IET Software

REVIEW

IET | The Institution of Engineering and Technology | WILEY

A systematic mapping study of source code representation for deep learning in software engineering

Hazem Peter Samoaa[1] ![ORCID] | Firas Bayram[2] | Pasquale Salza[3] | Philipp Leitner[1]

[1]Software Engineering and Interaction Design Division, Chalmers | University of Gothenburg, Gothenburg, Sweden

[2]Department of Mathematics and Computer Science, Karlstad University, Karlstad, Sweden

[3]Software Evolution & Architecture Lab, University of Zurich, Zurich, Switzerland

Correspondence

Hazem Peter Samoaa, Software Engineering and Interaction Design Division, Chalmers | University of Gothenburg, Lindholmsplatsen 1, Kuggen building, Room 22, Floor 3, 417 56, Gothenburg, Sweden.
Email: samoaa@chalmers.se

Abstract

The usage of deep learning (DL) approaches for software engineering has attracted much attention, particularly in source code modelling and analysis. However, in order to use DL, source code needs to be formatted to fit the expected input form of DL models. This problem is known as source code representation. Source code can be represented via different approaches, most importantly, the tree-based, token-based, and graph-based approaches. We use a systematic mapping study to investigate i detail the representation approaches adopted in 103 studies that use DL in the context of software engineering. Thus, studies are collected from 2014 to 2021 from 14 different journals and 27 conferences. We show that each way of representing source code can provide a different, yet orthogonal view of the same source code. Thus, different software engineering tasks might require different (combinations of) code representation approaches, depending on the nature and complexity of the task. Particularly, we show that it is crucial to define whether the DL approach requires lexical, syntactical, or semantic code information. Our analysis shows that a wide range of different representations and combinations of representations (hybrid representations) are used to solve a wide range of common software engineering problems. However, we also observe that current research does not generally attempt to transfer existing representations or models to other studies even though there are other contexts in which these representations and models may also be useful. We believe that there is potential for more reuse and the application of transfer learning when applying DL to software engineering tasks.

1 | INTRODUCTION

Machine learning (ML), and nowadays deep learning (DL), is increasingly used by software engineering (SE) researchers and practitioners for a wide range of tasks. Examples include source code classification [1–3], code clone detection [4–6], bug detection [7–9], or code summarisation [10–12]. The current interest in DL is enabled by the wide availability of large-scale data (e.g., through open-source systems hosted on platforms, such as GitHub). Particularly, DL is interesting to researchers as it promises good results (e.g., highly accurate code clone detection) without the need for cumbersome (and often limiting) explicit feature extraction process from the raw data as it is required by traditional machine learning models [13].

In classical machine learning approaches, a considerable amount of effort goes to the design of proper ways to capture the structure of the data, that is, feature engineering, which is a "human" effort in most of the cases. This is the reason why, in the last decade, attention in machine learning is moving to 'representation learning', which consists of automatically extracting or learning features without the need of human feature engineering. In representation learning, feature engineering and selection phases are taken away and replaced with deep learning neural networks. DL models are composed of multiple layers to learn data representations with multiple higher levels of abstraction [14]. The networks are supposed to learn the data representation automatically, simulating the human brain for learning and analysis. Moreover, neural networks

can be used to learn a representation of input data, such as program source code.

However, using a DL model does not entirely free researchers from all preparatory work. In order to use these techniques, appropriate features first need first to be extracted from the program source code and represented in a way the DL model can understand. This process is known as 'code representation'. Code representation is the process of transforming the textual program source code into a generic input format acceptable to the DL model [15]. Researchers can make use of different representation approaches, depending on the kind of information that needs to be extracted. Examples include token-based representation for lexical information, tree-based for extract syntactical information, and graph-based for semantic information.

No single DL and code representation approach is a silver bullet that works ideally on every case. Furthermore, in practice, choosing a suitable code representation approach is not trivial as the choice is heavily impacted not only by which DL models should be employed, but also by the requirements of the software engineering task that should be addressed. Some problems might require to focus on the semantics of the code rather than the syntax. For example, a research contribution in code summarization will require different types of information to be extracted than a clone detection approach. Currently, there is no study that has investigated which representation approaches are predominantly used for which types of problems, nor is there collective evidence regarding which approaches work better for which use cases.

In this paper, we address this gap through a systematic mapping study. We systematically collected a dataset of 103 studies published between 2014 and 2021 in 20 different conferences and journals. Our primary goal was to investigate academic studies that propose or evaluate the usage of DL and code representation to address practical software engineering tasks, such as source code classification [1] or code clone detection [5]. Our main acceptance criterion was that studies needed to (a) employ DL to address a practical software engineering task (excluding studies that use DL as a tool to conduct software engineering research, such as identifying automated code contributions [16]), and (b) explicitly discuss their code representation approach. The goal of this study is to provide an exhaustive analysis and overview on the progress achieved in using DL models in different software engineering tasks. We further discuss current best practices and elaborate on gaps in the current state of research.

We show that each way of code representation can provide a different, yet orthogonal view of the same source code. Thus, different SE tasks might require different (combinations of) code representation approaches, depending on the nature and complexity of the task. Particularly, we show that it is crucial to define whether the DL approaches require lexical, syntactical, or semantic code information. Our analysis shows that a wide range of different representations are used to tackle a wide range of common SE problems. We find that all three major types of code representation (token-, tree-, and graph-based)

are employed, but tree-based (typically based on Abstract Syntax Trees, ASTs) approaches are currently the most used. Graph-based representations are not yet common, but a growing area of research. Hybrid representations, which combine different representations approaches in a single approach, are also seeing increasing use.

Nevertheless, our results also show a lack of generalizability of the presented approaches to other tasks as well as a lack of validation based on industrial datasets. Most studies construct models for a single limited-scope task based on open-source data and rarely validate the constructed model outside of the open-source domain. Evidently, industrial datasets are not inherently superior to open-source ones. However, during our review, it became clear that virtually all analysed studies are based on open-source data, published data sets (which are often also constructed based on open-source data), or in some cases, artificial data. We argue that this limits the generalizability of the investigated studies to closed-source industrial applications and denotes a gap in the current research.

The rest of this paper is structured as follows. We present necessary background about code representation in Section 2. In Section 3, we detail the applied mapping study methodology and research questions and also provide an overview of the 103 papers that form the basis of our discussion. Afterwards, in Sections 4–8, we elaborate on the findings of the mapping study per research question. This is followed by presenting the research gaps and challenges in Section 9 and potential future directions in Section 10. Finally, we conclude the paper in Section 11.

2 | PRELIMINARIES

To contextualise the rest of this study, we now present some background about code representation. In particular, we introduce three possible forms about how source code can be represented in DL. In the literature, the code representation approaches are classified into four categories: Token-based, tree-based, graph-based, and others [17]. Every form maps different syntactical and semantic aspects of the source code to a specific data structure. These representations can then be embedded in a neural network so that they can use source code as input.

Source code is originally a text encoding representing a programme. This can be processed and further transformed into different representations forms. In this section, we describe three well-known representations, each one mapping certain aspects of the original source code. We use the C snippet depicted in Listing 1 as a running example.

Listing 1 Example of C code

```
1    void foo() {
2        int x = source();
3        if(x < MAX) {
4            int y = 2*x;
```

```
5              sink(y);
6           }
7       }
```

2.1 | Token-based representation

This representation treats code as free text. Thus, it converts the code into a list of tokens where each word (e.g., "void") is a token, but each special character (e.g., '(') is also a token (rather than considering it as part of a word). An example is given in Listing 2.

Listing 2 Token representation for the code in Listing 1

```
1    ['void', 'foo', '(', ')', '{', 'int',
'x',
2      '=', 'source', '(', ')', ';', 'if',
'(', 'x', '<', 'MAX', ')', '{', 'int', 'y',
'=',
3      '2*x', ';', 'sink', '(', 'y', ')', ';',
'}', '}']
```

Then, each token will be encoded into a vector of numbers using different statistical language models, such as word embedding [18] or n-grams [19]. In principle, word embedding is a learned representation for text where words that have the same meaning get a similar representation. Technically, word embeddings are a class of techniques where individual words are represented as real-valued vectors in a predefined vector space [20]. Each word is mapped to one vector and the vector values are learned in a way that resembles a neural network and hence, the technique is often lumped into the field of DL. As for n-grams, they are useful abstractions for modelling sequential data, such as text, where there are dependencies among the terms in a sequence. However, a corpus of code can be regarded as a sequence of sequences, and corpus-based models, such as n-grams, learn conditional probability distributions from the order of terms in a corpus. Corpus-based models can be used for many different types of tasks, such as discriminating instances of data or generating new data that are characteristic of a domain. Embeddings can be considered as a way to represent words and help the DL model to learn the representation of the source code. N-grams are several words appearing together. An embedding can be trained to represent n-grams or just individual words.

2.2 | Tree-based representation

This representation treats the abstract syntactic structure of the source code. ASTs are a kind of tree representation approach that is widely used by a programming language and SE tools.

Figure 1 shows an example of an AST representation. The nodes of the AST tree are related to constructs or symbols of
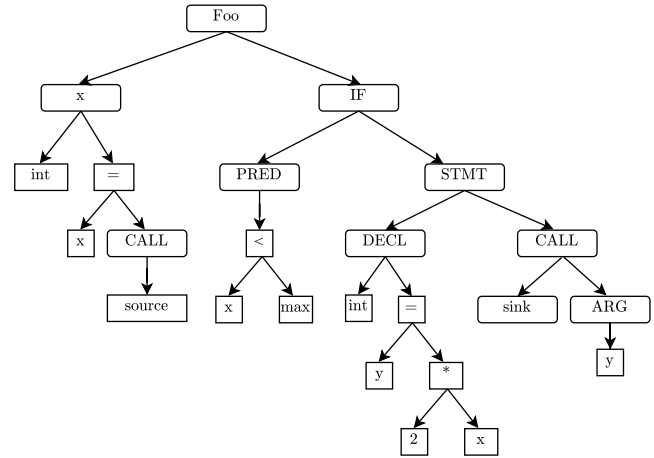

FIGURE 1 Abstract syntax tree for the code snippet in Listing 1

the source code. In comparison to the token-based approach, AST representation is abstract and does not include all available details, such as punctuation and delimiters. Theoretically, ASTs can be used to illustrate the lexical information and the syntactic structure of source code, such as the function name, and the flow of the instructions (e.g., in an if or while construct). Recently, some approaches combined neural networks and ASTs to constitute tree-based neural networks (TNNs) [21]. Given a tree, TNNs learn the vector representation by recursively computing node embeddings in a bottom-up way. Popular TNN models are the Recursive Neural Network (RvNN) [22], Tree-based Convolutional Neural Network (TBCNN) [3], and Tree-based Long Short-Term Memory (Tree-LSTM) [23].

2.3 | Graph-based representation

In this approach, source code is represented as a graph on many different levels. Levels of representation will define the type of the representation graph. Thus, a control flow graph (CFG, see Figure 2a) describes the sequence in which the instructions of a programme will be executed. Thus, the graph is determined by conditional statements, for example, if, for, and switch statements. In CFGs, nodes denote statements and conditions, and they are connected by directed edges to indicate the transfer of control.

Alternatively, the representation might be a data flow that is variable-oriented. Thus, a data flow graph (DFG) is used to follow and track the usage of the variables through the CFG. A DFG edge represents the subsequent access or modification onto the same variables. Call flow graph (CallFG) captures the relation between a statement which calls a function and the called function [24]. Finally, the entire programme can be represented as a graph using a programme dependency graph (PDG, see Figure 2b), where statements and predicate expressions can be characterised by the nodes. In this study, we differentiate between the tree- and graph-based approaches since each representation approach is used to retrieve a
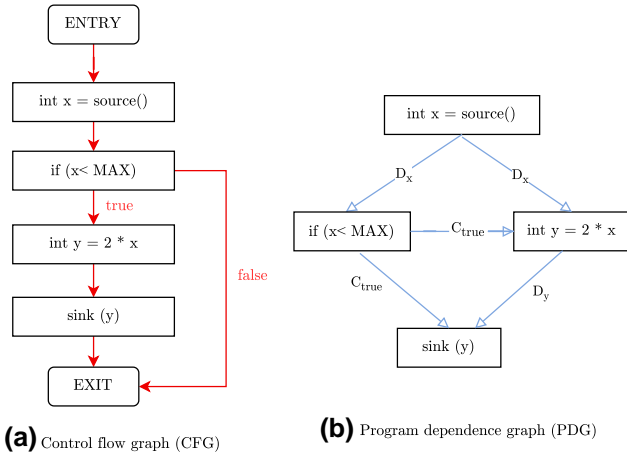
(a) Control flow graph (CFG) **(b)** Program dependence graph (PDG)

FIGURE 2   Graph-based representations for the code snippet in Listing 1

different level of information from the source code. Thus, the tree-based approach, such as using the AST, is used to extract the syntactical information from the source code [21], whereas graph-based approaches, such CFG or DFG, extract semantic information [25].

3 | RESEARCH METHODOLOGY

Our goal is to study what code representation approaches are used in combination with DL within the field of software engineering, and which code representation approaches are suitable for which tasks. Our primary method is a systematic mapping study. Systematic Mapping studies are a commonly used research method to systematically analyse a mature body of research and to derive recommendations from a disparate, large body of published works.

3.1 | Research questions

To effectively conduct a systematic mapping study, it is crucial to have well-defined research questions. The research questions analyse the main attributes of the study, which are code representation, DL, and tackled software engineering tasks, on multiple levels. In the following, we present the headlines of our research questions along with the corresponding detailed questions.

RQ 1 *Main Attributes Analysis*

In RQ1, we are primarily interested in which software engineering tasks, DL models, and code representation approaches are currently prominently investigated in the field of study.

RQ 1.1 *Software Engineering Tasks: For which software engineering tasks are DL and code representation being used?*

This research question explores the software engineering problems that are commonly tackled with DL using the code representation. This is crucial to contextualise and further analyse our subsequent findings.

RQ 1.2 *DL Models: Which DL models are being used in conjunction with code representation in software engineering research?*

While other review studies DL and SE tasks in more details, our goal is also to investigate what DL models are specifically used with a strong emphasis on code representation.

RQ 1.3 *Code Representation Approaches: Which code representation approaches are being used?*

Finally, it is evidently important to our study goal to identify the basic code representation approaches that literature currently has to offer to software engineering researchers.

RQ 2 *Detailed Analysis Based on SE Tasks*

Within RQ2, we conduct a deeper analysis of our dataset to identify which code representation approaches, on one side, and DL, on the other side, are commonly used to tackle which kinds of problems. Particularly, we are interested in identifying characteristics and commonalities of tasks that make them particularly amenable to a specific type of representation or model.

RQ 2.1 *Tasks and Models: Which DL models are being used to tackle which software engineering tasks?*

Firstly, we correlate software engineering tasks with used DL models with the goal of identifying which models are particularly suitable to solve which tasks.

RQ 2.2 *Tasks and Representations: Which software engineering tasks and representation approaches are being used?*

Secondly, we further correlate software engineering tasks with used code representation approaches.

RQ 3 *Main Attributes- Cross Analysis—Which code representation and DL models are commonly used approaches to solve a specific software engineering task?*

In RQ3, we perform the analysis on all three main attributes (task, representation, and model) together to map the code representation and DL models with different software tasks.

RQ 4 *Hybrid Approach Analysis*

In RQ4, we analyse the studies that combine different approaches in one framework. In the rest of this paper, we will

be referring to the overall solution presented in the retrieved studies as the 'framework'. These approaches are considered to provide valuable characteristics since they have either a wider scope to solve multiple tasks simultaneously, or more powerful capabilities in fulfilling many requirements by integrating multiple representation approaches. However, this integration between multiple approaches would increase the cost of implementing these (fairly complex) frameworks.

RQ 4.1 *Hybrid Software Tasks: What are the characteristics of frameworks that handle multiple software tasks? How are the different software tasks processed?*

We first scrutinise the studies that are set to solve different software tasks at once. The overarching aim is to elicit insights into the strategies followed to tackle multiple different tasks.

RQ 4.2 *Hybrid Representation Approaches: Which frameworks utilise multiple representation approaches? How are the different representations integrated?*

In this research question, we study research that exploits multiple representation approaches at the same time. We also examine how this integration is carried out to expand the efficiency of the framework.

RQ 5 *Gaps in the Literature: What are current research gaps and challenges in the software engineering field?*

Finally, our study raises the question which promising areas are currently underexplored, and warrant future research in the software engineering field.

3.2 | Literature search and selection

To conduct our study, we followed the process outlined in Figure 3. We used a two-step method for literature search. Firstly, we collected an initial set of candidate papers through a database search. Secondly, we used iterative backward and forward snowballing to extent this initial candidate set (the seed).

For constructing the initial candidate set, we have relied on a single primary search database (Google Scholar) rather than aggregating results from different digital libraries, such as the ACM Digital Library or IEEEXplorer. The reason for this was two-fold: (1) Google Scholar has a highly complete index, and it is unlikely that searching in other libraries would lead to additional search results, and (2) since we heavily made use of snowballing, completeness of the initial candidate set was deemed less crucial (as important missing work would appear during the snowballing process).

