1-2022

A survey on deep learning for software engineering

Yanming YANG

Xin XIA

David LO
Singapore Management University, davidlo@smu.edu.sg

Published in ACM Computing Surveys, 54(10), art. no. 206. (2023) DOI: 10.1145/3505243

A Survey on Deep Learning for Software Engineering

YANMING YANG, Faculty of Information Technology, Monash University, Australia
XIN XIA, Faculty of Information Technology, Monash University, Australia
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 analyse the relevant studies published since 2006. We first provide an example to illustrate how deep learning techniques are used in SE. We then summarize and classify different deep learning techniques used in SE. We analyzed key optimization technologies used in these deep learning models, and finally describe a range of key research topics using DNNs in SE. 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 [84]. This pioneered a new era of research on artificial intelligence (AI). In 2006, Hinton et al. [40] 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 [87], showing great ability to solve complex problems. DL has since attracted attention of academics and industry practioners for many tasks. Development of Nvidia's graphics 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

Authors' addresses: Yanming Yang, Yanming.Yang@monash.edu, Faculty of Information Technology, Monash University, Melbourne, Australia; Xin Xia, Faculty of Information Technology, Monash University, Melbourne, Australia, Xin.Xia@monash.edu; 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.

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 [1, 100, 127]. In software design and modeling, DL has been leveraged for design pattern detection [108], UI design search [14], and software design mining [81]. During software implementation, researchers and developers have used DL for source code generation [26], source code modeling [49], software effort/cost estimation [7], 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 [118], bug localization [63], vulnerability prediction [37]. It has been used for a varietu of software testing applications, such as test case generation [73], and automatic testing [144]. Researchers have applied DL to SE tasks to facilitate software maintenance and evolution, such as code clone detection [90], feature envy detection [69], code change recommendation [106], user review classification [28], 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 142 papers published in 20 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 on 142 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 provide 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 the key factors that impact the performance of DL models in SE, including dataset, model optimization, and model evaluation.
(4) We provide 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, ranking, and generation tasks.
(5) 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 introduces the workflow of a DL model through an example. Section 3 presents our Systematic Literature Review methodology. Section 4 investigates the distribution and evolution of DL studies for SE tasks, and Section 5 gives an overall analysis on various DL techniques used in primary studies, including classifying 30 DNN models based on their architectures and summarizing the model selection strategies adopted by studies. Section 6 analyzes a set of key techniques from four perspectives – datasets, model optimization, model evaluation, and the accessibility of source code. Section 7 lists research topics involved in primary studies and makes a briefly description of each work. Section 8 presents limitations of this study and its main threats to validity. Section 9 discusses the challenges that still need to be solved in future work and outlines a clear research road-map of research opportunities. Section 10 concludes this paper.

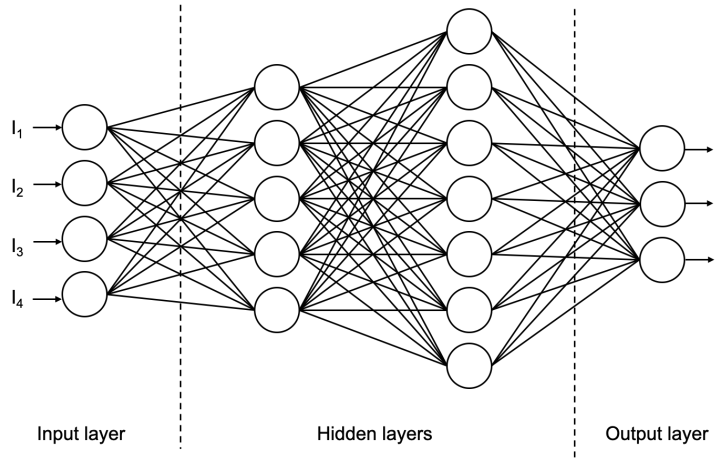


Fig. 1. The basic structure of a deep learning model.

2 DEEP LEARNING

2.1 Basic Structure of DL

Most learning algorithms are shallow models with one or two non-linear feature representation layers, such as GMM, HMM, SVM, and MLP. The limitation of such a shallow model is the lack of ability to express complex functions. Their generalization ability is restricted for the complexity of problems, resulting in decreased learning ability.

Deep learning allows computational models composed of multiple layers to learn data representations with multiple higher levels of abstraction [61]. This builds a neural network that simulates the human brain for analysis and learning. Similar to traditional ML, DL is suitable for various types of problems, such as regression, clustering, classification, ranking, and generation problems. We present the basic structure of a DNN in Fig. 1.

Based on the position and function of different layers, layers in a DNN can be classified into three categories, i.e., the input layer, the hidden layer, and the output layer. Generally, the first layer denotes the input layer, where the preprocessed data can be fed into DNNs; the last layer denotes the output layer, from which the results of a model can be achieved, e.g., classification results, regression results, generation results, etc. The middle layers between the input layer and the output layer are hidden layers. DNNs usually contain multiple hidden layers for enhancing the expressive ability of DNNs and learning high-level feature representation. Besides, the way to connect between different layers may vary, and as shown in Fig. 1, adjacent layers are full-connected (aka., full-connected layer), meaning that any neuron in the i_{th} layer are connected to any neuron in the $i + 1_{th}$ layer. In some DNNs with complex structures, for tackling different SE issues, not only the fully connected layer can be used as a hidden layer, but also a layer composed of other types of neurons can also be used as the hidden layer of a DNN, such as convolution layer, pooling layer, LSTM layers, etc..

Currently, diverse DNNs and learning methods are used for SE tasks, such as Feedback Neural Network (FNN), Convolutional Neural Networks (CNN), Recurrent Neural Networks (RNN), AutoEncoders, Generative Adversarial Networks (GAN), and Deep Reinforcement Learning [31].

2.2 Workflow when Using DL in SE

We present an example of using DL for a representative SE task. A novel sequence-to-sequence neural network is used to automatically perform comment updates with code changes [77]. Code comments are a vital source of software documentation, helping developers to have a better understanding of source code and are beneficial to the

communication between developers. However, developers sometimes neglect to update comments after changing source code, leading to obsolete or inconsistent comments. It is necessary to perform Just-In-Time (JIT) code comments updating, which aims to reduce and even avoid bad comments. To achieve this, a new approach called CUP builds a novel neural seq2seq model. Using DL techniques for such an SE task can be broken into six steps: (1) data collection, (2) data processing, (3) input construction, (4) model training, (5) model optimization, (6) model testing, and (6) model application.

Data collection: Collecting data is a key step when building and training a DL model. Different SE tasks need to process and analyze different data types, such as requirements documents for requirements analysis, bug reports for bug report summarization, source code for code clone detection or code search, etc. In this example, for updating JIT code comments, the commonly available data are method-level code change datasets with comments. Each qualified instance contains old code snippets, new code snippets, old code comments, and new code comments.

Data processing: Processing raw data involves a number of steps, including data filtering, data splitting, data cleaning, data augmentation, and data segmentation for eliminating noise in data. For JIT code comments updating, some instances are removed with unqualified comments, and no differences or empty between old and new comments. Instances containing abstract methods are deleted to reduce method mismatching.

Input construction: Since most DL models have strict requirements on the input-form, such as the input requiring to be numeric and fixed size, it is necessary to transform SE data into multi-dimensional vectors in order to use DL models. In this JIT code comment updating example, code comments and code changes can be viewed as text-based data, processed by using a token-based method. Code changes and code comments are converted into sequences so that they can be fed into their seq2seq model through data flattening and tokenization. After tokenization, old comments are converted into a token sequence. While to better represent code changes, each change is aligned into two token sequences and construct a triple $< t_i, t_i^{'}, a_i >$ as an edit sequence to respectively record old source code, new source code, and an edit action, i.e., insert, delete, equal or replace.

Model training: DL users need to select suitable DL techniques and different datasets, construct the structure of a model, and decide model configuration, e.g., the number of layers and neural units of each layers. In our example, a seq2seq DL model was built by training an encoder-decoder LSTM, since it is good at process nature language text and token-based source code. In this model, the edit sequence of a code change and a token sequence of its old comment were fed into the input layer. To capture the relationship between the code change and the old comment, the encoder was composed of 4 layers: an embedding layer, a contextual embed layer, a co-attention layer, and a modeling layer, where each layer had its role. The decoder included 2 layers: a LSTM layer and a dense layer. The output of decoder was a new comment based on the corresponding captured relationship.

Model optimization: After model design and construction, the designed model will be trained with the training set for achieving an effective DL model. Whether a model can work depends on thousands of parameters (aka., weights), connecting neural units adjacent layers. Hence, model training is to fine-tune these weights to minimize the loss of the model. For the seq2seq model in the example, the weights in the seq2seq neural network are trained by minimizing the difference between the real new comment and the generated new comment in a supervised way.

Model testing: Generally, a training set is usually divided into two subsets of unequal sizes. The big subset is used for training the DL model, while the small one will be used for validating and testing the performance of the model when meeting new data. In this example, 20% of samples in the training set are put into the validation and test set to ensure the effectiveness of CUP.

Model application: Finally, the trained DL model can be applied to tackle practical SE tasks. In this example, the trained model leverages two distinct encoders and a co-attention layer to learn the relationships between the code change and the old comment. The LSTM-based decoder is used to generate new comments whose tokens are copied from both the new code and the old comments.

3 METHODOLOGY

We performed a systematic literature review (SLR) following Kitchenham and Charters [52] and Petersen et al. [98]. In this section, we present details of our SLR methodology.

3.1 Research Questions

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 wanted to answer the following research questions:

(1) **RQ1: What are the trends in the primary studies on the use of DL in SE?**
(2) **RQ2: What DL techniques have been applied to support SE tasks?**
(3) **RQ3: What key factors contribute to difficulties in training DNNs for SE tasks?**
(4) **RQ4: What types of SE tasks and which SE phases have been facilitated by DL-based approaches?**

RQ1 analyzes the distribution of relevant publications that used DL in their studies since 2006 to give an overview of the trend of DL in SE. RQ2 provides a classification of different DL techniques supporting SE tasks and analyze their popularity based on their frequency of use in SE. RQ3 explores key technologies and factors that may affect the efficiency of the DNN training phase. RQ4 investigates what types of SE tasks and which SE phases have been facilitated by DNNs.

3.2 Literature search and selection

To collect DL related papers in SE, we identified a search string including several DL related terms frequently appeared in SE papers that make use of DL. We then refined the search string by checking the title and abstract of a small number of relevant papers. After that, we used logical ORs to combine these terms, and the search string is:

("deep" OR "neural" OR "Intelligence" OR "reinforcement")

We specified the range the papers are published later: 2006-July 2020. Following previous studies [43, 47, 67], we selected 22 widely read SE journals (10) and conferences (12) listed in Table 1 to conduct a comprehensive literature review. We run the search string on three databases (i.e., ACM digital library [1], IEEE Explore [2], and Web of Science [3]) looking for publications in the 22 publication venues whose meta data (including title, abstract and keywords) satisfies the search string. Our search returns 655 relevant papers. After discarding duplicate papers, we applied some inclusion/exclusion criteria (presented in Section 3.3) by reading their title, abstract and keywords, and narrow the candidate set to 146 studies. After reading these 146 studies in full to ensure their relevance, we retained 142 studies.

3.3 Inclusion and Exclusion Criteria

After retrieving studies that match our search string, it is necessary to filter unqualified studies, such as studies with insufficient contents or missing information. To achieve this, we applied our inclusion and exclusion criteria to determine the quality of candidate studies for ensuring that every study we kept implemented and evaluated a full DL approaches to tackle SE tasks.

The following inclusion and exclusion criteria are used:

✔ The paper must be written in English.
✔ The paper must adopt DL techniques to address SE problems.
✔ The length of paper must not be less than 6 pages.
✘ Books, keynote records, non-published manuscripts, and grey literature are dropped.

