNet_{CR}$, $ConvNet_{TR}$, and $ConvNet_{NR}$) with respect to different parameter settings. Because tuning is normally time-consuming, we expect our ConvNets to be relatively stable.

Approach: Two hyperparameters require fine-tuning: the filter size and the number of feature maps. To answer this research question, we vary the values of these two hyperparameters and evaluate their effects on the model performance in terms of accuracy.

RQ2: What is the benefit of the ensemble architecture?

Motivation: To improve model generality, we propose a novel architecture that combines $ConvNet_{CR}$, $ConvNet_{TR}$, and $ConvNet_{NR}$. This research question involves investigation of the efficacy of this ensemble method.

Approach: We evaluate the performance of DeepCRM against the fine-tuned $ConvNet_{CR}$, $ConvNet_{TR}$, and $ConvNet_{NR}$ using accuracy and f-measure.

RQ3: How accurate is DeepCRM in code readability classification when compared to traditional models?

Motivation: This study is the first to apply ConvNets to the field of code readability classification. It is important to know whether our approach can outperform state-of-the-art code readability models.

Approach: To answer this research question, we compare DeepCRM with five competitors that are state-of-the-art models for code readability classification.

- **Buse and Weimer's model** [5]: Buse and Weimer proposed a set of local code features to represent code readability. They claimed that the model performed just as good as a human.
- **Posnett et al.'s model** [7]: Improving upon the model of Buse and Weimer, Posnett et al. presented a simpler readability model: $z = 8.87 - 0.033V + 0.40Lines - 1.5Entropy$.
- **Dorn's model** [8]: Making use of visual, spatial, and linguistic features, Dorn aimed to construct a universal model for code readability.
- **Scalabrino et al.'s model** [9]: To improve the performance of the existing models, Scalabrino et al. further introduced a set of textual features targeted on identifiers and comments.
- **A comprehensive model** [9]: A code readability model that includes each of the features mentioned above.

We denote the aforementioned models as M_{Buse}, $M_{Posnett}$, M_{Dorn}, $M_{Scalabrino}$, and M_{All} successively. For a fair and easy comparison, these models are all trained using logistic regression as the underlying classifier [9]. Note that we do not replicate M_{Buse}, $M_{Posnett}$, M_{Dorn}, $M_{Scalabrino}$, and M_{All}. Rather, we use results reported by previous studies [5,7–9] to address this RQ.

RQ4: What is the time and space cost of training the proposed ConvNets?

Motivation: Despite the advantages that deep learning may offer, its training process tends to be computationally expensive. In this research question, we aim to provide researchers and practitioners with a clear understanding.

Approach: We explicitly record the time and space cost of $ConvNet_{CR}$, $ConvNet_{TR}$, $ConvNet_{NR}$, and DeepCRM during the training process.

In summary, we begin by choosing the optimal hyperparameters for each ConvNet (RQ1). We then combine them into an ensemble model and evaluate its efficacy (RQ2). After that, we compare the performance of our approach with those of the state-of-the-art models (RQ3). Finally, we report the time and space cost incurred during the training process (RQ4).

5.2. Dataset construction

Our data are extracted from prior code readability studies and open source software projects. In this section, we detail our data collection process.

We first gather a set of code snippets from previous studies, namely, D_{Buse}, D_{Dorn}, and $D_{Scalabrino}$, as shown in the first part of Table 1. In D_{Buse}, D_{Dorn}, and $D_{Scalabrino}$, each code snippet is evaluated by multiple human annotators on a five-point Likert scale [44] ranging from 1 (very unreadable) to 5 (very readable). We aggregate these subjective ratings by taking the mean to represent the readability degree of each code snippet. Following a similar approach to that of Lee et al. [13], we construct two representative groups for each dataset: the *Readable* group (i.e., the top 25% code snippets with high readability scores) and the *Unreadable* group (i.e., the bottom 25% code snippets with low readability scores), whereas the code snippets with neutral or ambiguous judgments (i.e., the middle 50% samples) are excluded.

Only a few human-annotated code snippets are available in the literature (see Table 1 for details), which may not be adequate to train the proposed ConvNets. We must extend our dataset with more samples. To achieve this, we first fully download ten Java projects from various sources (e.g., Github[12] and SourceForge[13]) across multiple application domains (e.g., web server and testing tool), this is to increase the generalizability of our findings. Similar to the approach used in the prior study [9], we extract from the selected projects all complete code entities (i.e., methods) whose size is between 10 and 50 lines. Afterwards, we calculate the MD5 hash for each file to remove duplicates. As shown in the second part of Table 1, we obtain a total of 51831 code snippets. The next step is to classify them as readable or unreadable. According to the definition of code readability, the most accurate method to judge whether a piece of source code is readable is to conduct a large-scale survey involving as many domain experts as possible.

[12] https://github.com.
[13] https://sourceforge.net.

Table 1
Statistical summary of the dataset.

