A Systematic Literature Review on the Use of Deep Learning in Software Engineering Research

CODY WATSON*, Washington & Lee University
NATHAN COOPER*, William & Mary
DAVID NADER PALACIO*, William & Mary
KEVIN MORAN, George Mason University
DENYS POSHYVANYK, William & Mary

An increasingly popular set of techniques adopted by software engineering (SE) researchers to automate development tasks are those rooted in the concept of Deep Learning (DL). The popularity of such techniques largely stems from their automated feature engineering capabilities, which aid in modeling software artifacts. However, due to the rapid pace at which DL techniques have been adopted, it is difficult to distill the current successes, failures, and opportunities of the current research landscape. In an effort to bring clarity to this cross-cutting area of work, from its modern inception to the present, this paper presents a systematic literature review of research at the intersection of SE & DL. The review canvases work appearing in the most prominent SE and DL conferences and journals and spans 128 papers across 23 unique SE tasks. We center our analysis around the *components of learning*, a set of principles that govern the application of machine learning techniques (ML) to a given problem domain, discussing several aspects of the surveyed work at a granular level. The end result of our analysis is a *research roadmap* that both delineates the foundations of DL techniques applied to SE research, and highlights likely areas of fertile exploration for the future.

CCS Concepts: • **Software and its engineering** → *Software creation and management*; *Software development techniques*;

Additional Key Words and Phrases: deep learning, neural networks, literature review, software engineering, machine learning

ACM Reference Format:
Cody Watson, Nathan Cooper, David Nader Palacio, Kevin Moran, and Denys Poshyvanyk. 2021. A Systematic Literature Review on the Use of Deep Learning in Software Engineering Research. *ACM Trans. Softw. Eng. Methodol.* ##, #, Article ### (2021), 59 pages. https://doi.org/#.##/####.####

1 INTRODUCTION

Software engineering (SE) research investigates questions pertaining to the design, development, maintenance, testing, and evolution of software systems. As software continues to pervade a wide range of industries, both open- and closed-source code repositories have grown to become

*Authors have contributed equally

Authors' addresses: Cody Watson, cwatson@wlu.edu, Washington & Lee University, 204 W Washington St., Lexington, Virginia, 24450; Nathan Cooper, nacooper01@email.wm.edu, William & Mary, 251 Jamestown Rd., Williamsburg, Virginia, 23185; David Nader Palacio, danaderpalacio@email.wm.edu, William & Mary, 251 Jamestown Rd., Williamsburg, Virginia, 23185; Kevin Moran, kpmoran@cs.wm.edu, George Mason University, 4400 University Drive, Fairfax, Virginia, 22030; Denys Poshyvanyk, denys@cs.wm.edu, William & Mary, 251 Jamestown Rd., Williamsburg, Virginia, 23185.

1049-331X/2021/#-ART### $15.00
https://doi.org/#.##/####.####

unprecedentedly large and complex. This has resulted in an increase of unstructured, unlabeled, yet important data including requirements, design documents, source code files, test cases, and defect reports. Previously, the software engineering community has applied canonical machine learning (ML) techniques to identify patterns and unique relationships within this data to automate or enhance many tasks typically performed manually by developers. Unfortunately, the process of implementing ML techniques can be a tedious exercise in careful feature engineering, wherein researchers experiment with identifying salient attributes of data that can be leveraged to help solve a given problem or automate a given task.

However, with recent improvements in computational power and the amount of memory available in modern computer architectures, an advancement to traditional ML approaches has arisen called Deep Learning (DL). Deep learning represents a fundamental shift in the manner by which machines learn patterns from data by *automatically* extracting salient features for a given computational task as opposed to relying upon human intuition. Deep Learning approaches are characterized by architectures comprised of several layers that perform mathematical transformations on data passing through them. These transformations are controlled by sets of learnable parameters that are adjusted using a variety of learning and optimization algorithms. These computational layers and parameters form models that can be trained for specific tasks by updating the parameters according to a model's performance on a set of training data. Given the immense amount of structured and unstructured data in software repositories that are likely to contain hidden patterns, DL techniques have ushered in advancements across a range of tasks in software engineering research including automatic program repair [159], code suggestion [62], defect prediction [165], malware detection [98], feature location [35], among many others [65, 106, 109, 114, 156, 162, 172, 176]. A recent report from the 2019 NSF Workshop on Deep Leaning & Software Engineering has referred to this area of work as Deep Learning for Software Engineering (DL4SE) [134].

The applications of DL to improve and automate SE tasks points to a clear synergy between ongoing research in SE and DL. However, in order to effectively chart the most impactful path forward for research at the intersection of these two fields, researchers need a clear map of what has been done, what has been successful, and what can be improved. In an effort to map and guide research at the intersection of DL and SE, we conducted a systematic literature review (SLR) to identify and systematically enumerate the synergies between the two research fields. As a result of the analysis performed in our SLR, we synthesize a detailed *research roadmap* of past work on DL techniques applied to SE tasks[1] (*i.e.*, DL4SE), complete with identified open challenges and best practices for applying DL techniques to SE-related tasks and data. Additionally, we analyzed the impacts of these DL-based approaches and discuss some observed concerns related to the potential reproducibility and replicability of our studied body of literature.

We organize our work around five major Research Questions (RQs) that are fundamentally centered upon the *components of learning*. That is, we used the various components of the machine learning process as enumerated by Abu-Mostafa [5], to aid in grounding the creation of our research roadmap and exploration of the DL4SE topic. Our overarching interest is to identify best practices and promising research directions for applying DL frameworks to SE contexts. Clarity in these respective areas will provide researchers with the tools necessary to effectively apply DL models to SE tasks. To answer our RQs, we created a taxonomy of our selected research papers that highlights important concepts and methodologies characterized by the types of software artifacts analyzed, the learning models implemented, and the evaluation of these approaches. We discovered that while

[1]It should be noted that another area, known as Software Engineering for Deep Learning (SE4DL), which explores improvements to engineering processes for DL-based systems, was also identified at the 2019 NSF workshop. However, the number of papers we identified on this topic was small, and mostly centered around emerging testing techniques for DL models. Therefore, we reserve a survey on this line of research for future work.

DL in SE has been successfully applied to many SE tasks, there are common pitfalls and details that are critical to the components of learning that are often omitted. Therefore, in addition to our taxonomy that describes how the components of learning have been addressed, we provide insight into components that are often omitted, alongside strategies for avoiding such omissions. As a result, this paper provides the SE community with important guidelines for applying DL models that address issues such as sampling bias, data snooping, and over- and under-fitting of models. Finally, we provide an online appendix with all of our data and results to facilitate reproducibility and encourage contributions from the community to continue to taxonomize DL4SE research[2] [167, 168].

2 RESEARCH QUESTION SYNTHESIS

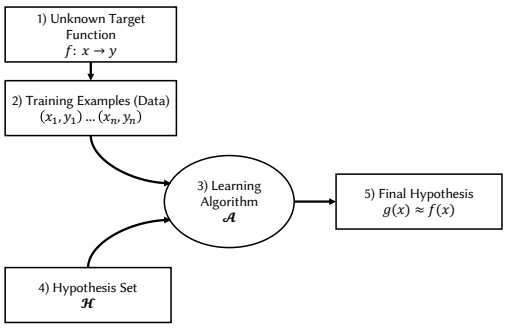

Fig. 1. The Components of Learning

The synthesis and generation of research questions (RQs) is an essential step to any systematic literature review (SLR). In order to study the intersection of DL & SE, our intention was to formulate RQs that would naturally result in the derivation of a taxonomy of the surveyed research, establish coherent guidelines for applying DL to SE tasks, and address common pitfalls when implementing these complex models. Therefore, in order to properly accomplish these tasks and frame our review, we centered the synthesis of our RQs on the *components of learning* [5], which are illustrated in Figure 1. The components of learning are a formalization introduced by Abu-Mostafa [5] in an effort to enumerate the conditions for computational learning. By framing our top-level research questions according to these components, we can ensure that that analysis component of our literature review effectively captures the essential elements that any research project applying a *deep* learning-based solution should discuss, allowing for a thorough taxonomic inspection. Given that these components represent essential elements that should be described in any application of computational learning, framing our research questions in this manner allows for the extrapolation of observed trends related to those elements that are commonly included or omitted from the surveyed literature. This, in turn, allows us to make informed recommendations to the community related to the reproducibility of our surveyed work. In the remainder of this section, we detail how each of our top-level research questions were derived from the elements of learning. Note that, in order to perform our analysis to a sufficient level of detail, in addition to our top-level RQs, we also define several Sub-RQs that allow for a deeper analysis of some of the more complex elements of learning. We provide the full list of all the research questions at the end of this section.

2.1 The First Element of Learning: The Target Function

The first component of learning is an unknown *target function* ($f : x \rightarrow y$), which represents the relationship between two observed phenomenon x and y. The target function is typically tightly coupled to the task to which a learning algorithm is applied. By analyzing the target function to be learned, one can infer the input and output of the model, the type of learning, hypothetical features to be extracted from the data and potential applicable architectures. To capture the essence of this component of learning we formulated the following research question:

[2]http://wm-semeru.github.io/dl4se/

> ***RQ*$_1$*:*** *What types of SE tasks have been addressed by DL-based approaches?*

In understanding what SE tasks have been analyzed, we are able to naturally present a taxonomy of what tasks have yet to be explored using a DL-based approach. We were also able to infer why certain SE tasks may present unique challenges for DL models as well as the target users of these DL approaches, given the SE task they address.