[1] https://dl.acm.org
[2] https://ieeexplore.ieee.org
[3] http://apps.webofknowledge.com

Table 1. Publication venues for manual search

No.	Acronym	Full name
1.	ICSE	ACM/IEEE International Conference on Software Engineering
2.	ASE	IEEE/ACM International Conference Automated Software Engineering
3.	ESEC/FSE	ACM SIGSOFT Symposium on the Foundation of Software Engineering/European Software Engineering Conference
4.	ICSME	IEEE International Conference on Software Maintenance and Evolution
5.	ICPC	IEEE International Conference on Program Comprehension
6.	ESEM	International Symposium on Empirical Software Engineering and Measurement
7.	RE	IEEE International Conference on Requirements Engineering
8.	MSR	IEEE Working Conference on Mining Software Repositories
9.	ISSTA	International Symposium on Testing and Analysis Working Conference on Mining Software Repositories
10.	SANER	IEEE International Conference on Software Analysis, Evolution and Reengineering
11.	ICST	IEEE International Conference on Software Testing, Verification and Validation
12.	ISSRE	IEEE International Symposium on Software Reliability Engineering
13.	TSE	IEEE Transactions on Software Engineering
14.	TOSEM	ACM Transactions on Software Engineering and Methodology
15.	ESE	Empirical Software Engineering
16.	JSS	Journal of Systems and Software
17.	IST	Information and Software Systems
18.	ASEJ	Automated Software Engineering
19.	IETS	IET Software
20.	STVR	Software Testing, Verification and Reliability
21.	JSEP	Journal of Software: Evolution and Process
22.	SQJ	Software Quality Journal

✘ If a conference paper has an extended journal version, the conference version is excluded.

3.4 Data Extraction and Collection

After removing the irrelevant and duplicated papers, we extracted and recorded the essential data and performed overall analysis for answering our four RQs. Table 2 described the detailed information being extracted and collected from 142 primary studies, where the column $'ExtractedDataItems'$ lists the related data items that would be extracted from each primary study, and the column 'RQ' denotes the related research questions to be answered by the extracted data items on the right. To avoid making mistakes in data collection, two researchers extracted these data items from primary studies together and then another researcher double checked the results to make sure of the correctness of the extracted data.

4 RQ1: WHAT ARE THE TRENDS IN THE PRIMARY STUDIES ON USE OF DL IN SE?

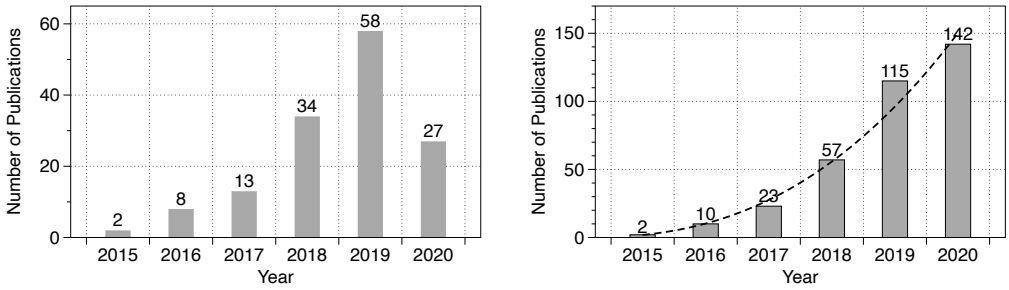
We analyzed the basic information of primary studies to comprehend the trend of DL techniques used in SE in terms of the publication date, publication venues, and main contribution types of primary studies.

4.1 Publication trends of DL techniques for SE

We analyzed the publication trends of DL-based primary studies published between 2006 and the middle of 2020. Although the concept of "Deep Learning" has been proposed in 2006 and DL techniques had been widely used in many other fields in 2009, we did not find any studies using DL to address SE tasks before 2015. Fig. 2(a) shows the number of relevant studies published in predefined publication venues since the middle of 2020. It can be observed that the number of publications from 2015 to 2019 shows a significant increase, with the number reaching 58 papers in 2019. In data collection, we only collect papers whose initial publication date is on July 2020

Table 2. Data Collection for Research Questions

RQs	Extracted data items
RQ1	Basic information of each primary study (i.e., title, publication year, authors, publication venue)
RQ1	The type of main contribution in each study (e.g., empirical study, case study, survey, or algorithm)
RQ2	DL techniques used in each study
RQ2	Whether and how the authors describe the rationale behind techniques selection
RQ3	Dataset source (e.g., industry data, open source data, or collected data)
RQ3	Data type (e.g., source code, nature language text, and pictures)
RQ3	The process that datasets are transformed into input sets suitable for DNNs
RQ3	Whether and what optimization techniques are used
RQ3	What measures are used to evaluate the DL model
RQ3	Presence / absence of replication package
RQ4	The practical problem that a SE task tries to solve
RQ4	The SE activity in which each SE task belongs
RQ4	The approach used for each SE task (e.g., regression, classification, ranking, and generation)



(a) Number of publications per year.

(b) Cumulative number of publications per year.

Fig. 2. Publication trends of DL-based primary studies in SE.

or earlier; thus, the number of relevant studies in 2020 cannot reveal the overall trend of DL in 2020. However, extrapolating from the number of primary studies in previous years, we can estimate that there may be over 65 relevant publications using various DL techniques to solve SE issues by the end of 2020.

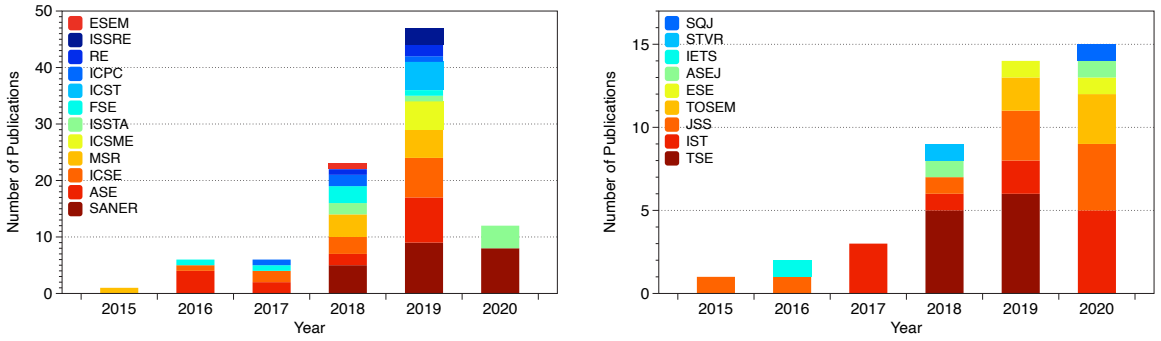
We also performed an analysis of the cumulative number of publications as shown in Fig. 2(b). We fit the cumulative number of publications as a power function, showing the publication trend in the last five years. We can notice that the slope of the curve fitting the distribution increases substantially between 2015 and 2019, and the coefficient of determination (R^2) attains the peak value (0.99447), which indicates that the number of relevant studies using DL in SE intends to experience a strong rise in the future. Therefore, after analyzing Fig. 2, it can be foreseen that using DL techniques to address various SE tasks has become a prevalent trend since 2015, and huge numbers of studies will adopt DL to address further challenges of SE.

4.2 Distribution of publication venues

We reviewed 142 studies published in various publication venues, including 12 conference proceedings and symposiums as well as 10 journals, which covers most research areas in SE. Table 3 lists the number of relevant papers published in each publication venue. 69% of publications appeared in conferences and symposiums, while only 31% of journal papers leveraged DL techniques for SE tasks. Among all conference papers, only 4 different conferences include over 10 studies using DL in SE in the last five years, i.e., SANER, ASE, ICSE, and MSR. Compared with other conference proceedings, SANER is the most popular one containing the highest number of primary study papers (22), followed by ASE (16). There are 13 and 6 relevant papers published in ICSE and

Table 3. Publication Venues with DL-based Studies.

Conference venue	#Studies	Journal venue	#Studies
SANER	22	TSE	11
ASE	16	IST	11
ICSE	13	JSS	10
MSR	10	TOSEM	5
ICSME	8	ESE	2
ISSTA	7	ASEJ	2
FSE	6	IETS	1
ICST	5	STVR	1
ICPC	4	SQJ	1
RE	3		
ISSRE	3		
ESEM	1		



(a) Number of primary studies published in various conference proceedings.

(b) Number of primary studies published in various journals.

Fig. 3. Distribution of different publication venues.

FSE, respectively. Meanwhile, in all journals, TSE and IST include the highest number of relevant papers (11). Ten studies related to DL techniques were published in JSS, and 5 were published in TOSEM. Almost half of the publication venues only published not more than 5 relevant papers.

We also checked the distribution of primary studies published in conferences and journals between 2015 and 2020, shown in Fig. 3. Fig 3(a) illustrates that the publication trend of various conference proceedings and symposiums has a noticeable increase from 2015 to 2019. 70.4% of conference papers were published in 2018 and 2019, while only a few different conferences or symposium venues included relevant papers between 2015 and 2017, which demonstrates a booming trend in the last few years.

Fig. 3(b) shows the number of primary study papers published in different journal venues. It can be seen that there is an increasing trend in the last five years, especially between 2018 and 2020. Furthermore, the relevant papers published in TSE, as one of the most popular journals, accounts for the largest proportion in 2018 and 2019; while another popular journal, IST, also makes up a large percentage in 2019 and 2020.

4.3 Types of main contributions

We summarized the main contribution of each primary study and then categorized these studies according to their main contributions into five categories, i.e., New technique or methodology, Tool, Empirical study, Case study, and User study. We gave the definition of each main contribution in Table 4. The main contribution of 76% primary studies was to build a novel DNN as their proposed new technique or methodology for dealing with various

Table 4. The definition of five main contributions in primary studies.

Main contribution	Definition
New technique or methodology	The study provided a solid solution or developed a novel framework to address specific SE issues.
Tool	The study implemented and published a new tool or tool demo targeting SE issues.
Empirical study	The study collected primary data and performed a quantitative and qualitative analysis on the data to explore interesting findings.
Case study	The study analyzed certain SE issues based on one or more specific cases.
User study	The study conducted a survey to investigate the attitudes of different people (e.g., developers, practitioners, users, etc) towards SE issues.

problems in different SE activities. 10% of relevant studies concentrated on performing assessment and empirical studies for exploring the benefits of DL towards different SE aspects, such as research on the differences between ML and DL to solve certain SE tasks, the performance of using DL to mine software repositories, applying DL in testing, etc. The main contribution of 9% was case studies. 2 primary studies (1%) that both proposed a novel methodology and evaluated the novel methodology via a user study. Therefore, the main contribution of these two studies spans across two categories, i.e., New technique or methodology and user study.

Summary

(1) DL has shown a booming trend in recent years
(2) Most of primary study papers were published between 2018 and 2020
(3) The number of conference papers employing DNNs for SE significantly exceeds that of journal papers.
(4) SANER is the conference venue publishing the most DL-based papers (22), while TSE and IST include the highest number of relevant papers among all journals (11).
(5) Most DL-based studies were only published in a few conference proceedings (e.g., SANER, ASE, ICSE, MSR) and journals (e.g., TSE, IST, JSS, and TOSEM).
(6) The main contribution of 75% primary studies is to propose a novel methodology by applying various DL techniques, while only two primary studies performed a user study to better understand users' attitudes and experience toward various DNNs used for solving specific SE tasks.

5 RQ2: WHAT DL TECHNIQUES ARE APPLIED TO SUPPORT SE TASKS?

5.1 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 3 categories: the layered architecture, AutoEncoder (AE), and Encoder-Decoder [20, 91]. We provided a detailed classification of DNNs into five categories, i.e., RNN, CNN, FNN, GNN, and tailored DNN models, where tailored DNNs include the DNNs not often used in SE, e.g., DBN, HAN, etc. Table 5 shows the variety of different DNNs, and also lists the number of times these models have been applied in SE.

As can be seen from Table 5 that compared Encoder-Decoder and AutoEncoder (AE) architectures, layered based DNNs are the most popular and widely used architecture. In the layered architecture, 72 primary studies used nine different kinds of RNN-based models to solve practical SE issues, where LSTM is the most often applied RNN-based model (35), followed by standard RNN (23). The variants of LSTM, such as GRU and Bi-LSTM, are often adopted by researchers in multiple research directions. 48 primary studies employed CNN-based models,

Table 5. The number of various DNNs applied in per year.