The initial candidate list was generated by executing the following search term on Google Scholar:

code representation for deep learning
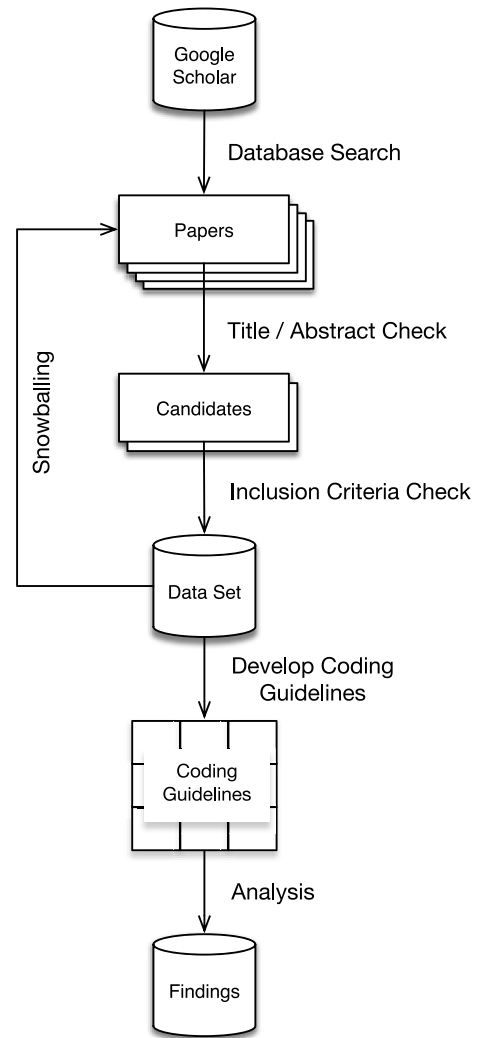
FIGURE 3 Overview of systematic mapping study process

We screened the first five pages of search results based on paper title and abstract. These potentially relevant papers were then evaluated with regard to our inclusion criteria (see Section 3.3). If a paper matched the criteria, it was added to the study dataset. After five pages (and initial snowballing), we have observed saturation, that is, investigation of the next two pages of search results did not lead to further papers matching the inclusion criteria. Hence, we stopped the search at this point.

We used explicit backward and forward snowballing to extend our initial set of candidate papers: for each selected paper, we further screened the reference list for additional relevant papers and also used Google Scholar's *"cited by"* functionality to discover later papers that have referenced papers in our initial set. We applied the same basic strategy to these additional candidate papers (screening based on title and abstract, followed by an explicit evaluation of inclusion criteria). This process has been repeated iteratively until no new papers could be found.

3.3 | Inclusion criteria

To clearly delineate papers that are within the scope of our study, we defined the following inclusion criteria:

- **I1:** Published in 2014 or later. We chose 2014 as a cutoff point because this was the year the TensorFlow system was initially released.
- **I2:** Making use of DL as a core contribution of the paper and explicitly reporting on the used code representation approach. To illustrate this criterion, we discuss the following study as a counterexample [26]. In this study, Laaber et al. tackled an SE task (predictability of system performance) and the authors used an artificial neural network (ANN) as a DL model for that task. However, the authors do not report on a specific code representation approach as they relied on the static features of the source code (e.g., lines of code or the number of loops). Hence, this study does not match the inclusion criterion I2.
- **I3:** Reporting on research in the wider field of software engineering. Particularly, we did not include pure DL research with no clear connection to software engineering.
- **I4:** Explicitly reporting (a) what software engineering task is being addressed, (b) what DL model is being used, (c) what code representation approach is being used, (d) what programming language(s) are being used, and (e) on what level (lines of code or functions/methods) DL is applied.

I1–I3 define the topical relevance to our study goals. I4 was important to ensure that all the data required for our study are actually reported by the papers in our dataset. We did not focus on publications in a specific venue and also accepted unpublished academic preprints if no published version of the paper exists. To be selected into the dataset, a paper had to fulfill all the four inclusion criteria.

3.4 | Resulting study dataset

Applying this literature search and selection procedure resulted in a dataset of 103 relevant studies, which are listed in Appendix A.

Figure 4 indicates the distribution of the papers in our dataset over time between 2014 and 2021. It can be observed that the number of relevant studies has increased over the years. With only two relevant publications in 2014 to reach 30 publications in 2019, then we observe a slight decline in 2020 (the last complete year in our study) with 19 publications. It is also interesting to notice the steadily increasing fraction of publications in academic journals rather than conferences or workshops.

In Figure 5, we have summarised the conference venues, which are common targets in this field of study. Conference venues with only one publication are not depicted in the figure. Unsurprisingly, ICLR, which is dedicated to presenting the advancement in representation learning, is the biggest contributor to our dataset with nine studies. It is followed by
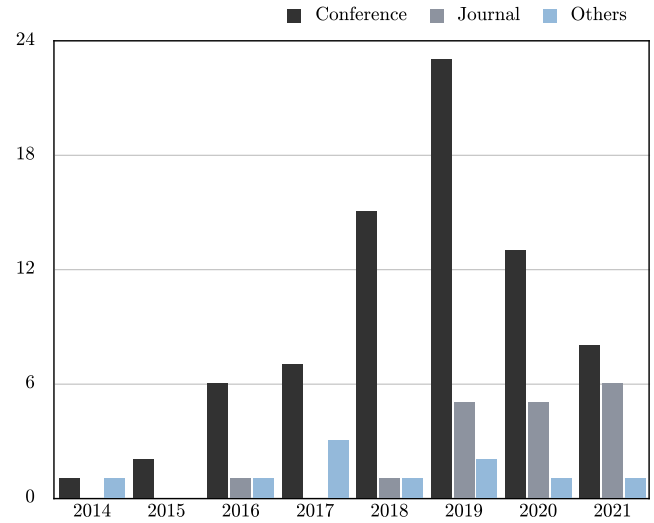
FIGURE 4 Number of publications per year. "Others" includes academic workshops and pre-prints for which no published versions exist
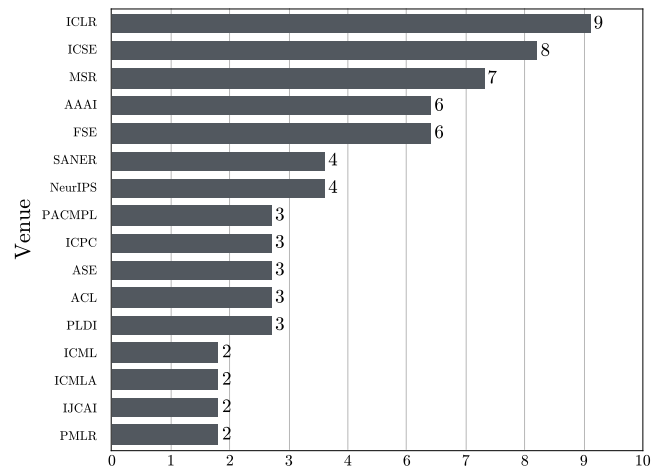
FIGURE 5 Number of publications per distinct conference venue

ICSE, which is widely seen as one of the highest ranked software engineering conferences, with eight studies, and MSR with seven studies. A smaller subset of our dataset has been published in ML venues, such as AAAI, NeurIPS, or ICML, or in programming languages venue, such as PLDI. The abbreviations of the venues presented in Figure 5 are listed in Appendix B.

3.5 | Data extraction, coding, and analysis

To analyse this dataset that answers the research questions of this study, a coding taxonomy was developed. The taxonomy is presented in Table 1. We consider the three categories (code representation approach, DL model, and Software Tasks) as primary attributes whereas we mentioned the code-level and programming languages in RQ2.3, in part related to code representation. Following our research questions, we iteratively

TABLE 1 Overview of systematic mapping study coding attributes

Code representation approach				Programming language	Code-level granularity	Deep learning models		Software task	
Tree based	Graph based	Token based	Others			Main models	Others	Task	Others
AST	CFG	Embedding	ByteCode	C	Method level	ANN	DBN	Code clone detection	Error handling
	DFG	n-grams	ASCII	C++	Statement level	RNN	NMT	Code similarity detection	Fixing format
	PDG		Code gadget	C#		LSTM	RL	Programme repair	Traceability
	CallFG		LSI	Java		CNN		Code completion	Compiler analysis
			Binary visualisation	JavaScript		GNN		Programme generation	Programme synthesise
				Python		Auto-Encoder		Vulnerability detection	Malicious behaviour detection
						Attention mechanism		Source code classification	Performance prediction
								Bug detection	Code smell detection
								Code summarisation	Type signature prediction
								Identifier generation	
								Code search	

Abbreviations: ANN, artificial neural network; AST, abstract syntax tree; CallFG, call flow graph; CFG, control flow graph; CNN, convolutional neural network; DBN, deep belief network; DFG, data flow graph; GNN, graph neural network; LSI, latent semantic indexing; LSTM, long short-term memory; NMT, neural machine translation; PDG, programme dependency graph; RL, reinforcement learning; RNN, recurrent neural network.

developed a coding guide with the following top-level codes: (1) programming language, (2) code-level granularity, (3) used code representation approach, (4) used DL model, and (5) the software task. Each publication in the dataset was coded by the first and second authors according to the taxonomy (in addition to collecting basic bibliographical information, such as the publication date and venue) with the other authors serving as sounding board and helping to resolve possible ambiguities. The resulting data were then analysed and plotted using Python scripts. We make the final coding sheet as well as the analysis script available in a replication package [27].

The detailed coding taxonomy is sketched in Table 1 and discussed in the following.

Programming Language: while DL is in principle not dependent on a specific programming language, concrete feature extraction techniques for code representation need to be built custom for individual programming languages. In our study, C, C++, C#, Java, JavaScript, and Python have emerged as target programming languages.

Code-Level Granularity: programme code can fundamentally be represented on different levels in a code representation approach. In our study, we distinguish between approaches that consider methods, functions, or similar as atomic unit [5, 28], from those that attempt to represent the programme code on a statement level [29, 30].

Code Representation: as the main target of this research, different code representation approaches were distinguished

on a fine-grained level. We distinguish between token-based, tree-based, graph-based, and other approaches. For token-based approaches, word embedding and n-grams [31] have emerged as clearly distinct groups. The only tree-based approaches [32] in our dataset are based on abstract syntax tree (AST). For graph-based approaches [33], we distinguish between CFG-, DFG-, PDG, and CallFG-based approaches, which capture the relation between a statement that calls a function and the called function [24]. Other code representation approaches that do not fall clearly into these groups are bytecode, ASCII, code gadget, latent semantic indexing, and binary visualisation since each approach has appeared only once in the retrieved list of papers. More examples for each approach will be mentioned as part of the discussion of results in Section 8.2.

DL Models: the main DL models that emerged in our coding as common methods in software engineering research are ANN [34], Convolutional Neural Network (CNN) [35], Recurrent Neural Network (RNN) [36], Graph Neural Network (GNN) [37], Long-Short Term Memory (LSTM) [38], and autoencoder and attention mechanism [39]. Additionally, three further models [deep belief network (DBN), neural machine translation (NMT), and deep reinforcement learning (RL)] emerged in two, four, and one publications, respectively, and we combine those in the group 'Others'. It is worth mentioning that we distinguished LSTM from RNN and was listed as a separate type (and not counted when referring to

RNN) since there are frameworks that combine AST with LSTM, which is referred as tree LSTM [23], and other frameworks that combine AST with RNN, which is referred as RvNN [22].

Software Engineering Tasks: to identify for which projects' code representation gets used, we also extracted the one or multiple software engineering tasks from the papers in the dataset. We observed that many common fields of study within software engineering were present. Particularly, we observed works related to code clone detection, code similarity detection [4], programme repair, programme generation [40], vulnerability detection [41], source code classification [1], bug detection [42], code summarisation [43], identifier generation [44], and code search [45]. Other tasks that emerged, but were investigated less frequently, were related to fixing formatting [46], traceability [47], compiler analysis [24], programme synthesis [10], malicious behaviour detection [48], performance prediction [49], code smell detection [50], and error handling [51].

3.6 | Data validation

To conduct a preliminary validation of the completeness of our data set, we selected five recent studies from high-profile software engineering venues that applied machine learning to one of the tasks in our study (see Table 1). We checked each reference cited by these recent studies against our inclusion criteria and validated for each study matching our criteria, whether they were indeed contained in our study set. No publications have been found to be missing.

3.7 | Threats to validity

Despite following a well-defined methodology, a review study such as ours is always subject to limitations and threats to validity. We use the classification proposed by Ampatzoglou et al. [52] to contextualise these threats.

- **Construction of the Search Process and Generalisability:** We chose to construct our dataset based on an initial search on Google Scholar followed by extensive snowballing, rather than a more conventional search strategy using major digital libraries, such as Scopus, IEEE Xplore, ScienceDirect, or the ACM Digital Library. We argue that relying on snowballing leads to a more complete and comprehensive dataset than traditional search, which suffers from limitations due to inconsistent naming and

terminology. However, one challenge is that it is hard to conduct an identical replication of our study since Google Scholar personalises search results. To mitigate this threat, we provide a replication package that includes all studied manuscripts as well as our resulting coding sheet.

- **Study Inclusion/Exclusion Bias:** DL is a rapidly growing area of research within software engineering. Hence, we needed to make decision when to stop accepting newly appearing papers into our dataset. While we do not believe that the overall findings would have been impacted if we had collected studies for a longer period of time, readers should still take our data collection period in mind when interpreting our results.
- **Validity of Primary Studies:** Four studies in our dataset are pre-prints retrieved from arXiv. While those are not peer-reviewed, the included studies are highly cited and highly influential in our field. Hence, we consider it important to include them in the analysis despite the threat that is introduced by the lack of peer review.
- **Data Extraction Bias:** While many of our coding dimensions lend themselves to objective categorisation, judgement calls still needed to be made in some cases. In these cases, we discussed among the author group to reach a consensus decision.

4 | AN OVERVIEW OF USAGE OF DEEP LEARNING IN SE TASKS

This section allows us to establish a general "process" overview of the steps required to make DL work in software engineering. While it is not expected that this general framework will differ drastically from DL in other domains, it will allow us to put the rest of the survey in context, identify the place of code representation in this general process, and serve as a guiding rail for novices to the domain. Thus, we provide a general framework of code representation and DL models' usage for tasks in software engineering based on the reviewed studies. This model has emerged from qualitatively investigating the DL models of the studies in our dataset.

4.1 | High-level process

The resulting model is depicted in Figure 6. Unsurprisingly, the high-level architecture is comparable to the usage of DL in other domains and consists of the well-known phases of data collection, data preparation and preprocessing, as well as learning and validation.
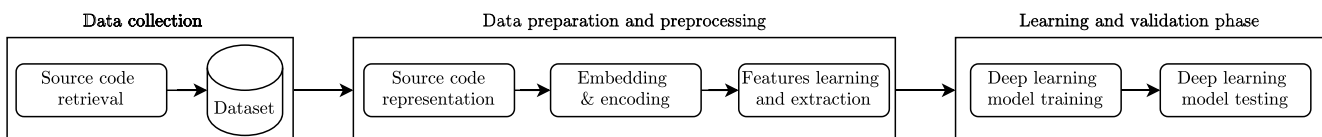

FIGURE 6 Abstracted general code representation and DL models in software engineering

Data Collection: The process starts with data collection, which in the domain of software engineering typically entails collecting the source code files for a specific programming language (e.g., through repository mining). Subsequently, the dataset needs to be annotated to serve as a training set. The annotation process is custom to the specific software engineering task that is intended to be tackled, for example, bug prediction evidently requires different annotations than, for example, code clone detection. The dataset is either ready and pre-annotated by domain experts or the researchers that conduct the study annotate the source code themselves. Annotations are task-specific and may for example, include information about the presence of bugs, or if the two code files are to be considered code clones [53].

Data Preparation and Preprocessing: Afterwards, in the data preparation and preprocessing phase, the collected code must be represented in a form that is compatible with DL. This is where code representation, the main subject of our study, comes into play. For example, in an AST representation, the collected code is converted into a tree form; then the tree paths need to be encoded or embedded as numeric values (vectorisation) using approaches such as one-hot encoding or word embedding. On the contrary, in a graph-based representation, a variety of graph embedding techniques are used, such as Graph2vec [54], HOPE [55], SDNE [56], or Node2vec [57]. Features can now be extracted from those vectors through different approaches, such as convolutional or sequential neural networks.

Learning and Validation Phase: Finally, the DL model will be trained and validated based on the tackled software engineering task.

4.2 | Examples

To concretise this process, we now present two examples of publications that follow the framework shown in Figure 6.

Example 1 *(Zhang et al. Retrieval-based neural source code summarisation, ICSE'20)*:

The first example [58] proposes a framework for (information retrieval) based neural source code summarisation. The solution specifically makes use of an attention encoder-decoder model. Figure 7 depicts the approach using the model introduced previously.

After collecting training data as a first step, source code is represented as ASTs, which are then turned into syntactic token sequences by tree traversal. Then, a trained encoder based on LSTM units is used to embed the code into a semantic vector using pooling, which is used to progressively reduce the spatial size of the representation to reduce the amount of parameters and computation in the network and preserve the most important features. Afterwards, a bidirectional LSTM decoder is used to capture the semantic context to generate natural-language summaries. The motivation behind this solution is that recent studies that use models of neural networks prefer high-frequency words in the corpus while struggling with low-frequency ones. The proposed method takes advantage of both neural and retrieval-based techniques to alleviate this problem.

Projecting Figure 7 on the main representative Figure 6, the code fragment part maps the data collection from AST to semantic vector is mapped to data preparation and preprocessing. The attention part, along with the bidirectional-LSTM decoder, presents the learning and validation phase.

