CodeGrid: A Grid Representation of Code

Abdoul Kader Kaboré
University of Luxembourg
Luxembourg
abdoulkader.kabore@uni.lu

Earl T. Barr*
University College London
United Kingdom
earl.barr@ucl.ac.uk

Jacques Klein
University of Luxembourg
Luxembourg
jacques.klein@uni.lu

Tegawendé F. Bissyandé
University of Luxembourg
Luxembourg
tegawende.bissyande@uni.lu

ABSTRACT

Code representation is a key step in the application of AI in software engineering. Generic NLP representations are effective but do not exploit all the rich structure inherent to code. Recent work has focused on extracting abstract syntax trees (AST) and integrating their structural information into code representations. These AST-enhanced representations advanced the state of the art and accelerated new applications of AI to software engineering. ASTs, however, neglect important aspects of code structure, notably control and data flow, leaving some potentially relevant code signal unexploited. For example, purely image-based representations perform nearly as well as AST-based representations, despite the fact that they must learn to even recognize tokens, let alone their semantics. This result, from prior work, is strong evidence that these new code representations can still be improved; it also raises the question of just *what signal image-based approaches are exploiting*.

We answer this question. We show that code is *spatial* and exploit this fact to propose CodeGrid, a new representation that embeds tokens into a grid that preserves code layout. Unlike some of the existing state of the art, CodeGrid is agnostic to the downstream task: whether that task is generation or classification, CodeGrid can complement the learning algorithm with spatial signal. For example, we show that CNNs, which are inherently spatially-aware models, can exploit CodeGrid outputs to effectively tackle fundamental software engineering tasks, such as code classification, code clone detection and vulnerability detection. PixelCNN leverages CodeGrid's grid representations to achieve code completion. Through extensive experiments, we validate our spatial code hypothesis, quantifying model performance as we vary the degree to which the representation preserves the grid. To demonstrate its generality, we show that CodeGrid augments models, improving their performance on a range of tasks. On clone detection, CodeGrid improves ASTNN's performance by 3.3% F1 score.

*Some work carried out while a visiting scholar at Google DeepMind.

ISSTA '23, July 17–21, 2023, Seattle, WA, USA
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0221-1/23/07.
https://doi.org/10.1145/3597926.3598141

CCS CONCEPTS

• **Computing methodologies** → *Machine learning*; *Image processing*; • **Software and its engineering** → *Software system structures*; *Software creation and management*; • **General and reference** → **Cross-computing tools and techniques**.

KEYWORDS

Code TypeSetting, Spatial-Aware Neural Network

ACM Reference Format:
Abdoul Kader Kaboré, Earl T. Barr, Jacques Klein, and Tegawendé F. Bissyandé. 2023. CodeGrid: A Grid Representation of Code. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23), July 17–21, 2023, Seattle, WA, USA*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597926.3598141

1 INTRODUCTION

Machine learning (ML) is transforming the digital economy: it has achieved, even exceeded, human performance, on a wide range of tasks, such as natural language translation [41] and image recognition [59, 63], to name a few. It is poised to transform software development [62]. Code is more constrained than the examples of natural phenomena, like natural language text or images, on which ML is usually trained: code obeys an artificial (human-devised), formal grammar; it intermingles that language with natural text in the form of identifiers and comments [13], and it specifies an execution. Its formality and executeability give it unique properties like relatively rigid syntax, unambiguous abstract syntax trees (AST) [35], and control and data flow.

Researchers applying ML to code have been seeking effective representations that exploit the signal inherent to code's relatively greater structure than text [1, 2, 17]. AST-based representations have been prominent [4, 5, 54] and their results on tasks, like code classification and code clone detection [78] as well as code completion [45], have been promising. Unfortunately, learning over AST representations tends to have trouble with large and deep ASTs, on which they can fall prey to the vanishing gradient problem [8]. Despite the wide variety of ML architectures and methods for capturing rich structure applied to code, models using a stream of code tokens as their representation continue to achieve state of the art performance with structure providing only small improvements [16, 24, 33]. In short, *code contains unexploited signal.*

Keller *et al.* [39] recently substantiated this claim, as a side-effect of their work, by exploiting a previously untapped source of code signal. They proposed WySiWiM ('What you See is What it Means'), a novel code representation based on an editor's visual rendering of code. WySiWiM takes screenshots of the code, then trains CNN-based computer vision models for software engineering tasks on those screenshots. WySiWiM's visual representation does not directly capture code's rich structure and mostly contains noise: all the pixel data that does not contain text. Despite these handicaps, WySiWiM matched state of the art (SOTA) performance on code clone detection and code vulnerability prediction tasks. This pioneering work shows the effectiveness of a visual representation of source code and the authors suggested that the success of WySiWiM may, in part, be due to the maturity and effectiveness of computer vision models, but they left open the question of just what signal WySiWiM does exploit.

This paper answers this question. The first step is to consider WySiWiM's construction. Computer vision models, especially CNN models, are built for grid data. WySiWiM trains a state of the art CNN computer vision model on screenshots, which are pixel grids. Keller *et al.*'s WySiWiM is spatial from the ground up. Thus, WySiWiM primarily, and perhaps exclusively, relies on spatial relations in code's layout to achieve its results. It must even learn tokens, if indeed it does learn them, let alone any relations, like type or dependence, among them.

In stark contrast, machine learning approaches to software engineering tasks have, to date, used code representations, whether AST, GNN, or token stream, that destroy code's layout. Transformer models' use of byte-pair encoding does capture whitespace, but not 2d layout [24]. Thus, these representations have sought to capture code's rich structural information and discounted the importance of code layout. Certainly, code layout is irrelevant to a compiler and to the execution of a binary. These facts, however, neglect layout's importance to developers.

In any text, layout, or typesetting, matters because they make it easier to understand. In 'natural' text, paragraphing consolidates ideas into blocks and help the reader to navigate a big block of text [23]. In code, coding conventions determine how code is typeset in a project. Whether imposed by fiat or emergent in a code base [3], projects adopt coding conventions to speed code maintenance [10]. Some coding conventions have been deemed so effective that languages have been written that enforce them: Python's (in)famous whitespace sensitivity is perhaps the most prominent example [19]. Indeed, Hindle *et al.* showed indenting alone is an effective proxy for code complexity [29]. Ranging farther afield, experiments show that words are often recognised by their shape, not attending to their characters [12]. Thus, we speculate that *one reason that the spatiality of code has such strong signal has to do with the fact that it permits developers to make quick, imperfect, yet still frequently useful, assessments of a code snippet's purpose at a glance.* Here, we have in mind System 1 thinking, which is integral to Daniel Kahneman's work [36].

So, in practice, developers often obey coding conventions and carefully typeset their code both horizontally and vertically. Horizontal type setting concerns rules for spacing characters, like operators or delimiters, within a single line; some practitioners will admit to having debated such details as whether to require spaces around operators or permit "`){`" *vs.* "`) {`". Consider Figure 1 to see how vertical alignment can be critical [71]. Figure 1a, on the left,

```
int robert_age = 32;
int annalouise_age = 25;
int bob_age = 250;
int dorothy_age = 56;
```

(a) Standard Coding Style.

```
int      robert_age = 32;
int  annalouise_age = 25;
int         bob_age = 250;
int     dorothy_age = 56;
```

(b) Grid Alignment.

Figure 1: Code is spatial! The shared suffix and the 250 outlier are obscured on the left and jump out on the right.

shows a sequence of assignments, conventionally typeset, using ragged right, obeying typesetting conventions only horizontally, within each line. Figure 1a, on the right, shows the same snippet, spatially typeset. The shared "`_age`" suffix jumps out, as well as the anomalous, almost certainly wrong, "`250`".

This paper. All the aforementioned examples intuitively argue in favor of attempting to explore a signal in code with respect to its layout. The performance of the WySiWiM approach provides an insight that code spatiality may have a much stronger importance than ever considered in the literature of code representation.

Guided by the importance of code layout, we introduce CodeGrid, a new code representation that is spatially-aware and built for consumption by architectures that exploit spatial relations, like CNNs.

Unlike WySiWiM, CodeGrid is aware of tokens. It vectorises tokens by mapping each token to a vector value using three different methods of varying complexity: (1) a naive "**Color Vectorizing**" method that uses code colors for each token, where the color is selected according to the TF-IDF value of the token. Token vectors are constructed with 3-dimensional values that explore the RGB color space where color vectors are ordered to take into account brightness following the approach of Bezryadin *et al.* [9]. The token with the highest average TF-IDF value will be mapped with the vector associated to the highest brightness score; (2) the "**Word2Vec Vectorizing**" that is directly based on a re-trained Word2Vec model using the datasets of our study; and (3) the "**Code2Vec Vectorizing**" that leverages the state of the art pre-trained Code2Vec [5] model built based on AST paths in code snippets. We consider its keyedvectors[1] format, which yields a 300 dimension vector for every token in the training vocabulary.