Architecture	Family	Model Name	2015	2016	2017	2018	2019	2020	Total
Layered architecture (157)	RNN-based model (72)	RNN	1		3	7	10	2	23
		RtNN		1		1			2
		Bidirectional RNN (BRNN)						1	1
		LSTM			3	10	16	6	35
		Bi-LSTM			1	1		2	4
		Siamese LSTM						1	1
		GRU					1	3	4
		Bidirectional GRU						1	1
		Recurrent Highway Network					1		1
	CNN-based model (48)	CNN		2	2	13	20	6	43
		Tree-based CNN (TBCNN)					2	1	3
		RCNN						1	1
		Deep Residual Network						1	1
	FNN-based model (25)	FNN		3	1	8	7	3	22
		RBFNN	1						1
		Deep Sparse FNN				1			1
		Deep MLP					1		1
	GNN-based model (6)	GGNN					4	1	5
		Graph Matching Network (GMN)						1	1
	Tailored model (4)	Deep Beliefe Network (DBN)		1		1			2
		HAN					1		1
		Deep Forest					1		1
Encoder-Decoder (15)	RNN-based model (12)	RNN		1		1	6		8
		LSTM				2		2	4
	CNN-based model (1)	CNN					1		1
	FNN-based model (2)	FNN				1		1	2
AutoEncoder (7)	RNN-based model (1)	GRU						1	1
	CNN-based model (1)	CNN						1	1
	FNN-based model (5)	FNN			2	1	2		5

where almost 90% of studies employed CNN. FNN-based model is the third most frequently used family with 25 studies using FNNs, followed by GNN-based models and tailored models. There are 24 combined DNNs were proposed in tailored models.

15 primary studies leveraged different types of DNNs following the Encoder-Decoder architecture, where RNNs were used in 12 studies, which is much higher than the number of other models used, i.e., CNN and FNN. In the last architecture, over 70% of studies used FNN-based AEs as their proposed novel approaches; only 2 studies selected GRU and CNN to construct AEs respectively.

5.2 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 [12, 23, 33, 145]. 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 [10, 49, 137]. 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 [2, 22, 110].

We noticed that the most commonly selection strategy is *S*1 (i.e., Characteristic-based selection strategy), accounting for **69%**, nearly 3 times that of *S*2 **(25%)**. Only **6%** of primary studies adopt *S*3 to select their suitable DL algorithms.

Summary

(1) There are 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 three types: Layered architecture, Encoder-Decoder, and AutoEncoder (AE); the family can be classified into five categories: RNN-based, CNN-based, FNN-based, GNN-based, and Tailored models.
(3) Compared with Encoder-Decoder and AE, 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 (43), LSTM (35), RNN (23), and FNN (22), and each of them has several variants that are also often used in many SE tasks.
(5) We summarized three types of DNN-based model selection strategies. The majority of studies adopted *S*1 to select suitable DL algorithms. Only 6% of primary studies used *S*3 as the model selection strategy due to the heavy workload brought by *S*3.

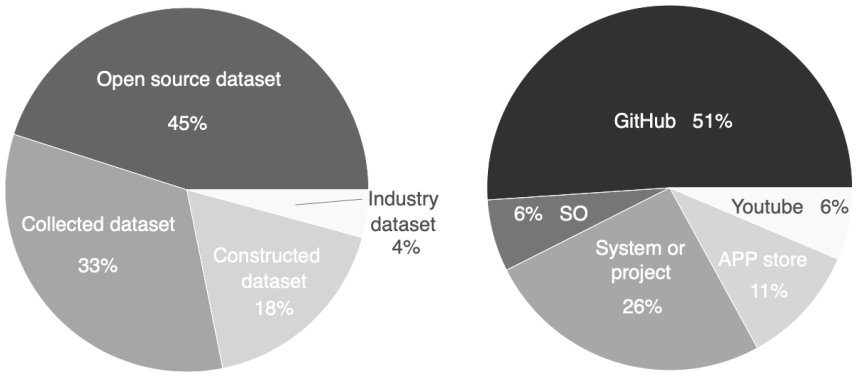
6 RQ3: WHAT KEY FACTORS CONTRIBUTE TO DIFFICULTY WHEN TRAINING DNNS IN SE?

Since analyzing a DL architecture can provide a lot of insight, we investigated the construction process of a DL framework from three aspects: techniques used in data processing, model optimization, evaluation, and the accessibility of primary studies.

6.1 How were datasets collected, processed, and used?

Data is one of the most important roles in the training phase. Unsuitable datasets can result in failed approaches or tools with the low performance. We focused on the data used in primary studies and conducted a comprehensive analysis on the steps of data collection, data processing, and data application.

6.1.1 What were the sources of datasets used for training DNNs? Fig. 4 shows the sources of datasets in the selected primary studies. It can be seen that 45% of primary studies trained DNNs by using an open-source dataset. One reason for choosing an open-source dataset is that studies are willing to pick up these datasets to evaluate the effectiveness of proposed DL-based approaches due to the existence of widely accepted datasets in



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

Fig. 4. The source of datasets used in primary study papers

certain SE issues (e.g., code clone detection, software effort/cost prediction, etc). Others are used because the datasets were applied in related previous work. Due to the lack of available and suitable datasets, 18% of primary studies constructed new datasets. Real-world datasets from industry are only used by 4% of studies.

33% of studies performed a series of experiments on large-scale datasets so as to verify the scalability and robustness of their models. To achieve this, many studies collected multiple small datasets from different sources. Fig. 4(b) describes the sources of collected datasets. As tens of thousands of developers contribute to GitHub community by uploading source code of their software artifacts, GitHub has become the most frequently used source of collected data (51%). 26% of studies collected their datasets from different systems and projects. For instance, Deshmukh et al. [21] 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. The app store is the third-largest source (11%), followed by Stack Overflow (SO) and Youtube.

6.1.2 What are 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. We summarize the data types in primary studies and interpreted how data types determine the choice of DNNs.

We classified the data types of used datasets into four categories – code-based, text-based, software repository-based, and user-based data types. Table 6 describes specific data in each data type. 104 primary studies collected data from source code, and most of these studies used source code directly in some important SE activities, such as software testing and maintenance. Datasets containing various metrics were employed in 8 relevant studies, followed by code comments, defects (7), and test cases (6). Whereas few studies focused on analyzing the code annotation, pull-request, and patches. 8 primary studies used a multitude of screencasts as their datasets, where 4 studies selected program screencasts to analyze developers' behavior and 4 studies researched UI images for improving the quality of APPs.

Text-based data types were the second most popular, including 13 different kinds of documentation. Bug report and requirements documentation are the two most commonly applied text-based data types in primary studies. Some types rarely appeared, such as logs, certifications, design documentation, etc.

Since software repositories, especially SO and GitHub, contain a lot of useful patterns or information, we classified the type of the information collected from these repositories into $'Software\ repository-based\ data\ types'$. 12% of studies concentrated on obtaining and learning useful information and patterns by crawling

Table 6. Data types of datasets involved in primary studies.

Family	Data types	#Studies	Total
Code-based data types	Source code	61	104
	Software/code metric	8	
	Code comment	7	
	Defects	7	
	Test case	6	
	program screencasts	4	
	UI images	4	
	Code change	2	
	Code annotation	2	
	Pull-requests	2	
	Patch	1	
Text-based data types	Bug report	9	23
	Requirement documentation	4	
	configuration documentation	2	
	APP description	2	
	Software version information	2	
	Design documentation	1	
	Log information	1	
	Certification	1	
	Protocol message	1	
	Patch	1	
Software repository-based data types	Q&A in SO	6	17
	Tags in SO	5	
	Issues and commits	4	
	Pull-requests	2	
User-based data types	User behavior	3	5
	User review	1	
	Interaction traces	1	

related contents from SO (e.g., Q&A (questions and answers) and tags) and GitHub (e.g., issues, commits and, pull-requests).

User-based data generally contains a great deal of user information, which can promote developers to better comprehend user needs and behavior targeting different applications. Only 5 studies adopted user-based data types (i.e., user behavior, review, and interactions) to solve relevant SE tasks.

6.1.3 What input forms were datasets transformed into when training DNNs? The inputs of DNNs need to be various forms of vectors. We found two techniques were often used to transform different source data types into vectors: One-hot encoding and Word2vec. Only 5 studies produced the input of their models by adopting the One-hot technique. We described input forms using 5 categories referring to their data types.

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.

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.

Table 7. The various input forms of DL-based models proposed in primary studies.

Family	Input forms	#Studies	Total
Token-based input	Code in tokens	17	64
	Text in tokens	34	
	Code and text in tokens	13	
Tree/graph-based input	Code in tree structure	25	29
	Code in graph structure	4	
Feature-based input	feature/metric	33	33
Pixel-based input	pixel	9	9
Hybrid input	Code in tree structure + text in token	4	7
	Code features + text in token	2	
	Code in tree structure + features	1	

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.

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.

Combined 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. [60] proposed a novel approach for generating summaries of programs not only by analyzing their source code but also their code comments.

Table 7 depicts the input formats of DL-based models. We can see that over 45% of studies transformed data (i.e., source code and various documentations) into the token-based input form (64), where 17 studies considered source code as texts and thus converted code into token sequences as the input of models. 13 studies used both source code and text-based materials and also constructed a token-based data structure as the input form of their proposed models. 25 studies utilize tree-based input form to analyze the source code, and only 4 studies transform source code into a graph-based structure for extracting essential information. 33 studies adopted embedding techniques to generate feature-based vectors. Furthermore, 8 studies using image-based datasets split screencasts into pixels as the basic unit of the input form. Only 7 studies processed datasets into multiple forms.

6.2 What techniques were 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.

6.2.1 What learning algorithms are 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 of the DNN as training progresses.

Out of the 142 studies analyzed, 131 identified the specific optimization method, but 11 studies did not mention what optimizers were used to adjust parameters in their work. Fig. 5 illustrated the frequency of the use of 20 optimization methods used in all primary studies. We see that 6 optimizers were used in no less than 5 studies, where Adam optimizer is the most commonly used optimization method. Stochastic gradient descent (SGD) and gradient descent (GD) are also popular optimizers, which were used by 21 and 15 studies respectively, followed

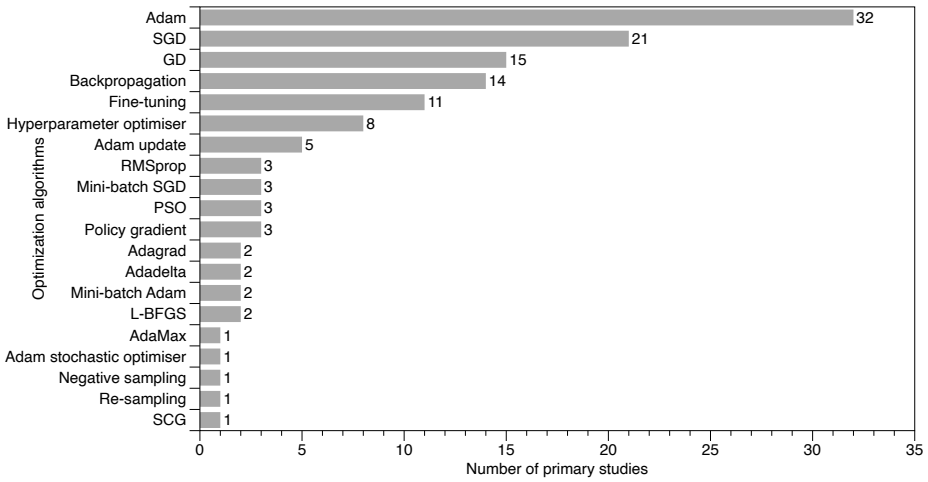


Fig. 5. Various optimization algorithms used in primary studies.

by back-propagation (14) and fine-tuning (11). Besides, some optimization methods are not often used, such as Adagrad and Adadelta.