2.2 The Second Element of Learning: The (Training) Data

The second component of learning is defined by the *data* that is presented to a given learning algorithm in order to learn this unknown target function. Here, we primarily focused on studying the input and output training examples and the techniques used in DL approaches to prepare the data for the model. An understanding of the training examples presents greater insight into the target function while also providing further intuition about the potential features and applicable DL architectures that can be used to extract those features. Thus, in capturing this component of learning, we aimed to derive a taxonomy of the data used, how it was extracted and preprocessed, and how these relate to different DL architectures and SE tasks. This taxonomy captures relationships between data and the other elements of learning, illustrating effective (and ineffective) combinations for various SE-related applications. Our intention is that this information can inform researchers of effective combinations and potentially unexplored combinations of data/models/tasks to guide future work. Thus, our second RQ is formulated as follows:

> ***RQ*$_2$*:*** *How are software artifacts being extracted, prepared, and used in DL-based approaches for SE tasks?*

Given the multi-faceted nature of selecting, creating, and preprocessing data, we specifically examine three sub-research questions that explore the use of SE data in DL approaches in depth:

- ***RQ*$_{2a}$*:*** *What types of SE data are being used?*
- ***RQ*$_{2b}$*:*** *How is this data being extracted and pre-processed into formats that are consumable by DL approaches?*
- ***RQ*$_{2c}$*:*** *What type of exploratory data analysis is conducted to help inform model design and training?*

RQ_{2a} explores the different types of data that are being used in DL-based approaches. Given the plethora of different software artifacts currently stored in online repositories, it is important to know which of those artifacts are being analyzed and modeled. RQ_{2b} examines how data is being extracted and pre-processed into a format that a DL model can appropriately consume. The results of this RQ will enumerate the potential tools and techniques to mine different types of data for various DL applications within SE. Additionally, the representation of data is often dependent on the DL architecture and its ability to extract features from that representation, which lends importance to the discussion of the relationship between DL architectures and the data they process. RQ_{2c} investigates what type of exploratory data analysis is conducted to help inform model design and training. In order to perform effectively, DL models typically require large-scale datasets, and the quality of the learned hypothesis is a product of the quality of data from which the model learns. Therefore, since the quality of a given DL model is often directly associated with its data, we examined how research performed (or didn't perform) various analyses to avoid common data-related pitfalls recognized by the ML/DL community, including sampling bias and data snooping.

2.3 The Third & Fourth Elements of Learning: The Learning Algorithm & Hypothesis Set

Next, we jointly examine both the third and fourth components of learning, the *learning algorithm* and *hypothesis set*, in a single research question due to their highly interconnected nature. The learning algorithm is a mechanism that navigates the hypothesis set in order to best fit a given model to the data. The learning algorithm typically consists of a numeric process that uses a probability distribution over the input data to appropriately approximate the optimal hypothesis from the hypothesis set. The hypothesis set is a set of all hypotheses, based on the learning algorithm, to which the input can be mapped. This set changes because it is a function of the possible outputs given the input space, and is dependent on the learning algorithm's ability to model those possible outputs. Taken together the learning algorithm and the hypothesis set are referred to as the learning model, thus, our third RQ is formulated as follows:

> ***RQ*$_3$:** *What deep learning models are used to support SE tasks?*

Given the various types of DL model architectures and optimization techniques that may be applicable to various SE tasks, we examine RQ_3 through the lens of three sub-RQs, which address the aforementioned attributes of the learning model individually.

- ***RQ*$_{3a}$:** *What types of model architectures are used to perform automated feature engineering of the data related to various SE tasks?*
- ***RQ*$_{3b}$:** *What learning algorithms and training processes are used in order to optimize the models?*
- ***RQ*$_{3c}$:** *What methods are employed to combat over- and under-fitting of the models?*

Firstly, RQ_{3a} explores the different types of model architectures that are used to perform automated feature engineering of different SE artifacts for various SE tasks. As part of the analysis of this RQ we also examine how the type of architecture chosen to model a particular target function relates to the types of features that are being extracted from the data. Secondly, RQ_{3b} examines the different learning algorithms and training processes that are used to optimize various DL models. As part of this analysis, we explore a variety of different learning algorithms whose responsibility is to properly capture the hypothesis set for the given input space. The different optimization algorithms and training processes used to tune the weights of the model are an important step for finding the target hypothesis that best represents the data. Lastly, RQ_{3c} analyses the methods used to combat over- and under-fitting. Our intention with this RQ is to understand the specific methods (or lack thereof) used in SE research to combat over- or under-fitting, and the successes and shortcomings of such techniques.

2.4 The Fifth Element of Learning: The Final Hypothesis

Our fourth RQ addresses the component of learning known as the *final hypothesis*, which is the target function learned by the model that is used to predict aspects of previously unseen data points. In essence, in order to investigate this component of learning in the context of SE applications, we examine the *effectiveness* of the learned hypothesis as reported according to a variety of metrics across different SE tasks. Our intention with this analysis is to provide an indication of the advantages of certain data selection and processing pipelines, DL architectures, and training processes that have been successful for certain SE tasks in the past. Thus, our fourth RQ is formulated as follows:

> ***RQ*$_4$:** *How well do DL tasks perform in supporting various SE tasks?*

Analyzing the effectiveness of DL applied to a wide range of SE tasks can be a difficult undertaking due to the variety of different metrics and evolving evaluation settings used in different contexts. Thus we examined two primary aspects of the literature as sub-RQs in order to provide a holistic illustration of DL effectiveness in SE research:

- ***RQ*$_{4a}$*: What "baseline" techniques are used to evaluate DL models and what benchmarks are used for these comparisons?***
- ***RQ*$_{4b}$*: How is the impact or automatization of DL approaches measured and in what way do these models promote generalizability?***

Understanding the metrics used to quantify the comparison between DL approaches is important for informing future work regarding methods for best measuring the efficacy of newly proposed techniques. Thus, RQ_{4a} explores trade-offs related to model complexity and accuracy. In essence, we examine applications of DL architectures through the lens of the *Occam's Razor Principal*, which states that "the least complex model that is able to learn the target function is the one that should be implemented" [135]. We attempted to answer this overarching RQ by first delineating the baseline techniques that are used to evaluate new DL models and identifying what metrics are used in those comparisons. An evaluation that contains a comparison with a baseline approach, or even non-learning based solution, is important for determining the increased effectiveness of applying a new DL framework. RQ_{4b} examines how DL-based approaches are impacting the automatization of SE tasks through measures of their effectiveness and in what ways these models generalize to practical scenarios, as generalizability of DL approaches in SE is vital for their usability. For instance, if a state-of-the-art DL approach is only applicable within a narrowly defined set of circumstances, then there may still be room for improvement.

2.5 Analyzing Trends Across RQs

Our last RQ encompasses all of the components of learning by examining the extent to which our analyzed body of literature properly accounts for and describes each element of learning. In essence, such an analysis explores the potential *reproducibility & replicability* (or lack thereof) of DL applied to solve or automate various SE tasks. Therefore, our final RQ is formulated as follows:

> ***RQ*$_5$*: What common factors contribute to the difficulty when reproducing or replicating DL4SE studies?***

Our goal with this RQ is to identify common DL components which may be absent or underdescribed in our surveyed literature. In particular, we examined both the *reproducibility* and *replicability* of our primary studies as they relate to the sufficient presence or absence of descriptions of the elements of computational learning. Reproducibility is defined as the ability to take the exact same model with the exact same dataset from a primary study and produce the same results [3]. Conversely, replicability is defined as the process of following the methodology described in the primary study such that a similar implementation can be generated and applied in the same or different contexts [3]. The results of this RQ will assist the SE community in understanding what factors are being insufficiently described or omitted from approach descriptions, leading to difficulty in reproducing or replicating a given approach.

Lastly, given the analysis we perform as part of RQ_5 we derive a set of guidelines that both enumerate methods for properly applying DL techniques to SE tasks, and advocate for clear descriptions of the various different elements of learning. These guidelines start with the identification of the SE task to be studied and provide a step by step process through evaluating the new DL

approach. Due to the high variability of DL approaches and the SE tasks they are applied to, we synthesized these steps to be flexible and generalizable. In addition, we provide checkpoints throughout this process that address common pitfalls or mistakes that future SE researchers can avoid when implementing these complex models. Our hope is that adhering to these guidelines will lead to future DL approaches in SE with an increased amount of clarity and replicability/reproducibility.

2.6 Research Questions At-a-Glance

We provide our full set of research questions below:

- ***RQ$_1$: What types of SE tasks have been addressed by DL-based approaches?***
- ***RQ$_2$: How are software artifacts being extracted, prepared, and used in DL-based approaches for SE tasks?***
  - ***RQ$_{2a}$:*** *What types of SE data are being used?*
  - ***RQ$_{2b}$:*** *How is this data being extracted and pre-processed into formats that are consumable by DL approaches?*
  - ***RQ$_{2c}$:*** *What type of exploratory data analysis is conducted to help inform model design and training?*
- ***RQ$_3$: What deep learning models are used to support SE tasks?***
  - ***RQ$_{3a}$:*** *What types of model architectures are used to perform automated feature engineering of the data related to various SE tasks?*
  - ***RQ$_{3b}$:*** *What learning algorithms and training processes are used in order to optimize the models?*
  - ***RQ$_{3c}$:*** *What methods are employed to combat over- and under-fitting of the models?*
- ***RQ$_4$: How well do DL tasks perform in supporting various SE tasks?***
  - ***RQ$_{4a}$:*** *What "baseline" techniques are used to evaluate DL models and what benchmarks are used for these comparisons?*
  - ***RQ$_{4b}$:*** *How is the impact or automatization of DL approaches measured and in what way do these models promote generalizability?*
- ***RQ$_5$: What common factors contribute to the difficulty when reproducing or replicating DL studies in SE?***

3 METHODOLOGY FOR SYSTEMATIC LITERATURE REVIEW

We followed a systematic methodology to conduct our literature review in order to uphold the integrity of our analysis and provide a reproducible process. Within the field of SE, SLRs have become a standardized practice to communicate the past and present state of a particular research area. The most widely followed set of SLR standards were developed by Kitchenham *et al.* [89], thus we adopt these guidelines and procedures in our review process. As is described in Kitchenham's guidelines, we synthesized research questions (Sec. 2) before beginning the search process. This aided in naturally guiding our SLR procedure and focused the search only on primary studies pertinent to these RQs. We then performed the following steps when conducting our review:

(1) Searching for primary studies;
(2) Filtering studies according to inclusion criteria;
(3) Probability Sampling;
(4) Non-Probability Sampling;
  (a) Snowballing and manual addition of studies;
  (b) Applying exclusion criteria and performing alignment analysis;
(5) Data Extraction, Synthesis, and Taxonomy Derivation;
(6) Exploratory Data Analysis (or EDA)

A full enumeration of our methodology is illustrated in Figure 2.

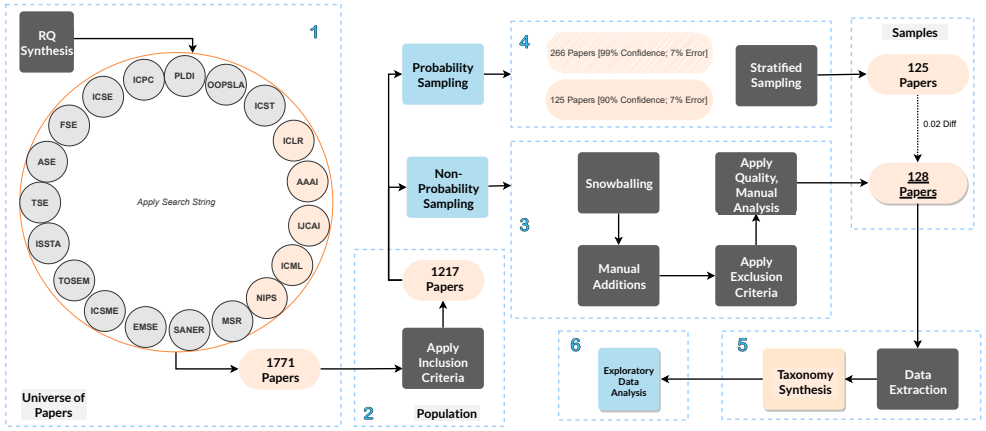

Fig. 2. SLR Methodology. Numbers correspond to each review step.

