The Journal of Systems & Software 184 (2022) 111141

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Deep learning application on code clone detection: A review of current knowledge☆

Maggie Lei, Hao Li, Ji Li, Namrata Aundhkar, Dae-Kyoo Kim *

Computer Science and Engineering, Oakland University, Rochester, MI, 48309, United States

ARTICLE INFO

Article history:
Received 14 May 2021
Received in revised form 18 October 2021
Accepted 21 October 2021
Available online 8 November 2021

Keywords:
Code clone
Code smell
Deep learning
Duplicate code
Machine learning
Literature review

ABSTRACT

Bad smells in code are indications of low code quality representing potential threats to the maintainability and reusability of software. Code clone is a type of bad smells caused by code fragments that have the same functional semantics with syntactic variations. In the recent years, the research on duplicate code has been dramatically geared up by deep learning techniques powered by advances in computing power. However, there exists little work studying the current state-of-art and future prospects in the area of applying deep learning to code clone detection. In this paper, we present a systematic review of the literature on the application of deep learning on code clone detection. We aim to find and study the most recent work on the subject, discuss their limitations and challenges, and provide insights on the future work.

1. Introduction

Code smells indicate symptoms in a program that can cause a negative impact on software maintenance and evolution (Fowler, 2018). They can make source code error-prone, increase the risk of failure, and consequently deteriorate the quality of the system (Fontana et al., 2016; Azeem et al., 2019). Code clone is a type of code smells that is caused by code fragments that have the same functional semantics with syntactic variations. Code clone has been an active area of research in its own as witnessed in a large amount of work (Sobrinho et al., 2021). In recent years, the research on code clones (e.g., White et al., 2016; Li et al., 2017; Wei and Li, 2017; Zhang et al., 2019; Yuan et al., 2020; Hua et al., 2020), in particular code clone detection, has rapidly adopted deep learning, a nascent field of machine learning which leverages neural networks. The application of deep learning to clone detection enables to make highly accurate decision on duplicity of code fragment.

Recently, there have been many studies on applying deep learning to code clone detection with various approaches since 2016. However, there exists little work providing comparative analysis of those studies in terms of how their approaches are different and how they perform, and insights on what to be done with respect to limitations and challenges. Rattan et al. (2013) presented a literature review on software clone in general and detection techniques for software clone that were published up until 2012. However, deep learning was not of their focus. More recently, Azeem et al. (2019) reviewed the application of machine learning to detect various types of code smells including duplicate code. However, clone detection was not of their focus. Only three studies were covered on duplicate code up until 2015.

In this paper, we present a systematic literature review on the application of deep learning to code clone detection in the work that was published between 2016 and 2020. To the best of our knowledge, this is the first work focusing on the application of deep learning to code clone detection. In the review, we first formulate a review protocol and establish research questions to answer in this work. Then, we identify primary work to study based on the review protocol and conduct the review based on an analysis form listing items to analyze and the quality attributes concerned in the primary work. Based on the findings in the review, we will try to answer the research questions. In this study, we aim to find recent trends on applying deep learning to code clone detection and provide insights on the state-of-the-art in the field. Specifically, we seek to answer the following questions – (a) what are the latest techniques applying deep learning to detecting code clones? (b) what is their performance, complexity, and scalability? (c) what are their limitations and challenges? and (d) what should be the future direction in the area? The study found that various deep learning-based approaches have been applied to detecting code clones where deep neural networks and recurrent neural networks are used most; deep learning-based approaches demonstrate high performance in terms of precision, recall, and F-measure, moderate complexity, and mixed scalability and configurability; limitations and challenges vary as the approaches are diverse, but a common limitation is that only a

☆ Editor: Raffaela Mirandola.

* Corresponding author.
E-mail address: kim2@oakland.edu (D.-K. Kim).

https://doi.org/10.1016/j.jss.2021.111141
0164-1212/

few studies reported the evaluation on complexity and scalability and only a few tools provide flexible configurability, and a notable challenge is that there exists only a little work on detecting clones across different languages; the future direction is aligned with the limitations and challenges.

The rest of this paper is organized as follows. Section 2 discusses related literature reviews, Section 3 provides a background on deep learning and code clone, Section 4 contains the detailed description of our approach and findings, Section 5 answers the research questions based on the findings, and Section 7 concludes the study.

2. Related work

Roy and Cordy (2007) presented a comprehensive literature review on non-deep learning techniques up until 2006. The review includes an overview of clone definitions, clone taxonomies, detection techniques and their evaluation, applications of clone detection, and clone management. The studied detection techniques are categorized into text-based, token-based, tree-based, Program Dependency Graph (PDG)-based, metrics-based, and hybrid approaches. While the techniques studied in their work are outdated, their work provides a decent background on clone detection for those who are interested in initiating research in the area.

The report by Rattan et al. (2013) surveyed software clones in general and techniques and tools for detecting software clones, covering the work published up to 2012. Their study includes basic clone types (Type 0 to Type 4), structural clones, function clones, and model-based clones. They called for an increased awareness of potential benefits of software clone management and proposed the need for developing techniques to detect semantic and model clones. Their work provided a comprehensive review on software clones, but did not focus on deep learning.

Similar to Roy and Cordy's work Roy and Cordy (2007) and Saini et al. (2018b) categorized clone detection techniques into text-based, token Based, Abstract Syntax Tree (AST)-based, PDG-based, metric-based, and hybrid approaches. They compare these categories in terms of portability, efficiency, integrality, transformation, computational complexity, refactoring opportunity, representation, and types of clone detected. However, it is not clear what techniques belong to each category as there are no references available. They also discussed nine individual techniques, but they are not tied to the categorization.

Azeem et al. (2019) reported a survey on applying machine learning to identifying code smells in the works that were published from 2000 to 2017 and identified 20 code smells detected in 15 primary works. For the identified code smells, the report discusses the setup of the machine learning techniques applied and the design of evaluation methods and provides a meta-analysis on the performance of the techniques. However, their work did not focus on code clones and deep learning.

Min and Ping (2019) categorized code clone detection techniques into text-based detection, token-based detection, tree-based detection, PDG-based detection, and metric-based detection. The techniques discussed in their work are mostly non-deep learning techniques similar to those studied in Rattan et al.'s work (Rattan et al., 2013). They focused on structural review with no comparative study on performance evaluation. They also discussed clone refactoring, plagiarism detection, clone avoidance, and bug detection as applications of code clone detection techniques. However, little discussion was made on deep learning-based techniques.

Sobrinho et al. (2021) presented a general review on bad smells in terms of types, the evolution of researchers' interest, experimental settings, findings, interested groups of people, and paper venues. A main interest of their work is to understand how different types of smells are studied together. Their study shows that code clone has been an independent subject of research among other types of bad smells and its research community is largely separated from the communities of other types of bad smells. It is also noted that code clone has been studied far earlier than other types of code smells.

Our study differs from the existing work in that we aim to report findings on deep learning applications on detecting code clones and focus on the papers that have been published since 2016 up until now.

3. Background

In this section, we give a background on deep learning and code clone.

3.1. Deep learning

Deep learning (DL) is an emerging field in machine learning, employing the concept of neural network which is a set of pattern recognition algorithms imitating the process of human decision making by modeling high-level abstractions of data (Goodfellow et al., 2016). Each node or neuron of a neural network works as a function to extract patterns from its input dataset.

A typical neural network consists of three sequential layers — input layer, hidden layer, and output layer. There can be one or more hidden layers depending on the need of different function characteristics. Each layer contains one or more neurons. The output of a neuron is passed as input to other neurons in the next layer. The input layer takes in the input data. The data is then passed to the hidden layer for non-linear transformation process. The output of the hidden layer is taken by the output layer to produce recognized patterns in the form of a multi-dimensional vector representation (Bishop, 1995).

There are different types of neural networks including deep neural network (DNN), recurrent neural network (RNN/RtNN), recursive neural network (RvNN), graph neural network (GNN), and convolutional neural network (CNN). DNN refers to those that contain two or more hidden layers, enabling accurate data-driven decision making (Kelleher, 2019). RNN refers to those whose neurons in hidden layers are connected to not only other neurons, but also themselves. The recurrent nature enables the memory capability to process input within the context of what was previously processed (Kelleher, 2019). RNN is especially suitable to process sequential data or time-series data (Kelleher, 2019; Scarselli et al., 2009). It is also used in natural language processing (Mikolov et al., 2010). Long Short-Term Memory (LSTM), a specialized type of recurrent neural networks, is capable of handling the problem of gradient vanishing in recurrent neural networks (Kelleher, 2019). RvNN, which is the generalization of recurrent neural networks [20], contains non-recursive neurons that are connected to other neurons recursively (Stubberud and Bruce, 1998). RvNN is scalable (Zeng et al., 2019) and can be applied to look for similarities in features of datasets (e.g., Recursive Autoencoder [RAE] Socher et al., 2011). GNN was developed to better process non-Euclidean and non-sequential type of data to prevent information loss (Liu and Zhou, 2020; Wang et al., 2020). It takes as input a graph which can be acyclic, cyclic, directed, undirected, or any combination of the aforementioned (Scarselli et al., 2009). GNN takes into consideration not only the neurons in the network, but also their relationships with neighbors (Wang et al., 2020). CNN was designed for use in computer vision such as image recognition, object detection, and handwriting recognition (Aggarwal, 2018). It takes as input an image in the form of a matrix and processes the input through convolutional layers,

subsampling layers, and fully connected layers (Aggarwal, 2018). A convolutional layer defines convolution operations which applies n filters to the input to extract n feature maps as its output. Taking the feature maps as input, a subsampling layer performs pooling operations on each feature map to divide the map into subregions to obtain a value for each region. This process works to reduce the size of the feature maps and discard unnecessary and redundant features. The result is then passed on to the fully connected layer(s) whose neurons help determine which features correspond to which classifications prior to feeding into the classifier (Balas et al., 2019).

3.2. Code clones

Code clones refer to duplicate code fragments at the method level or the class level (Roy and Cordy, 2018). At the method level, two methods that are clone of each other are called clone method pair and the methods in the same pair are referred to as clone peer. At the class level, two classes that are clone of each other are called clone class, clone group, or clone community.

By the degree of similarity, code clone can be categorized as follows (Rattan et al., 2013; Sullivan, 2019; Yu et al., 2019) – Type 0 which includes completely identical code fragments, Type 1 which includes fragments that are different in white space and comments, Type 2 which includes fragments that are different in identifier names, literal values, types, and layouts, Type 3 which includes fragments that involve added or deleted statements, and Type 4 which includes fragments that have the same functionality, but are different in syntactic expression. Type 0, Type 1, and Type 2, which are relatively simple, have been well studied (Wu et al., 2020) using textual similarity analysis techniques such as longest common sequence, token-based methods, and n-gram techniques, which are non-machine learning-based. In this work, we focus on Type 3 and 4 which can be further classified by semantic similarity as follows (Roy and Cordy, 2018) – Very Strongly Type 3 ($90\% \leq$ syntactic similarity of clone $<$ 100%), Strongly Type 3 ($70\% \leq$ syntactic similarity of clone $<$ 90%), Moderately Type 3 ($50\% \leq$ syntactic similarity of clone $<$ 70%), and Weakly Type 3/Type 4 (syntactic similarity of clone $<$ 50%). Clone has been also discussed at a higher level of abstraction in terms of function and model (Rattan et al., 2013; Basit and Jarzabek, 2005; Deissenboeck et al., 2008).