Category	Dataset	Description	Source	# of Code Snippets		Lines of Code[b]	
				Total	Selected	Mean	SD
D_{CRS} (Labeled by human annotators)	D_{Buse}	Provided by Buse and Weimer [5]	SourceForge	100	50	7.80	2.47
	D_{Dorn}[a]	Provided by Dorn [8]	SourceForge	360	60	30.81	16.66
	$D_{Scalabrino}$	Provided by Scalabrino et al. [9]	SourceForge	200	100	26.61	10.38
D_{OSS} (Labeled by automated tools)	$D_{Hibernate}$	Object Relational Mapping Tool	Github	12,254	6127	23.27	12.37
	$D_{HyperSQL}$	Relational Database Engine	SourceForge	4017	2009	23.50	12.99
	D_{Jetty}	Web Container and Clients	Github	7661	3831	23.48	12.68
	D_{JUnit}	Unit Testing Framework	Github	1128	564	17.44	7.94
	D_{Log4j}	Logging Utility	Apache	3052	1526	20.16	10.75
	D_{Maven}	Build Automation Tool	Github	1901	951	21.75	11.70
	D_{Quartz}	Job Scheduler	Github	1065	533	22.87	11.22
	D_{Roller}	Blog Server	Apache	2666	1333	25.24	12.09
	D_{Spring}	J2EE Framework	Github	14,345	7173	20.27	10.61
	D_{Struts}	Web Application Framework	Github	3742	1871	21.45	11.78

[a] D_{Dorn} contains code snippets written in Java, Python, and CUDA languages. We concern only the Java samples.
[b] All lines in the text of source code are counted, including blank and comment lines.

The process is inevitably labor-intensive and time-consuming. Considering that we have more than 50,000 code snippets to evaluate, the human assessment approach is practically impossible. We thus propose an alternative method to help classify these samples.

Lee et al. [13] found that source codes that significantly violate programming guidelines/conventions/styles [45,46] are less readable than those that do not. Buse and Weimer [5] showed that code readability correlates strongly with external notions of software quality (e.g., defect density and code churn). According to their conclusions, we rely on the total rule violations to obtain a rough estimate of whether a source code is readable. Specifically, we count the number of rule violations for each file making use of automated tools PMD[14] and CheckStyle[15]. PMD examines for common programming flaws like unused variables and empty catch blocks, whereas CheckStyle detects whether a Java code complies with a programming standard. For each project, we consider the top 25% of code snippets with more rule violations as the *Unreadable* group and the bottom 25% of code snippets with fewer rule violations as the *Readable* group. The average violations for the two comparison groups are 84.84 and 30.17, respectively. In this way, we obtain an additional 25,000 samples to adequately train the proposed ConvNets. A statistical summary of the new dataset (D_{OSS}) is presented in Table 1. Because D_{OSS} has not been validated by human annotators, it will be used only for tentative explorations (see Section 7.1 for details).

Note that this method is only applicable for dataset construction (i.e., for identification of two groups of *relatively* readable/unreadable code snippets). It is unreasonable to use it for code readability classification. In other words, when given a new instance, we cannot determine its readability degree based merely on its number of rule violations.[16]

5.3. Evaluation metrics

To measure classification results, we use accuracy and f-measure as the evaluation metrics, which have commonly been used in previous code readability studies [5,8,9]. The higher the metric value, the better the model performance.

$$Accuracy = \frac{True\ Positive + True\ Negative}{True\ Positive + False\ Positive + True\ Negative + False\ Negative} \quad (4)$$

$$F - Measure = \frac{2*Precision*Recall}{Precision + Recall} \quad (5)$$

where *True Positive* is the number of readable instances that are correctly classified as readable, *True Negative* is the number of unreadable instances that are correctly classified as unreadable, *False Positive* is the number of unreadable instances that are wrongly classified as readable, *False Negative* is the number of readable instances that are wrongly classified as unreadable, *Precision* refers to $\frac{True\ Positive}{True\ Positive + False\ Positive}$, and *Recall* refers to $\frac{True\ Positive}{True\ Positive + False\ Negative}$. In summary, accuracy shows the ratio of correctly classified instances, whereas f-measure shows the weighted harmonic mean of *Precision* and *Recall*.

6. Results

In this section, we present the experimental results according to each research question. For simplicity, we denote the previous datasets (see details in the first part of Table 1) as D_{CRS} (CRS stands for Code Readability Survey) and the additional datasets (see details in the second part of Table 1) as D_{OSS} (OSS stands for Open Source Software Projects). Note that in RQ1, RQ2, and RQ3, our evaluations use only D_{CRS} for a fair comparison, whereas both D_{CRS} and D_{OSS} are used in RQ4. To help mitigate the problem of over-fitting, we perform 10-fold cross-validation in all RQs.

RQ1: Do different parameter settings affect the model performance?

In this RQ, we detail the tuning process with respect to each hyperparameter.

- **Filter size**

The filter size corresponds to the identifiers $k_1 \times k_2$ (i.e., *width* × *height*) in Fig. 4. To investigate the effect of the filter size, we first fix the number of feature maps as a constant (here, 100). Considering that code readability is context sensitive (just like natural language), we set the width k_1 as the maximum line length, which varies by each ConvNet. Following the approach given by Zhang and Wallace [47], we then fine-tune k_2.

