# A Survey on Deep Learning for Software Engineering

YANMING YANG, School of Computer Science and Technology, Zhejiang University, China
XIN XIA*, Software Engineering Application Technology Lab, Huawei, China
DAVID LO, School of Information Systems, Singapore Management University, Singapore
JOHN GRUNDY, Faculty of Information Technology, Monash University, Australia

In 2006, Geoffrey Hinton proposed the concept of training "Deep Neural Networks (DNNs)" and an improved model training method to break the bottleneck of neural network development. More recently, the introduction of AlphaGo in 2016 demonstrated the powerful learning ability of deep learning and its enormous potential. Deep learning has been increasingly used to develop state-of-the-art software engineering (SE) research tools due to its ability to boost performance for various SE tasks. There are many factors, e.g., deep learning model selection, internal structure differences, and model optimization techniques, that may have an impact on the performance of DNNs applied in SE. Few works to date focus on summarizing, classifying, and analyzing the application of deep learning techniques in SE. To fill this gap, we performed a survey to analyze the relevant studies published since 2006. We first provide an example to illustrate how deep learning techniques are used in SE. We then conduct a background analysis (BA) of primary studies and present four research questions to describe the trend of DNNs used in SE (BA), summarize and classify different deep learning techniques (RQ1), analyze the data processing including data collection, data classification, data pre-processing, and data representation (RQ2). In RQ3, we depicted a range of key research topics using DNNs and investigated the relationships between DL-based model adoption and multiple factors (i.e., DL architectures, task types, problem types, and data types). We also summarized commonly used datasets for different SE tasks. In RQ4, we summarized the widely used optimization algorithms and provided important evaluation metrics for different problem types, including regression, classification, recommendation, and generation. Based on our findings, we present a set of current challenges remaining to be investigated and outline a proposed research road map highlighting key opportunities for future work.

Additional Key Words and Phrases: Deep learning, neural network, machine learning, software engineering, survey

## 1 INTRODUCTION

In 1943, Warren Mcculloch and Walter Pitts first introduced the concept of the Artificial Neural Network (ANN) and proposed a mathematical model of an artificial neuron [7]. This pioneered a new era of research on artificial intelligence (AI). In 2006, Hinton et al. [4] proposed the concept of "Deep Learning (DL)". They believed that an ANN with multiple layers possessed extraordinary feature learning ability, which allows the feature data learned to represent the essence of the original data. In 2009, they proposed Deep Belief Networks(DBN) and an unsupervised greedy layer-wise pre-training algorithm [8], showing great ability to solve complex problems. DL has since attracted attention of academics and industry practioners for many tasks. Development of NvidiaâĂŹs graphics

*Corresponding author: Xin Xia

Authors' addresses: Yanming Yang, yym_1022@163.com, School of Computer Science and Technology, Zhejiang University, Hangzhou, China; Xin Xia, Software Engineering Application Technology Lab, Huawei, Hangzhou, China, xin.xia@acm.org; David Lo, School of Information Systems, Singapore Management University, Singapore, davidlo@smu.edu.sg; John Grundy, Faculty of Information Technology, Monash University, Melbourne, Australia, John.Grundy@monash.edu.


0360-0300/2021/1-ART1 $15.00
https://doi.org/10.1145/3505243

processing units (GPUs) significantly reduced the computation time of DL-based algorithms. DL has now entered a period of great development. In 2012 Hinton's research group participated in an image recognition competition for the first time and won the championship in a landslide victory by training a CNN model called AlexNet on the ImageNet dataset. AlexNet outperformed the second best classifier (SVM) by a substantial margin. In March 2016, AlphaGo was developed by DeepMind, a subsidiary of Google, which defeated the world champion of Go by a big score. With continuous improvements in DL's network structures, training methods and hardware devices, DL has been widely used to solve a wide variety of research problems in various fields.

Driven by the success of DL techniques in image recognition and data mining, industrial practitioners and academic researchers have shown great enthusiasm for exploring and applying DL algorithms in diverse software engineering (SE) tasks, including requirements, software design and modeling, software implementation, testing and debugging, and maintenance. In requirements engineering, various DL algorithms have been employed to extract key features for requirement analysis, and automatically identify actors and actions (i.e., user cases) in natural language-based requirement descriptions [EL09, IEEE94]. In software design and modeling, DL has been leveraged for design pattern detection [IEEE109], UI design search [ACM33], and software design mining [IEEE97]. During software implementation, researchers and developers have used DL for source code generation [IEEE112], source code modeling [EL16], software effort/cost estimation [IET01], etc. In software testing and debugging, various DL algorithms have been designed for detecting and fixing defects and bugs existed in software products, e.g., defect prediction [IEEE43], bug localization [IEEE102], vulnerability prediction [IEEE57]. It has been used for a variety of software testing applications, such as test case generation [ACM22], and automatic testing [IEEE07]. Researchers have applied DL to SE tasks to facilitate software maintenance and evolution, such as code clone detection [IEEE09], feature envy detection [IEEE10], code change recommendation [IEEE99], user review classification [IEEE111], etc.

However, there is a lack of a comprehensive survey of deep learning usage to date in SE. This study performs a detailed survey to review, summarize, classify, and analyze relevant papers in the field of SE that apply DL models. We collected, reviewed, and analyzed 250 papers published in 32 major SE conferences and journals since "deep learning" was introduced in 2006. We then analyzed the development trends of DL in SE, classified various DL techniques used in diverse SE tasks, analyzed DL's construction process, and summarized the research topics tackled by relevant papers. This study makes the following contributions:

(1) We conducted a detailed analysis of 250 relevant studies that used DL techniques in terms of publication trend, distribution of publication venues, and types of contributions. We analyzed an example in detail to describe the basic framework and the usage of DL techniques in SE.
(2) We provided a classification of DL models used in SE based on their architectures and an analysis of DL technique selection strategy.
(3) We performed a comprehensive analysis on data processing including data collection, data classification, data pre-processing, data representation. we also present the common ways to get the ground truth.
(4) We provided a description of each primary study according to six different SE activities and conducted an analysis on these studies based on their task types. These include regression, classification, recommendation, and generation tasks. We summarized commonly used datasets for different SE tasks.
(5) We investigated the key factors that impact the performance of DL models in SE, including model optimization and model evaluation.
(6) We discuss distinct technical challenges of using DL in software engineering and outline key future avenues for research on using DL in software engineering.

Section 2 presents researcher questions (RQs) and our Systematic Literature Review methodology. Section 3 investigates the distribution and evolution of DL studies for SE tasks. Section 4 conducted a comprehensive analysis on data processing from four perspectives – data collection, data classification, data pre-processing, and data

representation. Section 5 classifies research topics involved in primary studies, lists benchmark datasets for different SE tasks, and analyzes the relationships of DNNs with respect to multiple factors. Section 6 analyzes the commonly used optimization algorithms in DNNs and provides a set of evaluation metrics for different problem types. Section 7 presents the limitations of this study and its main threats to validity. Section 8 discusses the challenges that still need to be solved in future work and outlines a clear research road-map of research opportunities. Section 9 concludes this paper.

## 2 METHODOLOGY

We performed a systematic literature review (SLR) following Kitchenham and Charters [6] and Petersen et al. [10]. In this section, we present five research questions (RQs) and details of our SLR methodology.

We want to analyse the history of using DL models in SE by summarizing and analyzing the relevant studies, and providing the guidelines on how to select and apply the DL techniques. To achieve this, we first conducted a background analysis (BA) on primary studies and wanted to answer the following four research questions:

(1) **BA: What are the trends in the primary studies on the use of DL in SE? (See appendix B)**
(2) **RQ1: What DL techniques have been applied to support SE tasks?**
(3) **RQ2: How are datasets collected, pre-processed, and used?**
(4) **RQ3: What types of SE tasks and which SE phases have been facilitated by DL-based approaches?**
(5) **RQ4: What techniques are used to optimize and evaluate DL-based models in SE?**

The background analysis presents the distribution of relevant publications that used DL in their studies since 2006 to give an overview of the trend of DL in SE (See Appendix B). RQ1 provides a classification of different DL techniques supporting SE tasks and analyze their popularity based on their frequency of use in SE. RQ2 conducts a comprehensive analysis on the datasets in terms of data collection, data type classification, data pre-processing, data input forms, and the ground truth generation. RQ3 investigates what types of SE tasks and which SE phases have been facilitated by DNNs. RQ4 explores key optimization techniques and common evaluation metrics used in primary studies.

To perform a systematic literature review of DL for SE, we need to retain relevant studies published in a time range as many as possible and conduct a comprehensive analysis on those relevant studies for answering the above five RQs. To achieve this, we designed an effective search method including three steps: literature search and selection, literature filtering, and data extraction and collection. For space limitations, we include a detailed description of each step in our SLR in Appendix A. In the following sections we answer each of our RQs in detail from the SLR analysis.

## 3 RQ1: WHAT DL TECHNIQUES ARE APPLIED TO SUPPORT SE TASKS?

### 3.1 Classification of DNNs in SE

Many sorts of DNNs have been proposed, and certain neural network architectures contain diverse DNNs with different implementations. For instance, although LSTM and GRU are considered two different DNNs, they are both RNNs. We categorized DL-based models according to their architecture and different DNNs used. We classified the architecture of various DNNs into 4 categories: the layered architecture, Encoder-Decoder, AutoEncoder (AE), and Siamese Network [2, 9]. We provided a detailed classification of DNNs into six categories, i.e., RNN, CNN, FNN, GNN, transformers, and tailored DNN models where tailored DNNs include the DNNs not often used in SE, e.g., DBN, HAN, etc. Table **??** (See Appendix C) shows the variety of different DNNs and also provides specific references in every category.

As can be seen from Table **??** (See Appendix C) where we compare DL architectures, layered-based DNNs are the most popular and widely used architecture for SE tasks. In the layered architecture, almost 200 primary studies used 8 different kinds of RNN-based models to solve practical SE issues, where LSTM is the most often

applied RNN-based model, followed by standard RNN. The variants of LSTM, such as GRU and Bi-LSTM, are often adopted by researchers in multiple research directions, such as program repair, bug detection, etc. 74 primary studies employed CNN-based models, where almost 90% of studies employed CNN. The FNN-based model is the third most frequently used family, followed by GNN-based models and tailored models. There are 13 combined DNNs proposed in tailored models.

37 primary studies leveraged different types of DNNs following the Encoder-Decoder architecture, where RNN-based models were used in 28 studies, which is much higher than the number of other models used, i.e., CNN and FNN. In the AE architecture, 8 studies used FNN-based AEs as their proposed novel approaches; only 5 and 1 studies selected GRU and CNN to construct AEs respectively. There are only 3 studies published in 2020 that built DL models using the Siamese Network architecture to address SE tasks. For example, Wu et al. [IEEE27] leveraged the Siamese Network to detect functional clones. Pan et al. [ACM25] proposed a novel approach to conduct automated testing towards Android applications by using the Siamese Network.

In addition, as Google presents a language understanding model so-called BERT in 2019, which is built by pre-training a bidirectional Transformer and achieves good performance in 11 different tasks, the transformer has become popular recently. A number of studies applied the transformer [ACM10, ACM26, IEEE36] or pre-trained BERT [ACM07, IEEE19, IEEE29, IEEE34] to get the vector representation of code and natural language in different SE tasks, such as code comment generation [IEEE36], code generation [ACM10], code completion [ACM34], vulnerability detection [ACM26], program repair [IEEE29], etc.

### 3.2 DL technique selection strategy

Since heterogeneous DNNs have been used for SE tasks, selecting and employing the most suitable network is a crucial factor. We scanned the relevant sections of DL technique selection in all of the selected primary studies and classified the extracted rationale into three categories.

**Characteristic-based selection strategy** (*S*1): The studies justified the selected techniques based on their characteristics to overcome the obstacles associated with a specific SE issue [ACM02, EL12, IEEE41, SP10]. For instance, most of the seq2seq models were built by using RNN-based models thanks to their strong ability to analyze the sequence data.

**Selection based on prior studies** (*S*2): Some researchers determined the most suitable DNN used in their studies by referring to the relevant DL techniques in the related work [ACM01, ACM23, EL16]. For instance, due to the good performance of CNN in the field of image processing, most studies selected CNN as the first option when the dataset contains images.

**Using multiple feasible DNNs** (*S*3): Though not providing any explicit rationale, some studies designed experiments for technique comparisons that demonstrated that the selected algorithms performed better than other methods. For example, some studies often selected a set of DNNs in the same SE tasks to compare their performance and picked up the best one [IEEE104, EL03, SP08].

We noticed that the most commonly selection strategy is *S*1 (i.e., Characteristic-based selection strategy), accounting for **68.8%**, nearly 3 times that of *S*2 **(27.2%)**. Only **10%** of primary studies adopt *S*3 to select their suitable DL algorithms.

**Summary**

(1) There are 4 different DNN architectures and over 30 different DNNs used in the selected primary studies.
(2) We used a classification of DL-based algorithms from two perspectives, i.e., their architectures and the families to which they belong. The architecture can be classified into four types: Layered architecture, Encoder-Decoder, AutoEncoder (AE), and Siamese Network; the family can be classified into six categories: RNN-based, CNN-based, FNN-based, GNN-based, Transformer-based, and Tailored models.

(3) Compared with other DNN architectures, the layered architecture of DNNs is by far the most popular option in SE.
(4) Four specific DNNs are used in more than 20 primary studies, i.e., CNN, LSTM, RNN, and FNN, and each of them has several variants that are also often used in many SE tasks.
(5) As Google proposed a transformer-based language understanding model named Bert in 2019, many studies used the transformer-based models (e.g., Bert) to generate the vector representation of code and natural language.
(6) We summarized three types of DNN-based model selection strategies. The majority of studies adopted *S*1 to select suitable DL algorithms. Only 10% of primary studies used *S*3 as the model selection strategy due to the heavy workload brought by *S*3.

## 4 RQ2: HOW ARE DATASETS COLLECTED, PRE-PROCESSED, AND USED?

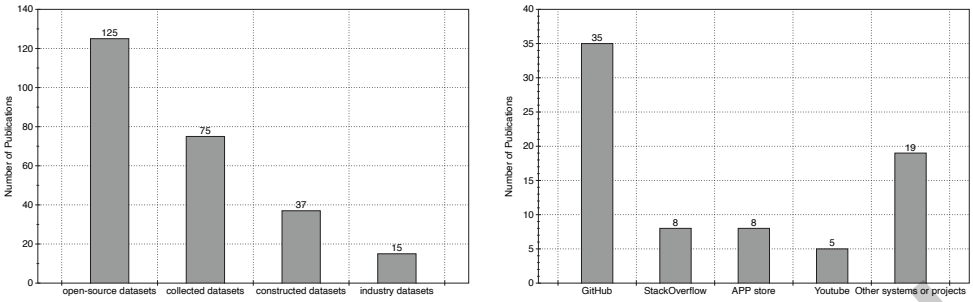
Data is one of the most important roles in the training phase. There are many factors that may impact the quality of datasets, such as the source of a dataset, the dataset scale, the dataset whether has been preprocessed and the ground truth in the dataset. For example, using unsuitable datasets or data processing techniques can result in failed approaches or tools with low performance. We focused on the data used in primary studies and conducted a comprehensive analysis on the steps of data collection, data classification, data pre-processing, and data representation.