6.2.2 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 describe 9 general ways to combat overfitting [38, 107] by considering 3 aspects: data processing, model construction as well as model training, and then analyzed the usage distribution of these methods in relevant studies.

Cross-validation: A cross-validation algorithm can split a dataset into k groups (k-fold cross-validation). Researchers often leave one group to be the validation set and the others as the training set. This process will be repeated until each group has been used as the validation set. Since a remaining subset of data is new towards the training process of DL-based models, the algorithm can only rely on the learned knowledge from other groups of data to predict the results of the remaining subset, preventing overfitting.

Feature selection: Overfitting can be prevented by selecting several of the most essential features for training DL-based models can effectively avoid overfitting. Therefore, researchers can pick up some key features by using feature selection methods, train individual models for these features, and evaluate the generalization capabilities of models. For instance, Pooling is a typical technique to prevent overfitting since pooling can reserve main features while reducing the number of parameters and the amount of computation, and improve the generalization ability of the model.

Regularization: Regularization is a technique to constrain the network from learning a model that is too complex, which therefore can avoid overfitting. A penalty term would be added in the cost function to push the estimated coefficients towards zero by applying L1 or L2 regularization.

Dropout: By applying dropout, a form of regularization, to the layers of DNNs, a part of neurons were ignored with a set probability. Therefore, dropout can reduce interdependent learning among units to avoid overfitting.

Data augmentation: A larger dataset can reduce the chance of overfitting. Data augmentation is a good way to artificially increase the size of our dataset for improving the performance of a DNN when the scale of data was constrained due to difficult to gather more data. For example, many studies performed various image transformations

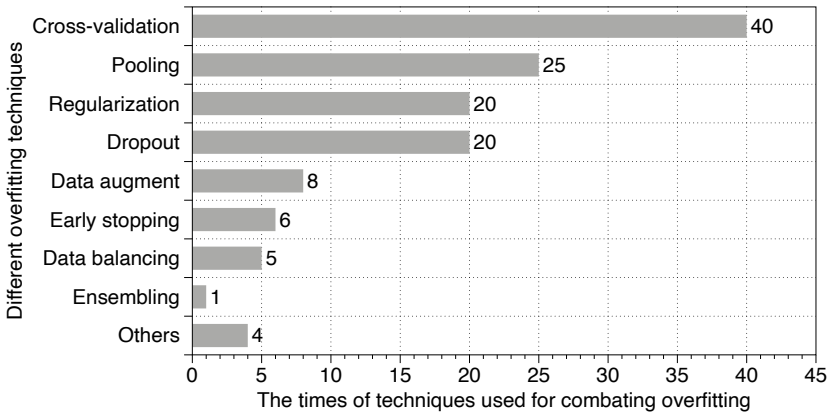


Fig. 6. The distribution of various overfitting techniques used in primary studies

to the image dataset (e.g., flipping, rotating, rescaling, shifting) for enlarging data size in the image classification task.

Early stopping: Early stopping is an effective method for avoiding overfitting by truncating the number of iterations, that is, stopping iterations before DL-based models converge on the training dataset to eliminate the impact on overfitting.

Data balancing: With imbalanced data, DL-based models are likely to occur the overfitting problem since models will learn imbalanced knowledge with a disproportionate ratio of observations in each class. Using some data balancing techniques can effectively alleviate the impact on models' performance caused by overfitting.

Ensembling: Ensembles are a set of machine learning methods for combining predictions from multiple separate models. For instance, Bagging as an ensemble learner can reduce the chance of overfitting complex models by training a large number of "strong" learners in parallel without restriction.

Fig. 6 illustrates the distribution of the techniques used for combating overfitting problems. Cross-validation has been used frequently among the selected studies to prevent overfitting; it is used in 40 studies, followed by pooling (25). Regularization and dropout are the third most popular techniques used in 20 studies. There are 8 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. 6 studies used early stopping and 5 selected data balancing to combat the overfitting problem. Ensembling is the least frequently used one (1) compared with other techniques (1). Furthermore, among all primary studies, 4 studies used several new 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.

6.2.3 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.

Table 8 summarizes the commonly used evaluation metrics in the primary studies, used in no less than 3 studies. Precision, recall, F1-measure, and accuracy are widely accepted metrics for evaluating the performance of DL-based models. Some studies adopted MRR and BLEU as evaluation metrics in their work, potentially indicating that many studies focused on addressing ranking and translation tasks by training various DNNs. Another interesting

Table 8. Metrics used for evaluation.

Metrics	#Studies
Precision@k	69
Recall@k	59
F1@k	53
Accuracy	26
Mean Reciprocal Rank (MRR)	15
BLEU	13
Running time	13
AUC	11
Mean Average Precision (MAP)	7
Matthews Correlation Coefficient (MCC)	5
P-value	4
METEOR	4
ROC	4
ROUGE	4
Mean Absolute Error (MAE)	4
SuccessRate@k	3
Cliff's Delta	3
Coverage	3
Bal (Balance)	3
Standardized Accuracy (SA)	3
Others	11

observation from Table 8 is that running time is selected as a performance indicator by a set of studies, which does not occur frequently when using non-learning techniques. This is because that learning algorithms, especially DNNs, require more time during their construction, training, and testing phases due to the high complexity of these networks (e.g., numerous types of layers, a great many neurons, and different optimization methods). Also, almost half of metrics are not commonly used in relevant studies, which are only used in 3 or 4 studies (e.g., P-value, ROC, ROUGE, Coverage, Balance, etc.) and thus these metrics can reflect their respective characteristics of different SE tasks.

6.3 Accessibility of DL-based models used in primary studies.

We checked whether the source code of DL-based models is accessible for supporting replicability and reproducibility. **53** studies provided the replication packages of their DL-based models, only accounting for **37.3%** of all primary studies. **89** 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) Most datasets are available online and 33% of datasets consist of multiple small-scale ones collected from GitHub, software systems and projects, and some software repositories.
(2) 5 different data types are used in the primary studies, i.e., code-based, text-based, software repository-based, graph-based, and user-based data types, where code-based and text-based types are the two main data types being used in 82.3% of primary studies.

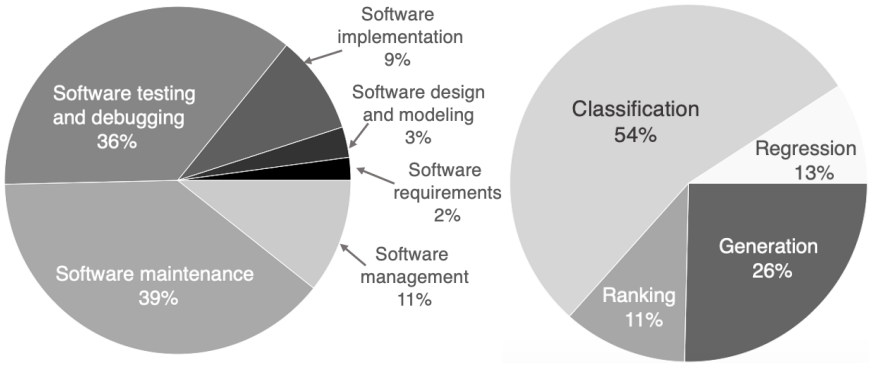


Fig. 7. The distribution of DL techniques in Fig. 8. The classification of primary studies. Different SE activities.

(3) Most studies parse source code into token, tree, or 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.
(4) We observed that Adam is the most popular optimization algorithm being used in 32 studies, followed by SGD and GD, and several variants of Adam are still commonly used in SE. There are also some well-known optimization algorithms used in primary studies, such as back propagation, fin-tuning, and hyperparameter optimizer.
(5) 9 different ways are used for combating the overfitting problem. 4 techniques were widely used – cross-validation, feature selection, regularization, and dropout. We found that studies applying CNNs would choose the pooling method to prevent overfitting with high probability, and data type and input form does not influence technique selection.
(6) There are over 20 different metrics used to verify the performance of DL-based models. Precision, recall, F1-measure, and accuracy were used commonly in primary studies.
(7) Only 53 studies provided a public link of their models in their papers and yet 62.7% of proposed models are difficult to be reproduced since the source code is unavailable.

7 RQ4: WHAT TYPES OF SE TASKS AND WHICH SE PHASES HAVE BEEN FACILITATED BY DL-BASED APPROACHES?

In this section, we first categorise a variety of SE tasks into six SE activities referring to Software Engineering Body of Knowledge [8], i.e., software requirements, software design, software implementation, software testing and debugging, software maintenance, and software management. We then analyze the distribution of DL-based studies for different SE activities. We present a short description of each primary study, including the specific SE issue each study focused on, which and how DL techniques are used, and the performance of each DL model used.

7.1 Distribution of DL techniques in different SE activities

We analysed which SE activities and specific SE tasks each selected primary study tried to solve. As shown in Fig. 7, the largest number of primary studies focused on addressing SE issues in software maintenance (39%). 36% of studies researched software testing and debugging. Software management was the topic of 11% of primary studies, followed by software implementation (9%). Software design and modeling (3%) and software requirements (2%) are addressed in very few studies.

Table 9. The specific research topics where DL techniques are often applied.

SE activities	specific research topics	#Studies	Total
Software design	Source code representation	5	5
Software implementation	Code search	5	9
	Code programming	4	
Software testing and debugging	Defect prediction	11	37
	Bug localization	7	
	Application testing	7	
	Program analysis	5	
	Test case generation	4	
	Reverse execution	3	
Software maintenance	Code clone detection	11	28
	Program repair	6	
	Code comment generation	4	
	Software quality evaluation	4	
	Source code representation	4	
Software management	Software repository mining	19	25
	Effort cost prediction	6	

We classified all primary studies into four categories based on the types of their SE tasks, i.e., the regression task, classification task, ranking task, and generation task. Fig. 8 describes the distribution of different task types where DL techniques were applied. Classification and generation tasks account for almost 80% of primary studies, where classification is the most frequent task (54%). 13% 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 ranking task, accounting for 11% of all studies.

We summarized a set of research topics in which DL was engaged. Table 9 lists the research topics containing no less than three related studies. Software testing and debugging, as the most prevalent SE activity, has 37 primary studies in six topics. The most popular study is defect prediction (11 studies), followed by bug localization (7) and application testing (7). Software maintenance, as the second most popular activities, involves five research topics with 28 relevant studies, where code clone detection is the most popular research topic. Two SE activities, software implementation and software management, both contain two important research topics, where 19 primary studies mined software repositories by training DNNs, 6 studies estimated development cost, and 5 studies applied DL for code search. Software design and modeling only involve one popular topic, i.e., source code representation/modeling. There are no topics with more than three studies using DL techniques in software requirements.

7.2 Software requirements

7.2.1 Requirements analysis. A number of natural language-based documents that describe users' specific needs or services of a software product can be referred to as user requirements (aka, use cases, or actions) [127]. Extracting use cases of a product from a large volume of textual requirement documentation is a common but labor-intensive task. Since the manual mapping system states between requirements and simulation is a time-consuming task, Pudlitz et al. [100] proposed a self-trained Named-entity Recognition model combined with Bi-LSTM and CNN to extract the system states from requirements specification, working to reduce labor cost when linking the state extracted from requirements to the simulation signal.

7.2.2 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. [128] 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.

7.3 Software design

7.3.1 Software design patterns detection. UI design is an essential component of software development, yet previous studies cannot reliably identify relevant high-fidelity UI designs from large-scale datasets. Martín et al. [78] 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. [108] 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.

7.3.2 GUI modeling. Chen et al. [12] 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. [88] 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%.

7.4 Software implementation

7.4.1 Code search. Gu et al. [33] 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. [32] 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.