4. Systematic literature review

In this section, we present a systematic literature review (Azeem et al., 2019) on the application of deep learning to code clone detection. The review was developed and performed based on the guidelines by Kitchenham et al.'s work (Kitchenham and Charters, 2007). The review consists of the following steps – (1) formulating a review protocol, (2) conducting the review, (3) reporting the results, and (4) discussing findings. The review protocol used in this review includes research questions, database search, identification and evaluation of primary studies, and data extraction and findings. To avoid bias, the development and review of the protocol were conducted independently among the authors.

In this study, we aim to address the following research questions.

RQ1 What are the latest DL techniques for detecting Type 3 and Type 4? Code clones of Type 3 and Type 4 are difficult to detect and thus, requires more accurate techniques. We would like to know how DL has been adopted lately in the effort of detecting code clones of the hard types.

RQ2 What is the performance, complexity, and scalability of DL techniques? The application of DL has been adopted with expectation of improved qualities. We would like to know how the use of DL improves the performance, complexity, and scalability in clone detection and any trade-offs made in quality attributes.

RQ3 What are the limitations and challenges of DL techniques? There can be many different ways of applying DL to the clone detection problem and we would like to know the limitations and challenges in their approaches.

RQ4 What should be the direction for future research? After learning the limitation and challenges of DL techniques, we would like to know the areas where more research endeavors should be made, which can provide guidance for future research.

4.1. Identifying primary works

We obtained the primary work to study in this review by searching IEEE Xplore, ACM Digital Library, Engineering Village (Elsevier), and Google Scholar which contain research in computer science and engineering. We also considered the work studied in the existing reviews pertaining to this study. To identify search terms, we followed the subsequent steps – (1) extract major terms from the research questions, (2) generate a list of synonyms and alternatives for each major term, (3) for each major term, combine it with its synonyms and alternatives with the OR operator, and (4) link major terms with the AND operator. The global search string resulted from the execution of the process is "(machine learning OR deep learning OR neural) AND (duplicate OR duplication OR similarity) AND (code OR software OR application) AND (clone OR cloning OR copy)".

Individual papers identified by the database search were evaluated for inclusion in this study based on the following criteria.

- It should be written in English.
- It should address detecting Type 3 and/or Type 4 based on DL.
- The title of the paper should be pertaining to the research questions.
- The abstract and keywords of the paper should have a clear link to the research questions.
- It should be published in 2016 to 2020.
- The full text of the paper should be available.

Papers that are written in a non-English language or discuss non-deep learning detection techniques or whose full text is not available and other literature reviews are excluded. Snowballing (Wohlin, 2014), which is a process of finding relevant papers from the references of the paper under study, was also performed to each identified paper to search any missing references. Only the first-level (the direct references of the identified papers) was conducted. The search resulted in a total of 21 papers which include 13 papers in IEEE Xplore, 5 papers in ACM Digital Library, 2 papers in Engineering Village (Elsevier), and 1 paper in Google Scholar. Any paper that was found in more than one database was counted only once. Google Scholar was mainly used for snowballing. We acknowledge that although we tried our best to scrutinize all relevant works, there might be some works left out unintentionally. Table 1 show the list of the identified works and Fig. 1 shows the number of papers by year.

4.2. Structural analysis

The identified works are analyzed in terms of clone types, programming languages, tool availability, classification type, training strategy, neural network, other models, approach, datasets, evaluation metrics, and limitations. Table 2 shows the analysis form.

Table 1
Identified primary works.

Year	Title	Venue
2020	From Local to Global Semantic Clone Detection (Yuan et al., 2020)	DSA
2020	FCCA: Hybrid Code Representation for Functional Clone Detection Using Attention Networks (Hua et al., 2020)	TR
2020	Detecting Code Clones with Graph Neural Network and Flow-Augmented Abstract Syntax Tree (Wang et al., 2020)	SANER
2020	Functional Code Clone Detection with Syntax and Semantics Fusion Learning (Fang et al., 2020)	ISSTA
2020	Review Sharing via Deep Semi-Supervised Code Clone Detection (Guo et al., 2020)	Access
2020	A Deep Learning Approach for a Source Code Detection Model Using Self-Attention (Meng and Liu, 2020)	Complexity
2019	A Novel Neural Source Code Representation Based on Abstract Syntax Tree (Zhang et al., 2019)	ICSE
2019	Fast Code Clone Detection Based on Weighted Recursive Autoencoders (Zeng et al., 2019)	Access
2019	Neural Detection of Semantic Code Clones via Tree-based Convolution (Yu et al., 2019)	ICPC
2019	Go-Clone: Graph-Embedding Based Clone Detector for Golang (Wang et al., 2019)	ISSTA
2019	Capturing Source Code Semantics via Tree-based Convolution over API-enhanced AST (Chen et al., 2019)	ICCF
2019	TECCD: A Tree Embedding Approach for Code Clone Detection (Gao et al., 2019)	ICSME
2019	Cross-Language Clone Detection by Learning over Abstract Syntax Trees (Perez and Chiba, 2019)	MSR
2019	CLCDSA: Cross Language Code Clone Detection using Syntactical Features and API Documentation (Nafi et al., 2019)	ASE
2018	Oreo: Detection of Clones in the Twilight Zone (Saini et al., 2018a)	ESEC/FSE
2018	DeepSim: Deep Learning Code Functional Similarity (Zhao and Huang, 2018)	ESEC/FSE
2018	CCDLC Detection Framework: Combining Clustering with Deep Learning Classification for Semantic Clones (Sheneamer, 2018)	ICMLA
2018	Positive and Unlabeled Learning for Detecting Software Functional Clones with Adversarial Training (Wei and Li, 2018)	IJCAI
2017	Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code (Wei and Li, 2017)	IJCAI
2017	CCLearner: A Deep Learning-Based Clone Detection Approach (Li et al., 2017)	ICSME
2016	Deep Learning Code Fragments for Code Clone Detection (White et al., 2016)	ASE

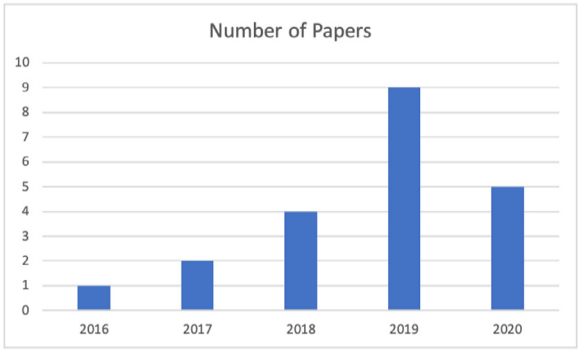


Fig. 1. Number of papers by Year.

4.2.1. Programming languages and clone types

In this section, we discuss the programming languages and clone types concerned in the studied works. Table 3 shows the programming languages and clone types addressed in the studied works and Fig. 2 shows the analysis of the works. The programming languages the studied works dealt with include C, Java, C#, C++, Python, and Go. 84% of the works used Java or C programs for clone detection. Other 16% of the works used C#, C++, Python, and Go. 90% of the works dealt with single language in clone detection, i.e., the detection was conducted on code written in one language, and only 10% of the works address cross-language (CL) clone detection (denoted in the CL column) concerning detecting the same function across different languages. This suggests further research to be done on the topic. With respect to clone types, 86% of the works studied Type 4 and only 14% studied Type 3. A further analysis reveals that 19% of the works exclusively focused on Type 3 and/or type 4, leaving out Type 0, Type 1, and Type 2 which are in general considered as simple and thus, often omitted. Among those works that focus on subtypes of Type 3, 60% addressed Weakly Type 3/Type 4, 30% focused on Moderately Type 3, and only 10% studied Strongly Type 3. Although there is no work explicitly mentioning Very Strong Type 3 (which is easy to detect due to high similarity in syntax), the works that studied Type 3 in fact addressed Very Strong Type 3.

Table 2
Analysis form.

Item	Description
Title	The title of paper.
Tool/Approach name	Proposed tool/approach name. List the researchers if not named.
Year	Publication year.
Venue	Publication venue.
Clone types & Level	Types of studied clones and their level (e.g., method-level, class-level).
Programming language	The programming languages used in the work.
Tool availability	Public availability of the tool used in the work.
Classification type	Binary classification or multi-valued classification.
Training strategy	Supervised, unsupervised, or semi-supervised.
Neural network	Deep learning neural network architecture used in the work.
Other models	Any non-deep learning-based models used in the work.
Approach	The proposed approach in terms of source code representation, detection algorithms, embedding techniques, and input to neural network.
Dataset & Benchmark	Dataset used for training, testing, and validation. Benchmark published or previously used by other researchers.
Compared approaches	The approaches compared with the work.
Evaluation metrics	Evaluation metrics used to assess the performance of the work.
Limitations	Limitations of the work.

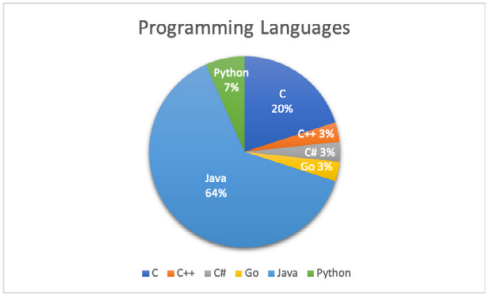
4.2.2. Neural networks and underlying techniques

In this section, we discuss the neural networks and underlying techniques used in the studied works. Table 4 shows the neural network architectures and models used in the studied works and other models used for validation of their work. The analysis in Fig. 3a shows that DNN is most used (33%) and GNN and RvNN are least used (5% each). 19% of the studied works use a combination of different types (e.g., RtNN+RvNN). In the following, we discuss the analysis in more detail in terms of source code representation, embedding techniques, filters, and training strategy.

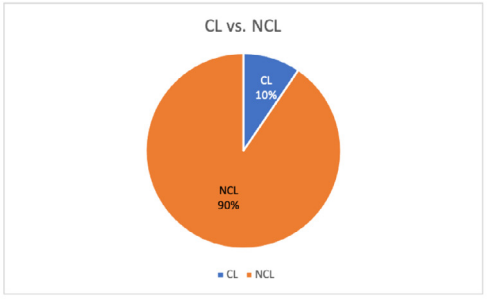
Table 3
Programming languages and clone types.

