Deep Learning Representations of Programs: A Systematic Literature Review

DEEPIKA SHANMUGASUNDARAM, University at Buffalo, Buffalo, United States
PALLAVI ARIVUKKARASU, University at Buffalo, Buffalo, United States
HUAMING CHEN, The University of Sydney - Camperdown and Darlington Campus, Sydney, Australia
HAIPENG CAI, University at Buffalo, Buffalo, United States

In the contemporary era, deep learning (DL) is increasingly recognized as a promising approach for enabling and optimizing various techniques, notably in the domain of *DL for code* (software programs). In essence, deep learning is mainly representation learning, which naturally holds for this domain. Thus, at the core of DL for code is deep representation learning for programs. The learned program representations can then be applied to various coding-related tasks, such as detecting vulnerabilities, providing recommendations for API usage, and extracting semantic and syntactic insights from extensive code lines. This is achieved by harnessing deep neural network *architectures* and deep-learning *algorithms* that take programs as *inputs*, serving various software engineering *applications*.

In this article, we conduct a systematic literature search to review studies pertaining to the representation of programs using deep learning approaches and their corresponding applications. Our search yielded 178 primary studies published between 2017 and 2023. Through these studies in the latest literature, we provide a systematization of knowledge in deep learning representation of programs, concerning the *raw inputs* to the learning pipeline, *neural network architecture* employed, learning algorithm utilized, and downstream tasks (i.e., *applications*) of the learned representations. While examining the current landscape, we also identify limitations and challenges faced in the state-of-the-art, as well as promising future research directions in deep program representation learning.

CCS Concepts: • **Computing methodologies → Learning latent representations**; • **Theory of computation → Program analysis**; • **General and reference → Surveys and overviews;**

Additional Key Words and Phrases: Program representation, deep learning, input, algorithms, neural network architecture, application

ACM Reference Format:
Deepika Shanmugasundaram, Pallavi Arivukkarasu, Huaming Chen, and Haipeng Cai. 2025. Deep Learning Representations of Programs: A Systematic Literature Review. *ACM Comput. Surv.* 58, 5, Article 127 (November 2025), 37 pages. https://doi.org/10.1145/3769008

We thank the reviewers for insightful and constructive comments. This work was supported in part by the U.S. Office of Naval Research (ONR) under Grant N000142212111 and N000142512252, Army Research Office (ARO) through Grant W911NF-21-1-0027, and National Science Foundation (NSF) under Grant No. CCF-2146233 and CCF-2505223.
Authors' Contact Information: Deepika Shanmugasundaram, University at Buffalo, Buffalo, New York, United States; e-mail: deepika.s25999@gmail.com; Pallavi Arivukkarasu, University at Buffalo, Buffalo, New York, United States; e-mail: pallavi.arivukkarasu@gmail.com; Huaming Chen, The University of Sydney—Camperdown and Darlington Campus, Sydney, New SouthWales, Australia; e-mail: huaming.chen@sydney.edu.au; Haipeng Cai (corresponding author), University at Buffalo, Buffalo, New York, United States; e-mail: haipengc@buffalo.edu.

ACM 0360-0300/2025/11-ART127
https://doi.org/10.1145/3769008

1 Introduction

Deep Learning (**DL**) has emerged as a transformative force in various technological domains, notably in **software engineering** (**SE**) where they enhance program understanding and analysis. The significance of DL lies in its ability to automate and refine processes such as vulnerability detection, code summarization, performance optimization, and automated documentation, among other SE tasks. By leveraging extensive datasets, DL models can learn to uncover intricate patterns within complex program structures, offering invaluable tools for developers and researchers.

Various DL architectures have been widely applied to a range of SE problems, demonstrating their efficacy and versatility. For instance, **Convolutional Neural Networks** (**CNNs**) and **Recurrent Neural Networks** (**RNNs**) have been successfully used for feature extraction in code representation and anomaly detection in software patterns. Similarly, **Graph Neural Networks** (**GNNs**) have shown promise in understanding and representing the non-linear relationships in program syntax and logic. The merits of DL models include their ability to learn hierarchical representations and generalize to unseen data, making them indispensable tools for advancing the analysis, management, and improvement of modern software systems.

At the core of DL's success in SE, including software security, is *representation learning*—a pivotal step where raw program data is transformed into a format that DL models can efficiently process. The effectiveness of DL applications in SE hinges on how well these representations capture the underlying structure and semantics of programs in a comprehensive way. With precise and robust program representations, DL models can effectively address a wide array of application tasks, from predicting software defects to automating code documentation.

Despite the prevalent applications of DL in SE and significant advances made in this area, there remains a gap in dedicated, comprehensive surveys focusing on how programs are represented for DL applications. While several surveys have touched upon various aspects of DL in SE, they often concentrate on specific applications or DL architectures without a thorough examination of *program representation learning*. For instance, the most relevant prior surveys address the SE tasks and process phases served by DL as well datasets and evaluation methods, covering each briefly in a nutshell [231], focus exclusively on generative tasks and particular model architecture (e.g., probabilistic modeling [4], encoder-decoder [96] architectures), or examine only a specific application task (e.g., code clone detection [3], malware detection [164], vulnerability detection [121, 150], and defect prediction [3]). However, since the essence of DL lies in representation learning [14] and one type of learned representation can serve across a variety of application tasks, including in program representations in SE, looking at the downstream application tasks only does not sufficiently capture the fundamental methodologies that underpin them. Moreover, given the fast-evolving nature of the field of DL and its application in SE, existing relevant surveys, which cover up to five years ago or even earlier, also need to be updated to reflect the current state of the field.

In this article, we present the first comprehensive and up-to-date survey that crosses the boundaries of specific applications or DL architectures. To systematically curate and critically analyze the existing literature, a principled approach for data collection is employed in this work for comprehensive coverage. Utilizing a combination of keyword searches on major academic platforms like Google Scholar and methodical snowballing techniques, we have curated a robust collection of relevant studies. This meticulous process ensures that our survey encapsulates the most significant and impactful research in the domain.

This survey synthesizes the diverse approaches and methodologies employed in the representation of programs using DL, offering a granular understanding of the field's current state and its future directions. By discussing the strengths and limitations of different DL methods and their applications in SE, we provide a valuable resource that can guide future research and practice in this evolving field.

Our comprehensive survey of DL-based representations of programs reveals several key insights across different aspects of the field. Firstly, regarding the **input** to representation learning, there is a strong preference for syntactic inputs, such as Abstract Syntax Trees, used in 70% of the studies, highlighting the crucial role of structural data in program analysis. Text inputs also remain foundational, underscoring their importance in initial representation stages. In terms of **model architectures** used for program representation learning, feed-forward networks, particularly CNNs, are predominant, favored for their effectiveness in interpreting structural patterns in code. We also observed an increasing adoption of graph-based models such as GNNs, which effectively capture complex relationships within program structures. Concerning program representation **learning algorithms**, we found a significant focus on classification (48% of articles) and optimization algorithms (43%), reflecting a strong preference for tasks emphasizing prediction accuracy and performance enhancement. However, regression algorithms are less common, suggesting a more limited application of DL in the respective SE domain. Finally, when it comes to the **applications**, the primary uses of the learned program representations include bug detection and code quality assessment, which benefit from the predictive capabilities of classification and optimization algorithms. In contrast, applications in more complex tasks such as code generation and documentation are less explored, presenting potential areas for future research.

In sum, our survey enhances the body of knowledge in deep program representation learning by offering the following key contributions.

— We conducted an extensive, systematic, and up-to-date literature survey on DL methods applied to representing programs. This survey integrates a wide array of sources to ensure a detailed and thorough review of the field. We applied meticulous approaches to data collection, analysis, and systematization.
— We derived a detailed taxonomy that classifies the diverse approaches to deep program representation learning. This taxonomy not only delineates the types of model inputs and architectures but also extends to the learning algorithms used and their applications. Our taxonomy provides a structured framework to understand and analyze the depth and variety within the field of program representation learning. We then revealed both quantitative and qualitative findings in this field by mapping state-of-the-art studies to the taxonomy.
— We identified and discussed current challenges and limitations within the field of DL-based program representation. We provide a critical analysis of existing methodologies and highlight gaps that suggest avenues for future research, offering a roadmap for future work that could lead to significant advancements in the practical use of DL for program analysis and other SE tasks.

2 Background

To facilitate understanding of the field of program presentation learning, we first provide basic background on raw program representations. Then, we overview popular DL methods capable of learning program representations.

2.1 Raw Representations of Programs

Learning a representation of programs via DL necessitates structured and interpretable forms of input data. These inputs are derived from the raw representations of programs, each tailored to capture different facets of program information, which are crucial for the learning process. Common raw representations of programs used as input to DL-based representation learning models include:

— Code Tokens: This fundamental representation involves decomposing a program into its atomic elements or tokens. These tokens include keywords, identifiers, operators, punctuation, and whitespace. Tokenization transforms the program code into a sequence of tokens that DL models can process, serving as the basis for tasks ranging from syntax checking to more complex analyses like style violation detection.
— **Abstract Syntax Tree** (**AST**): An AST breaks down the syntactic structure of code into a tree-like format, where each node represents a programming construct such as expressions, statements, or declarations. ASTs are pivotal for tasks that require an understanding of the hierarchical structure of code, such as static code analysis, automated refactoring, and more sophisticated semantic analyses.
— **Control Flow Graph** (**CFG**): CFGs are graphical representations of all paths that might be traversed through a program during its execution. Each node in a CFG represents a block of atomic instructions, and edges define the control flow between these blocks. CFGs are essential for understanding the dynamic behavior of programs, including identifying potential points of failure, understanding loop dynamics, and performing security audits.
— **Call Graph** (**CG**): These graphs model the calling relationships between different routines in a program. Nodes in a CG represent functions or methods, and edges indicate that one function calls another. CGs are crucial for enabling data/control flow understanding across functions.

From these basic raw representations, more sophisticated ones, such as **data flow graph** (**DFG**) and **control/data/program dependency graph** (**CDG/DDG/PDG**), can be constructed via program analysis techniques. These advanced raw representations are sometimes also used as direct inputs to program representation learning.

2.2 Major DL Methods for Program Analysis

The evolution of DL has brought forth a variety of models and algorithms that are effectively employed for the analysis and representation of programs. Some of the major DL methods include:

— CNNs: Primarily known for their application in image processing, CNNs have also been adapted to analyze structural patterns in code. In program analysis, CNNs can process ASTs, token sequences, and even CFGs to detect patterns and anomalies in the code's structure, aiding in tasks such as bug detection and code quality assessment.
— RNNs: RNNs are adept at handling data sequences, making them ideal for processing sequences of code tokens or AST nodes. Their capability to maintain a state or memory of previous inputs allows them to perform well in tasks that require an understanding of the entire or large parts of the code sequence, such as code generation and feature extraction for code summarization.
— **Long Short-Term Memory Networks** (**LSTMs**): A sophisticated extension of RNNs, LSTMs are designed to overcome the problem of long-term dependencies by incorporating mechanisms that regulate the flow of information. This ability makes them particularly useful in programming contexts where the understanding of long-range dependencies within the code is crucial, such as in the detection of complex software bugs that manifest over multiple functions or modules.
— GNNs: As code can naturally be represented as graphs (such as CFGs and PDGs), GNNs are poised to analyze these structures. GNNs apply neural network transformations directly to graphs, allowing them to learn intricate representations of code that encapsulate both local interactions (e.g., relationships between consecutive code lines) and broader structural patterns (e.g., overall architecture of a software system).

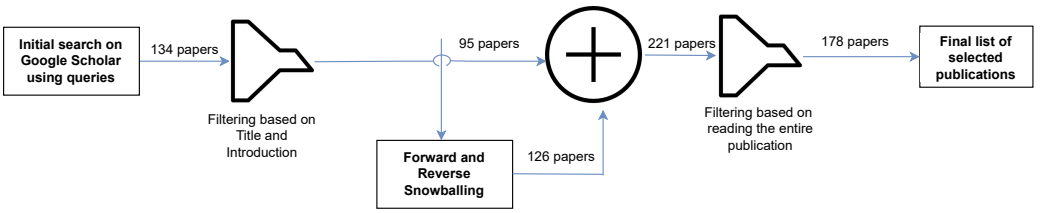


Fig. 1. Overview of our systematic literature search process.

— Transformer Models: Building on the concept of attention mechanisms, transformers represent a significant leap forward, particularly in handling sequences. By focusing on different parts of the input data at different times, transformers can model relationships in data that are widely separated in sequence—a common scenario in programs. They are increasingly being used for tasks such as code translation, where understanding the context across large sections of code is essential, and generating documentation automatically from codebases.

3 Methodology

To explore the effectiveness and range of DL approaches in program representation learning, we adopted a **Systematic Literature Review** (**SLR**) strategy. It starts with (1) literature search, followed by (2) data extraction and analysis towards categorization (to derive the taxonomy), and ending with (3) critical review and gap identification. This principled approach offers comprehensive coverage, objectivity, and transparency by adhering to a predefined protocol for selecting and evaluating studies. It ensures high-quality evidence synthesis, identifies research gaps, and provides a rigorous, unbiased overview of the literature. Next, we elaborate each of these four steps of SLR for our survey.

3.1 Literature Search

Figure 1 depicts the overview of our first step—literature search, including four substeps as described below.

3.1.1 Initial Search. As illustrated, our literature search began with a structured querying process on Google Scholar to compile an initial list of potentially relevant studies from various academic digital libraries such as IEEE Xplore, ACM, and ArXiv. The search queries are based on terms (keywords) intuitively relevant to our study, including "representation learning," "program representation," "deep learning in program," and "code representation." This initial search ended up with **134** articles identified. Despite the high relevance of these search keywords to our study topic, a refined search was necessary as evidenced by our subsequent results: some of these initially identified studies are not really relevant, as they do not directly address program representation learning.

3.1.2 Filtering. To focus our study on the chosen topic with respect to the study goal, we applied specific filters to refine the search results further. These filters are defined through inclusion and exclusion criteria outlined below.

— Inclusion Criteria: Publications from January 2017 to December 2023 were included to focus on the most recent advancements in DL for program representation. Full-length articles consisting of at least six pages were primarily considered. We chose 2017 as the starting year because our preliminary, broader-range study revealed that very few relevant works exist prior to that year that are actually relevant to our studied topic.

— Exclusion Criteria: Short articles, abstracts, posters, and non-English publications were excluded. Studies exclusively available on ArXiv were also omitted to ensure a focus on peer-reviewed studies.

Moreover, we checked the relevancy of each article as per its title and introduction section. Applying these filters left us with **95** articles—the other 39 were excluded per the inclusion/exclusion criteria or because the content (introduction section in particular) revealed that they were not really relevant.

3.1.3 Snowballing Approach. Following the initial search and filtering we employed both forward and reverse (backward) snowballing techniques. This method helped identify additional relevant studies either cited in the references of the initially selected articles or citing them. Note that during this process, for each article identified via snowballing, we adopted the same filtering approach as described above to decide on its relevancy hence inclusion. In the end, **126** new articles were added to our list through this snowballing process, resulting in a total of **221** articles for final confirmation.

3.1.4 Further Inspection. Upon carefully analyzing the entire content of each article we collected, we determined whether the article is actually relevant to the topic of DL representation of programs. Specifically, in addition to the title, abstract, and introduction section of the article, we also read through the rest of the article, making sure the content consisted of a DL approach and a neural network that performed the representation learning on programs. If it passed this further inspection, it was added to our pool of article collection, culminating in a final pool of **178** articles.

3.2 Data Extraction and Analysis for Categorization

Upon the completion of the literature collection, a meticulous process of data extraction and analysis commenced. This involved delving into each article to gather detailed information on four key aspects, which included:

— **Raw Input**: Types of data fed into the **deep neural network** (**DNN**) used for learning program representations. We found that the raw inputs to the representation learning pipeline in our surveyed studies include text input (i.e., treating program as text tokens), syntactic input (syntactic forms of programs such as ASTs), semantic input (semantic forms of programs such as CFGs and PDGs), and hybrid input (combining more than one of the other types of inputs).