Example 2 *(Wang et al. Detecting code clones with graph neural network and flow-augmented abstract syntax tree, SANER'20)*:

A second example [59] uses code representation and DL for code clone detection. In this work, and as shown in Figure 8, the authors treat the AST as a graph by following a flow-augmented abstract syntax tree (FA-AST) to build a graph representation for code fragments. This is done by adding edges representing control and data flow to the AST. Graph representation is applied here as AST-based approaches cannot fully leverage the structural information of code fragments, especially semantic information, such as the control and data flow. After representing the AST as a graph, the vectors of
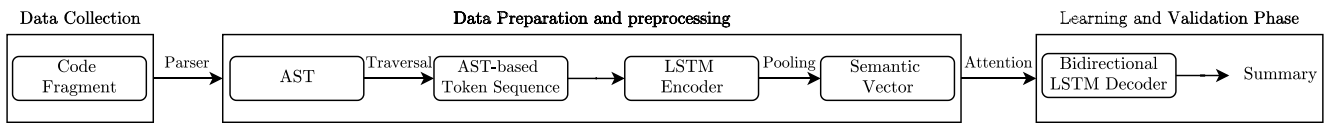

FIGURE 7   An example framework for code summarisation, based on Zhang et al. [58]. AST, abstract syntax tree; LSTM, long short-term memory
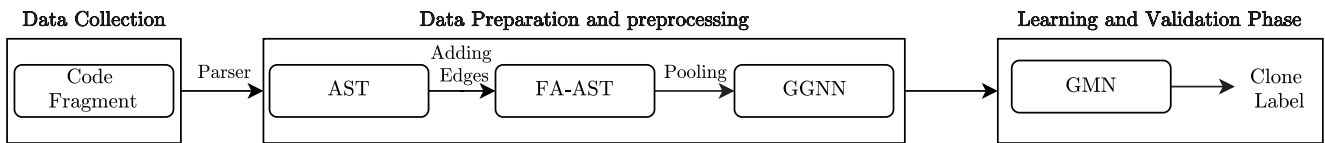

FIGURE 8   An example framework for code clone detection based on Wang et al. [59]. AST, abstract syntax tree; FA-AST, flow-augmented abstract syntax tree; GGNN, gated graph neural network; GMN, graph matching network

nodes are pooled into a graph-level vector representation. Hence, two different types of graph neural networks (GNN) are used: a gated graph neural network (GGNN) for graph embedding and a graph matching network (GMN), which can jointly learn embedding for a pair of graphs.

When mapping the approach explained in Figure 8 to the common architecture in Figure 6, the code fragment is part of data collecting, while going from AST to GGNN represents data preparation and preprocessing. Finally, the GMN is part of the learning and validation phase.

5 | MAIN ATTRIBUTES ANALYSIS

In this section, we will answer RQ1 by exploring this study's three main attributes in isolation. We answer the question about which software engineering tasks are tackled by the studies in our dataset, and what code representation and DL approaches are being used to do so.

5.1 | Software engineering tasks

DL is used for a large variety of different tasks in software engineering. Hence, to answer RQ1.1, we cluster the tasks into four broad groups inspired by work from Microsoft[1] based on high-level techniques and goals. Groups and concrete tasks, as well as their absolute and relative frequencies in our dataset, are shown in Table 2. It should be noted that the sum of percentages does not add up to 100% as some publications tackle multiple problems simultaneously.

Code-Code: the model's input is the code, and the model's output is also a source code (e.g., complete programs or code snippets). Example of tasks clustered under code-code are clone and similarity detection, code completion, programme generation and repair. Less-frequent code-code task in our dataset (grouped as "other") is fixing formatting, traceability, and compiler analysis. Code-code tasks are a natural fit for DL and hence a frequent target in our dataset, representing 46% of all studies. Code clone detection is the most frequent individual task, followed by the (very related) task of code similarity detection and programme repair.

Code-Text: the input of the learning model is code, whereas the output is (often natural-language) text. A canonical example of this type of task is code summarisation, where the goal is to produce natural language summaries of source code constructs. The only other code-text task we found is identifier generation, which includes suggestions of method or variable names based on code information. As a group, code-text approaches represent about 20% of the studies in our dataset. However, this is primarily due to code summarisation individually being a common area of interest in DL for software engineering (representing 15% of the studies). Identifier

TABLE 2 Number and percentage of publications classified per addressed software engineering task

Code-code	46 (44.7%)
Code clone detection	16 (15.5%)
Code similarity detection	9 (8.7%)
Programme repair	9 (8.7%)
Code completion	7 (6.8%)
Programme generation	6 (5.8%)
Other	3 (2.9%)
Code-prediction	**39 (37.8%)**
Bug detection	14 (13.6%)
Vulnerability detection	11 (10.7%)
Source code classification	6 (5.8%)
Performance prediction	2 (1.9%)
Type signature prediction	2 (1.9%)
Malicious behaviour detection	2 (1.9%)
Others	2 (1.9%)
Code-text	**21 (20%)**
Code summarisation	15 (14.6%)
Identifier generation	7 (6.8%)
Text-code	**6 (5.8%)**
Code search	5 (4.9%)
Programme synthesis	1 (1%)

Generation appears in 7 studies. The total count of the papers that tackle code-text is 21, as one study [60] is about both, summarisation and identifier generation.

Text-Code: this group is the opposite of the previous group, where the input is the natural language text with code output. The only two tasks in our dataset of this type are code search and programme synthesis. As for code search, it uses the query text to find the corresponding source code. This task represents about 5% of the dataset. Programme synthesis, on the other hand, takes free text descriptions of programme functions as an input and returns source code as an output. There is only one study in our dataset that tackles this task.

Code-Prediction: finally, DL can be used to predict qualities based on code, such as detecting vulnerabilities, bugs, or malicious behaviour. We also group source code classification in this category. Two studies are grouped as "other" in this group: error handling and code smell detection. As a group, code-prediction is quite prevalent, accounting for 37.8% of the studies in our dataset. Within this group, different tasks are well distributed with the most common one being bug detection (14%) followed by vulnerability detection (11%).

We present the complete mapping of papers to our taxonomy of SE tasks in Appendix C.

RQ 1.1 Summary *We categorise the studies in our dataset in four main groups, depending on the inputs and outputs of DL. Code-code and code prediction tasks are most prevalent in our data. Code-text and text-code studies are more limited; however, these are also 'smaller' groups with a lower number of concrete subtasks.*

5.2 | Deep learning models

We now present the DL models used in the retrieved studies, as per RQ1.2. Various DL models have been identified in software engineering research. A graphical overview is given in Figure 9. LSTM [38], which is a type of RNN, is the most used DL approach and found in 49 (48%) studies. LSTM copes with the problem of RNNs known as "vanishing gradients" by adding the mechanism of "cell states" to selectively remember, or forget, part of the information that is needed during training [61]. Attention mechanism [39] and CNN [62] are the second and third most used DL models with 35 and 28 publications, respectively. CNNs are particularly efficient since they can work in parallel on sequences and have a structure for which the output and input have a logarithmic distance in terms of layers, which is linear for RNNs and LSTMs. The use of CNNs together with an attention mechanism (specifically "self-attention") defines the architecture of 'Transformers'. Autoencoders [63] and RNNs are almost equally present. The least applied DL models in our dataset are ANN and GNN. The category 'Other' includes deep belief networks, neural machine translation, and reinforcement learning.

It should be noted that counts in Figure 9 add up to substantially more than the total number of studies in our dataset (103) as many papers in practice combine multiple DL models. Particularly, we observe that there are specific DL models that are commonly used together for solving specific downstream tasks, such as studies that use attention mechanisms. The attention mechanism emerged as an improvement over the encoder decoder-based neural machine translation system based on encoder-decoder RNNs/LSTMs. Both encoder and decoder are stacks of LSTM/RNN units. Further,
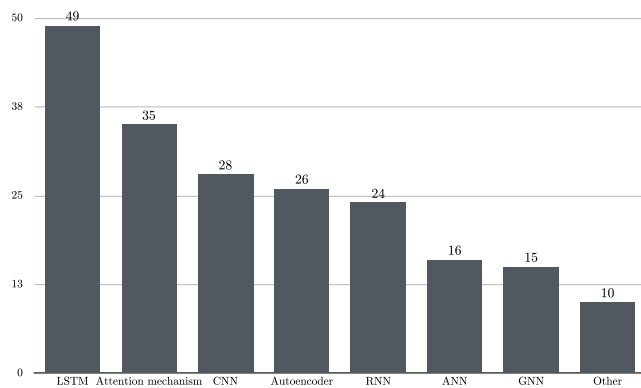
hybrid DL models are commonly used for tasks in the code-text or text-code groups as these require different models for different input and output. These issues will be discussed in more detail in Section 6.1.

RQ 1.2 Summary *Software engineering research uses a wide variety of DL models with LSTM and attention mechanisms currently receiving most attention.*

5.3 | Source code representation

We now turn towards what representations are being used in conjunction with these DL models to answer RQ1.3. We analysed the source code representation approaches that are utilised to encode source code into a form that is meaningful and can be fed into ML models. Three primary (groups of) techniques have emerged from our analysis: token-based representation, tree-based representation, and graph-based representation. Five concrete representation approaches emerged that do not clearly belong into any of these groups and have hence been categorised as 'Other'. These are code gadget (the number of lines of code that are semantically related to each other [64]), binary visualisation (the raw representation of any type of file stored in the file system, which exhibits similar behaviours of the code while being syntactically different [65]), ASCII which used by Wang et al. [66] to convert each letter of JavaScript code into eight bit binary, latent semantic indexing (LSI, a method of analysing a set of documents in order to discover statistical co-occurrences of words that appear together which then give insights into the topics of those words and documents [47]), and bytecode (in this representation, a code fragment is expressed as a stream of bytecode mnemonic opcodes forming the compiled code [67]). An overview over the prevalence of the four groups is given in Figure 10.
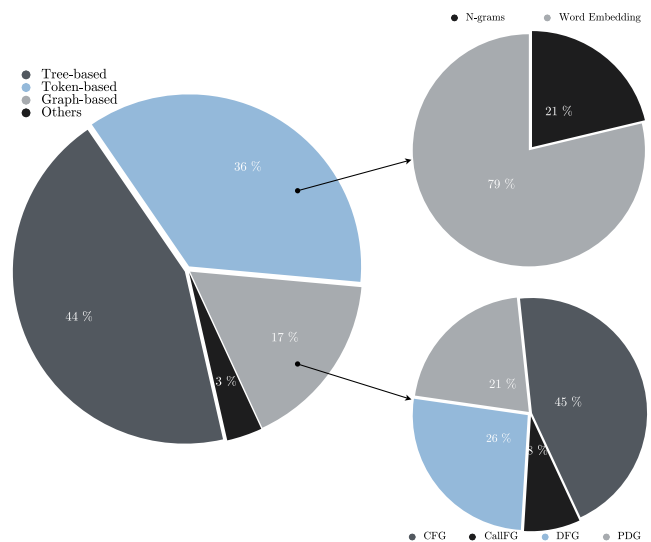

FIGURE 9 Summary of DL models used in conjunction with code representation in software engineering research

FIGURE 10 Summary of code representation approaches

All three groups see frequent use in software engineering. Tree-based and token-based representations are most common and are both utilised in over half of the studies in our dataset (66% or 64% and 54% or 52%, respectively). As before, some studies employ multiple representation approaches simultaneously. Graph-based approaches are less common and only used in 25 (24%) of studies, but the usage is increasing. The remaining techniques are only used in five individual publications.

For tree-based representation, the only specific technique that emerged from our study is AST. However, both token-based and graph-based representations can be split up into further subcategories. For token-based approaches, these are word embedding and n-grams, with word embedding being the dominant technique (used in 37% or 79% of the studies using a token-based approach, see also Figure 10).

There are a larger number of choices of graph-based representations, which are depicted in Figure 10. The most common ones are CFG (17% or 45%). Other options include PDG, DFG, and CallFG.

5.3.1 | Alternative representation approaches

In contrast, some studies have made use of code representation approaches without direct adoption of any of the methods that are categorised in Table 1. To take token-based approaches as an example, some works have tokenised the text without using word embedding or n-grams techniques. In a study by Fernandes et al. [68], the proposed framework breaks up all identifier tokens (i.e., variables, methods, classes, etc.) of the source code into sub-tokens by splitting them according to specific heuristics (*camelCase* and *pascal_case*). Gupta et al. [69] use an encoding map for each programme to map every token, based on its type (such as function, literal, variable, etc.), to a unique name in a pool of names. Similarly, there is a subset of graph-based solutions that have not used any graph-based methods that are classified in Figure 10. Yasunaga and Liang [70] have proposed a programme-feedback graph to model the reasoning process and capture the semantic correspondence involved in programme repair. Similarly, Fernandes et al. [71] extend sequence encoders with a graph neural network that can reason about long-distance relationships. Finally, Brockschmidt et al. [72] decode the code in a graph representation using GNN for partial programs to incorporate rich semantic information that is useful in programme repair tasks.

5.3.2 | Code representation depending on code-level granularity

Another question our review can answer is whether different code representation approaches are more commonly used to handle code on the statement or method levels. The results of this analysis are shown in Table 3.

T A B L E 3 Main code representation approaches and code-level granularity

	Token (%)	Tree (%)	Graph (%)
Statement level	73	71	64
Method level	27	29	36
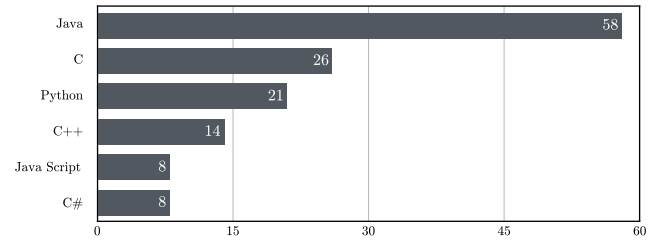
F I G U R E 11 Programming languages considered in the dataset

As we can see, there is no clear-cut difference in the usage of representation approaches depending on the code level. However, token-based approaches are slightly more commonly used in studies that work with code at a statement level. This intuitively makes sense as such studies are less concerned with preserving the syntactical or semantic context of a software project.

5.3.3 | Code representation for different programming languages

As a final exploration of code representation approaches, we map which programming languages the studies in our corpus use. This is shown in Figure 11.

Unsurprisingly, Java is by far the most commonly considered programming language and is considered in over half the studies in the corpus (58 studies, or 56%). This can be explained by the wide availability of parsing tools that parse Java code into AST, which is compatible with the findings in Figure 10 that show AST to be the most common representation approach. Examples of common Java parsers are the Eclipse Java development tools (JDT) used by Büch and Andrzejak [73], SrcML [74] used by Bui et al. [4], or JavaParser used by Alon et al. [75]. However, SrcNL is a universal AST system that uses the same AST representations for multiple languages (Java, C#, C++, and C)

For graph-based representation approaches, different tooling is required. For example, Ben-Nun et al. [48] convert Java code to statements in an Intermediate Representation (IR) using the LLVM Compiler Infrastructure [76], which is then processed to contextual flow graphs. Mehrotra et al. [6] use the Soot optimization framework [77] to build program dependence graphs for Java code, followed by the Cytron's method [78] to compute control dependence. Reaching definition [79] and upward exposed analysis [80] are both used for computing data dependence graphs.

RQ 1.3 Summary *All three main groups of code representa-tion introduced in Section 2 are used in literature with tree-based and token-based code representations being most prevalent. It is also notable that a substantial number of publications use a hybrid representation approach, combining multiple different representations.*

6 | DETAILED ANALYSIS BASED ON SOFTWARE ENGINEERING TASKS

So far, our analysis discussed the three main dimensions of the study (tasks, DL models, and code representation approaches) in isolation. Now, we turn to investigating the interplay between these dimensions as part of RQ2. Particularly, we investigate how DL models and chosen representation depend on tasks (Sections 6.1 and 6.2, respectively).

6.1 | Software tasks and DL models

In this section, we will discuss the results that explain RQ2.1, where we map the chosen DL models to tackle software engineering tasks. Figure 12 depicts a mapping of specific DL models identified in the study to the four high-level categories of tasks as a bubble plot.

We observe that a wide variety of models have been applied to the tasks in the code-code group, whereas there appears to be more dominant methods for code-text (LSTM with autoencoders and attention mechanisms) as well as code-prediction (CNN and LSTM). The data for the text-code group are too sparse to come to a clear conclusion, but initial evidence suggests that researchers also use a variety of models for this task. Further, LSTM is commonly used and proportionally distributed for all types of tasks. However, CNN is most frequently used for tasks in the group code-prediction. Both, autoencoders and attention mechanisms are used frequently for code-code and code-text tasks, but rarely for other tasks.

Figure 13 drills deeper into this and depicts the usage of different DL models for specific tasks in the code-code group. We observe that a variety of models are used for all specific tasks.

In programme repair, some approaches use sequence to sequence networks with encoder-decoder models attached with attention mechanisms. Bi-directional LSTM is mainly used in both encoder and decoder. However, attention might be used in the decoder part [40] or in encoder [70]. However, other approaches for handling programme repair do not rely on the encoder-decoder model. For example, Vasic et al. [81] use LSTM and attention mechanism to locate and handle the misuse of the variable defined in the programme. Other studies rely on sequential models for handling programme repair without using the encoder-decoder attention model [69, 82], whereas Dinella et al. [83] rely on graph neural networks for learning graph transformation to repair the bugs in the Java-Script programs.

Figure 14 presents a similar analysis for specific code-text tasks. It becomes evident that autoencoders are an important facet of contemporary code summarisation research. These approaches are based on the sequence-to-sequence paradigm over the words of some text with a sequence encoder (typically a RNN, but sometimes using self-attention [12]) processing the input and a sequence decoder generating the output. Recent successful implementations of this paradigm have substantially improved performance by focussing on the decoder, extending it with an attention mechanism over the input sequence and copying facilities [68]. However, while standard encoders (e.g., LSTMs) can in theory handle arbitrary long-distance relation-ships, in practice, they often fail to handle long texts (sum-marisation output) correctly [84].