We begin by identifying the best single filter size. Specifically, we

[14] https://pmd.github.io.
[15] http://checkstyle.sourceforge.net.
[16] Code readability is essentially a subjective concept [5] that has been found to be associated with various factors such as comment lines [15], naming conventions [10,11], defect density [8], and rule violations [13]. However, statistical associations do not imply causation. We cannot equate code readability with any of these factors [4,33].

Table 2
Effect of the single filter size.

Filter Size	$ConvNet_{CR}$	$ConvNet_{TR}$	$ConvNet_{NR}$
1	85.5%	77.3%	67.3%
2	**85.8%**	**84.0%**	58.8%
3	79.8%	76.0%	58.5%
4	83.0%	82.3%	64.0%
5	78.3%	80.5%	63.8%
6	75.8%	79.0%	66.5%
7	74.5%	78.0%	69.8%
8	76.0%	78.3%	67.3%
9	76.8%	80.5%	**74.3%**
10	76.5%	83.5%	72.5%

Table 3
Effect of the multiple filter size.[a]

$ConvNet_{CR}$		$ConvNet_{TR}$		$ConvNet_{NR}$	
Filter size	Accuracy	Filter size	Accuracy	Filter size	Accuracy
–	–	–	–	**(7,8,9)**	69.5%
(1,2,3)	86.5%	(1,2,3)	75.5%	(8,9,10)	**75.5%**
(2,3,4)	81.5%	(2,3,4)	80.0%	(9,10,11)	74.5%
(2,2,2)	**88.0%**	(2,2,2)	**81.0%**	(9,9,9)	69.5%

[a] To have ConvNets with identical architectures, we consider only the set of size three.

first select a finite set of values for k_2 and then evaluate each ConvNet's performance regarding mean accuracy. Note that we bias our selection to small values to capture low-level features and because the average LOC in D_{CRS} is not very large (see Table 1 for details). The results are presented in Table 2. The best single filter sizes for $ConvNet_{CR}$, $ConvNet_{TR}$, and $ConvNet_{NR}$ are 2, 2, and 9, respectively.

Next, we explore the effect of multiple filter sizes. As advised by Zhang and Wallace [47], we choose a set of values near the best single filter size and then perform the same evaluations as above. Table 3 demonstrates our selections and the corresponding results (the best results are highlighted in bold). From Table 3, we decide to use (2,2,2), (2,2,2), and (8,9,10) as the filter sizes for $ConvNet_{CR}$, $ConvNet_{TR}$, and $ConvNet_{NR}$, respectively.

- **Number of feature maps**

Based on the selected filter size, we now fine-tune the number of feature maps. According to Zhang and Wallace [47], the value to be evaluated should not exceed 600. A much longer time is required to train the model if the number of feature maps grows beyond 600, but the model performance improves only a little, so the process is not cost-effective. In addition, the model performance may even worsen due to over-fitting. We thus comply with their suggestions and consider the following values: {10, 50, 100, 200, 400, 600}. The mean accuracy is plotted in Fig. 6 regarding each ConvNet. It can be observed that 100 is the optimal number of feature maps for $ConvNet_{CR}$, $ConvNet_{TR}$, and $ConvNet_{NR}$.

Taken together with Tables 2 and 3, and Fig. 6, we notice that the model performance varies on a small scale (less than 10 percentage points), which indicates that our ConvNets are somewhat robust to different parameter settings.

RQ2: What is the benefit of the ensemble architecture?

We propose an ensemble architecture that integrates $ConvNet_{CR}$, $ConvNet_{TR}$, and $ConvNet_{NR}$. Because each ConvNet is constructed with a different view (see details in Figs. 3 and 5), we expect the resulting model to be more generic and thus help improve the classification performance.

We present the evaluation results in the first row of Table 4. It can be observed that DeepCRM provides the best performance, with 83.8% accuracy and 83.5% f-measure, which validates the effectiveness of our ensemble method. The last two rows of Table 4 are used to display the results of an exploratory experiment on D_{OSS}. Further details are provided in Section 7.1.

Note that the evaluations are conducted on a hold-out test set from D_{CRS}, which has not been used for hyperparameter optimization (RQ1). Therefore, the accuracy obtained here is generally not equal to that in RQ1.

RQ3: How accurate is DeepCRM in code readability classification when compared to traditional models?

The aim of this RQ is to determine the extent to which DeepCRM can predict human readability judgments compared to previously reported models (i.e., M_{Buse}, $M_{Posnett}$, M_{Dorn}, $M_{Scalabrino}$, and M_{All}). Because we use the same dataset (D_{CRS}) as previous code readability studies [5,7–9], we directly present the results reported by them to address this RQ. As shown in Fig. 7, DeepCRM outperforms all competitors (the improvement in accuracy ranges from 2.4% with respect to M_{All} to 17.2% with respect to $M_{Posnett}$), which suggests the promising future of applying deep learning techniques to the field of code readability classification.

RQ4: What is the time and space cost of training the proposed ConvNets?