The code grid. We consider that code is a character-by-line grid; CodeGrid must preserve this coordinate system. In the code grid, each token has a character length. Each token's embedding vector also has length, but in a different dimension, with different semantics than its layout length. There are various ways to map a token's vector into the cells its raw lexeme occupies in the grid: place the vector in one cell and zero vectors into the rest, one could average the vector across the cells or one could repeat the vector in each cell. In this work, we show that the last option works best in practice. Future work would be to employ an architecture that learns its own solution to this mapping problem.

Our experimental results show the value of building a spatial representation that is directly aware of code and its properties.

[1]https://radimrehurek.com/gensim/models/keyedvectors.html

Models trained using CodeGrid consistently outperform WySiWiM. CodeGrid achieves near SOTA results on code classification, vulnerable code prediction, and code completion tasks. This last task is impossible for WySiWiM, because it is unaware of tokens.

Finally, we validate our "code is spatial" hypothesis, using our four tasks. We compare two grid representations: one that utterly destroys code layout and an intermediate one that obscures it against CodeGrid and show that models trained with CodeGrid achieve 2.4% to 11.9% more performance on precision. Our main contributions are as follow:

- We demonstrate that code is spatial — that its layout, both horizontal and vertical, carry useful signal that machine learning models can effectively exploit.
- We introduce CodeGrid, a novel grid representation of code that combines the spatial layout of code with lexical information.
- We show that models built using CodeGrid achieve high performance on code clone detection, code classification, and vulnerability detection (coming within [2..5%] on various measures).

2 APPROACH

CodeGrid takes a code fragment and builds a grid representation. A major step in CodeGrid is the construction of the code grid, where the main idea is to ensure that the coordinate system in the code layout is preserved (Section 2.2). Finally, grid representations of code are fed into spatial-aware neural network architectures to train models for specific software engineering tasks (Section 2.3). We discuss implementation details for replication purposes (Section 2.4).

2.1 Code Is Spatial

We assert that understanding and navigating code involves system 1, which, by definition, brings to bear a collection of quick heuristics for whatever task it is asked to solve. For coding tasks, some of these heuristics exploit code layout, or typesetting. Good code typesetting, we claim, even permits perceiving high-level code semantics visually. This is why code typesetting impacts code readability and why many coding conventions specify detailed typesetting rules. Indeed, in some program languages, notably Python, spatial properties (*e.g.*, indentation) are even part of the syntax. Fundamentally, typesetting implies that program code has a character-based coordinate system. Humans view and edit it with IDES that preserve this character-granular coordinate system. Developers themselves both define and exploit this coordinate system when they add arbitrary spaces, tabs, *etc.* This work rests on the assumption that code has visual semantics that contains strong and exploitable signal. Recast as the principle, this assumption means that any code representation targeting software engineering tasks has two implications:

- *Code representations should preserve the coordinate system (*i.e., *the grid) of code.* Indeed, code has visual semantics as experimentally suggested by the performance achieved with image-based representations of code [39].
- *Models should themselves be spatially-aware to exploit spatial code representations.* Deep Convolutional Neural Networks (CNNs) are appealing in this regard since the convolutional structure in a neural network is built to exploit spatially ordered data [26].

2.2 Representation: Constructing a Code Grid

Given a code sample, CodeGrid unfolds a multi-step process to produce a grid representation where each code token is carefully encoded, in a way that captures its importance for modeling, and mapped to a grid cell. The size of the grid as well as it organisation (i.e., the position of cells) faithfully account for the spatial dimension of code. Figure 2 unfolds, with a running example, the different steps that are carried out to produce the final code grid representation. Note that the colorful visualization that is provided is only for illustration purposes in this paper (i.e., the target output of CodeGrid is not an image, but rather a matrix grid with numerical values).

❶-❷ *Preprocessing and Tokens Extraction.* CodeGrid exclusively targets the representation of source code. Annotations such as code comments are left out from the design of CodeGrid. Its preprocessing step removes comments before splitting code into its constituting tokens. Every single token, including punctuation, is extracted and kept as such, because, like whitespace, punctuation is integral to typsetting. All tokens seen in the overall dataset of code samples constitute the code vocabulary.

❸ *Retrieving the Coordinate System.* When extracting tokens, CodeGrid preserves the information about the spatial positions that they occupy when code is viewed in a text editor. To that end, CodeGrid scans the code using a cursor that moves through the whole code to record the coordinates of each code token. The initial position (top-left) is assigned the origin position $(0, 0)$. The cursor moves from left to right per character and each encountered token t is assigned the coordinates (x_t, y_t) of its first character.

Because CodeGrid faithfully follows the spatial properties of each code sample, the inferred grid has a variable size. Concretely, given a sample C_i, the grid size will have a height H_i and a width W_i that are derived as follows:

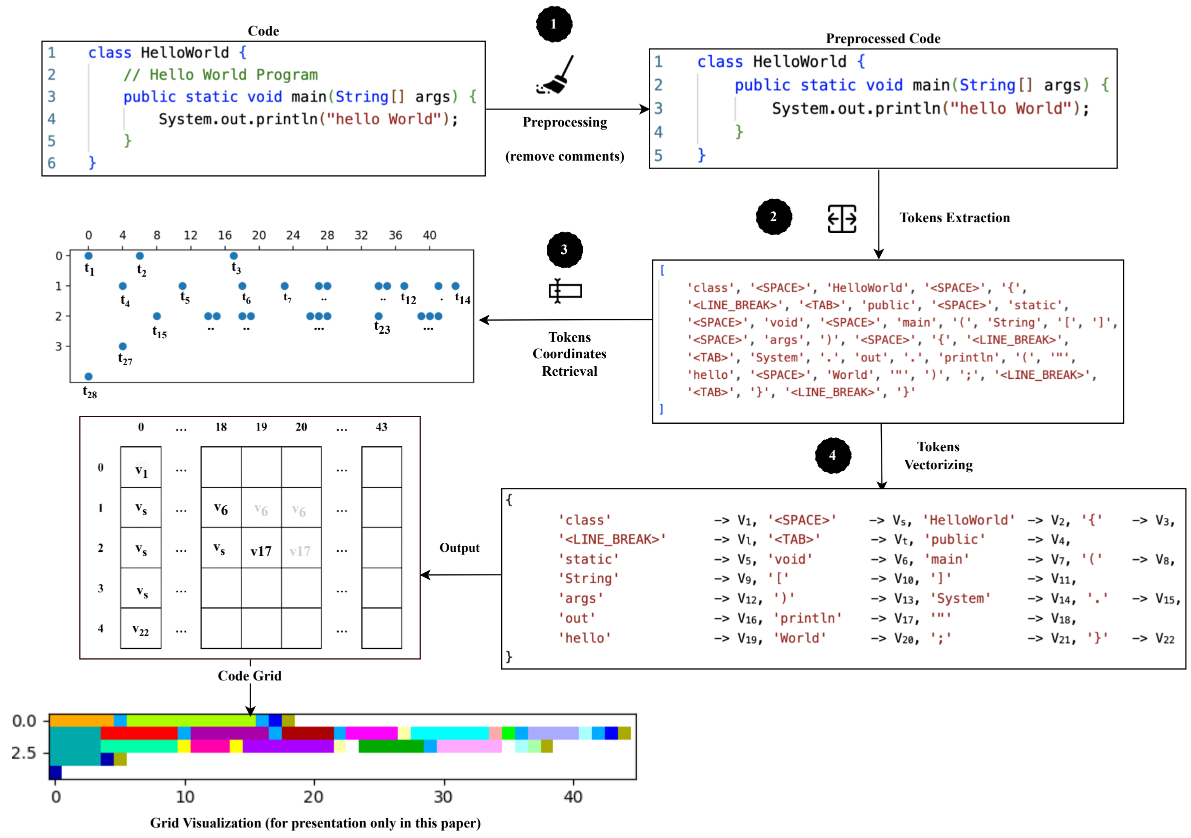
$$H_i = \#\, of\ lines\ in\ C_i \tag{1}$$

$$W_i = \max_{j \in [1 \dots H_i]} (len(line_j)) \tag{2}$$

where $line_j$ is the j^{th} line of code and len is a function that counts the number of characters (including spaces).

❹ *Vectorizing Tokens.* CodeGrid constructs its representation by vectorizing character-based tokens. To realize CodeGrid, we considered and experimented with three different vectorizations: word2vec, code2vec, and a novel colorization technique. To vectorize via Word2Vec, we re-trained a Word2Vec [53] model on our study's datasets (Section 3.1) and used its embeddings. Code2Vec [5] is a more recent, code-focused approach that learns token representations from ASTs. For CodeGrid, we directly used its keyedvectors format, which maps every token to a 128-dimension vector.