3.1 Searching for Primary Studies

The first step in our methodology was to search for primary studies that would aid in our ability to properly address the synthesized RQs. We first began by focusing on the time period we would evaluate. We chose a 10 year period of January 1st, 2009 to June 1st, 2019, which corresponded roughly to when we started our search. We chose this 10 year period because we wanted to capture the time shortly before, during, and after the seminal AlexNet work by Krizhevsky *et al.* in 2012 [90] that reinvigorated interest in neural networks. Next, we identified venues that would encompass our search. We selected the conference and journal venues which are generally recognized to be the top peer-reviewed and influential publication venues in the fields of SE and programming languages (PL), given in Table 1. We opted to include PL venues as SE & PL research often overlap, and we have observed DL approaches for code being published at PL venues. Additionally, we considered the top conferences dedicated to machine learning and deep learning in order to capture papers focused on the creation of a DL approach that also apply that approach to a SE task. These venues are also given in Table 1. This selection of venues helps to ensure all relevant research was found and considered for the generation of our DL4SE taxonomy.

Once we established the venues to search, we developed a search string to query four electronic databases in order to identify appropriate studies from these venues. The databases that we considered were: IEEE Xplore, the ACM Digital Library, Springer Link, and DBLP. The development of the search string was primarily based upon key terms present in our formulated RQs. In order to ensure that our search string was robust enough to return as many relevant primary studies as possible, we empirically evaluated multiple search strings using a variety of derived terms. We compiled ten separate search strings with a variety of terms extracted from our synthesized RQs. Each string was evaluated for its ability to extract relevant studies for consideration. The results returned by each candidate string were cross-referenced with returned candidates from other search string combinations by one author to determine the most proficient search string (see Sec. 9 for more information). We eventually found that the string *("Deep" OR "Learning" OR "Neural")* was best suited to our search for the following reasons: (i) this is the most general search string of our candidates and returned the widest variety of papers, and (ii) we believe that the generality of this term limited potential bias introduced by more focused terms. In summary, we decided to make a trade off by using a more general search term that was likely to have a higher rate of recall but resulted in more false positives for us to sort through.

The four major databases we considered provided a mechanism for advanced search that facilitates the accurate representation of our search string. Although there are nuances associated with each database, the method for gathering studies was consistent. The search string provided an initial filtering of the primary studies, then additional features of the advanced search allowed us to add discriminatory criteria. These additional features allowed us to limit our search results by year and venue. In addition to the four major databases, we also searched Google Scholar for papers through the *Publish or Perish* software [161]. To search our AI related venues, we augmented our search string with SE terms that would only return papers addressing a SE task (see our appendix for full term list 14). We gathered these terms from ICSE (the flagship academic conference in software engineering), as they represent a set of topics for technical papers generally agreed upon by the research community to represent topics of interest. We iterated through each term and appended it to our search string. The results of these searches were manually inspected for relevant papers to SE. After searching the databases with the specified search string, our initial results yielded 1,771 potentially relevant studies.

3.2 Filtering Studies According to Inclusion Criteria

In this step of the SLR, we defined the inclusion criteria that determined which primary studies would be included in our taxonomy. To an extent, part of our inclusion criteria was already used to extract the primary studies. The year and the venue were used in advanced searches to filter out extraneous research that our search string returned. However, after the search phase concluded, we filtered our results based on a more strict set of inclusion criteria. This ensured that the primary studies would be a useful addition to the taxonomy and would contain the necessary information to answer the RQs (*e.g.*, removing papers with search terms that appear only in reference lists). Our full set of inclusion and exclusion criteria considered is listed in our online appendix [167, 168].

The primary studies gathered through the use of the search string, the snowballing method, or through manual addition (as enumerated in the following steps of the search) were subjected to the same criteria for inclusion. Three of the authors divided the works and labeled them for inclusion based upon a careful reading of the abstract and approach/methodology of the paper. A fourth author then manually reviewed each classification for potential errors. If the classifications between one of the three authors and the fourth were in disagreement, then all authors reviewed and discussed the work in question until a unanimous consensus was achieved. At the end of this phase of the SLR, we were left with 1,145 studies to be further considered for inclusion in our study, narrowed down from the 1,699 studies returned by our chosen search string.

3.2.1 Snowballing and Manual Addition of Studies. Our next step was snowballing and manual inclusion of known studies. After extracting primary studies from the specified venues and subjecting those studies to our inclusion criteria, we performed snowballing on the resulting studies. Snowballing helped to ensure we were gathering other related works that were not included in our main search process. Our snowballing process looked at every reference within the studies passing our inclusion criteria and determined if any of those references also passed our inclusion criteria. Lastly, using our previous expertise in this research area, we manually added any related works that may have been missed according our knowledge of the field. We performed this manual addition for completeness of the survey and made note of which studies were manually added in our taxonomy. Only one paper which was published in arXiv in 2019 was manually added as we believe it is relevant to the DL4SE field and has been cited by a number of impactful, peer reviewed publications in well regarded SE venues [6, 85, 132, 153]

3.2.2 Exclusion Criteria and Alignment Analysis. Next, we applied our exclusion criteria to determine if there were any primary studies which were misaligned with our SLR and needed to be

filtered. This involved a significant manual analysis to closely analyze how each study incorporated the use of DL when analyzing a SE task. Specifically, we excluded papers that fell outside of our SLR's timeline, did not solve some sort of software engineering task, were outside the scope of software engineering, were not published in one of our selected venues, did not implement a deep learning model, or only used a deep learning model as a baseline. In particular, we found many studies that only used a DL-based model as a baseline to compare against. We also found instances where DL was discussed as an idea or part of the future work of a study. We therefore excluded these works to ensure that every paper we analyzed both implemented and evaluated a DL approach to address a SE task. This process resulted in 128 studies, all of which were included in the data extraction and taxonomy synthesis phases. Of these 128 papers, 110 were captured by our initial search string, 17 were added as a result of snowballing, and one was added manually. In Figure 3 we illustrate the venues from which these 128 studies originated. Next, we provide empirical evidence that our filtering methodology according to our inclusion and exclusion criteria was able to capture a representative group of DL4SE papers.

3.3 Comparison Between Probability Sampling of Papers and Non-Probability (Expert) Sampling Based on Exclusion Criteria

The goal of a given sampling strategy is to select a representative number of papers according to a given goal. Our goal in our SLR is to use non-probabilistic (also called expert) sampling methodology by applying our exclusion criteria in order to sample papers only relevant to our goals. Our exclusion criteria removed a significant number of papers from our initial list of papers that passed our inclusion criteria. However, in this subsection, we aim to empirically illustrate that our methodology is comparable to stratified sampling techniques that could be used to derive a random statistically significant sample from our set of 1,217 papers. As such we argue that our expert filtering methodology returned a *representative* set of DL4SE papers.

Table 1. Venues Included After SLR Filtering

Topic	Venue	Search String	Inclusion Criteria Filtering	SLR Sampling		Relative Difference $\Delta_{.90}$
				Random Stratified	Non-Random	
SE & PL	**ICSE**	455	90	10	16	0.39
	FSE	142	130	14	11	0.22
	ASE	162	142	15	13	0.15
	OOPSLA	36	32	3	4	0.13
	ISSTA	38	25	3	2	0.26
	EMSE	144	59	6	1	0.84
	ICSME	105	91	10	7	0.29
	MSR	52	27	3	8	0.27
	TSE	97	97	11	10	0.14
	ICPC	22	15	2	2	0.19
	PLDI	76	76	8	1	0.88
	TOSEM	24	24	3	0	1.00
	SANER	72	72	7	13	0.43
	ICST	43	42	5	1	0.78
AI/ML/DL	**AAAI**	103	98	11	7	0.34
	ICLR	45	44	5	12	0.60
	ICML	56	55	6	8	0.15
	NIPS	30	29	3	12	0.74
	IJCAI	69	69	7	0	1.00
Statistics	**Total**	1,771	1,217	**125**	**128**	0.02
	Avg ± *Std*	93.2 ± 97.3	64.1 ± 36.9	6.6 ± 3.8	6.74 ± 5.2	**0.48** ± **0.32**
	Median	69	59	6	7	0.13

We employed *Stratified Sampling* to compute the required number of samples by conference to obtain a representative set of papers. This sampling was performed for a total universe of 18 venues: 12 venues in Software Engineering and 6 venues in AI. We used confidence intervals at 90% and 99% with a margin error of %7. We then calculated the relative difference between the non-probability and the stratified sampling in order to quantify how distant our "expert criteria" filtering procedure (*i.e.*, applying our inclusion and exclusion criteria) was from a statistically significant sampling strategy. The relative difference is defined as $\Delta_{ci} = (r - s)/f(r - s)$ where r is the number of

samples obtained by stratification, s is the number of samples obtained by expert criteria, and ci is the Confidence Interval. We utilized $f(r, s) = max(r, s)$ as the relative function.

The required conventional sampling at 90% confidence is 125 papers and at 99% is 266 papers for a population of 1,217 papers (after applying inclusion criteria) from 18 venues. On the other hand, our expert criteria filtering reduced the population to a total of 128 papers. Consequently, the relative stratified difference is $\widehat{\Delta_{.99}} = 0.48 \pm 0.32$ and the relative population difference is $\widehat{\Delta_{.90}} = 0.02$ between expert sampling and probability sampling. In summary, the sampling strategy suggests that the target set of 128 papers is statistically representative for a confidence interval at 90% with a relative median difference $\tilde{x}_{.90} = 0.13$. This sampling strategy remains true under the given preconditions of time period and conferences' quality.