Training Deep Neural Networks usually consumes a considerable amount of resources. However, most deep learning studies do not report their efforts [48]. To provide a reference for future studies, we explicitly recorded our time and memory space cost during the training process. The results are given in Table 5.

All of our experiments are conducted on a server with Intel Xeon CPU E5-4620 2.20 GHz (32 Cores), 128GB RAM. As shown in Table 5, the memory space cost varies within a small range (from 11.10 MB to 19.05 MB), mainly because the number of training instances used in one iteration (i.e., the batch size) is fixed. As for the time cost, training on D_{CRS} requires approximately 0.22 –0.47 s for each epoch, whereas the value increases to 29.74–71.46 s if we further introduce D_{OSS}, the ratio reaches 129.97–163.20.

Admittedly, our approach has the downside of being time-consuming during the training process. However, once training is complete, our approach can be used directly at an acceptably low time cost (the prediction time is less than 0.001 s for a new instance), which confirms its practical applicability.

7. Discussion

In this section, we investigate how much improvement can be expected from a large training set. Then we discuss the advantages and disadvantages of applying deep learning techniques to the field of code readability classification.

7.1. An exploratory experiment on D_{OSS}

Convolutional Neural Networks (ConvNets) risk over-fitting if the training set is not sufficiently large. To adequately train the proposed ConvNets, we also collect more than 25,000 code snippets from open source software projects (see details in the second part of Table 1). In this exploratory experiment, we evaluate whether training on the new dataset (D_{OSS}) can help improve the model performance.

We train $ConvNet_{CR}$, $ConvNet_{TR}$, $ConvNet_{NR}$, and DeepCRM on D_{OSS}, and compare their performances with those trained using D_{CRS} (the old dataset provided by previous code readability studies [5,8,9]). The evaluation results are presented in the last two rows of Table 4. As

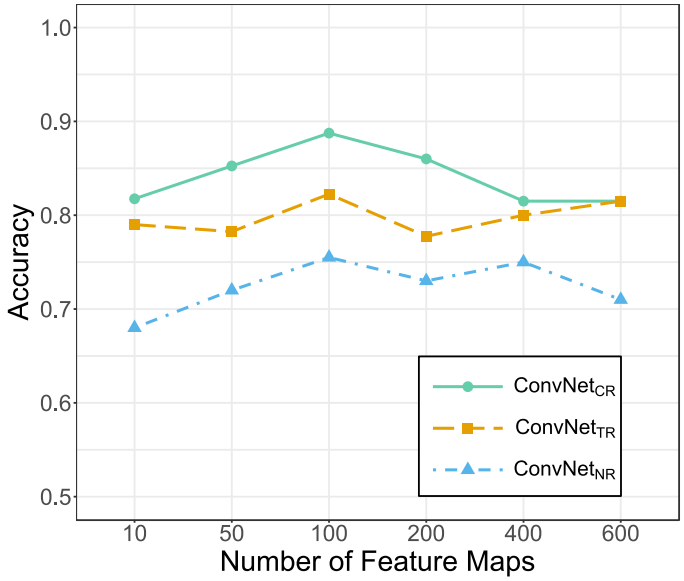

Fig. 6. Effect of the number of feature maps.

Table 4
Evaluation of the ensemble architecture.

Dataset	$ConvNet_{CR}$		$ConvNet_{TR}$		$ConvNet_{NR}$		DeepCRM	
	Accuracy	F-measure	Accuracy	F-measure	Accuracy	F-measure	Accuracy	F-measure
D_{CRS}	82.3%	81.0%	77.3%	75.3%	71.8%	70.9%	**83.8%**	**83.5%**
D_{CRS} + D_{OSS}	88.9%	89.4%	74.3%	77.8%	51.5%	54.8%	**91.5%**	**91.7%**
Improvement	8.0%	10.4%	− 3.9%	3.3%	− 28.3%	− 22.7%	9.2%	9.8%

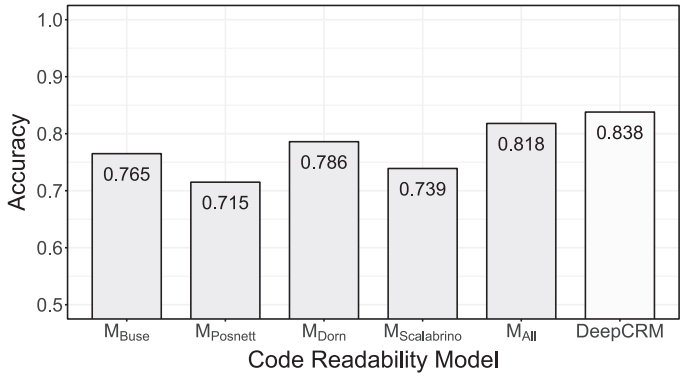

Fig. 7. Comparison between DeepCRM and the traditional code readability models.