### 4.1 What were the sources of datasets used for training DNNs?

DL models often have the data-hungry problem [van2014modern], i.e., the scale of datasets is not large enough to effectively train a DL model. Hence, where and how to collect large-scale datasets is a key research question for DL model construction. In this section, we first investigated the methods for obtaining datasets. Through analysis on the data collection methods, datasets can be classified into four categories by utilizing different collection methods: open-source datasets, collected datasets, constructed datasets, and industry datasets. Open-source datasets essentially incorporate benchmarks, datasets that have been utilized in prior studies, and public datasets. Collected datasets represent the datasets that comprise various software projects and those projects are gathered from certain forums and websites, such as GitHub, Stack Overflow, Youtube, etc. Since there are no public and suitable datasets for some SE tasks, many studies constructed the datasets according to their specific requirements. Besides, a number of studies adopted industry datasets to evaluate their DL models, but most of these are not available, i.e., closed-source data, which increases the difficulty for other studies to reuse them.

Fig. 1(a) shows the source of datasets in the primary studies. It can be seen that 125 studies trained DNNs by using open-source datasets. One potential reason for choosing open-source datasets is that using open-source datasets is a very convincing way to evaluate the performance of DNNs, which is beneficial for other studies to reproduce and replicate those DNNs. For this reason, the existence of widely accepted datasets in certain SE issues (e.g., code clone detection, software effort/cost prediction, etc) are the first choice for most studies. However, if no such datasets exist, studies can only collect related materials as a dataset from some forums or just construct a dataset. 75 studies were estimating the effectiveness of DNNs by using collected datasets, followed by constructed datasets (36). Only 15 studies partnered with companies and used industry data to train DNNs, accounting for 6%.

In addition, we found two studies whose datasets come from two different data sources. Li et al. [IEEE118] trained a DL model to extract features and variabilities from requirement specifications and utilized an open-source dataset where the requirement documentations in the dataset are gathered from Body Comfort System (BCS). To enlarge the scale of data, they also collected requirement specifications from different domains. Thus, the data sources in this study combine the open-source and collected data. Wu et al. [IEEE138] trained their DL models with two different training datasets, respectively. One is an open-source dataset, which has been used in prior work for




(a) The collection strategies of DL-related dataets. (b) The sources of collected datasets.

Fig. 1. The source of datasets used in primary study papers

the programming language syntax correction task. And they crawled useful content from the Codeforces website [1], which is an online judge website, as a supplement. Thus, there are two data sources for datasets used in this study: open-source and constructed data.

A large proportion of studies performed a series of experiments on large-scale datasets so as to verify the scalability and robustness of their models. However, the lack of open-source datasets during solving certain SE issues facilitates practitioners to collect multiple small datasets from different places. Fig. 1(b) describes the source of collected datasets. As tens of thousands of software practitioners contribute to the GitHub community by uploading source code of their software artifacts, GitHub has become the most frequently used source of collected data (46.7%). Stack Overflow and APP stores are the second most common sources for collecting data. We discussed the reason why the number of studies that collected data from Stack Overflow is significantly smaller than that of studies gathering data from GitHub. One reason is that since most of the studies leverage the source code as their datasets (see Section 5.1), Stack Overflow contains source code on a small scale from a huge number of knowledge units (i.e, Q&A pairs), and even APP store only involves text-based data, such as the description of APP, user comments, etc. Yet, GitHub not only includes huge amounts of source code of numerous software systems but also much text-based information that can be used, e.g., pull requests and commit messages. Such difference between GitHub and Stack Overflow as well as APP stores makes most researchers tend to find out valuable data on GitHub. Five studies collected screenshots from YouTube, only accounting for 6.7%. Apart from these commonly used data sources for collected data, 19 studies collected their datasets to train DL-based models from different systems and projects. For instance, Deshmukh et al. [IEEE54] collected bug reports from several bug tracking systems, i.e., issue tracking systems of Open Office, Eclipse, and Net Beans projects, as datasets to build a DL-based model for duplicated bug detection.

## 4.2 What were the types of SE datasets used in prior DL studies?

The datasets used in primary studies are of various data types. It is essential to analyze data types of datasets since the relationship between the type of implicit feature being extracted and the architecture has a dominating influence on model selection. Besides, the difference in data types used in primary studies also impacts the data pre-processing technique adoption. Therefore, we classify and summarize the data types in primary studies. Then, we further conduct an analysis to interpret how data types determine the choice of data pre-processing techniques and DNNs.

***Data type Classification.*** We classified the data types of used datasets into six categories – code-based, text-based, metric-based, graph-based, software repository-based, and combined data types. Table **??** (See Appendix D) describes specific data in each data type. We summarize the datasets used in 250 primary studies. We observe that

[1] https://codeforces.com/

most of these studies adopted code-based datasets to conduct their experiments, where over 80% of them used source code directly in some important SE activities, such as software testing and maintenance. This phenomenon indicates that source code, as the most valuable data type in SE, can provide more complete information for software in contrast with other code-based data types. Text-based datasets include the most data types (19). The bug report (11) and requirement documentation (8) are the two most commonly studied data types, followed by the issue report (4). In addition, the types of the vulnerability report, code comment, and log information, as common text-based information in SE, are used as the datasets with high frequency. Some studies used metric-based datasets to address specific SE tasks, such as software metrics and code metrics. For example, Kumar et al. [EL18] used a metric-based dataset to predict the maintainability of an object-oriented software system. The dataset contains eight software metrics: Weighted method per class (WMC), Depth of inheritance tree (DIT), Number of children (NOC), Data abstraction coupling (DAC), Message-passing coupling(MPC), Response for class (RFC), Lack of cohesion among methods (LCOM), Number of methods (NOM), SIZE1, and SIZE2, and they also described the definition of each of them. The graph-based datasets used in SE often involve GUI images, programming or video screenshots, etc. Those datasets can be used to detect GUI elements in Android applications or identify incorrect programs from students' programming tasks. To the datasets used in primary studies, most of the graph-based datasets consist of a major number of GUI images, followed by programming screenshots. Only two image-based datasets contain video screenshots and the behavior trajectory of the model class, respectively.

Other than the above types of datasets, there are several datasets constructed by collecting the domain-specific data elements from open-source software repositories and forums, such as GitHub and Stack Overflow (SO). We summarized these datasets and classified them into software repository-based ones. As we can see from Table **??**, the knowledge unit (QA pairs) in SO is the most commonly used data type, and seven studies selected id as their datasets. Apart from knowledge units in SO, GitHub is also an important software repository as the source of SE datasets, which includes lots of valuable content that can be explored further for finding out the solutions to specific SE problems, such as pull-request, issues, and commits.

The datasets in a few research encompass more than one single sort of data type, and we classified those datasets into the "mixed dataset" category. From Table **??** (See Appendix D), we can see that there are 8 varieties of mixed datasets, where all of them contain code-related data, which also indicates source code is the most important data in SE. Besides, 7 out of 8 mixed datasets contain two different data types, and only one study used three different kinds of data, i.e., source code, diff files, and commit message. For instance, Xu et al. [MK02] present a novel DL model to automatically generate commit messages for code changes by way of the use of three different data types as their dataset, e.g., source code, diff files, and the commit messages. Besides, "source code and its comments" is the most common combination for mixed datasets, and this combination can be used in multiple SE tasks, such as code generation, code completion, comment generation, etc.

### 4.3 How have data types determined the choice of data-preprocessing techniques?

Raw datasets used in studies need to undertake several data processing techniques to attain clean datasets. In addition, different dataset types affect the adoption of data pre-processing strategies. Therefore, there are near and apparent relationships between the types of datasets and the data pre-processing strategies used. To reveal those relationships, we collected the data pre-processing process for each type of dataset and summarized all steps and strategies utilized in related studies. [2] In this section, we present the data pre-processing procedures for four different types of data, including code-, text-, metric-, and graph-based data. These four data pre-processing procedures can generally clean all types of datasets.

[2]Note that: if the reader has a tendency to take the data processing steps of certain data types provided in this article, he just needs to pick some of the appropriate steps in keeping with the characteristics of their dataset.

Table 1. The data pre-processing procedure for code-based datasets.

| Pre-processing techniques | Description | Examples | References |
|---|---|---|---|
| Data extraction | Extract useful code blocks from the corresponding datasets according to different gradularities and SE tasks. | function-level, file-level code snippets, or the source code of the whole software systems | [IEEE32, SP07, ACM13, IEEE32, SP07] |
| Unqualified data deletion | Apply some rules to remove some unqualified samples and retain appropriate samples for different SE tasks. | remove the code samples without frequent occurrences in the dataset; retain the functions with certain method names | [MITP02, IEEE06, IEEE32, IEEE35] |
| Duplicated instance deletion | Remove duplicated samples in the dataset. | | [ICLR10, MITP02, IEEE06, IEEE34, IEEE35] |
| Data compilation | Compile the code blocks to generate the compiled files. | Transform java files into .class files throughout compilation | [IEEE27, ICLR06, ICLR04, ICLR07] |
| Uncompilable data deletion | Remove the code samples that cannot be compiled. | | [IEEE27] |
| Code representation | Token-based code representation | Tokenize the source code or binary code into tokens | [IEEE17, IEEE32] |
| | Tree-based code representation | Parse the source code or binary code into AST | [ACM23, ICLR10, IEEE47, IEEE51, MK09, EL16, EL20, IEEE72] |
| | Graph-based code representation | Generate the source code or binary code into XFG (CFG, DFG) | [EL03, MITP01, MITP02, MITP03, MITP05] |
| Data segmentation | Split the code samples for training or testing the model. | A dataset is often divided into two parts (e.g., training set and test set) or three parts (e.g., training set, validation set, and test set) | [IEEE35, ICLR02, ACM16, IEEE24] |



**The data pre-proceesing procedure for code-based datasets.** In this section, we summarized the data pre-processing strategies used in primary studies for cleaning code-based datasets. There are mainly seven steps during the complete pre-processing procedure for code-based datasets. We describe the specific strategies in Table 1.

The initial phase in the code-based data pre-processing procedure is code-based data extraction, i.e., not all parts of the dataset are helpful for SE tasks. Accordingly, researchers need to filter out superfluous data and hold valuable source code. For example, researchers need to eliminate the code comments in datasets when performing clone detection as the input of code clone detectors is only source code. Yet by far, for the vast majority of code generation tools, code comments are the valuable data, assisting tools to understand the semantic of source code needed to be generated. After code extraction, researchers often provide some rules to remove unqualified code. For instance, Ben et al. [MITP02] discard code statements that occur less than 300 times in their datasets, and Lacomis et al. [IEEE06] filtered out the methods that have no renamed variables together with ones with more than 300 AST nodes in their data pre-processing phase. The "Duplicated instance deletion" phase is a necessary step for most studies. The two pre-processing steps, "data compilation" and "uncompilable data deletion", are frequently utilized by certain studies that need to generate binary code, Assembly code, or intermediate representation (IR) of code. After obtaining clean source code, there are three main methods to represent code, i.e., token-based, tree-based, and graph-based code representation methods. For studies using source code as their datasets, the most commonly

Table 2. The data pre-processing procedure for text-based datasets.

| Pre-processing techniques | Description | Examples | References |
|---|---|---|---|
| Data extraction | Extract useful texts from different SE documentations, and sometimes extract texts according to the time order. | Software requirements, issue reports, bug reports, code comments etc. | [ACM03, IEEE03, IEEE22, SP05] |
| Initial data segmentation | Split the data to meet the requirements of different tasks. | Different studies may split texts into sentences or worlds. | [ACM07, IEEE04, IEEE21] |
| Unqualified data deletion | Eliminate useless or unqualified parts in the text. | Filter out source code fragments in pull requests | [ACM07, ACM21, IEEE04, IEEE19, SP02] |
| Text pre-processing | Conduct further pre-processing operations towards text | Lowercase the texts, remove stop words and white spaces, etc | [ACM21, IEEE03, IEEE21, IEEE30, SP02] |
| Duplicated instance deletion | Remove duplicate samples in the datasets | | [IEEE03] |
| Data tokenization | Use token-based text representation | Tokenize the texts, sentences, or words into tokens | [IEEE04, IEEE60, IEEE61, EL12] |
| Data segmentation | Split the text-based samples for training or testing the model. | A dataset is often divided into two parts (e.g., training set and test set) or three parts (e.g., training set, validation set, and test set) | [IEEE03, IEEE40] |

used representation is to parse the source code into ASTs, followed by the graph-based representation way. The last step in the data pre-processing procedure is data segmentation or data splitting. Studies using DL-based models need to split the dataset into two or three parts for training, validating, and testing DL models.

**The data pre-processing procedure for text-based datasets.** Compared with code-based datasets, there are some differences in the data pre-processing strategy for text-based datasets. We listed the specific steps in Table 2. Seven phases are commonly used for pre-processing text-based datasets, where some steps are similar to those for code-based datasets, such as filtering out unqualified texts, duplicated instance deletion, and data splitting. There are still some pre-processing techniques with the characteristic of the text. For instance, some data needs to be extracted in chronological order from SE documentation. Hence, researchers require to consider the influence of time in the text-based data extraction step. Different from source code, text-based data can be cut into sentences or words according to the need of specific SE tasks. The two steps, unqualified data deletion and text pre-processing, are often combined together by many studies [SP02, ACM21]. In these two steps, like code-based data pre-processing strategies, researchers also present some rules to eliminate the useless contents and also conduct further processing on the text-based data, such as lowercasing the texts of the data, removing stop words, etc. Two approaches are frequently used in DL models to tokenize the text-based data, i.e., word2vec and one-hot encoding. For word2vec is a general concept as a number of studies encoded each word into a vector, but some studies considered a sentence as a unit and leveraged the approach similar to word2vec to generate a representation vector for a sentence. Finally, data segmentation is still a necessary phase for text-based datasets.