3.4 Data Extraction, Synthesis, and Taxonomy Derivation

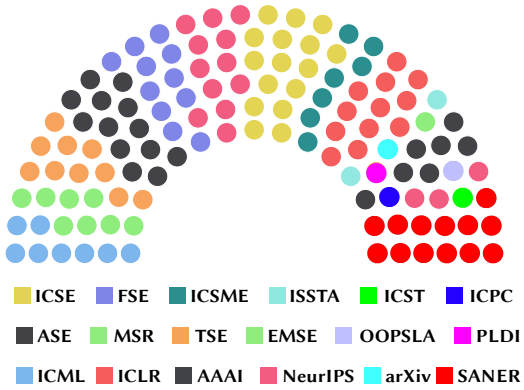

Fig. 3. Venue distribution of DL4SE

3.4.1 Data Extraction. Our next step in the SLR methodology was to extract the necessary data from the primary studies. This step involved the development of an extraction form for identifying attributes of papers that correspond to our research questions. Each author was tasked with extracting data from a subset of the 128 primary studies selected using the methodology outlined above. The results of the extraction phase were confirmed by a separate author to ensure that all important details were gathered and that papers were properly represented within the taxonomy. Each category of our extraction form is listed in the first column of Figure 4 and we provide our completed data extraction forms as part of our online appendix [167, 168].

3.4.2 Data Synthesis and Taxonomy Derivation. In order to build our taxonomy on the use of DL approaches in SE research, we followed several complementary methodologies. The first of these was an open coding methodology consistent with constructivist grounded theory [24]. Following the advice of recent work within the SE community [148], we stipulate our specific implementation of this type of grounded theory while discussing our deviations from the methods in the literature. We derived our implementation from the material discussed in [24] involving the following steps: (i) establishing a research problem and questions, (ii) data-collection and initial coding, and (iii) focused coding. We excluded other steps described in [24], such as memoing because we were focused on the construction of a taxonomy. The first two steps of this process were largely encompassed by the previously discussed methodological steps of the SLR, whereas the focused coding was used to derive the final category labels of our taxonomy that aided in answering each of our RQs. The initial coding was performed by one of the authors and then refined during the focused coding process by two others until an agreement was made among all three. The results of these coding steps formed our taxonomy. Finally, we normalized each topic in our taxonomy into detailed classes to perform a data analysis on them. This process is visualized in Figure 4.

After taxonomy construction, we make use of a combination of *descriptive statistics* and our *exploratory data analysis* in order to synthesize and describe our results to each RQ. Our use of descriptive statistics enumerates various trends we identify in our taxonomy, whereas our

exploratory data analysis investigates statistical relationships between various aspects of our taxonomy treated as "features". We further describe this analysis methodology in the following subsection. The answers to our RQs naturally define a holistic taxonomy that researchers can use to determine what types of SE tasks can be better studied using certain DL-based approaches, as well as look for future applications of DL to model complex software artifacts. We oriented our discussion of the results of our RQs to provide an understanding about the process of applying a DL-based approach in SE.

3.5 Exploratory Data Analysis

In order to gain a more holistic view of our taxonomy constructed based on the extracted data, we performed an analysis to determine the types of relationships between the various taxonomy categories, which we use to form a set of *features* used to conduct an exploratory data analysis (EDA). The mined associations between different paper attributes allowed us to provide more complete and holistic answers to several research questions, thus supporting the conclusions of our SLR. We utilized a data mining pipeline and set of statistical processes in order to uncover hidden relationships among the different extracted attributes of our primary studies.

Our data mining and analysis process was inspired by classical Knowledge Discovery in Databases, or KDD [53]. The KDD process extracts knowledge from the data gathered during our extraction process that was then converted into a database format, and involved five stages:

1. Selection. This stage was encompassed by the data extraction process explained in the beginning of this section. After collecting all the papers and extracting the relevant paper attributes, we organized the data into 55 explicit features or attributes extracted from the original data taken from the primary studies. A complete list of these features is provided in our online repository [167, 168].

2. Preprocessing. We applied a preprocessing technique that transformed and normalized categories into nominal features as depicted by the concept map in Figure 4. For instance, the category "Avoiding Over-UnderFitting" was transformed into "Over-fitting Techniques" in the taxonomy; then, such taxonomy was normalized into 10 different features (e.g., "Tokenization", "Neural Embedding", "I/O Vectors", "Hyper-Tuning", and so on). Similarly, this preprocessing pipeline was applied to other categories such as "SE Task Addressed" and "Evaluation Metrics Used". This normalization into more detailed classes contributes to a more holistic understanding of the papers via data mining methods.

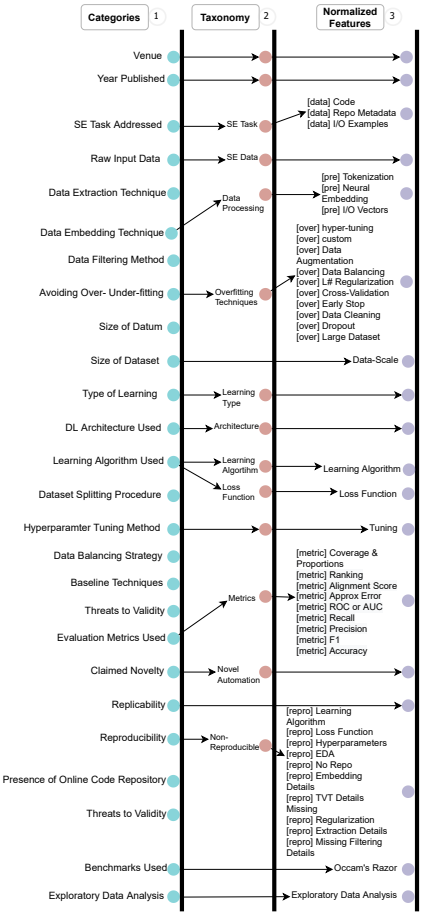

Fig. 4. SLR Concept Map: 1. Categories for Data Extraction, 2. Taxonomy Derivation, and 3. Normalized Features from KDD

3. Data Mining. In this stage, we employed two well-established data mining tasks to better understand our data: Correlation Discovery and Association Rule Learning. We oriented our KDD

process to uncover hidden relationships on the normalized features. We provide further details related to this stage in Section 3.5.1 and 3.5.2.

4. Interpretation/Evaluation We used the knowledge discovered to automatically find patterns in our papers that resemble actionable knowledge. This actionable knowledge was generated by applying a reasoning process on the data mining outcomes. Although this reasoning process produced a support analysis formally presented in our online appendix [167, 168], each summary of the outcome of a RQ contains a brief description of the results of the exploratory data analysis. We used RapidMiner [4] to conduct the data analysis, and the procedures and pipelines for using this tool are published in our companion online repository [167, 168]. Before carrying out any mining tasks, we decided to address a basic descriptive statistics procedure for each feature. These statistics exhibit basic relative frequencies, counts, and quality metrics. We then utilized four quality metrics: ID-ness, Stability, Missing, and Text-ness. ID-ness measures to what extent a feature resembles an ID, where as the title is the only feature with a high ID-ness value. Stability measures to what extent a feature is constant (or has the same value). Missing refers to the number of missing values. Finally, Text-ness measures to what extent a feature contains free-text. The results for each feature were calculated using the RapidMiner interface.

3.5.1 Correlation Discovery. Due to the nominal nature of the SLR features, we were unable to infer classic statistical correlations for our data (*i.e.*, Pearson's correlation). However, we adapted an operator based on attributes information dependencies. This operator is known as mutual information [116], which is measured in bits *B*. Similar to covariance or Pearson's correlation, we were able to represent the outcomes of the operator in a confusion matrix. Figure 5 depicts feature correlations greater than 1.0*B* from the confusion matrix.

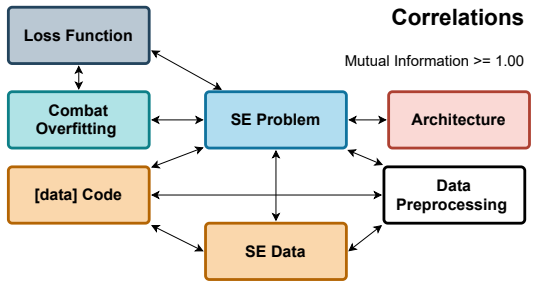

Fig. 5. SLR Feature Correlations

The mutual information measures to what extent one feature "knows" about another. High mutual information values represent less uncertainty; therefore, we built arguments such as whether the deep learning architecture used on a paper is more predictable given a particular SE task or the reported architectures within the papers are mutually dependent upon the SE task. The difference between correlation and association rules depends on the granularity level of the data (*e.g.*, paper, feature, or class). The correlation procedure was performed at the feature level, while the association rule learning was performed at the class or category level.

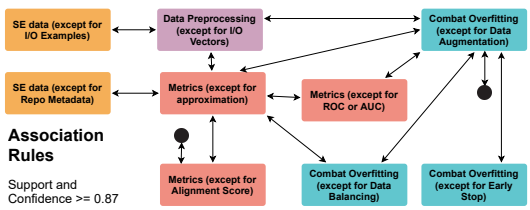

Fig. 6. SLR Association Rules

3.5.2 Association Rule Learning. For generating the association rules, we employed the classic Frequent Pattern (FP)-Growth algorithm in order to observe associations among variables. FP-Growth computes frequently co-occurring items in our transactional database with a minimum support of 0.95 and a minimum number of itemsets of 100 (Table 2 depicts the parameter values employed for FP-Grow and Association Rule Mining components). These items comprise each class per feature. For instance, the

feature Loss Function exhibited a set of classes (or items) such as NCE, Max Log Likelihood, Cross-Entropy, MSE, Hinge Loss, N/A-Loss, and so on. The main premise of the FP-Growth can be summarized as: if an item set is frequent (*i.e.*, MSE, RNN), then all of its item subsets are also frequent (*i.e.*, MSE and RNN).

Once the FP-Growth generated the item sets (*e.g.*, MSE, Hinge Loss), the algorithm analyzed the item sets support in continuous scans of the database (DB). Such support measures the occurrences of the item set in the database. The FP-Growth scans the database, which brings about a FP-tree data structure. We recursively mined the FP-tree data structure to extract frequent item sets [70]. Nonetheless, the association rule learning requires more than the FP-tree extraction.

Table 2. RapidMiner Parameters

Component	Parameter	Value
FP-Grow	min support	0.95
	min items per itemset	1
	max items per itemset	0
	max # of itemsets	10^6
	min # of itemsets	100
Association Rules	min confidence	0.8

An association rule serves as an if-then (or premise-conclusion) statement based on frequent item set patterns. Let's observe the following rule mined from our dataset: *Software Engineering data, excluding Input/Output Vectors, determine the type of data preprocessing performed by the authors with a support of 0.88 and a confidence of 0.99*. We observe the association rule has an antecedent (*i.e.*, the feature SE data, but for I/O vectors) and a consequent (*i.e.*, the feature Data Preprocessing). These relationships are mined from item sets that usually have a high support and confidence. The confidence is a measure of the number of times that premise-conclusion statement is found to be true. We kept association rules that have both confidence and support greater than 0.8.