Table 5
Average time and space cost per epoch.[a]

Dataset	$ConvNet_{CR}$		$ConvNet_{TR}$		$ConvNet_{NR}$		DeepCRM	
	Time (Second)	Memory (MB)	Time (Second)	Memory (MB)	Time (Second)	Memory (MB)	Time (Second)	Memory (MB)
D_{CRS}	0.22	11.74	0.20	11.10	0.23	11.52	0.47	12.14
D_{CRS} + D_{OSS}	32.40	15.50	32.41	13.75	29.74	14.70	71.46	19.05
Ratio	144.85	1.32	163.20	1.24	129.97	1.28	152.14	1.57

[a] An epoch is an iteration through the entire training set.

shown, $ConvNet_{CR}$ and DeepCRM have better performance on D_{OSS} than on D_{CRS}. The improvement in accuracy is 8.0% and 9.2%, respectively. $ConvNet_{TR}$ yields comparable results, whereas the performance of $ConvNet_{NR}$ becomes much worse, probably because $ConvNet_{NR}$ is trapped in a local optimum.[17] We will further investigate this problem in a future study.

In this exploratory experiment, we examine the benefits of adopting a large training set (D_{OSS}). Note that because D_{OSS} has not been validated by human annotators, the current results should be considered preliminary.

7.2. Advantages and disadvantages of deep learning

The experimental results show that the proposed ConvNets perform better than the traditional models [5,7–9] for code readability classification (the improvement in accuracy ranges from 2.4% to 17.2%). In this section, we discuss why our approach achieves the best performance.

- First, we eliminated the need for manual feature engineering. Inspired by image recognition, we treat a source code as a matrix of symbols and leverage ConvNets to automatically learn features directly from the input data. Our approach has the advantage of requiring no human intervention and thus can effectively avoid personal biases and neglects of certain features.
- Second, we proposed a representation strategy (with different granularities) to preserve the source code's original information. Despite little tuning of the hyperparameters, a simple ConvNet can achieve fairly good results based on the resulting integer matrices (RQ1 and RQ2).
- Third, we integrated multiple ConvNets with an ensemble architecture to improve model generality and applicability. The experimental results show that the ensemble model can surpass the state-of-the-art, achieving 83.8% accuracy and 83.5% f-measure (RQ2 and RQ3).
- Finally, a new dataset with more than 25,000 code snippets was used to fully train the proposed ConvNets. We observe further improvements by about 9.2% in accuracy and 9.8% in f-measure (The exploratory experiment presented in Section 7.1).

By eliminating the need for manual feature engineering (one of the most time-consuming parts of traditional machine learning), our approach provides improved performance, confirming the feasibility of deep learning techniques for code readability classification. Although deep learning is an important area of research with many promising applications, several drawbacks and limitations accompany its use.

- **Data dependency**: Deep learning requires a large corpus of data with which to work, which may not always be available. Without large, well-maintained datasets, deep learning may only be able to yield results comparable to those of traditional machine learning (RQ3).
- **Time cost**: Because deep learning inherently involves a large amount of matrix multiplication operations (e.g., Eqs. (1) and (3)), the training of Deep Neural Networks is usually computationally expensive (RQ4). This leads to the next disadvantage.
- **Hardware dependency**: Unlike traditional machine learning, deep learning is quite resource-demanding and depends heavily on high-performance machines (preferably with powerful GPUs).
- **Model interpretability**: Deep learning is in principle a black-box approach. Unlike traditional machine learning (e.g., decision trees), deep learning is incapable of explaining why a certain decision has been reached. In other words, we cannot map decisions back to individual features.

Although deep learning has achieved impressive (and often state-of-the-art) results in multiple domains, it is certainly not flawless. The caveat is that deep learning should not be applied to all possible applications without weighing its pros and cons.

8. Threats to validity

In this section, we describe the potential threats to the validity of our results.

- **Data quality**

The most obvious threat is the quality of the training data. Broadly, we have two data sources. A minor part of our dataset (approximately 0.4%) comes from previous code readability studies [5,8,9], and the rest is taken from open source software projects. We outline the possible threats subject to each data source.

To abstract people's perceptions of whether a source code is readable, the most frequently used method is to conduct a survey. During the survey process, participants are required to rate a number of code snippets based on their knowledge and experience. Considering that the process in itself is neither amusing nor rewarding, it is possible that the participants will feel little motivation to contribute, resulting in negative behaviors such as random responses [49], which can significantly affect the quality of the survey data. In addition, most participants involved in code readability surveys are students with less programming experience than industrial practitioners. Although the use of college students imposes some limits on data validity, it is not believed that the overall results are skewed. Moreover, prior work has validated their datasets with inter-rater reliability tests (i.e., Cronbach's alpha [50]). Considering the positive results (e.g., 0.96 for D_{Buse} and 0.98 for $D_{Scalabrino}$), we regard these datasets as the reliable oracle.

Given that the literature contains only a few human-annotated code snippets, which may not be sufficient to train the proposed ConvNets, we turn to open source Java projects. Although we sampled projects of various sizes and from various communities, they cannot represent all software projects. Additionally, we biased our selection to popular projects to ensure the size of the dataset. It is possible that other open source (or industrial) projects could yield different conclusions. Besides, we used automated tools to roughly classify the code snippets. Although our approach is defensible, further study is needed to justify the new dataset.