**Data pre-processing procedure for metric-based datasets.** Table 3 shows the data pre-processing procedure for metric-based data. There are three steps to clean the metric-based datasets, including duplicated instance deletion, unqualified data deletion, data normalization, and data segmentation. When getting a dataset that contains a set of metrics, duplication instance deletion is still a necessary step for data pre-processing. Different from the pre-processing procedure for code-based and text-based datasets, the "unqualified data deletion" phase for metric-based datasets is target to eliminate the instances without metric values or instances with wrong metric values. Throughout the above data pre-processing steps, relatively clean data can be achieved but the ranges of

Table 3. The data pre-processing procedure for metric-based datasets.

| Pre-processing techniques | Description | Examples | References |
|---|---|---|---|
| Duplicated instance deletion | eliminate the i with same metric values and class labels. | Those defects whose metric values are all the same need to be deleted. | [EL10, IEEE87] |
| Unqualified data deletion | Some instances with wrong or without metric values need to be eliminated. | If one or more metrics in an instance are missing due to some reasons. | [EL10, IEEE86] |
| Data Normalization | The metric values need to be normalized when those value have different orders of the magnitude. | The value range for a metric is between 0 and 10, but that for another one is over 1000. | [EL10, ACM33] |
| Data segmentation | Split the metric-based instances for training or testing the model. | A dataset is often divided into two parts (e.g., training set and test set) or three parts (e.g., training set, validation set, and test set) | [EL10, ACM33, IEEE87] |

Table 4. The data pre-processing procedure for image-based datasets.

| Pre-processing techniques | Description | Examples | References |
|---|---|---|---|
| Data collection | Collect images from different software systems. | Collect or take the screenshots of the app GUI and the programming platforms. | [IEEE44, ACM24] |
| Duplicated instance deletion | Eliminate the duplicated figures in the datasets. | Remove the repeated screenshots from the dataset. | [SP08, IEEE44] |
| Unqualified instance deletion | Remove the unqualified instances from the dataset. | Some studies eliminated the UI screenshots that only contain Layout components, WevViews, or the components that only appeared very few times. | [ACM24, IEEE129] |
| Data augmentation | To ensure the proper training support, the data augmentation technique can be adopted to enlarge the scale of datasets. | Data augmentation can generate new images to increase the size of the datasets by rotating the existing images or changing the colors of the images. | [IEEE129] |
| Data trimming | Tune the scale of data, i.e., the action can resize images into other appropriate sizes. | The images are rescaled into 300*300 pixels. | [IEEE78, IEEE23] |
| Data annotation | Data annotation is the categorization and labeling of data for different applications. | Give a correct label for every image in the dataset throughout manual verification. | [SP08, ACM24] |
| Data segmentation | Split the image-based samples for training or testing the model. | A dataset is often divided into two parts (e.g., training set and test set) or three parts (e.g., training set, validation set, and test set) | [IEEE129] |



metric values in a dataset are pretty different. Therefore, adopting data normalization can effectively normalize the range of metric values, which facilitates DL models to further operate the metric values.

**Data pre-processing procedure for image-based datasets.** Table 4 describes the data pre-processing procedure for image-based data, involving seven pre-processing phases: image-based data collection, duplicated instance deletion, unqualified data deletion, data augmentation, data trimming, data annotation, and data segmentation. Different from other types of data, there are few open-source image-based datasets in SE, and thus most studies created images and collected those images as their datasets. For this reason, we use the name "Data collection" to express the first phase for pre-processing image-based data. For instance, Chen et al. [IEEE44] took the screenshots for the GUI of APPs to obtain image-based datasets. "Duplicated instance deletion" and "Unqualified instance deletion" are still applied in some studies [SP08, IEEE44, ACM24, IEEE129]. Since training a DL model requires

the dataset on a large scale, "Data augmentation" becomes a good option to increase the dataset size when the preprocessing procedure is conducted for image-based data. After a dataset containing enough image-based data, these instances often need to be cropped by the "Data trimming" phase to the required size. Another problem is that Since most of the image-based data in SE are generated by collecting or creating images from forums or apps (e.g. GitHub, SO, Android, etc ), which are not open-source datasets or benchmarks. For this reason, many image instances have no or wrong labels, and therefore the "Data annotation" phase usually occurs in the processing of image data pre-processing. In this phase, researchers need to provide a correct label for each image to ensure that the dataset can be used to train an effective DL model. It is indisputable that the "Data segmentation" phase is the final step of data pre-processing.

### 4.4 What input forms were datasets transformed into when training DNNs?

After selecting suitable datasets and obtaining clean data through data pre-processing, data needs to be transformed into appropriate forms that can be used as the input of DNNs. In this section, we summarized five commonly used input forms of datasets when training DL models: Token-based input, Tree/graph-based input, Feature/metric-based input, Pixel-based input, and Hybrid input. We give a definition of each input form and also provide an example for each input form for better understanding. Finally, we describe usage trends for different input forms during training models.

**Token-based input.** Since some studies treated source code as text, they used a simple program analysis technique to generate code tokens into sequences and transformed tokens into vectors as the input of their DL-based models. A token-based input form can be applied to source code and text-based data when processing related datasets. For instance, Liu et al. [IEEE34] used Bert to generate the vector representation of source code. They treated source code as texts and transformed source code into token sequences as the input of their model.

**Tree/graph-based input.** To better comprehend the structure of source code, several studies convert source code into Abstract Syntax Trees (AST) or Control Flow Graphs (CFGs), and then generate vector sequences by traversing the nodes in each tree or graph. For instance, different from the approach proposed by Liu et al. [IEEE34], Alone et al. [ICLR07] also proposed an alternative approach to encode source code by using a decoder-encoder architecture with LSTM cells. To generate the input of their model, they first parsed source code into ASTs, which is a tree-based representation way. After that, they represent each node using a learned embedding matrix and encoded the entire sequence using Bi-LSTM.

**Feature/metric-based input.** For analyzing the characteristics of software artifacts, some studies applied datasets consisting of features or metrics extracted from different products, and thus the input form of the models proposed in these studies is software feature/metric-based vectors. For example, Barbez wt al. [IEEE50] present a DL-based approach to detect anti-patterns. They first mined version control systems to get the historical values of structural code metrics and trained a CNN model to infer anti-pattern classes from these metric values. To identify anti-patterns from classes, they considered seven different source code metrics: Assess To Foreign Data (ATFD), Lack of COhesion in Methods (LCOM5), Line of Code (LOC), Number of Attributes Declared (NAD), Number of Associated Data Classes (NADC), Number of Methods Declared (NMD), and Weighted Method Count (WMC). Zhao et al. [IEEE90] proposed a model, named Simplified Deep Forest (SDF) to perform Just-in-Time (JIT) defect prediction. To identify JIT defects, they generated the feature representation of defects by using six features to depict commits in the defect data, including the Number of unique changes to modified files (NUC), Lines of code deleted (LD), Lines of code added (LA), Number of modified files (NF), Number of modified directories (ND), and Number of developers working on the files (NDEV).

**Pixel-based input.** Some studies used datasets containing a large number of images and screencasts, e.g., program screencasts, UI images, etc. When preprocessing these datasets, they often broke down screencasts into pixels as an effective input form, for analyzing graph-based datasets in different SE tasks, such as bug detection,

code extraction, etc. For instance, Chen et al. [IEEE44] trained a neural machine translator to translate a UI design image into a GUI skeleton. Since their dataset consists of lots of images, they transformed images into pixels and constructed input matrices as the input of their translator.

**Hybrid input.** Many studies combined two or more data types extracted from software products to build comprehensive datasets with more information for enhancing the quality and accuracy of proposed models. For instance, Leclair et al. [ACM16] proposed a novel approach for generating summaries of programs not only by analyzing their source code but also their code comments. For example, Leclair et al. [IEEE136] trained a GNN model for code summarization. The dataset used in their study contains java methods and their comments. They parsed source code into ASTs and tokenized comments into tokens. Thus, they used hybrid inputs to represent their data, i.e., two different input forms: tree-based and token-based input forms.

**Embedding.** In addition, only the vectorization of data can be the input of DNNs. We found two techniques were often used to transform different input forms of data into vectors: One-hot encoding and Word2vec. For example, when source code is divided into node tokens, researchers require to adopt a proper embedding method to transform tokens into vectors. In primary studies, only 5 studies [ACM06, SP10, IEEE50, IEEE53, IEEE76] produced the input of their models by adopting the One-hot technique, and other studies adopted Word2vec and its variants to generate vector representation of inputs.

Table 5 depicts the input formats of DL-based models in primary studies. We can see that over 47.6% of studies transformed data (i.e., source code and various documentations) into a token-based input form (119), where 51 studies treated source code as texts and directly converted code into token sequences as the input of models. Also, 13 studies used both source code and text-based data, and also constructed a token-based data structure as the input of DNNs.

59 studies processed source code into tree- or graph-based representations. In these 59 studies, over 74% of studies parsed source code into ASTs and only around 26% selected XFG (e.g. CFG and DFG) to describe the control and data flow of code. This trend is quite beyond our expectation. Our intuition was that graph-based code representation can retain much more structural and functional information of code than tree-based representation, and thus studies would be more likely to select to parse source code into XFGs for achieving the better performance of DNNs. After analysis, we think of a potential reason on why studies transform source code into ASTs is that parsing code into ASTs is much easier than converting them into XFGs.For example, Gao et al. [54] proposed an embedding approach to detect code clones. They constructed ASTs from code and transformed them into vector sequences. Then they treated the vector sequences as text-based data and leveraged the sent2vec model to generate code embeddings. But, for studies using graph-based code representations,they have less choice for DL model selection. Only 37 studies constructed input matrices for transforming different metrics into the metric/feature-based input form of DNNs, followed by the pixel-based input form. 13 studies using image-based datasets split screenshots into pixels as the basic unit of the input form

As shown in Table 5, there are 22 studies that used more than one single data type, combining source code and text-based data. 19 studies parsed source code into ASTs and tokenized text-based data and only one study used graph-based representation to describe code information. There are two studies that mixed features and information extracted from source code and text-based data to train DNNs.

## 4.5 How did studies generate the ground-truth for unlabeled datasets?

A ground truth dataset refers to a dataset whose instances have correct labeled instances. For most SE issues, the ground truth dataset is an important factor to accurately evaluate the performance of DNNs. In fact, most of the ground-truth datasets are open-source ones. However, there are no open-source datasets in many SE tasks, and thus many studies have to customize the datasets that meet their requirements. However, those datasets are unlabeled, unable to be directly used to train and test DNNs. To address this problem, there are several methods to generate

Table 5. The various input forms of DL-based models proposed in primary studies.

| Family | Input forms | #Studies | Total | References |
|---|---|---|---|---|
| Token-based input | Code in tokens | 51 | 119 | [AAAI01, AAAI03, AAAI05, AAAI06, ACM01, ACM06, ACM09, IEEE07, IEEE15, IEEE32, IEEE17, IEEE34, SP02, ACM10, ICLR02, ICLR03, ICLR04, ICLR09, ICLR11, ACM18, ACM13, IEEE42, IEEE66, IEEE55, IEEE24, IEEE67, MK06, MK07, ACM27, MITP08, ACM39, IEEE84, IEEE91, IEEE101, IEEE103, IEEE112, IEEE121, IEEE115, IEEE116, IEEE122, IEEE126, IEEE53, ACM31, W01, ACM15, ACM26, ACM17, IEEE123, ACM12, ACM30, IEEE08, IEEE83] |
| | Text in tokens | 55 | | [ACM03, ACM05, ACM07, ACM08, IEEE03, IEEE04, IEEE11, IEEE12, IEEE13, IEEE16, IEEE191, IEEE22, IEEE36, IEEE30, SP03, SP05, IEEE39, ACM21, ACM22, IEEE45, IEEE46, IEEE48, IEEE59, IEEE54, IEEE57, IEEE58, IEEE62, MK02, IEEE70, EL02, EL05, EL09, EL12, IEEE71, IEEE76, IEEE79, IEEE88, IEEE94, IEEE95, IEEE97, IEEE99, IEEE100, IEEE118, IEEE119, IEEE58, IEEE120, SP10, ACM34, IEEE134, fIEEE125, IEEE130, IEEE137, IEEE139, IEEE142, IEEE98] |
| | Code and text in tokens | 14 | | [IEEE29, IEEE73, IEEE75, IEEE25, IEEE108, IEEE104, IEEE105, IEEE106, IEEE18, IEEE107, IEEE26, IEEE135, ACM19, IEEE132] |
| Tree/graph-based input | Code in tree structure | 43 | 59 | [AAAI02, AAAI04, AAAI08, ACM02, IEEE01, IEEE06, IEEE31, IEEE37, ICLR07, ICLR08, ICLR05, ICLR10, ACM20, IEEE38, ACM23, IEEE43, IEEE46, IEEE51, IEEE63, MK04, MK08, MK09, MK05, ACM28, EL16, EL20, EL13, IEEE82, IEEE72, IEEE85, IEEE92, IEEE113, ACM35, IEEE131, IEEE138, IEEE141, IEEE74, IEEE77, IEEE80, IEEE81, MITP06, MITP07, MITP04] |
| | Code in graph structure | 16 | | [ACM04, IEEE27, SP09, ICLR06, ICLR01, ACM11, IEEE64, MK03, EL03, MITP01, MITP02, MITP03, MITP05, ACM38, IEEE28, ACM37, IEEE98] |
| Feature-based input | feature/metric | 37 | 37 | [IEEE10, IEEE14, IEEE21, SP01, SP04, SP06, IEEE40, ACM14, IEEE52, IEEE56, IEEE60, IEEE61, IEEE65, IET01, IEEE68, ACM25, ACM29, EL01, EL06, EL07, EL10, EL11, EL15, EL17, EL18, EL19, IEEE86, IEEE87, IEEE90, IEEE96, IEEE19, ACM33, IEEE114, IEEE117, ACM32, IEEE127, IEEE50, IEEE117] |
| Pixel-based input | pixel | 13 | 13 | [IEEE23, SP08, ACM24, IEEE41, IEEE44, IEEE69, EL04, IEEE78, IEEE93, ACM36, ACM33, IEEE129, IEEE140] |
| Hybrid input | Code in tree structure + text in token | 19 | 22 | [IEEE09, IEEE33, IEEE35, IEEE02, SP07, IEEE49, ACM16, IEEE20, MK01, EL08, EL14, EL21, IEEE89, IEEE110, IEEE124, IEEE128, IEEE127, IEEE135, IEEE136, IEEE135] |
| | Code in grap structure + text in token | 1 | | [AAAI07] |
| | Code features + text in token | 2 | | [IEEE05, IEEE111] |



labels for the datasets without ground truth. We sampled 35 primary studies that trained DNNs on closed-source datasets (i.e., using collected datasets, constructed datasets, and industry datasets) without ground truth labels to investigate how these studies label their datasets. We summarized data labeling methods and classified them into three categories, which are listed in Table 6.