It is possible that Association Rule Mining can lead to spurious correlations. In order to avoid this, we organized the rules into an interconnected net of premises/conclusions based upon our formulated RQs in order to find explanations around techniques and methodologies reported on the papers. Figure 6 depicts such an interconnected net with the highest support and confidence. Any non-logical rule was disregarded as well as rules that possessed a lower support. Non-logical rules are association rules where the premise and conclusion are easy to falsify. Generally, if we decrease the minimum support parameter, non-logical association rules might arise.

We discuss the results of this data analysis process where appropriate as they relate to our RQs in the following sections of this SLR. The full results of our exploratory data analysis and source code of our entire EDA process can be found in our supplemental material [167, 168].

4 RQ_1: WHAT TYPES OF SE TASKS HAVE BEEN ADDRESSED BY DL-BASED APPROACHES?

This RQ explores and quantifies the different applications of DL approaches to help improve or automate various SE tasks. Out of the 128 papers we analyzed for this SLR, we identified 23 separate SE tasks where a DL-based approach had been applied. Figure 7 provides a visual breakdown of how many SE tasks we found within these 128 primary studies across a 10 year period. Unsurprisingly, there was very little work done between the years of 2009 and 2014. However, even after the popularization of DL techniques brought about by results achieved by approaches such as AlexNet [90], it took the SE community nearly $\approx$ 3 years to begin exploring the use of DL techniques for SE tasks. This also coincides with the offering and popularization of DL frameworks such as PyTorch and TensorFlow. The first SE tasks to use a DL technique were those of *Source Code Generation*, *Code Comprehension*, *Source Code Retrieval & Traceability*, *Bug-Fixing Processes*, and *Feature Location*. Each of these tasks uses source code as their primary form of data. Source code served as a natural starting point for applications of DL techniques given the interest in large scale mining of open source software repositories in the research community, and relative availability of large-scale code datasets to researchers. Access to a large amount of data and a well-defined task is

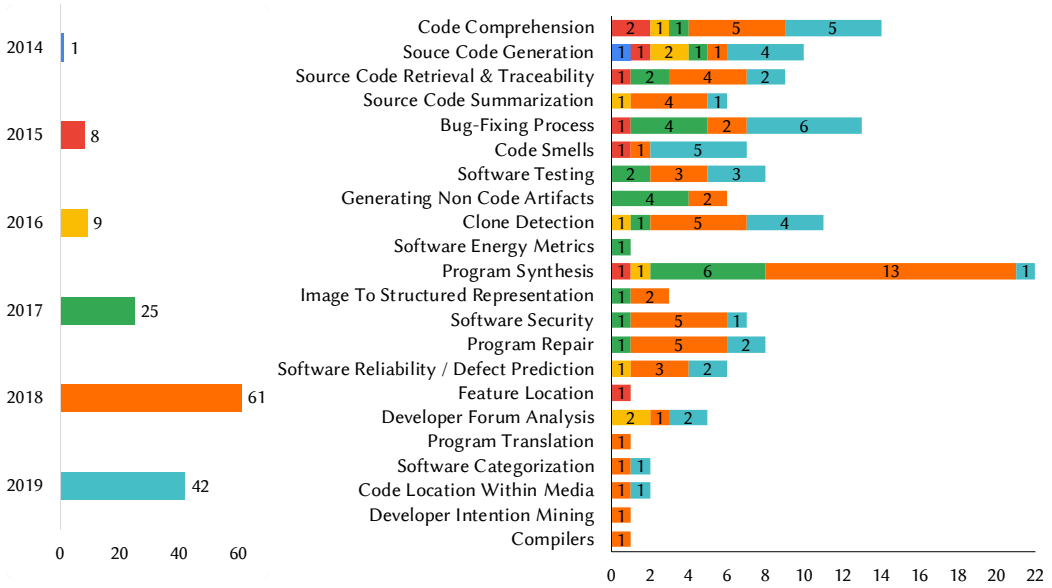

Fig. 7. Papers published per year according to SE task. Note that a single paper can be associated with multiple SE Tasks.

important for DL4SE, since in order for DL to have an effective application two main components are needed: i) a large-scale dataset of data to support the training of multi-parameter models capable of extracting complex representations and ii) a task that can be addressed with some type of predictable target. One of the major benefits of DL implementations is the ability for automatic feature extraction. However, this requires data associated with the predicted target variable.

It was not until 2017 that DL was used extensively in solving SE tasks as shown in Figure 7, with a large increase in the number of papers, more than doubling over the previous year from 9 to 25. During this period, the set of target SE tasks also grew to become more diverse, including tasks such as *Code Smell Detection*, *Software Security*, and *Software Categorization*. However, there are three main SE tasks that have remained the most active across the years: *Code Comprehension*, *Source Code Retrieval & Traceability*, and *Program Synthesis*. The most popular of the three being Program Synthesis, composing a total of 22 papers out of the 128 we collected. We suspect that a variety of reasons contribute to the multiple applications of DL in program synthesis. First and foremost, is that the accessibility to data is more prevalent. Program synthesis is trained using a set of input-output examples. This makes for accessible, high quality training data, since one can train the DL model to generate a program, given a set of existing or curated specifications. The second largest reason is the clear mapping between training examples and a target programs. Given that it has proven difficult to engineer effective features that are capable to predict or infer programs, DL techniques are able to take advantage of the structured nature of this problem and extracting effective hierarchical representations. We display the full taxonomy in Table 3, which associates the cited primary study paired with its respective SE task.

4.1 Results of Exploratory Data Analysis

In performing our exploratory data analysis, we derived two primary findings. First, it is clear that SE researchers apply DL techniques to a diverse set of tasks, as 70% of our derived SE task distribution was comprised of distinct topics that were evenly distributed (≈ 3-5%). Our second

Table 3. SE Task Taxonomy

SE Task	Papers
Code Comprehension	[9, 10, 16, 64, 73, 77, 93, 97, 119, 121, 130, 144, 180, 185]
Souce Code Generation	[26, 37, 57, 63, 74, 84, 124, 136, 151, 173]
Source Code Retrieval & Traceability	[10, 27, 29, 41, 62, 65, 92, 175, 179]
Source Code Summarization	[11, 29, 77, 95, 162, 179]
Bug-Fixing Process	[41, 67, 68, 80, 92, 96, 101, 107, 122, 133, 157, 180, 189]
Code Smells	[9, 49, 106, 108, 117, 155, 157]
Software Testing	[36, 60, 109, 110, 146, 187, 192]
Non Code Related Software Artifacts	[32–34, 78, 82, 94, 143]
Clone Detection	[21, 56, 99, 104, 129, 140, 158, 172, 181, 185, 191]
Software Energy Metrics	[139]
Program Synthesis	[13–15, 20, 22, 30, 42, 44, 47, 48, 58, 75, 83, 102, 105, 123, 128, 137, 145, 149, 186, 195]
Image To Structured Representation	[26, 40, 120]
Software Security	[25, 39, 52, 56, 71, 72, 193]
Program Repair	[17, 67, 68, 72, 107, 160, 163, 171]
Software Reliability / Defect Prediction	[38, 76, 111, 164, 165, 170]
Feature Location	[35]
Developer Forum Analysis	[28, 64, 103, 166, 176]
Program Translation	[31]
Software Categorization	[18, 19]
Compilers	[86]
Code Location Within Media	[126, 190]
Developer Intention Mining	[78]
Software Resource Control	[69, 94]

finding is that the SE task was the most informative feature we extracted ($\approx 4.04B$), meaning that it provides the highest level of discriminatory power in predicting the other features (*e.g.*, elements of learning) related to a given study. In particular, we found that SE tasks had strong correlations to data (1.51B), the loss function used (1.14B) and the architecture employed (1.11B). This suggests that there are DL framework components that are better suited to address specific SE tasks, as authors clearly chose to implement certain combinations of DL techniques associated with different SE tasks. For example, we found that SE tasks such as program synthesis, source code generation and program repair were highly correlated with the preprocessing technique of tokenization. Additionally, we discovered that the SE tasks of source code retrieval and source code traceability were highly correlated with the preprocessing technique of neural embeddings. When we analyzed the type of architecture employed, we found that code comprehension, prediction of software repository metadata, and program repair were highly correlated with both recurrent neural networks and encoder-decoder models. When discussing some of the less popular architectures we found that clone detection was highly correlated with siamese deep learning models and security related tasks were highly correlated with deep reinforcement learning models. Throughout the remaining RQs, we look to expand upon the associations we find to better assist software engineers in choosing the most appropriate DL components to build their approach.

4.2 Opportunities for Future Work

Although the applications of DL-based approaches to SE related tasks is apparent, there are many research areas of interest in the SE community as shown in ICSE'20's topics of interest[3] that DL has not been used for. Many of these topics have no *readily apparent* applicability for a DL-based solution. Still, some potentially interesting topics that seem well suited or positioned to benefit from

[3]https://conf.researchr.org/track/icse-2020/icse-2020-papers#Call-for-Papers

DL-based techniques have yet to be explored by the research community or are underrepresented. Topics of this unexplored nature include software performance, program analysis, cloud computing, human aspects of SE, parallel programming, feature location, defect prediction and many others. Some possible reasons certain SE tasks have yet to gain traction in DL-related research is likely due to the following:

- There is a lack of available, "clean" data in order to support such DL techniques;
- The problem itself is not well-defined, such that a DL model would struggle to be effectively trained and optimized;
- No current architectures are adequately fit to be used on the available data.

We believe that one possible research interest could be in the application of new DL models toward commonly explored SE tasks. For example, a DL model that is gaining popularity is the use of transformers, such as BERT, to represent sequential data [43]. It is possible that models such as this could be applied to topics related to clone detection and program repair. There is sufficient exploration in the use of DL within these topics to determine if these new architectures would be able to create a more meaningful representation of the data when compared to their predecessors.

Summary of Results for RQ_1:

Researchers have applied DL techniques to a diverse set of tasks, wherein *program synthesis*, *code comprehension*, and *source code generation* are the most prevalent. The SE task targeted by a given study is typically a strong indicator of the other details regarding the other components of learning, suggesting that certain SE tasks are better suited to certain combinations of these components. Our associative rule learning analysis showed a strong correlation amongst SE task, data type, preprocessing techniques, loss function used and DL architecture implemented, indicating that the SE task is a strong signifier of what other details about the approach are present. While there has been a recent wealth of work on DL4SE, there are still underrepresented topics that should be considered by the research community, including different topics in software testing and program analysis.