— **Neural Network Architecture**: Configurations and structures of the neural networks used. At a high level, these networks can be categorized into two categories: feed-forward (e.g., single-/multi-layer perceptron, CNNs, **generative adversarial networks** (**GANs**), **autoencoders** (**AE**), Transformers) and feed-backward (i.e., feedback/recurrent) networks (e.g., LSTM, **gated recurrent units** (**GRUs**), **neural turing machines** (**NTMs**), universal Transformers).

— **Learning Algorithm**: The machine learning algorithm used for learning the program representations. Generally, this includes supervised, unsupervised learning, and reinforcement learning. For learning the representation of programs, our studies covered supervised and unsupervised learning algorithms.

— **Application**: This concerns the real-world utility of the resulting, deep-learned representation of programs in solving practical tasks, including detection-based applications, **information retrieval** (**IR**), and code generation, as revealed in the surveyed studies.

Figure 2 gives our categorization/taxonomy of deep program representation learning at a high level, which addresses all of the four common aspects of representation learning in general: raw input, model architecture, learning algorithm, and application. This structured approach ensures a

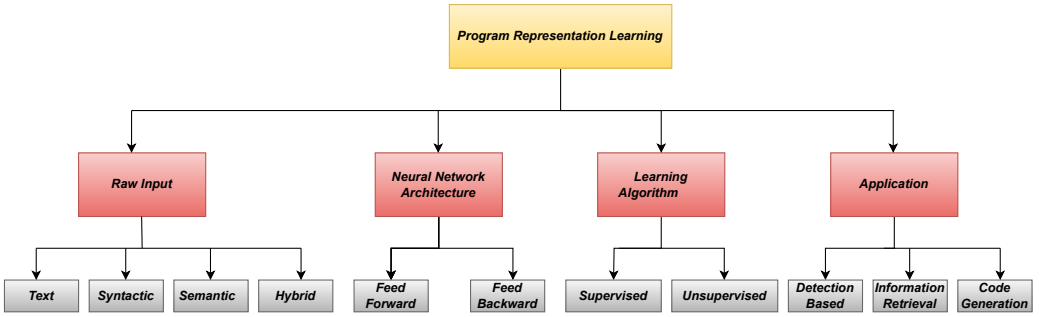


Fig. 2. High-level taxonomy of deep program representation learning.

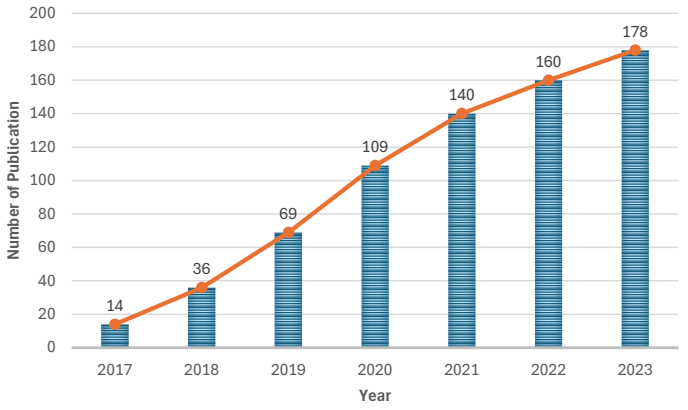


Fig. 3. Overall trend of research published on deep program representation learning.

comprehensive understanding of how DL technologies are employed in various program representation learning.

3.3 Critical Review and Gap Identification

While a major purpose of this survey is to systematize the current state of knowledge in program representation learning, we also mindfully identify limitations/gaps and areas for improvement in the current relevant literature through critical analyses. This motivation is that such a critical review can lead to envisioning future research directions and strategies beyond mere knowledge systematization.

4 Survey Results

In this section, we present the main findings from our survey. We start with an overview of the entire field of deep program representation learning in terms of the trend of research work published during the focused period of time. Then, we systematize the surveyed studies in each of the four aspects of our survey according to our derived taxonomy (Figure 2).

4.1 Publication Trends

We first look at the trends in publications related to DL techniques for program representation from 2017 to 2023, as depicted in Figure 3. Over the years, there has been a continuous and steady growth in the aggregate attention of researchers to program representation learning, as clearly

Table 1. Categorization of Raw Input to Deep Program Representation Learning

Category	Description and examples
Text Input	Text input usually consists of source code, which is then fed into a lexical analyzer or parser. Performing tokenization on it can further classify the source code as a word, character, sub-word (n-gram), or even sentences [1, 5, 6, 17, 35, 48, 53, 55, 69, 70, 78, 82, 85, 93, 94, 101, 108, 117, 124, 131–133, 142, 144, 168, 176, 178, 183, 185, 186, 202, 215, 221, 233].
Syntactic Input	The syntax of data is its structure described as a data type, independent of any particular representation or encoding. This is particularly used in the representation of text in computer languages, which are generally stored in a tree structure as an AST [7–9, 19, 27, 34, 46, 111, 112, 115, 136, 157, 166, 173, 174, 187, 196, 203, 206, 241, 244].
Semantic Input	Semantic input is often used in semantic analysis, which is the process of gathering necessary semantic information (such as control/data flow/dependence) from the source code [44, 45, 119, 127, 158, 211, 235].
Hybrid Input	A combination of two or more types of input described above [179, 181, 216].

evidenced in the statistics. As noted earlier, there were very few research on this topic published prior to 2017. Ever since, new research in this field has emerged every year.

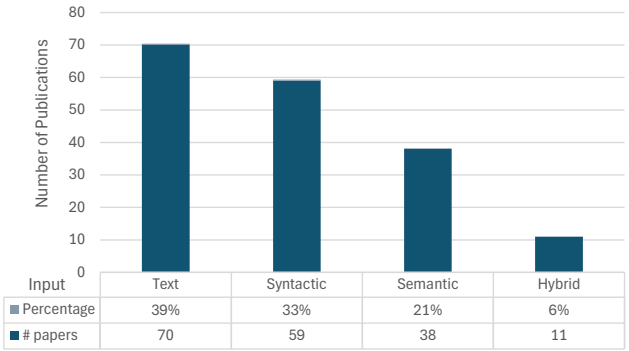
The observed trends highlight significant academic and possibly industrial interest in the application of DL techniques to the representation of programs, reflecting both the advancements in the field and the evolving challenges that researchers are addressing. Looking at such trends, we are confident that deep representation learning on programs has received growing interest and continues to be a very active field of research.

4.2 Raw Input to the Neural Network

Raw data is described as data exactly as it was collected before pre-processing it or cleaning it. In machine learning, the very first step is to acquire the relevant raw dataset to be pre-processed and fed into the neural network. Raw data may include mismatch of data labels, outliers, missing data or mismatch in data dimensions, making it necessary that the data be pre-processed. Pre-processing the data helps remove duplicates or format the dataset in an order it must be fed into the neural network. On the other hand, the raw data is essential for representation learning as it is the source of information the neural networks learn the representation from. For program representation learning, the raw data is the input program in various forms, for which our survey revealed four categories: *Text*, *Syntactic*, *Semantic*, and *Hybrid* inputs, as explained and exemplified in Table 1.

Figure 4 illustrates the distribution of the surveyed studies across different types of input data used in DL models for program representation, based on our systematic literature review conducted. Most publications, 70 in total, utilize *Text* input, indicating a strong preference for this type of data in the studies reviewed. Following closely is the *Syntactic* input with 59 publications, suggesting significant reliance on the structural aspects of program data. *Semantic* input, which focuses on the meaning (code semantics) and implications of the data, is used in 38 publications. Notably, *Hybrid* input, which combines multiple types of data, has the least presence with only 11 publications. This distribution highlights the varying methodologies and focuses in the field of program representation learning, each catering to different aspects of understanding and analyzing software programs through DL.

4.2.1 Text Input. A straightforward way of learning program representations is to treat programs as texts. Tokenization, a common task in DL, involves breaking down a piece of text into smaller units known as tokens. Tokenization serves as the initial step in modeling text data. It involves processing



Input	Text	Syntactic	Semantic	Hybrid
Percentage	39%	33%	21%	6%
# papers	70	59	38	11

Fig. 4. Distribution of surveyed studies over different categories of raw input to program representation learning.

the corpus to extract tokens, which are then used to construct a vocabulary set. This vocabulary comprises the unique tokens present in the corpus. These tokens can represent **words**, **characters**, or **sub-words**, corresponding to three primary types of tokenization schemes: word, character, and sub-word (n-gram characters) tokenization. Our survey shows that existing deep program representation learning commonly incorporated tokenization as part of the input (pre)processing step.

The majority of surveyed studies (e.g., [40, 54, 64, 69, 85, 131, 144, 178]) used **word** tokenization specifically involves dividing text into words, essential for machine learning tasks like sentiment analysis, where individual words are analyzed, classified, and counted for sentiment purposes. For example, Chen et al.[35] used the `python-sqlparse` library for SQL code tokenization and modified version of an `ANTLR` parser for C tokenization. Allamanis et al. [6] performed analysis and processing at the level of individual words, which facilitates comprehension and subsequent tasks such as information extraction and semantic analysis. Lun et al. [81, 185] tokenized the code commit sequences with white space and punctuation, removing non-informative tokens and filtering out poorly written commit messages.

Character-based tokenizers break down raw text into individual characters. This approach is based on the premise that while languages may contain a vast array of words, they comprise only a finite set of characters, leading to a minimal vocabulary. This straightforward method significantly reduces memory and time complexity. Qing et al. [78] illustrate this with the CDRL model, which tokenizes method names, API sequences, and code tokens by parsing camel case and treating API method invocations as separate entities. Similarly, descriptions extracted from JavaDoc comments are tokenized, with additional steps like removing non-alphanumeric characters to simplify the text further. This efficient breakdown into tokens facilitates deeper analysis and processing. Key studies such as those in [6, 52, 76, 77] have utilized this approach, with additional applications highlighted in [5, 78, 82, 132, 133].

Subword-based tokenization offers a middle ground between character and word-based methods, aiming to address the limitations of both. It reduces the semantic ambiguity of character-based tokenization and the large vocabulary size of word-based tokenization, which often results in many out-of-vocabulary tokens. Subword tokenization adheres to two main principles: (1) frequently used words are not split into subwords and (2) rare words are divided into smaller, meaningful subwords. This method has been employed in various studies, including those in [1, 17, 48, 55, 93, 101, 117, 124, 142, 168, 176, 183, 202, 215, 221].

N-gram tokenization, as Zhen et al. [116] describe, involves segmenting text into contiguous sequences of n items from characters to words. For example, the sentence “The quick brown fox”

could be divided into sequences like "The quick," "quick brown," and "brown fox." This technique is particularly useful for language modeling and text prediction, where understanding the context provided by neighboring words is essential. Shuai et al. [176] apply this method to segment code snippets into n-grams of method names, API sequences, and tokens, capturing essential contextual information for analysis and modeling. This technique is also favored in studies by [1, 16, 203, 225] for its ability to enhance text understanding.

4.2.2 Syntactic Input. Compared to treating programs as text input to DL models, a more program-specific representation learning would at least consider the syntax of the programming language used in developing the program. The most common syntactic input to deep program representation learning as found in our survey is AST (Section 2). Alon et al. [7] utilize the AST as a structured representation of code in their research, which captures the hierarchical and syntactic relationships between code components, organizing them into a tree-like structure. Each node represents a language construct, such as a function, loop, or conditional statement, with edges illustrating the relationships between these constructs. Leveraging the AST allows their `code2seq` model to encode the syntactic information of the code effectively, enabling it to generate meaningful sequences of tokens that reflect the code's structural and syntactic properties. Wenyan et al. [9] utilized a compiler to parse source code, generating node information and forming an AST. This approach provided a detailed representation of code syntax, significantly enhancing the accuracy of vulnerability detection within their framework. Similarly, other studies [22, 24, 110] employ compilers that parse through the source code to generate node information and thus form the AST structure.

In [18], the input source code is represented in an AST format, which is justified by a promising balance between precision and recall metrics. The authors utilize AST node types, identifiers, and literal values to create node representations, enhancing the model's effectiveness in interpreting and analyzing code.

Dinella et al. [51] parse a program's source code into an AST that captures its syntactic structure. This AST is further enhanced by connecting leaf nodes with "SuccToken" edges and introducing "value nodes" that store the actual content of these leaves. These value nodes are interconnected with unique "ValueLink" edges, establishing a strategy for code representation that is independent of variable names. The graph encapsulates the code in a structured form suitable for analysis. This graph structure is then processed using a GNN, which maps it into a fixed-dimensional vector space. This transformation facilitates the learning of a sequence of graph transformations aimed at identifying and correcting bugs in the code. The model predicts the positions of bug nodes and suggests necessary edits to fix them. Implemented in the tool Hoppity, this approach has proven effective, correctly predicting 9,490 out of 36,361 code changes in real programs on GitHub, demonstrating the utility of AST-based graphs in automated code repair.

4.2.3 Semantic Input. Semantic input is often used in semantic representation learning, which involves gathering necessary semantic information from the source code, often represented graphically. One common semantic type of input is the CG (Section 2). Zugner et al. [247] utilize CGs, constructed from static analysis of Android apps, to model method calling relationships. Attributes like API calls and permissions are assigned as features to graph nodes. These enriched CGs are processed through a **Graph Convolutional Network** (**GCN**) to learn detailed representations. The learned embeddings from the GCN help classify apps as benign or malicious, enhancing detection accuracy by capturing complex interactions within applications. This approach allows for nuanced malware detection beyond traditional static analysis methods.

Phan et al. [158] generate CFGs from assembly code snippets, with each vertex representing an instruction and directed edges indicating the sequence of instructions. An algorithm constructs a

CFG from an assembly file by processing the assembly code. Conversely, Yu et al. [235] take the CFG as input, building an end-to-end model that trains the node embeddings together rather than using pre-trained block embeddings. The final CFG representation is computed using Set2Set [191] after multiple message passes between adjacent nodes using a **Gated Graph Neural Network** (**GGNN**). Another approach, GINN [211], uses CFGs to represent programs, applying heightening operators to increase graph order and then restoring local propagation to recover the original CFG node states. GINN captures the global properties of a graph and computes precise node representations.

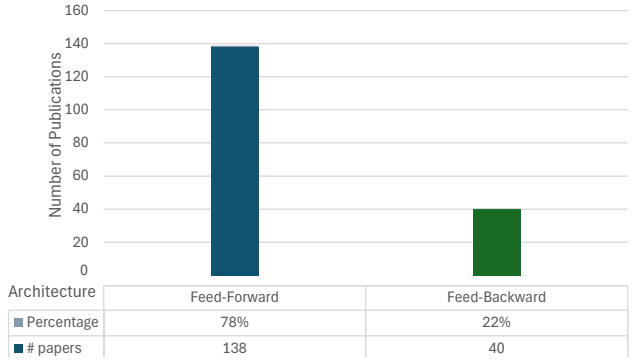
Cornaglia et al. [44], the binary CFG is reconstructed by processing traces step by step, with hardware-dependent information extraction generating dictionaries that map between **intermediate representations** (**IRs**) and binary CFGs. The study in [45] divides vulnerabilities, patches, and testing programs into functions, each transformed into a CFG. These CFGs are then sliced into smaller subgraphs based on vulnerability-sensitive keywords, retaining only nodes and statements with control and data dependencies related to these keywords. The study in [127] uses multiple raw input types for neural networks, with CFGs displaying statement execution orders and conditions for these executions.

A **data-flow graph** (**DFG**) comprises nodes and arcs where nodes represent locations of variable assignments or usage, and arcs illustrate the relationship between these assignments and their subsequent use. In DFGs, actors, representing operations or functions, are enabled when data objects are available at input ports and conditions are met, allowing for parallel or sequential firing. Cummins et al. [46] employ DFGs in their PROGRAML framework to capture control, data, and call relations, enhancing machine learning-based program analysis through message-passing neural networks. Gao et al. [59] developed a model that uses DFGs to capture global semantic structure and guides the attention mechanism hierarchically, enhancing the representation of long-range dependencies in the code. SG-Trans significantly outperforms existing models on benchmark datasets. The use of DFGs allows the model to effectively capture data dependencies and improve summarization accuracy.