RQ 2.1 Summary *Most of the software tasks studied are mainly tackled using the LSTM model. However, autoencoders and attention mechanisms are also widely adopted, particularly in code-code and code-text tasks. A high number of code-prediction publications utilise CNNs.*
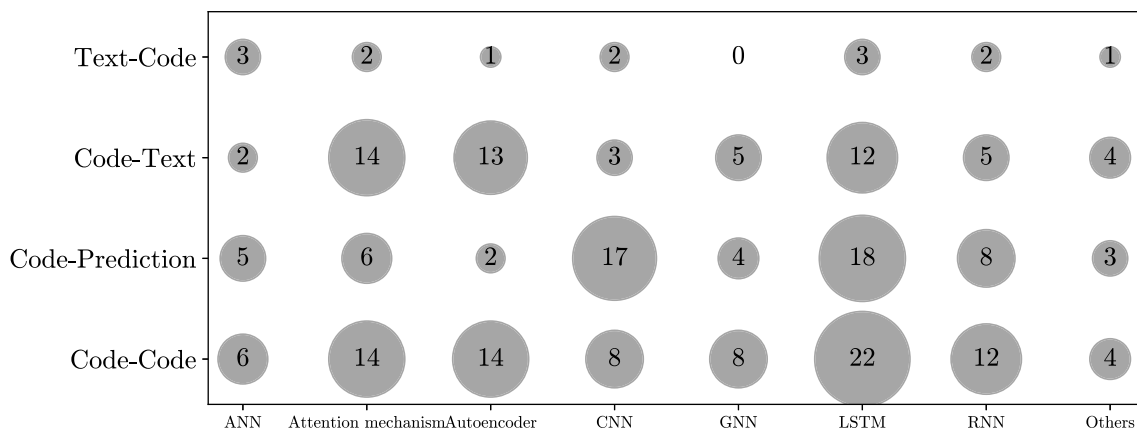
FIGURE 12 Software engineering tasks and applied DL approaches. ANN, artificial neural network; CNN, convolutional neural network; DL, deep learning; GNN, graph neural network; LSTM, long short-term memory; RNN, recurrent neural network

6.2 | Software tasks and code representation

We now turn towards RQ2.2 and explore how the choice of code representation approach is impacted by the chosen software engineering task. An overview for the four groups of tasks is provided in Figure 15.

We observe that the various code representation approaches are used across software engineering tasks. Text-code tasks are commonly addressed using token-based approaches. Only one study uses a tree-based approach for this type of task [10], and none uses a graph-based approach. However, this study handles multiple tasks within the same study. More

specifically, the authors have built multiple representations to handle tasks separately. The tree-based approach addresses code summarisation (a code-text task), whereas a token-based approach is used for code retrieval (text-code). Hence, we conclude that for text-code tasks, for example, code search, a token-based representation is the only method that is seeing current use. This can be explained as the freeform text of, for example, a query is better treated using natural language processing (NLP) techniques than the more code-specific tree- and graph-based representations.

Graph-based approaches are most commonly used in code-code tasks. However, also 38% of graph-based

FIGURE 13 Applied DL approaches for specific code-code tasks. ANN, artificial neural network; CNN, convolutional neural network; DL, deep learning; GNN, graph neural network; LSTM, long short-term memory; RNN, recurrent neural network
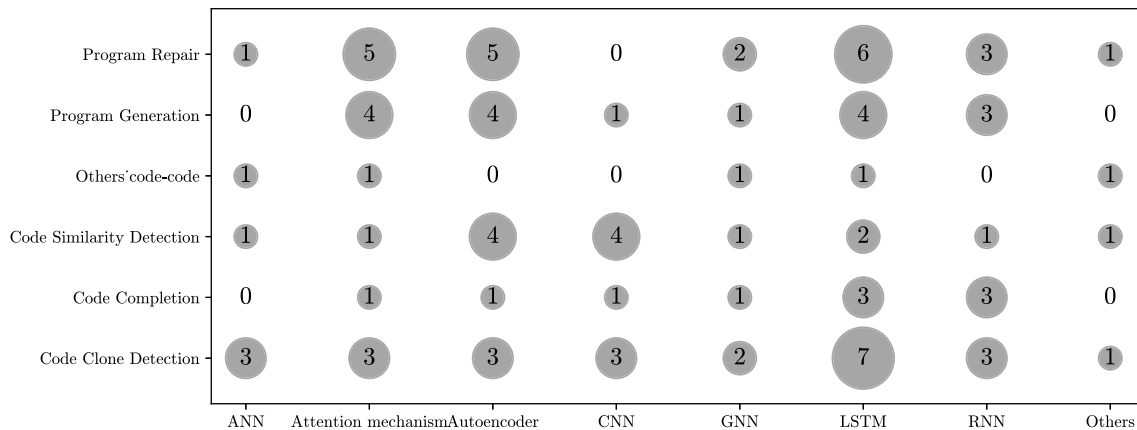

FIGURE 14 Applied DL approaches for specific code-text tasks. ANN, artificial neural network; CNN, convolutional neural network; DL, deep learning; GNN, graph neural network; LSTM, long short-term memory; RNN, recurrent neural network
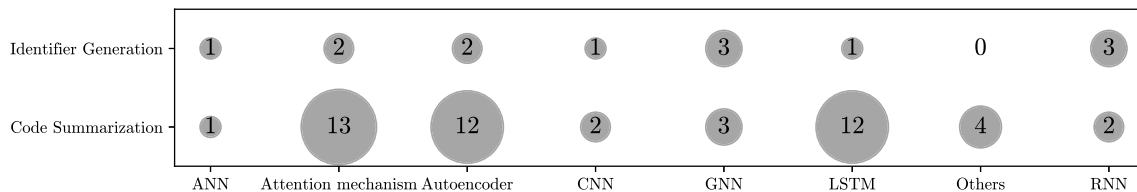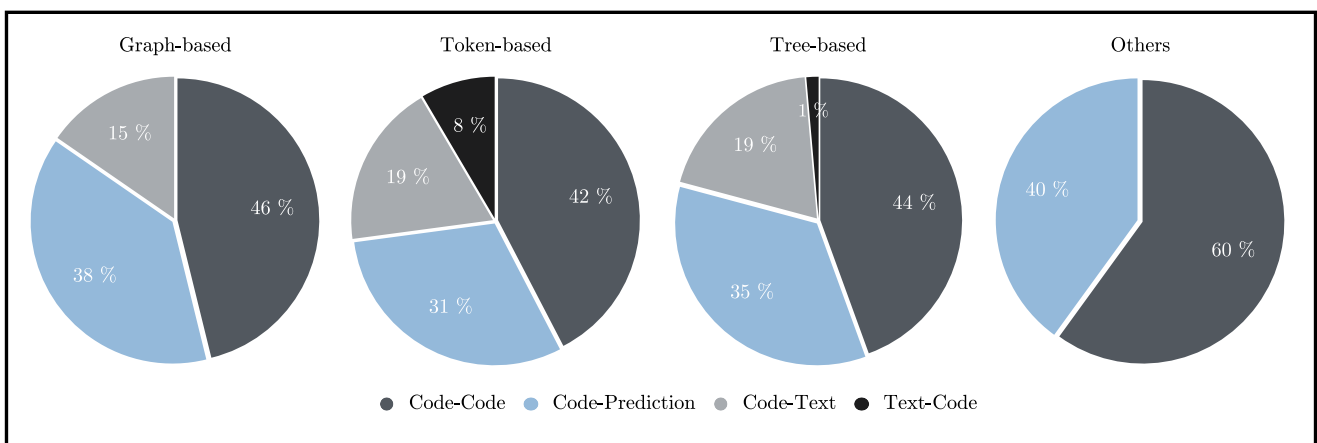

FIGURE 15 Code representation approaches per group of software engineering tasks

approaches are used for code-prediction tasks. To better understand this observation, we have again detailed further into specific tasks. In Figure 16, we present how often specific tasks in the code-code groups use a graph-based approach to represent the source code.
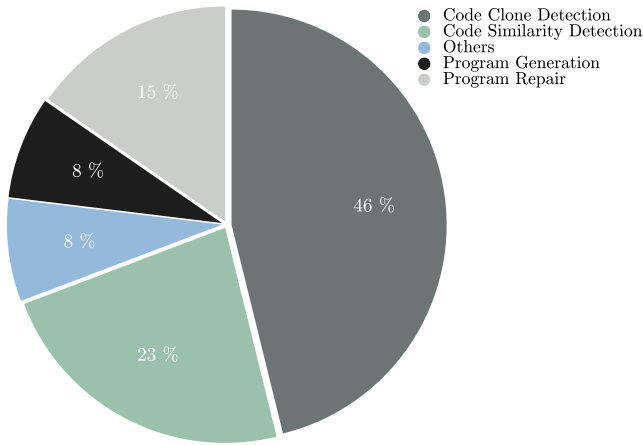

FIGURE 16 Usage of graph-based representation for specific code-code tasks
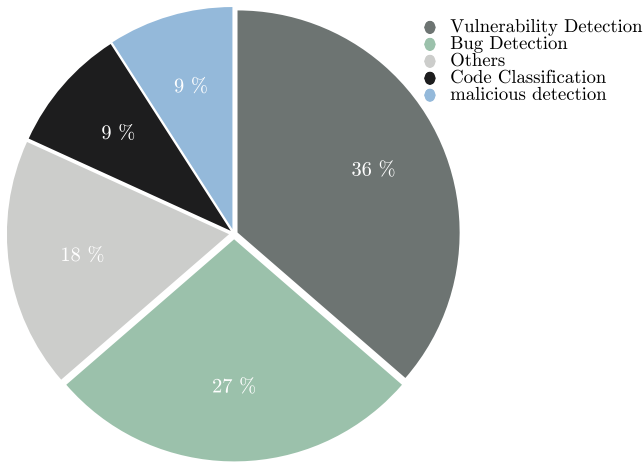
FIGURE 17 Usage of graph-based representation for specific code-prediction tasks

Both code clone and code similarity detection are proportionally overrepresented here. This is interesting, especially since these tasks have many similarities. It can be argued that a graph representation is highly appropriate for solving the problem of identifying similar code elements. By representing code snippets as a graph, those graphs are embedded into vectors (one vector for each graph). To measure the similarity, one can then simply compute the distance between those graphs. This approach is arguably more simple and effective than breaking each piece of code into tokens and then embedding each token into a vector.

We now conduct a similar analysis for the usage of graph-based representations in code-prediction tasks (Figure 17). We observe that graph-based representation approaches are commonly utilised in vulnerability and bug detection, together amounting to about two thirds of all usage of graph-based representation in code-prediction tasks. For these approaches, researchers commonly need to preserve semantic information for which graph representations are most suitable.

RQ 2.2 Summary *We were not able to identify a clear pattern that specific representations are more common for specific types of tasks with one exception: text-code tasks frequently call for token-based methods. Aside from this, software engineering researchers have tested different combinations of representations and tasks, and no clear consensus what the ideal way to address any specific task (except text-code tasks) has emerged yet.*

7 | MAIN ATTRIBUTES – CROSS ANALYSIS

We now discuss the interplay of all three dimensions of this study—tasks, DL approach, and code representation approach, answering RQ3. In the previous sections, we have separately analysed the three dimension task, DL model, and code representation approach. To answer RQ3 and get deeper insights into the current trends in the field, we now investigate all three dimensions together. Results of this analysis are summarised in Table 4.

LSTM is the most commonly used model for code-code tasks, using both tree- and token-based representations as

TABLE 4 Analysis of the main attributes

■ Code-Code   ■ Code-Text   ■ Text-Code   ■ Code Prediction

	Tree Based				Graph Based				Token Based				Others			
ANN	4	1	0	3	3	0	0	2	4	1	3	3	1	0	0	0
CNN	5	1	1	14	1	0	0	5	6	3	2	8	1	0	0	0
RNN	8	4	0	5	3	1	0	4	9	2	2	6	1	0	0	0
LSTM	15	8	1	10	5	2	0	5	12	7	3	12	0	0	0	1
Attention Mechanism	9	8	0	4	6	3	0	1	4	8	2	3	0	0	0	0
Auto-encoder	10	9	1	1	4	3	0	0	8	7	1	0	1	0	0	1
GNN	5	4	0	3	7	4	0	3	1	1	0	1	0	0	0	0
Others	4	4	1	2	1	0	0	1	1	1	1	2	0	0	0	0

well as, to a certain extent, autoencoders. In contrast, LSTM and autoenconders are almost equally frequently used for code-text tasks. LSTM and autoencoders go hand to hand in solving sequential problems by treating the code as a sequence of tokens (using a token-based representation) or sequence of nodes (in the tree-based representation). Hence, sequential models, such as LSTM, are the most appropriate approach for such a problem. The sequential model needs to be encapsulated into an encoder-decoder model because for a code-text task, it is necessary to encode the code consistently through one model in order to generate natural language sequences from the corresponding source code. Attention mechanisms are used to dynamically select the distribution over the combined representations while decoding or encoding is selecting the relevant path in the AST [85].

Unsurprisingly, GNN is the most commonly used architecture in conjunction with a graph representation in the majority of SE tasks. In contrast, no common DL models can be identified for text-code tasks across all the representations. Instead, various different models are used across the studies in our dataset. This is because the text that represents the input in a text-code task can be treated using natural language processing (NLP) techniques, which according to literature, all DL models work properly on.

As for code-prediction tasks, CNN is the most dominant model in conjunction with a tree-based representation, while LSTM is most commonly used for a token-based representation. This difference is rooted in the different goals underlying tasks in the code-prediction group—in these tasks, the goal is not to generate code as in code-code and text-code tasks, or generating text as in code-text tasks. Much more, code-prediction tasks tend to deal with classical DL prediction problems, that is, classification and regression. For instance, bug or vulnerability detection is a binary classification problem to decide whether or not the code includes a bug or vulnerability. The same is true in performance prediction, where a specific performance value is predicted as a regression problem.

RQ 3 Summary *We analysed the retrieved frameworks from the viewpoint of the main three dimensions of our study, software task, code representation approach, and deep learning model applied. LSTM and autoencoders are the most used deep learning for code-code and code-text tasks using tree-based and token-based representations. While GNN is the most used model with graph representation with most of the SE tasks. For code-prediction, CNN with tree-based representation and LSTM with token-based representation are the most common techniques used in the studies.*

8 | ANALYSIS OF HYBRID APPROACHES

In this section, we will answer RQ4 by exploring frameworks that address either multiple SE tasks or which use multiple representations. We refer to such studies as using a hybrid approach.

8.1 | Hybrid software tasks within one framework

In this section, we address RQ4.1 and identify characteristics and main properties of frameworks that solve multiple SE tasks simultaneously. No study in our dataset is general in the sense that it is able to addresses all SE tasks.

8.1.1 | Solving many tasks with one framework

Bui et al. [10] propose an approach that integrates three different tasks—it tackles code similarity detection as a code-code task, code search as text-code tasks, and code summarisation as a code-text task. This study is singular in that it combines text-code tasks with other SE tasks. The proposed method is a self-supervised learning framework for source code modelling designed to mitigate the need for labelled data for different SE tasks. The key innovation here is that the source code model is trained to detect the similarity and dissimilarity across code snippets. This study also makes use of a hybrid representation approach, by merging an AST-based strategy with a token-based approach. The representation approaches are used in the encoder component of the discussed system. Hence, well-know AST-based code modelling techniques, such as Code2vec [44], TBCNN [3] are used besides token-based approaches by handling the source code as a sequence of tokens using a neural machine translation (NMT) baseline. Those techniques utilise node type and token information to initialise AST nodes. The hybrid representation approach will be discussed in more detail in Section 8.2. Throughout this approach, various encoders are used, and the choice of encoder depends on the task.

8.1.2 | Frameworks that solve two tasks

Besides the aforementioned study, we find that three other approaches tackle combinations of code-code and code-text tasks. Cvitkovic et al. [60] design a framework that solves code completion as a code-code task and identifier generation as a code-text task. They use ASTs to represent the source code. This tree is augmented with semantic information, such as data– and control–flow to eventually obtain an augmented AST as a directed multigraph. The augmented AST is then further augmented by adding a Graph Structured Cache. They add a node to the augmented AST for each token in the input instance. Then, all the nodes are vectorised to be processed than with the graph neural network.

Kang et al. [86] evaluate the generalisability of the Code2vec modelling technique by applying it along with a sequential model to address code clone detection as a code-code task as well as code summarisation as a code-text task. Then, they compare the results obtained from these techniques with a task-specific baseline. In this study, the authors do not focus on the overall effectiveness of the methods. Instead, they evaluate if the use of Code2vec can improve the performance

of the baselines. Based on their results, the authors claim that no improvements had been achieved by applying Code2vec.

Code summarisation is also investigated through one framework proposed by Wei et al. [87] that is generalised to solve programme synthesis as a text-code task. They use a token-based approach for code representation. The proposed framework consists of three main parts: a code summarisation model, a programme synthesis model, and dual constraints. The code summarisation and programme synthesis models both rely on a sequence-to-sequence neural network and an encoder-decoder attached with attention mechanism between encoding and decoder. To leverage the contextual information within the word embedding, a token-based, bi-directional LSTM is used as a unit in the encoder. Another LSTM is also used in the decoder. Dual constraints are used by adding regularisation terms in the loss function to constrain the duality between two models, which are enlightened by the probabilistic correlation and the symmetry of attention weights between code summarisation and programme synthesis models.