Ref.	Tool name	PL	CL	Clone type
White et al. (2016)	White et al.	Java	N	Type 0 to Type 3
Li et al. (2017)	CCLearner	Java	N	Types 0 to Strongly Type 3
Wei and Li (2017)	CDLH	Java, C	N	Types 0 to Type 4 (focus on Type 4)
Zhang et al. (2019)	ASTNN	Java, C	N	Type 0 to Type 4
Yuan et al. (2020)	Yuan et al.	Java	N	WT3/T4
Hua et al. (2020)	FCCA	Java	N	Type 0 to Type 4 (focus on WT3/T4)
Zeng et al. (2019)	Zeng et al.	Java	N	Type 0 to Type 4
Wang et al. (2020)	FA-AST+GMN	Java	N	Types 0 to Type 4 (focus on MT3 and WT3/T4
Yu et al. (2019)	TBCCD	Java, C	N	Types 0 to Type 4 (focus on MT3 and WT3/T4)
Fang et al. (2020)	FCDetector	C++	N	Type 4
Guo et al. (2020)	Rsharer+	Java	N	Type 0 to Type 4
Meng and Liu (2020)	At-BiLSTM	Java, C	N	Type 0 to Type 4 (focus on WT3/T4)
Wang et al. (2019)	Go-Clone	Go	N	Type 0 to Type 4
Chen et al. (2019)	TBCAA	Java, C	N	Types 0 to Type 4 (focus on Type 4)
Gao et al. (2019)	TECCD	Java	N	Type 0 to 3 (focus on Type 3)
Perez and Chiba (2019)	Perez & Chiba	Java, Python	Y	Type 4 (in [Java & Python])
Nafi et al. (2019)	CLCDSA	Java, Python, C#	Y	Type 4 (in [Java &Python], [Java &C#], [C# &Python])
Saini et al. (2018a)	Oreo	Java	N	Types 0 to Type 4 (focus on MT3 and WT3/T4)
Zhao and Huang (2018)	DeepSim	Java	N	Type 0 to Type 4 (focus on WT3/T4)
Sheneamer (2018)	CCDLC	Java	N	Type 0 to Type 4
Wei and Li (2018)	CDPU	Java, C	N	Types 0 to Type 4 (focus on Type 4)

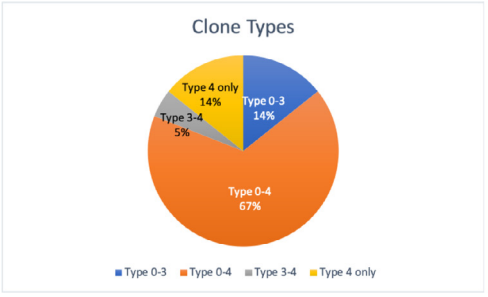
PL: Programming Language; CL: Cross-Language.



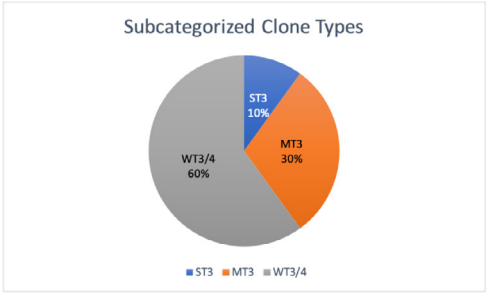
(a) The Percentage of Papers to Programming Languages



(b) The Percentage of Papers to Cross Language (CL) and Non-Cross Language (NCL)



(c) The Percentage of Papers to Clone Types



(d) The Percentage of Papers to Subcategorized Clone Types– ST3: Strong Type 3, MT3: Moderately Type 3, WT3/4: Weakly Type 3/Type 4

Fig. 2. Analysis of studied papers in terms of programming languages and clone types.

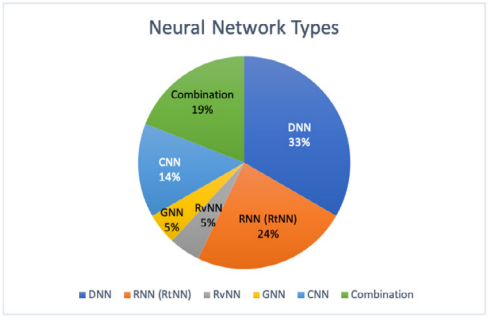
Source code representation. The table shows that the studied works used traditional techniques to represent source code into a process-friendly format that can carry a large amount of information for scalability and semantic information for improved accuracy, which helps detect Type 3 and Type 4 clones, especially those in the twilight zone (Saini et al., 2018a). Types of source code representation include Abstract Syntax Tree (AST) (White et al., 2016) capturing the syntactical information of source code sometimes with additional techniques (e.g., flow, token); Control Flow Graph (CFG) (Hua et al., 2020) representing the control flow in source code along with structural information; Data Flow Graph (DFG) (Sheneamer, 2018) tracking the data flow of source code, Program Dependency Graph (PDG) (Sheneamer, 2018) containing both control flow and data flow in source code, Bytecode Dependency Graph (BDG) (Sheneamer, 2018) depicting the low-level control flow in the Bytecode of source code, Application User Interface (API) documentation describing API call similarity in code fragments across different programming languages (Nafi et al., 2019; Chen et al., 2019); Function Call Graph (FCG) containing caller–callee relationships (Fang et al.,

Table 4
Types of neural networks and models used in studied works.

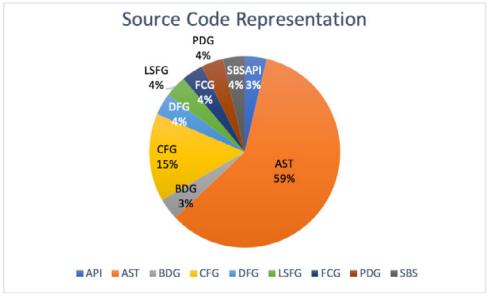
Ref.	NNA	Approach	Comparative evaluation
White et al. (2016)	RtNN + RvNN	AST+FBT+Greedy+RAE+CVV	RNN+RAE+AST
Li et al. (2017)	DNN	CCLearner: 2 Hidden Layers in DNN+ AST+Token	Different # of hidden layers+different # of iterations+different feature sets
Wei and Li (2017)	RtNN	CDLH: Word2Vec+Hash+AST+LSTM	DECKARD (Jiang et al., 2007), DLC (White et al., 2016), SourcererCC (Sajnani et al., 2016)
Zhang et al. (2019)	RvNN + RNN	ASTNN: AST+RAE+Bi-GRU+DBA	LSTM (instead of GRU), AST-Full, AST-Block, AST-Node, Removing Pooling-I, Removing Pooling-II, Long code fragments, TextCNN, TBCNN, LSCNN, [PDG+HOPE], [PDG+GGNN]
Yuan et al. (2020)	RtNN	Bi-RtNN+GCN+DTW+CFG	–
Hua et al. (2020)	Deep RtNN	FCCA: Text+AST+CFG+Deep RtNN+ ATTN	Plain Text, AST, CFG, [Text+AST+CFG]
Zeng et al. (2019)	WRAE	AST+WRAE+ANNS	[AST+RAE+ANNS], [AST+RRAE+ANNS]
Wang et al. (2020)	GNN	FA-AST+GGNN+GMN	FA-AST+GGNN
Yu et al. (2019)	CNN	TBCCD: TBC+token+PACE	TBCCD, [TBCCD+token], [TBCCD+token−type]
Fang et al. (2020)	DNN	FCDetector: AST+FCG+Word2vec+DNN+ Graph2vec	[Word2vec+HOPE+DNN], [Word2vec+SDNE+DNN], [Word2vec+Node2vec+DNN], [GloVe+Graph2vec+DNN], [GloVe+HOPE+DNN], [GloVe+SDNE+DNN], [Glove+Node2vec+DNN]
Guo et al. (2020)	CNN	Rsharer+: CNN+DNAE+AST+Word2vec+ CVV	[Word2vec+SVM], [Word2vec+CNN]
Meng and Liu (2020)	Bi-LSTM	AST+Bi-LSTM+SAL	[AST+Bi-LSTM], [AST+Bi-LSTM+SAL w/ no parameter]
Wang et al. (2019)	DNN	Go-Clone: llvm IR+LSFG+5 Hidden Layers+EV	–
Chen et al. (2019)	CNN	API-enhanced-AST +CNN+SA	AST+CNN+SA
Gao et al. (2019)	DNN	TECCD: Tree-EV+AST+DNN+RW	–
Perez and Chiba (2019)	RtNN	AST+Token EV+TBSG+ LSTM+HL+SA	[Token EV+TBSG+LSTM+HL+SA], [PTT EV+TBSG+LSTM+HL+SA], [RIT EV+ TBSG+LSTM+HL+SA], [PTT EV, no values+TBSG+ LSTM+HL+SA]
Nafi et al. (2019)	DNN	CLCDSA: AST+Metrics+AF+DNN+SA	DNN+SA−AF
Saini et al. (2018a)	DNN	Oreo: SBS+Metrics+AF+SA	Logistic Regression+ Shallow NN+ Random Forest+ Plain DNN
Zhao and Huang (2018)	DNN	DeepSim: MLP+SFM+CFG+DFG	DECKARD (Jiang et al., 2007), RtvNN (White et al., 2016), CDLH (Wei and Li, 2017)
Sheneamer (2018)	CNN	CCDLC: AST+BDG+PDG+ sIB-C+CNN	[AST+BDG+PDG+sIB-C+Naive Bayes], [AST+BDG+PDG+sIB-C+RBF Classifier] [AST+BDG+PDG+sIB-C+RBF Network], [AST+BDG+PDG+sIB-C+MLP] [AST+BDG+PDG+sIB-C+LIBLINEAR], [AST+BDG+PDG+sIB-C+IBK Classifier]
Wei and Li (2018)	RtNN	CDPU: Word2Vec+Hash+PUD+AST+ LSTM+adversarial training	[DLC (White et al., 2016)+Hash/LSH], [P-CNN (Huo et al., 2016)+Hash/LSH], [Doc2Vec+Hash/LSH]

AE: Autoencoder; ANNS: Approximate Nearest Neighbors Search; AF: Action Filter; AST: Abstract Syntax Tree; ATTN: Attention Mechanism; BDG: Bytecode Dependency Graph; Bi-GRU: Bidirectional Gated Recurrent Unit; Bi-LSTM: Bidirectional Long/Short Term Memory; Bi-RtNN: Bidirectional Recurrent Neural Network; FBT: Full Binary Tree; CDLH: Clone Detection with Learning to Hash; CDPU: Clone Detection with Positive-Unlabeled learning; CFG: Control Flow Graph; CNN: Convolutional Neural Network; CVV: Continuous-Valued Vector; DLC: Deep Learning Code; DNAE: Denoise Autoencoder; DAE: Deep Autoencoder; DBA: Dynamic Batching Algorithm; DNN: Deep Neural Network; DTW: Dynamic Time Warping; EV: Embedding Vector; FA-AST: Flow-Augmented Abstract Syntax Tree; GCN: Graph Convolutional Network; GNN: Graph Neural Network; GMN: Graph Matching Network; GGNN: Gated GNN; HL: Hash Layer; HOPE: High Order Proximity preserved Embedding; MLP: Multi-Layer Perceptron; llvm IR: low-level virtual machine Intermediate Representation; LSFG: Labeled Semantic Flow Graph; LSH: Locality Sensitive Hashing; LSTM: Long/Short Term Memory; MLP: Multi-Layer Perceptron; P-CNN: Programming language CNN; PDG: Program Dependency Graphs; PTT: Pre-Trained Token; PUD: Positive Unlabeled Data; RAE: Recursive Autoencoder; RRAE: Random Recursive Autoencoder; RtNN: Recurrent Neural Network; RvNN: Recursive Neural Network; RBF: Radial Basis Function; RIT: Randomly Initialized Token; RW: Random Walk; SA: Siamese Architecture; SAL: Self-Attention Layer; SBS: Size-Based Sharding; SFM: Semantic Feature Matrix; sIB-C: Sequential Information Bottleneck Clustering; SDNE: Structural Deep Network Embedding; SVM: Support Vector Machine; TBCCD: Tree-Based Convolution for Clone Detection; TBCNN: Tree-Based CNN; LSCNN: Localized Spectral CNN; TBSG: Tree-Based Skip-Gram; WRAE: Weighted RAE.

2020); Size-Based Sharding (SBS) (Saini et al., 2018a) creating indexes for the action tokens of a method (representing the methods called and fields accessed by the method) to speed up clone detection in subsequent filters. The above representations can be used individually or in combination and applied prior to being input to the neural network or automatically by the neural network after being input to the network. The analysis in Fig. 3b



(a) The Percentage of Papers to Neural Network Types



(b) The Percentage of Papers to Source Code Representation Techniques

Fig. 3. Analysis of studied papers in terms of neural network and source code representation.

shows that AST and CFG are dominant, accounting for 59% and 15% respectively.