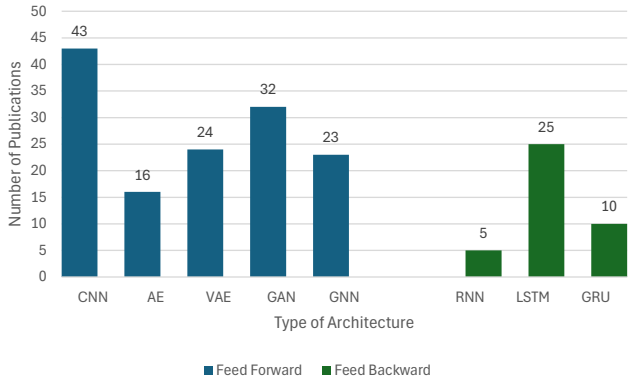
The **program dependence graph** (**PDG**) explicitly represents data and control dependencies within code. Li et al. [118] proposed a framework that uses PDGs to help in the process of transforming **Syntax-based Vulnerability Candidates** (**SyVCs**) into **Semantics-based Vulnerability Candidates** (**SeVCs**). The SeVCs not only consider the syntax of potential vulnerability points but also the semantic information, which includes the dependencies extracted from PDGs. This enables the framework to more comprehensively understand the context and potential impact of each vulnerability, leading to more effective detection. Zhang et al. [237] utilize PDGs to learn semantic structures in code, asserting that PDGs capture code semantics more effectively than ASTs by characterizing the execution order and data interactions within programs.

4.2.4 Hybrid Input. Several studies have adopted a cumulative or hybrid approach to input data representation to enhance code analysis and comprehension. These approaches integrate AST with other representations like CFG or PDG. This combination of raw inputs facilitates deeper learning of code structure and dependencies, leading to a more comprehensive understanding and analysis of programs.

A **Code Property Graph** (**CPG**) is an innovative data structure that integrates various program representations, including ASTs, CFGs, and PDGs. This unified structure enables sophisticated analysis and querying capabilities, providing a comprehensive view of a program's semantics, structure, and flow, which is particularly useful in security vulnerability detection and software optimization. Various prior studies leverage CPG [52, 181, 219] to deep semantic representations of programs. For instance, Suneja et al. [181] combine ASTs, CFGs, and PDGs as hybrid input to the DL model, capturing both syntactic and semantic information from the source code. The integration of



(a) Distribution over high-level categories of architectures.



(b) Distribution over specific model architectures.

Fig. 5. Distribution of surveyed studies over different (deep) neural network architectures.

these graphical representations enhances the effectiveness of vulnerability detection by providing a comprehensive understanding of code structure and behavior.

Cummins et al. [46] obtain semantic inputs from compiler's IR of a program in three stages: control flow, data flow, and call flow. This method involves adding control flow edges to depict execution paths, data-flow edges to capture relationships between constants and variables, and call edges to represent the interactions between calling statements and called functions. The deep representation of programs is then learned from this semantic input to enable program analysis and optimizations.

4.3 Neural Network Architecture

DNNs are the backbone of deep representation learning. The "deep" in both terms refers to the multiple layers of transformation that enable complex feature learning. DNNs learn representations in a hierarchical way, meaning the network learns progressively abstract features at each layer. Thus, the architecture of DNNs intuitively plays a key role in how effectively deep representations of programs can be learned. Figure 5(a) illustrates the use of two high-level categories of DNN architectures across reviewed publications, as explained and exemplified in Table 2. Feed-forward architectures are more prevalent, with 138 publications employing this type, underscoring its widespread adoption in the field. Conversely, feed-backward architectures, which include recurrent

Table 2. High-level Categorization of Neural Network Architectures Used in Deep Program Representation Learning

Category	Description and examples
Feed-forward	Feed-forward neural networks allow signals to travel in one direction only, from input to output. There is no feedback (loops), meaning the output of any layer does not affect that same layer. These networks are relatively simple and are extensively used in pattern recognition (e.g., [6, 35, 69, 70, 78, 85, 93, 108, 133, 142, 168, 176, 185])
Feed-backward	In these neural network models, signals traverse in both directions through the loops (hidden layers) in the network. This type is widely applied in signal processing and optimal computation. A traditional feedback neural network model generally has time-invariant inputs (e.g., [1, 5, 48, 55, 82, 117, 132, 178, 183])

processes, are utilized in 40 publications, highlighting their specific suitability for tasks requiring memory of previous inputs, such as sequence prediction or temporal data analysis. The significant preference for feed-forward architectures may reflect their general applicability and simpler implementation in diverse tasks, from classification to regression, within program representation studies. Figure 5(b) illustrates the further distribution of publications across various specific neural network architectures within the two high-level categories. Among the feed-forward models, the distribution highlights the dominance of CNNs and GANs, followed by **Variational Autoencoders** (**VAEs**) and GNNs. In contrast, LSTMs dominate feed-backward models, where other popular architectures are GRUs and RNNs.

4.3.1 Feed-forward Networks. In this architecture, signals flow exclusively in one direction, towards the output layer. These networks typically comprise an input layer and a single output layer, with the option for zero or multiple hidden layers. Widely employed in pattern recognition tasks, several neural network models utilizing feed-forward architecture include CNN, AE, VAE, GAN, and GNN.

— CNN: CNNs are powerful variants of feed-forward neural networks, typically used in image processing tasks due to their ability to automatically detect and learn important features from input data. It employs layers of convolutional filters to process data through hierarchical layers, enabling it to capture spatial and temporal dependencies in images. For instance, Guo et al. [66] used CNNs for feature extraction from code metrics. Asymmetric convolutional kernels are applied to one-dimensional sequences of code metrics to extract relevant features, which serve as input to the subsequent LSTM network for vulnerability detection. Other studies [86, 161, 219] also adopt CNN architectures to address long-dependency issues and capture semantic information embedded within source code structures.
— AE: The architecture of an autoencoder is designed to compress input data into a shorter code (encoded form) and then reconstruct the output to match the input as closely as possible. This is achieved through two main parts: the encoder, which compresses the input, and the decoder, which reconstructs the input from the compressed form. Zhang et al. [239] employed positional encoding and a Transformer encoder to generate a sequence of vectors for each method, which are then fed into the decoder to predict method names.
— VAE: VAE is a specialized type of autoencoder designed to prevent overfitting through regularization during training. It comprises both encoders and decoders and is trained to minimize the reconstruction error between the original data and the encoded-decoded data. In [68], a VAE is employed to produce latent representations of the source code, which are used to extract semantic features for input into a logistic regression classifier to detect code smells.

— GAN: GANs learn to generate new data samples with similar statistical properties as the training set. Harer et al. [70] utilized GANs to address issues in the automatic repair of source code vulnerabilities, enabling training without paired examples. Sharma et al. [172] introduced **Ensemble GAN** (**EGAN**), a variant of GANs for code readability classification, which employs multiple granularities to encode source code into integer matrices as input for GANs.
— GNN: GNNs are designed for tasks involving graphs, offering capabilities for node-level, edge-level, and graph-level predictions. GNNs have revolutionized the way complex graph-structured data is analyzed by leveraging the power of DL. Various studies [12, 128, 166, 201, 247] have applied GNNs in different contexts. Cummins et al. [46] adopted a **Message Passing Neural Network** (**MPNN**) framework [60] for their system. They focused on learning ProgramL representations of compiler IRs by mapping graph vertices to initial state vectors using embeddings and iteratively updating vertex states through a sequence of message-passing steps. The vertex states are aggregated into either a single graph-level vector or a set of vertex-level vectors.

4.3.2 Feed-backward Neural Networks. In this architecture, recurrent or interactive networks leverage their internal state (memory) to handle sequential input sequences. These networks facilitate bidirectional signal propagation through loops (hidden layers), making them well-suited for time-series and sequential tasks. Examples of neural networks employing this architecture include RNN, LSTM, and GRU.

— RNN: RNNs employ specialized cells called recurrent cells, where each hidden cell receives its own output after a fixed delay of one or more iterations. This architecture has been used in learning program representations in a number of prior studies [18, 139, 219, 233]. For instance, Buch et al. [18] utilized RNNs in a Siamese network setup to identify code clones, normalizing ASTs to binary trees with nodes capable of holding two vectors mapped to values. Additionally, a single LSTM unit was incorporated to average the error gradient.
— LSTM: LSTM networks are an advanced type of RNN designed to learn long-term dependencies in data sequences. The LSTM architecture incorporates a specialized memory cell capable of handling data with temporal gaps or lags, as utilized in several prior program representation learning approaches [66, 185, 225]. For instance, Guo et al. [66] used LSTM networks to capture long-term dependencies and sequential patterns within source code, effectively identifying vulnerabilities by understanding the context and relationships between different parts of the code over extended periods.
— GRU: GRUs extend the LSTM architecture by incorporating gating mechanisms. Cummins et al. [46] employed GRU as part of the ProgramL framework, specifically in the update function during the iterative process of updating vertex states based on learned messages collected from the graph-based network. This enables efficient processing and representation learning of programs within the ProgramL framework for program analysis and optimization purposes. Notably, Choi et al. [42] used GRU as the update function across all experimental setups when learning the neural representations of programs that may have buffer overruns.

A few approaches combine both feed-forward and feed-backward architectures for representation learning. For instance, Recoder [246] utilizes self-attention mechanisms, gating layers, and tree convolutional layers, which are commonly associated with feed-forward networks. It also employs recurrent components such as the AST Reader and Edit Decoder. The AST Reader component, for example, encodes the partial generated AST of the edit, which involves processing a sequence of inputs and utilizing a self-attention layer, gating layer, and tree convolutional layer. This aspect resembles elements of a feed-forward architecture. On the other hand, the Edit Decoder incorporates

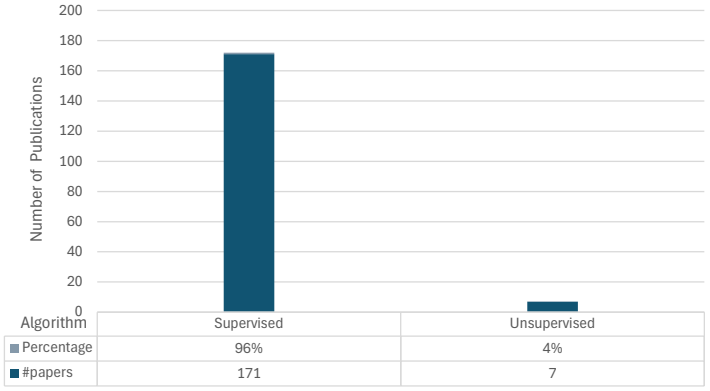


Fig. 6. Distribution of program representation learning algorithms over general categories.

recurrent mechanisms to generate edits based on the encoded information. It utilizes a decider to estimate the probability of using each provider, which involves processing information over multiple steps and making decisions based on previous outputs. This aspect resembles elements of a feed-back architecture. Note that some neural networks, although involving feedback connections (e.g., in an encoder-decoder structure), are still considered feed-forward networks in terms of the overall architecture (e.g., [64, 162]).

4.4 Learning Algorithm

Learning algorithms are crucial to deep representation learning because they determine how the model updates its parameters to extract meaningful features from data. In general, these algorithms can be broadly divided into three main categories: supervised learning, unsupervised learning, and reinforcement learning. As illustrated in Figure 6, among our surveyed works, there is a vast disparity in the number of articles across these general categories: the vast majority (i.e., 96%) of the 178 articles used supervised learning to learn deep representations of programs, program representations learned in unsupervised ways are very few (4%), and reinforcement learning has not been seen for representation learning in our studied literature. This visual comparison starkly highlights the overwhelming preference for supervised learning in current research within the field of program representation learning.

One main justification for this clear preference is that supervised learning generally achieves better performance than unsupervised learning on the same problem and datasets, for several reasons. First, supervised learning algorithms leverage labeled data to directly learn the mapping between inputs and desired outputs, which guides the DL model toward task-specific decision boundaries. Second, the availability of labels reduces ambiguity in interpreting data structure, allowing the DL model to focus on features most relevant to prediction. Finally, evaluation and optimization are explicitly tied to minimizing errors against known ground truth, whereas unsupervised methods rely only on intrinsic patterns or similarity measures that may not align well with the actual task objective. Comparison between these two classes of learning algorithms in our surveyed literature reveals the same observations. For example, Pravilov et al. [162] learn distributed code-change representations based on unsupervised pre-training of a DNN and then apply the learned embeddings for two tasks: code editing and commit message generation. For the first task, the unsupervised learning model only achieves up to 10% accuracy, while for the second task, the unsupervised learning attained a mean BLEU score of 39.3%. In contrast, a supervised learning algorithm for code

Table 3. Categorization of Learning Algorithms in Deep Program Representation Learning

Category	Description and examples
Optimization Algorithms	These are prevalent across all types of learning tasks. They focus on how models adjust their parameters during training to improve performance, whether the learning is supervised or unsupervised [6, 35, 53, 69, 70, 85, 108, 131, 144, 178, 185, 186, 209, 233].
Classification Algorithms	These algorithms handle discrete, categorical predictions, where the output variable is a category. They are primarily used in supervised learning contexts where the goal is to predict discrete labels or categories [1, 17, 48, 55, 72, 93, 101, 117, 124, 139, 142, 168, 176, 183, 202, 215, 221].
Regression Algorithms	These algorithms involve predicting a continuous quantity, mapping input data to a continuous output. They fall under supervised learning and are used for predicting continuous outcomes [68, 86, 161, 219].
Other Learning Algorithms	This category includes unconventional or emerging approaches, such as semi-supervised learning which uses both labeled and unlabeled data for representation learning. These algorithms do not fit neatly into the first three categories but are crucial for specific applications, such as natural language processing tasks associated with coding related tasks [18, 61].

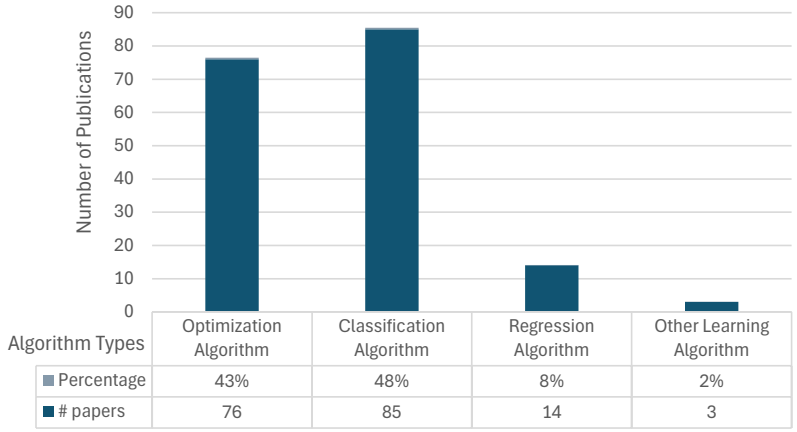


Fig. 7. Distribution of surveyed research across our categorization of learning algorithms.

editing [51] achieved nearly 60% accuracy [148] or even higher [149], although not on the same dataset [25] as for the unsupervised approach, and supervised learning based commit message generation reached almost 50% BLEU score [130]. In fact, this performance advantage of supervised over unsupervised learning algorithms has been demonstrated in traditional machine learning based classifications (e.g., for predictive interprocess communication assessment [57] and quality anomaly detection [58]—98% versus 86% F1 for efficiency anomaly prediction on the same dataset) as well.

On the other hand, given this sharply uneven distribution, we derived a more granular categorization based on the specific objectives of the algorithms in the field of SE: optimization, classification, regression, and other types of learning algorithms. Table 3 summarizes these categories with explanations and example publications in our survey. Compared to the general categorization from the perspectives of how features are extracted and the model (neural network) parameters (weights and biases) are optimized to minimize a loss function, our specific categorization categorizes learning algorithms by the learning objectives/tasks they perform.