As shown in Table 6, manual data labeling is the most frequently used way to generate ground-truth labels for datasets, including five main methods. Among them, one of the standard labeling methods is card sorting. For

Table 6. The methods used for generating the ground-truth for unlabeled datasets.

| Family | Methods | Description | Examples |
|---|---|---|---|
| Manual method | Card sorting | The process of data labeling conforms to that of cart sorting. | [IEEE19, IEEE30, IEEE76, IEEE97, IEEE140] |
| | Rule-based data labeling | Researchers present some rules and justify labels of instances in their datasets based on them. | [IEEE05, IEEE11, ACM19, IEEE50, IEEE117, IEEE26] |
| | Data labeling by user study/experience | Researchers hire or invite domain experts or other SE practitioners to do data labeling. | [ACM08, IEEE18, IEEE22, IEEE58, IEEE78, MITP03, IEEE118, IEEE119, IEEE58, IEEE120] |
| | Simple manual data labeling | Only a simple description (only one or several sentences) is provided for the process of their manual labeling. | [ACM03, SP06, IEEE70, ACM25, ACM26, MK06, IEEE72, IEEE79] |
| | Random data labeling | Researchers randomly sample a part of instances in the datasets to conduct cart sorting. | [IEEE13, IEEE15] |
| Automated method | Tool-based data labeling | Researchers use proper tools to automatically label the ground truth from datasets. | [SP08, ICLR04, IEEE54, MITP04] |
| Combined method | Tool-based manual labeling | Researchers manually label the datasets with the help of tools. | [EL08] |

example, Zhao et al. [ACM07] detailed depicted the tagging process. They invited five persons, including four students and one senior researcher. To best avoid bias, they divided those four students into two groups to do data labeling, and the senior researcher worked as the mediator in case of conflicts. Different from cart sorting, a few studies made the rules and followed them to generate ground-truth labels for their datasets. For instance, Liu et al. [ACM19] collected data from four java projects, i.e., Apache, Spring, Hibernate, and Google, and trained a DNN to detect inconsistent method names. In order to label their datasets, they made two rules. They considered the buggy versions of the method names and give the "inconsistent' labels. They labeled the method names in the fixed version as consistent. Some studies hired or invited the corresponding domain experts to label data by leveraging their rich experience in SE. However, we noticed that some studies didn't adopt certified methods for data labeling. A few studies gave the data labeling process using a simple description and only mentioned they manually labeled the datasets, lacking the necessary descriptions to detail the process, such as how many researchers are involved, how to generate the labels for each instance, and how they manage when meeting conflicts. Besides, other studies conducted the data labeling process selectively, i.e., they only labeled a part of instances that they randomly sampled from their datasets, and used sampled instances to train DNNs (random data labeling). Actually, the scale of data decreased by using the random data labeling method, which may reduce the performance of trained DL models. Therefore, we don't recommend studies using the "simple manual data labeling" and "random data labeling" methods to generate ground truth labels for datasets.

A significant disadvantage of manual labeling methods is high labor and time costs. To alleviate this problem, some studies utilized existing tools to simplify the data labeling process. For example, Alahmad et al. [SP08] trained a CNN model to locate code fragments from screenshots. In order to add correct labels for images, they leveraged Dataturks [3], a cloud online image annotation service, for data labeling. In order to identify and label the error code, Le et al. [ICLR04] used Joern [4], Which is a code analysis shell to automatically identify sloppy coding practices and variants of known vulnerabilities, to capture error messages by parsing the semantic and syntactical relationships

[3]https://dataturks.com
[4]http://mlsec.org/joern/

of code fragments. For the combined method for data labeling, Mi et al. [EL08] proposed the use of CovNets to improve the code readability classification. They found that source codes that violate programming guidelines, conventions, or styles are unreadable than others. Based on their findings, they selected 13 software projects to find out rule violations from source code as their datasets by using manual annotation as well as automated methods, i.e., PMD [5] and ChekStyle [6].

**Summary**

(1) We classified the data collection methods into four types, including open-source, collected, constructed, and industry datasets. Most studies used open-source datasets to train their models.
(2) We noticed that GitHub is the most commonly used SE community for collecting non-open source datasets since GitHub involves a huge volume of source code compared with other communities.
(3) 6 different data types are used in the primary studies, i.e., code-, text-, metric-, graph-, software repository-based, and mixed data types, where code-based and text-based types are the two main data types being used in about 80% of primary studies.
(4) Source code is the most essential data type for solving SE issues. Few studies conducted experiments on the image-based datasets due to lacking image-based open-source datasets in software repositories and a limited number of SE issues image-based datasets can solve.
(5) We summarized the data pre-processing procedure used in different data types. We found that several data pre-processing methods are public phases for multiple data types, such as "Data extraction", "unqualified data deletion", "duplicated instance deletion", and "data segmentation". However, in the data segmentation step, some studies were only divided into two parts: the training set and the test set, without the validation set.
(6) Most studies parse source code into the token, tree/graph structures, or extract features from programs. When the raw datasets are documentation, studies would convert them into token-based vectors as the input form of their models.
(7) We also investigated what methods primary studies used for data labeling. A manual-based method is the most commonly used for generating ground truth labels. However, some studies used the manual-based method in the wrong way.
(8) Only a few studies used an automated method for data labeling. Therefore, we suggest further studies can target different data types to propose more advanced automatical tools for labeling the ground truth in datasets.



## 5 RQ3: WHAT TYPES OF SE TASKS AND WHICH SE PHASES HAVE BEEN FACILITATED BY DL-BASED APPROACHES?

In this section, we explore how DNNs work for different SE activities and types of problems that deep learning can solve. In this section, we mainly explored how DNNs work in different SE activities (i.e., software requirement, software development, software testing, software maintenance, and software management) and problem types (i.e., regression, classification, generation, and recommendation problems). We first present the overall trend of DL-based studies for different SE activities and types of problems. Then, we performed an exhaustive analysis on usages of DL models for each activity respectively. To achieve this, we investigated relationships of DL models with respect to specific SE tasks, problem types, and data types, and also summarized the open-source datasets used in diverse SE topics.

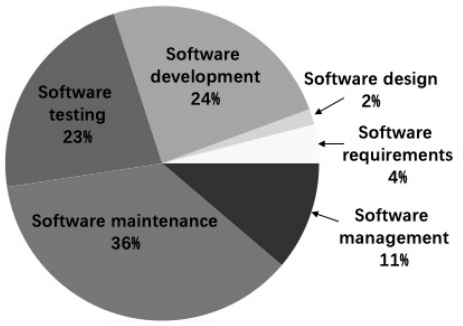



Fig. 2. The distribution of DL techniques in Different SE activities.

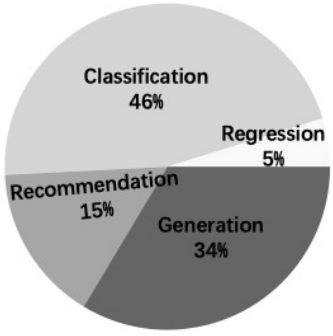



Fig. 3. The classification of primary studies.

## 5.1 What were the distributions of DL techniques over different SE activities and problem types?

We analyzed which SE activity and specific SE task each selected primary study tried to solve, and then the SE task into corresponding SE activity. As shown in Fig. 2, the largest number of primary studies focused on addressing SE issues in software maintenance (36%). 23% of studies researched software testing and debugging, and 24% of studies focused on solving SE tasks in software development. Software management was the topic of 11% of primary studies, followed by Software design and modeling (2%) and software requirements (4%). In Fig. 2, we observed that software development, testing, and maintenance are the three most important activities throughout the whole Software Development Life Cycle (SDLC).

We classified all primary studies into four categories based on the types of their SE tasks, i.e., the regression, classification, recommendation, and generation problems. Fig. 3 describes the distribution of different task types where DL techniques were applied. Classification and generation tasks account for 80% of primary studies, where classification is the most frequent task (45%). 5% of studies belong to the regression task and the output of their proposed models is a prediction value, such as effort cost prediction. In SE, some studies adopted DL to concentrate on a recommendation task, accounting for 15% of all studies. Therefore, most SE studies trained DNNs for classification and generation tasks.

## 5.2 How DNNs were used in software requirements?

In this section, we analyzed main primary studies in software requirements to present the relationships of DNNs with respect to specific SE tasks, problem types, data types, and commonly used datasets.

*5.2.1 SE tasks in software requirements.* Table **??** (See Appendix E) describes how DNNs used in different SE tasks. Among all primary studies, 10 studies contributed to solving three types of SE issues in software requirements: **requirement extraction**, **requirement traceability**, and **requirement validation**. We observed that most studies are classification tasks and they trained DNNs for identifying requirements from requirement specifications. Some of them trained DNN-based classifiers to extract requirements. For instance, Li et al. [IEEE19] defined the requirements discovery (DR) as a binary classification task and requirement annotation (RA) as a multi-label classification task. Therefore, they used Bert to generate the representation of requirement sentences and trained a Bi-LSTM classifier to identify valid requirements and give them annotations from issue reports.

[5]https://pmd.github.io/
[6]https://checkstyle.sourceforge.io/

Table 7. The open-source datasets in software requirement.

| Data type | Dataset | Reference |
|---|---|---|
| text-based data | Body Comfort System (BCS) | [IEEE118, IEEE120] |
| | IT4RE | [EL09] |
| | Danfoss | [IEEE58] |
| | MODIS | [IEEE96] |
| | CM-12 | [IEEE96] |

But other studies treat the requirement extraction task as a generation problem. For example, Li et al. [IEEE581] trained an LSTM-CRF model with the encoder-decoder architecture to generate a sequence for each requirement documentation and extracted requirement entities from the sequence.

Except for using DNNs for requirement extraction, two related studies leveraged DNNs for requirement traceability and requirement validation. To conduct automated requirements tracing, Wang et al. [IEEE96] trained an FNN model with one input layer, one output layer, and three hidden layers, to generate embeddings of requirement terms. They then fed the embeddings into a term-pair ranking model for obtaining the requirement traces. For requirement validation, the requirements specification may be subject to validation and verification procedures, ensuring that developers have understood the requirements and the requirements conform to company standards. Winkler et al. [IEEE95] present an automatic approach to identify and determine the method for requirement validation. They predefined six possible verification methods and trained a CNN model as a multiclass and multilabel classifier to classify requirements with respect to their potential verification methods. The mixed results revealed that the imperfect training data impacted the performance of their classifier, but it still achieved good results on the testing data.

*5.2.2 The open-source datasets in software requirements.* In addition, we listed the open-source datasets used in software requirements in Table 7. All datasets in this SE activity are the text-based open-source data type, i.e., requirement documentation, where BCS had been used in two primary studies. Apart from studies listed in Table 7, some studies [IEEE94, IEEE95] used industry datasets to train DNNs.

## 5.3 How DNNs were used in Software design?

Table **??** (See Appendix E) shows the usages of DNNs in software design. We noticed that both of them trained CNNs as classifiers to detect design patterns although only two studies focused on this topic. UI design is an essential component of software development, yet previous studies cannot reliably identify relevant high-fidelity UI designs from large-scale datasets. MartÃ■n et al. [IEEE129] proposed a DL-based search engine to detect UI designs in various software products. The core idea of this search engine is to build a CNN-based wireframe image autoencoder to automatically generate labels on a large-scale dataset of Android UI designs. After manual evaluation of experimental results, they confirmed that their search engine achieved superior performance compared with image-similarity-based and component-matching-based methods. Thaller et al. [IEEE109] proposed a flexible human- and machine-comprehensible software representation algorithm, namely Feature Maps. They first extracted subtrees from the system's abstract semantic graph (ASG). Then their algorithm pressed the high-dimensional and inhomogeneous vector space of these micro-structures into a feature map. Finally, they adopted a classical machine learning model and a DL model (i.e., Random Forest and CNN) to identify instances of design patterns in source code. Their evaluation suggested that Feature Map is an effective software representation method, revealing important information hidden in the source code.

We describe the way of DNNs are used in software design. There are only four studies that trained DNNs for addressing two SE issues (i.e., software design pattern detection) in software design.

Table 8. The open-source datasets in software design.

| Data type | Dataset | Reference |
|---|---|---|
| image-based data | Google Play | [ACM33, IEEE129, IEEE41] |
| text-based data | Pattern-like Micro-Architecture Repository (P-MARt) | [IEEE109] |

*5.3.1 SE tasks in software design.* For GUI modeling is a generation task to transform other materials into software design. Chen et al. [IEEE41] proposed a neural machine translator to learn a crowd-scale knowledge of user interfaces (UI). Their generative tool encoded the spatial layouts of visual features learned from a UI image and learned to generate its graphical user interface (GUI) skeleton by combining RNN and CNN models. Its performance had been verified on the large-scale UI data from real-world applications. Moran et al. [IEEE129] proposed a strategy to facilitate developers automate the process of prototyping of GUIs in 3 steps: detection, classification, and assembly. First, they used computer vision techniques to detect logical components of a GUI from mock-up metadata. They then trained CNNs to category GUI-components into domain-specific types. Finally, a KNN algorithm was applied to generate a suitable hierarchical GUI structure to assemble prototype applications. Their evaluation achieved an average GUI-component classification accuracy of 91%.

*5.3.2 The open-source datasets in software design.* In Table 8, 3 out of 4 studies addressed specific software design problems by using image-based datasets. After analysis on relevant studies, we observed that the design of mobile applications is a main research direction in software design as most of the relevant studies used GUI or application screenshots as their datasets. However, since few such open-source datasets consisting of a large number of mobile application screenshots exist in SE communities, the studies built their datasets by collecting useful screenshots from certain well-known mobile application markets, such as Google Play and Apple Store. Therefore, we summarized the common place where relevant studies collect image-based data in the first row of Table 8. Besides, we also list an open-source text-based dstaset, P-MARt, which is a repository of pattern-like micro-architecture and is commonly used in the design pattern identification task.

## 5.4 How DNNs were used in software development?

In this section, we depicted how DNNs were used in software development and analyzed the usages of different DL models combining with specific SE tasks, data types, problem types, and common datasets in software development.

*5.4.1 SE tasks in software development.* **Code representation.** Table **??** (See Appendix E) described the usages of DNNs in software development in detail and present the relationships among DNNs, task types, problem types, DL architectures, and data types. Most of the studies (13) trained DNNs to design a new code representation for better understanding the semantic of code. For instance, different from traditional information retrieval methods that treat source code as texts to generate code representation, Zhang et al. [ACM23] trained an AST-based neural network for code representation. In order to reduce the scale of ASTs, they first divided a large AST into multiple sub-ASTs, and each sub-tree represented a statement of code. After obtaining a sequence of sub-ASTs, they captured the lexical and syntactical knowledge from them and trained a Bi-LSTM to produce the vector representation of code. We observed that all studies in this SE task used source code as their datasets without the combination of text data (e.g., code comments) as auxiliary information to produce code representation. Most of the studies adopted RNN-based DNNs to process code-based data and two studies used FNN for source code modeling, which indicates that over 90& of studies considered these trees as sequences by traversing them although source code was parsed into tree structures. [ACM38] is a special work, they utilized a Graph-based neural network to learn semantic program embeddings. Specifically, they constructed the CFGs of source code and trained a GNN to learn the semantic of code from CFGs for generating code vectors.