Finally, four studies design solutions that are transferable across code-code and code prediction tasks [21, 81–83]. Three of those proposed frameworks that tackle programme repair as code-code tasks and bug detection as code prediction tasks. These tasks are related in the sense that a bug is first detected in the code, which is subsequently fixed through programme repair. Hence, it makes sense to have one solution that addresses these tasks simultaneously. In the same context, one of those studies [83] uses a hybrid code representation approach by combining tree- and graph-based approaches. Thus, code is parsed into an AST to capture the programme's syntactic structure; then, the leaf nodes are connected with *SuccToken* edges. Additionally, the value of nodes that store the content of the leaf nodes is added with special semantic *ValueLink* edges connecting them together. Based on the study, the ultimate aim of introducing this additional set of nodes is to provide a name-independent strategy for code representation and modification. After representing the programme as a graph, a GNN is used to map the graph into a fixed dimension vector space. An LSTM is then trained to locate the bug through a sequence of graph transformations. That means that, given a buggy programme modelled by a graph structure, the proposed framework makes a sequence of predictions, including the position of bug nodes and corresponding graph edits to produce a fix.

The other related approaches [81, 82] use only a token-based approach combined with LSTM to locate and repair the bug in the programme. Moreover, the fourth study in this group [21] defines an AST-based neural network for source code representation in order to solve code-clone detection as a code-code task and code classification as a code-prediction task. This study discusses the problem of the long depth of the AST, which causes a long dependency between the sequence of nodes, leading to vanishing problems when injected into the sequential model. Thus, the tree is divided into a sequence of small statement trees. Those trees are encoded to be used with a bidirectional RNN model to leverage the naturalness of statements to achieve the tasks.

Statement trees are constructed using the preorder traversal algorithm.

It is interesting to observe that no study in our dataset proposes a framework that addresses a combination of code-text and code-prediction tasks, nor combinations of code-prediction and text-code tasks.

RQ 4.1 Summary *The integration between multiple tasks within one framework relies on the relatedness between these tasks. However, there currently appears to be no truly general framework for DL in software engineering, which could be applied independently of the tackled software tasks.*

8.2 | Hybrid representation approaches

Some studies have utilised a hybrid approach for code representation to capture more information on the source code. This is often promising as tree-based approaches capture syntactical information, graph-based approaches are better at retaining semantics, and token-based approaches preserve lexical information.

Table 5 summarises how often different types of code representation approaches are used alone or in conjunction. The diagonal elements represent the frequency of the frameworks that have used a single representation approach, while the non-diagonal elements represent the frequency of the frameworks that have used hybrid representations. Seven studies [5, 7, 21, 42, 67, 88, 89] combined representations from all three groups. The most common hybrid approach is a combination of token- and tree-based approaches, used by 25 studies, or almost a fourth of our dataset, in total (note that 18 approaches combine only tree- and token-based representation, plus the seven studies that use all three). Combinations of tree- and graph-based approaches are also fairly popular, used by 16 studies in total.

Particularly interesting are the seven studies that have used all three representation approaches in conjunction. For example, Hua and Li et al. [7, 42] present work on bug detection. The two approaches start with constructing AST representations of the source code in order to locate sensitive point-like object construction, method invocation, expression statement, conditional statement, and loop statements. Sensitive points are the syntax characteristics where most 'simple' bugs manifest. Then, Word2Vec [20], a token-based representation approach, is employed by taking all of the AST nodes of a method as the input and generating a learned vector representation for each given AST node. This vector

T A B L E 5 Frequency of combinations of (types of) representation approaches

All = 7	Token	Tree	Graph
Token	25	18	4
Tree		32	9
Graph			5

representation is later used as input to the DL model. However, the local context of the method representation from AST node representations is preserved by representing each path as an ordered set of node vectors. Since the bug can be involved in multiple methods, it is crucial to capture also the global context by modelling the relations between different methods through a dependency graph (a PDG). Thus, semantic information in the source code, such as data and control flow, is traced. Then, when the graphs are generated, different embedding techniques for graphs are used on nodes, edges or the entire graph. For example, Node2Vec [90] is used to vectorise the nodes.

Similarly, other studies that use all three representation approaches are tackling code clone detection [5, 21, 67]. These studies show that using a stream of identifiers to represent the code, DL can effectively replace manual and hand-crafted feature engineering. Moreover, these works show that representation of the code at different levels of abstraction (identifiers, AST, and CFG) can provide a different, yet orthogonal, view of the same code fragment, thus enabling more reliable detection of code similarities.

Sonnekalb and Li et al. [42, 89] investigate a combination of all three main representation approaches for the task of vulnerability detection. These studies claim that there is a need to represent programs in a way that can adequately accommodate the syntax and semantic information related to vulnerabilities. This enables multiple kinds of neural networks to detect various kinds of vulnerabilities.

RQ 4.2 Summary *62 (60%) frameworks of the retrieved studies have used only one type of representation approach, while 31 (30%) studies have combined representations from two groups. Seven (7%) studies utilised representations of all three main groups in conjunction.*

9 | GAPS IN THE LITERATURE

In this section, we will discuss perceived limitations, research gaps, and challenges that we derived from the retrieved studies, addressing RQ5.

- **Lack of Topic Coverage**: Even though we have found DL to be applied to a wide variety of SE tasks, some crucial tasks appear to be underrepresented. For example, we have identified only one or two studies each tackling performance prediction, code smell detection, or traceability. This is surprising, as these tasks could profit substantially from an investment in DL. Taking performance prediction as an example, performance is often seen as a crucial non-functional property of software systems, and traditional performance engineering is challenging [91] and error-prone [92]. A deeper investment in DL in the style of some code clone detection or programme repair studies seems promising in these domains.
- **Lack of Generalisability**: According to Figure 6, DL models can be used in two phases—in the data preparation and

preprocessing phase for learning the representation of code (representation learning), and then again in the learning and validation phase to achieve the SE task. In principle, representation learning is independent of the tackled SE task. Transfer learning [93] could be used to generalise and reuse pre-trained models for representation learning to different tasks. In other application domains of DL (such as computer vision or NLP), transfer learning has led to generally useful models such as DenseNET [94] or BERT [95]. We observe a lack of such models in software engineering. However, we made the observation in this study that most of the proposed approaches are highly domain and problem-dependent. Thus, very few retrieved studies are applied to different SE tasks. Very few solutions are transferable or easily adapted to other SE problems. There are some approaches that explicitly present generalised SE representations [44, 85]. However, these approaches are for fixed code units, such as tokens, statements, or functions. They are not sufficiently flexible to generate encoding and embeddings for different units. Thus, the learned code representation may not be effective for a multitude of tasks. Two studies in our dataset already attempt to provide such a generalised representation model [4, 10]. We argue that this is an important area of future research that should be a focal point for future investigations.

- **Lack of Industrial Data**: Unsurprisingly, the vast majority of approaches in our dataset are trained and tested on open-source projects extracted from platforms, such as GitHub. However, validation of the resulting models on industrial data is rare. This is understandable especially in supervised learning model, which requires annotated datasets of considerable size. Annotations often need to be manually labelled by humans according to a specific downstream task. To address this challenge, and connecting to the previous point, recent research uses self-supervised learning [4, 10] to leverage unlabelled data to pre-train code representations which are reusable for building general models that are suitable for various downstream tasks. While the type of data that led to this challenge was not an explicit dimension that we coded for this study, it became abundantly clear during the review that virtually all analysed studies are based on open source data, published data sets (which are often also constructed based on open-source data), or in some cases artificial data.

RQ 5 Summary *We conclude that the core research gaps currently prevalent in the literature relate to a lack of coverage for some relevant SE tasks, a lack of the application of transfer learning, and a lack of validation based on industrial data.*

10 | DISCUSSION

- **Towards AST-Based Neural Networks:** As our work shows, token-based approaches are common in software engineering literature. These approaches tend to either

treat the code as a token sequence or bags of tokens, or they rely on latent semantic indexing (LSI) and latent dirichlet allocation (LDA) to represent the code. The problem of those token-based approaches is that they treat the source code as a natural language. To improve these approaches, code syntax and semantics need to be taken into account [96]. Some existing work [3, 22, 23] provide strong evidence that syntactical knowledge contributes positively and leads to better representations than traditional token-based methods. We speculate that this is the reason why ASTs are used in so many different approaches. Through the AST, researchers can easily capture lexical as well as syntactical information. Hence, many research works try to combine ASTs with deep learning, which is referred to as AST-based neural networks. Theses approaches combine ASTs with Recursive Neural networks (RvNN) [22], tree-based CNNs [3], or tree LSTMs [23].

- **The Limitations of Tree-based Approaches:** Despite the effectiveness of such tree-based neural network approaches in extracting both lexical and syntactical information, there are limitations. Similar to long texts in NLP, tree-based neural models are vulnerable to the gradient vanishing problem, where the gradient becomes vanishingly small during the training (especially when the tree is very large and deep, which it often is for real-life source code). Hence, traversing and encoding the entire AST tree in a bottom up way [22, 23] or using a sliding window technique [3] may lose long-term context information [21]. Another limitation of AST-based neural networks is that those approaches transform the AST or present it as full binary trees to improve simplicity and efficiency. However, this in turn destroys the original syntactic structure of the source code and makes the AST even deeper. Moreover, the transformed and deeper AST reduces the capability of neural network models to capture more real and complex semantics [21]. Finally, some SE tasks require not only syntactical, but also semantic information.

- **Towards Graph-based Code Representation:** Due to the problems of leveraging semantic information with AST-based approaches, more and more newer DL papers adopt graph-based representations, such as long-term CFG and Data Dependencies Graph (DDG). These representation approaches can overcome some of the limitations of AST-based neural networks. Examples of such works are Zhao et al. [25], who extract semantic features from the CFG of represented code, Allamanis et al. [33], who consider the long-range dependencies induced by the same variable or function in distant locations, or Tufano et al. [67], who directly construct CFGs of code fragments.

- **The Limitations of Graph-Based Approaches:** However, graph-based representation is not with challenges either. The drawback of CFGs is that they lack data flow information. Furthermore, most CFGs only contain control flows between code blocks and exclude the low-level syntactic structure within code blocks [59]. Another drawback of CFGs is that in some programming languages, CFGs are much harder to obtain than ASTs. Nevertheless, Henkel

et al. [97] show that embeddings learned from (mainly) semantic abstractions provide nearly triple the accuracy of those learned from (mainly) syntactic abstractions. Ultimately, many solution approaches choose to use a syntactic representation [75], because it was shown to be useful as a representation for modelling programming languages in machine learning models. It was also shown that they are more expressive than n-grams and manually designed features [44]. Other solutions use approaches based on semantic context [98] in which programme elements are graph nodes and semantic relations are edges in the graph. Due to the gap between syntax (e.g., tokens or ASTs) and the semantics of a procedure in a programme, the abstractions of traces obtained from symbolic execution of a programme are also used as a representation for learning word embeddings [97].

Based on the aforementioned discussion and the ongoing developments and current promising research directions, we expect a move towards more graph-based code representation as these representation models make it easier to learn semantic information. However, graph-based approaches are not without challenges, and more research in this direction will be needed.

11 | CONCLUSION

This study has presented a systematic mapping study on 103 primary studies that use code representation in the context of DL for software engineering. Our mapping study has classified the software task into four main categories depending on the input and output of the DL model (code-code, code-prediction, code-text, and text-code). Our study showed that code-code and code-prediction are the most addressed software tasks. We have also observed that tree-based and token-based approaches are the most common representation approaches applied in the investigated studies. However, we have also observed that there is a trend towards hybrid representations (which combine multiple different representation approaches) as well as the preferred usage of graph-based representations in newer studies. We identify two primary challenges in current literature: (1) there is a lack of generalisability of the presented approaches to other tasks (i.e., there are few attempts at transfer learning between tasks) and (2) very few studies validate the proposed framework on industrial datasets. We argue that these two problems constitute severe threats to the practical usefulness of current code representation research in the field of software engineering.

IoT (Rek:20200067), and by the Swiss National Science Foundation (SNSF) project "Melise—Machine Learning Assisted Software Development" (SNSF 204632).

CONFLICT OF INTEREST

DATA AVAILABILITY STATEMENT

ORCID

Hazem Peter Samoaa 🔘 https://orcid.org/0000-0001-5293-3388

REFERENCES

1. Bui, N., Jiang, L., Yu, Y.: Cross-language learning for program classification using bilateral tree-based convolutional neural networks. In: The Workshops of the Thirty-Second AAAI Conference on Artificial Intelligence (2018)

2. Kanade, A., et al.: Pre-trained contextual embedding of source code. In: ICLR 2020 Conference Program Chairs (2020)

3. Mou, L., et al.: Convolutional neural networks over tree structures for programming language processing. In: Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, AAAI '16, pp. 1287–1293. AAAI Press (2016)

4. Bui, N.D.Q., Yu, Y., Jiang, L.: Infercode: self-supervised learning of code representations by predicting subtrees. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), pp. 1186–-1197 (2021)

5. Fang, C., et al.: Functional code clone detection with syntax and semantics fusion learning. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2020, pp. 516–527. Association for Computing Machinery, New York, NY, USA (2020)

6. Mehrotra, N., et al.: Modeling functional similarity in source code with graph-based siamese networks. IEEE Trans. Softw. Eng.(01), 1 (2020). https://doi.org/10.1109/tse.2021.3105556

7. Hua, J., Wang, H.: On the effectiveness of deep vulnerability detectors to simple stupid bug detection. In: 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), pp. 530–534 (2021)

8. Li, Y., Wang, S., Nguyen, T.: Fault localization with code coverage representation learning. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), pp. 661–673 (2021)

9. Shi, K., et al.: Mpt-embedding: an unsupervised representation learning of code for software defect prediction. J. Softw.: Evol. Process. 33(4), e2330 (2021). e2330 smr.2330. https://doi.org/10.1002/smr.2330

10. Bui, N.D.Q., Yu, Y., Jiang, L.: Self-supervised contrastive learning for code retrieval and summarization via semantic-preserving transformations. In: SIGIR '21. Association for Computing Machinery, New York, NY, USA (2021)

11. Liu, S., et al.: Retrieval-augmented generation for code summarization via hybrid GNN. In: International Conference on Learning Representations (2021)

12. Zhang, J., et al.: Retrieval-based neural source code summarization. In: 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE), pp. 1385–1397 (2020)

13. Jebnoun, H., et al.: Clones in deep learning code: what, where, and why? (2021). https://doi.org/10.48550/arXiv.2107.13614

14. Bengio, Y.: Learning deep architectures for AI. Now Publishers Inc., Delft (2009)

15. Keller, P., et al.: What you see is what it means! semantic representation learning of code based on visualization and transfer learning. ACM Trans. Softw. Eng. Methodol. 31(2), 1–34 (2021). https://doi.org/10.1145/3485135

16. Dey, T., et al.: Detecting and characterizing bots that commit code. In: Kim, S., et al. (eds.) MSR '20: 17th International Conference on Mining Software Repositories, Seoul, Republic of Korea, 29–30 June, 2020, pp. 209–219. ACM (2020)

17. Zhang, C., et al.: A survey of automatic source code summarization. Symmetry. 14(3), 471 (2022). https://doi.org/10.3390/sym14030471

18. Teller, V.: Speech and language processing: an introduction to natural language processing, computational linguistics, and speech recognition. Comput. Ling. 26(4), 638–641 (2000)

19. Niesler, T., Woodland, P.: A variable-length category-based n-gram language model. In: 1996 IEEE International Conference on Acoustics, Speech, and Signal Processing Conference Proceedings, vol. 1, pp. 164–167 (1996)

20. Mikolov, T., et al.: Efficient estimation of word representations in vector space. In: ICLR (Workshop Poster) (2013)

21. Zhang, J., et al.: A novel neural source code representation based on abstract syntax tree. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), pp. 783–794 (2019)

22. White, M., et al.: Deep learning code fragments for code clone detection. In: 2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 87–98 (2016)

23. Wei, H.-H., Li, M.: Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. In: Proceedings of the 26th International Joint Conference on Artificial Intelligence, IJCAI'17, pp. 3034–3040. AAAI Press (2017)

24. Cummins, C., et al.: Programl: graph-based deep learning for program optimization and analysis. In: arXiv preprint, arXiv:2003.10536 (2020)

25. Zhao, G., Huang, J.: Deepsim: deep learning code functional similarity. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 141–151 (2018)

26. Laaber, C., Basmaci, M., Salza, P.: Predicting unstable software benchmarks using static source code features. Empir. Softw. Eng. 26(5), 114 (2021). https://doi.org/10.1007/s10664-021-09996-y

27. Samoaa, H.P., et al.: A Structured Literature Study of Source Code Representation for Deep Learning in Software Engineering [Replication Package]. Zenodo (2021). https://doi.org/10.5281/zenodo.6466506

28. Devlin, J., et al.: Semantic code repair using neuro-symbolic transformation networks. In: arXiv preprint, arXiv:1710.11054(2017)

29. Malik, R.S., Patra, J., Pradel, M.: Nl2type: inferring javascript function types from natural language information. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), pp. 304–315 (2019)

30. Pradel, M., et al.: Typewriter: neural type prediction with search-based validation. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2020, pp. 209–220. Association for Computing Machinery, New York, NY, USA (2020)

31. Cambronero, J., et al.: When deep learning met code search. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2019, pp. 964–974. Association for Computing Machinery, New York, NY, USA (2019)

32. Maurel, H., Vidal, S., Rezk, T.: Statically identifying XSS using deep learning. In: SECRYPT 2021 – 18th International Conference on Security and Cryptography, Virtual, France, July 2021 (2021)