As illustrated in Figure 7, a significant majority of the articles used classification algorithms (48%, 85 articles) and optimization algorithms (43%, 76 articles). This indicates a strong preference

for tasks involving the prediction of discrete categories and improving model performance across various tasks. Regression algorithms were less common (8%, 14 articles), reflecting their more limited application in the SE field compared to classification tasks. Finally, various other kinds of learning algorithms accounted for a small fraction (2%, 3 articles) in total, suggesting that traditional learning paradigms dominate the landscape. This distribution underscores a heavy emphasis on applications that benefit from classification and optimization, such as bug detection and code quality assessment, while also highlighting potential areas for further exploration in program representation learning research.

4.4.1 Optimization Algorithms. An optimization algorithm is a method used to identify the optimal solution from a set of feasible solutions by minimizing or maximizing a function. These algorithms play a crucial role in machine learning, particularly in training models. They achieve this by minimizing a loss function, which evaluates the model's performance based on its parameters. The neural network's training process, aimed at enhancing its learning capabilities, involves such optimizations, facilitated by a tool known as the *optimizer*. In machine learning, optimization often pertains to continuous function optimization, where the inputs are real-valued numeric values and the outputs are real-valued evaluations of these inputs.

In particular, tailored for first-order gradient-based optimization of stochastic objective functions, the Adam optimizer utilizes adaptive estimates of lower-order moments to optimize representation learning efficiently. It features straightforward implementation, computational efficiency, minimal memory requirements, and is suitable for large-scale problems with abundant data and parameters. Adam also performs well in scenarios with non-stationary objectives or when dealing with noisy and sparse gradients. Its hyperparameters offer intuitive interpretations and typically require minimal tuning. For instance, Wang et al. [209] utilize a variant of the Adam optimizer, AdamW, with weight decay regularization for training. AdamW is favored for its effectiveness in optimizing large-scale neural networks, to learn program representations underlying a range of code understanding and generation tasks. Lin et al. [120] also use the Adam optimizer to train their patch correctness assessment model that learns from labeled data indicating whether a particular patch introduces a bug or not, allowing it to predict the likelihood of future patches being buggy based on their code characteristics. This optimizer was more recently used to learn program representations that differentiate vulnerable versus non-vulnerable code [66].

4.4.2 Classification Algorithms. Classification algorithms are designed to make precise predictions about class labels by automatically extracting relevant features from input data. As input signals are received and processed through activation functions, each neuron transmits its output to the subsequent layer.

As an example, Hin et al. [72] describe LineVD, a classification task designed to categorize code instances into two classes: vulnerable and non-vulnerable. The goal is to predict the class label (vulnerability or non-vulnerability) for each input code snippet based on its features. During training, LineVD learns a mapping from input features, representing the source code, to discrete output labels indicating the presence or absence of vulnerabilities. This mapping is honed using optimization techniques such as gradient descent, where the model's parameters are iteratively adjusted to minimize the difference between predicted and actual labels.

Another example, Mou et al. [139] proposes **tree-based CNN** (**TBCNN**), a model that uses convolution for programming language processing. TBCNN classifies programming code by functionality and detects specific code patterns. It outperforms traditional models by leveraging the structural details inherent in programming languages. TBCNN's effectiveness is demonstrated in classifying programs and detecting patterns like bubble sort within code. This approach enhances automated tasks in SE, such as code review and repository management.

4.4.3 Regression Algorithms. Regression analysis is a fundamental tool in machine learning, used to estimate the relationship between a dependent variable and one or more independent variables. Essentially, it involves fitting a selected function to the sampled data under a specified error function, enabling predictions. Generally, regression algorithms can be divided into two major categories: *linear regression* predicting a dependent variable value (y) based on a given independent variable (x), and *logistic regression* modeling the probability of a binary outcome using the sigmoid function. In our surveyed studies, both kinds of regression algorithms have been employed for representation learning.

For instance, Hadj-Kacem et al. [68] employed logistic regression as the final classification step in their approach to detect code smells. After generating semantic features using a VAE, the logistic regression classifier assesses whether the code contains a code smell based on these features. Results show this regression algorithm was effective for learning features specific to this task: using the Apache datasets, they achieved an average accuracy of 75% and recall of 73%.

4.4.4 Other Learning Algorithms. In addition to the mainstream algorithms previously discussed, several other learning methods are primarily used in **natural language processing** (**NLP**) tasks in support of deep program representation learning. These methods include Word2Vec, **Continuous Bag of Words** (**CBOW**), and Skip-gram, employed in various studies to generate word embeddings which are dense vector representations of words (code tokens) in a continuous vector space. For instance, Büch et al. [18] utilized Word2Vec to learn distributed representations of words based on their co-occurrence patterns. By adjusting word vectors during training, the model captures semantic similarities, which enables its application in code clone detection.

By employing these various other types of learning algorithms, prior works enhance program representation learning by leveraging the rich semantic relationships captured through word embeddings to improve both the understanding and processing of code information within software development contexts.

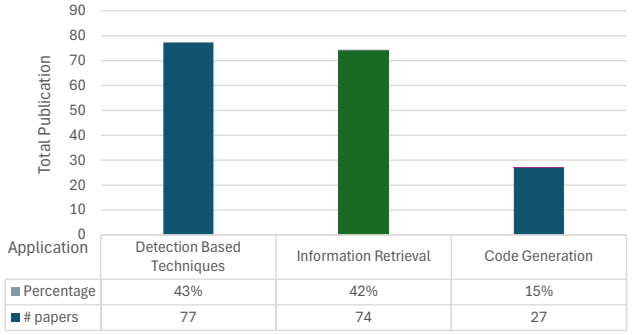
4.5 Applications

DL has profoundly impacted the field of program representation learning, offering advanced solutions across various domains crucial for a variety of SE tasks. This survey highlights key application areas where DL techniques have been effectively integrated.

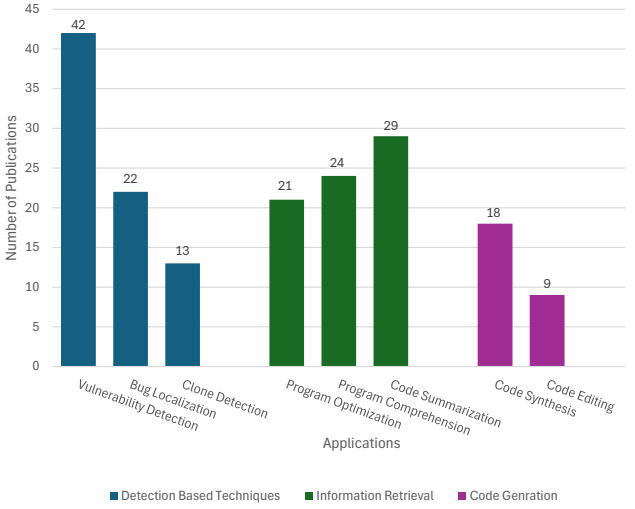
Figure 8(a) illustrates the distribution of the surveyed publications across three high-level application categories: detection-based techniques, IR, and code generation, as explained and exemplified in Table 4. Detection-based techniques are the most prevalent, accounting for 43% of the total publications we examined. IR follows with 42% of the publications. In contrast, code generation represents only 15% of the total, with 27 publications. This disparity underscores the significant focus on detection based techniques within the research community, while code generation remains largely underexplored.

Figure 8(b) details the specific areas covered by the publications categorized by their primary applications of DL-based program representations. The distribution of research focus underscores a predominant interest in detection and security, with growing attention towards improving code management and introducing automation in software development processes. Next, we discuss specific applications in each of these categories.

4.5.1 Vulnerability Detection. DL significantly enhances automated vulnerability detection by leveraging sophisticated representation learning techniques. Studies such as [16, 45, 52, 117–119, 177, 215, 221] use learned program representations to capture and analyze complex code patterns, improving the accuracy and efficiency of vulnerability detection. In [45], vulnerability detection is conducted through a static technique called VulDetector, which utilizes a **weighted feature graph**



(a) Distribution over high-level application categories.



(b) Distribution over detailed categorization of applications.

Fig. 8. Distribution of surveyed studies over different applications of learned program representations.

(**WFG**) model. This model slices and analyzes control flow graphs by focusing on segments of code identified through vulnerability-sensitive keywords. It enhances detection accuracy by comparing these sliced graphs with known vulnerability graphs and patched versions, using both syntactic and semantic code features. The approach significantly reduces false positives and improves the precision and efficiency of detecting security flaws in C/C++ software programs. Xu et al. [221] introduced a novel approach to detecting vulnerabilities in source code by employing a machine learning model based on LSTM networks contextualized with code attributes. This method enhances the accuracy of vulnerability detection by incorporating contextual information about the code, such as data flow and control structures, alongside traditional syntactic features. This approach allows for a more nuanced understanding of code, leading to better identification of potential security weaknesses in software applications.

4.5.2 Bug Localization. DL methods have been employed to automatically pinpoint potential faults in software using IR-based approaches and DL models [27, 67, 131]. These techniques utilize embeddings that capture the semantic essence of both source code and bug reports, thus enhancing

Table 4. High-level Categorization of the Applications of Deep Program Representation Learning

Category	Description and examples
Detection-based Techniques	These applications utilize the deep representations of programs for various detection purposes, mostly prominently to identify vulnerabilities and bugs in source code [16–18, 27, 45, 47, 51, 52, 54, 66, 67, 69, 77, 115, 117–119, 131, 157, 161, 169, 173, 177, 183, 188, 215, 218, 221]. The majority of publications in category focus on vulnerability detection, with 40+ publications, reflecting a strong emphasis on security within SE. Bug localization and clone detection are also prominent, with about 22 and 13 publications, respectively, emphasizing the importance of code quality and maintenance.
IR	These applications use the learned program representations for SE tasks in areas such as program optimization, program comprehension, and code summarization, each garnering over 20 publications. Broadly speaking, these are techniques for retrieving semantic information from source code and summarizing source code [2, 10, 13, 22, 34, 54, 59, 89, 125, 127, 182, 194, 196, 202, 203, 206, 211, 239, 245].
Code Generation	Although with a lesser presence, this category covers emerging areas such as code synthesis and editing, with about 18 and 9 publications, respectively, both involving generating code at the core [81, 94, 98, 209].

the accuracy and efficiency of bug localization. BugPecker [27] can be integrated into the software development lifecycle to assist developers in quickly identifying faulty methods based on bug reports. By leveraging DL on revision graphs, BugPecker can pinpoint the exact methods that are likely to be causing issues, thereby reducing the time and effort required for debugging and fixing bugs.

4.5.3 Clone Detection. Clone detection identifies duplicate or similar code blocks, where DL helps distinguish clones based on syntactic and semantic features. Studies such as [54, 77, 204] apply DL to enhance the detection of both syntactic and semantic clones. Fang et al. [54] apply clone detection by combining syntactic and semantic analysis using AST and CFG representations. These features are transformed into fixed-length vectors through Word2Vec [61] and Graph2Vec [140], and then fused together. A DNN is trained on these fused vectors to classify code fragment pairs as clones or non-clones. This method significantly improves the accuracy of detecting functional code clones, aiding in refactoring, bug detection, code reuse, and search.

4.5.4 Program Comprehension. DL aids in understanding the functionality and structure of code, crucial for code review, maintenance, and enhancement. Techniques like [2, 13, 22, 34, 54, 89, 194, 202, 211, 245] utilize DL models to capture rich semantics embedded in source code identifiers, thus improving code comprehensibility and usability. For instance, the Corder framework [22] significantly enhances program comprehension by enabling semantic code retrieval, generating concise code summaries, and producing high-quality vector embeddings. It applies semantic-preserving transformations to show different syntactic forms of the same logic, aiding developers in recognizing varied coding styles. Corder facilitates enhanced code search, automated documentation, and refactoring assistance, improving code understanding and maintenance. It also aids in debugging and learning by providing relevant code examples and explanations. Overall, Corder improves developer productivity and code quality. As another example, Zhang et al. [237] also employ DL models to predict method names, enhancing program comprehension.

4.5.5 Program Optimization. Program optimization uses DL to enhance software systems' performance or reduce resource usage. Baghdadi et al. [10] present a DL-based cost model for automatic code optimization, addressing challenges in modern x86 architectures without heavy feature engineering. Integrated with the TIRAMISU compiler [11], it predicts the performance impact of code

transformations, enabling automatic selection of the best sequences. The model achieves a 16% mean absolute percentage error in predicting speedups and is evaluated using Beam Search and Monte Carlo Tree Search methods. The proposed approach outperforms or matches state-of-the-art compilers, particularly in scientific computing benchmarks. This advancement simplifies and enhances the efficiency of program optimization in modern compilers. In [46], the authors introduce PROGRAML, a graph-based representation for programs that captures control, data, and call relations, aiding in program optimization. Using MPNNs, PROGRAML models effectively perform traditional compiler analysis tasks like control flow reachability, dominator trees, data dependencies, variable liveness, and common subexpression detection. This representation is compiler-independent, supporting LLVM and XLA IRs, and significantly improves optimization tasks. PROGRAML sets new performance benchmarks in heterogeneous device mapping and algorithm classification. It automates heuristic construction from real performance data, replacing fragile, hand-tuned heuristics, and leading to more efficient and adaptive program optimizations.

4.5.6 Code Summarisation. Code summarization benefits from DL by automating the generation of concise natural language descriptions from code snippets. This is treated as a **neural machine translation** (**NMT**) task, where syntactic and semantic representations of code are crucial. Gao et al. [59] propose SG-Trans, a novel approach for code summarization that integrates both local symbolic information and global syntactic structures into Transformer models to generate concise and accurate code summaries. By distributing these structural properties hierarchically across the Transformer's layers, SG-Trans captures multi-level code characteristics effectively. Extensive experiments demonstrate that SG-Trans outperforms state-of-the-art models, improving program comprehension and documentation generation. This approach is particularly beneficial for large-scale software maintenance and educational tools. Wan et al. [196] enrich encoder-decoder frameworks using AST representations to enhance the contextuality of generated code summaries.

4.5.7 Code Synthesis and Code Editing. In recent years, deep representation learning on programs also facilitates generating, modifying, or enhancing code through automated means [147, 151]. CodeT5+ [209] revolutionizes code synthesis and editing by translating natural language specifications into functional code, providing real-time, context-aware coding suggestions directly within IDEs. It automates routine coding tasks, identifies syntax and logical errors, and offers corrective suggestions, enhancing productivity and reducing errors. The model also assists with code refactoring and optimization, promoting best practices in code structure and efficiency. Integration with version control systems ensures that only high-quality code is committed to repositories. By increasing coding efficiency and maintaining high standards of code quality, CodeT5+ acts as both a productivity tool and an educational aid for developers at all levels, while CodeFill [81] enhances IDEs by predicting multiple tokens, not just one, to synthesize complete code statements. It employs a parallel Transformer architecture to understand relationships in code, learning from both source code token names and their AST types. The system uses multi-task learning to improve prediction of token types and values, making it adept at context-aware suggestions that streamline code editing. Especially useful in dynamically typed languages, it minimizes common coding errors by providing accurate completions. Trained on extensive datasets, CodeFill adapts to various coding styles and practices, aiding in both simple edits and complex refactoring. Its intelligent suggestions not only boost coding efficiency but also serve as a learning tool for developers.

5 Discussion

Program representation learning is a crucial aspect of SE (especially AI4SE) research. Its primary goal is to extract meaningful features from source code to support a variety of applications. However, this field encounters several challenges, such as data preprocessing, architectural limitations, and

the selection of appropriate learning methods. Based on our survey, we delve into these challenges and propose potential solutions. Our recommendations are based on an in-depth analysis of recent advances in representation learning techniques.

5.1 Issues and Challenges

Current methods of representation learning for code often treat source code merely as sequences of natural language text, primarily focusing on surface-level syntax. This approach is **inadequate in capturing the deeper semantic** [186, 202, 215, 233] and syntactic [108, 188, 206, 248] elements inherent in the code. As a result, the representations generated may not fully reflect the complex structures and code semantics typical in real-world programs. For instance, if representation learning relies only on the token sequence of code without accounting for its syntactic structure, algorithms such as code clone detection might fail to recognize true clones in cases where one version is a simple syntactic refactoring of another. Moreover, focusing solely on syntactic details, such as AST, without integrating code semantics can lead to malware detection algorithms missing semantically equivalent malware samples. Furthermore, many existing models concentrate only on static features extracted from the code, such as its structure or control flow graphs, and overlook dynamic aspects such as execution traces. This omission can result in imprecise representations that do not accurately represent the program's behavior, undermining the effectiveness of the learned representations of code in downstream applications.