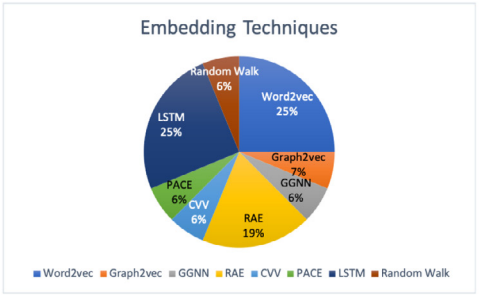
Embedding techniques. The studied works used various embedding techniques to represent data (e.g., words, graphs) as real-valued vectors in a lower dimensional space. They include word2vec (Mikolov et al., 2013) which is a family of learning algorithms for embedding words into a vector that encodes the meaning of the word using unsupervised learning. The words that are closer in the vector space are expected to be similar in meaning; graph2vec (Narayanan et al., 2017) which is a neural learning method for embedding graphs using unsupervised learning. Inspired by doc2vec (Le and Mikolov, 2014), it finds a representation of a graph by considering the graph as a document and its rooted subgraphs as words in the document; position aware character embedding (PACE) (Yu et al., 2019) which is based on one-hot embedding (Koehren, 2018) to encode a token in source code as a position-weighted combination of character one-hot embeddings; continuous-valued vector (CVV) (White et al., 2016) which encodes similar terms in fragments to similar vectors; recursive autoencoder (RAE) (Zeng et al., 2019) which encodes a program into a set of high-dimensional vectors; gated graph neural network (GGNN) (Li et al., 2015) which learns a vector representation of graphs representing the source program by combining gated recurrent units with GNN (Scarselli et al., 2009); long/short term memory (LSTM) networks which learn a vector representation of a sequence of statement trees representing the source program; random walk (Grover and Leskovec, 2016) which is an algorithm used in node2vec (Grover and Leskovec, 2016) to create a path that consists of random steps on a mathematical space. The analysis in Fig. 4a shows that word2vec, LSTM, and RAE are more common than others, accounting for 69%, but there is no prominently dominant one.

Filters. Some work used filters to filter out data for the purpose of improving efficiency and scalability. They can be categorized into reducing line of code (LOC), using action filters, using hash layers, and using sequential information bottleneck (sIB) clustering. Some work (White et al., 2016; Li et al., 2017) considered only the methods that satisfy a certain condition on the size of the methods, for example the methods that have 10 to 50 lines of code (White et al., 2016) or excludes a method pair from the candidate list of clone pairs if the LOC of one method in the pair is 3 times the other method or any method in the pair has LOC less than 6 (Li et al., 2017). Some other work (Zeng et al., 2019; Nafi et al., 2019; Saini et al., 2018a) used an action filter to filter out non-probable clones to improve scalability by considering semantic similarity (Saini et al., 2018a) or functionality (Zeng et al., 2019) in the same language or across different programming languages (Nafi et al., 2019). Sequential information bottleneck (sIB) clustering is also used as a filter to add a new feature to the vector representation of a method pair, which improve the accuracy of clone detection (Sheneamer, 2018). Among those works that used a filter, action filter is the most used method (50%) as shown in Fig. 4b.

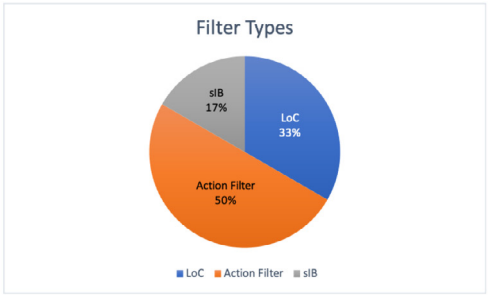
Training strategy. Some of the studied work utilized notable training strategies to improve efficiency and accuracy. They include attention mechanisms, using negative samples, and adversarial training and positive unlabeled learning (PU). Inspired by face recognition systems, an attention mechanism enables identifying important features in code by calculating the contribution of code features, which improves detection accuracy (Hua et al., 2020; Meng and Liu, 2020). The use of negative samples reduces false positives by training the model using very different, but hard to distinguish negative samples for the model (Perez and Chiba, 2019). Some work (Wei and Li, 2018) used adversarial training and PU learning together where adversarial training is used to make the model more robust against the non-clone pairs that look very similar in syntax, but different in semantics, and PU learning is used to improve detection performance by leveraging unlabeled data.

4.2.3. Benchmarks and datasets

We also analyzed the benchmarks and datasets used in the studied work. Table 5 shows the analysis. In the table, Tr, Vd, and Ts denote training, validation, and testing, respectively. The studied work used BigCloneBench and OJClone as primary benchmarks and other datasets from code repositories and other sources. BigCloneBench (Svajlenko et al., 2014) is a collection of 8 million validated clones within IJaDataset 2.0, a big data software repository of 25,000 open-source Java systems. OJClone (Mou et al., 2016) is another dataset used for evaluating code-clone detection, consisting of 500 verified solutions in C for 104 programming questions on OpenJudge where solutions for the same question are considered clones. Other datasets include ANTLR 4, Apache Ant 1.9.6, ArgoUML 0.34, CAROL 2.0.5, dnsjava 2.0.0, Hibernate 2, JDK 1.4.2, and JHotDraw 6 (White et al., 2016); IJaDataset (Saini et al., 2018a); Suple, netbeans-javadoc, eclipse-ant, and EIRC (Sheneamer, 2018); a cross-language dataset in Java and Python mined from The Apache Software Foundation and GitHub (Perez and Chiba, 2019); Google Code Jam(GCJ) (Wang et al., 2020; Nafi et al., 2019; Zhao and Huang, 2018); JDK 1.2.2, OpenNLP 1.8.1, Maven 3.5.0, Ant 1.10.1, Commons Lang 3–3.7, jEdit 5.4.0 (Gao et al., 2019); Apache commons imaging, Colt, Catalano Framework, Weka (without gui), Apache commons math3, Codewars Java corpus (Yuan et al., 2020); AtCoder, CoderByte (Nafi et al., 2019); datasets mined from code depositories (Guo et al., 2020).



(a) The Percentage of Papers to Embedding Techniques



(b) The Percentage of Papers to Filter Types

Fig. 4. Analysis of studied papers in terms of embedding techniques and filter types.

Table 5
Benchmarks and dataset.

Approach	BigCloneBench			OJClone			Other Datasets		
	Tr	Vd	Ts	Tr	Vd	Ts	Tr	Vd	Ts
White et al. (2016)	–	–	–	–	–	–	x	x	x
CCLearner (Li et al., 2017)	x	x	x	–	–	–	–	–	–
CDLH (Wei and Li, 2017)	x	x	x	x	x	x	–	–	–
ASTNN (Zhang et al., 2019)	x	x	x	x	x	x	–	–	–
Yuan et al. (2020)	–	–	–	–	–	–	x	x	x
FCCA (Hua et al., 2020)	x	x	x	–	–	–	–	–	–
Zeng et al. (2019)	x	x	x	–	–	–	–	–	–
FA-AST + GMN (Wang et al., 2020)	x	x	x	–	–	–	x	x	x
TBCCD (Yu et al., 2019)	x	x	x	x	x	x	–	–	–
FCDetector (Fang et al., 2020)	–	–	–	x	x	x	–	–	–
RSharer+ (Guo et al., 2020)	x	x	x	–	–	–	x	x	x
At-BiLSMT (Meng and Liu, 2020)	x	x	x	x	x	x	–	–	–
Go-Clone (Wang et al., 2019)	–	–	–	x	x	x	–	–	–
TBCAA (Chen et al., 2019)	x	x	x	x	x	x	–	–	–
TECCD (Gao et al., 2019)	x	x	x	–	–	–	x	x	x
Perez and Chiba (2019)	–	–	–	–	–	–	x	x	x
CLCDSA (Nafi et al., 2019)	–	–	–	–	–	–	x	x	x
Oreo (Saini et al., 2018a)	–	x	x	–	–	–	x	x	x
DeepSim (Zhao and Huang, 2018)	x	x	x	–	–	–	x	x	x
CCDLC (Sheneamer, 2018)	–	–	–	–	–	–	x	x	x
CDPU (Wei and Li, 2018)	x	x	x	x	x	x	–	–	–

Tr: Training; Vd: Validation; Ts: Testing.

Table 6 show the list of publicly available tools developed by the studied works. Only 57% of the studied works made their tools publicly available.

4.3. Qualitative analysis

The studied works are analyzed for qualitative measures for comparison. We first identified a list of quality features that clone detection tools should have based on the publicly available tools in Table 6. The identified features include performance (precision, recall, F-measure), scalability, complexity, and configurability. Table 7 shows the availability of these features in the studied tools. The table shows that many tools provide features for performance and configurability, but only a few tools support scalability and complexity. In the remaining of this section, we will discuss the details of these features. The discussion includes not only those studies whose tools are publicly available, but also those that do not have a tool or whose tools are not publicly available, but reported in their papers.

4.3.1. Performance analysis

We compare the performance of the studied works in terms of precision, recall and F-measures based on the evaluation reported in their studies. Precision reflects on the percentage of true positives that are detected by the tool. High precision means that detected clones are actual clones, while low precision means many false positives existing and the clones that are detected as true are in fact not true clones. Recall reflects on the percentage of all the clones found by the tool. High recall means that most clones are found, while low recall means many false negatives (true clones are reported as non-clones) existing in the results. F-measure is the harmonic mean of precision and recall (Nichols et al., 2019). The higher the value of precision, recall, and F-measure, the better the performance of the tool.

Table 8 shows the performance comparison. In the table, the non-clone column (denoted as NC) under Precision and Recall is for the learning models that have multi-value classifications (not binary). The table shows that the studied works performed well on detecting Type 3 and Type 4 clones, which demonstrates the effectiveness of DL. The earlier work shows relatively less performance (Wei and Li, 2017; Sheneamer, 2018; Wei and Li, 2018), did not report performance (White et al., 2016; Wang et al., 2019; Sheneamer, 2018), or was able to detect only up to VST3 clones (Li et al., 2017). More recent work demonstrates better performance (Zhang et al., 2019; Hua et al., 2020; Wang et al., 2020; Yu et al., 2019; Fang et al., 2020; Meng and Liu, 2020; Chen et al., 2019) among which TBCCD (Yu et al., 2019) outperforms in terms of F-measure. With respect to benchmarks, there are six

Table 6
Publicly available deep learning-based clone detection tools developed by studied works.

Tools/Approaches	Links
White et al. (2016)	https://sites.google.com/site/deeplearningclone/
CCLearner (Li et al., 2017)	https://github.com/liuqingli/CCLearner
ASTNN (Zhang et al., 2019)	https://github.com/zhangj1994/astnn
FCCA (Hua et al., 2020)	https://github.com/preesee/CodeCloneDetection
TBCCD (Yu et al., 2019)	https://github.com/yh1105/datasetforTBCCD
FCDetector (Fang et al., 2020)	https://github.com/shiyy123/FCDetector
TBCAA (Chen et al., 2019)	https://github.com/milkfan/TBCAA
TECCD (Gao et al., 2019)	https://github.com/YangLin-George/TECCD
Perez and Chiba (2019)	https://www.csg.ci.i.u-tokyo.ac.jp/projects/clone/
CLCDSA (Nafi et al., 2019)	https://github.com/Kawser-nerd/CLCDSA
Oreo (Saini et al., 2018a)	https://github.com/Mondego/oreo
DeepSim (Zhao and Huang, 2018)	https://github.com/parasol-aser/deepsim

Note that at the time of this study, the web page for the TECCD tool states that the tool is under refactoring and will be made available soon.