33. Allamanis, M., Brockschmidt, M., Khademi, M.: Learning to represent programs with graphs. In: arXiv preprint, arXiv:1711.00740 (2017)

34. Liu, W., et al.: A survey of deep neural network architectures and their applications. Neurocomputing. 234, 11–26 (2017). https://doi.org/10.1016/j.neucom.2016.12.038

35. Arel, I., Rose, D.C., Karnowski, T.P.: Deep machine learning-a new frontier in artificial intelligence research [research frontier]. IEEE

Comput. Intell. Mag. 5(4), 13–18 (2010). https://doi.org/10.1109/mci.2010.938364

36. Sherstinsky, A.: Fundamentals of recurrent neural network (rnn) and long short-term memory (lstm) network. Phys. D: Nonlinear Phenom. 404, 132306 (2020). https://doi.org/10.1016/j.physd.2019.132306

37. Scarselli, F., et al.: The graph neural network model. IEEE Trans. Neural Network. 20(1), 61–80 (2008). https://doi.org/10.1109/tnn.2008.2005605

38. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural Comput. 9(8), 1735–1780 (1997). https://doi.org/10.1162/neco.1997.9.8.1735

39. Vaswani, A., et al.: Attention is all you need. In: Advances in Neural Information Processing Systems, pp. 5998–6008 (2017)

40. Chakraborty, S., et al.: Codit: code editing with tree-based neural models. IEEE Trans. Softw. Eng., 48(4), 1–1399 (2020). https://doi.org/10.1109/tse.2020.3020502

41. Cao, D., et al.: Ftclnet: convolutional lstm with Fourier transform for vulnerability detection. In: 2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), pp. 539–546 (2020)

42. Li, Y., et al.: Improving bug detection via context-based code representation learning and attention-based neural networks. Proc. ACM Program. Lang. 3(OOPSLA), 1–30 (2019). https://doi.org/10.1145/3360588

43. Ahmad, W.U., et al.: A transformer-based approach for source code summarization. In: arXiv preprint, arXiv:2005.00653 (2020)

44. Alon, U., et al.: Code2vec: learning distributed representations of code. Proc. ACM Program. Lang. 3(POPL), 1–29 (2019). https://doi.org/10.1145/3290353

45. Shuai, J., et al.: Improving code search with co-attentive representation learning. In: Proceedings of the 28th International Conference on Program Comprehension, ICPC '20, pp. 196–207. Association for Computing Machinery, New York, NY, USA (2020)

46. Markovtsev, V., et al.: Style-analyzer: fixing code style inconsistencies with interpretable unsupervised algorithms. In: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), pp. 468–478 (2019)

47. Csuvik, V., Kicsi, A., Vidács, L.: Source code level word embeddings in aiding semantic test-to-code traceability. In: 2019 IEEE/ACM 10th International Symposium on Software and Systems Traceability (SST), pp. 29–36 (2019)

48. Ben-Nun, T., Jakobovits, A.S., Hoefler, T.: Neural code comprehension: a learnable representation of code semantics. In: Proceedings of the 32nd International Conference on Neural Information Processing Systems, NIPS'18, pp. 3589–3601. Curran Associates Inc, Red Hook, NY, USA (2018)

49. Ramadan, T., et al.: Comparative code structure analysis using deep learning for performance prediction. In: 2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pp. 151–161 (2021)

50. Hadj-Kacem, M., Bouassida, N.: Deep representation learning for code smells detection using variational auto-encoder. In: 2019 International Joint Conference on Neural Networks (IJCNN), pp. 1–8 (2019)

51. DeFreez, D., Thakur, A.V., Rubio-González, C.: Path-based function embedding and its application to error-handling specification mining. In: ESEC/FSE 2018, pp. 423–433. Association for Computing Machinery, New York, NY, USA (2018)

52. Ampatzoglou, A., et al.: Identifying, categorizing and mitigating threats to validity in software engineering secondary studies. Inf. Softw. Technol. 106, 201–230 (2019). https://doi.org/10.1016/j.infsof.2018.10.006

53. Cheng, D., et al.: Manifesting bugs in machine learning code: an explorative study with mutation testing. In: 2018 IEEE International Conference on Software Quality, Reliability and Security (QRS), pp. 313–324. IEEE (2018)

54. Narayanan, A., et al.: graph2vec: Learning distributed representations of graphs. In: arXiv Preprints, arXiv:1707.05005v1 (2017)

55. Ou, M., et al.: Asymmetric transitivity preserving graph embedding. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '16, pp. 1105–1114. Association for Computing Machinery, New York, NY, USA (2016)

56. Goyal, P., Ferrara, E.: Graph embedding techniques, applications, and performance: a survey. Knowl. Base Syst. 151, 78–94 (2018). https://doi.org/10.1016/j.knosys.2018.03.022

57. Grover, A., Leskovec, J.: Node2vec: scalable feature learning for networks. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '16, pp. 855–864. Association for Computing Machinery, New York, NY, USA (2016)

58. Zhang, J., et al.: Retrieval-based neural source code summarization. In: Rothermel, G., Bae, D. (eds.) ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June – 19 July, 2020, pp. 1385–1397. ACM (2020)

59. Wang, W., et al.: Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In: Kontogiannis, K., et al. (eds.) 27th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2020, London, ON, Canada, February 18–21, 2020, pp. 261–271. IEEE (2020)

60. Cvitkovic, M., Singh, B., Anandkumar, A.: Open vocabulary learning on source code with a graph-structured cache. In: International Conference on Machine Learning, pp. 1475–1485. PMLR (2019)

61. Gers, F.A., Schmidhuber, J., Cummins, F.: Learning to forget: continual prediction with lstm. Neural Comput. 12(10), 2451–2471 (2000). https://doi.org/10.1162/089976600300015015

62. LeCun, Y., et al.: Gradient-based learning applied to document recognition. Proc. IEEE. 86(11), 2278–2324 (1998). https://doi.org/10.1109/5.726791

63. Rumelhart, D.E., Hinton, G.E., Williams, R.J.: Learning Internal Representations by Error Propagation (Technical report). California Univ San Diego La Jolla Inst for Cognitive Science (1985)

64. Li, Z., et al.: Vuldeepecker: a deep learning-based system for vulnerability detection. In: Proceedings 2018 Network and Distributed System Security Symposium (2018)

65. Marastoni, N., Giacobazzi, R., Dalla Preda, M.: A deep learning approach to program similarity. In: MASES 2018, pp. 26–35. Association for Computing Machinery, New York, NY, USA (2018)

66. Wang, Y., Cai, W.-d., Wei, P.-c.: A deep learning approach for detecting malicious javascript code. Secur. Commun. Network. 9(11), 1520–1534 (2016). https://doi.org/10.1002/sec.1441

67. Tufano, M., et al.: Deep learning similarities from different representations of source code. In: 2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR), pp. 542–553 (2018)

68. Fernandes, P., Allamanis, M., Brockschmidt, M.: Structured neural summarization. In: Conference paper at ICLR (2019)

69. Gupta, R., et al.: Fixing common c language errors by deep learning. In: Thirty-First AAAI Conference on Artificial Intelligence (2017)

70. Yasunaga, M., Liang, P.: Graph-based, self-supervised program repair from diagnostic feedback. In: International Conference on Machine Learning, pp. 10799–10808. PMLR (2020)

71. Fernandes, P., Allamanis, M., Brockschmidt, M.: Structured neural summarization. In: International Conference on Learning Representations (2019)

72. Brockschmidt, M., et al.: Generative code modeling with graphs. In: arXiv Preprint, arXiv:1805.08490 (2018).

73. Büch, L., Andrzejak, A.: Learning-based recursive aggregation of abstract syntax trees for code clone detection. In: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 95–104 (2019)

74. Collard, M.L., Decker, M.J., Maletic, J.I.: srcml: An infrastructure for the exploration, analysis, and manipulation of source code: a tool demonstration. In: 2013 IEEE International Conference on Software Maintenance, pp. 516–519 (2013)

75. Alon, U., et al.: A general path-based representation for predicting program properties. SIGPLAN Not. 53(4), 404–419 (2018). https://doi.org/10.1145/3296979.3192412

76. Lattner, C., Adve, V.: Llvm: a compilation framework for lifelong program analysis & transformation. In: International Symposium on Code Generation and Optimization, 2004. CGO 2004, pp. 75–86 (2004)

77. Lam, P., et al.: The soot framework for java program analysis: a retrospective. In: Conference: Cetus Users and Compiler Infrastructure Workshop (CETUS 2011) (2011)

78. Cytron, R., et al.: Efficiently computing static single assignment form and the control dependence graph. ACM Trans. Program Lang. Syst. 13(4), 451–490 (1991). https://doi.org/10.1145/115372.115320

79. Grunwald, D., Srinivasan, H.: Data flow equations for explicitly parallel programs. SIGPLAN Not. 28(7), 159–168 (1993). https://doi.org/10.1145/173284.155349

80. Allen, F.E., Cocke, J.: A program data flow analysis procedure. Commun. ACM. 19(3), 137 (1976). https://doi.org/10.1145/360018.360025

81. Vasic, M., et al.: Neural program repair by jointly learning to localize and repair. In: International Conference on Learning Representations (2019)

82. Santos, E.A., et al.: Syntax and sensibility: using language models to detect and correct syntax errors. In: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 311–322 (2018)

83. Dinella, E., et al.: Hoppity: learning graph transformations to detect and fix bugs in programs. In: International Conference on Learning Representations (2020)

84. Jia, R., Liang, P.: Adversarial examples for evaluating reading comprehension systems. In: arXiv preprint, arXiv:1707.07328 (2017)

85. Alon, U., et al.: code2seq: Generating sequences from structured representations of code. In: International Conference on Learning Representations (2019)

86. Kang, H.J., Bissyandé, T.F., Lo, D.: Assessing the generalizability of code2vec token embeddings. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 1–12 (2019)

87. Wei, B., et al.: Code generation as a dual task of code summarization. In: 33rd Conference on Neural Information Processing Systems (NeurIPS) (2019)

88. Li, Z., et al.: Sysevr: a framework for using deep learning to detect software vulnerabilities. IEEE Trans. Dependable Secure Comput., 1 (2021). https://doi.org/10.1109/tdsc.2021.3051525

89. Sonnekalb, T.: Machine-learning supported vulnerability detection in source code. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2019, pp. 1180–1183. Association for Computing Machinery, New York, NY, USA (2019)

90. Grover, A., Leskovec, J.: node2vec: Scalable feature learning for networks. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 855–864 (2016)

91. Costa, D., et al.: What's wrong with my benchmark results? Studying bad practices in jmh benchmarks. IEEE Trans. Softw. Eng. 47(7), 1452–1467 (2021). https://doi.org/10.1109/tse.2019.2925345

92. Laaber, C., Scheuner, J., Leitner, P.: Software microbenchmarking in the cloud. How bad is it really? Empir. Softw. Eng. 24(4), 2469–2508 (2019). https://doi.org/10.1007/s10664-019-09681-1

93. Tan, C., et al.: A survey on deep transfer learning. In: Kůrková, V., et al. (eds.) Artificial Neural Networks and Machine Learning – ICANN 2018, pp. 270–279. Springer International Publishing, Cham (2018)

94. Huang, G., et al.: Densely connected convolutional networks. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (2017)

95. Devlin, J., et al.: Bert: pre-training of deep bidirectional transformers for language understanding. In: arXiv preprint, arXiv:1810.04805 (2018)

96. Panichella, A., et al.: How to effectively use topic models for software engineering tasks? An approach based on genetic algorithms. In: 2013 35th International Conference on Software Engineering (ICSE), pp. 522–531 (2013)

97. Henkel, J., et al.: Code vectors: understanding programs through embedded abstracted symbolic traces. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2018, pp. 163–174. Association for Computing Machinery, New York, NY, USA (2018)

98. Allamanis, M., Brockschmidt, M., Khademi, M.: Learning to represent programs with graphs. In: International Conference on Learning Representations (2018)

99. Chen, Z., et al.: Sequencer: sequence-to-sequence learning for end-to-end program repair. IEEE Trans. Softw. Eng. 47(9), 1943–1959 (2021). https://doi.org/10.1109/tse.2019.2940179

100. Cheng, X., et al.: Deepwukong: statically detecting software vulnerabilities using deep graph neural network. ACM Trans. Softw. Eng. Methodol. 30(3), 1–33 (2021). https://doi.org/10.1145/3436877

101. Li, Z., et al.: Vuldeelocator: a deep learning-based fine-grained vulnerability detector. IEEE Trans. Dependable Secure Comput. (2021)

102. Amodio, M., Chaudhuri, S., Reps, T.W.: Neural attribute machines for program generation. In: arXiv e-prints, arXiv:1705.09231 (2017)

103. Haque, S., et al.: Improved automatic summarization of subroutines via attention to file context. In: Proceedings of the 17th International Conference on Mining Software Repositories, MSR '20, pp. 300–310. Association for Computing Machinery, New York, NY, USA (2020)

104. Fujiwara, Y., et al.: Code-to-code search based on deep neural network and code mutation. In: 2019 IEEE 13th International Workshop on Software Clones (IWSC), pp. 1–7 (2019)

105. Gupta, R., Kanade, A., Shevade, S.: Neural attribution for semantic bug-localization in student programs. In: Wallach, H., et al. (eds.) Advances in Neural Information Processing Systems, vol. 32. Curran Associates, Inc., New York (2019)

106. Liu, S., et al.: Deepbalance: deep-learning and fuzzy oversampling for vulnerability detection. IEEE Trans. Fuzzy Syst. 28(7), 1329–1343 (2020). https://doi.org/10.1109/tfuzz.2019.2958558

107. Nair, A., Roy, A., Funcgnn, K.M.: A graph neural network approach to program similarity. In: Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), ESEM '20. Association for Computing Machinery, New York, NY, USA (2020)

108. Sheneamer, A., Kalita, J.: Semantic clone detection using machine learning. In: 2016 15th IEEE International Conference on Machine Learning and Applications (ICMLA), pp. 1024–1028 (2016)

109. Svyatkovskiy, A., et al.: Fast and memory-efficient neural code completion. In: 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), pp. 329–340 (2021)

110. Liu, K., et al.: Learning to spot and refactor inconsistent method names. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), pp. 1–12 (2019)

111. Shi, K., et al.: Pathpair2vec: an ast path pair-based code representation method for defect prediction. J. Comput. Lang. 59, 100979 (2020). https://doi.org/10.1016/j.cola.2020.100979

112. Li, J., et al.: Software defect prediction via convolutional neural network. In: 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS), pp. 318–328 (2017)

113. Perez, D., Chiba, S.: Cross-language clone detection by learning over abstract syntax trees. In: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), pp. 518–528 (2019)

114. Li, L., et al.: A deep learning-based clone detection approach. In: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 249–260 (2017)

115. Gu, X., Zhang, H., Kim, S.: Deep code search. In: 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), pp. 933–944 (2018)

116. Hu, X., et al.: Deep code comment generation. In: 2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC), pp. 200–20010 (2018)

117. Sun, Z., et al.: A grammar-based structural cnn decoder for code generation. Proc. AAAI Conf. Artif. Intell. 33, 7055–7062 (2019). https://doi.org/10.1609/aaai.v33i01.33017055

118. Wang, R., et al.: Fret: functional reinforced transformer with bert for code summarization. IEEE Access. 8, 135591–135604 (2020). https://doi.org/10.1109/access.2020.3011744

119. Murali, V., et al.: Neural sketch learning for conditional program generation. In: International Conference on Learning Representations (2018)

120. Svyatkovskiy, A., et al.: Pythia: AI-assisted code completion system. In: Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery; Data Mining, KDD '19, pp. 2727–2735. Association for Computing Machinery, New York, NY, USA (2019)

121. Wang, W., et al.: Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In: 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 261–271 (2020)

122. Wu, H., Zhao, H., Zhang, H.: Code summarization with structure-induced transformer. In: Association for Computational Linguistics: ACL-IJCNLP 2021 (2021)

123. Li, J., et al.: Code completion with neural attention and pointer networks. In: Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence (2018)

124. Pradel, M., Sen, K.: Deep learning to find bugs. TU Darmstadt, Department of Computer Science. 4, 1 (2017)

125. Lin, G., et al.: Deep learning-based vulnerable function detection: a benchmark. In: Zhou, J., et al. (eds.) Information and Communications Security, pp. 219–232. Springer International Publishing, Cham (2020)

126. Iyer, S., et al.: Summarizing source code using a neural attention model. In: Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pp. 2073–2083 (2016)

127. Raychev, V., Vechev, M., Yahav, E.: Code completion with statistical language models. SIGPLAN Not. 49(6), 419–428 (2014). https://doi.org/10.1145/2666356.2594321

128. Xie, C., et al.: A source code similarity based on siamese neural network. Appl. Sci. 10(21), 7519 (2020). https://doi.org/10.3390/app10217519

129. Wang, S., Liu, T., Tan, L.: Automatically learning semantic features for defect prediction. In: 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), pp. 297–308 (2016)

130. Tufano, M., et al.: An empirical study on learning bug-fixing patches in the wild via neural machine translation. ACM Trans. Softw. Eng. Methodol. 28(4), 1–29 (2019). https://doi.org/10.1145/3340544

131. Dam, H.K., Tran, T., Pham, T.: A deep language model for software code. In: arXiv preprint, arXiv:1608.02715 (2016)

132. Pradel, M., Sen, K.: Deepbugs: a learning approach to name-based bug detection. Proc. ACM Program. Lang. 2(OOPSLA), 1–25 (2018). https://doi.org/10.1145/3276517

133. Russell, R., et al.: Automated vulnerability detection in source code using deep representation learning. In: 2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA), pp. 757–762 (2018)