In contrast, approaches like those proposed in [201] address these challenges by incorporating both semantic and dynamic information into representation learning. However, integrating such information presents its own set of challenges. Semantic analysis, which involves understanding the meaning behind code elements, requires extensive processing and is particularly demanding for large codebases. This can be time-consuming and resource-intensive, making it less feasible in practical, real-world applications. Additionally, dynamic information such as execution traces, which are crucial for understanding program behavior, can also be costly to collect. Generating these traces necessitates executing programs with a variety of inputs, which may not always be practical, especially for large-scale software systems. Furthermore, relying on dynamic information from a limited set of inputs can lead to overfitting and restricted generalizability. Since such information captures only a fraction of possible program behaviors, the resulting representations may not generalize well to new, unseen scenarios, potentially affecting the model's recall and generalizability. Thus, while the integration of semantic and dynamic information into representation learning holds promise for enhancing the robustness and accuracy of the resulting models, it also introduces **practical challenges related to complexity, resource demands, and scalability**. Overcoming these challenges necessitates innovative strategies that effectively balance accuracy, efficiency, and scalability.

Current program representation learning approaches often utilize sequence-based models, such as RNNs or LSTM networks, to decipher the structural patterns of code. However, these models have inherent **limitations in capturing the complex, non-linear relationships typical in programs**. For instance, the multifaceted interactions between various code entities of a program are not adequately addressed by sequential processing alone. In this regard, graph representation learning offers a promising approach to address the complexities inherent in program representation by modeling code as a graph. In this model, nodes represent code elements such as variables and functions, while edges capturing relationships like function calls and data dependencies. This methodology allows for learning both local and global dependencies, thus providing a holistic understanding of the program.

However, transitioning to graph representation learning introduces several new challenges. The complexity of graph-based models exceeds that of traditional sequential models, necessitating

specialized architectures and advanced training procedures. This can pose significant hurdles, especially for developers who are not accustomed to graph-based techniques. Moreover, graph representation learning typically requires substantial amounts of training data. The process of constructing and labeling graph data is labor-intensive and time-consuming, particularly for expansive software projects. The **lack of readily available annotated graph datasets** in certain domains further complicates the application of graph-based methods in real-world settings. The scarcity of comprehensive graph representations in actual software projects can impede the effective application of these models.

Addressing these obstacles requires innovative strategies that leverage the strengths of graph representation learning while managing the practical constraints of training complexity, data availability, and application in real-world scenarios. Techniques such as semi-supervised learning, transfer learning, and data augmentation can alleviate the challenges of data scarcity. Advances in model architectures and training algorithms are also critical for improving the efficiency and scalability of graph-based methods. Encouraging collaboration between academic researchers and industry practitioners can further facilitate the adoption of graph-based representation learning in practical SE contexts.

Traditionally, program representation learning has relied heavily on supervised learning, where models are trained on labeled datasets to predict specific properties or classifications of code elements. However, this **reliance on labeled data** introduces significant challenges, particularly in the realm of obtaining large-scale, diverse datasets. The process of labeling code is not only labor-intensive and costly but also requires specific domain expertise.

To overcome these limitations, researchers have explored unsupervised learning as a viable alternative for program representation. Unsupervised learning seeks to extract meaningful patterns from unlabeled data, reducing reliance on extensively labeled datasets. Techniques such as autoencoders, GANs, and self-supervised learning are utilized to learn from raw code without the need for explicit labels. Yet, although unsupervised learning enhances data efficiency and scalability, it generally does not achieve the same level of accuracy and informativeness in representations as supervised learning, as we discussed earlier (Section 4.4). Unsupervised models may struggle to fully capture the complex semantic or structural properties of code without labeled guidance, potentially leading to the acquisition of **irrelevant features and suboptimal representations**.

Finally, extant deep representation learning methods for software programs are commonly limited to single-language software, largely dismissing the fact that real-world software systems are typically written in multiple programming languages [102, 104, 105]. However, multi-language software also suffers prevalent development issues [228, 229], functional bugs [227], and code vulnerabilities [106, 107, 138]. In response, researchers have approached these new challenges with data-driven (especially DL) approaches [109, 230]. However, like many similar prior works, the actual subject that some of these approaches deal with is single-language code, tackling one language at a time, rather than multilingual programs with language interoperability [103]. xLoc [230] actually aimed at programs with multiple, interacting languages, yet it fine-tunes a DL pretrained on code corpus of multiple individual languages using a cross-language API misuse dataset. As a result, it does **not learn cross-language code representations** either.

5.2 Future Directions

Current practices in program representation learning often treat input programs merely as sequences of natural language text, neglecting essential semantic and syntactic information. This approach significantly limits the robustness and effectiveness of the resulting models. To enhance the fidelity of program representations, future research should **focus on sophisticated input treatments that delve deeper into the code's semantics and syntax**. Techniques like AST

parsing, semantic parsing, and program dependency analysis should be employed to extract crucial structural information, such as function calls, variable dependencies, and control flow patterns. Integrating these structural properties into the input representation will allow models to better capture the code's underlying semantics, leading to more accurate and nuanced learning outcomes.

Moreover, the predominance of sequence learning models in current approaches does not adequately address the complex, non-sequential relationships inherent in software systems, such as those found in graph data. This gap can be bridged by **adopting graph representation learning techniques**. GNNs, in particular, are well-suited for capturing relational dependencies between code elements, including variables, functions, and control flows. However, the transition to graph-based models introduces challenges related to specialized architecture needs, training complexities, and data availability. Future research should aim to develop efficient and scalable graph representation learning algorithms that can effectively decode the rich semantics of code structures while addressing computational and data constraints.

Additionally, the majority of existing program representation learning research relies on supervised learning, which depends on access to large-scale, labeled datasets. Given the challenges and costs associated with obtaining such datasets, it is crucial to **explore alternative learning paradigms**. Unsupervised learning techniques, like autoencoders and VAEs, offer a pathway to learn from unlabeled data, thereby enabling more scalable and cost-effective training solutions. Furthermore, self-supervised learning approaches, which use pretext tasks such as predicting masked code elements or reconstructing corrupted code snippets, can facilitate the learning of meaningful representations without the need for manual annotations. Adopting these unsupervised and self-supervised techniques will help overcome the inherent limitations of supervised learning, promoting the development of more effective and accurate program representations.

Notably, a major inherent limitation of DL, including in the domain of program representation learning, lies in the lack of (or at least limited) generalizability. A fundamental reason behind this challenge is that the DL models are only trained on a relatively small scale of datasets. The advent and emerging power of **large language models** (**LLMs**) have shown to hold promise for closing this critical gap: in particular, foundation-scale LLMs such as the GPT and Claude series have demonstrated unprecedented, exciting generalizability—these LLMs have learned highly generalized deep representation of programs. Recent and increasingly more works have explored the potential of LLMs for many of the applications of deep program representations we discussed earlier in this article, including code synthesis/editing [49] and bug localization [30] as well as vulnerability detection [146] and repair [152]. Thus, we foresee growing efforts that **leverage the deep program representations in LLMs** for various future downstream applications.

In summary, future research in program representation learning should prioritize developing integrated approaches that leverage multiple input modalities—semantic, syntactic, and dynamic—to create more comprehensive and context-aware representations of software code. Researchers should also focus on creating more scalable and efficient algorithms capable of handling large-scale codebases and real-world programming scenarios. Advancements in program representation learning will enable a broader range of applications, from code analysis and synthesis to deeper understanding, thereby advancing the fields of SE in general and AI4SE in particular.

6 Related Work

Our survey extensively evaluates DL techniques in program representation learning, analyzing various neural network architectures, their inputs, and their applications in coding-related tasks such as code summarization, vulnerability detection, and software optimizations. This methodology allows us to delve deeper into specific applications and challenges in the core representation learning step of DL for SE, in comparison with existing literature.

Building on the work of Le et al. [96], we emphasize the critical role of DL in enhancing SE tasks. However, we extend this earlier exploration by providing a detailed analysis of neural network architectures, highlighting their capabilities and limitations in addressing complex SE challenges. This comparative analysis showcases the breadth of DL applications and emphasizes our novel contributions, particularly in terms of insights into model architectures and learning algorithms. Echoing the focus on structural data representation found in [231], we highlight the importance of leveraging ASTs and CFGs to enhance the interpretability and effectiveness of DL models for code. Unlike the earlier discussions in related work, our survey integrates these structures with advanced DL models like Transformers and GNNs, demonstrating how these sophisticated techniques can significantly enhance learning code representations from complex program structures.

Addressing the complexities of applying DL to programming languages, as discussed in [220], we recognize similar challenges but discuss more innovative methodologies that adapt DL models to the unique syntactic and semantic complexities of programming languages, offering practical solutions for real-world program representation learning. In alignment with Akimova et al. [3], which covers the broad applications of DL in SE, our survey takes a step further. We focus on the evolving nature of DL techniques for handling complex structures within source code, including in-depth discussions on effective application in code summarization, bug detection, and other critical SE tasks, providing a forward-looking perspective on potential transformations in software practices.

Similar to Lin et al. [121], which explores the impact of DL on software vulnerability detection, our survey not only addresses these specific applications but also extends the discussion to examine a much wider range of applications while focusing on the perspective of how the program representation learned can be utilized to support various SE tasks. In a similar contrast to Nong et al. [150] examining the state of and challenges in reproducibility and replicability of DL-based vulnerability detection, our survey provides a more exhaustive review of specific techniques, tools, and challenges involved in learning program representations. In agreement with Qiu et al. [164] on DL's role in malware detection on Android platforms, our survey broadens this view to encompass a variety of programming tasks where DL can offer significant benefits, including detailed analysis of its applicability across different programming environments and scenarios, highlighting its versatility and potential in enhancing security and performance.

In summary, our survey not only aligns with the existing body of work by underscoring the significance of DL in SE but also uniquely advances the discussion and knowledge systematization by providing a detailed analysis of the core step—representation learning, diving deeper in its vital aspects including neural network inputs, architectures, and learning algorithms. We also provide a more comprehensive analysis of the various applications of learned program representations. We further discuss promising strategies to tackle the inherent challenges in learning accurate (e.g., semantic) representations of complex code, emphasizing the necessity for adopting more advanced neural network architectures and developing more capable, semantics-aware learning algorithms.

7 Conclusion

In this article, we systematize the significant advancements in DL-based representations of software programs, highlighting the pivotal role of DNNs in code understanding and analysis. As DL advances, its application in SE, particularly in program representation, has shown considerable potential for tackling longstanding and emerging challenges. Specifically, this survey details a variety of DL approaches that transform raw program representations into latent representations (embeddings) that immediately enable/facilitate enhanced program comprehension, security defense, code editing/synthesis, and other critical SE tasks. The adoption of diverse neural network architectures, from convolutional and recurrent networks to advanced graph-based models, illustrates the adaptability of DL to the varying demands in learning code representations. Additionally, the review emphasizes

the ongoing need for comprehensive and scalable learning models capable of dealing with the complexities of real-world programs. This includes the integration of semantic understanding and dynamic information processing to more accurately learn the behaviors of programs.

Looking ahead, the survey suggests future research directions aimed at addressing current limitations related to code complexity, representation accuracy, and the extensive need for labeled datasets in program representation learning.

References

[1] Kumar Abhinav, Vijaya Sharvani, Alpana Dubey, Meenakshi D'Souza, Nitish Bhardwaj, Sakshi Jain, and Veenu Arora. 2021. RepairNet: Contextual sequence-to-sequence network for automated program repair. In *Proceedings of the International Conference on Artificial Intelligence in Education*. Springer, 3–15.

[2] Maristella Agosti, Stefano Marchesin, and Gianmaria Silvello. 2020. Learning unsupervised knowledge-enhanced representations to reduce the semantic gap in information retrieval. *ACM Transactions on Information Systems* 38, 4 (2020), 1–48.

[3] Elena N. Akimova, Alexander Yu Bersenev, Artem A. Deikov, Konstantin S. Kobylkin, Anton V. Konygin, Ilya P. Mezentsev, and Vladimir E. Misilov. 2021. A survey on software defect prediction using deep learning. *Mathematics* 9, 11 (2021), 1180.

[4] Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys* 51, 4 (2018), 1–37.

[5] Miltiadis Allamanis and Marc Brockschmidt. 2017. Smartpaste: Learning to adapt source code. arXiv:1705.07867. Retrieved from https://arxiv.org/abs/1705.07867

[6] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to represent programs with graphs. In *International Conference on Learning Representations*. 1–17.

[7] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019. code2seq: Generating Sequences from structured representations of code. In *International Conference on Learning Representations*. 1–22.

[8] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29.

[9] Wenyan An, Liwei Chen, Jinxin Wang, Gewangzi Du, Gang Shi, and Dan Meng. 2020. AVDHRAM: Automated vulnerability detection based on hierarchical representation and attention mechanism. In *Proceedings of the 2020 IEEE Intl Conf on Parallel and Distributed Processing with Applications, Big Data and Cloud Computing, Sustainable Computing and Communications, Social Computing and Networking (ISPA/BDCloud/SocialCom/SustainCom)*. IEEE, 337–344.

[10] Riyadh Baghdadi, Massinissa Merouani, Mohamed-Hicham Leghettas, Kamel Abdous, Taha Arbaoui, Karima Benatchba, and Saman Amarasinghe. 2021. A deep learning based cost model for automatic code optimization. *Proceedings of Machine Learning and Systems* 3 (2021), 181–193.

[11] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. 2019. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 193–205.

[12] Daniel Beck, Gholamreza Haffari, and Trevor Cohn. 2018. Graph-to-sequence learning using gated graph neural networks. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 273–283.

[13] Tal Ben-Nun, Alice S. Jakobovits, and Torsten Hoefler. 2019. Neural code comprehension: a learnable representation of code semantics. *Advances in Neural Information Processing Systems* 31 (2019), 3585–3597.

[14] Yoshua Bengio, Aaron Courville, and Pascal Vincent. 2013. Representation learning: A review and new perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 35, 8 (2013), 1798–1828.

[15] Alexandre Bérard, Christophe Servan, Olivier Pietquin, and Laurent Besacier. 2016. MultiVec: A multilingual and multilevel representation learning toolkit for NLP. In *Proceedings of the 10th International Conference on Language Resources and Evaluation (LREC'16)*. 4188–4192.

[16] Zeki Bilgin, Mehmet Akif Ersoy, Elif Ustundag Soykan, Emrah Tomur, Pinar Çomak, and Leyli Karaçay. 2020. Vulnerability prediction from source code using machine learning. *IEEE Access* 8 (2020), 150672–150684.

[17] Jón Arnar Briem, Jordi Smit, Hendrig Sellik, and Pavel Rapoport. 2019. Using distributed representation of code for bug detection. arXiv:1911.12863. Retrieved from https://arxiv.org/abs/1911.12863

[18] Lutz Büch and Artur Andrzejak. 2019. Learning-based recursive aggregation of abstract syntax trees for code clone detection. In *Proceedings of the 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 95–104.

[19] Nghi DQ Bui and Lingxiao Jiang. 2018. Hierarchical learning of cross-language mappings through distributed vector representations for code. In *Proceedings of the 2018 IEEE/ACM 40th International Conference on Software Engineering: New Ideas and Emerging Technologies Results (ICSE-NIER)*. IEEE, 33–36.

[20] Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. 2019. Autofocus: Interpreting attention-based neural networks by code perturbation. In *Proceedings of the 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 38–41.

[21] Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. 2021. Infercode: Self-supervised learning of code representations by predicting subtrees. In *Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1186–1197.

[22] Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. 2021. Self-supervised contrastive learning for code retrieval and summarization via semantic-preserving transformations. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 511–521.