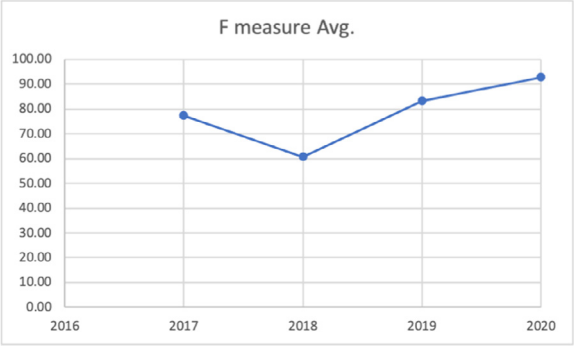


Fig. 5. The average of F-measures over Years.

studies (Wei and Li, 2017; Zhang et al., 2019; Yu et al., 2019; Meng and Liu, 2020; Chen et al., 2019; Wei and Li, 2018)) that evaluated their work on both BigCloneBench (denoted as BCB) and OJCLone(denoted as OJC). It is observed that performance was better on OJCLone except Wei et al.'s work (Wei and Li, 2017, 2018). The cross-language models (Perez and Chiba, 2019; Nafi et al., 2019) underperform other single-language tools, which suggests more study to be done in the field. Table 9 shows the average of f-measures over years, which is visualized in Fig. 5. The graph shows the improvement of performance over years, which demonstrates the advancement of DL application to clone detection. We think that the lower performance in 2017 and 2018 is attributed to the immaturity of techniques from the early adoption of deep learning in the area.

Fig. 6 shows the comparison relationships among the studied works. They are colored differently per the neural network type used in their work. The *Other* category includes any combination of different neural networks. The arrow on a relationship points to other tools that the study compared itself against. The figure shows that the two studies by White et al. (2016) and CDLH (Wei and Li, 2017) received the most comparison from other tools, which indicates their high influence in the field of clone detection. There are five studies that did not conduct comparative evaluation. The ovals with thicker round denote the work, i.e., Perez and Chiba (2019) and Nafi et al. (2019) concerned with cross-language clone detection.

The studied works also compared their work with traditional approaches. Table 10 shows the traditional tools that were used for comparison in the studied works. Deckard and SourcerCC were compared most in the studied works. Nafi et al. (2019) compared their work with three different approaches that were specific to cross-language clone detection.

As part of effort to improve performance, some studies (Chen et al., 2019; Perez and Chiba, 2019; Nafi et al., 2019; Saini et al., 2018a) adopted a Siamese architecture which helps assess the similarity of code fragments. It also has the symmetry property which is very important in clone detection i.e., the pair (a, b) is represented the same as the pair (b, a), which improves efficiency by excluding symmetric pairs. For non-RtNN implementations, this architecture also allows for parallel computing and sharing of parameters within the network which improves efficiency even more.

4.3.2. Scalability and complexity

In this section, we discuss the complexity and scalability of the studied works. Scalability is a system property to handle a growing amount of workload placed on the system. In clone detection, the size of the target system can be large involving several thousands of classes and it is important that clone detection techniques and tools should be able to detect clones in a reasonable amount of time using a reasonable amount of memory even when the detection is performed on a large-scale system. Scalability was discussed in seven studies (Li et al., 2017; Yuan et al., 2020; Hua et al., 2020; Zeng et al., 2019; Guo et al., 2020; Gao et al., 2019; Saini et al., 2018a). Table 11 shows their reports. In terms of LOC, Oreo (Saini et al., 2018a) is the most scalable; Zeng et al.'s work (Zeng et al., 2019) and CCLearner (Li et al., 2017) are less scalable; FCCA (Hua et al., 2020) and Yuan et al.'s work (Yuan et al., 2020) are least scalable. RSharer+ (Guo et al., 2020) did not report LOC.

With respect to complexity, only three studies (Li et al., 2017; Yuan et al., 2020; Zeng et al., 2019; Perez and Chiba, 2019) reported on the Big-O complexity and one study (Perez and Chiba, 2019) reported only the algorithms that are part of the system without calculating complexity. For those that reported complexity, they all reported the worst case $\mathcal{O}(n^2)$ for the entire system.

4.3.3. Configurability

Tools should be configurable to accommodate different needs depending on the project, which is important for the general use of tools. Several tools (Li et al., 2017; Zhang et al., 2019; Hua et al., 2020; Yu et al., 2019; Chen et al., 2019; Gao et al., 2019; Perez and Chiba, 2019; Saini et al., 2018a; Zhao and Huang, 2018) provide a way of configuring the settings. Table 12 shows the configurable items available in the studied tools. In terms of the number of configurable items, CCLearner (Li et al., 2017), FCCA (Hua et al., 2020), and Perez and Chiba's tool (Perez and Chiba, 2019) demonstrate higher configurability. By the type of configurable items, configuration can be categorized into architectural-level and language-level. Architectural-level configuration includes training speed, learning rate, batch size, the number of hidden layers, vector dimension, the number of iterations, the number of epochs, file paths, and embedding size (Li et al., 2017; Zhang et al., 2019; Hua et al., 2020; Gao et al., 2019; Perez and Chiba, 2019). Language-level configuration includes language name, maximum number of AST nodes, the number of samples per clone type, the number of features (e.g., reserved words, operators) (Li et al., 2017; Perez and Chiba, 2019). Some tools (Yu et al., 2019; Perez

Table 7
Qualitative features of tools.

Tools/Approaches	Precision	Recall	F-measure	Scalability	Complexity	Config.
White et al. (2016)	x	–	–	–	–	–
CCLearner (Li et al., 2017)	x	x	x	x	–	x
ASTNN (Zhang et al., 2019)	x	x	x	–	–	x
FCCA (Hua et al., 2020)	x	x	x	x		x
TBCCD (Yu et al., 2019)	x	x	x	–	x	x
FCDetector (Fang et al., 2020)	x	x	x	–	–	–
TBCAA (Chen et al., 2019)	x	x	x	–	–	x
TECCD (Gao et al., 2019)	x	x	x	x	–	x
Perez and Chiba (2019)	x	x	x	–	x	x
CLCDSA (Nafi et al., 2019)	x	x	x	–	–	–
Oreo (Saini et al., 2018a)	x	x	–	x	–	x
DeepSim (Zhao and Huang, 2018)	x	x	x	–	–	x

Table 8
Precision, Recall, F-measure of studied work.

Ref.	Precision							Recall							F-measure
	T1	T2	VST3	ST3	MT3	WT3/4	NC	T1	T2	VST3	ST3	MT3	WT3/4	NC	
White et al. (2016)	Only precision (varied values) is reported on file and method-level from 8 real-world Java systems														
Li et al. (2017)			93[a]		–	–	–	100	98	98	89	28	1	–	93[a]
Wei and Li (2017) on BCB				92			–				74			–	82
Wei and Li (2017) on OJC				47			–				73			–	57
Zhang et al. (2019) on OJC				98.9			–				92.7			–	95.5
Zhang et al. (2019) on BCB	100	100		99.9	100	99.8	–	100	100		94.2	91.7	88.3	–	93.8
Yuan et al. (2020)	–					86	–			–		95		–	91
Hua et al. (2020)				98			–				97			–	98
Zeng et al. (2019)			99.62		–	–	–	99	98.7	99	76	36.6	5.38	–	–
Wang et al. (2020) on GCJ				99			–				97			–	98
Wang et al. (2020) on BCB	100	100		100	100	95.7	–	100		100	99.6	96.5	93.5	–	94.6
Yu et al. (2019) on BCB				97			–				96			–	97
Yu et al. (2019) on OJC				99			–				99			–	99
Fang et al. (2020)	–	–	–	–	–	97	–	–	–	–	–	–	95	–	96
Guo et al. (2020) on BCB			98			80.6	78.6			99.1			77.7	81.2	85.9
Guo et al. (2020) on GK			90.2			67.9	84.2			92			74	88	82.5
Meng and Liu (2020) on OJC				96.8			–				95.3			–	96
Meng and Liu (2020) on BCB				95.6			–				91.1			–	93.4
Wang et al. (2019)				–							–				–
Chen et al. (2019) on BCB				95			–				93			–	94
Chen et al. (2019) on OJC				97			–				95			–	96
Gao et al. (2019)			88			0	–	100	96	99	87	23	0	–	–
Perez and Chiba (2019)	–	–	–	–	–	19	–	–	–	–	–	–	90	–	32
Nafi et al. (2019)	–	–	–	–	–	80[c]	–	–	–	–	–	–	48[c]	–	59[c]
Saini et al. (2018a)				89.5			–	100	99	100	89	30	0.7	–	–
Zhao and Huang (2018) on GCJ				71			–				82			–	76
Zhao and Huang (2018) on BCB				97			–				98			–	98
Sheneamer (2018)	Used sIB clustering as filter to combine high features from AST, PDG and BDG to improve accuracy														
Wei and Li (2018) on BCB[b]	100	100	98		70	51	–	100	100	98		70	51	–	83.8
Wei and Li (2018) on OJC	–	–	–	–	–	19	–	–	–	–	–	–	17	–	18

T1: Type 1; T2: Type 3; VST3: Very Strong Type 3; ST3: Strong Type 3; MT3: Moderately Type 3; WT3/4: Weakly Type 3/Type 4.
BCB: BigCloneBench; GCJ: Google Code Jam; GK: Ground Knowledge; NC: Non-Clone; OJC: OJCLone.
[a]denotes a modified version.
[b]indicates that F-measure is reported by clone type.
[c]denotes the average.

Table 9
The average of F-measures by Year.

Year	F-measures	Average
2020	91 (Yuan et al., 2020), 98 (Hua et al., 2020), 98 (Wang et al., 2020 on GCJ), 94.6 (Wang et al., 2020 on BCB), 96 (Fang et al., 2020), 85.9 (Guo et al., 2020 on BCB), 82.5 (Guo et al., 2020 on GK), 96 (Meng and Liu, 2020 on OJC), 93.4 (Meng and Liu, 2020 on BCB)	92.82
2019	95.5 (Zhang et al., 2019 on OJC), 93.8 (Zhang et al., 2019 on BCB), – (Zeng et al., 2019), 97 (Yu et al., 2019 on BCB), 99 (Yu et al., 2019 on OJC), – (Wang et al., 2019), 94 (Chen et al., 2019 on BCB), 96 (Chen et al., 2019 on OJC), – (Gao et al., 2019), 32 (Perez and Chiba, 2019), 59 (Nafi et al., 2019)	83.29
2018	– (Saini et al., 2018a), 76 (Zhao and Huang, 2018 on GCJ), 98 (Zhao and Huang, 2018 on BCB), – (Sheneamer, 2018), 51 (Wei and Li, 2018 on BCB), 18 (Wei and Li, 2018 on OJC)	60.75
2017	93 (Li et al., 2017), 82 (Wei and Li, 2017 on BCB), 57 (Wei and Li, 2017 on OJC)	77.33
2016	– (White et al., 2016)	–

– denotes "not reported".