Recently, several proposals use DL techniques for code search by embedding source code and given queries into vector space and calculating their semantic correlation [3]. Cambronero et al. [10] noticed that multiple approaches existed for searching related code snippets applied unsupervised techniques, while some adopted supervised ones to embed source code and queries for code search. They defined 3 RQs to investigate whether using supervised techniques is an effective way for code search and what types of DNNs and training corpus to use for this supervision. To understand these tradeoffs quantitatively, They selected and implemented four state-of-the-art code search techniques. They found that UNIF outperformed CODEnn and SCS models based on

their benchmarks and suggested evaluating simple components first before integrating a complicated one. Wan et al. [115] addressed the lack of analysis of structured features and inability to interpret search results in existing code search works. To address these issues, they presented a new approach MMAN, which adopted three different DNNs (i.e., LSTM, Tree-LSTM, and GGNN (Gated Graph Neural Network)) to analyze both shallow features and the semantic features in ASTs, and control-flow graphs (CFGs) of source code. The final results on a large-scale real-world dataset demonstrated that MMAN accurately provided code snippets. Huang et al. [45] proposed an attention-based code-description representation learning model (CDRL) to refine the general DL-based code search approaches. They only picked up description terms and semantically related code tokens to embed a given query and its code snippet into a shared vector space.

7.4.2 Programming. Gao et al. [26] 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. Alahmadi et al. [2] applied a CNN model to automatically identify the exact location of code in images for reducing the noise. They extracted 450 screencasts covering C#, Java, and Python programming languages to evaluate their model, and the final result showed that the accuracy of their model achieved 94%. Wang et al. [121] proposed a Neural Network-based translation model to address the domain-specific rare word problem when carrying out software localization. They trained an RNN encoder-decoder framework and enhanced it by adding linguistic information. Nguyen et al. [92] proposed a DL-based language model, Dnn4C, which augmented the local context of lexical code elements with both syntactic and type contexts by using an FNN model. Empirical evaluation on code completion showed that Dnn4C improved accuracy by 24.9% on average over four baseline approaches.

7.5 Software testing and debugging

7.5.1 Defect prediction. Defect prediction is the most extensive and active research topic in use of DL techniques in software maintenance. Almost 30% of primary studies focused on identifying defects [4, 18, 74, 111, 123, 133, 147].

Metrics-based 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. Tong et al. [111] proposed a novel two-stage approach, SDAEsT, which is build based on stacked denoising autoencoders (SDAEs) and a two-stage ensemble (TSE) learning strategy. Specifically, in the first stage, they used SDAEs to extract more robust and representative features. To mitigate the impact on the class imbalance problem and eliminate the overfitting problem, they propose TSE learning strategy as the second phase. They evaluated their work using 12 open-source defect datasets. Xu et al. [133] 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. For better utilization of such sequence data, Wen et al. [123] built an RNN model to encode features from change sequences. They considered defect prediction as to the sequence labeling problem and performed fine-grained change analysis to extract six categories of change sequences, covering different aspects of software changes. Their evaluation on 10 public datasets showed that their approach achieved high performance in terms of F1-measure and AUC. To address the same problem, Liu et al. [74] proposed to obtain the Historical Version Sequence of Metrics (HVSM) from various software versions as defect predictors and leveraged RNN to detect defects. Barbez et al. [4] 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.

Semantic-based defect prediction. Wang et al. [117, 118] 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. The evaluation results confirmed that DBN-based semantic features significantly outperformed the previous defect prediction based on traditional features in terms of F1-measure. Similarly, Dam et al. [18] 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.

Just-In-Time (JIT) defect prediction. Hoang et al. [41] 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.

7.5.2 Bug detection. Wan et al. [114] implemented a Supervised Representation Learning Approach (SRLA) based on an autoencoder with double encoding-layers to conduct cross-project Aging-Related Bugs (ARBs) prediction. They compared SRLA with the state-of-the-art approach, TLAP, to prove the effectiveness of SRLA. Wang et al. [122] 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. Based on this intuition, Li et al. [63] trained a DL-based model to detect suspicious return statements. They used a CNN to determine whether a given return statement in source code matched its method signature. To reduce the impact of the lack of negative training data, they converted the correct return statements in real-world projects to incorrect ones. Li et al. [66] proposed an AIOps solution for identifying node failures for an ultra-large-scale cloud computing platform at Alibaba.

7.5.3 Vulnerability detection. Dam et al. [19] described a novel approach 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. Tian et al. [110] proposed to learn the fine-grained representation of binary programs and trained a Gated Recurrent Unit (BGRU) network model for intelligent vulnerability detection. Han et al. [37] 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.

7.5.4 Bug localization. To locate buggy files, Lam et al. [57] 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. Some studies proposed to exploit CNN in the bug localization task [48, 129, 139]. Zhang et al. [139] 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. Huo et al. [48] 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. Xiao et al. [129] 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. Li et al. [65] proposed a novel approach, DeepFL, to learn latent features for precise fault localization, adopting an RNN

model. The evaluation on the benchmark dataset, Defects4J, described that DeepFL significantly outperformed state-of-the-art approaches, i.e., TraPT/FLUCCS. Standard code parsers are of little help, typically resolving syntax errors and their precise location poorly. Santos et al. [105] proposed a new methodology for locating syntax errors and provided some suggestions for possible changes for fixing these errors. Their methodology was of practical use to all developers but especially useful to novices frustrated with incomprehensible syntax errors.

7.5.5 Test case generation. Liu et al. [73] proposed a novel approach to automatically generate the most relevant text of test cases based on the context of use cases for mobile testing. Koo et al. [54] implemented a novel approach, PySE1, to generate the test case. PySE1 tackled the limitations of symbolic execution schemes by proposing a DL-based reinforcement learning algorithm to improve the branch policy for exploring the worse case program execution. Zhao et al. [142] 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 network and generating fake but plausible messages as test cases. Liu et al. [72] proposed a deep natural language processing tool, DeepSQLi, to produce test cases used to detect 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.

7.5.6 Program analysis. Program analysis refers to any examination of source code or program executions that attempt to find patterns or anomalies thought to reveal specific behaviors of the software.

Static analysis. In Android mobile operating systems, applications communicate with each other a message-passing system, namely, Inter-Component Communication (ICC). Many serious security vulnerabilities may occur owing to misuse and abuse of communication links, i.e., ICCs. Zhao et al. [143] presented a new approach to determine communication links between Android applications. They augmented static analysis with DL techniques by encoding data types of the links and calculating the probability of the link existence. To reduce the number of false alarms, Lee et al. [62] trained a CNN model as an automated classifier to learn the lexical patterns in the parts of source code for detecting and classifying false alarms. Due to the impact of high false-positive rates on static analysis tools, Koc et al. [53] 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.

Type inference. Helledoorn et al. [39] developed an automated framework, DeepTyper, a DL-based model to analyze JavaScript language and learn types that naturally occurred in certain contexts. It then provided a type of suggestion when the type checker cannot infer the types of code elements, such as variables and functions. Malik et al. [82] 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.

7.5.7 Testing techniques. Many studies focus on new methods to perform testing, such as for apps [96], games [144], and other software systems [5, 11]. There are also some studies using well-known testing techniques (e.g., fuzzing [17, 30] and mutation testing [83]) for improving the quality of software artifacts. Zheng et al. [144] 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. [5] 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. [96] 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. [83] 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. Godefroid et al. [30] used DL-based statistical machine-learning techniques

to automatically generate input syntax suitable for input fuzzing. Cummins et al. [17] introduced DeepSmith, a novel LSTM-based approach, for reducing the development task when using Fuzzers to discover bugs in compilers. They accelerated compiler validation through the inference of generative models for compiler inputs, and then applied DeepSmith to automatically generate tens of thousands of realistic programs. Finally, they constructed differential testing methodologies on these generated programs for exposing bugs in compilers.

7.5.8 Reverse execution. A decompiler is a tool to reverse the compilation process for examining binaries. Lacomis et al. [56] introduced a novel probabilistic technique, namely Decompiled Identifier Renaming Engine (DIRE), which utilized both lexical and structural information recovered by the decompiler for variable name recovery. They also present a technique for generating corpora suitable for training and evaluating models of decompiled code renaming. Although reverse execution is an effective method to diagnose the root cause of software crashes, some inherent challenges may influence its performance. To address this issue, Mu et al. [89] present a novel DNN, which significantly increased the burden of doing hypothesis testing to track down non-alias relation in binary code and improved memory alias resolution. To achieve this, they first employed an RNN to learn the binary code pattern pertaining to memory access and then inferred the memory region accessed by memory references. Katz et al. [51] noticed that the source code generated by decompilation techniques are difficult for developers to read and understand. To narrow the differences between human-written code and decompiled code, they trained a non-language-specific RNN model to learn properties and patterns in source code for decompiling binary code.

7.6 Software maintenance

There are a lot of studies contributing to increasing maintenance efficiency, such as improving source code, logging information, software energy consumption, etc. [36, 42, 75, 80, 103].

7.6.1 Code clone detection. Code clone detection is a very popular SE task in software maintenance using DL, with around 20% of primary studies concentrating on this research topic.

RNN-based code clone detection. Most studies use RNNs including RtNN [27], RvNN [125], and LSTM [9, 97, 112] to identify clones in source code. White et al. [125] 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. [27] 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. [9] introduced a tree-based code clone detection approach, and traversed ASTs to form data sequences as the input of LSTM. Perez et al. [97] 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, Tufano et al. [112] conducted a series of experiments to demonstrate how DL can automatically learn code similarities from different representations.

FNN-based code clone detection. Some studies adopted FNNs for the code clone detection task [64, 90, 141]. Li et al. [64] implemented a DL-based classifier, CClearner, for detecting function-level code clones by training an FNN. Compared with the approaches not using DL, CClearner achieved competitive clone detection effectiveness with a low time cost. Zhao et al. [141] introduced a novel clone detection approach, which encoded data flow and control flow and into a semantic matrix and designed an FNN structure to measure the functional similarity between semantic representation of each code segment. Nafi et al. [90] proposed a cross-language clone detector without extensive processing of the source code and without the need to generate an intermediate representation. They

trained an FNN model, which can learn different syntactic features of source code across programming languages and identified clones by comparing the similarity of features.

Others. For detecting Type-4 code clones, Yu et al. [135] present a new approach that uses tree-based convolution to detect semantic clones, by capturing both the structural information of a code fragment from its AST and lexical information from code tokens. They also addressed the limitation of an unlimited vocabulary of tokens and models. Wang et al. [120] developed a novel graph-based program representation method, flow-augmented abstract syntax tree (FA-AST), to better capture and leverage control and data flow information. FA-AST augmented original ASTs with explicit control and data flow edges and then adopted two different GNN models (i.e., gated graph neural network (GGNN) and graph matching network (GMN)) to measure the similarity of various code pairs. To effectively capture syntax and semantic information from programs to detect semantic clones, Fang et al. [25] adopted fusion embedding techniques to learn hidden syntactic and semantic features by building a novel joint code representation. They also proposed a new granularity for functional code clone detection called caller-callee method relationships. Finally, they trained a supervised deep learning model to find semantic clones.

7.6.2 Code comment generation. Hu et al. [44] present a new approach that can automatically generate code comments for Java code to help developers better understand the functionality of code segments. They trained a LSTM-based framework to learn the program structure for better comments generation. The context information of the source code was not used and analyzed in previous automated comment summarization techniques. Ciurumelea et al. [16] proposed a semi-automated system to generate code comments by using LSTM. Zhou et al. [148] combined program analysis and natural language processing to build a Dl-based seq2seq model to generate Java code comments. To generate code summarization, Leclair et al. [60] proposed a DL-based model combining texts from code with code structure from an AST. They processed data source as a separate input to reduce the entire dependence on internal documentation of code. Wan et al. [116] noticed that most of the previous work used the Encoder-Decoder architecture to generate code summaries, which omitted the tree structure of source code and introduced some bias when decoding code sequences. To solve these problems, they trained a deep reinforcement learning framework that incorporated an abstract syntax tree structure as well as sequential content of code snippets. They trained this DNN by adopting an advantage reward composed of BLEU metric.