[23] Rudy Bunel, Matthew Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. 2018. Leveraging grammar and reinforcement learning for neural program synthesis. In *International Conference on Learning Representations*. 1–15.

[24] Rocío Cabrera Lozoya, Arnaud Baumann, Antonino Sabetta, and Michele Bezzi. 2021. Commit2vec: Learning distributed representations of code changes. *SN Computer Science* 2, 3 (2021), 1–16.

[25] Haipeng Cai, Yu Nong, Yuzhe Ou, and Feng Chen. 2023. Generating vulnerable code via learning-based program transformations. In *Proceedings of the AI Embedded Assurance for Cyber Systems*. Springer, 123–138.

[26] Defu Cao, Jing Huang, Xuanyu Zhang, and Xianhua Liu. 2020. FTCLNet: Convolutional LSTM with fourier transform for vulnerability detection. In *Proceedings of the 2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. IEEE, 539–546.

[27] Junming Cao, Shouliang Yang, Wenhui Jiang, Hushuang Zeng, Beijun Shen, and Hao Zhong. 2020. BugPecker: Locating faulty methods with deep learning on revision graphs. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 1214–1218.

[28] Sicong Cao, Xiaobing Sun, Lili Bo, Rongxin Wu, Bin Li, and Chuanqi Tao. 2022. MVD: Memory-related vulnerability detection based on flow-sensitive graph neural networks. In *Proceedings of the 44th International Conference on Software Engineering*. 1456–1468.

[29] Gabriel B. Cavallari, Leonardo S. F. Ribeiro, and Moacir A. Ponti. 2018. Unsupervised representation learning using convolutional and stacked auto-encoders: A domain and cross-domain feature space analysis. In *Proceedings of the 2018 31st SIBGRAPI Conference on Graphics, Patterns and Images (SIBGRAPI)*. IEEE, 440–446.

[30] Partha Chakraborty, Mahmoud Alfadel, and Meiyappan Nagappan. 2025. BLAZE: Cross-language and cross-project bug localization via dynamic chunking and hard example learning. *IEEE Transactions on Software Engineering* 01 (2025), 1–14.

[31] Saikat Chakraborty, Yangruibo Ding, Miltiadis Allamanis, and Baishakhi Ray. 2020. Codit: Code editing with tree-based neural models. *IEEE Transactions on Software Engineering* 48, 4 (2020), 1385–1399.

[32] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2021. Deep learning-based vulnerability detection: Are we there yet? *IEEE Transactions on Software Engineering* 48, 9 (2021), 3280–3296.

[33] Hayden Cheers and Yuqing Lin. 2020. A novel graph-based program representation for java code plagiarism detection. In *Proceedings of the 3rd International Conference on Software Engineering and Information Management*. 115–122.

[34] Long Chen, Wei Ye, and Shikun Zhang. 2019. Capturing source code semantics via tree-based convolution over API-enhanced AST. In *Proceedings of the 16th ACM International Conference on Computing Frontiers*. 174–182.

[35] Qingying Chen and Minghui Zhou. 2018. A neural framework for retrieval and summarization of source code. In *Proceedings of the 2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 826–831.

[36] Xinyun Chen, Chang Liu, and Dawn Song. 2018. Execution-guided neural program synthesis. In *Proceedings of the International Conference on Learning Representations*.

[37] Yahui Chen. 2015. *Convolutional Neural Network for Sentence Classification*. Master's thesis. University of Waterloo.

[38] Zimin Chen, Steve Kommrusch, and Martin Monperrus. 2022. Neural transfer learning for repairing security vulnerabilities in c code. *IEEE Transactions on Software Engineering* 49, 1 (2022), 147–165.

[39] Xiao Cheng, Haoyu Wang, Jiayi Hua, Guoai Xu, and Yulei Sui. 2021. Deepwukong: Statically detecting software vulnerabilities using deep graph neural network. *ACM Transactions on Software Engineering and Methodology* 30, 3 (2021), 1–33.

[40] Boris Chernis and Rakesh Verma. 2018. Machine learning methods for software vulnerability detection. In *Proceedings of the 4th ACM International Workshop on Security and Privacy Analytics*. 31–39.

[41] Kyunghyun Cho, Bart van Merriënboer, Çağlar Gülçehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using RNN encoderdecoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 1724–1734.

[42] Sehun Jeong, Hakjoo Oh, Minje Choi, and Jaegul Choo. 2017. End-to-End prediction of buffer overruns from raw source code via neural memory networks. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, International Joint Conferences on Artificial Intelligence Organization*. 1–7.

[43] Rhys Compton, Eibe Frank, Panos Patros, and Abigail Koay. 2020. Embedding java classes with code2vec: Improvements from variable obfuscation. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 243–253.

[44] Alessandro Cornaglia, Alexander Viehl, Oliver Bringmann, and Wolfgang Rosenstiel. 2019. Simultime: Context-sensitive timing simulation on intermediate code representation for rapid platform explorations. In *Proceedings of the 24th Asia and South Pacific Design Automation Conference*. 526–531.

[45] Lei Cui, Zhiyu Hao, Yang Jiao, Haiqiang Fei, and Xiaochun Yun. 2020. VulDetector: Detecting vulnerabilities using weighted feature graph comparison. *IEEE Transactions on Information Forensics and Security* 16 (2020), 2004–2017.

[46] Chris Cummins, Zacharias V. Fisches, Tal Ben-Nun, Torsten Hoefler, and Hugh Leather. 2020. Programl: Graph-based deep learning for program optimization and analysis. arXiv:2003.10536. Retrieved from https://arxiv.org/abs/2003.10536

[47] Valentin Dallmeier and Thomas Zimmermann. 2007. Extraction of bug localization benchmarks from history. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*. 433–436.

[48] Hoa Khanh Dam, Truyen Tran, Trang Pham, Shien Wee Ng, John Grundy, and Aditya Ghose. 2018. Automatic feature learning for predicting vulnerable software components. *IEEE Transactions on Software Engineering* 47, 1 (2018), 67–85.

[49] Seyed Shayan Daneshvar, Yu Nong, Xu Yang, Shaowei Wang, and Haipeng Cai. 2025. VulScribeR: Exploring RAG-based vulnerability augmentation with LLMs. *ACM Transactions on Software Engineering and Methodology* (2025).

[50] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. 2017. Robustfill: Neural program learning under noisy i/o. In *Proceedings of the International Conference on Machine Learning*. PMLR, 990–998.

[51] Elizabeth Dinella, Hanjun Dai, Ziyang Li, Mayur Naik, Le Song, and Ke Wang. 2020. Hoppity: Learning graph transformations to detect and fix bugs in programs. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

[52] Xu Duan, Jingzheng Wu, Shouling Ji, Zhiqing Rui, Tianyue Luo, Mutian Yang, and Yanjun Wu. 2019. VulSniper: Focus your attention to shoot fine-grained vulnerabilities. In *Proceedings of the IJCAI*. 4665–4671.

[53] Ahmed Elnaggar, Wei Ding, Llion Jones, Tom Gibbs, Tamas Feher, Christoph Angerer, Silvia Severini, Florian Matthes, and Burkhard Rost. 2021. CodeTrans: Towards cracking the language of silicon's code through self-supervised deep learning and high performance computing. arXiv:2104.02443. Retrieved from https://arxiv.org/abs/2104.02443

[54] Chunrong Fang, Zixi Liu, Yangyang Shi, Jeff Huang, and Qingkai Shi. 2020. Functional code clone detection with syntax and semantics fusion learning. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 516–527.

[55] Yong Fang, Shengjun Han, Cheng Huang, and Runpu Wu. 2019. TAP: A static analysis model for PHP vulnerabilities based on token and deep learning technology. *PloS One* 14, 11 (2019), e0225196.

[56] Michael Fu and Chakkrit Tantithamthavorn. 2022. Linevul: A transformer-based line-level vulnerability prediction. In *Proceedings of the 19th International Conference on Mining Software Repositories*. 608–620.

[57] Xiaoqin Fu, Boxiang Lin, and Haipeng Cai. 2022. DistFax: A toolkit for measuring interprocess communications and quality of distributed systems. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. 51–55.

[58] Xiaoqin Fu, Asif Zaman, and Haipeng Cai. 2025. DistMeasure: A framework for runtime characterization and quality assessment of distributed software via interprocess communications. *ACM Transactions on Software Engineering and Methodology* 34, 3 (2025), 1–53.

[59] Shuzheng Gao, Cuiyun Gao, Yulan He, Jichuan Zeng, Lunyiu Nie, Xin Xia, and Michael Lyu. 2023. Code structure–guided transformer for source code summarization. *ACM Transactions on Software Engineering and Methodology* 32, 1 (2023), 1–32.

[60] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. 2017. Neural message passing for quantum chemistry. In *Proceedings of the International Conference on Machine Learning*. PMLR, 1263–1272.

[61] Yoav Goldberg and Omer Levy. 2014. word2vec explained: Deriving Mikolov et al.'s negative-sampling word-embedding method. arXiv:1402.3722. Retrieved from https://arxiv.org/abs/1402.3722

[62] Jiatao Gu, Yong Wang, Kyunghyun Cho, and Victor OK Li. 2018. Search engine guided neural machine translation. In *Proceedings of the AAAI Conference on Artificial Intelligence*.

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

[64] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified cross-modal pre-training for code representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 7212–7225.

[65] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCodeBERT: Pre-training code representations with data flow. In *International Conference on Learning Representations (ICLR)*. 1–18.

[66] Junjun Guo, Zhengyuan Wang, Haonan Li, and Yang Xue. 2023. Detecting vulnerability in source code using CNN and LSTM network. *Soft Computing* 27, 2 (2023), 1131–1141.

[67] Kavi Gupta, Peter Ebert Christensen, Xinyun Chen, and Dawn Song. 2020. Synthesize, execute and debug: Learning to repair for neural program synthesis. *Advances in Neural Information Processing Systems* 33 (2020), 17685–17695.

[68] Mouna Hadj-Kacem and Nadia Bouassida. 2019. Deep representation learning for code smells detection using variational auto-encoder. In *Proceedings of the 2019 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 1–8.

[69] Jacob Harer, Tomo Lazovich, Rebecca Russell, Onur Ozdemir, and Louis Kim. 2020. Automated repair of bugs and security vulnerabilities in software. US Patent 10,866,877.

[70] Jacob A. Harer, Onur Ozdemir, Tomo Lazovich, Christopher P. Reale, Rebecca L. Russell, Louis Y. Kim, and Peter Chin. 2018. Learning to repair software vulnerabilities with generative adversarial networks. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*. 7944–7954.

[71] Chaoyang He, Tian Xie, Yu Rong, Wenbing Huang, Junzhou Huang, Xiang Ren, and Cyrus Shahabi. 2019. Cascade-BGNN: Toward efficient self-supervised representation learning on large-scale bipartite graphs. arXiv:1906.11994. Retrieved from https://arxiv.org/abs/1906.11994

[72] David Hin, Andrey Kan, Huaming Chen, and M. Ali Babar. 2022. Linevd: Statement-level vulnerability detection using graph neural networks. In *Proceedings of the 19th International Conference on Mining Software Repositories*. 596–607.

[73] Thong Hoang, Hong Jin Kang, David Lo, and Julia Lawall. 2020. Cc2vec: Distributed representations of code changes. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 518–529.

[74] Thong Hoang, Julia Lawall, Yuan Tian, Richard J Oentaryo, and David Lo. 2019. Patchnet: Hierarchical deep learning-based stable patch identification for the linux kernel. *IEEE Transactions on Software Engineering* (2019).

[75] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural Computation* 9, 8 (1997), 1735–1780.

[76] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *Proceedings of the 2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. IEEE, 200–20010.

[77] Wei Hua, Yulei Sui, Yao Wan, Guangzhong Liu, and Guandong Xu. 2020. FCCA: Hybrid code representation for functional clone detection using attention networks. *IEEE Transactions on Reliability* 70, 1 (2020), 304–318.

[78] Qing Huang, An Qiu, Maosheng Zhong, and Yuan Wang. 2020. A code-description representation learning model based on attention. In *Proceedings of the 2020 IEEE 27th International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 447–455.

[79] Sergey Ioffe and Christian Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the International Conference on Machine Learning*. PMLR, 448–456.

[80] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2073–2083.

[81] Maliheh Izadi, Roberta Gismondi, and Georgios Gousios. 2022. Codefill: Multi-token code completion by jointly learning from structure and naming sequences. In *Proceedings of the 44th International Conference on Software Engineering*. 401–412.

[82] Paras Jain, Ajay Jain, Tianjun Zhang, Pieter Abbeel, Joseph Gonzalez, and Ion Stoica. 2021. Contrastive code representation learning. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. 5954–5971.

[83] Vinoj Jayasundara, Nghi Duy Quoc Bui, Lingxiao Jiang, and David Lo. 2019. TreeCaps: Tree-structured capsule networks for program source code processing. arXiv:1910.12306. Retrieved from https://arxiv.org/abs/1910.12306

[84] Nan Jiang, Thibaud Lutellier, Yiling Lou, Lin Tan, Dan Goldwasser, and Xiangyu Zhang. 2023. Knod: Domain knowledge distilled tree decoder for automated program repair. In *Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1251–1263.

[85] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. CURE: Code-aware neural machine translation for automatic program repair. In *Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1161–1173.

[86] Rie Johnson and Tong Zhang. 2017. Deep pyramid convolutional neural networks for text categorization. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 562–570.

[87] John B. Kam and Jeffrey D. Ullman. 1977. Monotone data flow analysis frameworks. *Acta Informatica* 7, 3 (1977), 305–317.

[88] Hong Jin Kang, Tegawendé F Bissyandé, and David Lo. 2019. Assessing the generalizability of code2vec token embeddings. In *Proceedings of the 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1–12.

[89] Patrick Keller, Abdoul Kader Kaboré, Laura Plein, Jacques Klein, Yves Le Traon, and Tegawendé F. Bissyandé. 2022. What you see is what it means! semantic representation learning of code based on visualization and transfer learning. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 2 (2022), 1–34. DOI: https://doi.org/10.1145/3485135

[90] Shima Khoshraftar and Aijun An. 2024. A Survey on graph representation learning methods. *ACM Transactions on Intelligent Systems and Technology* 15, 1 (2024), 1–55.

[91] Shima Khoshraftar and Aijun An. 2024. A survey on graph representation learning methods. *ACM Transactions on Intelligent Systems and Technology* 15, 1 (2024), 1–55.

[92] Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. 2021. Code prediction by feeding trees to transformers. In *Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 150–162.

[93] Steve Kommrusch, Théo Barollet, and Louis-Noël Pouchet. 2021. Proving equivalence between complex expressions using graph-to-sequence neural models. *arXiv e-prints* (2021), arXiv–2106.

[94] Tomasz Korbak, Hady Elsahar, Marc Dymetman, and Germán Kruszewski. 2021. Energy-Based Models for Code Generation under Compilability Constraints. In *Proceedings of the First Workshop on Natural Language Processing for Programming (NLP4Prog)*. ACL, 1–18.

[95] Taku Kudo. 2018. Subword regularization: Improving neural network translation models with multiple subword candidates. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 66–75.

[96] Triet HM Le, Hao Chen, and Muhammad Ali Babar. 2020. Deep learning for source code modeling and generation: Models, applications, and challenges. *ACM Computing Surveys* 53, 3 (2020), 1–38.

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

[98] Jia Li, Yongmin Li, Ge Li, Zhi Jin, Yiyang Hao, and Xing Hu. 2023. Skcoder: A sketch-based approach for automatic code generation. In *Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2124–2135.

[99] Raymond Li, Loubna Ben allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia LI, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Joel Lamy-Poirier, Joao Monteiro, Nicolas Gontier, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Ben Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason T Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Urvashi Bhattacharyya, Wenhao Yu, Sasha Luccioni, Paulo Villegas, Fedor Zhdanov, Tony Lee, Nadav Timor, Jennifer Ding, Claire S Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro Von Werra, and Harm de Vries. 2023. StarCoder: may the source be with you! *Transactions on Machine Learning Research* (2023). Retrieved from https://openreview.net/forum?id=KoFOg41haE

[100] Runhao Li, Chao Feng, Xing Zhang, and Chaojing Tang. 2019. A lightweight assisted vulnerability discovery method using deep neural networks. *IEEE Access* 7 (2019), 80079–80092.