In addition to these two techniques, we devised a simple, interpretable heuristic that "colorizes" tokens by mapping them to RGB color vectors. Chen *et al.* [15] introduced a heuristic code vectorization that replaces each character with its ASCII decimal value. This heuristic is not immediately suitable to our task: CodeGrid



Code

```
class HelloWorld {
    // Hello World Program
    public static void main(String[] args) {
        System.out.println("hello World");
    }
}
```

Preprocessed Code

```
class HelloWorld {
    public static void main(String[] args) {
        System.out.println("hello World");
    }
}
```

```
[
    'class', '<SPACE>', 'HelloWorld', '<SPACE>', '{',
    '<LINE_BREAK>', '<TAB>', 'public', '<SPACE>', 'static',
    '<SPACE>', 'void', '<SPACE>', 'main', '(', 'String', '[', ']',
    '<SPACE>', 'args', ')', '<SPACE>', '{', '<LINE_BREAK>',
    '<TAB>', 'System', '.', 'out', '.', 'println', '(', '"',
    'hello', '<SPACE>', 'World', '"', ')', ';', '<LINE_BREAK>',
    '<TAB>', '}', '<LINE_BREAK>', '}'
]
```

```
{
    'class'          -> V1,  '<SPACE>'   -> Vs,  'HelloWorld'  -> V2,  '{'  -> V3,
    '<LINE_BREAK>'   -> Vl,  '<TAB>'     -> Vt,  'public'      -> V4,
    'static'         -> V5,  'void'      -> V6,  'main'        -> V7,  '('  -> V8,
    'String'         -> V9,  '['         -> V10, ']'           -> V11,
    'args'           -> V12, ')'         -> V13, 'System'      -> V14, '.'  -> V15,
    'out'            -> V16, 'println'   -> V17, '"'           -> V18,
    'hello'          -> V19, 'World'     -> V20, ';'           -> V21, '}'  -> V22
}
```

Figure 2: CodeGrid Representation Construction.

works on tokens, not characters, so its vocabulary is much larger (> 128). Inspired by this ASCII heuristic, we designed a naïve "Color Vectorizing" technique. As already mentioned, it maps tokens to RGB color codes. This encoding can handle a vocabulary of over 16 million tokens. As a reference point, our experimental datasets include 693 054 distinct tokens.

We turned to Information Retrieval (IR) to devise a token encoding that captures token importance. The Term Frequency-Inverse Document Frequency (TF-IDF) [64] is an prominent IR technique used in text vectorization algorithms [69]. Concretely, for each token in the vocabulary, CodeGrid computes the average of its TF-IDF values across the entire corpus following the formula in Equation (3) below.

$$tf(t,c) = \frac{n_{t,c}}{\sum_{k \in c} n_{k,c}} \tag{3}$$

$$idf(t,C) = \log\left(\frac{|C|}{|\{c \in C \mid t \in c\}|}\right) \tag{4}$$

$$tfidf(t,c,C) = tf(t,c) \cdot idf(t,C) \tag{5}$$

$$imp(t) = \frac{1}{|C|}\left(\sum_{c \in C} tfidf(t,c,C)\right) \tag{6}$$

where t is a token in a code sample c, $n_{t,c}$ is the number of occurrences (multiplicity) of t in c, and $n_{k,c}$ is the number of tokens (including t) in c.

To build "color vectors", we first sort all the tokens in the corpus by importance, then sort all RGB values by brightness, using a heuristic proposed by Bezryadin *et al.* [9]. Then, we map each token t to an RGB color vector c, where $imp(t_i) > imp(t_j)$ implies the brightness of c_i is greater than that of c_j.

With these token vectors, we convert the code layout into a grid that associates each cell a vector representing either a code token or a space. As core to typesetting, CodeGrid must explicitly encode spaces. The space character vector, denoted v_s, is *fresh*, *i.e.* different from any assigned token vector and is selected by picking a random vector among the unassigned vectors after vectorizing tokens. When placing a vector in the grid whose raw token has n characters, we place the vector at the coordinate position that corresponds to token's first character. If the cells corresponding to a token's remaining characters are left empty, the grid will be sparse, which challenges learning [22]. Therefore, we copy a token's vector value in all grid cells its token occupies. We refer the reader to Section 3.8 for a discussion on the validity of this design choice.

OOV tokens. For all three token encodings, we represent out-of-vocabulary tokens with an <UNK> token, whose value is fresh *w.r.t.* the vocabulary. Code2Vec already supported an <UNK> token, so we reused it. We manually added <UNK> to the Word2Vec and Color vectorizers.

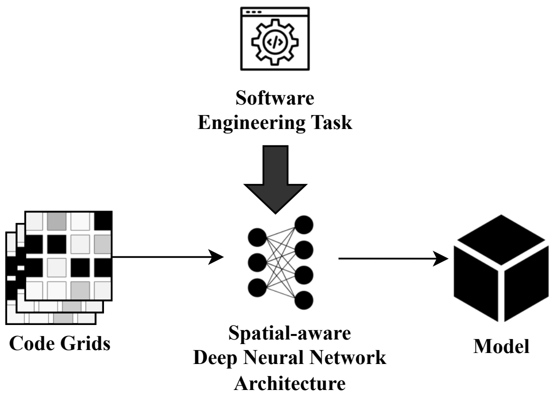

Software Engineering Task
Code Grids
Spatial-aware Deep Neural Network Architecture
Model

Figure 3: Learning on Grids

2.3 Learning on Grids

Models must be able to input and exploit CodeGrid's grid representations of code to produce spatially aware models as shown in Figure 3. Convolutional Neural Networks (CNNs) [44] match this criterion: composed of multiple building blocks, such as convolution layers, pooling layers, and fully connected layers, CNNs are designed to automatically and adaptively learn spatial hierarchies of features. Widely used in the context of computer vision [75], CNN models generally take as input digital images, which are in reality two-dimensional (2D) grids containing pixel values. A CNN-based model scans all the cells in these grids in order to extract any relevant features [77] while successfully capturing the spatial dependencies in an image through the application of relevant filters.

Researchers and practitioners have achieved exceptional performance using CNNs on a variety of tasks and are now popular across the AI community. Indeed, in image processing, CNNs effectively use adjacent pixels information to effectively downsample the image while preserving the spatial interactions among pixels. This way, CNNs can assign importance, which is characterized by learnable weights and biases, to various aspects/objects in the image and be able to differentiate one from another.

Our tool, CodeGrid, accounts for the spatial dimension of code by placing each token into the grid cell it occupies in a character-granular view of the source code. CNNs are therefore well suited for learning characteristics from our code representations. Given a code corpus and a task at hand (labels for classification), we feed the set of grids produced by CodeGrid as input data to a CNN-based architecture and train a model.

2.4 Implementation

We implemented CodeGrid in Python. We used Pandas and Numpy libraries to manipulate raw data and operate on matrices to construct the code grid. For manipulating the code layout, we used "GNU indent"[2], a program re-formats C to obey a coding convention. To compute TF-IDF, we used scikit-learn; to compute brightness, we used the nltk libraries. We build on Gensim [66], PyTorch [40] and PixelCNN [61] for training the Word2Vec model and conducting our deep learning experiments.

[2]https://www.gnu.org/software/indent/

Table 1: Summary of the datasets

Task	Total # of code samples	# of code samples in test set	References
Code Clone Detection	40,000	8,000	[39, 70, 78]
Code Classification	52,000	10,400	[39, 54, 55, 78]
Vulnerability Prediction	420,627	84,126	[48, 56, 57]
Code Completion	874,590	174,918	[4]

3 EVALUATION

We evaluate CodeGrid on four typical tasks that are widely used in the literature of AI applications in software engineering. Our experimental setup (Section 3.1) is guided by the requirements of the considered tasks and our validation is based on performance metrics that we have identified for comparison against strong baselines related to source code representation for learning-based software engineering (Sections 3.3-3.6). Subsequently, we propose to validate the "*code is spatial*" hypothesis by artificially manipulating the layout of the code snippets in our study dataset (Section 3.7). Finally, we present experimental results that highlight the validity of our design choices in the grid cell filling method (Section 3.8) and assess the token vectorizing techniques (Section 3.9).

3.1 Experimental Setup

Benchmarks. We use benchmarks that have been proposed in the literature to evaluate the performance AI models for software engineering tasks. Table 1 summarizes the size these benchmarks, along with references to state of the art works where they have been exploited. The tasks that we consider in our evaluation have been chosen because they are seminal software engineering tasks that have already been tackled in the AI for software engineering literature. Notably, these are code clone detection, code classification, vulnerable code detection and code completion tasks. We discuss, later in this section, each task in more detail.