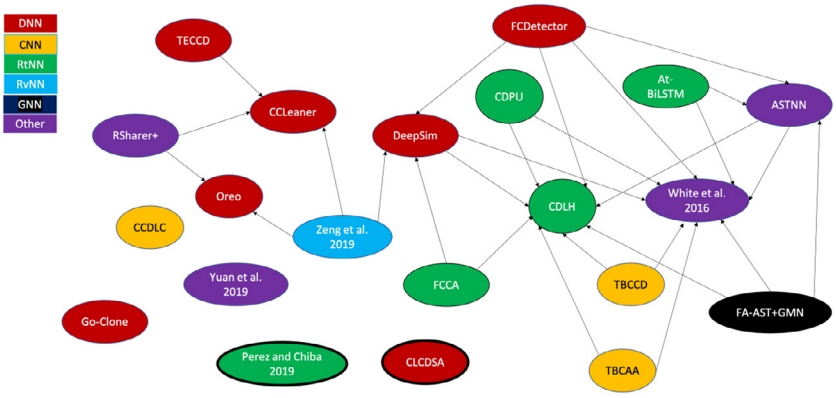


Fig. 6. Graph of 21 primary studies and their comparison relations and NN type. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Table 10
Traditional tools that the primary studies compared itself against with.

Traditional tools	Primary studies
Deckard (Jiang et al., 2007)	White et al. (2016), Li et al. (2017), Wei and Li (2017), Hua et al. (2020), Zeng et al. (2019), Wang et al. (2020), Yu et al. (2019), Fang et al. (2020), Gao et al. (2019), Saini et al. (2018a), Zhao and Huang (2018) and Wei and Li (2018)
SourcerCC (Sajnani et al., 2016)	Li et al. (2017), Wei and Li (2017), Hua et al. (2020), Zeng et al. (2019), Wang et al. (2020), Yu et al. (2019), Fang et al. (2020), Saini et al. (2018b) and Wei and Li (2018)
NiCad (Roy and Cordy, 2008)	Li et al. (2017), Yuan et al. (2020), Zeng et al. (2019), Gao et al. (2019) and Saini et al. (2018a),
CloneWorks (Svajlenko and Roy, 2017)	Zeng et al. (2019) and Saini et al. (2018a)
CCAligner (Wang et al., 2018)	Gao et al. (2019)
LICCA (Vislavski et al., 2018)	Nafi et al. (2019)
CLCMiner (Cheng et al., 2017)	Nafi et al. (2019)
AST Learner	Nafi et al. (2019)

Table 11
Scalability reported by 7 studies.

Approaches	Scalability
CCLearner (Li et al., 2017)	47 min for 3.6 M LOC
Yuan et al. (2020)	548 s for 19,016 LOC, 1142 s for 26,395 LOC, 3490 s for 42,657 LOC, 3754 s for 53,820 LOC, 5112 s for 52,737 LOC
FCCA (Hua et al., 2020)	Approximately 4000 s for 50k code pairs (approximated from the result graph at 100% of loaded data)
Zeng et al. (2019)	33 min to process 13,194,708 LOC
RSharer+ (Guo et al., 2020)	28.3 min for one sharing request
TECCD (Gao et al., 2019)	490 min for 2,688,875 LOC
Oreo (Saini et al., 2018a)	26 h 46 min on IJaDataset of 250M LOC

Table 12
Configurability of tools.

Tools/Approaches	Configurability
CCLearner (Li et al., 2017)	# of features, training speed, learning rate, batch size, # of hidden layers, # of iterations
ASTNN (Zhang et al., 2019)	Batch size
FCCA (Hua et al., 2020)	Size of hidden states, embedding size, clipping gradient range, # of epochs, learning rate, dropout, batch size
TBCCD (Yu et al., 2019)	Configured via files to create variants (e.g., TBCCD+Token, TBCCD+Token+PACE)
TBCAA (Chen et al., 2019)	Embedding and layer dimension, # of convolution kernels
TECCD (Gao et al., 2019)	Vector dimension, threshold for clone detection
Perez and Chiba (2019)	Configured via files to set language name, maximum AST nodes, # of samples per clone type, batch size, # of epochs, file paths
Oreo (Saini et al., 2018a)	# of LOC to eliminate unlikely clone pairs
DeepSim (Zhao and Huang, 2018)	Batch size, learning rate, layer size, class weights,

and Chiba, 2019) are configured by feeding in files defining configuration settings. It should be noted that configurability may affect performance and scalability. For example, a higher number of hidden layers may increase performance (e.g., high precision), but decrease scalability due to delayed time.

4.4. Limitations and challenges

The limitations of the studied works lie mainly in their architecture. Vector embedding is a common practice (e.g., word2vec, doc2vec) to represent source code as a real-valued vector for input to neural networks (White et al., 2016; Wei and Li, 2017;

Fang et al., 2020; Guo et al., 2020; Perez and Chiba, 2019; Wei and Li, 2018). Embedding is popular due to similarity-based encoding in a lower dimension. However, a vector representation has a fixed length which becomes an overhead when not all space is used or leads to information loss when the length is not big enough to accommodate the information to be represented. Word2vec, which is a word embedding technique, may produce ambiguous output. It uses a vocabulary dictionary and if the source code contains words that are not defined in the dictionary, the output becomes inaccurate and consequently the process afterward (e.g., building ASTs) is impacted. Some work (Meng and Liu, 2020; Wei and Li, 2018) used LSTM to address the gradient vanishing problem in RtNN. However, LSTM may procrastinate the training time as it does not allow parallel training.

A major challenge is found to be detecting clones across different languages (Perez and Chiba, 2019; Nafi et al., 2019) which has gained an increasing attention as software development becomes more multi-language-based. Applications based on microservices (Flower, 2014; Dragoni et al., 2017) and large web applications such as Facebook (Letuchy, 2021), Uber (Reinhold, 2016), and Netflix (Ed et al., 2016) are examples of using multi-languages in their architecture. However, there is only a little work existing on the topic. Another challenge is limited reports on the evaluation of quality aspects (e.g., scalability, complexity). While many studies reported on performance evaluation (e.g., precision, recall, F-measure) and configurability, only a few studies reported on the evaluation of scalability and complexity, which makes it difficult to analyze trade-off between quality attributes for balanced improvement. Another challenge with respect to comparative analysis is the lack of common benchmarks on which quality evaluation can be performed. Since different groups use different benchmarks for evaluation, it is difficult to have comparative analysis.

4.5. Recommendations

With respect to possible data loss in vector embedding of source code, there should be more efforts to be made towards the preservation of functional and semantic information in source code, while keeping up computational efficiency. Some work (Wei and Li, 2017; Fang et al., 2020; Guo et al., 2020; Gao et al., 2019; Perez and Chiba, 2019) used other techniques (e.g., AST, CFG, DFG) in conjunction with embedding techniques in the effort of preventing information loss. However, it come with computational cost. For example, the use of LSTM to improve the efficiency of RtNN causes delayed training time (Meng and Liu, 2020; Wei and Li, 2018). This can be addressed by adopting Parallel LSTM (PLSTM) (Bouaziz et al., 2016), a variant of LSTM which allows multiple synchronous streams to be taken simultaneously can be adopted. Alternatively, CNN may be used with a Siamese architecture to enable parallel processing, which is more compatible with AST to represent source code (Guo et al., 2020; Chen et al., 2019; Sheneamer, 2018).

The problem of cross-language clone detection needs more investigation. Only 10% of the existing work has studied the problem (see Fig. 2b). The evaluation of the existing work (Perez and Chiba, 2019; Nafi et al., 2019) indicates much work to be done in the area. Other than vectors and metrics used in the existing work to represent source code, graph-based approaches such as CFG, FCG, PDG, and BDG can be explored, which provides better abstraction for identifying corresponding clones across different languages. The existing work studied only Java, C#, and Python. Other widely used languages such as C and C++ should also be studied.

With respect to quality concerns, only a few studies reported on scalability (Li et al., 2017; Yuan et al., 2020; Hua et al., 2020; Zeng et al., 2019; Guo et al., 2020; Gao et al., 2019; Nafi et al., 2019) and complexity (Li et al., 2017; Yuan et al., 2020; Zeng et al., 2019; Perez and Chiba, 2019) which are important measures for practicality. These qualities in consideration of performance measures (precision, recall, F- measure) allow a better understanding of the correlations among qualities and trade-off analysis for balanced improvement. Also, the reported complexity (Li et al., 2017; Yuan et al., 2020; Zeng et al., 2019) offers opportunities for improvement. For comparative study, it is recommended to describe in detail as to how the experiment environment was set in terms of parameters (e.g., the number of layers, training speed, batch size), so that other studies can reproduce the environment for qualitative comparison. It is also observed that only 57% of the reported tools are publicly available (see Table 6), which creates another barrier for comparative study. More tools should be made accessible. Several tools (Li et al., 2017; Zhang et al., 2019; Hua et al., 2020; Yu et al., 2019; Chen et al., 2019; Gao et al., 2019; Perez and Chiba, 2019; Saini et al., 2018a; Zhao and Huang, 2018) are configurable, but only a few tools (Li et al., 2017; Hua et al., 2020; Perez and Chiba, 2019) provide flexible configurability in terms of configurable items. There should be studies on what can be configured and to what extent they might impact on other qualities.

5. Answering research questions

In this section, we answer the research questions posed early in Section 4 based on the findings in the same section.

RQ1: What are the latest DL techniques for detecting Type 3 and Type 4?. In this question, we wanted to know how DL has been adopted lately in the effort of detecting code clones of Type 3 and Type 4 which are hard to detect.

Answer to RQ1. We identified 21 primary studies on code clone detection using DL from an exhaustive search in various sources, i.e., IEEE Xplore, ACM Digital Library, Engineering Village (Elsevier), and Google Scholar (see Table 1 and Fig. 1).

As discussed in Section 3.2, code clones are categorized into Type 0 to Type 4 where Type 3 and 4 are be further classified into Very Strong Type 3, Strong Type 3, Moderately Type 3, and Weakly Type 3/Type 4. We found that all the studied works addressed Type 3, Type 4, or both (see Table 3) with more focus on Type 4. They used various ways of applying DL to the code clone problem (see Table 4). We found that DNN (33%), RNN (24%), combinations (19%), CNN (14%), RvNN (5%), and GNN (5%) in the order of popularity were adopted in the studied works (see Fig. 3(a)).

RQ2: What is the performance, complexity, and scalability of DL techniques? In this question, we wanted to know how the use of DL in clone detection improves performance, complexity, and scalability and any trade-offs made in quality attributes.

Answer to RQ2. We compared the performance of the studied works in terms of precision, recall and F-measure (see Table 8) based on the evaluation reported in their studies. Overall, the studied works performed well on detecting Type 3 and Type 4 clones, which demonstrates the effectiveness of DL. It is also observed that performance has been improved as the publication year goes, which demonstrates the advancement of DL techniques adopted in clone detection. It was notable that the cross-language models (Perez and Chiba, 2019; Nafi et al., 2019) underperformed other language tools, which shows the need for further study in the area. As part of effort to improve performance, some studies (e.g., Perez and Chiba, 2019; Chen et al., 2019) adopted a Siamese architecture which helps assess the similarity of code fragments and improve efficiency through parallel computing

and parameter sharing. Many works used BigCloneBench, and OJCLone(denoted as OJC), Google Code Jam, Ground Knowledge as benchmarks for performance evaluation. For performance comparison, the studied works used White et al.'s work (White et al., 2016) and CDLH (Wei and Li, 2017) most which shows high influence of White et al.'s work and CDLH in the area of clone detection (see Fig. 6). They also compared their work with traditional approaches (see Table 10) where Deckard and SourcerCC are most popular approaches used for comparison. The earlier studies showed relatively less performance or did not report performance analysis and some work was able to detect only up to VST3 clones; more recent works demonstrate better performance among which TBCCD outperformed in terms of F-measure (see 9 and 5).