[101] Ruitong Li, Gang Hu, and Min Peng. 2020. Hierarchical embedding for code search in software Q&A sites. In *Proceedings of the 2020 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 1–10.

[102] Wen Li, Li Li, and Haipeng Cai. 2022. On the vulnerability proneness of multilingual code. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 847–859.

[103] Wen Li, Li Li, and Haipeng Cai. 2022. PolyFax: A toolkit for characterizing multi-language software. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1662–1666.

[104] Wen Li, Austin Marino, Haoran Yang, Na Meng, Li Li, and Haipeng Cai. 2024. How are multilingual systems constructed: Characterizing language use and selection in open-source multilingual software. *ACM Transactions on Software Engineering and Methodology* 33, 3 (2024), 1–46.

[105] Wen Li, Na Meng, Li Li, and Haipeng Cai. 2021. Understanding language selection in multi-language software projects on GitHub. In *Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 256–257.

[106] Wen Li, Jiang Ming, Xiapu Luo, and Haipeng Cai. 2022. {PolyCruise}: A {Cross-Language} dynamic information flow analysis. In *Proceedings of the 31st USENIX Security Symposium (USENIX Security 22)*. 2513–2530.

[107] Wen Li, Jinyang Ruan, Guangbei Yi, Long Cheng, Xiapu Luo, and Haipeng Cai. 2023. {PolyFuzz}: Holistic greybox fuzzing of {multi-language} systems. In *Proceedings of the 32nd USENIX Security Symposium (USENIX Security 23)*. 1379–1396.

[108] Xin Li, Lu Wang, Yang Xin, Yixian Yang, and Yuling Chen. 2020. Automated vulnerability detection in source code using minimum intermediate representation learning. *Applied Sciences* 10, 5 (2020), 1692.

[109] Yang Li, Qin Luo, Peng Wu, and Hongdi Zheng. 2025. VDMAF: Cross-language source code vulnerability detection using multi-head attention fusion. *Information and Software Technology* 183 (2025), 107739.

[110] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard S. Zemel. 2016. Gated graph sequence neural networks. In *International Conference on Learning Representations (ICLR)*. 1–20.

[111] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. Dlfix: Context-based code transformation learning for automated program repair. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 602–614.

[112] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2021. Fault localization with code coverage representation learning. In *Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 661–673.

[113] Yi Li, Shaohua Wang, and Tien N Nguyen. 2022. Dear: A novel deep learning-based approach for automated program repair. In *Proceedings of the 44th International Conference on Software Engineering*. 511–523.

[114] Yi Li, Shaohua Wang, and Tien N Nguyen. 2023. Contextuality of code representation learning. In *Proceedings of the 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 548–559.

[115] Yi Li, Shaohua Wang, Tien N. Nguyen, and Son Van Nguyen. 2019. Improving bug detection via context-based code representation learning and attention-based neural networks. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–30.

[116] Zhen Li, Deqing Zou, Jing Tang, Zhihao Zhang, Mingqian Sun, and Hai Jin. 2019. A comparative study of deep learning-based vulnerability detection system. *IEEE Access* 7 (2019), 103184–103197.

[117] Zhen Li, Deqing Zou, Shouhuai Xu, Zhaoxuan Chen, Yawei Zhu, and Hai Jin. 2021. Vuldeelocator: a deep learning-based fine-grained vulnerability detector. *IEEE Transactions on Dependable and Secure Computing* 19, 4 (2021), 2821–2837.

[118] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. 2021. Sysevr: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing* 19, 4 (2021), 2244–2258.

[119] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. *Proceedings of the 2018 Network and Distributed System Security Symposium (NDSS '18)*. 1–15. arXiv:1801.01681. Retrieved from https://arxiv.org/abs/1801.01681

[120] Bo Lin, Shangwen Wang, Ming Wen, and Xiaoguang Mao. 2022. Context-aware code change embedding for better patch correctness assessment. *ACM Transactions on Software Engineering and Methodology* 31, 3 (2022), 1–29.

[121] Guanjun Lin, Sheng Wen, Qing-Long Han, Jun Zhang, and Yang Xiang. 2020. Software vulnerability detection using deep neural networks: A survey. *Proceedings of the IEEE* 108, 10 (2020), 1825–1848.

[122] Guanjun Lin, Jun Zhang, Wei Luo, Lei Pan, Olivier De Vel, Paul Montague, and Yang Xiang. 2019. Software vulnerability discovery via learning multi-domain knowledge bases. *IEEE Transactions on Dependable and Secure Computing* 18, 5 (2019), 2469–2485.

[123] Guanjun Lin, Jun Zhang, Wei Luo, Lei Pan, and Yang Xiang. 2017. POSTER: Vulnerability discovery with function representation learning from unlabeled projects. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2539–2541.

[124] Guanjun Lin, Jun Zhang, Wei Luo, Lei Pan, Yang Xiang, Olivier De Vel, and Paul Montague. 2018. Cross-project transfer representation learning for vulnerable function discovery. *IEEE Transactions on Industrial Informatics* 14, 7 (2018), 3289–3297.

[125] Bohong Liu, Tao Wang, Xunhui Zhang, Qiang Fan, Gang Yin, and Jinsheng Deng. 2019. A neural-network based code summarization approach by using source code and its call dependencies. In *Proceedings of the 11th Asia-Pacific Symposium on Internetware*. 1–10.

[126] Hao Liu, Jindong Han, Yanjie Fu, Jingbo Zhou, Xinjiang Lu, and Hui Xiong. 2020. Multi-modal transportation recommendation with unified route representation learning. *Proceedings of the VLDB Endowment* 14, 3 (2020), 342–350.

[127] Shangqing Liu, Yu Chen, Xiaofei Xie, Jing Kai Siow, and Yang Liu. 2020. Automatic code summarization via multi-dimensional semantic fusing in GNN. arXiv:2006.05405. Retrieved from https://arxiv.org/abs/2006.05405

[128] Shangqing Liu, Cuiyun Gao, Sen Chen, Lun Yiu Nie, and Yang Liu. 2020. Atom: Commit message generation based on abstract syntax tree and hybrid ranking. *IEEE Transactions on Software Engineering* 48, 5 (2020), 1800–1817.

[129] Shangqing Liu, Xiaofei Xie, Jingkai Siow, Lei Ma, Guozhu Meng, and Yang Liu. 2023. Graphsearchnet: Enhancing gnns via capturing global dependencies for semantic code search. *IEEE Transactions on Software Engineering* 49, 4 (2023), 2839–2855.

[130] Zhongxin Liu, Zhijie Tang, Xin Xia, and Xiaohu Yang. 2023. CCRep: Learning code change representations via pre-trained code model and query back. In *Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 17–29.

[131] Pablo Loyola, Kugamoorthy Gajananan, and Fumiko Satoh. 2018. Bug localization by learning to rank and represent bug inducing changes. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*. 657–665.

[132] Thibaud Lutellier, Lawrence Pang, Viet Hung Pham, Moshi Wei, and Lin Tan. 2019. ENCORE: Ensemble learning using convolution neural machine translation for automatic program repair. arXiv:1906.08691. Retrieved from https://arxiv.org/abs/1906.08691

[133] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. Coconut: Combining context-aware neural translation models using ensemble for program repair. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 101–114.

[134] Fei Lv, Hongyu Zhang, Jian-guang Lou, Shaowei Wang, Dongmei Zhang, and Jianjun Zhao. 2015. Codehow: Effective code search based on api understanding and extended boolean model (e). In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 260–270.

[135] Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. 2019. NL2Type: inferring JavaScript function types from natural language information. In *Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 304–315.

[136] Yi Mao, Yun Li, Jiatai Sun, and Yixin Chen. 2020. Explainable software vulnerability detection based on attention-based bidirectional recurrent neural networks. In *Proceedings of the 2020 IEEE International Conference on Big Data (Big Data)*. IEEE, 4651–4656.

[137] Antonio Mastropaolo, Simone Scalabrino, Nathan Cooper, David Nader Palacio, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. 2021. Studying the usage of text-to-text transfer transformer to support code-related tasks. In *Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 336–347.

[138] Samuel Mergendahl, Nathan Burow, and Hamed Okhravi. 2022. Cross-language attacks. In *Proceedings of the NDSS*. 1–18.

[139] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the 30th AAAI Conference on Artificial Intelligence*.

[140] Annamalai Narayanan, Mahinthan Chandramohan, Rajasekar Venkatesan, Lihui Chen, Yang Liu, and Shantanu Jaiswal. 2017. graph2vec: Learning distributed representations of graphs. arXiv:1707.05005. Retrieved from https://arxiv.org/abs/1707.05005

[141] Arvind Neelakantan, Tao Xu, Raul Puri, Alec Radford, Jesse Michael Han, Jerry Tworek, Qiming Yuan, Nikolas Tezak, Jong Wook Kim, Chris Hallacy, Johannes Heidecke, Pranav Shyam, Boris Power, Tyna Eloundou Nekoul, Girish Sastry, Gretchen Krueger, David Schnurr, Felipe Petroski Such, Kenny Hsu, Madeleine Thompson, Tabarak Khan, Toki Sherbakov, and Joanne Jang. 2022. Text and code embeddings by contrastive pre-training. *arXiv preprint arXiv:2201.10005* (2022).

[142] Van Nguyen, Trung Le, Tue Le, Khanh Nguyen, Olivier DeVel, Paul Montague, Lizhen Qu, and Dinh Phung. 2019. Deep domain adaptation for vulnerable code function identification. In *Proceedings of the 2019 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 1–8.

[143] Van-Anh Nguyen, Dai Quoc Nguyen, Van Nguyen, Trung Le, Quan Hung Tran, and Dinh Phung. 2022. Regvd: Revisiting graph neural networks for vulnerability detection. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. 178–182.

[144] Lun Yiu Nie, Cuiyun Gao, Zhicong Zhong, Wai Lam, Yang Liu, and Zenglin Xu. 2021. CoreGen: Contextualized code representation learning for commit message generation. *Neurocomputing* 459 (2021), 97–107.

[145] Changan Niu, Chuanyi Li, Vincent Ng, Jidong Ge, Liguo Huang, and Bin Luo. 2022. Spt-code: Sequence-to-sequence pre-training for learning source code representations. In *Proceedings of the 44th International Conference on Software Engineering*. 2006–2018.

[146] Yu Nong, Mohammed Aldeen, Long Cheng, Hongxin Hu, Feng Chen, and Haipeng Cai. 2024. Chain-of-thought prompting of large language models for discovering and fixing software vulnerabilities. arXiv:2402.17230. Retrieved from https://arxiv.org/abs/2402.17230

[147] Yu Nong, Richard Fang, Guangbei Yi, Kunsong Zhao, Xiapu Luo, Feng Chen, and Haipeng Cai. 2024. Vgx: Large-scale sample generation for boosting learning-based software vulnerability analyses. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

[148] Yu Nong, Yuzhe Ou, Michael Pradel, Feng Chen, and Haipeng Cai. 2022. Generating realistic vulnerabilities via neural code editing: An empirical study. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1097–1109.

[149] Yu Nong, Yuzhe Ou, Michael Pradel, Feng Chen, and Haipeng Cai. 2023. Vulgen: Realistic vulnerability generation via pattern mining and deep learning. In *Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 2527–2539.

[150] Yu Nong, Rainy Sharma, Abdelwahab Hamou-Lhadj, Xiapu Luo, and Haipeng Cai. 2022. Open science in software engineering: A study on deep learning-based vulnerability detection. *IEEE Transactions on Software Engineering* 49, 4 (2022), 1983–2005.

[151] Yu Nong, Haoran Yang, Feng Chen, and Haipeng Cai. 2024. VinJ: An automated tool for large-scale software vulnerability data generation. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*. 567–571.

[152] Yu Nong, Haoran Yang, Long Cheng, Hongxin Hu, and Haipeng Cai. 2025. {APPATCH}: Automated adaptive prompting large language models for {real-world} software vulnerability patching. In *Proceedings of the 34th USENIX Security Symposium (USENIX Security 25)*. 4481–4500.

[153] Safa Omri and Carsten Sinz. 2020. Deep learning for software defect prediction: A survey. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*. 209–214.

[154] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. 2016. Spoon: A library for implementing analyses and transformations of java source code. *Software: Practice and Experience* 46, 9 (2016), 1155–1179.

[155] Alexander Pechenkin and Roman Demidov. 2018. Applying deep learning and vector representation for software vulnerabilities detection. In *Proceedings of the 11th International Conference on Security of Information and Networks*. 1–6.

[156] Yuxin Peng and Jinwei Qi. 2019. CM-GANs: Cross-modal generative adversarial networks for common representation learning. *ACM Transactions on Multimedia Computing, Communications, and Applications* 15, 1 (2019), 1–24.

[157] Daniel Perez and Shigeru Chiba. 2019. Cross-language clone detection by learning over abstract syntax trees. In *Proceedings of the 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 518–528.

[158] Anh Viet Phan, Minh Le Nguyen, and Lam Thu Bui. 2017. Convolutional neural networks over control flow graphs for software defect prediction. In *Proceedings of the 2017 IEEE 29th International Conference on Tools with Artificial Intelligence (ICTAI)*. IEEE, 45–52.

[159] Weiguo Pian, Hanyu Peng, Xunzhu Tang, Tiezhu Sun, Haoye Tian, Andrew Habib, Jacques Klein, and Tegawendé F. Bissyandé. 2023. MetaTPTrans: A meta learning approach for multilingual code representation learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*. 5239–5247.

[160] Chanathip Pornprasit and Chakkrit Kla Tantithamthavorn. 2022. Deeplinedp: Towards a deep learning approach for line-level defect prediction. *IEEE Transactions on Software Engineering* 49, 1 (2022), 84–98.

[161] Michael Pradel and Koushik Sen. 2018. Deepbugs: A learning approach to name-based bug detection. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–25.

[162] Mikhail Pravilov, Egor Bogomolov, Yaroslav Golubev, and Timofey Bryksin. 2021. Unsupervised learning of general-purpose embeddings for code changes. In *Proceedings of the 5th International Workshop on Machine Learning Techniques for Software Quality Evolution*. 7–12.

[163] Jiezhong Qiu, Qibin Chen, Yuxiao Dong, Jing Zhang, Hongxia Yang, Ming Ding, Kuansan Wang, and Jie Tang. 2020. Gcc: Graph contrastive coding for graph neural network pre-training. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 1150–1160.

[164] Junyang Qiu, Jun Zhang, Wei Luo, Lei Pan, Surya Nepal, and Yang Xiang. 2020. A survey of android malware detection with deep neural models. *ACM Computing Surveys* 53, 6 (2020), 1–36.

[165] Md Rafiqul Islam Rabin, Vincent J. Hellendoorn, and Mohammad Amin Alipour. 2021. Understanding neural code intelligence through program simplification. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 441–452.

[166] Tarek Ramadan, Tanzima Z. Islam, Chase Phelps, Nathan Pinnow, and Jayaraman J. Thiagarajan. 2021. Comparative code structure analysis using deep learning for performance prediction. In *Proceedings of the 2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 151–161.

[167] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: A method for automatic evaluation of code synthesis. arXiv:2009.10297. Retrieved from https://arxiv.org/abs/2009.10297

[168] Muhammad Fakhrur Rozi, Sangwook Kim, and Seiichi Ozawa. 2020. Deep neural networks for malicious JavaScript detection using bytecode sequences. In *Proceedings of the 2020 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 1–8.

[169] Shubham Sangle, Sandeep Muvva, Sridhar Chimalakonda, Karthikeyan Ponnalagu, and Vijendran Gopalan Venkoparao. 2020. DRAST–a deep learning and AST based approach for bug localization. arXiv:2011.03449. Retrieved from https://arxiv.org/abs/2011.03449

[170] Florian Schroff, Dmitry Kalenichenko, and James Philbin. 2015. Facenet: A unified embedding for face recognition and clustering. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 815–823.