**Code generation.** The second common SE task in software development is code generation also known as code synthesis. Code generation is a typical generation problem, and 12 primary studies used DNNs to solve it. Over half of the studies constructed their DNNs using DL architectures for code generation, significantly higher the frequency of DL architectures used in other SE tasks of software development. Among them, five studies used the encoder-decoder architecture to train their models, but only one study adopted the transformer, one of the most popular DL architectures in recent years, to produce source code. Svyatkovskiy et al. [ACM10] utilized the state-of-the-art generative transformer model trained on large-scale source code in four programming languages (i.e., Python, C#, JavaScript, and TypeScript languages) to automatically generate the syntactically correct code. [ICLR09] is another interesting study for code generation since its data type was different from most relevant studies, using code- and text-based data and their proposed DL model combined two different DNNs, i.e., LSTM and CNN. Specifically, Bune et al. [ICLR09] combined source code and code comments as their datasets, and utilized CNN to generate joint embeddings of the input-output pairs independently. After that, they trained a seq2seq model with LSTM cells to generate the syntactically correct programs. Similar to code representation, almost all relevant studies leveraged RNN-based models to analyze the semantic of code- and text-based data. Hence, we apparently noticed that studies tend to use the variants of RNN-based models when solving SE tasks since source code is the main data type in SE.

**Code comment generation.** Code comments are crucial to program comprehension. If code generation was to automatically synthesize syntactically correct code based on developers' intentions, code comment generation was to produce one or several sentences to describe the semantic of code fragments. In this SE task, studies leveraged the materials of software projects to generate code comments. For instance, Wei et al. [IEEE33] trained a BI-LSTM model with the encoder-decoder architecture to leverage the existing comments of source code as exemplars for guiding the comment generation process. Zhou et al. [EL21] first parsed source code into ASTs and extracted context information and dependencies of code by using the program analysis technique. Then they trained an RNN model to generate code comments by using contexts and dependencies of code as input.

**Code search.** There are eight studies for searching code from large-scale databases. Gu et al. [ACM02] proposed DeepAPI, a DL-based approach to generate functional API usage sequences for a given natural language-based user query by using an attention-based GRU Encoder-Decoder. DeepAPI first encoded the user query into a fixed-length context vector and produced the API sequence according to the context vector. It also enhanced their model by considering the importance of individual APIs. To evaluate its effectiveness, they empirically evaluated their approach on 7 million code snippets. Gu et al. [IEEE20] proposed a code search tool, DeepCS by using a novel DNN model. They considered code snippets as well as natural language descriptions, and then embedded them into a high-dimensional unified vector representation. Thus, DeepCS gave the relevant code snippets by retrieving the vector of the corresponding natural language query. They evaluated DeepCS with a large-scale dataset collected from GitHub.

**Code localization.** Code localization is the task to identify source code from different data types. In our primary studies, five studies localized code fragments from image-based datasets. These five studies [SP08, ACM36, IEEE140, ACM24, IEEE78] extracted programming screenshots from YouTube as their datasets and all trained CNN models to find out the code fragments from images. Therefore, we observed that there are no open-source datasets for code localization and studies are used to adopting the CNN model to process image-based data since CNN is beneficial for extracting characteristics from images throughout multiple filters.

**Code completion.** There are four studies for code completion and code localization respectively. Code completion can help developers automatically fill up the missing code snippet from software systems. Therefore, source code is their main data type and for processing code data, RNN-based models become the main DNN type they selected. Different from other studies, Liu et al. [IEEE34] pre-trained Bert, the state-of-the-art language representation model proposed by Google with a transformer-based architecture, and fine-tuned this model for code understanding and code completion.

Table 9. The open-source datasets in software development.

| Task type | Data type | common dataset | Reference |
|---|---|---|---|
| Code representation | code-based data | Java-med | [ICLR07] |
| | code-based data | eth_py150_open | [ACM18, ACM38] |
| | code-based data | Code.orgâĂŹs Hour of Code (HOC) | [ACM20] |
| Code generation | code-based data | HearthStone (HS) | [SP09, MITP06] |
| | code-based data | Karel dataset | [ICLR09, MITP08, ICLR03] |
| | code-based data | Spider | [MITP06] |
| | code-based data | DeepCom | [MITP07] |
| Code comment generation | code-based data | Google Code Jam | [AAAI08] |
| | text-based data | WMT19 | [IEEE36] |
| | code- and text-baseddata | CODEnn | [IEEE36] |
| Code search | code- and text-based data | StaQC benchmark | [IEEE02] |
| | code- and text-based data | CODEnn | [IEEE24] |
| | code- and text-based data | COsBench | [IEEE24] |
| Code completion | code-based data | JavaScript (JS) | [MK09] |
| | code-based data | Python (PY) | [MK09] |
| Code localization | image-based data | YouTube | [SP08, ACM24, ACM36, IEEE140] |
| Code summarization | code-based data | NCF representation | [ICLR06] |
| | code- and text-baseddata | LeClair et al. | [IEEE136] |
| Method name generation | code- and text-baseddata | MCC corpus | [IEEE59] |

**Code summarization and method name generation.** There is a total of seven studies that researched code summarization (5) and method name generation (2). 4 out of 5 studies used RNN-based models to generate the description of code. Different from them, Leclair et al. [EEE136] used a graph-based neural architecture to match the default structure of the AST for generating code summarization. For method name generation, Gao et al. [IEEE112] combined source code and their method name as their datasets and introduced an attention-based Encoder-Decoder framework to directly generate sensible method names by considering the relationship between the functional descriptions and method names. To evaluate their model, experiments were performed on large-scale datasets for handling the cold-start problem, and the model achieved significant improvement over baselines. Nguyen et al. [IEEE59] used code- and text-based data, including the implementation, interface, and method names to train a seq2seq model for automatically generating method names for functions.

*5.4.2 The open-source datasets in software development.* Table 9 lists the used open-source datasets in these eight SE tasks. No benchmark datasets were used in code localization since most of code localization tasks trained DNNs on the image-based data, and thus we provided the website where the datasets in relevant studies were collected. We noticed that some datasets were used for multiple SE issues. For example, A dataset named CODEnn was used in code comment generation and code search, and The dataset, Google Code Jam, is also one of the benchmark datasets for code clone detection. Some common datasets were used in multiple studies for the same task. For instance, three datasets, i.e., eth_py150_open, HearthStone (HS), and the Karel dataset are used in 2, 2, and 3 studies respectively. Among these datasets, over 60% of datasets are code-based data, about 33% of one are combined datasets, including code- and text-based data types. In addition, we present the website, YouTube, which is frequently used to crawl the image-based data for many studies.

## 5.5 How DNNs were used in software testing?

In this section, we generalized specific primary studies into different SE tasks during the activity of testing and debugging. We then summarized the datasets frequently used in these tasks.

*5.5.1 SE tasks in software testing.* Table **??** (See Appendix E) lists the studies working for software testing, and includes eight different SE issues, i.e., bug-related detection, bug localization, testing techniques, test case generation, program analysis, vulnerability detection, certification validation, and stateful service virtualization.

**Bug-related detection.** In Table **??** (See Appendix C), we noticed that bug-related tasks (i.e., bug-related detection, bug localization, and vulnerability detection) are the three most popular research directions in software testing. A number of specific SE bug-related issues were generalized in bug detection. For example, Barbez et al. [IEEE50] analyzed and mined the version control system to achieve historical values of structural code metrics. They then trained a CNN based classifier, CAME, to infer the anti-patterns in the software products. Wan et al. [IEEE68] implemented a Supervised Representation Learning Approach (SRLA) based on an autoencoder with double encoding-layers to conduct cross-project Aging-Related Bugs (ARBs) prediction. Wang et al. [IEEE69] present a novel framework, Textout, for detecting text-layout bugs in mobile apps. They formulated layout bug prediction as a classification issue and addressed this problem with image processing and deep learning techniques. Thus, they designed a specifically-tailored text detection method and trained a CNN classifier to identify text-layout bugs automatically. Textout achieved an AUC of 95.6% on the dataset with 33,102 text-region images from real-world apps. Source code is composed of different terms and identifiers written in natural language with rich semantic information.

**Bug localization.** 12 related studies used DNNs for bug localization. According to their problem types, they can be classified into three categories: six classification tasks, six recommendation tasks, and one regression task. For classification tasks, to locate buggy files, Lam et al. [IEEE40] built an autoencoder in combination with Information Retrieval (IR) technique, rVSM, which learned the relationship between the terms used in bug reports and code tokens in software projects. For regression tasks, Huo et al. [IEEE124] present a deep transform learning algorithm, TRANP-CNN, for cross-project bug localization by training a CNN model to extract transferable semantic features from source code. The output of model output their relevant scores for bug localization. For recommendation tasks, Zhang et al. [IEEE103] proposed CNNFL, which localized suspicious statements in source code responsible for failures based on CNN. They trained this model with test cases and tested it by evaluating the suspiciousness of statements. Xiao et al. [EL07] used the word-embedding technique to retain the semantic information of the bug report and source code and enhanced CNN to consider bug-fixing frequency and recency in company with feature detection techniques for bug localization.

**Vulnerability detection.** Seven studies constructed DNNs for vulnerability detection. Among them, Tian et al. [EL03] proposed to learn the fine-grained representation of binary programs and trained four different DNNs (i.e., RNN, LSTM, GRU, and BRNN) for intelligent vulnerability detection. Apart from two studies using FNNs [ICLR04, MK03], most tasks trained RNN-based and CNN models to identify vulnerabilities in software. For example, Dam et al. [IEEE131] trained an LSTM model for vulnerability detection, which automatically captured both syntactic and semantic features of source code. The experiments on 18 Android applications and Firefox applications indicated that the effectiveness of their approach for within-project prediction and cross-project prediction. Han et al. [IEEE57] trained a shallow CNN model to capture discriminative features of vulnerability description and exploit these features for predicting the multi-class severity level of software vulnerabilities. They collected large-scale data from the Common Vulnerabilities and Exposures (CVE) database to test their approach.

**Testing techniques.** Many studies focus on new methods to perform testing, such as for apps [ACM25], games [IEEE07], and other software systems [IEEE14]. There are also some studies using well-known testing techniques (e.g., fuzzing [IEEE12, ACM31] and mutation testing [IEEE65]) for improving the quality of software artifacts. Zheng et al. [IEEE07] conducted a comprehensive analysis of 1,349 real bugs and proposed Wuji, a game testing

framework, which used an FNN model to perform automatic game testing. Ben et al. [IEEE14] also used the FNN to test Advanced Driver Assistance Systems (ADAS). They leveraged a multi-objective search to guide testing towards the most critical behaviors of ADAS. Pan et al. [ACM25] present Q-testing, a reinforcement learning-based approach, benefiting from both random and model-based approaches to automated testing of Android applications. Mao et al. [IEEE65] performed an extensive study on the effectiveness and efficiency of the promising PMT technique. They also complemented the original PMT work by considering more features and the powerful deep learning models to speed up this process of generating the huge number of mutants.

**Test case generation.** Test case generation is another important SE task in software testing, and three different DNNs (i.e., FNN, RNN, and LSTM) are used to generate test cases for enhancing testing efficiency. In these studies, two studies [ACM26, IEEE17] trained DNNs into the encoder-decoder architecture. Specifically, Liu et al. [ACM26] proposed a deep natural language processing tool, DeepSQLi, to produce test cases used for detecting SQLi vulnerabilities. They trained an encoder-decoder-based seq2seq model to capture the semantic knowledge of SQLi attacks and used it to transform user inputs into new test cases. Only one study combined two different DNNs for test case generation. For example, Zhao et al. [IEEE61] trained a DL-based model that combines LSTM and FNN to learn the structures of protocol frames and deal with the temporal features of stateful protocols for carrying out security checks on industrial networks and generating fake but plausible messages as test cases. In addition, the data types for test case generation are diverse.

**Program analysis.** Some studies utilized DNNs for program analysis, attempting to find patterns or anomalies thought to reveal specific behaviors of the software. These studies mainly focused on two directions: static analysis and type inference.

For example, Due to the impact of high false-positive rates on static analysis tools, Koc et al. [IEEE64] performed a comparative empirical study of 4 learning techniques (i.e., hand-engineered features, a bag of words, RNNs, and GNNs) for classifying false positives, using multiple ground-truth program sets. Their results suggest that RNNs outperform the other studied techniques with high accuracy. For type reference, Malik et al. [ACM17] formulated the problem of inferring Javascript function types as a classification task. Thus, they trained a LSTM-based neural model to learn patterns and features from code annotated programs collected from real-world projects, and then predicted the function types of unannotated code by leveraging the learned knowledge.

**Others.** Some studies leveraged DNNs on certain special SE tasks. For example, Chen et al.[IEEE52] applied deep reinforcement learning to automated testing of certificate verification in SSL/TLS implementations. They trained a CNN model for feature extraction and an RNN model for the next action recommendation, which guided to output newly generated certificates that could trigger discrepancies with high efficiency. Enicser et al. [SP10] used LSTM to conduct stateful service virtualization for improving software testing.

*5.5.2 The open-source datasets in software testing.* Table 10 lists the datasets that have been used in software testing for addressing different research topics. No benchmark datasets were used in the SE research topic of testing techniques. This is because 4 out of 6 studies in this research topic utilized industry datasets or constructed datasets by themselves. Only 2 studies trained DNNs using collected datasets. Actually, the datasets used in bug detection, bug localization, and bug classification tasks can act as generic bases being used for other bug-related studies, where code-based datasets generally consist of several open-source projects, and the text-based datasets are composed of a number of bug reports that provide the information or diverse metrics of bugs. Therefore, there are two normal pre-processing methods for bug reports and certain similar text-based datasets. one is to treat the content in those datasets as text and then tokenize the content according to suitable granularities, such as words, sentences, or paragraphs, etc. Another method is to extract valuable features/metrics from the dataset. CVE security vulnerability database (aka. CVE) is the most well-known dataset for vulnerability detection, but another study trained the DL model on the MC&NH dataset in this SE task. We found three open-source datasets that were used for test case generation. Two of three are image-based datasets, and these two are both well-known handwritten

Table 10. The open-source datasets in software testing.

| Task type | Data type | Dataset | Reference |
|---|---|---|---|
| Bug detection | code-based data | Linux, MySQL and Apache HTTPD server | [IEEE68] |
| | code-based data | [12] includes 986 apps,578 normal apps and 408 vulnerable apps | [EL17] |
| | text-based data | bug report from Mozilla | [EL12] |
| Bug localization | code-based data | Defects4J benchmark | [IEEE40] |
| | text-based data | [13] bug report benchmark | [IEEE40] |
| | code- and text-based data | AspectJ in Bugzilla, SWT, JDT, Tomcat | [IEEE135] |
| Vulnerability detection | code-based data | MC&NH dataset | [EL03] |
| | text-based data | CVE Details websit | [IEEE57] |
| Test case generation | code-based data | REAPER | [IEEE17] |
| | image-based data | MNIST, fashion-MNIST | [IEEE93] |
| | image-based data | CIFAR-10, CIFAR-100 | [IEEE93] |
| Program analysis | text-based data | static analysis alarm data | [IEEE62] |
| bug classification | code-based data | MozillaProject, Radare2Project | [EL13] |

digital datasets. For program analysis, only one benchmark dataset is noticed, containing many descriptions on static analysis alarm data. Therefore, researchers need to contribute to constructing benchmark datasets for certain SE tasks, such as program analysis and testing techniques. But, there are some well-known and public benchmark datasets for some SE topics, such as the relationship between CVE and vulnerability detection. In addition, the scale of datasets is another important point for further study.