- **Sample size**

Another limitation is that when addressing RQ1, RQ2, and RQ3, we use only D_{CRS} rather than $D_{CRS} + D_{OSS}$ for two primary reasons. First, to achieve a fair comparison, our ConvNets must be trained on the same dataset as in the previous studies. Second, the quality of D_{CRS}, but not D_{OSS}, has been validated by human annotators.[18] However, the sample size of D_{CRS} is admittedly small, which may threaten the internal validity of our results. Given that the proposed ConvNets can provide consistently high and stable performance, we consider the use of D_{CRS} in RQ1, RQ2, and RQ3 to be acceptable and feasible.

9. Conclusions and future work

In this paper, we introduced a completely new method for code readability classification based on Convolutional Neural Networks (ConvNets). We first proposed a simple and customizable

[17] Another possibility is that node-level representation is still too abstract to preserve sufficient information for code readability classification.

[18] Accordingly, D_{OSS} has only been used in RQ4 for a tentative exploration.

representation strategy (with different granularities) to transform source codes into integer matrices as the input to ConvNets. We then implemented DeepCRM, a novel code readability model consisting of three separate ConvNets with identical architectures. Our approach is able to learn various levels of features automatically from the source code. To evaluate DeepCRM, we conducted a series of experiments. The results show that DeepCRM can outperform state-of-the-art models (i.e., the models of *Buse and Weimer* [5], *Posnett et al.* [7], *Dorn* [8], and *Scalabrino et al.* [9] and *A Comprehensive Model* [9]). The improvement in accuracy ranges from 2.4% to 17.2%.

Our work serves as the first step toward deep learning-based code readability classification. We hope that the promising results will interest and encourage more in-depth research in this new field. To enable critical or extended analyses, our dataset and source code are publicly available.[1]

There are several directions for future research. The top priority is to further improve the model performance.[19] To achieve this, we can fine-tune model hyperparameters or attempt a different model architecture. Given that the data quality can significantly affect the model quality, we also plan to construct a better training set with more reliable data (i.e., human-annotated code snippets), which can hopefully improve our results. Another option is to optimize the strategy used to represent the source code, which is one of the major challenges in deep learning-based program analyses.

In this study, we treat a code snippet (a separate file) as an atomic unit for code readability classification, which may be somewhat coarse-grained. We expect that in future our approach can precisely locate unreadable regions rather than forcing users to search through the entire file. This drawback actually applies to all existing code readability studies.

Currently, our approach allows only Java code. We intend to extend it to other programming languages. We consider the plan to be feasible because the only language-dependent part is the lexical analysis process presented in Fig. 3. All remaining steps are adaptable with slight adjustments.

In addition, existing studies (including our own) tend to classify a piece of source code as either readable or unreadable. However, readability is not necessarily a binary decision. In the future, we plan to investigate the possibility of measuring code readability in terms of continuous values to make the output more precise and interpretable (just like *Reading Grade Level* for natural languages [16]).

Another direction for future work is to apply the proposed ConvNets in a real-world situation. For instance, we can integrate our model into modern IDEs (Integrated Development Environments) or version control systems to enable developers to continually monitor the readability of their code.

Acknowledgments

This work is supported in part by the General Research Fund of the Research Grants Council of Hong Kong (no. 11208017), and the research funds of City University of Hong Kong (no. 7004683 and 9678149).