[171] Mike Schuster and Kuldip K. Paliwal. 1997. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing* 45, 11 (1997), 2673–2681.

[172] Shashank Sharma and Sumit Srivastava. 2020. EGAN: An effective code readability classification using ensemble generative adversarial networks. In *Proceedings of the 2020 International Conference on Computation, Automation and Knowledge Management (ICCAKM)*. IEEE, 312–316.

[173] Abdullah Sheneamer and Jugal Kalita. 2016. Semantic clone detection using machine learning. In *Proceedings of the 2016 15th IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE, 1024–1028.

[174] Ke Shi, Yang Lu, Guangliang Liu, Zhenchun Wei, and Jingfei Chang. 2021. MPT-embedding: An unsupervised representation learning of code for software defect prediction. *Journal of Software: Evolution and Process* 33, 4 (2021), e2330.

[175] Richard Shin, Illia Polosukhin, and Dawn Song. 2018. Improving neural program synthesis with inferred execution traces. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*. 8931–8940.

[176] Jianhang Shuai, Ling Xu, Chao Liu, Meng Yan, Xin Xia, and Yan Lei. 2020. Improving code search with co-attentive representation learning. In *Proceedings of the 28th International Conference on Program Comprehension*. 196–207.

[177] Mohammed Latif Siddiq, Md Rezwanur Rahman Jahin, Mohammad Rafid Ul Islam, Rifat Shahriyar, and Anindya Iqbal. 2021. SQLIFIX: Learning based approach to fix SQL injection vulnerabilities in source code. In *Proceedings of the 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 354–364.

[178] Jing Kai Siow, Cuiyun Gao, Lingling Fan, Sen Chen, and Yang Liu. 2020. Core: Automating review recommendation for code changes. In *Proceedings of the 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 284–295.

[179] Yulei Sui, Xiao Cheng, Guanqin Zhang, and Haoyu Wang. 2020. Flow2vec: Value-flow-based precise code embedding. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–27.

[180] Martin Sundermeyer, Tamer Alkhouli, Joern Wuebker, and Hermann Ney. 2014. Translation modeling with bidirectional recurrent neural networks. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 14–25.

[181] Sahil Suneja, Yunhui Zheng, Yufan Zhuang, Jim Laredo, and Alessandro Morari. 2020. Learning to map source code to software vulnerability using code-as-a-graph. arXiv:2006.08614. Retrieved from https://arxiv.org/abs/2006.08614

[182] Kai Sheng Tai, Richard Socher, and Christopher D. Manning. 2015. Improved Semantic representations from tree-structured long short-term memory networks. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. 1556–1566.

[183] Yunosuke Teshima and Yutaka Watanobe. 2018. Bug detection based on lstm networks and solution codes. In *Proceedings of the 2018 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. IEEE, 3541–3546.

[184] Sebastian Thrun and Michael L. Littman. 2000. Reinforcement learning: An introduction. *AI Magazine* 21, 1 (2000), 103–103.

[185] Haoye Tian, Kui Liu, Abdoul Kader Kaboré, Anil Koyuncu, Li Li, Jacques Klein, and Tegawendé F Bissyandé. 2020. Evaluating representation learning of code changes for predicting patch correctness in program repair. In *Proceedings of the 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 981–992.

[186] Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. 2019. On learning meaningful code changes via neural machine translation. In *Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 25–36.

[187] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2018. Deep learning similarities from different representations of source code. In *Proceedings of the 2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. IEEE, 542–553.

[188] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2018. An empirical investigation into learning bug-fixing patches in the wild via neural machine translation. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 832–837.

[189] Rosalia Tufano, Simone Masiero, Antonio Mastropaolo, Luca Pascarella, Denys Poshyvanyk, and Gabriele Bavota. 2022. Using pre-trained models to boost code review automation. In *Proceedings of the 44th International Conference on Software Engineering*. 2291–2302.

[190] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of the Advances in Neural Information Processing Systems*. 5998–6008.

[191] Oriol Vinyals, Samy Bengio, and Manjunath Kudlur. 2016. Order matters: sequence to sequence for sets. In *International Conference on Learning Representations (ICLR)*. 1–11.

[192] Nickolay Viuginov, Petr Grachev, and Andrey Filchenkov. 2020. A machine learning based plagiarism detection in source code. In *Proceedings of the 2020 3rd International Conference on Algorithms, Computing and Artificial Intelligence*. 1–6.

[193] Petr Vytovtov and Kirill Chuvilin. 2019. Unsupervised classifying of software source code using graph neural networks. In *Proceedings of the 2019 24th Conference of Open Innovations Association (FRUCT)*. IEEE, 518–524.

[194] Yaza Wainakh, Moiz Rauf, and Michael Pradel. 2021. IdBench: Evaluating semantic representations of identifier names in source code. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 562–573.

[195] Yaza Wainakh, Moiz Rauf, and Michael Pradel. 2021. Idbench: Evaluating semantic representations of identifier names in source code. In *Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 562–573.

[196] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S. Yu. 2018. Improving automatic source code summarization via deep reinforcement learning. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 397–407.

[197] Chuangwei Wang, Yifan Wu, and Xiaofang Zhang. 2023. Mucha: Multi-channel based code change representation learning for commit message generation. In *Proceedings of the 2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security (QRS)*. IEEE, 593–603.

[198] Chaozheng Wang, Yuanhang Yang, Cuiyun Gao, Yun Peng, Hongyu Zhang, and Michael R. Lyu. 2022. No more fine-tuning? An experimental evaluation of prompt tuning in code intelligence. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 382–394.

[199] Deze Wang, Zhouyang Jia, Shanshan Li, Yue Yu, Yun Xiong, Wei Dong, and Xiangke Liao. 2022. Bridging pre-trained models and downstream tasks for source code understanding. In *Proceedings of the 44th International Conference on Software Engineering*. 287–298.

[200] Deze Wang, Yue Yu, Shanshan Li, Wei Dong, Ji Wang, and Liao Qing. 2021. MulCode: A multi-task learning approach for source code understanding. In *Proceedings of the 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 48–59.

[201] Ke Wang and Zhendong Su. 2020. Blended, precise semantic program embeddings. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 121–134.

[202] Song Wang, Taiyue Liu, Jaechang Nam, and Lin Tan. 2018. Deep semantic feature learning for software defect prediction. *IEEE Transactions on Software Engineering* 46, 12 (2018), 1267–1293.

[203] Song Wang, Taiyue Liu, and Lin Tan. 2016. Automatically learning semantic features for defect prediction. In *Proceedings of the 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 297–308.

[204] Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. 2020. Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In *Proceedings of the 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 261–271.

[205] Wenhan Wang, Ge Li, Sijie Shen, Xin Xia, and Zhi Jin. 2020. Modular tree network for source code representation learning. *ACM Transactions on Software Engineering and Methodology* 29, 4 (2020), 1–23.

[206] Wenhua Wang, Yuqun Zhang, Yulei Sui, Yao Wan, Zhou Zhao, Jian Wu, Philip S Yu, and Guandong Xu. 2020. Reinforcement-learning-guided source code summarization using hierarchical attention. *IEEE Transactions on Software Engineering* 48, 1 (2020), 102–119.

[207] Xin Wang, Yasheng Wang, Fei Mi, Pingyi Zhou, Yao Wan, Xiao Liu, Li Li, Hao Wu, Jin Liu, and Xin Jiang. 2021. Syncobert: Syntax-guided multi-modal contrastive pre-training for code representation. *arXiv preprint arXiv:2108.04556* (2021).

[208] Yequan Wang, Minlie Huang, Xiaoyan Zhu, and Li Zhao. 2016. Attention-based LSTM for aspect-level sentiment classification. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*. 606–615.

[209] Yue Wang, Hung Le, Akhilesh Gotmare, Nghi Bui, Junnan Li, and Steven Hoi. 2023. CodeT5+: Open code large language models for code understanding and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. 1069–1088.

[210] Yequan Wang, Aixin Sun, Jialong Han, Ying Liu, and Xiaoyan Zhu. 2018. Sentiment analysis by capsules. In *Proceedings of the 2018 World Wide Web Conference*. 1165–1174.

[211] Yu Wang, Ke Wang, Fengjuan Gao, and Linzhang Wang. 2020. Learning semantic program embeddings with graph interval neural network. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–27.

[212] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, Association for Computational Linguistics*. 8696–8708.

[213] Yunchao Wang, Zehui Wu, Qiang Wei, and Qingxian Wang. 2019. Neufuzz: Efficient fuzzing with deep neural network. *IEEE Access* 7 (2019), 36340–36352.

[214] Martin Weyssow, Houari Sahraoui, Benoıt Frénay, and Benoıt Vanderose. 2020. Combining code embedding with static analysis for function-call completion. arXiv:1701.00133. Retrieved from https://arxiv.org/abs/1701.00133

[215] Fang Wu, Jigang Wang, Jiqiang Liu, and Wei Wang. 2017. Vulnerability detection with deep learning. In *Proceedings of the 2017 3rd IEEE International Conference on Computer and Communications (ICCC)*. IEEE, 1298–1302.

[216] Yuelong Wu, Jintian Lu, Yunyi Zhang, and Shuyuan Jin. 2021. Vulnerability detection in C/C++ source code with graph representation learning. In *Proceedings of the 2021 IEEE 11th Annual Computing and Communication Workshop and Conference (CCWC)*. IEEE, 1519–1524.

[217] Chunqiu Steven Xia and Lingming Zhang. 2022. Less training, more repairing please: Revisiting automated program repair via zero-shot learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 959–971.

[218] Yan Xiao and Jacky Keung. 2018. Improving bug localization with character-level convolutional neural network and recurrent neural network. In *Proceedings of the 2018 25th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 703–704.

[219] Wang Xiaomeng, Zhang Tao, Wu Runpu, Xin Wei, and Hou Changyu. 2018. CPGVA: Code property graph based vulnerability analysis by deep learning. In *Proceedings of the 2018 10th International Conference on Advanced Infocomm Technology (ICAIT)*. IEEE, 184–188.

[220] Yutao Xie, Jiayi Lin, Hande Dong, Lei Zhang, and Zhonghai Wu. 2023. Survey of code search based on deep learning. *ACM Transactions on Software Engineering and Methodology* 33, 2 (2023), 1–42. DOI: https://doi.org/10.1145/3628161

[221] Aidong Xu, Tao Dai, Huajun Chen, Zhe Ming, and Weining Li. 2018. Vulnerability detection for source code using contextual LSTM. In *Proceedings of the 2018 5th International Conference on Systems and Informatics (ICSAI)*. IEEE, 1225–1230.

[222] Bowen Xu, Thong Hoang, Abhishek Sharma, Chengran Yang, Xin Xia, and David Lo. 2021. Post2vec: Learning distributed representations of stack overflow posts. *IEEE Transactions on Software Engineering* 48, 9 (2021), 3423–3441.

[223] Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*. 1–10.

[224] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 2017. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 363–376.

[225] Tatsuro Yamada, Shingo Murata, Hiroaki Arie, and Tetsuya Ogata. 2017. Representation learning of logic words by an RNN: from word sequences to robot actions. *Frontiers in Neurorobotics* 11, (2017), 70.

[226] Guanhua Yan, Junchen Lu, Zhan Shu, and Yunus Kucuk. 2017. Exploitmeter: Combining fuzzing with machine learning for automated evaluation of software exploitability. In *Proceedings of the 2017 IEEE Symposium on Privacy-Aware Computing (PAC)*. IEEE, 164–175.

[227] Haoran Yang and Haipeng Cai. 2025. Dissecting real-world cross-language bugs. *Proceedings of the ACM on Software Engineering* 2, FSE (2025), 1270–1292.

[228] Haoran Yang, Weile Lian, Shaowei Wang, and Haipeng Cai. 2023. Demystifying issues, challenges, and solutions for multilingual software development. In *Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1840–1852.

[229] Haoran Yang, Yu Nong, Shaowei Wang, and Haipeng Cai. 2024. Multi-language software development: Issues, challenges, and solutions. *IEEE Transactions on Software Engineering* 50, 3 (2024), 512–533.

[230] Haoran Yang, Yu Nong, Tao Zhang, Xiapu Luo, and Haipeng Cai. 2024. Learning to detect and localize multilingual bugs. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 2190–2213.

[231] Yanming Yang, Xin Xia, David Lo, and John Grundy. 2022. A survey on deep learning for software engineering. *ACM Computing Surveys* 54, 10s (2022), 1–73.

[232] Ziyu Yao, Frank F Xu, Pengcheng Yin, Huan Sun, and Graham Neubig. 2021. Learning structural edits via incremental tree transformations. In *Proceedings of the International Conference on Learning Representations*.

[233] Michihiro Yasunaga and Percy Liang. 2021. Break-It-Fix-It: Unsupervised learning for program repair. In *International Conference on Machine Learning*, PMLR, 11941–11952.

[234] Lantao Yu, Weinan Zhang, Jun Wang, and Yong Yu. 2017. Seqgan: Sequence generative adversarial nets with policy gradient. In *Proceedings of the AAAI Conference on Artificial Intelligence*.

[235] Zeping Yu, Wenxin Zheng, Jiaqi Wang, Qiyi Tang, Sen Nie, and Shi Wu. 2020. Codecmr: Cross-modal retrieval for function-level binary source code matching. *Advances in Neural Information Processing Systems* 33 (2020), 3872–3883.

[236] Mohammed Zagane, Mustapha Kamel Abdi, and Mamdouh Alenezi. 2020. Deep learning for software vulnerabilities detection using code metrics. *IEEE Access* 8 (2020), 74562–74570.

[237] Fengyi Zhang, Bihuan Chen, Rongfan Li, and Xin Peng. 2021. A hybrid code representation learning approach for predicting method names. *Journal of Systems and Software* 180 (2021), 111011.

[238] Jiyang Zhang, Sheena Panthaplackel, Pengyu Nie, Junyi Jessy Li, and Milos Gligoric. 2022. Coditt5: Pretraining for source code and natural language editing. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.

[239] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2020. Retrieval-based neural source code summarization. In *Proceedings of the 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 1385–1397.

[240] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In *Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 783–794.

[241] Xiyu Zhang, Yang Lu, and Ke Shi. 2020. CB-Path2Vec: A cross block path based representation for software defect prediction. In *Proceedings of the 2020 IEEE 6th International Conference on Computer and Communications (ICCC)*. IEEE, 1961–1966.

[242] Wei Zhao, Jianbo Ye, Min Yang, Zeyang Lei, Suofei Zhang, and Zhou Zhao. 2018. Investigating capsule networks with dynamic routing for text classification. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. 3110–3119.

[243] Mengya Zheng, Xingyu Pan, and David Lillis. 2018. CodEX: Source code plagiarism detection based on abstract syntax tree. In *Proceedings of the AICS*. 362–373.

[244] Wei Zheng, Abubakar Omari Abdallah Semasaba, Xiaoxue Wu, Samuel Akwasi Agyemang, Tao Liu, and Yuan Ge. 2021. Representation vs. model: What matters most for source code vulnerability detection. In *Proceedings of the 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 647–653.

[245] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. 10197–10207.

[246] Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. 2021. A syntax-guided edit decoder for neural program repair. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 341–353.

[247] Rui Zhu, Chenglin Li, Di Niu, Hongwen Zhang, and Husam Kinawi. 2018. Android malware detection using large-scale network representation learning. arXiv:1806.04847. Retrieved from https://arxiv.org/abs/1806.04847

[248] Daniel Zügner, Tobias Kirschstein, Michele Catasta, Jure Leskovec, and Stephan Günnemann. 2020. Language-agnostic representation learning of source code from structure and context. In *Proceedings of the International Conference on Learning Representations*.

[249] Daniel Zügner, Tobias Kirschstein, Michele Catasta, Jure Leskovec, and Stephan Günnemann. 2021. Language-agnostic representation learning of source code from structure and context. In *International Conference on Learning Representations*. 1–22.

Received 30 October 2024; revised 31 August 2025; accepted 15 September 2025