With respect to complexity, only three studies (see Section 4.3.1) reported on the Big-O complexity and one study (i.e., Perez and Chiba, 2019) reported algorithms only without complexity calculated. For those that reported complexity, they all reported the worst case $\mathcal{O}(n^2)$ for the entire system.

With respect to scalability, seven studies reported scalability (see Table 11) among which in terms of LOC, Oreo (Saini et al., 2018a) demonstrated the best scalability; Zeng et al.'s work (Zeng et al., 2019) and CCLearner (Li et al., 2017) showed less scalability; FCCA (Hua et al., 2020) and Yuan et al.'s work (Yuan et al., 2020) presented the worst scalability.

RQ3: What are the limitations and challenges? In this question, we wanted to know the limitations and challenges in their approaches.

Answer to RQ3. In answering this question, we first attempted to generalize limitations and challenges in all the studied works, but we soon encountered difficulty due to the great diversity of their approaches. Instead, we will discuss them by different sub-groups of the studied works where the limitations and challenges apply commonly.

(a) Architectural limitations – [i] Several studies, i.e., White et al. (2016), Wei and Li (2017), Fang et al. (2020), Guo et al. (2020), Perez and Chiba (2019) and Wei and Li (2018) (see Table 3), used vector representations as input to RtNN. However, the use of a vector representation requires its length to be fixed, which may create an overhead delaying the processing time if not all space is used up or incur information loss if the information size is larger than the length. The information loss can be addressed by adopting CFGs and DFGs which make it unnecessary to use word embedding techniques. However, the use of CFGs and DFGs requires GNN to be used together for its ability to access graphs as input; [ii] Several studies, i.e., Wei and Li (2017), Fang et al. (2020), Guo et al. (2020) and Wei and Li (2018), adopted Word2vec for embedding words as vectors in a lower dimension space. However, Word2vec may result in non-productive output. Word2vec uses a vocabulary dictionary with a fixed number of vocabulary. If the source code contains many "unseen data" (Yu et al., 2019) that are out-of-vocabulary, the output of Word2vec becomes non-productive because the output is used to build an AST and with many unseen data, the built AST is not very useful in making a precise decision. This applies to any word-based embedding techniques. The *unseen data* issue can be addressed by using graph-based representation (e.g., CFG, DFG); [iii] Some work, i.e., Meng and Liu (2020) and Wei and Li (2018) used LSTM to improve the efficiency of RtNN, but in fact, it can make the training time longer as it does not allow parallel processing (Bouaziz et al., 2016).

(b) Challenges with cross-language code clone detection — Some work, i.e., Perez and Chiba (2019) and Nafi et al. (2019), (see Table 3) initiated efforts on cross-language code clone detection using DL. Their initiatives are timely as software development has become more multi-language-based. However, the research on the topic is still in its infancy and should be developed more. Only 10% of the studied work has investigated the problem (see Fig. 2b). As shown in Table 8, their performance leaves much room for improvement in the area.

(c) Quality-related challenges — Many studies did not report their evaluation on scalability and complexity. Only seven studies reported scalability (see Table 11) and four studies reported complexity, i.e., Li et al. (2017), Zeng et al. (2019), Yuan et al. (2020) and Perez and Chiba (2019). Due to the scanty participation on reporting scalability and complexity, it is hard to carry out trade-off analysis among quality attributes for balanced improvement.

RQ4: What are recommendations for future research? In this question, we wanted to know where in the field more research endeavors should be made based on the limitations and challenges in *AnswertoRQ* 3, which provides guidance for future research.

Answer to RQ4. (a) Recommendations for architectural challenges – [i] General techniques that can prevent the loss of functional/semantic information on source code representation without compromising computational efficiency need to be further investigated.

Although the combination of AST, CFG and DFGs with the latest embedding techniques made it possible to prevent information loss to some extent (Wei and Li, 2017; Fang et al., 2020; Guo et al., 2020; Gao et al., 2019; Perez and Chiba, 2019), it is still computationally expensive; [ii] The limitation of protracted training time in LSTM can be addressed by utilizing CNN which allows parallel processing when implemented with a Siamese architecture. It also has a better generalization ability to handle AST without the need for transforming source code into another intermediate structure, which reduces information loss. For similar reasons, CNN was adopted in several studies, i.e., Guo et al. (2020), Chen et al. (2019) and Sheneamer (2018) (see Table 4). Alternatively, PLSTM may be adopted.

(b) Recommendations for the challenges in cross-language code clone detection — More research effort should be made on cross-language code clone detection. Specifically, performance needs to be improved. The studied work, i.e., Perez and Chiba (2019), Nafi et al. (2019) on the topic used either vectors or metrics only for representing source code. It is recommended to look into using graphs to represent source code, which is popular and resulted in good performance in single-language code clone detection (see Table 8). Also, more languages and their combinations should be explored such as C and C++ which are widely used in industry. The studied work considered only Java, C#, and Python in combination of (Java, C#), (C#, Python), and (Java and Python). Also, datasets that are of large scale and represent real-world examples in industry are needed for practical impact.

(c) Recommendations for quality-related challenges – [i] Only a few studies reported on scalability (Li et al., 2017; Yuan et al., 2020; Hua et al., 2020; Zeng et al., 2019; Guo et al., 2020; Gao et al., 2019; Nafi et al., 2019) and complexity (Li et al., 2017; Yuan et al., 2020; Zeng et al., 2019; Perez and Chiba, 2019) and we call for more report on scalability and complexity. Though scalability and complexity may be considered as less significant than performance in code clone detection, they enable a balanced gauge where to improve in the field and allow trade-off analysis among quality attributes. From a practical point of view, if scalability and complexity are poor, the technique is not practical to use even with high performance (Zeng et al., 2019). Beside, reflecting upon the high complexity of those studies (Li et al., 2017; Yuan et al., 2020; Zeng et al., 2019) that reported complexity (see Section 4.3.2), we call for more computationally efficient techniques to still be explored; [ii] Only a few tools (Li et al., 2017; Hua et al., 2020; Perez and Chiba, 2019) provide flexible configurability in

terms of configurable items (see Table 12) which is important for the general use of tools. we call for more tools to provide high configurability and studies on types of items that can be configured and their impact on other qualities. Configurability is also important for comparative studies; [iii] Based on the observation that the detail parameter configuration settings were missing in many studies, which makes it difficult for future researchers to duplicate the experiment for qualitative comparison or collaborative improvement, we call for detailed information on parameter configuration settings to be made available. In the same respect, we also call for more tools that were developed by the studied work to be made available publicly. Our study shows that only 57% of clone detection tools developed in the studied works were publicly available (see Table 6).

6. Discussion

In this section, we give a brief discussion on the performance of DL-based approaches and non-DL-based approaches based on the study in this work. Per the reported performance of non-DL-based approaches (Rattan et al., 2013; Ducasse et al., 2006; Dang, 2015), most DL-based approaches outperform non-DL-based approaches. However, there are some non-DL-based approaches (e.g., dup Ducasse et al., 2006, CCFinder Kamiya et al., 2002) that demonstrated competitive performance in terms of precision and recall for some specific applications. This is attributed to the normalization of identifiers and function names and removal of some noise (e.g., comments, white space) in favor of their approaches (Ducasse et al., 2006). Such a normalization is often time consuming and error prone. DL-based approaches do not require normalization, but involves training and labeling for supervised learning, which can be considered as overheads. However, the overheads in DL-based approaches have been constantly reduced by automation (Gu and Leroy, 2019). It is hard to say which approach has bigger overheads as it depends on many different factors (e.g., types of normalization, size of source code, amount of training dataset). Nonetheless, we think that DL-based approaches are preferred from an evolution perspective.

7. Conclusion

In this paper, we reported a systematic review of the literature on the topic of deep learning-based techniques for the detection of code clone with focus on Type 3 and Type 4. We studied 21 studies from the years 2016 to 2020 and analyzed them to answer the four research questions. Our results have highlighted some special neural network architectures used, some of the embedding techniques used for source code representation, benchmarks and datasets used, the availability of tools, and evaluation metrics. We hope that our study can serve as a base for researchers who are interested in the area of code clone detection using deep learning.

CRediT authorship contribution statement

Maggie Lei: Conceptualization, Methodology, Investigation, Writing – original draft. **Hao Li:** Investigation, Writing – original draft. **Ji Li:** Investigation, Writing – original draft. **Namrata Aundhkar:** Investigation, Revision preparation. **Dae-Kyoo Kim:** Supervision, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

Aggarwal, C.C., 2018. Neural Networks and Deep Learning: A Textbook. Springer.

Azeem, M.I., Palomba, F., Shi, L., Wang, Q., 2019. Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. Inf. Softw. Technol. 108, 115–138.

Balas, V.E., Roy, S., Sharma, D., Samui, P., 2019. Handbook of Deep Learning Applications. Springer.

Basit, H., Jarzabek, S., 2005. Detecting higher-level similarity patterns in programs. In: Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering.

Bishop, C.M., 1995. Neural Networks for Pattern Recognition. Oxford University Press.

Bouaziz, M., Morchid, M., Dufour, R., Linarès, G., Mori, R., 2016. Parallel long short-term memory for multi-stream classification. In: Proceedings of IEEE Workshop on Spoken Language Technology.

Chen, L., Wei, Y., Zhang, S., 2019. Capturing source code semantics via tree-based convolution over API-enhanced AST. In: Proceedings of the 16th ACM International Conference on Computing Frontiers.

Cheng, X., Peng, Z., Jiang, L., Lingxiao, Z., Yu, H., Zhazo, J., 2017. CLCMiner: Detecting cross language clones without intermediates. IEICE Trans. Inf. Syst. E100-D (2), 273–284.

Dang, S., 2015. Performance evaluation of clone detection tools. Int. J. Sci. Res. 4 (4), 1903–1906.

Deissenboeck, F., Hummel, B., Juergens, E., Schätz, B., Wagner, S., Girard, J., Teuchert, S., 2008. Clone detection in automotive model-based development. In: Proceedings of the 30th International Conference on Software Engineering. pp. 603–612.

Dragoni, N., Giallorenzo, S., Lafuente, A.L., Mazzara, M., Montesi, F., Mustafin, R., Safina, L., 2017. Microservices: Yesterday, today, and tomorrow. In: Mazzara, M., Meyer, B. (Eds.), Present and Ulterior Software Engineering. Springer, Cham, pp. 195–216.

Ducasse, S., Nierstrasz, O., Rieger, M., 2006. On the effectiveness of clone detection by string matching. J. Softw.: Maint. Evol. 18 (1), 37–58.

Ed, B., Brian, M., Mike, M., 2016. How we build code at netflix. https://netflixtechblog.com/how-we-build-code-at-netflix-c5d9bd727f15, [Accessed: 08-20-2021].

Fang, C., Liu, Z., Shi, Y., Huang, J., Shi, Q., 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.

Flower, M., 2014. Microservices — a definition of this new architectural term. https://martinfowler.com/articles/microservices.html, [Accessed: 08-20-2021].