References

[1] B. Boehm, V.R. Basili, Software defect reduction top 10 list, Software Eng. 34 (1) (2007) 75.
[2] J.V. Farr, Systems Life Cycle Costing: Economic Analysis, Estimation, and Management, CRC Press, 2011.
[3] J. Borstler, B. Paech, The role of method chains and comments in software readability and comprehension-an experiment, IEEE Trans. Software Eng. 42 (9) (2016) 886–898, https://doi.org/10.1109/TSE.2016.2527791.
[4] X. Xia, L. Bao, D. Lo, Z. Xing, A. E. Hassan, S. Li, Measuring program comprehension: alarge-scale field study with professionals, IEEE Trans. Software Eng. 5589 (c) (2017). 1. doi:10.1109/TSE.2017.2734091 .
[5] R.P.L. Buse, W.R. Weimer, Learning a metric for code readability, IEEE Trans. Software Eng. 36 (4) (2010) 546–558, https://doi.org/10.1109/TSE.2009.70.
[6] R.P.L. Buse, T. Zimmermann, Information needs for software development analytics, 2012 34th International Conference on Software Engineering (ICSE), IEEE, 2012, pp. 987–996, https://doi.org/10.1109/ICSE.2012.6227122.
[7] D. Posnett, A. Hindle, P. Devanbu, A simpler model of software readability, Proceeding of the 8th Working Conference on Mining Software Repositories - MSR '11, 11 ACM Press, New York, USA, 2011, p. 73, https://doi.org/10.1145/1985441.1985454.
[8] J. Dorn, A General Software Readability Model, MCS, 2012 Thesis. Available from (http://www.cs.virginia.edu/~weimer/students/dorn-mcs-paper.pdf)
[9] S. Scalabrino, M. Linares-Vasquez, D. Poshyvanyk, R. Oliveto, Improving code readability models with textual features, 2016 IEEE 24th International Conference on Program Comprehension (ICPC), volume 2016-July, IEEE, 2016, pp. 1–10, https://doi.org/10.1109/ICPC.2016.7503707.
[10] D. Binkley, M. Davis, D. Lawrie, C. Morrell, To camelcase or under_score, 2009 IEEE 17th International Conference on Program Comprehension, IEEE, 2009, pp. 158–167, https://doi.org/10.1109/ICPC.2009.5090039.
[11] B. Sharif, J.I. Maletic, An eye tracking study on camelcase and under_score identifier styles, 2010 IEEE 18th International Conference on Program Comprehension, IEEE, 2010, pp. 196–205, https://doi.org/10.1109/ICPC.2010.41.
[12] Y. Sasaki, Y. Higo, S. Kusumoto, Reordering program statements for improving readability, 2013 17th European Conference on Software Maintenance and Reengineering, IEEE, 2013, pp. 361–364, https://doi.org/10.1109/CSMR.2013.50.
[13] T. Lee, J.B. Lee, H.P. In, A study of different coding styles affecting code readability, Int. J. Software Eng. Appl. 7 (5) (2013) 413–422, https://doi.org/10.14257/ijseia.2013.7.5.36.
[14] X. Wang, L. Pollock, K. Vijay-Shanker, Automatic segmentation of method code into meaningful blocks: design and evaluation, J. Software 26 (1) (2014) 27–49, https://doi.org/10.1002/smr.1581.
[15] K. Aggarwal, Y. Singh, J. Chhabra, An integrated measure of software maintainability, Annual Reliability and Maintainability Symposium. 2002 Proceedings (Cat. No.02CH37318), IEEE, 2002, pp. 235–241, https://doi.org/10.1109/RAMS.2002.981648.
[16] R. Flesch, A new readability yardstick. J.Appl.Psychol. 32 (3) (1948) 221.
[17] J. Börstler, M.E. Caspersen, M. Nordström, Beauty and the beast: on the readability of object-oriented example programs, Software Qual. J. 24 (2) (2016) 231–246, https://doi.org/10.1007/s11219-015-9267-5.
[18] A. Krizhevsky, I. Sutskever, G.E. Hinton, ImageNet classification with deep convolutional neural networks, Adv. Neural Inf. Process. Syst. (2012) 1–9, https://doi.org/10.1016/j.protcy.2014.09.007.
[19] C. Szegedy, Wei Liu, Yangqing Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, A. Rabinovich, Going deeper with convolutions, 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), IEEE, 2015, pp. 1–9, https://doi.org/10.1109/CVPR.2015.7298594.
[20] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, P. Kuksa, Natural language processing (almost) from scratch, J. Mach. Learn. Res. 12 (2011) 2493–2537 doi:10.1.1.231.4614 ..
[21] T. Mikolov, G. Corrado, K. Chen, J. Dean, Efficient estimation of word representations in vector space, Proceedings of the International Conference on Learning Representations (ICLR 2013), (2013), pp. 1–12, https://doi.org/10.1162/153244303322533223.
[22] M. Choetkiertikul, H.K. Dam, T. Tran, T. Pham, A. Ghose, T. Menzies, A deep learning model for estimating story points, arXiv cs.SE 9 (2016) 00489. arXiv:1609.00489 .
[23] S. Wang, T. Liu, L. Tan, Automatically learning semantic features for defect prediction, Proceedings of the 38th International Conference on Software Engineering - ICSE '16, volume 14-22-May-, ACM Press, New York, USA, 2016, pp. 297–308, https://doi.org/10.1145/2884781.2884804.
[24] A.N. Lam, A.T. Nguyen, H.A. Nguyen, T.N. Nguyen, Combining deep learning with information retrieval to localize buggy files for bug reports (N), 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2015, pp. 476–481, https://doi.org/10.1109/ASE.2015.73.
[25] M. White, C. Vendome, M. Linares-Vasquez, D. Poshyvanyk, Toward deep learning software repositories, 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories, IEEE, 2015, pp. 334–345, https://doi.org/10.1109/MSR.2015.38.
[26] M. White, M. Tufano, C. Vendome, D. Poshyvanyk, Deep learning code fragments for code clone detection, Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering - ASE 2016, ACM Press, New York, USA, 2016, pp. 87–98, https://doi.org/10.1145/2970276.2970326.
[27] X. Gu, H. Zhang, D. Zhang, S. Kim, Deep API learning, Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016, ACM Press, New York, USA, 2016, pp. 631–642, https://doi.org/10.1145/2950290.2950334.
[28] H.K. Dam, T. Tran, J. Grundy, A. Ghose, DeepSoft: a vision for a deep model of software, Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016, 1691 ACM Press, New York, USA, 2016, pp. 944–947, https://doi.org/10.1145/2950290.2983985.
[29] A. Carruthers, J. Carruthers, Introduction, Dermatol. Surg. 39 (1pt2) (2013) 149, https://doi.org/10.1111/dsu.12130.
[30] Y. Lecun, L. Bottou, Y. Bengio, P. Haffner, Gradient-based learning applied to document recognition, Proc. IEEE 86 (11) (1998) 2278–2324, https://doi.org/10.1109/5.726791.