5 RQ_2: HOW ARE SOFTWARE ARTIFACTS BEING EXTRACTED, PREPARED, AND USED IN DL-BASED APPROACHES FOR SE TASKS?

In this research question, we analyze the type of SE data that is modeled by DL approaches applied to various SE tasks. Our aim with this analysis is to understand the various types of SE data used, how the data is extracted or preprocessed, the scale of the data being modeled, and the type of learning applied given the data. All four of these points are crucial to effectively understanding how a given approach is able to model specific software artifacts. These points also ground our discussion regarding future research and potential improvements of data filtering and preprocessing in order to improve the effectiveness of DL approaches; as in many cases, data must be cleaned or filtered in such a manner that can limit the ability of a DL technique to model a desired relationship.

5.1 RQ_{2A}: What types of SE data are being used?

To analyze the types of data being used in DL-based approaches, we provide a high level classification, along with descriptive statistics, as to why some types of data were used for particular tasks. Figure 8 provides an illustration of different data types used in conjunction with different SE tasks. We found that overwhelmingly the most common type of data being used is *source code*, albeit at

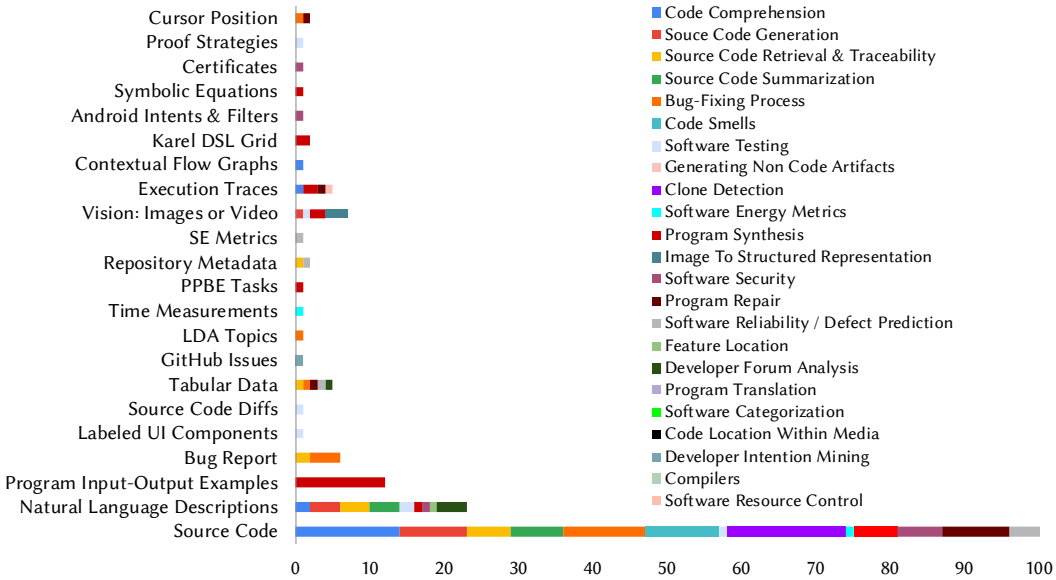

Fig. 8. Data Types With SE Tasks

a range of different granularities. In our identified studies, we found that source code is used at the binary level, code snippet level, method level, class level and project level. Source code is a popular data construct for DL-approaches for a number of reasons. First, source code is plentiful and can be found in a number of different online repositories. This availability helps to appease the "data-hungry" nature of DL techniques in order to learn an effective, representative target function. Second, a majority of SE tasks revolve around the generation, understanding, analysis, and documentation of source code, making it a popular artifact in the (deep) learning process.

In total we identified 152 uses of data in our DL4SE papers, 86 of them are attributed to source code, wherein certain studies utilized more than one type of data for their approach (*e.g.*, source code & natural language). Although source code is the primary type of data that DL models attempt to learn from, we found that the type of data used is heavily dependent on the SE task. Thus, the SE tasks that focus on the comprehension, generation, summarization, and testing of source code will frequently use source code at various granularities as part of their dataset. However, there are many SE tasks that address problems where source code may not be the most representative type of data from which to learn. As an example, the SE task of program synthesis primarily uses 12/27 input and output examples to comprise their dataset. This type of data incorporates attributes, which more strongly correlates to desired relationship for the model to learn. This pattern continues for SE tasks that can learn from textual data, software artifacts, and repository metadata.

Although we identified a variety of different data types used, the data must be accessible in large quantities in order to extract and learn the relevant target hypothesis. With an increase in the number of online source code repositories, opportunities for mining them have increased. This partially explains the dramatic increase in DL papers addressing SE tasks, which was a trend discussed in Section 4. Interestingly, the growth of online repositories does not only increase the amount of accessible source code but also other types of software artifacts. For example, analyzing SE tasks such as software security, bug fixing, etc. have only recently been addressed, in part due to the increasing amount of accessible online repository data/metadata. We anticipate that this trend continues as new datasets are gathered and processed for use in DL techniques.

When exploring online repositories, one attribute of the data that can be problematic for the use of some DL approaches and should be considered is that the data is frequently "unlabeled", meaning that there is no inherent target to associate with this data. For unsupervised learning techniques this is not problematic, however, for supervised learning techniques this type of data is not usable without first establishing a target label for the data. The use of code mined from online software repositories as unlabeled data was explored early in 2015 by White *et al.* [173]. Our findings illustrate that source code is used as an unlabeled, yet useful set of data across several SE tasks including: the localization of buggy files through bug reports [92], specification mining [93], mining fix patterns for bug violations [107], identification of source code clones [158] and repairing vulnerable software [72]. Additionally, we also found that researchers mined unlabeled data and manually labeled it in order to apply a particular DL architecture. We found examples of this when generating accurate method and class names [9], learning proper API sequences [63] and source code summarization [11]. With the increase in the number of online repositories and the variety of unlabeled data within those repositories, we expected the number of DL-based approaches analyzing software artifacts to increase.

5.1.1 Results of Exploratory Data Analysis. Our exploratory data analysis also highlighted these points. Our analysis demonstrated that the main data employed in DL4SE papers are input-output (I/O) examples, source code, natural language, repository metadata and visual data. In fact, these categories of data types comprised ≈ 78.3% of the distribution we found in this SLR. In conjunction with this finding, execution traces and bug reports represent ≈ 5.8% of the distribution of data. The remaining ≈ 15% is comprised of a variety of different data types.

5.1.2 Opportunities for Future Work. Throughout our study, it was clear that certain types of SE artifacts have not yet been analyzed using DL-based approaches. Specifically, we identified that software requirements, software dependencies, and software licenses have not been mined or studied using DL architectures. This suggests that these and other underrepresented data types not included in Fig. 8 could be ripe for future DL applications. In addition to analyzing some of the underrepresented data types, we believe there is an opportunity to combine multiple data types for a more complete representation of the SE data. The ability to consider and combine multiple representations should provide a more meaningful representation of the SE task and could lead to better deep learning solutions. Additionally, there are a number of different data types that could potentially be applied to different SE tasks. For example, contextual flow graphs (CFGs) have only been considered when looking at code comprehension task but information contained in CFGs could prove to be useful for tasks such as code summarization and program repair among others.

Summary of Results for RQ$_{2A}$:

Our analysis found that researchers have explored a variety of different types of SE data in conjunction with DL techniques, with the main types of data utilized being *source code* (≈ 59.78%), *I/O examples* (≈ 7.89%), *natural language* (≈ 12.16%), *repository metadata* (≈ 7.24%), and *visual data* (≈ 5.26%). These data types were often tightly associated with a given SE task.

5.2 *RQ$_{2B}$*: How is this data being extracted and pre-processed into formats that are consumable by DL approaches?

In Section 5.1, we analyzed the types of data that were being used to model complex relationships between software artifacts and the target output function as it relates to a specific SE task. In

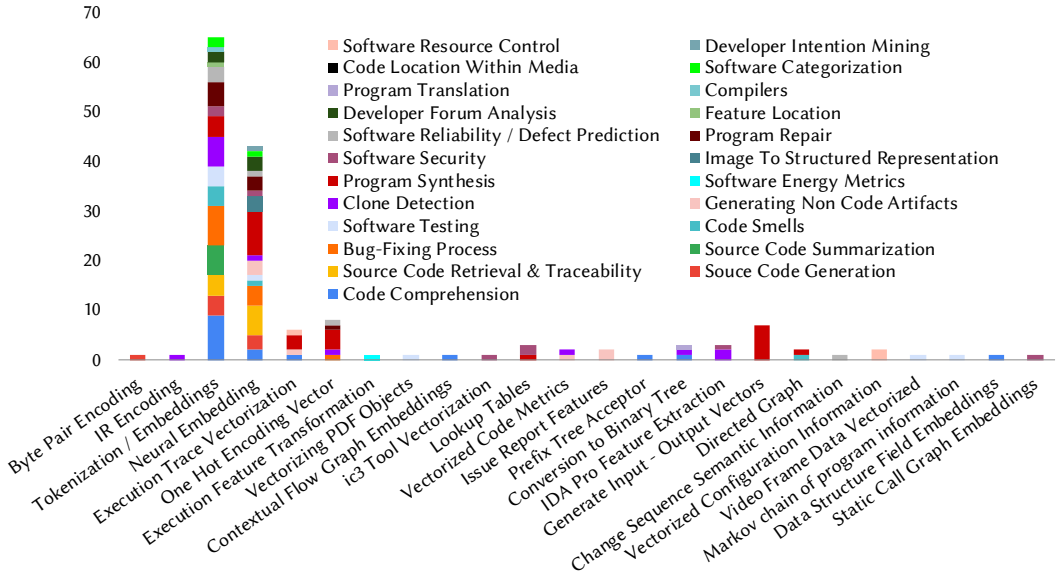

Fig. 9. Preprocessing Techniques by SE Task

RQ_{2b}, we examine the mechanism behind the extraction and preprocessing of this data. Typically, DL models are not amenable to raw data mined from the online repositories. Rather, the data is subjected to a preparation and formatting process before given to the DL model. For example, image data is downsampled and scaled, and text is preprocessed in order to preserve relevant words. This is an extremely important step for those applying DL to SE, since the process can dramatically affect the performance of the model and the resulting target hypothesis. For example, some primary studies represent source code in an abstracted format. This abstraction will inherently transform the features of the code and can affect the applicability of these features to different tasks. Such transformations could also address issues often associated with certain types of DL architectures, such as the "open vocabulary" problem in DL-based language models wherein the vocabulary size of a model is untenable due to source code's unrestricted vocabulary. Conversely, a limitation of these types of transformation processes is that it removes some complex, hierarchical features from the dataset, which can limit the model's ability to accurately predict the target in some cases.