Baselines. We compare our approach (*i.e.*, the CodeGrid code representations associated with a CNN-based architecture) to a variety of approaches. Concretely, we put a significant effort in identifying, for each task, strong baselines from the recent literature. Here, we compare our results to performance results on the same benchmarks reported by the authors in their published work, instead of reproducing their work. Our goal is to validate the spatial signal in code and show how it can augment existing models, not introduce a new technique or conduct a replication study. We acknowledge the threat this decision poses to our findings, due to variations in hardware, tool configuration, *etc.*. We note, however, that we experimentally confirmed on uniformly selected samples that the replication packages provided by the authors lead to very similar results as those reported in the papers in our environment.

- *SLM* [4] is a structural language model of code that has achieved high performance on code completion.
- *TBCNN* [54] implements tree-based convolutional neural networks to model programming languages where program vectors are explicitly learned from the AST tree features.
- *ASTNN* [78] develops a specialized neural network model for scaling to large ASTs.
- *WySiWiM* [39], applies represents code as images for learning representations that are exploited on software engineering tasks.

- *SySeVR* [48] is a deep learning-based approach to vulnerability detection that learns to identify program regions that are likely to contain vulnerabilities.

Our comparison focuses, for each baseline, on the task on which their original implementations have been shown to be successful in their original publication. We use *Checkmarx* as a baseline in the vulnerability detection task although it is not deep learning based, because it appears in the literature as a key baseline. As already noted, Table 3, Table 4, Table 5 and Table 6 contain data, and make comparisons, based on the published IR results of the cited prior work, as we do not intend to compare the inference time.

Architecture. We select CNN-based architectures for our experiments. Depending on the tasks, we use different CNN-based architectures to fit with our constraints. For tasks involving the training of a classifier, we rely on Deep Residual Network (ResNet) [28]. This architecture has been demonstrated to be one of the best CNN-based architectures for image classification [18].

For the code completion task, however, we require a generative architecture. We therefore selected PixelCNN [72], which is a model used for conditional image generation, i.e. predicting a pixel in an image given the previous pixels. We use PixelCNN to predict missing cell values in the grid.

Although pre-trained versions of these architectures exist, they already pre-set the grid sizes. Therefore, we adapt both ResNet and PixelCNN architectures to handle larger dimension vectors by replacing their input layers with new input layers that fit to CodeGrid grids' shape, before retraining them with our datasets.

Training. We trained the models for each task by using the appropriate benchmark's dataset (Table 1). Because the tasks are independent, CodeGrid handles each dataset independently. Before using the CodeGrid grids as input to each model, we handle their varying sizes by considering the highest width and height from each dataset and then applying padding to this size for each sample. For our evaluations, we divided the dataset into training (75%), validation (5%) and testing (20%) following prior work (e.g., [78]) to which we compare against. With ResNet architecture, the training is conducted over 150 epochs. With PixelCNN, the training is conducted over 200 epochs.

Metrics. We measure classification performance using the classical metrics of Accuracy, Precision, Recall and F1 score. For the generative task of code completion, we follow Alon *et al.* [4] and report the *exact match accuracy at k*, which indicates the number of relevant tokens among the first *k* predictions. As for the evaluation of the SLM [4] approach, we use 1 and 5 for *k* values.

CodeGrid **Evaluation objective**: By assessing the performance on a variety of tasks, we seek to demonstrate that:

(1) with CodeGrid, we capture a strong signal in code that is useful for learning to solve software engineering tasks;
(2) the learned representations are effective for diverse tasks, including generative ones;
(3) CodeGrid combined with Code2Vec as cell values can achieve high performance on several tasks.

Table 2: CodeGrid Ablation Study

(a) Classification Tasks

Task	Vectorizing Method	Accuracy	Precision	Recall	F1 score
Code Clone Detection	Word2Vec	-	98.4	95.7	97.0
	Code2Vec	-	99.6	96.1	97.8
	Color	-	95.9	94.2	95.0
Code Classification	Word2Vec	97.0	97.0	97.0	97.0
	Code2Vec	97.2	97.2	97.2	97.2
	Color	91.8	91.8	91.8	91.8
Vulnerability Detection	Word2Vec	96.2	93.8	-	90.7
	Code2Vec	98.4	94.9	-	92.9
	Color	93.8	90.7	-	92.2

(b) Code Completion Task

	acc@k (exact-match)	
Vectorizing Method	@1	@5
CodeGrid + Word2Vec	9.07	15.80
CodeGrid + Code2Vec	7.54	15.09
CodeGrid + Color Vectorizer	14.91	22.70

3.2 CodeGrid Ablation Study

CodeGrid must vectorize code tokens (Figure 2, stage 4). We consider three different tokens vectorizer methods (Section 2): the color-based vectorizing method which maps each token to a 3D vector representing a color in the RGB range, the Word2Vec-based vectorizing method and the Code2Vec-based vectorizing method which considers the keyed-format of Code2Vec pre-trained model. In this first experiment, we consider an ablation study to compare those three token vectorizer methods.

The Experiment. For this ablation study, we consider all the classifications tasks and the code completion task described in the sections 3.3, 3.4 and 3.5 and 3.6 respectively.

The Results. Table 2a demonstrates that, on the code classification, CodeGrid performs best when considering grid's cells are filled using Code2Vec's embeddings. For the code completion task, the "Color Vectorizing" method outperforms the "Word2 Vectorizing" and "Code2Vec Vectorizing" methods. Overall, CodeGrid performs best on three of the four tasks when using the "Code2Vec Vectorizing" method. Thus, in the remaining sections, we use CodeGrid to refer to CodeGrid Code2Vec token embeddings; all subsequent comparisons with the baselines are against this variant.

3.3 CodeGrid on Clone Detection

While the definition of *Code clones* varies across the literature where researchers consider different levels of similarity at the lexical, syntactic and semantic levels, the research direction on clone detection is fairly active [65, 67], with several applications in plagiarism detection, origin analysis, program understanding, code compacting, malicious code detection, etc. In recent years, machine learning techniques have been applied towards improving performance of code clone detection tools [74, 76, 78].

The Task Given a pair of code samples, the clone detection task consists in determining whether they are similar (at different levels: lexical, syntactic or semantic). The literature distinguishes several types of Cloning, including Type-1 (with identical code fragments, except for differences in whitespace, layout, and comments), Type-2 (with identical code fragments, except for differences in identifier names and literal values, in addition to Type-1 clone differences),

```java
public static boolean isPalindrome(String original) {
    String reverse = "";
    int length = original.length();
    for (int i = length - 1; i >= 0; i--)
        reverse = reverse + original.charAt(i);
    if (original.equals(reverse))
        return true;
    else
        return false;
}
```

(a) Code Fragment 2

```java
public static boolean isPalindrome(String string) {
    if (string.length() == 0)
        return true;
    int limit = string.length() / 2;
    for (int forward = 0, backward = string.length() - 1; forward < limit; forward++, backward--)
        if (string.charAt(forward) != string.charAt(backward))
            return false;
    return true;
}
```

(b) Code Fragment 1

Figure 4: Type-4 Code clone pair identified with CodeGrid

Type-3 (with syntactically similar code fragments that differ at the statement level) and Type-4 (with syntactically dissimilar code fragments that implement the same functionality). The latter are the most challenging to detect and have been scarcely investigated in the literature in contrast with former types on which high accuracy was achieved by lexical and syntax-based clone detectors such as CCFinder [37], Deckard [34], SourcererCC [68].

BigCloneBench (BCB) [70] is widely used in the community for assessing clone detection tools, as it includes a mix of different types of clones. BCB consists of ~8 million clone pairs mined from 25 000 open source Java projects in the IJaDataset-2.0 [6] (3 million source code files and 250 millions of lines of code). Nevertheless, to ensure a direct and unbiased comparison with our baselines, we evaluate on the same subset of 40k Type-4 clone pairs that ASTNN and WySiWiM were evaluated on. Figure 4 illustrates an example of Type-4 cloned methods: despite being named similarly, the significant differences in method bodies makes many token-based approaches fail to identify the cloning.

The Experiment. We train a binary classifier which yields a probability that two code fragments given as inputs constitute a clone pair. We keep to the default threshold probability of 0.5 for the classification decision. To include negative samples in the test and training sets, we randomly shuffle known clone pairs in the dataset and create new non-clone pairs.

The Results. Table 3 summarizes the performance results of a ResNet model trained with CodeGrid representations against the baselines. While ASTNN achieves higher precision by 0.3 percentage points, CodeGrid improves it by ~8 percentage points in terms of recall. Furthermore, CodeGrid combined with any token vectorizing method yields balanced scores in terms of precision and recall compared to ASTNN.