## 5.6 How DNNs were used in software maintenance?

There are a lot of studies contributing to increasing maintenance efficiency, such as improving source code, logging information, software energy consumption, etc. [IEEE123, IEEE125, IEEE126, IEEE56, ACM14]. We summarized the SE problems the related studies solved and classified them into different categories based on their research topics. Since over 100 studies are classified into this category, it is difficult to present them all in this section. Therefore, we listed 12 main search topics, where each topic was researched by no less than two studies. Then, we present the benchmark datasets being multiple used in these 12 topics.

*5.6.1 SE tasks in software maintenance.* **Defect prediction.** Defect prediction is the most extensive and active research topic in the use of DL techniques in software maintenance. In Table **??** (See Appendix E), we noticed that over 20% of primary studies focused on identifying defects [EL10, EL15, IEEE128, IEEE114, IEEE74, EL06]. These studies for defect prediction can be further classified into three categories: metric-based, semantic-based, and Just-In-Time(JIT) defect prediction.

Metrics or features extracted from a software product can give a vivid description of its running state, and thus it is easy for researchers and participants to use these software metrics for defect prediction. Xu et al. [EL15] built an FNN model with a new hybrid loss function to learn the intrinsic structure and more discriminative features hidden behind the programs. Previous studies obtained process metrics throughout analyzing change histories manually and often ignored the sequence information of changes during software evaluation. Liu et al. [IEEE114] proposed to obtain the Historical Version Sequence of Metrics (HVSM) from various software versions as defect predictors and leveraged RNN to detect defects.

For semantic-based defect prediction, Wang et al. [IEEE43, IEEE127] leveraged Deep Belief Network (DBN) to automatically learn semantic features from token vectors extracted from programs' ASTs, compared to most previous works that use manual feature specification. They evaluated their approach on file-level defect prediction tasks (within-project and cross-project) and change-level defect prediction tasks (within-project and cross-project) respectively. Similarly, Dam et al. [IEEE74] used a tree-based LSTM network, which can directly match with the AST of programs for capturing multiple levels of the semantics of source code.

For Just-In-Time (JIT) defect prediction, Hoang et al. [IEEE73] presented an end-to-end DL-based framework, DeepJIT, for change-level defect prediction, or Just-In-Time (JIT) defect prediction. DeepJIT automatically extracted features from code changes and commit messages, and trained a CNN model to analyze them for defect prediction. The evaluation experiments on two popular projects showed that DeepJIT achieved improvements over 10% for two open-source datasets in terms of AUC.

**Program repair.** Apart from defect prediction, diverse DNNs were frequently applied for repairing incorrect programs (13). For example, Bhatia et al. [IEEE42] proposed a novel neuro-symbolic approach combining DL techniques with constraint-based reasoning for automatically correcting programs with errors. Specifically, they trained an RNN model to perform syntax repairs for the buggy programs and ensured functional correctness by using constraint-based techniques. Tufano et al. [ACM35] proposed to leverage the proliferation of software development histories to fix common programming bugs. They used the Encoder-Decoder framework to translate buggy code into its fixed version after generating the abstract representation of buggy programs and their fixed code. White et al. [IEEE105] trained an autoencoder framework to reason about the repair ingredients (i.e., the code reused to craft a patch). They prioritized and transformed suspicious statements and elements in the code repository for patch generation by calculating code similarities. Lutellier et al. [ACM27] present a new automated generate-and-validate program repair approach, CoCoNuT, which trained multiple models to extract hierarchical features and model source code at different granularity levels (e.g., statement and function level) and then constructed a CNN model to fix different program bugs.

**Code clone detection.** Code clone detection is one of the most important aspects to assess software quality and reusability. 11 primary studies trained DL models for detecting code clones. In Table **??** (See Appendix E), we observed that the RNN-based model is the first choice for these studies to detect code clones, and the code-based dataset is the main data type for clone detection. Most studies use RNNs including RNN [IEEE51], RvNN [IEEE15], and LSTM [IEEE110, IEEE77] to identify clones in source code. White et al. [IEEE15] proposed a novel code clone detector by combining two different RNNs, i.e., RtNN and RvNN, for automatically linking patterns mined at the lexical level with patterns mined at the syntactic level. They evaluated their DL-based approach based on file- and function-level. Gao et al. [IEEE51] first transformed source code into AST by parsing programs and then adopted a skip-gram language model to generate vector representation of ASTs. After that, they used the standard RNN model to find code clones from java projects. Buch et al. [IEEE110] introduced a tree-based code clone detection approach, and traversed ASTs to form data sequences as the input of LSTM. Perez et al. [IEEE77] also used LSTM to learn from ASTs, and then calculated the similarities between ASTs written in Java and Python for identifying cross-language clones. Since source code can be represented at different levels of abstraction: identifiers, Abstract Syntax Trees, Control Flow Graphs, and Bytecode. One study treated code clone detection as a regression task, calculating the similarities between code fragments. For example, Tufano et al. [IEEE80] conducted a series of experiments to demonstrate how DL can automatically learn code similarities from different representations.

**Bug report related.** The bug report is one of the important crucial SE documentation for software maintenance, and bug reports with high quality can effectively reduce the cost of fixing bug programs. Six studies focused on improving the quality of bug reports, mainly including three research directions: bug report summarization [IEEE39, IEEE137], bug report recommendation [SP02], and valid bug report detection [IEEE142, IEEE64]. For example, Li et al. [IEEE39] performed the first exploration on bug report summarization by applying DL techniques. They proposed a new DL-based framework, called Deepsum, which used a stepped auto-encoder to integrate the features

of bug reports into the auto-encoder network. This framework was an unsupervised DNN, reducing the effort on labeling datasets. Choetkiertikul et al. [SP02] trained an LSTM model as a predictive model to automatically learn semantic features of bug reports and recommended the most relevant code and solutions for new issues. He et al. [IEEE142] trained a CNN model to determine the valid bug reports by capturing the contextual and semantic features of these reports.

**Software quality evaluation.** Some primary studies proposed DL-based models to evaluate software quality in terms of multiple perspectives, such as traceability, readability, reliability, maintainability, and trustworthiness. For example, Guo et al. [ACM21] present a solution to incorporate requirements artifact semantics and domain knowledge into the tracing solution for avoiding misunderstanding of software artifacts and delivering imprecise and inaccurate results. They implemented a tracing neural network architecture to generate trace links by using word Embedding and various RNN models. They evaluated their approach on 360 different configurations of the tracing network and found that BI-GRU was the best DNN model for the tracing task. Mi et al. [EL08] proposed to leverage CNN to improve code readability classification. First, they present a transformation strategy to generate integer matrices as the input of ConvNets. Then they trained Deep CRM, a DL-based model, which was made up of three separate ConvNets with identical architectures for code readability classification. For software maintainability and trustworthiness, Kumar et al. [EL18] performed two case studies and applied three DNNs i.e., FLANN-Genetic (FGA and AFGA), FLANN-PSO (FPSO and MFPSO), FLANN-CSA (FCSA), to design a model for predicting maintainability. They also evaluated the effectiveness of feature reduction techniques for predicting maintainability. The experimental result showed that feature reduction techniques can achieve better results compared with using DNNs. It is essential and necessary to evaluate software trustworthiness based on the influence degrees of different software behaviors for minimizing the interference of human factors. Tian et al. [EL04] constructed behavior trajectory matrices to represent the behavior trajectory and then trained the deep residual network (ResNet) as a software trustworthiness evaluation model to classify the current software behaviors. After that, they used the cosine similarity algorithm to calculate the deviation degree of the software behavior trajectory.

**Compiled-related.** Four studies focused on addressing two specific problems (i.e., compiled code generation and compiler optimization) in software compilation. These two studies [IEEE06, IEEE113] proposed DL-based models to generate decompiled source code and identifiers, and other studies [IEEE67, IEEE91] recommended optimization option sequences during compilation. The data type of their work is source code and 3 out of 4 studies adopted RNN-based models as their solutions.

**SATD detection and code smell detection.** Technical debt (TD) is a metaphor to reflect the trade-off developers make between short-term benefits and long-term stability. Self-admitted technical debt (SATD), a variant of TD, has been proposed to identify debt that is intentionally introduced during SDLC. Ren et al. [ACM34] proposed a CNN-based approach to determine code comments as SATD or non-SATD. They exploited the computational structure of CNNs to identify key phrases and patterns in code comments that are most relevant to SATD for improving the explainability of our modelâĂŹs prediction results. Zampetti et al. [IEEE100] proposed to automatically recommend SATD removal strategies by building a multi-level classifier on a curated dataset of SATD removal patterns. Their strategy was capable of recommending six SATD removal patterns, i.e., changing API calls, conditionals, method signatures, exception handling, return statements, or telling that a more complex change is needed.

**Code review and software/code classification.** For code review, Siow et al. [IEEE99] believed that the hinge of the accurate code review suggestion is to learn good representations for both code changes and reviews. Therefore, they designed a multi-level embedding framework to represent the semantics provided by code changes and reviews and then well-trained through an attention-based deep learning model, CORE. Guo et al. [IEEE106] proposed Deep Review Sharing, a new technique based on code clone detection for accurate review sharing among similar software projects, and optimized their technique by a series of operations such as heuristic filtering and review

deduplication. They evaluated Deep Review Sharing on hundreds of real code segments and it won considerable positive approvals by experts, illustrating its effectiveness.

For software/code classification, Bui et al. [IEEE104] described a framework of Bi-NN that built a neural network on top of two underlying sub-networks, each of which encoded syntax and semantics of code in a language. Bi-NN was trained with bilateral programs that implement the same algorithms and/or data structures in different languages and then be applied to recognize algorithm classes across languages. Software categorization is the task of organizing software into groups that broadly describe the behavior of the software. However, previous studies suffered very large performance penalties when classifying source code and code comments only. Leclair et al. [IEEE53] proposed a set of adaptations to a state-of-the-art neural classification algorithm and conducted two evaluations. In addition, we noticed that most studies in terms of these two SE tasks used code-based datasets to train different DNNs as classifiers for solving specific issues.

**Code change and incident detection.** For code change-related tasks, Tufano et al. [ACM15] trained an RNN model with an encoder-decoder architecture to learning code changes from pull requests. Hoang et al. [IEEE45] used the HAN model to learn from log messages for generating the suitable representation of code changes.

For incident detection. Chen et al. [ACM08] selected multiple models to identify linked incidents from all incidents collected from 465 different services. Chen et al. [IEEE22] trained an attention-based CNN model to characterize and prioritize incidents for large-scale online service systems.

**Others.** Apart from the studies concentrating on these 12 SE tasks, a number of studies applied DNNs on other research topics for improving software quality. For example, Romansky et al. [IEEE56] used an LSTM model to learn from features of applications to estimate the software energy consumption. Hoang et al. [IEEE123] trained a CNN-based model on patches that have been committed to mainline Linux kernel and detect stable patches from them.

*5.6.2 The open-source datasets in software maintenance.* Table 11 depicts the open-source datasets used for diverse SE tasks in software maintenance, where there are four SE tasks that no benchmark datasets were applied in, i.e., software quality evaluation, compiled-related, code smell detection, and incident detection. Open-source and benchmark datasets are applied in 8 different research directions. In defect prediction, the PROMISE dataset was used in five studies. Two studies for defect prediction utilized a series of open-source software systems as their datasets. There are six open-source datasets that can be used to repair bug programs, including four code-based and two combined data types. Defects4J, Bugs.jar, DeepFix datasets were used in multiple studies.

Three well-known benchmark datasets are built for code clone detection, where BigCloneBench is the most widely used dataset applied in nine different studies. Each dataset contains a large number of function pairs in different clone types. The BigCloneBench and OJClone are two benchmark datasets in Java, and GoogleCodeJam is the dataset in the C language. Three datasets have been used for bug report-related studies, and they contain information on bug reports. Hence, two different ways to process these datasets. One is to extract useful metrics from bug reports and another way is just to treat bug reports as pure texts in data pre-processing before further analysis. There are two open-source datasets for SADT detection, and these two datasets are also constructed and published by prior work. In all primary studies, only one open-source dataset exists in the field of code review, code change, and software/code classification.

## 5.7 How were DNNs used in software management?

Software management involves a series of SE tasks that can be classified into two categories: effort cost prediction and software repository mining. Actually, there are five sub-directions in software repository mining, i.e., mining GitHub, ming Stack Overflow (SO), app mining, tag mining, and developer-based mining.

Table 11. The open-source datasets in software maintenance.

| Task type | Data type | Dataset | Reference |
|---|---|---|---|
| defect prediction | code-based data | PROMISE dataset | [SP04, IEEE46, IEEE43, IEEE85, IEEE127] |
| | code-based data | Ivy, jEdit, Log4j, Lucene, PBeans, POI, Synapse, Velocity, Xalan-J, Xerces | [IEEE87, IEEE92] |
| | code-based data | cleaned NASA , AEEEM datasets | [SP04] |
| | code-based data | Bugzilla, Columba, Eclipse JDT, Eclipse Platform, Mozilla and PostgreSQL | [IEEE84] |
| | text-based data | CSIC dataset | [IEEE30] |
| program repair | code-based data | DeepFix dataset | [AAAI03, IEEE138] |
| | code-based data | SATE IV | [MITP01] |
| | code-based data | ETH-Py1502, MSR-VarMisuse | [ICLR02] |
| | code-based data | SPoC dataset | [ACM11] |
| | code- and text-based data | Bears, QuixBugs, ManySStuBs4J | [IEEE29] |
| | code- and text-based data | Bugs.jar | [IEEE29, IEEE46] |
| | code- and text-based data | Defects4J | [IEEE29, IEEE46, IEEE132] |
| code clone detection | code-based data | BigCloneBench | [AAAI04, ACM04, IEEE27, MK04, MK05, IEEE110, IEEE38, IEEE51, IEEE55] |
| | code-based data | OJClone | [AAAI04, MK04, MK05, ACM28, IEEE38] |
| | code-based data | Google Code Jam | [ACM04, IEEE27] |
| bug report related | text-based data | Summary Dataset(SDS) | [IEEE137, IEEE39] |
| | text-based data | Authorship Dataset (ADS) | [IEEE137, IEEE39] |
| | text-based data | OpenOffice, Eclipse, Net Beans | [IEEE139] |
| SADT detection | text-based | [14] | [IEEE100] |
| | text-based data | [1] | [IEEE28] |
| code review | code-based data | cloudstack, ambari,aurora, drillgit, accumulo and hbase-git. | [AAAI05] |
| code change | text-based data | QT, OPENSTACK datasets | [IEEE45] |
| software/code classification | code-based data | [AAAI02] | [AAAI02] |

*5.7.1 SE tasks in software management.* **Effort cost prediction.** Since only 39% of software projects are finished and published on time relative to the duration planned originally [EL19, IET01], it is necessary to assess the development cost to achieve reliable software within development schedule and budget. Table **??** (See Appendix E) shows the relationships between DNNs used for effort cost prediction and several factors that include task types, problem types, DL architecture, and data types. For example, Lopez et al. [EL19] compared three different neural network models for effort estimation. The experimental result demonstrated that MLP and RBFNN can achieve higher accuracy than the MLR model. Choetkiertiku et al. [IEEE134] observed that few studies focused on estimating effort cost in agile projects, and thus they proposed a DL-based model for predicting development cost based on combining two powerful DL techniques: LSTM and recurrent highway network (RHN). Phannachitta et al. [EL01] conducted an empirical study to revisit the systematic comparison of heuristics-based and learning-based software effort estimators on 13 standard benchmark datasets. Ochodek et al [EL02] employed several DNNs (i.e., CNN, RNN, Convolutional + Recurrent Neural Network (CRNN)) to design a novel prediction model, and

compared the performance of the DL-based model with three state-of-the-art approaches: AUC, AUCG, and BN-UCGAIN. They noticed that CNN obtained the best prediction accuracy among all software effort adaptors.