In light of the importance of preprocessing in DL-based approaches, we synthesized a taxonomy of preprocessing and formatting techniques. We also looked to analyze the relationships between the SE tasks, types of data and the preprocessing steps taken. A breakdown of the preprocessing and formatting techniques used in the primary studies we analyzed can be found in Figure 9. It is important to understand that Figure 9 shows the *general* preprocessing and formatting techniques according to SE tasks. However, within the same preprocessing technique, there exist nuances across different studies. There is rarely a standard method for the ability to preprocess the data in such a way that will fit the desired DL model. Therefore, we suggest interested readers refer to the primary studies for more details pertaining to a specific preprocessing or formatting technique of interest. However, there are some dominant techniques researchers use in order to prepare their data to be analyzed by a DL model. Specifically, the use of tokenization and neural embeddings are popular for a variety of different tasks. This was expected, given the prevalence of source code as a primary type of data used. Tokenization of that source code is an effective process for preparing different granularities of code to be fed into a DL architecture.

Even more popular than the use of tokenization was the use of neural embeddings (23.85% of the distribution). This technique uses canonical machine learning techniques or other DL architectures

to process the data, meaning that the output from these other models are then used as input to an additional DL architecture. We found that Word2Vec and recurrent neural networks (RNNs) were the most popular types of models for the preprocessing of data. Source code, natural language, and other repository metadata often have a sequential nature, which can be captured by these techniques. In both cases, the outputs of these models are a series of vectorized numerical values, which capture features about the data they represent. These vectors do not require much additional manipulation before they can be used again as input data to another DL model.

Although tokenization and neural embeddings are the most popular type of preprocessing techniques, there are many more that are required for a different types of data or SE tasks. The ability to accurately represent the data to the DL model is what provides training data for the algorithm to learn from. Any alteration to this process can result in the learning algorithm focusing on different features, leading to an entirely different final hypothesis.

5.2.1 Results of Exploratory Data Analysis. Our exploratory data analysis discovered that the steps taken in preprocessing were strongly dependent on the type of data employed (1.19B, according to the mutual information measure). This is intuitive, as the data and preprocessing techniques are inherently linked. We also found that the SE task and venue had high discriminatory power in determining the type of data preprocessing (given larger self-information values from the correlation analysis). This indicates that certain SE Tasks and preprocessing combinations are more likely for different SE venues.

5.2.2 Opportunities for Future Work. In the midst of our analysis it became clear that preprocessing techniques were often uniquely specialized to particular DL architectures. However, there are many commonalities amongst these preprocessing *pipelines* that could be standardized dependent upon the type of data being represented. For example, tokenization and neural embeddings were used extensively in a variety of tasks. It would be advantageous to standardize the data processing pipelines for these approaches related to source code, input/output examples, or textual data found in repository metadata. Additionally, exploring less popular preprocessing techniques such as directed graphs, lookup tables, and execution trace vectors for different SE tasks could lead to results which are orthogonally beneficial to those found using a more common techniques.

We also noticed many preprocessing techniques have not been formalized based on the SE task and DL architecture used. Additionally, it is not just the use of certain techniques, but the details used in applying those techniques that may lead to an increase or decrease in performance for DL models. For example, tokenization is a broad technique that takes sequential data and separates them into tokens. However, details such as token filtering, removal of stop words and vocabulary size can have a large affect on the process. We believe a more in-depth look at how preprocessing techniques affect the quality of DL solutions for SE tasks would be beneficial for future applications.

Summary of Results for RQ$_{2B}$:

Our analysis found that, while a number of different data preprocessing techniques have been utilized, *tokenization* ($\approx$ 51%) and *neural embeddings* ($\approx$ 35%) are by far the two most prevalent. We also found that data-preprocessing is tightly coupled to the DL model utilized, and that the SE task and publication venue were often strongly associated with specific types of preprocessing techniques. This coupling of paper attributes is likely due to the often tight coupling between preprocessing and allowable inputs of various DL architectures.

5.3 RQ_{2C}: What type of exploratory data analysis is conducted to help inform model design and training?

For RQ_{2C}, we analyzed our primary studies to determine if precautions were taken to limit the number of confounding factors that may exist should researchers have not properly analyzed their datasets prior to training a DL model. Primarily, we were interested in whether *sampling bias* or *data snooping* are present in studies that apply DL models to SE tasks. We found that when DL is applied to an SE task, there were many instances where no methods were discussed to protect against these confounding variables. This can severely limit the conclusions that can be drawn from the learned target hypothesis. Through analyzing the various strategies used to combat sampling bias and data snooping we found that many SE tasks working with source code (*i.e.*, source code generation, source code completion and bug fixing tasks) did not check for *duplicated examples* within their training and testing sets We also found examples of sampling bias in studies pulling code from multiple projects on GitHub. This included SE tasks that analyzed source code clones and software test methods. Extracting data from projects that are only of a particular size, developer, or rating could all lead to biased results that affect study claims. These findings corroborate the findings presented by Allamanis et al. [8], which described the adverse affects of duplicated examples within the training and testing sets of machine learning approaches. This is not only a problem for the SE task of clone detection, but it may also impact the findings of previous DL techniques trained on large-scale code datasets.

Sampling Bias occurs when data is collected from a dataset in such a way that it does not accurately represent the data's distribution due to a non-random selection process [127]. The repercussions of sampling bias normally result in a target hypothesis that is only useful for a small subset of the actual distribution of the data. When sampling bias is not appropriately considered authors can unknowingly make claims about their predicted target function that overestimate the actual capabilities of their model. One effective mechanism for mitigating sampling bias is to limit or carefully consider the types of filtering criteria applied to a dataset.

Data Snooping occurs when data is reused for the purpose of training and testing the model [141]. This means that the data the model was trained on is also incorporated into the testing and evaluation of the model's success. Since the model has already seen the data before, it is unclear whether the model has actually learned the target hypothesis or simply tuned itself to effectively reproduce the results for previously seen data points (overfitting). The overall result of data snooping is an overinflated accuracy reported for a given model. Therefore, it is important that software engineers apply methods to reduce data snooping in order to protect the integrity of their results. This can be done through a simple exploratory data analysis or the removal of duplicate data values within the training and testing sets.

In order to evaluate the extent to which methods that mitigate sampling bias and data snooping were used in our primary studies, we noted instances where the authors conducted some form of exploratory data analysis before training. Ideally, we hoped to see that authors performed exploratory data analysis in order to identify if the dataset that has been extracted is a good representation of the target distribution they are attempting to model. Exploratory data analysis can also provide insight into whether data snooping occurred.

There was a noticeable number of primary studies, $\approx$ 90% and $\approx$ 82% ,that did not mention or implement any methods to mitigate sampling bias or data snooping respectively. However, it should be noted that we focused our analysis on primary studies that included any sign of exploratory analysis on their dataset. This exploratory analysis ranged from basic statistical analyses to an in-depth study into the distribution of the dataset in order to mitigate these confounding factors. A majority of the methods we discovered that addressed sampling bias included putting the fewest

number of limitations on the filtering of source data as possible. For example, when mining GitHub for potential project candidates, studies would normally restrict the results based on necessary attributes of the data. Additionally, some studies included evaluations on entire projects that were not considered during training. A smaller number of studies ensured that data gathered was balanced by class. The studies that did attempt to combat data snooping ($\approx$ 16%) did so by ensuring the removal of duplicates within their dataset. This means that every input-output pairing used for training was unique in both the training and testing sets. Lastly, we found that 41 of the primary studies explicitly mention the use of a validation set to find optimal hyperparameter configurations. The use of this validation set helps to ensure exclusivity in the model evaluation on the test set, bolstering potential generalizability.

5.3.1 Results of Exploratory Data Analysis. Our data analysis showed that nearly $\approx$ 52% of the papers report exploratory techniques for analyzing their datasets. However, even among those papers that did perform some sort of exploratory analysis, most of the techniques utilized were relatively simple (e.g., checking for duplicate data entries). We posit that SE researchers could benefit dramatically from an expanded repertoire of EDA techniques. Furthermore, we detected that approximately 2% of the studies *directly* addressed sampling bias and data snooping. On the one hand, nearly 8% of the inspected approaches are explicitly susceptible to sampling bias. On the other hand, 16% of the data employed are explicitly susceptible to data snooping. In conclusion, we are unable to determine in 90% of the studies whether data snooping or sampling bias were controlled for. Similarly, in 82% of the studies, we were unable to determine the potential for data snooping due to a limited amount of associated descriptions

5.3.2 Opportunities for Future Work. Our primary objective within this RQ was to bring attention to the oversight of confounding factors that affect the treatment and preparation of data for DL models. DL models are heavily dependent on data to properly model a target hypothesis, thus we hope to encourage future work that carefully considers the makeup of their dataset and their methods for extraction. In particular, there are research opportunities related to standardizing the process for preventing sampling bias and data snooping. To our knowledge there has been no attempt at the generation of guidelines related to how one might prevent sampling bias when considering SE data. Likewise, there is potential for the creation of analytical tools to help evaluate the likelihood of data snooping in DL based approaches, or automated tools for combating bias by automatically separating data into a training, validation and testing set that removes duplicate examples between each set. It is also important to note that as time has progressed, the primary studies we analyzed generally included more details about their data exploration process. We hope to see this trend continue as DL is used to address SE tasks.

Summary of Results for RQ$_{2C}$:

Our analysis found that as many as 1/2 of our analyzed studies do not perform any type of exploratory data analysis in order to combat confounding factors such as data snooping or bias. Some of the most popular mitigation techniques employed were a detailed analysis of the duplicates found within the training and testing set, the use of a validation set to prevent tuning the parameters of the model to best perform on a test set, and the removal of restrictive data filtering in order to extract datasets that are as diverse as possible.