Table 3: Performances on Code Clone Detection.

Method	Precision	Recall	F1
ASTNN	99.8	88.3	93.7
WySiWiM	95.4	94.3	94.8
CodeGrid	99.6	96.1	97.8

3.4 CodeGrid on Code Classification

Reuse and maintenance activities in software engineering often require some degree of *comprehension* of what program code is doing, *i.e.*, what algorithm/functionality are implemented. To that end, code classification has been studied as an important software engineering task for benchmarking AI models targeting software engineering. Note that in some work, code classification is referred to as *algorithm classification* [7] and promoted as an important research direction in guided programming where developers are provided with alternate code suggestions based on the comprehension of what the developer code is about.

The Task. Given a sample method, the code classification task consists in predicting the label reflecting the implemented functionality. We use a dataset [55] containing C programs written by 500 students to answer 104 programming questions on OpenJudge[3]. Each question addresses a specific functionality, such as implementing bubble sort. For each question, OpenJudge verified 500 solutions as correct and thus share the same label. Figure 5 illustrates two examples of such correct solutions.

The Experiment. We train a multi-class classifier where CodeGrid representations are fed to a ResNet architecture. The resulting model yields scores for the probability that a given code sample is related to one of the 104 functionalities in the benchmark. The final classification selects the functionality for which the probability is the highest. For this multi-class classification task, we use micro-average[4] precision when computing the model scores.

The Results. Table 4 reports the performance results. ASTNN and TBCNN authors only reported Accuracy metrics on this benchmark. CodeGrid enables the ResNet-based model to achieve similar performance to the baselines (by 1% from ASTNN), while outperforming the WySiWiM baseline by about 3 percentage points. This result suggests that CodeGrid is effective in eliminating the pixel noise introduced by WySiWiM (which takes screenshots and also feeds them into a ResNet architecture), instead of extracting the most relevant information in the spatial layout of code.

CodeGrid also improves over TBCNN, which was the first model to be applied to the OpenJudge dataset. Finally, we note that the ResNet architecture trained with CodeGrid representations yields balanced precision and recall: both metrics provide a performance score at ~92%, suggesting a good trade-off between avoiding false positives and recalling labels for most samples.

[3]http://poj.openjudge.cn/
[4]https://scikit-learn.org/stable/modules/model_evaluation.html#multiclass-and-multilabel-classification

```
void main() {
  int i, n, max1 = 0,
       max2 = 0, a[100];
  scanf("%d", &n);
  for(i=0; i<n; i++) {
    scanf("%d", &a[i]);
    if(max1 < a[i])
      max1 = a[i];
  }
  printf("\n%d\n", max1);
  for(i=0; i<n; i++) {
    if(max2 < a[i] &&
         max1 > a[i])
      max2 = a[i];
  }
  printf("%d\n", max2);
}
```

(a) Program 1

```
main() {
  int i, n, a[100], max1
       , max2, temp;
  scanf("%d", &n);
  for(i=0; i<n; i++)
    scanf("%d", &a[i]);
  max1 = 0;
  for(i=1; i<n; i++)
    if(a[max1] < a[i])
      max1 = i;
  temp = a[0];
  a[0] = a[max1];
  a[max1] = temp;
  max2 = 1;
  for(i=2; i<n; i++)
    if(a[max2] < a[i])
      max2 = i;
  temp = a[1];
  a[1] = a[max2];
  a[max2] = temp;
  printf("\n%d", a[0]);
  printf("\n%d", a[1]);
}
```

(b) Program 2

Figure 5: Two student programs submitted to "*find and display the 2 largest numbers in an array*"

Table 4: Performance on Code Classification

Approach	Accuracy	Precision	F1
ASTNN	98.2	-	-
TBCNN	94.0	-	-
WySiWiM	89.7	85.7	86.3
CodeGrid	97.2	97.2	97.2

3.5 CodeGrid on Vulnerability Detection

Beyond code clone detection and code classification, software engineers must constantly be on the lookout for vulnerabilities in their code. These are bugs that expose the software system to security breaches. Their detection however is generally challenged by the lack of exploits (i.e., vulnerability-revealing test cases). To cope with this gap, the literature proposes approaches that statically analyse the code and leverages similarity-based [42, 47] or pattern-based techniques [14, 27, 30, 58] to predict the presence of vulnerabilities. The former techniques are rather ineffective [49] while the latter techniques are costly and cannot scale because they often require extensive human expert intervention to identify vulnerable code patterns, or to engineer features for prediction learning. In recent years, deep neural networks based approaches such as SySeVR [48] have been proposed for the task of predicting vulnerable code.

The Task. Given a code sample, the vulnerability detection task consists in predicting whether it is vulnerable or not. We used the evaluation dataset collected by SySeVR. It includes samples from two sources: the National Vulnerability Database (NVD) [56], which includes vulnerabilities from production software, and the Software Assurance Reference Dataset (SARD) [57], which includes vulnerabilities from production, synthetic and academic software.

Table 5: Performance on Vulnerability Detection

Method	Accuracy	Precision	F1
SySeVR	98.0	90.8	92.6
Checkmarx	72.9	30.9	36.1
CodeGrid	98.4	94.9	92.9

Each sample in the final dataset is labeled either as "good" (i.e. having no vulnerabilities) or "bad" (i.e. having vulnerabilities).

The Experiment. We train a binary classifier by feeding CodeGrid representations of code samples along with their label into a ResNet architecture. The included vulnerabilities span several issues related to pointer or array usage, arithmetic expressions, and API calls.

The Results. Table 5 reports the experimental results. We also include the performance reported by SySeVR authors on their own tool as well on the Checkmarx commercial tool. CodeGrid representations fed to a ResNet led to a performance improvement over SySeVR for the vulnerability detection task. The gap is particularly large in terms of Precision (by 4%).

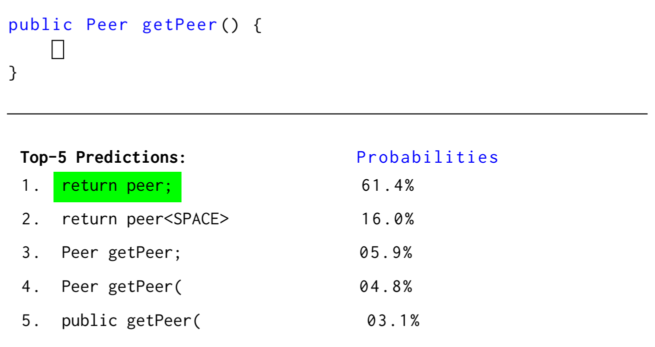
3.6 CodeGrid on Code Completion

In integrated development environments (IDEs), code completion is a feature that is highly sought by developers. It is about generating missing code at a specific location by using the surrounding code as context. It helps to speed up coding and can contribute to reducing programming mistakes. Actually, more generally, code completion can be seen as first step towards program synthesis, which remains one the oldest and most challenging problem in computer science [73]. It is thus an extremely challenging task that has uses even in other research directions of software engineering, including automatic program repair.

The Task. Given an incomplete program with some missing part, i.e., a hole, the code completion task consists in predicting the missing token(s). We rely on the java-small dataset released by Alon *et al.* [4], which includes code samples from 11 Java projects collected from GitHub.

The Experiment. We train a code completion engine by leveraging PixelCNN [72], a generative model targeting images. PixelCNN was initially proposed to generate image variations (e.g., portraits of the same person with different facial expressions) based on an image variant. By feeding PixelCNN with the CodeGrid representations of code, we hypothesize that the architecture can learn to produce variants. Thus, when given a partial grid, PixelCNN can suggest the next cell value, which can be automatically mapped to a single token in the vocabulary. For this experiment, we considered the default parameters in PixelCNN implementation, and fixed the size of the grid to a 32x32 matrix. Code completion is therefore performed by iteratively predicting missing grid cell values until a filled cell is encountered or until we hit the boundary of the grid. Figure 6 illustrates an example case where our model suggests several possibilities among which the exact-match of the missing code is predicting with the top-5 predictions.