134. Wang, K., Su, Z.: Learning blended, precise semantic program embeddings. Proc. ACM Program. Lang. 1, 1–25 (2019)

135. White, M., et al.: Sorting and transforming program repair ingredients via deep learning code similarities. In: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 479–490 (2019)

136. Yu, H., et al.: Neural detection of semantic code clones via tree-based convolution. In: 2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC), pp. 70–80 (2019)

137. Yin, P., Neubig, G.: A syntactic neural model for general-purpose code generation. In: Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (2017)

138. Zeng, J., et al.: Fast code clone detection based on weighted recursive autoencoders. IEEE Access. 7, 125062–125078 (2019). https://doi.org/10.1109/access.2019.2938825

139. White, M., et al.: Toward deep learning software repositories. In: 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories, pp. 334–345 (2015)

140. Mou, L., et al.: Tbcnn: a tree-based convolutional neural network for programming language processing. In: arXiv preprint, arXiv:1409.5718 (2014)

141. Zhou, M., et al.: Deeptle: learning code-level features to predict code performance before it runs. In: 2019 26th Asia-Pacific Software Engineering Conference (APSEC), pp. 252–259 (2019)

142. Allamanis, M., Peng, H., Sutton, C.: A convolutional attention network for extreme summarization of source code. In: Balcan, M.F., Weinberger, K. Q. (eds.) Proceedings of the 33rd International Conference on Machine Learning, Volume 48 of Proceedings of Machine Learning Research, 20–22 Jun 2016, pp. 2091–2100. PMLR, New York, New York, USA (2016)

143. Wan, Y., et al.: Improving automatic source code summarization via deep reinforcement learning. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, pp. 397–407. Association for Computing Machinery, New York, NY, USA (2018)

144. Zhou, Y., et al.: Devign: effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In: NeurIPS (2019)

APPENDICES

Analysis of the main attributes

Authors	Title	Venue	Year	Cit. Key
U. Alon, M. Zilberstein, O. Levy, and A. Yahav	code2vec: learning distributed representations of code	POPL	2019	[44]
U. Alon, S. Brody, O. Levy, E. Yahav	code2seq: Generating Sequences from Structured Representations of Code	ICLR	2019	[85]
M. Brockschmidt, M. Allamanis, A.L. Gaunt, O. Polozov	Generative Code Modelling with Graphs	ICLR	2019	[72]
W. U. Ahmad, S. Chakraborty, B. Ray, K. Chang	A Transformer-based Approach for Source Code Summarization	ACL	2020	[43]
Nghi D. Q. Bui, Lingxiao Jiang, Yijun Yu	Cross-Language Learning for Program Classification using Bilateral Tree-Based Convolutional Neural Networks	AAAI	2017	[1]

(Continues)

A P P E N D I X (Continued)

Authors	Title	Venue	Year	Cit. Key
Nghi D. Q. Bui, Yijun Yu, Lingxiao Jiang	Self-Supervised Contrastive Learning for Code Retrieval and Summarization via Semantic-Preserving Transformations	SIGIR	2021	[10]
J. Devlin, J. Uesato, R. Singh, P. Kohli	Semantic Code Repair using Neuro-Symbolic Transformation Networks	arXiv	2017	[28]
M. Allamanis, M. Brockschmidt, M. Khademi	Learning to Represent Programs with Graphs	ICLR	2018	[33]
L. Büch, A. Andrzejak	Learning-Based Recursive Aggregation of Abstract Syntax Trees for Code Clone Detection	SANER	2019	[73]
R. Gupta, S. Pal, A. Kanade, S. Shevade	DeepFix: Fixing Common C Language Errors by Deep Learning	AAAI	2017	[69]
J. Cambronero, H. Li, S. Kim, K. Sen, S. Chandra	When Deep Learning Met Code Search	FSE	2019	[31]
S. Chakraborty, Y. Ding, M. Allamanis, B. Ray	CODIT: Code Editing with Tree-Based Neural Models	TSE	2019	[40]
D. Cao, J. Huang, X. Zhang, X. Liu	FTCLNet: Convolutional LSTM with Fourier Transform for Vulnerability Detection	TrustCom	2020	[41]
Nghi D. Q. Bui, Y. Yu, L. Jiang	InferCode: Self-Supervised Learning of Code Representations by Predicting Subtrees	ICSE	2021	[4]
Z. Chen, S. Kommrusch, M. Tufano, L. Pouchet, D. Poshyvanyk, M. Monperrus	SEQUENCER: Sequence-to-Sequence Learning for End-to-End Program Repair	TSE	2021	[99]
X. Cheng, H. Wang, J. Hua, G. Xu, Y. Sui	DeepWukong: Statically Detecting Software Vulnerabilities Using Deep Graph Neural Network	TOSEM	2021	[100]
J. Hua, H. Wang	On the Effectiveness of Deep Vulnerability Detectors to Simple Stupid Bug Detection	MSR	2021	[7]
U. Alon, M. Zilberstein, O. Levy, E. Yahav	A general path-based representation for predicting program properties	SIGPLAN	2018	[75]
T. Ben-Nun, A. S. Jakobovits, T. Hoefler	Neural Code Comprehension: A Learnable Representation of Code Semantics	NeurIPS	2018	[48]
V. Csuvik, A. Kicsi, L. Vidács	Source Code Level Word Embeddings in Aiding Semantic Test-to-Code Traceability	ICSE	2019	[47]
M. Cvitkovic, B. Singh, A. Anandkumar	Open Vocabulary Learning on Source Code with a Graph-Structured Cache	ICML	2019	[60]
Z. Li, D. Zou, S. Xu, Z. Chen, Y. Zhu, H. Jin	VulDeeLocator: A Deep Learning-based Fine-grained Vulnerability Detector	TDSC	2021	[101]
Y. Li, S. Wang, T. N. Nguyen	Fault Localization with Code Coverage Representation Learning	ICSE	2021	[8]
S. Liu, Y. Chen, X. Xie, J. K. Siow, Y. Liu	Retrieval-Augmented Generation for Code Summarization via Hybrid GNN	ICLR	2021	[11]
H. Maurel, S. Vidal, T. Rezk	Statically Identifying XSS using Deep Learning	SECRYPT	2021	[32]
C. Cummins, Z. V. Fisches, T. Ben-Nun, T. Hoefler, H. Leather	PROGRAML: Graph-based Deep Learning for Program Optimization and Analysis	PMLR	2021	[24]
P. Fernandes, M. Allamanis, M. Brockschmidt	Structured Neural Summarization	ICLR	2019	[71]
M. Amodio, S. Chaudhuri, T. Reps	Neural Attribute Machines for Program Generation	arXiv	2021	[102]
E. Dinella, H. Dai, Z. Li, M. Naik, L. Song, K. Wang	Hoppity: Learning graph transformations to detect and fix bugs in programs	ICLR	2020	[83]
C. Fang, Z. Liu, Y. Shi, J. Huang, Q. Shi	Functional Code Clone Detection with Syntax and Semantics Fusion Learning	ISSTA	2020	[5]
S. Haque, A. LeClair, L. Wu, C. McMillan	Improved Automatic Summarization of Subroutines via Attention to File Context	MSR	2020	[103]
Y. Fujiwara, N. Yoshida, E. Choi, K. Inoue	Code-to-Code Search Based on Deep Neural Network and Code Mutation	IWSC	2019	[104]

APPENDIX (Continued)

Authors	Title	Venue	Year	Cit. Key
T. Ramadan, T.Z. Islam, C. Phelps	Comparative Code Structure Analysis using Deep Learning for Performance Prediction	ISPASS	2021	[49]
A. Kanade, P. Maniatis, G. Balakrishnan, K. Shi	Pre-trained Contextual Embedding of Source Code	ICLR	2020	[2]
D. DeFreez, A.V. Thakur, C. Rubio-González	Path-based function embedding and its application to error-handling specification mining	FSE	2018	[51]
R. Gupta, A. Kanade, S. Shevade	Neural Attribution for Semantic Bug-Localization in Student Programs	NeurIPS	2019	[105]
M. Hadj-Kacem, N. Bouassida	Deep Representation Learning for Code Smells Detection using Variational Auto-Encoder	IJCNN	2019	[50]
S. Liu, G. Lin, Q.L. Han, S. Wen, J. Zhang, Y. Xiang	DeepBalance: Deep-Learning and Fuzzy Oversampling for Vulnerability Detection	Transactions on Fuzzy Systems	2020	[106]
N. Mehrotra, N. Agarwal, P. Gupta, S. Anand, D. Lo, R. Purandare	Modelling Functional Similarity in Source Code with Graph-Based Siamese Networks	TSE	2020	[6]
K. Shi, Y. Lu, G. Liu, Z. Wei, J. Chang	MPT-embedding: An unsupervised representation learning of code for software defect prediction	SEP	2021	[9]
A. Nair, A. Roy, K. Meinke	funcGNN: A Graph Neural Network Approach to Program Similarity	ESEM	2020	[107]
A. Sheneamer, J. Kalita	Semantic Clone Detection Using Machine Learning	ICMLA	2016	[108]
H.J. Kang, T.F. Bissyandé, D. Lo	Assessing the Generalizability of code2vec Token Embeddings	ASE	2019	[86]
M. Pradel, G. Gousios, J. Liu, S. Chandra	TypeWriter: Neural-Type Prediction with Search-Based Validation	FSE	2020	[30]
Y. Li, S. Wang, T.N. Nguyen, S. Van Nguyen	Improving Bug Detection via Context-Based Code Representation Learning and Attention-Based Neural Networks	OOPSLA	2019	[42]
A. Svyatkovskiy, S. Lee, A. Hadjitofi	Fast and Memory-Efficient Neural Code Completion	MSR	2021	[109]
K. Liu, D. Kim, T.F. Bissyandé, T. Kim	Learning to Spot and Refactor Inconsistent Method Names	ICSE	2019	[110]
R.S. Malik, J. Patra, M. Pradel	NL2Type: Inferring JavaScript Function Types from Natural Language Information	ICSE	2019	[29]
K. Shi, Y. Lu, J. Chang, Z. Wei	PathPair2Vec: An AST path pair-based code representation method for defect prediction	JCL	2020	[111]
V. Markovtsev, W. Long, H. Mougard	STYLE-ANALYZER: fixing code style inconsistencies with interpretable unsupervised algorithms	MSR	2019	[46]
J. Li, P. He, J. Zhu, M.R. Lyu	Software Defect Prediction via Convolutional Neural Network	QRS	2017	[112]
D. Perez, S. Chiba	Cross-language clone detection by learning over abstract syntax trees	MSR	2019	[113]
L. Li, H. Feng, W. Zhuang, N. Meng	CCLearner: A Deep Learning-Based Clone Detection Approach	ICSME	2017	[114]
T. Sonnekalb	Machine-Learning Supported Vulnerability Detection in Source Code	FSE	2019	[89]
X. Gu, H. Zhang, S. Kim	Deep Code Search	ICSE	2018	[115]
J. Henkel, S.K. Lahiri, B. Liblit, T. Reps	Code Vectors: Understanding Programs Through Embedded Abstracted Symbolic Traces	FSE	2018	[97]
X. Hu, G. Li, X. Xia, D. Lo, Z. Jin	Deep Code Comment Generation	ICPC	2018	[116]
Z. Sun, Q. Zhu, L. Mou, Y. Xiong, G. Li	A Grammar-Based Structural CNN Decoder for Code Generation	AAAI	2019	[117]
J. Shuai, L. Xu, C. Liu, M. Yan, X. Xia, Y. Lei	Improving Code Search with Co-Attentive Representation Learning	ICPC	2020	[45]

APPENDIX   (Continued)

Authors	Title	Venue	Year	Cit. Key
R. Wang, H. Zhang, G. Lu, L. Lyu, C. Lyu	Fret: Functional Reinforced Transformer With BERT for Code Summarization	IEEE Access	2020	[118]
V. Murali, L. Qi, S. Chaudhuri, C. Jermaine	Neural Sketch Learning for Conditional Program Generation	ICLR	2017	[119]
A. Svyatkovskiy, Y. Zhao, S. Fu	Pythia: AI-assisted Code Completion System	SIGKDD	2019	[120]
W. Wang, G. Li, B. Ma, X. Xia, Z. Jin	Detecting Code Clones with Graph Neural Network and Flow-Augmented Abstract Syntax Tree	SANER	2020	[121]
H. Wu, H. Zhao, M. Zhang	SIT3: Code Summarization with Structure-Induced Transformer	ACL	2021	[122]
Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng	VulDeePecker: A Deep Learning-Based System for Vulnerability Detection	NDSS	2018	[64]
J. Li, Y. Wang, M.R. Lyu, I. King	Code Completion with Neural Attention and Pointer Networks	JICAI	2018	[123]
M. Pradel, K. Sen	Deep Learning to Find Bugs	arXiv	2017	[124]
M. White, M. Tufano, C. Vendome	Deep Learning Code Fragments for Code Clone Detection	ASE	2016	[22]
L. Mou, G. Li, L. Zhang, T. Wang, Z. Jin	Convolutional Neural Networks over Tree Structures for Programming Language Processing	AAAI	2016	[3]
G. Lin, W. Xiao, J. Zhang, Y. Xiang	Deep Learning-Based Vulnerable Function Detection	ICICS	2020	[125]
S. Iyer, I. Konstas, A. Cheung, L. Zettlemoyer	Summarizing Source Code using a Neural Attention Model	ACL	2016	[126]
H. Wei, M. Li	Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code	AAAI	2017	[23]
V. Raychev, M. Vechev, E. Yahav	Code completion with statistical language models	SIGPLAN	2014	[127]
N. Marastoni, R. Giacobazzi, M. Dalla Preda	A Deep Learning Approach to Program Similarity	MASES	2018	[65]
C. Xie, X. Wang, C. Qian, M. Wang	A Source Code Similarity Based on Siamese Neural Network	Applied Science	2020	[128].
Y. Wang, W. Cai, P. Wei	A deep learning approach for detecting malicious JavaScript code	SCN	2016	[66]
S. Wang, T. Liu, L. Tan	Automatically Learning Semantic Features for Defect Prediction	2016	ICSE	[129]
M. Yasunaga, P. Liang	Graph-based, Self-Supervised Program Repair from Diagnostic Feedback	ICML	2020	[70]
M. Tufano, C. Watson, G. Bavota, M.D. Penta	An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation	TOSEM	2019	[130]
H. Wei, M. Li	Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code	JICAI	2017	[23]
H.K. Dam, T. Tran, T. Pham	A deep language model for software code	arXiv	2016	[131]
M. Vasic, A. Kanade, P. Maniatis, D. Bieber	Neural program repair by jointly learning to localize and repair	ICLR	2018	[81]
M. Pradel, K. Sen	DeepBugs: A Learning Approach to Name-Based Bug Detection	OOPSLA	2018	[132]
R. Russell, L. Kim, L. Hamilton, T. Lazovich	Automated Vulnerability Detection in Source Code Using Deep Representation Learning	ICMLA	2018	[133]
K. Wang, Z. Su	Learning Blended, Precise Semantic Program Embeddings	PLDI	2020	[134]
E.A. Santos, J.C. Campbell, D. Patel	Syntax and Sensibility: Using Language Models to Detect and Correct Syntax Errors	SANER	2018	[82]
B. Wei, G. Li, X. Xia, Z. Fu, Z. Jin	Code Generation as a Dual Task of Code Summarization	NeurIPS	2019	[87]
M. White, M. Tufano, M. Martinez	Sorting and Transforming Program Repair Ingredients via Deep Learning Code Similarities	SANER	2019	[135]

A P P E N D I X (Continued)

Authors	Title	Venue	Year	Cit. Key
H. Yu, W. Lam, L. Chen, G. Li, T. Xie	Neural Detection of Semantic Code Clones Via Tree-Based Convolution	ICPC	2019	[136]
P. Yin, G. Neubig	A Syntactic Neural Model for General-Purpose Code Generation	ACL	2017	[137]
J. Zeng, K. Ben, X. Li, X. Zhang	Fast Code Clone Detection Based on Weighted Recursive Autoencoders	IEEE Access	2019	[138]
M. White, C. Vendome	Toward deep learning software repositories	MSR	2015	[139]
J. Zhang, X. Wang, H. Zhang, H. Sun	A Novel Neural Source Code Representation Based on Abstract Syntax Tree	ICSE	2019	[21]
M. Tufano, C. Watson, G. Bavota	Deep Learning Similarities from Different Representations of Source Code	MSR	2018	[67]
J. Zhang, X. Wang, H. Zhang, H. Sun	Retrieval-based Neural Source Code Summarization	ICSE	2020	[58]
L. Mou, G. Li, Z. Jin, L. Zhang, T. Wang	TBCNN: A tree-based convolutional neural network for programming language processing	arXiv	2014	[140]
L. Mou, G. Li, L. Zhang, T. Wang, Z. Jin	Convolutional Neural Networks over Tree Structures for Programming Language Processing	AAAI	2016	[3]
M. Zhou, J. Chen, H. Hu, J. Yu, Z. Li	DeepTLE: Learning Code-Level Features to Predict Code Performance before It Runs	APSEC	2019	[141]
M. Allamanis, H. Peng, C. Sutton	A Convolutional Attention Network for Extreme Summarization of Source Code	ICML	2016	[142]
G. Zhao, J. Huang	DeepSim: deep learning code functional similarity	FSE	2018	[25]
Y. Wan, Z. Zhao, M. Yang, G. Xu, H. Ying, J. Wu	Improving Automatic Source Code Summarization via Deep Reinforcement Learning	ASE	2018	[143]
Y. Zhou, S. Liu, J. Siow, X. Du, Y. Liu	Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks	NeurIPS	2019	[144]