Fontana, F.A., Mantyla, M.V., Zanoni, M., Marino, A., 2016. Comparing and experimenting machine learning techniques for code smell detection. Empir. Softw. Eng. 21 (3), 1143–1191.

Fowler, M., 2018. Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional.

Gao, Y., Wang, Z., Liu, S., Yang, L., Sang, W., Cai, Y., 2019. TECCD: A tree embedding approach for code clone detection. In: Proceedings of IEEE International Conference on Software Maintenance and Evolution.

Goodfellow, I., Bengio, Y., Courville, A., 2016. Deep Learning. MIT Press.

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

Gu, Y., Leroy, G., 2019. Mechanisms for automatic training data labeling for machine learning. In: Proceedings of the 40th International Conference on Information Systems.

Guo, C., Yang, H., Huang, D., Zhang, J., Dong, N., Xu, J., Zhu, J., 2020. Review sharing via deep semi-supervised code clone detection. IEEE Access (8), 24948–24965.

Hua, W., Sui, Y., Wan, Y., Liu, G., Xu, G., 2020. FCCA: Hybrid code representation for functional clone detection using attention networks. IEEE Trans. Reliab..

Huo, X., Li, M., Zhou, Z., 2016. Learning unified features from natural and programming Languages for Locating Buggy Source Code. In: Proceedings of the 25th International Joint Conference on Artificial Intelligence.

Jiang, L., Misherghi, G., Su, Z., Glondu, S., 2007. Deckard: Scalable and accurate tree-based detection of code clones. In: Proceedings of the 29th International Conference on Software Engineering.

Kamiya, T., Kusumoto, S., Inoue, K., 2002. CCFinder: A multi-linguistic token-based code clone detection system for large scale source code. IEEE Trans. Softw. Eng. 28 (6), 654–670.

Kelleher, J.D., 2019. Deep Learning. MIT Press Essential Knowledge Series.

Kitchenham, B., Charters, S., 2007. Guidelines for performing Systematic Literature Reviews in Software Engineering. Tech. Rep. EBSE-2007-01, School of Computer Science and Mathematics, Keele University, Keele, UK.

Koehren, W., 2018. Neural network embeddings explained. https://towardsdatascience.com/neural-network-embeddings-explained-4d028e6f0526, [Accessed: 03-20-2020].

Le, Q., Mikolov, T., 2014. Distributed representations of sentences and documents. In: Proceedings of the 31st International Conference on Machine Learning.

Letuchy, E., 2021. Facebook chat. https://www.facebook.com/note.php?note_id=14218138919, [Accessed: 08-20-2021].

Li, L., Feng, H., Zhuang, W., Meng, N., Ryder, B., 2017. CCLearner: A deep learning-based clone detection approach. In: Proceedings of the IEEE International Conference on Software Maintenance and Evolution.

Li, Y., Tarlow, D., Brockschmidt, M., Zemel, R., 2015. Gated graph sequence neural networks. https://arxiv.org/abs/1511.05493.

Liu, Z., Zhou, J., 2020. Introduction to Graph Neural Networks. Morgan & Claypool.

Meng, Y., Liu, L., 2020. A deep learning approach for a source code detection model using self-attention. Complexity 2020 (5027198).

Mikolov, T., Chen, K., Corrado, G., Dean, J., 2013. Efficient estimation of word representations in vector space. https://arxiv.org/abs/1301.3781.

Mikolov, T., Karafiat, M., Burget, L., Cernocky, J., Khudanpur, S., 2010. Recurrent neural network based language model. In: INTERSPEECH.

Min, H., Ping, Z., 2019. Survey on software clone detection research. In: Proceedings of the 3rd International Conference on Management Engineering, Software Engineering and Service Sciences. pp. 9–16.

Mou, L., Li, G., Zhang, L., Wang, T., Jin, Z., 2016. Convolutional neural networks over tree structures for programming language processing. In: Proceedings of the 30th AAAI Conference on Artificial Intelligence.

Nafi, K.W., Kar, T.S., Roy, B., Roy, C.K., Schneider, K.A., 2019. CLCDSA: Cross language code clone detection using syntactical features and API documentation. In: Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering.

Narayanan, A., Chandramohan, M., Venkatesan, R., Chen, L., Liu, Y., Jaiswal, S., 2017. Graph2vec: Learning distributed representations of graphs. http://arxiv.org/abs/1707.05005.

Nichols, L., Emre, M., Hardekopf, B., 2019. Structural and nominal cross-language clone detection. In: Proceedings of International Conference on Fundamental Approaches to Software Engineering. pp. 247–263.

Perez, D., Chiba, S., 2019. Cross-language clone detection by learning over abstract syntax trees. In: Proceedings of the 16th IEEE/ACM International Conference on Mining Software Repositories.

Rattan, D., Bhatia, R., Singh, M., 2013. Software clone detection: A systematic review. Inf. Softw. Technol. 55 (7), 1165–1199.

Reinhold, E., 2016. Rewriting uber engineering: The opportunities microservices provide. https://eng.uber.com/building-tincup-microservice-implementation, [Accessed: 08-20-2021].

Roy, C., Cordy, J., 2007. A Survey on Software Clone Detection Research. Tech. Rep. 2007–541, School of Computing, Queen's University at Kingston, Ontario, Canada.

Roy, C., Cordy, J., 2008. Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In: Proceedings of the 16th International Conference on Program Comprehension.

Roy, C.K., Cordy, J.R., 2018. Benchmarks for software clone detection: A ten-year retrospective. In: Proceedings of the 25th IEEE International Conference on Software Analysis, Evolution and Reengineering. pp. 26–37.

Saini, V., Farmahinifarahani, F., Lu, Y., Baldi, P., Lopes, C., 2018a. Oreo: Detection of clones in the twilight zone. In: Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.

Saini, N., Singhb, S., Sumanc, 2018b. Code clones: Detection and management. In: Proceedings of International Conference on Computational Intelligence and Data Science. pp. 718–727.

Sajnani, H., Saini, V., Svajlenko, J., Roy, C., Lopes, C., 2016. SourcererCC: Scaling code clone detection to big-code. In: Proceedings of the 38th International Conference on Software Engineering.

Scarselli, F., Gori, M., Tsoi, A.C., Hagenbuchner, M., Monfardini, G., 2009. The graph neural network model. IEEE Trans. Neural Netw. 20 (1), 61–80.

Sheneamer, A., 2018. CCDLC detection framework-combining clustering with deep learning classification for semantic clones. In: Proceedings of the 17th IEEE International Conference on Machine Learning and Applications.

Sobrinho, E.V.d.P., Lucia, A.D., Maia, M.d.A., 2021. A systematic literature review on bad smells — 5 w's: Which, when, what, who, where. IEEE Trans. Softw. Eng. 47 (1), 17–66.

Socher, R., Pennington, J., Huang, E.H., Ng, A.Y., Manning, C.D., 2011. Semi-supervised recursive autoencoders for predicting sentiment distributions. In: Proceedings of Empirical Methods in Natural Language Processing.

Stubberud, P., Bruce, J.W., 1998. An LMS algorithm for training single layer globally recursive neural networks. In: Proceedings of IEEE International Joint Conference on Neural Networks.

Sullivan, C., 2019. GitHub and deep learning on graphs of code. In: Proceedings of Data Innovation Summit.

Svajlenko, J., Islam, J.F., Keivanloo, I., Roy, C.K., Mia, M., 2014. Towards a big data curated benchmark of inter-project code clones. In: Proceedings of the International Conference on Software Maintenance and Evolution.

Svajlenko, J., Roy, C., 2017. Fast and flexible large-scale clone detection with cloneworks. In: Proceedings of the 39th International Conference on Software Engineering Companion.

Vislavski, T., Rakic, G., Cardozo, N., Budimac, Z., 2018. LICCA: A tool for cross-language clone detection. In: Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering.

Wang, C., Gao, J., Jiang, Y., Xing, Z., Zhang, H., Yin, W., Gu, M., Sun, J., 2019. Go-clone: Graph-embedding based clone detector for golang. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis.

Wang, W., Li, G., Ma, B., Xia, X., Jin, Z., 2020. Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In: Proceedings of IEEE 27th International Conference on Software Analysis, Evolution and Reengineering.

Wang, P., Svajlenko, J., Wu, Y., Xu, Y., Roy, C., 2018. CCAligner: A token based large-gap clone detector. In: Proceedings of the 40th International Conference on Software Engineering.

Wei, H., Li, M., 2017. Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. In: Proceedings of the 26th International Joint Conference on Artificial Intelligence.

Wei, H., Li, M., 2018. Positive and unlabeled learning for detecting software functional clones with adversarial training. In: Proceedings of the 27th International Joint Conference on Artificial Intelligence.

White, M., Tufano, M., Vendome, C., Poshyvanyk, D., 2016. Deep learning code fragments for code clone detection. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering.

Wohlin, C., 2014. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In: Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering. pp. 1–10.

Wu, X., Qin, L., Yu, B., Xie, X., Ma, L., Xue, Y., Liu, Y., Zhao, J., 2020. How are deep learning models similar? An empirical study on clone analysis of deep learning software. In: Proceedings of the 28th International Conference on Program Comprehension. pp. 172–183.

Yu, H., Lam, W., Chen, L., Li, G., Xie, T., Wang, Q., 2019. Neural detection of semantic code clones via tree-based convolution. In: Proceedings of the 27th IEEE/ACM International Conference on Program Comprehension. pp. 70–80.

Yuan, Y., Kong, W., Hou, G., Hu, Y., Watanabe, M., Fukuda, A., 2020. From local to global semantic clone detection. In: Proceedings of the 6th IEEE International Conference on Dependable Systems and their Applications.

Zeng, J., Ben, K., Li, X., Zhang, X., 2019. Fast code clone detection based on weighted recursive autoencoders. IEEE Access 7, 125062–125078.

Zhang, J., Wang, X., Zhang, H., Sun, H., Wang, K., Liu, X., 2019. A novel neural source code representation based on abstract syntax tree. In: Proceedings of the 41st IEEE/ACM International Conference on Software Engineering.

Zhao, G., Huang, J., 2018. DeepSim: Deep learning code functional similarity. In: Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.

Maggie Lei is a graduate student in the Department of Computer Science and Engineering at Oakland University. She received a bachelor's degree in Asian Studies from Wayne State University in 2011. Her research interests include software engineering, software security, and e-commerce.

Hao Li is a graduate student in the Department of Computer Science and Engineering at Oakland University. He received a master's degree in Mechanical Engineering from Wright State University in 2014. His research interests include software engineering, multiphysics simulation & optimization, machine learning, and engineering data analyst.

Ji Li is a graduate student in the Department of Computer Science and Engineering at Oakland University. He received a master's degree in Mechanical Engineering from University of Michigan-Dearborn in 2015.

Namrata Aundhkar is a Ph.D student in the Department of Computer Science and Engineering at Oakland University. She received her Master's and Bachelor's degree in Computer Engineering from Savitribai Phule Pune University in 2011 and 2002 respectively. She worked as an assistant professor in engineering colleges affiliated to Savitribai Phule Pune University from 2003 to 2014. Her research interests include Software Engineering, Usability, Human Computer Interaction, Requirements Engineering, Personalization.

Dae-Kyoo Kim is a professor in the Department of Computer Science and Engineering at Oakland University. He received a Ph.D. in computer science from Colorado State University in 2004. He also worked as a technical specialist at the NASA Ames Research Center in 2002. His research interests include software engineering, software security, and data modeling in IoT and smart grids.