[19] The model performance here refers to not only accuracy but also robustness, reliability, generality, etc.

[31] Backpropagation applied to handwritten zip code recognition, Neural Comput. 1 (4) (1989) 541–551, https://doi.org/10.1162/neco.1989.1.4.541.
[32] P. Kim, Convolutional Neural Network, Apress, Berkeley, CA, pp. 121–147. doi:10.1007/978-1-4842-2845-6_6.
[33] Y. Tashtoush, Z. Odat, I. Alsmadi, M. Yatim, Impact of programming features on code readability, Int. J. Software Eng. Appl. 7 (6) (2013) 441–458, https://doi.org/10.14257/ijseia.2013.7.6.38.
[34] S. Scalabrino, G. Bavota, C. Vendome, M. Linares-Vasquez, D. Poshyvanyk, R. Oliveto, Automatically assessing code understandability: how far are we? ASE 2017 - Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, (2017), pp. 417–427, https://doi.org/10.1109/ASE.2017.8115654.
[35] A. Hindle, E.T. Barr, Z. Su, M. Gabel, P. Devanbu, On the naturalness of software, Proceedings of the 34th International Conference on Software Engineering, ICSE '12, IEEE Press, Piscataway, NJ, USA, 2012, pp. 837–847.
[36] H. Peng, L. Mou, G. Li, Y. Liu, L. Zhang, Z. Jin, Building program vector representations for deep learning, Proceedings of the 8th International Conference on Knowledge Science, Engineering and Management - Volume 9403 KSEM 2015, Springer-Verlag New York, Inc., New York, USA, 2015, pp. 547–553, https://doi.org/10.1007/978-3-319-25159-2_49.
[37] H.A. Basit, S.J. Puglisi, W.F. Smyth, A. Turpin, S. Jarzabek, Efficient token based clone detection with flexible tokenization, The 6th Joint Meeting on European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering Companion Papers - ESEC-FSE Companion '07, January, ACM Press, New York, USA, 2007, p. 513, https://doi.org/10.1145/1295014.1295029.
[38] A.T. Nguyen, T.N. Nguyen, Graph-based statistical language model for code, 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, vol.1, IEEE, 2015, pp. 858–868, https://doi.org/10.1109/ICSE.2015.336.
[39] A.L. Maas, A.Y. Hannun, A.Y. Ng, Rectifier nonlinearities improve neural network acoustic models, Proceedings of the 30 th International Conference on Machine Learning, 28 (2013), p. 6.
[40] A. Doan, R. Ramakrishnan, A.Y. Halevy, Crowdsourcing systems on the world-wide web, Commun. ACM 54 (2011) 86, https://doi.org/10.1145/1924421.1924442.
[41] D.P. Kingma, J. Ba, Adam: A Method for Stochastic Optimization(2014) 1–15. http://doi.acm.org.ezproxy.lib.ucf.edu/10.1145/1830483.1830503.
[42] J.G. Ibrahim, M.-H. Chen, S.R. Lipsitz, Monte Carlo EM for missing covariates in parametric regression models, Biometrics 55 (2) (1999) 591–596, https://doi.org/10.1111/j.0006-341X.1999.00591.x.
[43] P.-T. de Boer, D.P. Kroese, S. Mannor, R.Y. Rubinstein, A tutorial on the cross-Entropy method, Ann. Oper. Res. 134 (1) (2005) 19–67, https://doi.org/10.1007/s10479-005-5724-z.
[44] R. Likert, A technique for the measurement of attitudes. Arch.Psychol. (1932).
[45] R. Green, H. Ledgard, Coding guidelines, Commun. ACM 54 (12) (2011) 57, https://doi.org/10.1145/2043174.2043191.
[46] Q. Mi, J. Keung, Y. Yu, Measuring the stylistic inconsistency in software projects using hierarchical agglomerative clustering, Proceedings of the The 12th International Conference on Predictive Models and Data Analytics in Software Engineering - PROMISE 2016, ACM Press, New York, USA, 2016, pp. 1–10, https://doi.org/10.1145/2972958.2972963.
[47] Y. Zhang, B. Wallace, A Sensitivity Analysis of (and Practitioners' Guide to) Convolutional Neural Networks for Sentence Classification (2015).
[48] W. Fu, T. Menzies, Easy over hard: a case study on deep learning, arXiv:1703.00133 (2017).
[49] T. Downes-Le Guin, R. Baker, J. Mechling, E. Ruylea, Myths and realities of respondent engagement in online surveys, Int. J. Market Res. 54 (5) (2012) 1–21, https://doi.org/10.2501/IJMR-54-5-000-000.
[50] L.J. Cronbach, Coefficient alpha and the internal structure of tests, Psychometrika 16 (3) (1951) 297–334, https://doi.org/10.1007/BF02310555.