Effort cost prediction is a vital regression task for software management. Table **??** (See Appendix E) shows that two data types were involved in this task, i.e., text- and metric-based data, and studies often applied FNN-based models to learn metric-based datasets in this task.

**Mining GitHub.** GitHub, as one of the most popular and widely used SE communities, involves the huge scale of source code and other forms of information (e.g., pull requests, commit messages, and issues) to help developers build and design software. According to different contents collected from GitHub, three interesting SE tasks arise in 6 primary studies. Among them, three studies worked for commit message generation [MK02, IEEE11, IEEE75], and two studies contributed to issue-commit link recovery [EL14, IEEE107]. Only one primary study focused on pull request description generation [IEEE04]. For instance, Xie et al. [IEEE107] proposed a new approach to recover issue-commit links. They constructed the code knowledge graph of a code repository and captured the semantics of issue- or commit-related text by generating embeddings of source code files. Then they trained a DNN model to calculate code similarity and semantic similarity using additional features. Ruan et al. [EL14] propose a novel semantically-enhanced link recovery method, DeepLink, using DL techniques. They applied word embedding and RNN to implement a DNN architecture to learn the semantic representation of code and texts as well as the semantic correlation between issues and commits. Liu et al. [IEEE04] proposed a DL-based approach to automatically generate pull request descriptions based on the commit messages and the added source code comments in pull requests. They formulated this problem as a text summarization problem and solved it, constructing an attentional encoder-decoder model with a pointer generator.

Each of the studies related to mining GitHub in Table **??** (See Appendix E) trained an RNN-based model as a generator for solving these three SE issues. All of them were used text-based data as the whole or a part of their datasets, where 4 out of 6 studies combined text data with source code. Throughout analysis on the relationships between these factors, studies tend to utilized RNN-based models for generation tasks, and the RNN models are better at learning something from text-based data.

**Mining SO and app mining.** Similar to mining GitHub, a number of studies adopted diverse DNNs to mine Stack Overflow (SO) and applications in Google Play. For mining SO, Chen et al. [IEEE16] also applied word embeddings and the CNN model to mine SO for retrieving cross-lingual questions. They compared their approach with other translation-based methods, and the final results showed that their approach can significantly outperform baselines and can also be extended to dual-language document retrieval from different sources. Yin et al. [IEEE79] proposed a new approach to pair the title of a question with the code in the accepted answer by mining high-quality aligned data from SO using two sets of features.

Except for mining from SO, several studies employed DL techniques to mine constructive contents from the application store. For example, Gao et al. [IEEE03] treated a review and its response descriptions as a unit and collected the units as their datasets to train an RNN model with an encoder-decoder architecture for generating the response for a new review. Guo et al. [IEEE111] used reviews in Google Play as their datasets and predefined a set of software metrics. They then pre-processed their text-based data into metrics and trained a CNN model to classify reviews based on their metric values.

**Tag mining and developer-based mining.** There are four tag-related studies, involving two recommendation tasks, one classification task, and one generation task. Among them, one empirical study [EL05] investigated whether deep learning is better than traditional approaches in tag recommendation for software information sites. They implemented four different DL-based approaches, i.e., TagCNN, TagRNN, TagHAN, and TagRCNN, and three traditional approaches, i.e., EnTagRec, TagMulRec, and FastTagRec, for evaluating their performance in tag recommendation. Their experimental results showed that the performance of these four DL-based models varies significantly, where TagCNN and TagHAN achieved worse performance than traditional approaches. But, the

Table 12. The open-source datasets in software management.

| Task type | Data type | common dataset | Reference |
|---|---|---|---|
| effort cost prediction | metric-based data | ISBSG | [EL19] |
| | metric-based data | COCOMO datase | [IET01] |
| GitHub mining | text-based | Jiang and McMillan | [MK02] |
| mining tag | text-based | McGill and Android tutorial datasets | [EL05] |



performance of two CNN-based models is better than traditional approaches in tag recommendation tasks. Thus, selecting appropriate DNNs for different SE tasks can achieve better performance than traditional approaches.

Some studies used DL-based approaches to mine developers' information. For example, Huang et al. [IEEE130] proposed a new model to classify sentences from issue reports of four projects in GitHub. They constructed a dataset collecting 5,408 sentences and refined previous categories (i.e., feature request, problem discovery, information seeking, information giving, and others). They then trained a CNN-based model to automatically classify sentences into different categories of intentions and used the batch normalization and automatic hyperparameter tuning approach to optimize their model. Zhang et al. [SP06] present a meta-learning approach to recommend suitable top $k$ developers with the highest possibility of finishing the corresponding tasks published in the competition-based crowdsourcing Software Development (CSD) platform.

*5.7.2 The open-source datasets in software management.* Table 12 shows several open-source datasets that were used in software management. We observed that two benchmark datasets are in the effort cost prediction and they are both metric-based datasets. There are only two open-source datasets used in data mining tasks, i.e., GitHub mining and mining tag. One potential reason is that most data mining tasks crawl appropriate contents from different SE communities, and thus few open-source datasets are in software repository mining.

Summary

(1) We grouped six SE activities based on the body knowledge of SE – Software requirements, Software design and modeling, Software implementation, Software testing and debugging, Software maintenance, and Software management – and provided an outline of the application trend of DL techniques among these SE activities.
(2) We summarized various SE tasks into four categories – regression task, classification task, recommendation task, and generation task – and classified all primary studies based on the task types.
(3) Software testing and software maintenance are the two SE activities containing the most related studies and include 21 specific SE research topics in which DL techniques were used.
(4) We discussed the relationships between DNNs, DL architectures, problem types, and data types. We found that most of the encoder-decoder and autoencoder architectures were used in generation tasks.
(5) We noticed that the DL-based model selection has a close relationship between the data input forms. Studies often adopted RNN-based models to train sequence data including text-based and a part of code-based data, and used CNN models to extract features from code-based data in tree and graph structures as well as image-based data.
(6) Text- and code-based datasets are the most common data types in different SE tasks.

# 6 RQ4: WHAT TECHNIQUES ARE USED TO OPTIMIZE AND EVALUATE DL-BASED MODELS IN SE?

In the training phase, developers attempt to optimize the models in different ways for achieving good performance. In this section, we summarized the information describing the optimization methods and evaluation process, and performed an analysis on key techniques.

## 6.1 What learning algorithms were used in order to optimize the models?

The performance of DL-based models is dependent on selected optimization methods, which can systematically adjust the parameters and learning rates of the DNN as training progresses.

We identified the optimization algorithms in primary studies and separated them into two categories. One category consists of the optimizers for the way DNNs converge and adjust parameters. Another category includes optimizers that can adjust the learning rate. Fig 4 illustrates the frequency of the use of parameter adjustment and learning rate optimization methods used in primary studies. We see that from Fig. 4(a), Gradient descent is one of the preferred ways to optimize neural networks and many other machine learning methods, which was used by 15 different primary studies [AAAI03, IEEE03, IEEE33, ACM22, IEEE50]. Besides, its six different gradient descent variants, including SGD, mini-batch GD, policy GD, gradient clipping, conjugate gradient, and scaled gradient, are also widely used in primary studies, where SGD is the most popular optimizer selected by over 30 studies [AAAI05, AAAI07, AAAI08, ACM05, IEEE19, SP05, ACM10, ICLR11, MK07, MK08, IEEE85, ACM19]. Except for gradient-based optimizers, five other types of parameter optimization algorithms were used in over 20 studies, where 11 studies mentioned they used fine-tuning [IEEE44, IEEE97, ACM27, IEEE97] to optimize their parameter values, followed by the Hyperparameter optimization algorithm [IEEE50, IEEE81, IEEE83]. There are seven studies that selected PSO, L-BFGS, and Bayesian optimization algorithms to tune their parameters.

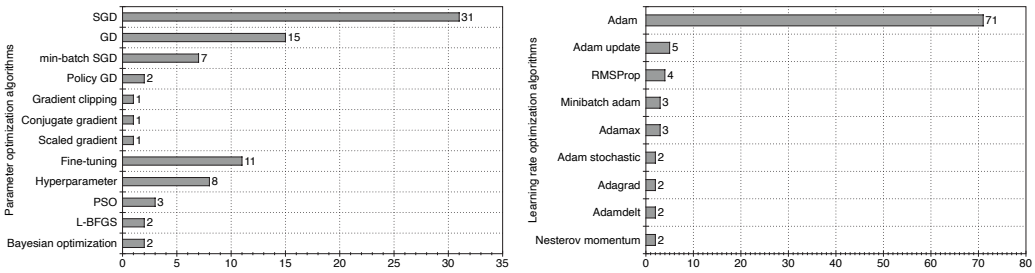
Apart from tuning parameters in neural networks, choosing a proper learning rate can be also important to improve the performance of DL-based models. Fig. 4(b) shows the distribution of the usage of learning rate optimization algorithms. We noticed that Adam was apploed in a majority of studies [AAAI01, IEEE02, IEEE18, IEEE24, IEEE28, IEEE30, IEEE32, SP02, IEEE34, IEEE35, ICLR01, SP09, ICLR03, ICLR04, ICLR05, ICLR06, ICLR08, ACM11, ACM12] and its use frequency is significantly higher than other optimizers. Several variants of Adam were also used in some primary studies, such as Adam update [IEEE57], Adagrad [IEEE65], Adadelt [IEEE11], minibatch [IEEE03], Adam stochastic [ACM10], and Adamax [IEEE91]. In addition, RMSProp [IEEE39, IEEE77] is also a common used to optimize the learning rate, followed by Nesterov Momentum [ICLR07].

Throughout the analysis on parameter and learning rate optimization algorithms, we noticed that the studies using the same optimization algorithm distributed in different SE activities and problem types, i.e., some of the studies focused on classification problems in software testing and others addressed the recommendation tasks in software maintenance. And studies applying different kinds of DNNs can use the same optimization algorithm. Therefore, the optimization algorithm selection is independent of DNN model selection, task types, and problem types.

## 6.2 What methods were used to alleviate the impact of Overfitting?

One major problem associated with applying any type of learning algorithm is overfitting. Overfitting is the phenomenon of a DNN learning to fit the noise in the training data extremely well, yet not being able to generalize to unseen data, which is not a good approximation of the true target function that the algorithm is looking forward to learn. We summarized six general methods that used in primary studies to combat overfitting.

Table. **??** (See Appendix F) illustrates the distribution of the techniques used for combating overfitting problems. Dropout has been used frequently among the selected studies to prevent overfitting; it is used in 47 studies, followed by the pooling method (26). Regularization, including L1 and L2 methods, is the third most popular technique




(a) Various parameter optimization algorithms used in primary studies.

Fig. 4. Distribution of optimizers used in primary studies.

used in 24 studies. There are 9 studies that prevent overfitting by enlarging the scale of data, such as using a large-scale dataset, combining multiple datasets, and using different data augmentation techniques. 4 studies used early stopping. In order to data balancing, 5 studies reduced data to combat the overfitting problem. Furthermore, among all primary studies, some studies used other algorithms proposed by some studies to solve overfitting. We analyzed which factors may have an impact on the overfitting technique selection. We noticed that the techniques used for combating overfitting have no strong association with either data types or input forms. However, there is a special relationship between model selection and these techniques. Most of the studies that adopted CNNs to address specific SE tasks selected pooling as their first choice for preventing the overfitting problem.

There are several findings during the analysis on overfitting techniques. First, Dropout method is the most commonly adopted method to combat overfitting. Second, Regularization mainly consists of two specific methods, i.e., L1 and L2 regularization. Most studies employed L2 methods compared with L1 regularization methods. A majority of studies using CNNs adopted the Pooling layer to select main features for combating overfitting. Besides, except for pooling, adoption for other overfitting methods has no close relationship with the variety of used DL models.

## 6.3 What measures are used to evaluate DL-based models?

Accessing appropriate benchmarks is a crucial part of evaluating any DL-based models in SE. We also explored the frequent metrics used to measure the performance of DL-based models applied to respective SE tasks. In fact, the evaluation measure adoption is related to the problem types of different SE tasks. For example, the metric, BLEU score, is often used to evaluate the performance for generation tasks and MRR is one of the evaluation metrics for recommendation tasks. Therefore, we summarized and classified the common evaluation metrics based on different problem types, i.e., regression, classification, recommendation, and generation.

Table **??** (See Appendix F) summarizes the commonly used evaluation metrics in different problem types, used in no less than 2 studies. Only four regression evaluation metrics were used in primary studies due to few studies using DNNs for regression problems. Precision, recall, and F1-score are the three most commonly used evaluation metrics for classification tasks, and all of them were used in over 60 studies. In classification tasks, the second tier of evaluation metrics is composed of F1-score and accuracy, which were both used over 20 times. MCC is a metric for evaluating the quality of binary classifications, used in 8 different studies, followed by the ROC metric. Several studies used simple metrics, such as true positive rate, false-positive rate, and false-negative rate.

There are five frequently used metrics for commendation tasks. MRR and MAP are common evaluation metrics in information retrieval, and they were used in 17 and 11 recommendation tasks respectively. Besides, for ranking/recommendation tasks, researchers often estimated the performance of the top $k$ ranking or recommendation results in terms of precision, recall, and F1-score, i.e., precision@k, recall@k, and F1-score@k.

Several special evaluation metrics for language tasks appeared in the generation problem, such as BLEU and ROUGE. BLEU is a frequently used metric for language translation, but it is also a usual metric in seq2seq tasks. For example, some comment generation tasks [IEEE11, IEEE33, IEEE36, SP07] trained DNNs to learn semantic information from source code and comments in software systems for generating comments for unseen code fragments. It is because a code comment generation task can be also treated as a language translation task, i.e., using DDNs translates source code into natural language. For the similar reasons, BLEU were also used in code summarization [IEEE31, IEEE136, ACM16], commit message generation [IEEE75, IEEE33], and code generation tasks [SP09, MITP07, MITP06]. ROUGE is another language metric, which is often used for summarization tasks, including bug report summarization [IEEE137, IEEE39], code summarization [IEEE136, IEEE31], etc. The Exact match metric is the most straightforward evaluation metric, measuring the similarity between the generated sequences and the ground truth, which was used in seven studies, followed by the Running time (6). METEOR and PP are also two metrics that are commonly used for generation tasks, and there are 5 and 2 studies that used them respectively.