B List of Venue Acronyms

Acronyms	Venue
AAAI	Association for the Advancement of Artificial Intelligence
ACL	Association for Computational Linguistics
ASE	International Conference on Automated Software Engineering
FSE	Fast Software Encryption
ICLR	International Conference on Learning Representations
ICML	International Conference on Machine Learning
ICMLA	International Conference on Machine Learning and Applications
ICPC	International Conference on Program Comprehension
ICSE	International Conference on Software Engineering
IJCAI	International Joint Conference on Artificial Intelligence
MSR	Mining Software Repositories
NeurIPS	Neural Information Processing Systems
PACMPL	Proceedings of the ACM on Programming Languages
PLDI	Programming Language Design and Implementation
SANER	International Conference on Software Analysis, Evolution and Reengineering

C SE Tasks and Related Papers
C.1 Main SE Tasks and Related Papers

Title	Code-code	Code-text	Text-code	Code-prediction
code2vec: Learning Distributed Representations of Code		✓		
code2seq: Generating Sequences from Structured Representations of Code		✓		
Generative Code Modelling with Graphs	✓			
A Transformer-based Approach for Source Code Summarization		✓		
Cross-Language Learning for Program Classification using Bilateral Tree-Based Convolutional Neural Networks				✓
Self-Supervised Contrastive Learning for Code Retrieval and Summarization via Semantic-Preserving Transformations	✓	✓	✓	
Semantic Code Repair using Neuro-Symbolic Transformation Networks				✓
Learning to Represent Programs with Graphs		✓		
Learning-Based Recursive Aggregation of Abstract Syntax Trees for Code Clone Detection	✓			
DeepFix: Fixing Common C Language Errors by Deep Learning	✓			
When Deep Learning Met Code Search			✓	
CODIT: Code Editing with Tree-Based Neural Models	✓			
FTCLNet: Convolutional LSTM with Fourier Transform for Vulnerability Detection				✓
InferCode: Self-Supervised Learning of Code Representations by Predicting Subtrees	✓			
SEQUENCER: Sequence-to-Sequence Learning for End-to-End Program Repair	✓			
DeepWukong: Statically Detecting Software Vulnerabilities Using Deep Graph Neural Network				✓
On the Effectiveness of Deep Vulnerability Detectors to Simple Stupid Bug Detection				✓
A general path-based representation for predicting program properties		✓		
Neural Code Comprehension: A Learnable Representation of Code Semantics				✓
Source Code Level Word Embeddings in Aiding Semantic Test-to-Code Traceability	✓			
Open Vocabulary Learning on Source Code with a Graph-Structured Cache	✓	✓		
VulDeeLocator: A Deep Learning-based Fine-grained Vulnerability Detector				✓
Fault Localization with Code Coverage Representation Learning				✓
SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities				✓
Retrieval-Augmented Generation for Code Summarization via Hybrid GNN		✓		
Statically Identifying XSS using Deep Learning				✓
PROGRAML: GRAPH-BASED DEEP LEARNING FOR PROGRAM OPTIMIZATION AND ANALYSIS	✓			
STRUCTURED NEURAL SUMMARIZATION		✓		
Neural Attribute Machines for Program Generation	✓			
Hoppity: Learning graph transformations to detect and fix bugs in programs.	✓			✓
Functional Code Clone Detection with Syntax and Semantics Fusion Learning	✓			
Improved Automatic Summarization of Subroutines via Attention to File Context		✓		
Code-to-Code Search Based on Deep Neural Network and Code Mutation			✓	
Comparative Code Structure Analysis using Deep Learning for Performance Prediction				✓
PRE-TRAINED CONTEXTUAL EMBEDDING OF SOURCE CODE				✓
Path-based function embedding and its application to error-handling specification mining				✓
Neural Attribution for Semantic Bug-Localization in Student Programs				✓

APPENDIX   (Continued)

Title	Code-code	Code-text	Text-code	Code-prediction
Deep Representation Learning for Code Smells Detection using Variational Auto-Encoder				✓
DeepBalance: Deep-Learning and Fuzzy Oversampling for Vulnerability Detection				✓
Modelling Functional Similarity in Source Code with Graph-Based Siamese Networks	✓			
MPT-embedding: An unsupervised representation learning of code for software defect prediction				✓
funcGNN: A Graph Neural Network Approach to Program Similarity	✓			
Semantic Clone Detection Using Machine Learning	✓			
Assessing the Generalizability of code2vec Token Embeddings	✓	✓		
TypeWriter: Neural Type Prediction with Search-Based Validation				✓
Improving Bug Detection via Context-Based Code Representation Learning and Attention-Based Neural Networks				✓
Fast and Memory-Efficient Neural Code Completion	✓			
Learning to Spot and Refactor Inconsistent Method Names		✓		
NL2Type: Inferring JavaScript Function Types from Natural Language Information				✓
PathPair2Vec: An AST path pair-based code representation method for defect prediction				✓
STYLE-ANALYZER: fixing code style inconsistencies with interpretable unsupervised algorithms	✓			
Software Defect Prediction via Convolutional Neural Network				✓
Cross-language clone detection by learning over abstract syntax trees	✓			
CCLearner: A Deep Learning-Based Clone Detection Approach	✓			
Machine-Learning Supported Vulnerability Detection in Source Code				✓
Deep Code Search			✓	
Code Vectors: Understanding Programs Through Embedded Abstracted Symbolic Traces	✓			
Deep Code Comment Generation		✓		
A Grammar-Based Structural CNN Decoder for Code Generation	✓			
Improving Code Search with Co-Attentive Representation Learning			✓	
Fret: Functional Reinforced Transformer With BERT for Code Summarization		✓		
Neural Sketch Learning for Conditional Program Generation	✓			
Pythia: AI-assisted Code Completion System	✓			
Detecting Code Clones with Graph Neural Network and Flow-Augmented Abstract Syntax Tree	✓			
SIT3: Code Summarization with Structure-Induced Transformer		✓		
VulDeePecker: A Deep Learning-Based System for Vulnerability Detection				✓
Code Completion with Neural Attention and Pointer Networks	✓			
Deep Learning to Find Bugs (With focus on name-based bug detectors)				✓
Deep Learning Code Fragments for Code Clone Detection	✓			
Convolutional Neural Networks over Tree Structures for Programming Language Processing				✓
Deep Learning-Based Vulnerable Function Detection: A Benchmark				✓
Summarizing Source Code using a Neural Attention Model		✓		
Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code	✓			
Code completion with statistical language models	✓			

(Continues)

A P P E N D I X   (Continued)

Title	Code-code	Code-text	Text-code	Code-prediction
A Deep Learning Approach to Program Similarity	✓			
A Source Code Similarity Based on Siamese Neural Network	✓			
A deep learning approach for detecting malicious JavaScript code				✓
Automatically Learning Semantic Features for Defect Prediction				✓
Graph-based, Self-Supervised Program Repair from Diagnostic Feedback	✓			
An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation	✓			
Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code	✓			
A deep language model for software code	✓			
Neural program repair by jointly learning to localize and repair	✓			✓
DeepBugs: A Learning Approach to Name-Based Bug Detection				✓
Automated Vulnerability Detection in Source Code Using Deep Representation Learning				✓
Blended, precise semantic program embeddings		✓		
Syntax and Sensibility: Using Language Models to Detect and Correct Syntax Errors	✓			✓
Code Generation as a Dual Task of Code Summarization		✓	✓	
Sorting and Transforming Program Repair Ingredients via Deep Learning Code Similarities	✓			
Neural Detection of Semantic Code Clones Via Tree-Based Convolution	✓			
A Syntactic Neural Model for General-Purpose Code Generation	✓			
Fast Code Clone Detection Based on Weighted Recursive Autoencoders	✓			
Toward deep learning software repositories	✓			
A Novel Neural Source Code Representation Based on Abstract Syntax Tree	✓			✓
Deep Learning Similarities from Different Representations of Source Code	✓			
Retrieval-based Neural Source Code Summarization		✓		
TBCNN: A tree-based convolutional neural network for programming language processing				✓
Convolutional Neural Networks over Tree Structures for Programming Language Processing				✓
DeepTLE: Learning Code-Level Features to Predict Code Performance before It Runs				
A Convolutional Attention Network for Extreme Summarization of Source Code		✓		
DeepSim: deep learning code functional similarity	✓			
Improving Automatic Source Code Summarization via Deep Reinforcement Learning		✓		
Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks				✓

C.2 Code–Code Tasks and Related Papers

Title	Code clone detection	Traceability	Code similarity detection	Program repair	Fixing format	Code completion	Compiler analysis	Program generation
Generative Code Modelling with Graphs								✓
Self-Supervised Contrastive Learning for Code Retrieval and Summarization via Semantic-Preserving Transformations			✓					

APPENDIX (Continued)

Title	Code clone detection	Traceability	Code similarity detection	Program repair	Fixing format	Code completion	Compiler analysis	Program generation
Learning-Based Recursive Aggregation of Abstract Syntax Trees for Code Clone Detection	✓							
DeepFix: Fixing Common C Language Errors by Deep Learning				✓				
CODIT: Code Editing with Tree-Based Neural Models				✓				✓
InferCode: Self-Supervised Learning of Code Representations by Predicting Subtrees	✓		✓					
SEQUENCER: Sequence-to-Sequence Learning for End-to-End Program Repair				✓				
Source Code Level Word Embeddings in Aiding Semantic Test-to-Code Traceability		✓						
Open Vocabulary Learning on Source Code with a Graph-Structured Cache						✓		
PROGRAML: GRAPH-BASED DEEP LEARNING FOR PROGRAM OPTIMIZATION AND ANALYSIS							✓	
Neural Attribute Machines for Program Generation								✓
Hoppity: Learning graph transformations to detect and fix bugs in programs.				✓				
Functional Code Clone Detection with Syntax and Semantics Fusion Learning	✓							
Modelling Functional Similarity in Source Code with Graph-Based Siamese Networks	✓							
funcGNN: A Graph Neural Network Approach to Program Similarity			✓					
Semantic Clone Detection Using Machine Learning	✓							
Assessing the Generalizability of code2vec Token Embeddings	✓							
Fast and Memory-Efficient Neural Code Completion						✓		
STYLE-ANALYZER: fixing code style inconsistencies with interpretable unsupervised algorithms					✓			
Cross-language clone detection by learning over abstract syntax trees	✓							
CCLearner: A Deep Learning-Based Clone Detection Approach	✓							
Code Vectors: Understanding Programs Through Embedded Abstracted Symbolic Traces			✓					
A Grammar-Based Structural CNN Decoder for Code Generation								✓
Neural Sketch Learning for Conditional Program Generation						✓		
Pythia: AI-assisted Code Completion System						✓		
Detecting Code Clones with Graph Neural Network and Flow-Augmented Abstract Syntax Tree	✓							

(Continues)

APPENDIX (Continued)

Title	Code clone detection	Traceability	Code similarity detection	Program repair	Fixing format	Code completion	Compiler analysis	Program generation
Code Completion with Neural Attention and Pointer Networks						✓		
Deep Learning Code Fragments for Code Clone Detection	✓							
Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code	✓							
Code completion with statistical language models						✓		
A Deep Learning Approach to Program Similarity			✓					
A Source Code Similarity Based on Siamese Neural Network			✓					
Graph-based, Self-Supervised Program Repair from Diagnostic Feedback				✓				
An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation				✓				
Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code	✓							
A deep language model for software code						✓		
Neural program repair by jointly learning to localize and repair				✓				
Syntax and Sensibility: Using Language Models to Detect and Correct Syntax Errors				✓				
Sorting and Transforming Program Repair Ingredients via Deep Learning Code Similarities				✓				
Neural Detection of Semantic Code Clones Via Tree-Based Convolution	✓							
A Syntactic Neural Model for General-Purpose Code Generation								✓
Fast Code Clone Detection Based on Weighted Recursive Autoencoders	✓		✓					
Toward deep learning software repositories								✓
A Novel Neural Source Code Representation Based on Abstract Syntax Tree	✓							
Deep Learning Similarities from Different Representations of Source Code	✓		✓					
DeepSim: deep learning code functional similarity			✓					

C.3 Code Prediction Tasks and Related Papers

Title	Source code classification	Code smell detection	Error handling	Bug detection	Malicious behaviour detection	Vulnerability detection	Performance prediction	Type signature prediction
Cross-Language Learning for Program Classification using Bilateral Tree-Based Convolutional Neural Networks	✓							

APPENDIX (Continued)

Title	Source code classification	Code smell detection	Error handling	Bug detection	Malicious behaviour detection	Vulnerability detection	Performance prediction	Type signature prediction
Semantic Code Repair using Neuro-Symbolic Transformation Networks				✓				
FTCLNet: Convolutional LSTM with Fourier Transform for Vulnerability Detection						✓		
DeepWukong: Statically Detecting Software Vulnerabilities Using Deep Graph Neural Network						✓		
On the Effectiveness of Deep Vulnerability Detectors to Simple Stupid Bug Detection				✓				
Neural Code Comprehension: A Learnable Representation of Code Semantics					✓			
VulDeeLocator: A Deep Learning-based Fine-grained Vulnerability Detector						✓		
Fault Localization with Code Coverage Representation Learning				✓				
SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities						✓		
Statically Identifying XSS using Deep Learning						✓		
Hoppity: Learning graph transformations to detect and fix bugs in programs				✓				
Comparative Code Structure Analysis using Deep Learning for Performance Prediction							✓	
PRE-TRAINED CONTEXTUAL EMBEDDING OF SOURCE CODE	✓							
Path-based function embedding and its application to error-handling specification mining			✓					
Neural Attribution for Semantic Bug-Localization in Student Programs				✓				
Deep Representation Learning for Code Smells Detection using Variational Auto-Encoder		✓						
DeepBalance: Deep-Learning and Fuzzy Oversampling for Vulnerability Detection						✓		
MPT-embedding: An unsupervised representation learning of code for software defect prediction				✓				
TypeWriter: Neural Type Prediction with Search-Based Validation								✓
Improving Bug Detection via Context-Based Code Representation Learning and Attention-Based Neural Networks								
NL2Type: Inferring JavaScript Function Types from Natural Language Information								✓

APPENDIX (Continued)

Title	Source code classification	Code smell detection	Error handling	Bug detection	Malicious behaviour detection	Vulnerability detection	Performance prediction	Type signature prediction
PathPair2Vec: An AST path pair-based code representation method for defect prediction				✓				
Software Defect Prediction via Convolutional Neural Network				✓				
Machine-Learning Supported Vulnerability Detection in Source Code						✓		
VulDeePecker: A Deep Learning-Based System for Vulnerability Detection						✓		
Deep Learning to Find Bugs (With focus on name-based bug detectors)				✓				
Convolutional Neural Networks over Tree Structures for Programming Language Processing	✓							
Deep Learning-Based Vulnerable Function Detection: A Benchmark						✓		
A deep learning approach for detecting malicious JavaScript code					✓			
Automatically Learning Semantic Features for Defect Prediction				✓				
Neural program repair by jointly learning to localize and repair				✓				
DeepBugs: A Learning Approach to Name-Based Bug Detection				✓				
Automated Vulnerability Detection in Source Code Using Deep Representation Learning						✓		
Syntax and Sensibility: Using Language Models to Detect and Correct Syntax Errors				✓				
A Novel Neural Source Code Representation Based on Abstract Syntax Tree	✓							
TBCNN: A tree-based convolutional neural network for programming language processing	✓							
Convolutional Neural Networks over Tree Structures for Programming Language Processing	✓							
DeepTLE: Learning Code-Level Features to Predict Code Performance before It Runs							✓	
Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks						✓		

C.4 Code-Text and Related Papers

Title	Identifier generation	Code summarisation
code2vec: Learning Distributed Representations of Code	✓	
code2seq: Generating Sequences from Structured Representations of Code		✓
A Transformer-based Approach for Source Code Summarization		✓
Self-Supervised Contrastive Learning for Code Retrieval and Summarization via Semantic-Preserving Transformations		✓
Learning to Represent Programs with Graphs	✓	
A general path-based representation for predicting program properties	✓	
Open Vocabulary Learning on Source Code with a Graph-Structured Cache	✓	
Retrieval-Augmented Generation for Code Summarization via Hybrid GNN		✓
STRUCTURED NEURAL SUMMARIZATION	✓	✓
Improved Automatic Summarization of Subroutines via Attention to File Context		✓
Assessing the Generalizability of code2vec Token Embeddings		✓
Learning to Spot and Refactor Inconsistent Method Names	✓	
Deep Code Comment Generation		✓
Fret: Functional Reinforced Transformer With BERT for Code Summarization		✓
SIT3: Code Summarization with Structure-Induced Transformer		✓
Summarizing Source Code using a Neural Attention Model		✓
Blended, precise semantic program embeddings	✓	
Code Generation as a Dual Task of Code Summarization		✓
Retrieval-based Neural Source Code Summarization		✓
A Convolutional Attention Network for Extreme Summarization of Source Code		✓
Improving Automatic Source Code Summarization via Deep Reinforcement Learning		✓

C.5 Text-Code and Related Papers

Title	Program synthesis	Code search
Self-Supervised Contrastive Learning for Code Retrieval and Summarization via Semantic-Preserving Transformations		✓
When Deep Learning Met Code Search		✓
Code-to-Code Search Based on Deep Neural Network and Code Mutation		✓
Deep Code Search		✓
Improving Code Search with Co-Attentive Representation Learning		✓
Code Generation as a Dual Task of Code Summarization	✓	