7.6.3 Program repair. Bhatia et al. [6] 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. Through evaluation, their approach was able to repair syntax errors in 60% of submissions and identified functionally correct repairs for 24% submissions. Tufano et al. [113] 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. [124] 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. [79] 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. Liu et al. [71] proposed an automated approach for detecting and refactoring inconsistent method names by using Paragraph Vector and a CNN. Ni et al. [93] exploited the bug fixes of historical bugs to classify bugs into their cause categories based on the intuition that historical information may reflect the bug causes. They first defined the code-related bug classification criterion from the perspective of the cause of bugs and generated ASTs from diff source code to construct fixed trees. Then, they trained Tree-based Convolutional Neural Network (TBCNN) to represent each fixed tree and classified bugs into their cause categories according to the relationship between bug fixes and bug causes.

7.6.4 Source code representation. White et al. [126] conducted an empirical study to adopt DL in software language modeling and highlight the fundamental differences between state-of-the-practice software language models and DL models. Their intuition is that the representation power of the abstractions is the key element of improving the quality of software language models. Therefore, the goal of this study was to improve the quality of the underlying abstractions by using Neural Network Language Models (i.e., Feed-forward neural networks (FNN), RNN) for numerous SE issues. They pinpointed that DL had a strong capability to model semantics and consider rich contexts, allowing it performed better at source code modeling. They evaluated these DL-based language models at a real SE task (i.e., code suggestion). Their evaluation results suggested that their model significantly outperformed the traditional language models on 16,221 Java projects.

Hussain et al. [49] introduced a gated recurrent unit-based model, namely CodeGRU, to model source code by capturing its contextual, syntactical, and structural dependencies. The key innovation of their approach was to performing simple program analysis for capturing the source code context and further employed GRU to learn variable size context while modeling source code. They evaluated CodeGRU on several open-source java projects, and the experimental results verified that the approach alleviated the out of vocabulary issue. Using abstract syntax tree (AST)-based DNNs may induce a long-term dependency problem due to the large size of ASTs. To address this problem, Zhang et al. [137] present an advanced AST-based Neural Network (ASTNN) for source code representation. The advanced ASTNN cut each entire AST into a set of small statement trees, and transform these subtrees into vectors by capturing the lexical and syntactical knowledge of statement trees. They applied a bidirectional RNN model to produce the vector representation of a code snippet. They used ASTNN to detect code clones and classify source code for evaluating its performance. Gill et al. [29] introduced a lightweight framework, ThermoSim, to simulate the thermal behavior of computing nodes in the Cloud Data Center (CDC) and measure the effect of temperatures on key performance parameters. They extended the previous framework, i.e., the CloudSim toolkit by presenting an RNN-based temperature predictor for helping to analyze the performance of some key parameters. The final results demonstrated that ThermoSim accurately modeled and simulated the thermal behavior of a CDC in terms of energy consumption, time, cost, and memory usage.

7.6.5 Code classification. Bui et al. [22] 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. [59] proposed a set of adaptations to a state-of-the-art neural classification algorithm and conducted two evaluations.

7.6.6 Code smell detection. Fakhoury et al. [24] reported their experience in building an automatic linguistic anti-pattern detection using DNNs. They trained several traditional machine learning and DNNs to identify linguistic anti-patterns. A big challenge for DL-based code smell detection is the lack of a large number of labeled datasets, and thus Liu et al. [68] present a DL-based approach to automatically generating labeled training data for DL-based classifiers. They applied their approach to detecting four common and well-known code smells, i.e., feature envy, long method, large class, and misplaced class.

7.6.7 Self-Admitted Technical Debt (SATD) detection. Technical debt (TD) is a metaphor to reflect the tradeoff 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. [102] 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 model's prediction results. Zampetti et al. [136] 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.

7.6.8 Code review. Siow et al. [106] 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. [34] 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.

7.6.9 Software quality evaluation. Variety evaluation metrics can be used to describe the quality of software products [101].

Software trustworthiness. 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. [109] constructed behaviour trajectory matrices to represent the behaviour 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.

Readability. Mi et al. [86] 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.

Maintainability. Kumar et al. [55] 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.

7.7 Software management

7.7.1 Effort estimation. Since only 39% of software projects are finished and published on time relative to the duration planned originally [7, 78], it is necessary to assess the development cost to achieve reliable software within development schedule and budget. Lopez et al. [78] 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. [15] 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. [99] 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.[94] 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.

7.7.2 Software repository mining. Some primary studies use DL techniques to mine the contents in different software repositories [35, 58, 81]. In this section, we introduce three most widely used repositories, i.e., Stack Overflow (SO) [70, 146], GitHub [50], and Youtube [95].

Mining Stack Overflow (SO). The questions and answers in SO contain a great deal of useful information that is beneficial for programmers to address some tough problems when implementing software products. Considering a question and its answers in Stack Overflow as a knowledge unit, Xu et al. [131, 132] extracted the knowledge units and analyzed the potential semantic relationship between Q and A in each unit. They formulated the problem of predicting semantically linkable knowledge units as a multi-class classification problem and adopted a CNN model combining with word-embedding to capture and classify document-level semantics of knowledge units. Chen et al. [13] 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. [134] 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. To validate whether DL was the best solution in all research topics, Menzies et al. [85] conducted a case study to explore faster approaches for text mining SO. They compared nine different solutions including traditional machine learning algorithms and DL algorithms and noticed that a tuned SVM performs similarly to a deep learner and was over 500 times faster than DL-based models. Zhang et al. [138] performed an empirical study to mine the posts in SO for investigating what potential challenges developers face when using DL. They also built a classification model to quantify the distribution of different Sort of DL related questions. They summarized the three most frequently asked questions and provided five future research directions. Since answers to a question may contain extensive irrelevant information, Wang et al. [119] proposed a DL-based approach, DeepTip, using different CNN architectures to extract short practical and useful tips and filtered useless information from developer answers. They conducted a user study to prove the effectiveness of their approach and the extensive empirical experiments demonstrated that DeepTip can extract useful information from answers with high precision and coverage, significantly outperforming two state-of-the-art approaches.

Mining GitHub. Huang et al. [46] 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. Xie et al. [130] 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.[104] 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. They compared DeepLink with the state-of-the-art approach on 10 GitHub Java projects to evaluate the effectiveness of DeepLink. Liu et al. [76] 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.

Mining Youtube. Ott et al. [95] employed CNN and AutoEncoder to identify Java code from videos on Youtube. Their approach was able to identify the presence of typeset and handwritten source code in thousands of video images with 85.6%-98.6% accuracy based on syntactic and contextual features learned through DL techniques. Zhao et al. [140] present a new technique for recognizing workflow actions in programming screencasts. They

collected programming screencasts from Youtube and trained a CNN model to identify nine classes of frequent developer actions by employing the image differencing technique and training a CNN model.

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, ranking task, and generation task – and classified all primary studies based on the task types. Most studies can be mapped to classification tasks, and only 11% of primary studies are mapped to ranking tasks.
(3) Software testing and software maintenance are the two SE activities containing the most related studies and include 17 specific SE research topics in which DL techniques were used.

8 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.

9 CHALLENGES AND OPPORTUNITIES

Using DL in more SE activities. 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.

The transparency of DL. In this study, we discussed 142 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. 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.

DL in real scenarios. We analyzed the source of datasets used for training DNNs in RQ3 and noticed that only 4% studies used industry datasets to train and evaluate their proposed models. In fact, most studies contribute to addressing real-world SE issues, but the novel solutions or DL-based models have not been evaluated on industry data. There is a room for more industry-academia collaboration so that the proposed models can be validated on real industry data (which can be of very different nature than open-source data.

Data hungry. 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 meta-learning.

Performance of DL and traditional techniques. 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? These questions are almost unanswered and neglected by most researchers. A potential opportunity is to answer these questions. researchers can conduct empirical studies to investigate what SE tasks or environments are suitable for DL and compare the performance between DL and traditional techniques in some important SE research topics where most of Dl algorithms were applied.

10 CONCLUSION

This work performed a SLR on 142 primary studies related to DL for SE from 21 publication venues, including conference proceedings, symposiums, and journals. We established 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, three 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 processing, model optimization, and model evaluation in RQ3. In RQ4, we analyzed the distribution of DL techniques used in different SE activities, 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 23 SE topics, covering 6 SE activities. Finally, we identified a set of current challenges that still need to be addressed in future work on using DLs in SE.

REFERENCES

[1] Aysh Al-Hroob, Ayad Tareq Imam, and Rawan Al-Heisa. 2018. The use of artificial neural networks for extracting actions and actors from requirements document. *IST* 101 (2018), 1–15.

[2] Mohammad Alahmadi, Abdulkarim Khormi, Biswas Parajuli, Jonathan Hassel, Sonia Haiduc, and Piyush Kumar. 2020. Code Localization in Programming Screencasts. *ESE* 25, 2 (2020), 1536–1572.

[3] Lingfeng Bao, Zhenchang Xing, Xin Xia, David Lo, Minghui Wu, and Xiaohu Yang. 2020. psc2code: Denoising Code Extraction from Programming Screencasts. *TOSEM* 29, 3 (2020), 1–38.

[4] Antoine Barbez, Foutse Khomh, and Yann-Gaël Guéhéneuc. 2019. Deep Learning Anti-patterns from Code Metrics History. In *ICSME*. IEEE, 114–124.

[5] Raja Ben Abdessalem, Shiva Nejati, Lionel C Briand, and Thomas Stifter. 2016. Testing advanced driver assistance systems using multi-objective search and neural networks. In *ASE*. 63–74.

[6] Sahil Bhatia, Pushmeet Kohli, and Rishabh Singh. 2018. Neuro-symbolic program corrector for introductory programming assignments. In *ICSE*. IEEE, 60–70.

[7] Manjubala Bisi and Neeraj Kumar Goyal. 2016. Software development efforts prediction using artificial neural network. *IETS* 10, 3 (2016), 63–71.

[8] Pierre Bourque, Richard E Fairley, et al. 2014. *Guide to the software engineering body of knowledge (SWEBOK (R)): Version 3.0*. IEEE Computer Society Press.

[9] Lutz Büch and Artur Andrzejak. 2019. Learning-based recursive aggregation of abstract syntax trees for code clone detection. In *SANER*. IEEE, 95–104.

[10] Jose Cambronero, Hongyu Li, Seohyun Kim, Koushik Sen, and Satish Chandra. 2019. When deep learning met code search. In *FSE*. 964–974.

[11] Chao Chen, Wenrui Diao, Yingpei Zeng, Shanqing Guo, and Chengyu Hu. 2018. DRLgencert: Deep learning-based automated testing of certificate verification in SSL/TLS implementations. In *ICSME*. IEEE, 48–58.

[12] Chunyang Chen, Ting Su, Guozhu Meng, Zhenchang Xing, and Yang Liu. 2018. From ui design image to gui skeleton: a neural machine translator to bootstrap mobile gui implementation. In *ICSE*. 665–676.

[13] Guibin Chen, Chunyang Chen, Zhenchang Xing, and Bowen Xu. 2016. Learning a dual-language vector space for domain-specific cross-lingual question retrieval. In *ASE*. IEEE, 744–755.

[14] Jieshan Chen, Chunyang Chen, Zhenchang Xing, Xin Xia, Liming Zhu, John Grundy, and Jinshui Wang. 2020. Wireframe-based UI design search through image autoencoder. *TOSEM* 29, 3 (2020), 1–31.

[15] Morakot Choetkiertikul, Hoa Khanh Dam, Truyen Tran, Trang Pham, Aditya Ghose, and Tim Menzies. 2018. A deep learning model for estimating story points. *TSE* 45, 7 (2018), 637–656.

[16] Adelina Ciurumelea, Sebastian Proksch, and Harald C Gall. 2020. Suggesting Comment Completions for Python using Neural Language Models. In *SANER*. IEEE, 456–467.

[17] Chris Cummins, Pavlos Petoumenos, Alastair Murray, and Hugh Leather. 2018. Compiler fuzzing through deep learning. In *ISSTA*. 95–105.

[18] Hoa Khanh Dam, Trang Pham, Shien Wee Ng, Truyen Tran, John Grundy, Aditya Ghose, Taeksu Kim, and Chul-Joo Kim. 2019. Lessons learned from using a deep tree-based model for software defect prediction in practice. In *MSR*. IEEE, 46–57.