### 6.4 Accessibility of DL-based models used in primary studies.

The replicability and reproducibility of DL applications have a great impact on the transfer of research results into industry practices. According to the ACM policy on artifact review and badging, replicability refers to the ability of an independent group to obtain the same result using the authorâĂŹs own artifacts. Likewise, reproducibility is the act of obtaining the same result without the use of original artifacts (but generally using the same methods). Reproducibility is clearly the ultimate goal, but replicability is an intermediate step to promote practices. Somewhat unfortunately, according to Fu and Menzies [3], it is hard to replicate or reproduce DL applications from SE research due to the nondisclosure of datasets and source code.

We checked whether the source code of DL-based models is accessible for supporting replicability and reproducibility. **101** studies provided the replication packages of their DL-based models, only accounting for **40.4%** of all primary studies. Other studies proposed novel DL-based models without publicly available source code, making it difficult for other researchers to reproduce their results; some of these studies only disclosed their datasets. Based on this observation, obtaining open-source code of DNNs is still one of the challenges in SE because many factors may result in never realizing the replicability and reproducibility of DL application, e.g., data accessibility, source code accessibility, different data preprocessing techniques, optimization methods, and model selection methods. Therefore, we recommend future DL studies to release replication packages.

Summary

(1) We summarized widely used parameter and learning rate optimization algorithms in DNNs. We noticed that most of studies used SGD and Adam as their parameter and learning rate optimizers.
(2) We noticed that optimization algorithm adoption is not affected by other factors, including DNNs, data types, task types, and problem types.
(3) We summarized the use frequency of techniques that combat the overfitting problem in primary studies, and found that Pooling was comonly used in CNNs and other DL models are used to adopting dropout technique.
(4) We summarized frequently used evaluation metrics in different problem types, i.e., regression, classification, recommendation, and generation. 10 different evaluation metrics were used in classification tasks, followed by generation tasks (9).

## 7 LIMITATIONS

**Data Extraction.** There are several potential limitations to our work. One limitation is the potential bias in data collection. Although we have listed the data items used for analysis in RQs in Section 3.4, some disagreements still

appeared inevitably when extracting related content and classifying these data items. Two researchers first recorded the relevant descriptions in each primary study and then discussed and generalized temporary categories based on all data in one item by comparing the objectives and contributions with related studies. If they were unable to reach an agreement on classification, another researcher would join in and resolve differences, which can mitigate the bias in data extraction to study validity.

**Study Selection Bias.** Another threat might be the possibility of missing DL related studies during the literature search and selection phase. We are unable to identify and retrieve all relevant publications considering the many publication venues publishing SE relevant papers. Therefore, we carefully selected 21 publication venues, including conference proceedings, symposiums, and journals, which can cover many prior works in SE. Besides, we identified our search terms and formed a search string by combining and amending the search strings used in other literature reviews on DL. These could keep the number of missed primary studies as small as possible.

## 8 CHALLENGES AND OPPORTUNITIES

**Challenge 1: The generalizability for DNNs.** In section 5, when analyzing the studies related to code clone detection, we found that several open-source public data sets are often used repeatedly in these studies to evaluate their proposed models. A similar situation also exists in other research topics. These highlight the dependence on some studies on large publicly available labelled datasets. One reason is that training a DNN requires a massive volume of data, but copious amounts of training data are rarely available in most SE tasks. Besides, it is impossible to give every possible labeled sample of a problem space to a DL algorithm. Therefore, it will have to generalize or interpolate between its previous samples in order to classify data it has never seen before. It is a challenge to tackle the problem that DL techniques currently lack a mechanism for learning abstractions through explicit, verbal definition and only work best with thousands, millions, or even billions of training examples. One solution is to construct widely accepted datasets by using industrial labeled data or crawling software repositories to collect related data samples and label them as public datasets. Another is to develop new DL techniques, which can learn how to learn and be trained as an effective model with as little data size as possible, such as few shot learning.

**Challenge 2: The transparency of DL.** In this study, we discussed 250 studies that used DL to address various SE issues. We noticed that few studies declared the reason for the architecture they chose and explained the necessity and value of each layer in DNN, which leads to low transparency of the proposed DL solutions. Because it is inherently difficult to comprehend what drives the decisions of the DNN due to the black-box nature of DL. Humans only pay attention to the output of DNNs since they can provide wise and actionable suggestions for humans. Furthermore, DL algorithms sift through millions of data points to find patterns and correlations that often go unnoticed by human experts. The decision they make based on these findings often confound even the engineers who created them. For example, many studies present novel DL-based models for defect prediction, but it is difficult to apply them in the practical use. Since developers cannot be convinced by the DL-based predictive models to believe that there are potential defects in the code that functions well. New methods and studies on explaining the decision-making process of DNNs should be an active research direction, which facilitates software engineers to design more effective DNN architectures for specific SE problems.

**Challenge 3: Validity of Parameters.** Many internal parameters are required for DNN construction, such as batch size, stop condition, optimizers, learning rate, etc. Dl-related studies have usually initialized these parameters with default settings and do not verify the effectiveness of this parameter choice. They have also sometimes tuned the parameters without clarifying the reasons or validity.

**Challenge 4: Limited Performance Measures.** Although researchers used diverse performance metrics to measure the performance of DNN models, as listed in Table **??**, some evaluation aspects are not fully considered by most related studies.

1) ***Efficiency.*** Most SE tasks have a high demand for the efficiency of DNN models, i.e., these tasks require getting correct results as fast as possible, such as code clone detection, code completion, code search. However, most primary studies did not estimate the performance of DNNs in terms of running/training time. Especially, DL-based models often have more time cost, which may lead to the time cost being not acceptable even if these models achieve high performance in terms of other evaluation metrics.

2) ***Scalability.*** Most DL-based studies adopted limited datasets to train and test their DNNs, which causes many DNN models can only achieve high performance on used datasets. Therefore, it is also necessary to evaluate the scalability of DNNs proposed on different scales of datasets.

**Opportunity 1: Using DL in more SE activities.** In Section 5, we found that DL has been widely used in certain SE topics, such as defect prediction, code clone detection, software repository mining, etc. However, few studies used DL for some SE research topics compared with other techniques or other learning algorithms. Although Software requirements and software design are the most two important documentations during SDLC, not many studies focus on these two SE activities. Therefore, one potential opportunity is that researchers can utilize DL techniques to explore new research topics or pay attention to classical topics in software requirements and design.

**Opportunity 2: The new input form of natural language.** In Table 5, we noticed that for the natural language, the input form only exists the "text in token", but without other forms, such as "text in tree structure". Only code can be transformed in tree or graph structures. Single input form for natural language narrows the scope of model selection. Therefore, in Table **??**, we can see that the majority of studies selected RNN- or FNN-based models to analyze requirement documentation. Actually, natural language can be also pared into the tree structure based on the lexical structure of sentences, and using the tree structure to represent the texts may enable using CNN- or GNN-based models to train texts possible. Besides, using syntax trees to represent natural language can better describe structures rather than identifiers.

**Opportunity 3: Comparison study on code embedding in DL.** In Section 5.4, we noticed that several studies utilized DL-based models to propose new code embeddings. But, unlike BRET being well-known in NLP, no one of these similar embedding approaches becomes a widely used embedding approach for source code, and few studies investigate their characteristics. We think there are two potential reasons for this phenomenon. First, the code representation methods lack extensive evaluation on multiple tasks compared with BRET. When Google proposed BRET, the authors evaluated the performance of BRET in over 10 different specific tasks, but these code embedding approaches are only evaluated on two or three SE tasks. Second, lack of studies that performed systematic experiments to summarize their respective advantages, disadvantages as well as usage scenarios. Therefore, researchers need to conduct more studies on classifying DNNs according to their functions and compare the performance and characteristics among themselves and traditional techniques in different SE research topics where most DL algorithms were applied.

**Opportunity 4: No open-source datasets in certain fields.** In Section 5, we present commonly used datasets in different SE tasks. But, we noticed that almost no open-source image-based datasets although image-based datasets were widely used in 5 out of 6 SE activities, including software design pattern detection, GUI modeling, code localization, bug detection, and software quality assessment. It is actually a huge contribution to constructing image-based benchmark datasets. Since more open-source and benchmark datasets have been developed for a SE field, the field has more opportunities for development. future studies can work on building benchmarks for SE tasks having no open-source datasets.

**Opportunity 5: Performance comparison between DNNs and traditional techniques in SE tasks.** DL has been gradually used in more and more SE tasks, replacing the status of traditional algorithms. However, are DL algorithms really more efficient than traditional algorithms? What SE tasks are suitable for DL algorithms? What factors determine whether DL algorithms are better or worse than traditional algorithms? In our work, we

summarized three related studies that compared the performance of DNNs and traditional models in different SE tasks: code clone detection, tag recommendation, and the classifying relatedness between knowledge units. According to the experimental results in their studies, we noticed that traditional approaches can achieve good performance in detecting Type 1 Type 2 and Type 3 code clones. And DL-based clone detectors have an advantage over semantic clone detection. In the task of prediction of relatedness in stack overflow, the performance of traditional approaches (SVM) is slightly better than DL-based. Besides, some of the traditional approaches for tag recommendation outperform DL-based models. However, there is a lack of studies to compare the performance differences of traditional algorithms (e.g., rule-based algorithms), machine learning algorithms, and deep learning algorithms on different problem types (i.e., regression, classification, recommendation, and generation).

Besides, based on our findings in our work, we think that DL-based models may not be suitable for some SE tasks. If a SE issue can be generalized into rules, i.e. rule-based, using traditional approaches achieves good performance with high probability, and if a SE issue involves semantic understanding, DL techniques can be the first choice.

**Opportunity 6: Semantic understanding using DNNs.** The existing traditional clone detectors [5, 11] (i.e., not using DNNs) have achieved well performance (over 90% F1-score), especially detecting type 1-3 clones. Hence, a question arises, i.e., is it necessary for DNNs to be used for code clone detection? After further research, we found that compared with traditional tools, DL-based approaches can generally achieve better performance on type-4 clone detection. This finding may answer this question to some extent. That is, the DL is better at understanding the semantic of data, no matter code, and natural language.

**Opportunity 7: Code transformation.** Many SE studies require solving specific issues across multiple languages. For example, many identical functions can be written in different languages, such as the quick sort algorithm in C and Java respectively. However, most DNN models only target one specific program language. Therefore, a potential opportunity is to construct the mappings between code fragments written in different program languages with the same functionalities by leveraging the characteristic that DNNs can better understand the code semantics. In this way, it is can alleviate the language limit for DNNs to some extent.

## 9 CONCLUSION

This work performed a SLR on 250 primary studies related to DL for SE from 32 publication venues, including conference proceedings, symposiums, and journals. We established a background analysis of primary studies and four research questions to comprehensively investigate various aspects pertaining to applications of DL models to SE tasks. Our SLR showed that there was a rapid growth of research interest in the use of DL for SE. Through an elaborated investigation and analysis, four DL architectures containing 30 different DNNs were used in primary studies, where RNN, CNN, and FNN are the three most widely used neural networks compared with other DNNs. We also generalized three different model selection strategies and analyzed the popularity of each one. To comprehensively understand the DNN training and testing process, we provided a detailed overview of key techniques in terms of data collection, data classification, data processing, data representation RQ2. IN RQ3, we analyzed the distribution of DL techniques used in different SE activities, investigated the relationships of DNNs with respect to DL architecture, task types, problem types, and data types. We also classified primary studies according to specific SE tasks they solved and gave a brief summary of each work. We observed that DL techniques were applied in 40 SE topics, covering 6 SE activities. IN RQ4, we summarized the widely used optimization algorithms in DNNs and classified important evaluation metrics used in regression, classification, recommendation, and generation tasks. Finally, we identified a set of current challenges that still need to be addressed in future work on using DLs in SE.

## ACKNOWLEDGEMENTS

This research is supported by ARC Laureate Fellowship (FL190100035) and the National Research Foundation, Singapore under its Industry Alignment Fund âĂŞ Pre-positioning (IAF-PP) Funding Initiative. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore.

## REFERENCES

[1] Everton da Silva Maldonado, Emad Shihab, and Nikolaos Tsantalis. 2017. Using natural language processing to automatically detect self-admitted technical debt. *IEEE Transactions on Software Engineering* 43, 11 (2017), 1044–1062.

[2] Li Deng. 2014. A tutorial survey of architectures, algorithms, and applications for deep learning. *APSIPA Transactions on Signal and Information Processing* 3 (2014).

[3] Wei Fu and Tim Menzies. 2017. Easy over hard: A case study on deep learning. In *FSE*. 49–60.

[4] Geoffrey E Hinton, Simon Osindero, and Yee-Whye Teh. 2006. A fast learning algorithm for deep belief nets. *Neural computation* 18, 7 (2006), 1527–1554.

[5] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. 2007. Deckard: Scalable and accurate tree-based detection of code clones. In *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 96–105.

[6] Staffs Keele et al. 2007. *Guidelines for performing systematic literature reviews in software engineering*. Technical Report. Technical report, Ver. 2.3 EBSE Technical Report. EBSE.

[7] Warren S McCulloch and Walter Pitts. 1943. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics* 5, 4 (1943), 115–133.

[8] Abdel-rahman Mohamed, George Dahl, and Geoffrey Hinton. 2009. Deep belief networks for phone recognition. In *Nips workshop on deep learning for speech recognition and related applications*, Vol. 1. Vancouver, Canada, 39.

[9] Tapas Nayak and Hwee Tou Ng. 2020. Effective modeling of encoder-decoder architecture for joint entity and relation extraction. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 8528–8535.

[10] Kai Petersen, Sairam Vakkalanka, and Ludwik Kuzniarz. 2015. Guidelines for conducting systematic mapping studies in software engineering: An update. *IST* 64 (2015), 1–18.

[11] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. 2016. Sourcerercc: Scaling code clone detection to big-code. In *Proceedings of the 38th International Conference on Software Engineering*. 1157–1168.

[12] Xi Xiao, Ruibo Yan, Runguo Ye, Qing Li, Sancheng Peng, and Yong Jiang. 2015. Detection and prevention of code injection attacks on HTML5-based apps. In *2015 Third International Conference on Advanced Cloud and Big Data*. IEEE, 254–261.

[13] Xin Ye, Razvan Bunescu, and Chang Liu. 2014. Learning to rank relevant files for bug reports using domain knowledge. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 689–699.

[14] Fiorella Zampetti, Alexander Serebrenik, and Massimiliano Di Penta. 2018. Was self-admitted technical debt removal a real removal? an in-depth perspective. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. IEEE, 526–536.