The Results. Table 6 reports experimental results achieved with CodeGrid (using PixelCNN) on the code completion task. We also



```
public Peer getPeer() {
    ▯
}
```

```
Top-5 Predictions:              Probabilities
1.  return peer;                 61.4%
2.  return peer<SPACE>           16.0%
3.  Peer getPeer;                05.9%
4.  Peer getPeer(                04.8%
5.  public getPeer(               03.1%
```

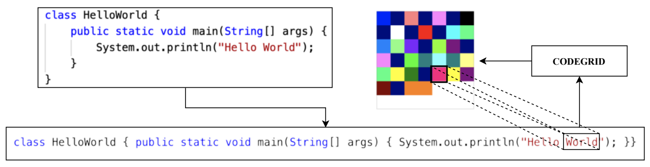
Figure 6: Example of Code Completion achieved with PixelCNN based on code representations yielded by CodeGrid

Table 6: Performance on Code Completion

	acc@k (exact-match)	
Approach/Model	@1	@5
SLM	18.04	24.83
CodeGrid	7.54	15.09

provide results achieved by SLM, on the same dataset. To facilitate comparisons, we use the acc@k (accuracy at k) metric proposed by Alon *et al.* [4], which seeks to check whether the model can produce the exact-match. The experimental data show that the PixelCNN model trained on CodeGrid representations achieves low performances compared to SLM in both measures. acc@1 and acc@5. This suggests that a grid representation of code using Code2Vec token's embeddings as cell values is not suitable for such a task. Despite these results, we also note that while our natural reference is WySiWiM because it inspired our principled approach to account for the spatial dimension of code, WySiWiM is not aware of code tokens and therefore cannot even be applied to code completion task, unlike CodeGrid. Finally, the results of our ablation study on the code completion task (Table 2b) show that the "Color Vectorizing" method provides substantially better results compared to the "Word2Vec Vectorizing" and the "Code2Vec Vectorizing" methods. The code completion task is a token generation which is effectively a multi-class prediction problem. Code2vec produces much larger vectors, so we speculate that the color vectors are effectively performing a rank reduction increasing the probability weight on more likely tokens. And our results suggest that this reduction works particularly well for code completion where it is safer relative to natural language to ignore the long tail of tokens, which include variables that are out of scope.

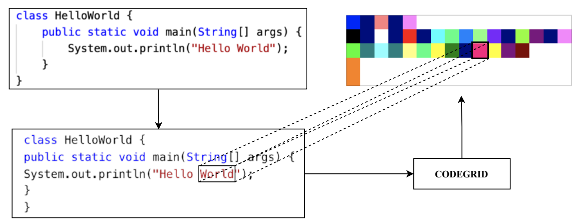
Findings: Trained on CNN architectures, CodeGrid code representations yields near the baseline's performance on all three classification tasks. The experimental results show that the code spatiality signal captured by CodeGrid (with the Code2Vec token's embeddings) is indeed useful for learning to solve software engineering tasks.



```
class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World");
    }
}
```

```
class HelloWorld { public static void main(String[] args) { System.out.println("Hello World"); }}
```

Figure 7: Constructing a Destroyed-Typesetting Grid



```
class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World");
    }
}
```

```
class HelloWorld {
public static void main(String[] args) {
System.out.println("Hello World");
}
}
```

Figure 8: Constructing a Left-Aligned Grid

3.7 Validating That Code Is Spatial

CodeGrid is based on a fundamental assertion, which we made, that code is spatial (2.1). Our representation design follows this assertion and strives to faithfully capture code layout. When employed with CNN-based architectures CodeGrid representations achieve near the considered baseline performance on a variety of software engineering tasks where AI solutions are increasingly sought. We propose to further validate our initial assertion through adapted ablation studies.

The experimental objective is to assess to what extent being faithful to the developer code layout is important. Our assumption is that if the coordinate system is not preserved as is, the resulting representations should lead to less performance on the software engineering tasks. We consider two variations for the code layout, where the typesetting is entirely destroyed (i.e., loss of horizontal and vertical spatial information) or the code is simply left-justified (i.e., loss of horizontal information only).

Untypeset Grid. Our first variation emulates the case where *typesetting does not matter*. For this variant, we destroy both the spatial dimension of code — both horizontal and vertical — when constructing the grid, as illustrated in Figure 7. To that end, we remove all spatial properties that are not necessary for execution: we drop line breaks and convert tabs into single spaces. After flattening the code into a single line, we build a square matrix whose boundaries no longer map to the initial code lines. We set the grid size based on the following equation:

$$grid_{with} = grid_{height} = ceil(sqrt(N)) \tag{7}$$

where N is the code snippet length including all white-spaces.

Left-Aligned Grid. Our second variation emulates the case where the spatial dimension is only partially destroyed. For example, we lose the horizontal dimension information (*e.g.*, indentation) but retain vertical dimension (*e.g.* line breaks are kept, but all tabs are removed to left-align the code as in Figure 8). In this configuration, the grid size is constrained by the number of lines of code and the number of tokens and spaces in the longest line of code.

Table 7: Comparing performance with different variations of the code grid representation

Task	Code Representation	Accuracy	Precision	Recall	F1
Code Classification	Untypeset Grid	80.8	80.8	80.8	80.8
	Left-Aligned Grid	85.8	85.8	85.8	85.8
	CodeGrid	97.2	97.2	97.2	97.2
Code Clone Detection	Untypeset Grid	89.2	89.1	85.6	87.3
	Left-Aligned Grid	93.9	92.0	97.2	94.5
	CodeGrid	99.6	99.8	96.1	97.9
Vulnerability Prediction	Untypeset Grid	88.7	88.5	88.6	88.5
	Left-Aligned Grid	88.5	88.4	88.5	88.4
	CodeGrid	98.4	94.9	91.0	92.9

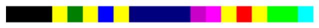
Experiments. This ablation study builds the grid from code tokens as usual. We consider both typesetting-destroying prepossessing variations described above. Due to computational costs, we do not consider experimentations on the code completion task.
Results. Table 7 presents the experimental results with the code grid representation variations. Destroying the typesetting entirely leads to the largest performance gap (up to 16% Accuracy in code classification) compared to the designed CodeGrid representation. Models trained with left-aligned grid representations also underperform compared to CodeGrid, although they generally overperform models trained with untypeset grids. These results clearly confirm that typesetting matters in code representation and that code is indeed spatial.

The experimental results confirm that preserving typesetting in code representations improves the performance of AI architectures in software engineering tasks. The hypothesis that *code is spatial* is thus validated.

3.8 Variations in Grid Cell Filling

As introduced in Section 2, the grid cells in CodeGrid representations are filled with the objective of keeping the coordinate system: we implement two vector filling mechanisms:
◗ *Exact-copy.* In this mechanism, which is the one adopted for CodeGrid, all grid cells spanning the character positions of the token are filled with the same vector value associated to the token. Figure 9a illustrates what a grid row (representing a line of code) would look like. We remind the reader that the final representations are not the illustrative images shown for readability in this paper. Instead, the representations are grids (high-dimensional matrices) with token vector values.
◗ *First cell only.* In this alternate mechanism, only the cell associated to the position of the first character is filled with the value of the token vector. The token's remaining default to the grid's cell initialization value). Figure 9b illustrates what a grid row would look like. The grid is therefore sparse.

We apply variants of CodeGrid representations considering both mechanisms on the code classification task. Experimental results show that the variant yielding sparse grids substantially underperforms against our initial design of CodeGrid: f1-score drops by 12%.



(a) All the token cells are filled

(b) Fist cell of each token is filled

Figure 9: Examples of Variations in Grid Cell Filling with the statement `"int r = rand() % 20;"`

3.9 Grid Construction Methods

The main contribution of our work is its principled demonstration of the signal inherent to the spatial dimension of code via a spatially aware code representation. We show that AI models can benefit from this representation. Our design focuses on the layout information while retaining tokens and their spatial relations. We devise an ad-hoc heuristic of importance based on the TF-IDF algorithm to map tokens to vectors in the RGB color-coding scheme (using brightness for sorting). We also consider the use of other methods (the use a pre-trained Code2Vec models as well as Word2Vec newly trained model) for inferring the tokens vectors. Overall results on Table 3, Table 4 and Table 5 suggest that our Code2Vec coding scheme is generally more effective than Word2Vec, which is also more effective than Color vectorizing. This performance order reflects the richness of information carried out by the token embeddings: on the one end of the spectrum, Color-based embeddings carry no semantics; on the other end Code2Vec embeddings are built based on AST paths of code snippets in the dataset.

4 RELATED WORK

Our work is related to various research directions in the literature. We overview the most recent works and discuss how CodeGrid contributes to the domains of code representation learning for automated problem-solving in software engineering.

4.1 Code Representation Learning in General

Efficient representation of source code is essential for various software engineering tasks. While initial representations were built on architectures such as Word2Vec [52], Doc2Vec [43] or BERT [21], either by using their pre-trained models or retraining them (e.g., CodeBERT [24]), there is a momuntum of research works that propose specialized architectures to statically deep-learn structural and semantic representations of code. Code2vec [5] is a prominent such example that develops a path-based approach with an attention-based neural network to learn code embeddings. The Tree-based Convolutional Neural Network (TBCNN) approach proposed by Mou *et al.* [54] was also successfully applied in various tasks. Similarly, ASTNN *et al.* [78] later managed to outperform TBCNN and other baselines in several tasks.
➥ Unlike the aforementioned approaches which are known as embeddings, because they are produced based on learning through deep neural networks, CodeGrid builds a code grid representation that preserves the coordinate system that is inherent to code and by vectorizing code tokens via a simple and transparent mapping mechanism where token importance in the corpus is encoded. Both approaches can be complementary, where token vectors can be

learned with specialized architectures to capture even more contextual information, and the spatial dimension, whose importance was demonstrated in our experiments, is preserved by CodeGrid.