[19] Hoa Khanh Dam, Truyen Tran, Trang Thi Minh Pham, Shien Wee Ng, John Grundy, and Aditya Ghose. 2018. Automatic feature learning for predicting vulnerable software components. *TSE* (2018).

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

[21] Jayati Deshmukh, KM Annervaz, Sanjay Podder, Shubhashis Sengupta, and Neville Dubash. 2017. Towards accurate duplicate bug retrieval using deep learning techniques. In *ICSME*. IEEE, 115–124.

[22] Bui Nghi DQ, Yijun Yu, and Lingxiao Jiang. 2019. Bilateral dependency neural networks for cross-language algorithm classification. In *SANER*. IEEE, 422–433.

[23] Hasan Ferit Enişer and Alper Sen. 2020. Virtualization of stateful services via machine learning. *SQJ* 28, 1 (2020), 283–306.

[24] Sarah Fakhoury, Venera Arnaoudova, Cedric Noiseux, Foutse Khomh, and Giuliano Antoniol. 2018. Keep it simple: Is deep learning good for linguistic smell detection?. In *SANER*. IEEE, 602–611.

[25] Chunrong Fang, Zixi Liu, Yangyang Shi, Jeff Huang, and Qingkai Shi. 2020. Functional code clone detection with syntax and semantics fusion learning. In *ISSTA*. 516–527.

[26] Sa Gao, Chunyang Chen, Zhenchang Xing, Yukun Ma, Wen Song, and Shang-Wei Lin. 2019. A neural model for method name generation from functional description. In *SANER*. IEEE, 414–421.

[27] Yi Gao, Zan Wang, Shuang Liu, Lin Yang, Wei Sang, and Yuanfang Cai. 2019. TECCD: A Tree Embedding Approach for Code Clone Detection. In *ICSME*. IEEE, 145–156.

[28] Necmiye Genc-Nayebi and Alain Abran. 2017. A systematic literature review: Opinion mining studies from mobile app store user reviews. *JSS* 125 (2017), 207–219.

[29] Sukhpal Singh Gill, Shreshth Tuli, Adel Nadjaran Toosi, Felix Cuadrado, Peter Garraghan, Rami Bahsoon, Hanan Lutfiyya, Rizos Sakellariou, Omer Rana, Schahram Dustdar, et al. 2020. ThermoSim: Deep learning based framework for modeling and simulation of thermal-aware resource management for cloud computing environments. *JSS* (2020), 110596.

[30] Patrice Godefroid, Hila Peleg, and Rishabh Singh. 2017. Learn&fuzz: Machine learning for input fuzzing. In *ASE*. IEEE, 50–59.

[31] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep learning*. MIT press.

[32] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *ICSE*. IEEE, 933–944.

[33] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API learning. In *FSE*. 631–642.

[34] Chenkai Guo, Dengrong Huang, Naipeng Dong, Quanqi Ye, Jing Xu, Yaqing Fan, Hui Yang, and Yifan Xu. 2019. Deep review sharing. In *SANER*. IEEE, 61–72.

[35] Chenkai Guo, Weijing Wang, Yanfeng Wu, Naipeng Dong, Quanqi Ye, Jing Xu, and Sen Zhang. 2019. Systematic comprehension for developer reply in mobile system forum. In *SANER*. IEEE, 242–252.

[36] Huong Ha and Hongyu Zhang. 2019. Deepperf: performance prediction for configurable software with deep sparse neural network. In *ICSE*. IEEE, 1095–1106.

[37] Zhuobing Han, Xiaohong Li, Zhenchang Xing, Hongtao Liu, and Zhiyong Feng. 2017. Learning to predict severity of software vulnerability using only vulnerability description. In *ICSME*. IEEE, 125–136.

[38] Douglas M Hawkins. 2004. The problem of overfitting. *Journal of chemical information and computer sciences* 44, 1 (2004), 1–12.

[39] Vincent J Hellendoorn, Christian Bird, Earl T Barr, and Miltiadis Allamanis. 2018. Deep learning type inference. In *FSE*. 152–162.

[40] 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.

[41] Thong Hoang, Hoa Khanh Dam, Yasutaka Kamei, David Lo, and Naoyasu Ubayashi. 2019. DeepJIT: an end-to-end deep learning framework for just-in-time defect prediction. In *MSR*. IEEE, 34–45.

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

[43] Seyedrebvar Hosseini, Burak Turhan, and Dimuthu Gunarathna. 2017. A systematic literature review and meta-analysis on cross project defect prediction. *TSE* 45, 2 (2017), 111–147.

[44] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *ICPC*. IEEE, 200–20010.

[45] Qing Huang, An Qiu, Maosheng Zhong, and Yuan Wang. 2020. A Code-Description Representation Learning Model Based on Attention. In *SANER*. IEEE, 447–455.

[46] Qiao Huang, Xin Xia, David Lo, and Gail C Murphy. 2018. Automating intention mining. *TSE* (2018).

[47] Rubing Huang, Weifeng Sun, Yinyin Xu, Haibo Chen, Dave Towey, and Xin Xia. 2019. A survey on adaptive random testing. *TSE* (2019).

[48] Xuan Huo, Ferdian Thung, Ming Li, David Lo, and Shu-Ting Shi. 2019. Deep transfer bug localization. *TSE* (2019).

[49] Yasir Hussain, Zhiqiu Huang, Yu Zhou, and Senzhang Wang. 2020. CodeGRU: Context-aware deep learning with gated recurrent unit for source code modeling. *IST* (2020), 106309.

[50] Siyuan Jiang, Ameer Armaly, and Collin McMillan. 2017. Automatically generating commit messages from diffs using neural machine translation. In *ASE*. IEEE, 135–146.

[51] Deborah S Katz, Jason Ruchti, and Eric Schulte. 2018. Using recurrent neural networks for decompilation. In *SANER*. IEEE, 346–356.

[52] 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.

[53] Ugur Koc, Shiyi Wei, Jeffrey S Foster, Marine Carpuat, and Adam A Porter. 2019. An empirical assessment of machine learning approaches for triaging reports of a java static analysis tool. In *ICST*. IEEE, 288–299.

[54] Jinkyu Koo, Charitha Saumya, Milind Kulkarni, and Saurabh Bagchi. 2019. Pyse: Automatic worst-case test generation by reinforcement learning. In *ICST*. IEEE, 136–147.

[55] Lov Kumar and Santanu Ku Rath. 2016. Hybrid functional link artificial neural network approach for predicting maintainability of object-oriented software. *JSS* 121 (2016), 170–190.

[56] Jeremy Lacomis, Pengcheng Yin, Edward Schwartz, Miltiadis Allamanis, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. 2019. Dire: A neural approach to decompiled identifier naming. In *ASE*. IEEE, 628–639.

[57] An Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. 2017. Bug localization with combination of deep learning and information retrieval. In *ICPC*. IEEE, 218–229.

[58] Tien-Duy B Le and David Lo. 2018. Deep specification mining. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 106–117.

[59] Alexander LeClair, Zachary Eberhart, and Collin McMillan. 2018. Adapting neural text classification for improved software categorization. In *ICSME*. IEEE, 461–472.

[60] Alexander LeClair, Siyuan Jiang, and Collin McMillan. 2019. A neural model for generating natural language summaries of program subroutines. In *ICSE*. IEEE, 795–806.

[61] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *nature* 521, 7553 (2015), 436–444.

[62] Seongmin Lee, Shin Hong, Jungbae Yi, Taeksu Kim, Chul-Joo Kim, and Shin Yoo. 2019. Classifying false positive static checker alarms in continuous integration using convolutional neural networks. In *ICST*. IEEE, 391–401.

[63] Guangjie Li, Hui Liu, Jiahao Jin, and Qasim Umer. 2020. Deep Learning Based Identification of Suspicious Return Statements. In *SANER*. IEEE, 480–491.

[64] Liuqing Li, He Feng, Wenjie Zhuang, Na Meng, and Barbara Ryder. 2017. Cclearner: A deep learning-based clone detection approach. In *ICSME*. IEEE, 249–260.
[65] Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. 2019. Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization. In *ISSTA*. 169–180.
[66] Yangguang Li, Zhen Ming Jiang, Heng Li, Ahmed E Hassan, Cheng He, Ruirui Huang, Zhengda Zeng, Mian Wang, and Pinan Chen. 2020. Predicting Node Failures in an Ultra-large-scale Cloud Computing Platform: an AIOps Solution. *TOSEM* 29, 2 (2020), 1–24.
[67] Chao Liu, Cuiyun Gao, Xin Xia, David Lo, John Grundy, and Xiaohu Yang. 2020. On the Replicability and Reproducibility of Deep Learning in Software Engineering. *arXiv preprint arXiv:2006.14244* (2020).
[68] Hui Liu, Jiahao Jin, Zhifeng Xu, Yifan Bu, Yanzhen Zou, and Lu Zhang. 2019. Deep learning based code smell detection. *TSE* (2019).
[69] Hui Liu, Zhifeng Xu, and Yanzhen Zou. 2018. Deep learning based feature envy detection. In *ASE*. 385–396.
[70] Jin Liu, Pingyi Zhou, Zijiang Yang, Xiao Liu, and John Grundy. 2018. FastTagRec: fast tag recommendation for software information sites. *ASEJ* 25, 4 (2018), 675–701.
[71] Kui Liu, Dongsun Kim, Tegawendé F Bissyandé, Taeyoung Kim, Kisub Kim, Anil Koyuncu, Suntae Kim, and Yves Le Traon. 2019. Learning to spot and refactor inconsistent method names. In *ICSE*. IEEE, 1–12.
[72] Muyang Liu, Ke Li, and Tao Chen. 2020. DeepSQLi: Deep Semantic Learning for Testing SQL Injection. *arXiv preprint arXiv:2005.11728* (2020).
[73] Peng Liu, Xiangyu Zhang, Marco Pistoia, Yunhui Zheng, Manoel Marques, and Lingfei Zeng. 2017. Automatic text input generation for mobile testing. In *ICSE*. IEEE, 643–653.
[74] Yibin Liu, Yanhui Li, Jianbo Guo, Yuming Zhou, and Baowen Xu. 2018. Connecting software metrics across versions to predict defects. In *SANER*. IEEE, 232–243.
[75] Zhongxin Liu, Xin Xia, David Lo, Zhenchang Xing, Ahmed E Hassan, and Shanping Li. 2019. Which variables should i log? *TSE* (2019).
[76] Zhongxin Liu, Xin Xia, Christoph Treude, David Lo, and Shanping Li. 2019. Automatic generation of pull request descriptions. In *ASE*. IEEE, 176–188.
[77] Zhongxin Liu, Xin Xia, Meng Yan, and Shanping Li. [n. d.]. Automating Just-In-Time Comment Updating. In *ASE*.
[78] Cuauhtémoc López-Martín and Alain Abran. 2015. Neural networks for predicting the duration of new software projects. *JSS* 101 (2015), 127–135.
[79] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. CoCoNuT: combining context-aware neural translation models using ensemble for program repair. In *ISSTA*. 101–114.
[80] Suyu Ma, Zhenchang Xing, Chunyang Chen, Cheng Chen, Lizhen Qu, and Guoqiang Li. 2019. Easy-to-Deploy API Extraction by Multi-Level Feature Embedding and Transfer Learning. *TSE* (2019).
[81] Alvi Mahadi, Karan Tongay, and Neil A Ernst. 2020. Cross-Dataset Design Discussion Mining. In *SANER*. IEEE, 149–160.
[82] Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. 2019. Nl2type: inferring javascript function types from natural language information. In *ICSE*. IEEE, 304–315.
[83] Dongyu Mao, Lingchao Chen, and Lingming Zhang. 2019. An extensive study on cross-project predictive mutation testing. In *ICST*. IEEE, 160–171.
[84] 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.
[85] Tim Menzies, Suvodeep Majumder, Nikhila Balaji, Katie Brey, and Wei Fu. 2018. 500+ times faster than deep learning:(a case study exploring faster methods for text mining stackoverflow). In *MSR*. IEEE, 554–563.
[86] Qing Mi, Jacky Keung, Yan Xiao, Solomon Mensah, and Yujin Gao. 2018. Improving code readability classification using convolutional neural networks. *IST* 104 (2018), 60–71.
[87] 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.
[88] Kevin Patrick Moran, Carlos Bernal-Cárdenas, Michael Curcio, Richard Bonett, and Denys Poshyvanyk. 2018. Machine learning-based prototyping of graphical user interfaces for mobile apps. *TSE* (2018).
[89] Dongliang Mu, Wenbo Guo, Alejandro Cuevas, Yueqi Chen, Jinxuan Gai, Xinyu Xing, Bing Mao, and Chengyu Song. 2019. RENN: efficient reverse execution with neural-network-assisted alias analysis. In *ASE*. IEEE, 924–935.
[90] Kawser Wazed Nafi, Tonny Shekha Kar, Banani Roy, Chanchal K Roy, and Kevin A Schneider. 2019. CLCDSA: cross language code clone detection using syntactical features and API documentation. In *ASE*. IEEE, 1026–1037.
[91] 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.
[92] Anh Tuan Nguyen, Trong Duc Nguyen, Hung Dang Phan, and Tien N Nguyen. 2018. A deep neural network language model with contexts for source code. In *SANER*. IEEE, 323–334.