4.2 Image-Based Representations of Code

Successful applications of AI in software engineering has attracted interest of researchers from various disciplines. For example, in the malware detection community, image-based representations of applications are already considered to exploit the capabilities of computer vision algorithms in order to learn to discriminate malware from goodware [20, 32]. In software engineering, there are increasing attempts in this respect. Chen *et al.* [15] have proposed an approach where characters of source code are converted into pixels whose colors are decided based on the ASCII decimal value of each character. The resulting pixels are then arranged in a matrix, which is viewed as an image. Besides the fact that tokens are lost in this translation, the construction of the matrix does not take into account the developer typesetting. More recently, Keller *et al.* [39] proposed to consider code screenshots as inputs to visual representation learning. While the authors indeed capture the spatial dimension of code, the fact is that the representation made with raw pixels introduces a significant amount of noise.
➥ While visual representations of code experimentally provide appealing performance, we have shown that CodeGrid representations captures better the underlying signal related to the spatial dimension of code. We expect future work to further investigate this principled methodology towards producing richer representations.

4.3 AI for Software Engineering

Research on the automation of several software engineering tasks is being rebooted lately due to the improved performance of AI algorithms, the availability of data and compute capacity as well as the democratization of various deep learning programming frameworks. Tasks such as code classification, code clone detection [54, 78], code completion [4, 11, 50, 51], defect/vulnerability predictions [25, 46, 48] are now classically used to benchmark research advances in AI for software engineering. Existing approaches exploit different signals in the code: lexical information in tokens [31, 38, 60], structural information and ASTs and other program tree representations [4, 54, 78], or visual information [39, 48].
➥ While some research work develop representation learning approaches for a specific task, our CodeGrid representations can be inferred for any multiline code fragments and be applied to any task. In contrast, representations such as Code2vec, which require AST path traversals, are challenging to apply to arbitrary code fragments; similarly, approaches, such as WySiWiM, cannot be applied to tasks that require decoding the representation back to code (*e.g.*, code completion). Finally, representations such as SLM specifically aim a single task where they are extremely effective. While CodeGrid is generic, it helped classical CNN-based learning architectures achieve near SOTA results for various software engineering tasks.

5 CONCLUSION

We have argued that code typesetting should be preserved when constructing representations of code to feed to neural network architectures targeting software engineering tasks. We therefore investigated the spatial dimension of code and designed the CodeGrid approach as a principled methodology for representing code in a way that preserves its visual layout. Spatially-aware architectures are best placed to leverage CodeGrid representations. We showed that augmenting them with CodeGrid provides performance that is on-par with literature baselines for several software engineering tasks. We further validated our initial assertion that code is spatial by experimentally showing that, when representations are not faithful to the typesetting, models under-perform compared to when they use CodeGrid representations. Future work will investigate embedding techniques for token vectorizing that directly account for spatial information.

DATA AVAILABILITY

For the sake of Open Science, we provide to the community all the artifacts used in our study. The project's repository including all artifacts (tool, datasets, etc.) is available at:

https://github.com/itscodegrid/codegrid

ACKNOWLEDGMENTS

This work was partly supported (1) by the Luxembourg National Research Fund (FNR) - NERVE project, ref. 14591304, (2) by the Luxembourg Ministry of Foreign and European Affairs through their Digital4Development (D4D) portfolio under project LuxWAyS and (3) by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (Project NATURAL - grant agreement N^o 949014).

REFERENCES

[1] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A transformer-based approach for source code summarization. *arXiv preprint arXiv:2005.00653* (2020). https://doi.org/10.48550/arXiv.2005.00653
[2] Miltiadis Allamanis. 2021. Graph Neural Networks on Program Analysis. In *Graph Neural Networks: Foundations, Frontiers, and Applications*, Lingfei Wu, Peng Cui, Jian Pei, and Liang Zhao (Eds.). Springer, Singapore, Chapter need number, need pages. https://doi.org/10.1007/978-981-16-6054-2_22
[3] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. 2014. Learning Natural Coding Conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 281–293. https://doi.org/10.1145/2635868.2635883
[4] Uri Alon, Roy Sadaka, Omer Levy, and Eran Yahav. 2020. Structural language models of code. In *International Conference on Machine Learning*. PMLR, 245–256.
[5] 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.
[6] Ambient Software Evoluton Group. 2013. . https://sites.google.com/site/asegsecold/projects/seclone.
[7] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. 2018. Neural code comprehension: A learnable representation of code semantics. *arXiv preprint arXiv:1806.07336* (2018).
[8] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. 1994. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks* 5, 2 (1994), 157–166. https://doi.org/10.1109/72.279181
[9] Sergey Bezryadin, Pavel Bourov, and Dmitry Ilinih. 2007. Brightness calculation in digital image processing. In *International symposium on technologies for digital photo fulfillment*, Vol. 2007. Society for Imaging Science and Technology, 10–15. https://doi.org/10.2352/ISSN.2169-4672.2007.1.0.10

[10] Cathal Boogerd and Leon Moonen. 2008. Assessing the value of coding standards: An empirical study. In *2008 IEEE International Conference on Software Maintenance*. 277–286. https://doi.org/10.1109/ICSM.2008.4658076

[11] Marcel Bruch, Martin Monperrus, and Mira Mezini. 2009. Learning from Examples to Improve Code Completion Systems. 213–222. https://doi.org/10.1145/1595696.1595728

[12] Harold E. Burtt. 1949. Typography and Readability. *Elementary English* 26, 4 (April 1949), 212–221. https://www.jstor.org/stable/41383630

[13] Casey Casalnuovo, Earl T. Barr, Santanu Kumar Dash, Prem Devanbu, and Emily Morgan. 2020. A Theory of Dual Channel Constraints. In *2020 IEEE/ACM 42nd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. 25–28. https://doi.org/10.1145/3377816.3381720

[14] Checkmarx. [n. d.]. Checkmarx. ttps://www.checkmarx.com/.

[15] J. Chen, K. Hu, Y. Yu, Z. Chen, Q. Xuan, Y. Liu, and V. Filkov. 2020. Software Visualization and Deep Transfer Learning for Effective Software Defect Prediction. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. 578–589. https://doi.org/10.1145/3377811.3380389

[16] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. https://doi.org/10.48550/arXiv.2107.03374 arXiv:2107.03374 [cs.LG]

[17] Zimin Chen and Martin Monperrus. 2019. A Literature Study of Embeddings on Source Code. *CoRR* abs/1904.03061 (2019). arXiv:1904.03061 http://arxiv.org/abs/1904.03061

[18] Cody Coleman, Deepak Narayanan, Daniel Kang, Tian Zhao, Jian Zhang, Luigi Nardi, Peter Bailis, Kunle Olukotun, Chris Ré, and Matei Zaharia. 2017. Dawnbench: An end-to-end deep learning benchmark and competition. *Training* 100, 101 (2017), 102.

[19] Wikipedia contributors. 2021. Python (programming language) Indentation. https://en.wikipedia.org/wiki/Python_(programming_language)#Indentation https://en.wikipedia.org/wiki/Python_(programming_language).

[20] Nadia Daoudi, Jordan Samhi, Abdoul Kader Kaboré, Kevin Allix, Tegawendé F. Bissyandé, and Jacques Klein. 2021. DexRay: A Simple, yet Effective Deep Learning Approach to Android Malware Detection based on Image Representation of Bytecode. In *The 2nd International Workshop on Deployable Machine Learning for Security Defense (MLHat@KDD)* (Singapore, Singapore). https://doi.org/10.1109/BigData.2018.8622324

[21] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018). https://doi.org/10.48550/arXiv.1810.04805

[22] Iain S Duff, Albert Maurice Erisman, and John Ker Reid. 2017. *Direct methods for sparse matrices*. Oxford University Press.

[23] Rick Eden and Ruth Mitchell. [n. d.]. Paragraphing for the reader. *College Composition and Communication* 37, 4 ([n. d.]), 416–441. https://doi.org/10.2307/357912

[24] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020). https://doi.org/10.48550/arXiv.2002.08155

[25] 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* (Pittsburgh, Pennsylvania) *(MSR '22)*. Association for Computing Machinery, New York, NY, USA, 608–620. https://doi.org/10.1145/3524842.3528452

[26] Rohan Ghosh and Anupam K Gupta. 2019. Investigating convolutional neural networks using spatial orderness. In *Proceedings of the IEEE/CVF International Conference on Computer Vision Workshops*. 0–0.

[27] Gustavo Grieco, Guillermo Luis Grinblat, Lucas Uzal, Sanjay Rawat, Josselin Feist, and Laurent Mounier. 2016. Toward large-scale vulnerability discovery using machine learning. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*. 85–96.

[28] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[29] Abram Hindle, Michael W. Godfrey, and Richard C. Holt. 2009. Reading beside the lines: Using indentation to rank revisions by complexity. *Science of Computer Programming* 74(7) (2009), 414–429. http://softwareprocess.ca/pubs/hindle2009SCP-Reading-beside-the-lines.pdf

[30] Aram Hovsepyan, Riccardo Scandariato, Wouter Joosen, and James Walden. 2012. Software vulnerability prediction using text analysis techniques. In *Proceedings of the 4th international workshop on Security measurements and metrics*. 7–10.

[31] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2020. Deep code comment generation with hybrid lexical and syntactical information. *Empirical Software Engineering* 25, 3 (2020), 2179–2217.

[32] T. H. Huang and H. Kao. 2018. R2-D2: ColoR-inspired Convolutional NeuRal Network (CNN)-based AndroiD Malware Detections. In *2018 IEEE International Conference on Big Data (Big Data)*. 2633–2642. https://doi.org/10.1109/BigData.2018.8622324

[33] Kevin Jesse, Premkumar T. Devanbu, and Toufique Ahmed. 2021. Learning type annotation: is big data enough?. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta (Eds.). ACM, 1483–1486. https://doi.org/10.1145/3468264.3473135

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

[35] Joel Jones. 2003. Abstract Syntax Tree Implementation Idioms. In *Proceedings of the 10th Conference on Pattern Languages of Programs (PLoP2003)*. http://www.hillside.net/plop/plop2003/Papers/Jones-ImplementingASTs.pdf Proceedings of the 10th Conference on Pattern Languages of Programs (PLoP2003)http://hillside.net/plop/plop2003/papers.html.

[36] Daniel Kahneman. 2011. *Thinking, fast and slow*. Macmillan.

[37] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering* 28, 7 (2002), 654–670.

[38] WANG Ke, Jian-Hong JIANG, and MA Rui-Yun. 2018. A code classification method based on TF-IDF. *DEStech Transactions on Economics, Business and Management* eced (2018).

[39] Patrick Keller, Laura Plein, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2021. 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 - To appear* (2021).

[40] Nikhil Ketkar. 2017. Introduction to pytorch. In *Deep learning with python*. Springer, 195–208.

[41] Fazeel Ahmed Khan and Adamu Abubakar. 2020. Machine Translation in Natural Language Processing by Implementing Artificial Neural Network Modelling Techniques: An Analysis. *International Journal on Perceptive and Cognitive Computing* 6, 1 (2020), 9–18.

[42] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. 2017. Vuddy: A scalable approach for vulnerable code clone discovery. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 595–614.

[43] Quoc Le and Tomas Mikolov. 2014. Distributed representations of sentences and documents. In *International conference on machine learning*. PMLR, 1188–1196.

[44] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. 1989. Backpropagation applied to handwritten zip code recognition. *Neural computation* 1, 4 (1989), 541–551.

[45] Jian Li, Yue Wang, Michael R Lyu, and Irwin King. 2017. Code completion with neural attention and pointer networks. *arXiv preprint arXiv:1711.09573* (2017).

[46] 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.

[47] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Hanchao Qi, and Jie Hu. 2016. Vulpecker: an automated vulnerability detection system based on code similarity analysis. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*. 201–213.

[48] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. 2018. Sysevr: A framework for using deep learning to detect software vulnerabilities. *arXiv preprint arXiv:1807.06756* (2018).

[49] 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. *arXiv preprint arXiv:1801.01681* (2018).

[50] Fang Liu, Ge Li, Bolin Wei, Xin Xia, Zhiyi Fu, and Zhi Jin. 2020. A self-attentional neural architecture for code completion with multi-task learning. In *Proceedings of the 28th International Conference on Program Comprehension*. 37–47.

[51] Fang Liu, Ge Li, Yunfei Zhao, and Zhi Jin. 2020. Multi-task Learning based Pre-trained Language Model for Code Completion. *CoRR* abs/2012.14631 (2020). arXiv:2012.14631 https://arxiv.org/abs/2012.14631

[52] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

[53] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems* 26 (2013).

[54] Lili Mou, Ge Li, Zhi Jin, Lu Zhang, and Tao Wang. 2014. TBCNN: A Tree-Based Convolutional Neural Network for Programming Language Processing. *CoRR*

abs/1409.5718 (2014). arXiv:1409.5718 http://arxiv.org/abs/1409.5718

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

[56] National Institute of Standards and Technology. 2018. National Vulnerability Database. http://nvd.nist.gov/.

[57] National Institute of Standards and Technology. 2018. Software Assurance Reference Dataset. https://samate.nist.gov/SRD/index.php.

[58] Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. 2007. Predicting vulnerable software components. In *Proceedings of the 14th ACM conference on Computer and communications security*. 529–540.

[59] Lawrence C Ngugi, Moataz Abelwahab, and Mohammed Abo-Zahhad. 2021. Recent advances in image processing techniques for automated leaf pest and disease recognition–A review. *Information processing in agriculture* 8, 1 (2021), 27–51.

[60] Tung Thanh Nguyen, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. 2013. A statistical semantic language model for source code. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 532–542.

[61] Aaron van den Oord, Nal Kalchbrenner, Oriol Vinyals, Lasse Espeholt, Alex Graves, and Koray Kavukcuoglu. 2016. Conditional image generation with pixelcnn decoders. *arXiv preprint arXiv:1606.05328* (2016).

[62] OpenAI. 2021. *GitHub Copilot - Your AI Pair Programmer*. https://copilot.github.com

[63] Alice J O'Toole and Carlos D Castillo. 2021. Face Recognition by Humans and Machines: Three Fundamental Advances from Deep Learning. *Annual Review of Vision Science* 7 (2021).

[64] Juan Ramos et al. 2003. Using tf-idf to determine word relevance in document queries. In *Proceedings of the first instructional conference on machine learning*, Vol. 242. Citeseer, 29–48.

[65] Dhavleesh Rattan, Rajesh Bhatia, and Maninder Singh. 2013. Software clone detection: A systematic review. *Information and Software Technology* 55, 7 (2013), 1165–1199.

[66] Radim Řehřek, Petr Sojka, et al. 2011. Gensim—statistical semantics in python. *Retrieved from genism. org* (2011).

[67] Chanchal Kumar Roy and James R Cordy. 2007. A survey on software clone detection research. *Queen's School of Computing TR* 541, 115 (2007), 64–68.

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

[69] Hajah T Sueno, Bobby D Gerardo, and Ruji P Medina. 2020. Converting Text to Numerical Representation using Modified Bayesian Vectorization Technique for Multi-Class Classification. *International Journal* 9, 4 (2020).

[70] Jeffrey Svajlenko, Judith F Islam, Iman Keivanloo, Chanchal K Roy, and Mohammad Mamun Mia. 2014. Towards a big data curated benchmark of inter-project code clones. In *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 476–480.

[71] Tyler Neylon. 2015. Vertical code alignment. https://medium.com/@tylerneylon/vertical-code-alignment-9635bd2ee08c.

[72] Aäron van den Oord, Nal Kalchbrenner, Oriol Vinyals, Lasse Espeholt, Alex Graves, and Koray Kavukcuoglu. 2016. Conditional Image Generation with PixelCNN Decoders. *CoRR* abs/1606.05328 (2016). arXiv:1606.05328 http://arxiv.org/abs/1606.05328

[73] Richard J Waldinger and Richard CT Lee. 1969. PROW: A step toward automatic program writing. In *Proceedings of the 1st international joint conference on Artificial intelligence*. 241–252.

[74] Huihui Wei and Ming Li. 2017. Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code.. In *IJCAI*. 3034–3040.

[75] Hyeon-Joong Yoo. 2015. Deep convolution neural networks in computer vision: a review. *IEIE Transactions on Smart Processing and Computing* 4, 1 (2015), 35–43.

[76] Hao Yu, Wing Lam, Long Chen, Ge Li, Tao Xie, and Qianxiang Wang. 2019. Neural detection of semantic code clones via tree-based convolution. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 70–80.

[77] Matthew D Zeiler and Rob Fergus. 2014. Visualizing and understanding convolutional networks. In *European conference on computer vision*. Springer, 818–833.

[78] 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 *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 783–794.

Received 2023-02-16; accepted 2023-05-03