[93] Zhen Ni, Bin Li, Xiaobing Sun, Tianhao Chen, Ben Tang, and Xinchen Shi. 2020. Analyzing bug fix for automatic bug cause classification. *JSS* 163 (2020), 110538.
[94] Mirosław Ochodek, Sylwia Kopczyńska, and Miroslaw Staron. 2020. Deep learning model for end-to-end approximation of COSMIC functional size based on use-case names. *IST* (2020), 106310.
[95] Jordan Ott, Abigail Atchison, Paul Harnack, Adrienne Bergh, and Erik Linstead. 2018. A deep learning approach to identifying source code in images and video. In *MSR*. IEEE, 376–386.
[96] Minxue Pan, An Huang, Guoxin Wang, Tian Zhang, and Xuandong Li. 2020. Reinforcement learning based curiosity-driven testing of Android applications. In *ISSTA*. 153–164.
[97] Daniel Perez and Shigeru Chiba. 2019. Cross-language clone detection by learning over abstract syntax trees. In *MSR*. IEEE, 518–528.
[98] Kai Petersen, Sairam Vakkalanka, and Ludwik Kuzniarz. 2015. Guidelines for conducting systematic mapping studies in software engineering: An update. *IST* 64 (2015), 1–18.
[99] Passakorn Phannachitta. 2020. On an optimal analogy-based software effort estimation. *IST* (2020), 106330.
[100] Florian Pudlitz, Florian Brokhausen, and Andreas Vogelsang. 2019. Extraction of system states from natural language requirements. In *RE*. IEEE, 211–222.
[101] Pooja Rani and GS Mahapatra. 2018. Neural network for software reliability analysis of dynamically weighted NHPP growth models with imperfect debugging. *STVR* 28, 5 (2018), e1663.
[102] Xiaoxue Ren, Zhenchang Xing, Xin Xia, David Lo, Xinyu Wang, and John Grundy. 2019. Neural network-based detection of self-admitted technical debt: from performance to explainability. *TOSEM* 28, 3 (2019), 1–45.
[103] Stephen Romansky, Neil C Borle, Shaiful Chowdhury, Abram Hindle, and Russ Greiner. 2017. Deep green: Modelling time-series of software energy consumption. In *ICSME*. IEEE, 273–283.
[104] Hang Ruan, Bihuan Chen, Xin Peng, and Wenyun Zhao. 2019. DeepLink: Recovering issue-commit links based on deep learning. *JSS* 158 (2019), 110406.
[105] Eddie Antonio Santos, Joshua Charles Campbell, Dhvani Patel, Abram Hindle, and José Nelson Amaral. 2018. Syntax and sensibility: Using language models to detect and correct syntax errors. In *SANER*. IEEE, 311–322.
[106] Jing Kai Siow, Cuiyun Gao, Lingling Fan, Sen Chen, and Yang Liu. 2020. CORE: Automating Review Recommendation for Code Changes. In *SANER*. IEEE, 284–295.
[107] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research* 15, 1 (2014), 1929–1958.
[108] Hannes Thaller, Lukas Linsbauer, and Alexander Egyed. 2019. Feature maps: A comprehensible software representation for design pattern detection. In *SANER*. IEEE, 207–217.
[109] Junfeng Tian and Yuhui Guo. 2020. Software trustworthiness evaluation model based on a behaviour trajectory matrix. *IST* 119 (2020), 106233.
[110] Junfeng Tian, Wenjing Xing, and Zhen Li. 2020. BVDetector: A program slice-based binary code vulnerability intelligent detection system. *IST* 123 (2020), 106289.
[111] Haonan Tong, Bin Liu, and Shihai Wang. 2018. Software defect prediction using stacked denoising autoencoders and two-stage ensemble learning. *IST* 96 (2018), 94–111.
[112] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2018. Deep learning similarities from different representations of source code. In *MSR*. IEEE, 542–553.
[113] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *TOSEM* 28, 4 (2019), 1–29.
[114] Xiaohui Wan, Zheng Zheng, Fangyun Qin, Yu Qiao, and Kishor S Trivedi. 2019. Supervised Representation Learning Approach for Cross-Project Aging-Related Bug Prediction. In *ISSRE*. IEEE, 163–172.
[115] Yao Wan, Jingdong Shu, Yulei Sui, Guandong Xu, Zhou Zhao, Jian Wu, and Philip Yu. 2019. Multi-modal attention network learning for semantic source code retrieval. In *ASE*. IEEE, 13–25.
[116] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S Yu. 2018. Improving automatic source code summarization via deep reinforcement learning. In *ASE*. 397–407.
[117] Song Wang, Taiyue Liu, Jaechang Nam, and Lin Tan. 2018. Deep semantic feature learning for software defect prediction. *TSE* (2018).
[118] Song Wang, Taiyue Liu, and Lin Tan. 2016. Automatically learning semantic features for defect prediction. In *ICSE*. IEEE, 297–308.
[119] Shaohua Wang, NhatHai Phan, Yan Wang, and Yong Zhao. 2019. Extracting API tips from developer question and answer websites. In *MSR*. IEEE, 321–332.
[120] Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. 2020. Detecting Code Clones with Graph Neural Network and Flow-Augmented Abstract Syntax Tree. In *SANER*. IEEE, 261–271.
[121] Xu Wang, Chunyang Chen, and Zhenchang Xing. 2019. Domain-specific machine translation with recurrent neural network for software localization. *ESE* 24, 6 (2019), 3514–3545.

[122] Yaohui Wang, Hui Xu, Yangfan Zhou, Michael R Lyu, and Xin Wang. 2019. Textout: Detecting Text-Layout Bugs in Mobile Apps via Visualization-Oriented Learning. In *ISSRE*. IEEE, 239–249.
[123] Ming Wen, Rongxin Wu, and Shing-Chi Cheung. 2018. How well do change sequences predict defects? sequence learning from software changes. *TSE* (2018).
[124] Martin White, Michele Tufano, Matias Martinez, Martin Monperrus, and Denys Poshyvanyk. 2019. Sorting and transforming program repair ingredients via deep learning code similarities. In *SANER*. IEEE, 479–490.
[125] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *ASE*. IEEE, 87–98.
[126] Martin White, Christopher Vendome, Mario Linares-Vásquez, and Denys Poshyvanyk. 2015. Toward deep learning software repositories. In *MSR*. IEEE, 334–345.
[127] Karl Wiegers and Joy Beatty. 2013. *Software requirements*. Pearson Education.
[128] Jonas Paul Winkler, Jannis Grönberg, and Andreas Vogelsang. 2019. Predicting How to Test Requirements: An Automated Approach. In *RE*. IEEE, 120–130.
[129] Yan Xiao, Jacky Keung, Kwabena E Bennin, and Qing Mi. 2019. Improving bug localization with word embedding and enhanced convolutional neural networks. *IST* 105 (2019), 17–29.
[130] Rui Xie, Long Chen, Wei Ye, Zhiyu Li, Tianxiang Hu, Dongdong Du, and Shikun Zhang. 2019. DeepLink: A code knowledge graph based deep learning approach for issue-commit link recovery. In *SANER*. IEEE, 434–444.
[131] Bowen Xu, Amirreza Shirani, David Lo, and Mohammad Amin Alipour. 2018. Prediction of relatedness in stack overflow: deep learning vs. SVM: a reproducibility study. In *ESEM*. 1–10.
[132] Bowen Xu, Deheng Ye, Zhenchang Xing, Xin Xia, Guibin Chen, and Shanping Li. 2016. Predicting semantically linkable knowledge in developer online forums via convolutional neural network. In *ASE*. IEEE, 51–62.
[133] Zhou Xu, Shuai Li, Jun Xu, Jin Liu, Xiapu Luo, Yifeng Zhang, Tao Zhang, Jacky Keung, and Yutian Tang. 2019. LDFR: Learning deep feature representation for software defect prediction. *JSS* 158 (2019), 110402.
[134] Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to mine aligned code and natural language pairs from stack overflow. In *MSR*. IEEE, 476–486.
[135] Hao Yu, Wing Lam, Long Chen, Ge Li, Tao Xie, and Qianxiang Wang. 2019. Neural detection of semantic code clones via tree-based convolution. In *ICPC*. IEEE, 70–80.
[136] Fiorella Zampetti, Alexander Serebrenik, and Massimiliano Di Penta. 2020. Automatically learning patterns for self-admitted technical debt removal. In *SANER*. IEEE, 355–366.
[137] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In *ICSE*. IEEE, 783–794.
[138] Tianyi Zhang, Cuiyun Gao, Lei Ma, Michael Lyu, and Miryung Kim. 2019. An empirical study of common challenges in developing deep learning applications. In *ISSRE*. IEEE, 104–115.
[139] Zhuo Zhang, Yan Lei, Xiaoguang Mao, and Panpan Li. 2019. CNN-FL: An effective approach for localizing faults using convolutional neural networks. In *SANER*. IEEE, 445–455.
[140] Dehai Zhao, Zhenchang Xing, Chunyang Chen, Xin Xia, and Guoqiang Li. 2019. ActionNet: vision-based workflow action recognition from programming screencasts. In *ICSE*. IEEE, 350–361.
[141] Gang Zhao and Jeff Huang. 2018. Deepsim: deep learning code functional similarity. In *FSE*. 141–151.
[142] Hui Zhao, Zhihui Li, Hansheng Wei, Jianqi Shi, and Yanhong Huang. 2019. SeqFuzzer: An industrial protocol fuzzing framework from a deep learning perspective. In *ICST*. IEEE, 59–67.
[143] Jinman Zhao, Aws Albarghouthi, Vaibhav Rastogi, Somesh Jha, and Damien Octeau. 2018. Neural-augmented static analysis of Android communication. In *FSE*. 342–353.
[144] Yan Zheng, Xiaofei Xie, Ting Su, Lei Ma, Jianye Hao, Zhaopeng Meng, Yang Liu, Ruimin Shen, Yingfeng Chen, and Changjie Fan. 2019. Wuji: Automatic online combat game testing using evolutionary deep reinforcement learning. In *ASE*. IEEE, 772–784.
[145] Cheng Zhou, Bin Li, and Xiaobing Sun. 2020. Improving software bug-specific named entity recognition with deep neural network. *JSS* (2020), 110572.
[146] Pingyi Zhou, Jin Liu, Xiao Liu, Zijiang Yang, and John Grundy. 2019. Is deep learning better than traditional approaches in tag recommendation for software information sites? *IST* 109 (2019), 1–13.
[147] Tianchi Zhou, Xiaobing Sun, Xin Xia, Bin Li, and Xiang Chen. 2019. Improving defect prediction with deep forest. *IST* 114 (2019), 204–216.
[148] Yu Zhou, Xin Yan, Wenhua Yang, Taolue Chen, and Zhiqiu Huang. 2019. Augmenting Java method comments generation with context information based on neural networks. *JSS* 156 (2019), 328–340.