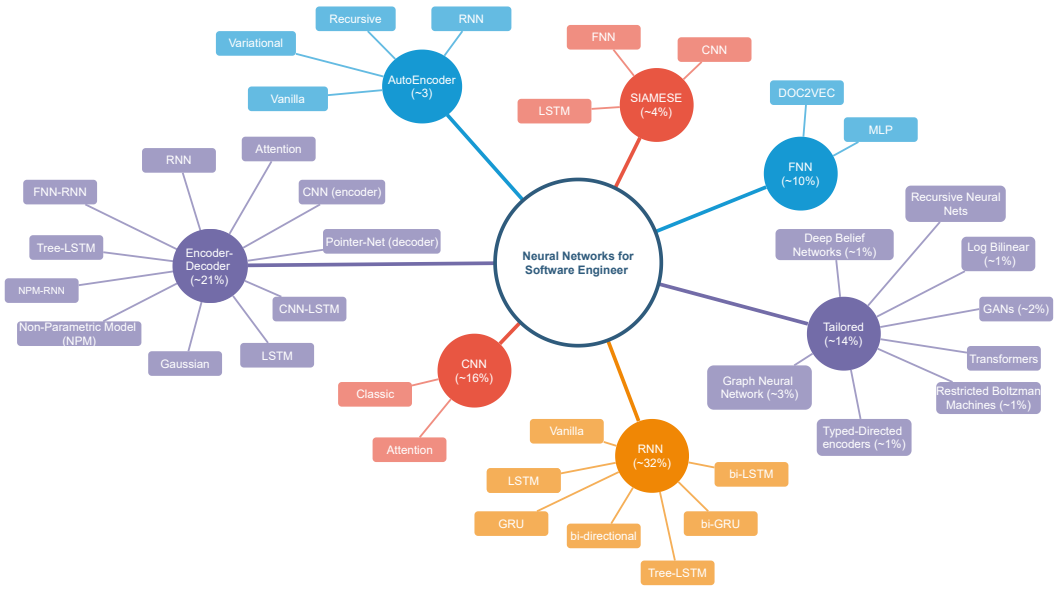

Fig. 10. DL Model Taxonomy & Type Distribution

6 RQ_3: WHAT DEEP LEARNING MODELS ARE USED TO SUPPORT SE TASKS?

In Section 5 we investigated how different types of SE data were used, preprocessed, and analyzed for use in DL techniques. In this section, we shift our focus to the two key components of DL models: the *architecture* and the *learning algorithm*. The type of architecture selected for use in a DL application reveals key aspects of the types of features that researchers hope to model for a given SE task. Thus, we aim to empirically determine if certain architectures pair with specific SE tasks. Additionally, we aim to explore the diversity of the types of architectures used across different SE tasks and whether or not idiosyncrasies between architectures might be important when considering the specific SE task at hand. We also examined how various architectures are used in conjunction with different learning or optimization algorithms. Specifically, we aimed to create a taxonomy of different learning algorithms and determine if there was a correlation between the DL architectures, the learning algorithms and the SE tasks.

6.1 RQ_{3A}: What types of model architectures are used to perform automated feature engineering of the data related to various SE tasks?

In this section, we discuss the different types of DL models software engineers are using to address SE tasks. Figure 10 illustrates the various different DL architecture types that we extracted from our selected studies. We observe seven major architecture types: *Recurrent Neural Networks (RNNs)* ($\approx$ 45%), *Encoder-Decoder Models* ($\approx$ 22%), *Convolutional Neural Networks (CNNs)* ($\approx$ 21%), *Feed-Forward Neural Networks (FNNs)* ($\approx$ 13%), *AutoEncoders* ($\approx$ 8%), *Siamese Neural Networks* ($\approx$ 5%), as well as a subset of other custom, highly tailored architectures. We observe an additional level of diversity within each of these different types of architectures with Encoder-Decoder models illustrating the most diversity, followed by RNNs and the tailored techniques. The diversity of Encoder-Decoder models is expected, as this type of model is, in essence, a combination of two distinct model types, and is therefore extensible to a range of different combinations and hence architectural variations. The variance in RNNs is also logical. RNNs excel in modeling sequential

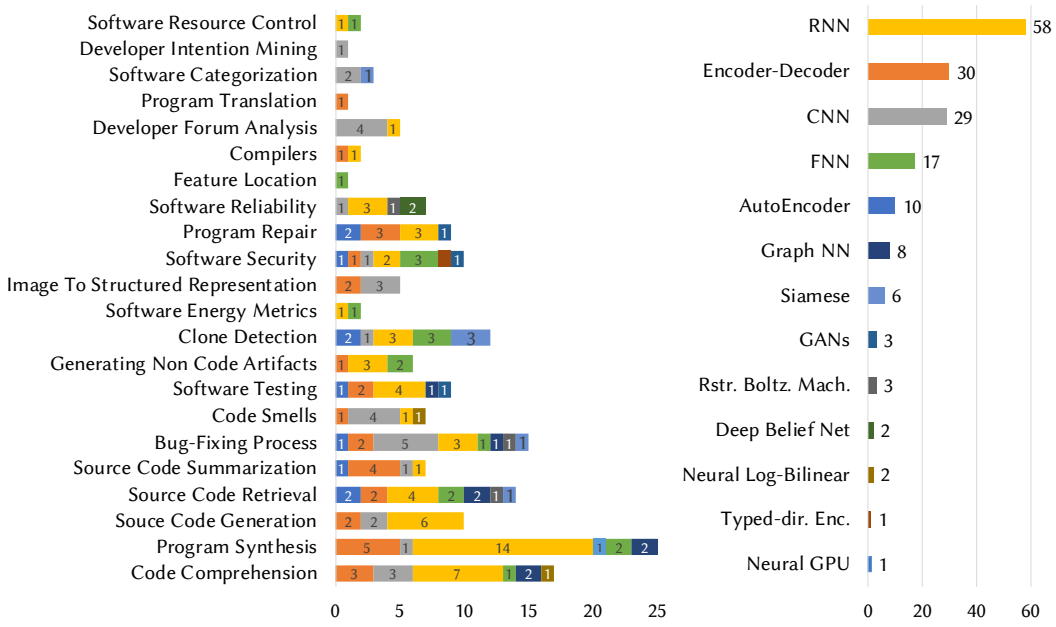

Fig. 11. DL Architectures by the Task

data since the architecture is formulated such that a weight matrix is responsible for representing the features between the sequence of inputs [61], making them suitable to source code. Given that one of the most popular SE data types is source code which is inherently sequential data, the varied application of RNNS is expected. We also observe a number of architectures, such as Graph Neural Networks, that are specifically tailored for given SE tasks. For instances, graph-based neural networks have been adapted to better model the complex *structural* relationships between code entities.

Figure 11 delineates the prevalence of various different types of architectures according to the SE tasks to which they are applied. The popularity of our identified techniques closely mirrors their diversity. Examining this data, we find that RNNs are the most prevalent architectures, followed by Encoder-Decoder models, CNNs, and FNNs. The prevalence of RNNs is not surprising given the prevalence of source code as a utilized data type, as discussed above. The flexibility of Encoder-Decoder models is also expected as they excel at understanding and "translating" between parallel sets of sequential data, which is a common scenario in SE data (*e.g.*, code and natural language). The encoder's responsibility is to translate the raw data into a latent representation that the decoder is capable of understanding and decoding into the target. Therefore, since neural embeddings were such a popular preprocessing technique for data formatting and preparation, it aligns with the high prevalence of the encoder-decoder DL architecture. CNNs serve as the most popular architectures for processing visual data, such as images, and hence are popular for visual SE data.

In addition to the prevalence, we observed certain trends between the DL architecture utilized and the corresponding SE task, as illustrated in Figure 11. As expected, most of the SE tasks having to do with source code generation, analysis, synthesis, traceability, and repair make use of RNNs and encoder-decoder models. Likewise, SE tasks involving the use of images or media data have CNNs commonly applied to them.

We also observed some pertinent trends related to some of the less popular types of DL architectures, including: siamese networks, deep belief networks, GNNs and auto-encoders. While these architectures have only been applied to a few tasks it is important to note that they have only recently gained prominence and become accessible outside of ML/DL research communities. It is possible that such architectures can highlight orthogonal features of SE data that other architectures may struggle to observe. For example, the use of GNNs may better capture the structure or control flow of code or possibly the transition to different mobile screens within a mobile application. There may also be an opportunity for the use of Siamese networks in software categorization, as they have been shown to classify data into unique classes accurately based only on a few examples [140]. One notable absence from our identified architecture types is *deep reinforcement learning*, signaling its relative lack of adoption within the SE community. Deep reinforcement learning excels at modeling decision-making tasks. One could argue that deep reinforcement learning is highly applicable to a range of SE tasks that can be modeled as decisions frequently made by developers. This is a fairly open area of DL in SE that has not been sufficiently explored. The only type of SE task that had an application of Reinforcement Learning was related to program verification. In this paper the authors propose an approach that constructs the structural external memory representation of a program. They then train the approach to make multi-step decisions with an autoregressive model, querying the external memory using an attention mechanism. Then, the decision at each step generates subparts of the loop invariant [146].

In addition to the discussion around the DL architectures and their relations to particular SE tasks, it is also important to understand trends related to the *explicit* and *implicit* features extracted from these different architectures. As we discussed in Section 5.2 (RQ_{2B}), it is common for data to be fed into DL models only after being subjected to certain preprocessing steps. However, in supervised learning, once that data has been preprocessed, the DL model automatically extracts implicit features from the preprocessed data in order to associate those features with a label or classification. In unsupervised learning, the model extracts implicit features from the preprocessed data and groups similar datum together as a form of classification. We refer to the preprocessing steps as highlighting certain explicit features, since these steps frequently perform dimensionality reduction while maintaining important features. In our analysis we found the most common techniques for highlighting explicit features to be tokenization, abstraction, neural embeddings and vectorizing latent representations. These techniques attempt to highlight explicit features that are uniquely tailored to the data being analyzed. Once the data is fed into the model itself, the model is responsible for extracting implicit features to learn a relationship between the input data and target function. The extraction of explicit and implicit features dramatically impacts a DL model's ability to represent a target function, which can be used to predict unobserved data points.

Figure 12 shows a breakdown of DL architectures by the type of data to which they are applied. This relationship between data and architecture is important since the architecture is partially responsible for the type of implicit features being extracted. For example, images and other visual data are commonly represented with a CNN. This is because CNNs are particularly proficient at modeling the spatial relationships of pixel-based data. We also discovered a strong correlation between RNNs and sequential data such as source code, natural language and program input-output examples. This correlation is expected due to RNNs capturing implicit features relating to the sequential nature of data. The models are able to capture temporal dependencies between text and source code tokens. Another correlation we observed was the use of CNNs for visual data or data which requires dimensionality reduction. This included the data types of images, videos, and even natural language and source code. CNNs have the ability to reduce features within long sequential data which makes them useful for tasks involving sentiment analysis or summarization. We also observed less popular combinations such as the use of deep belief networks (DBNs) for defect

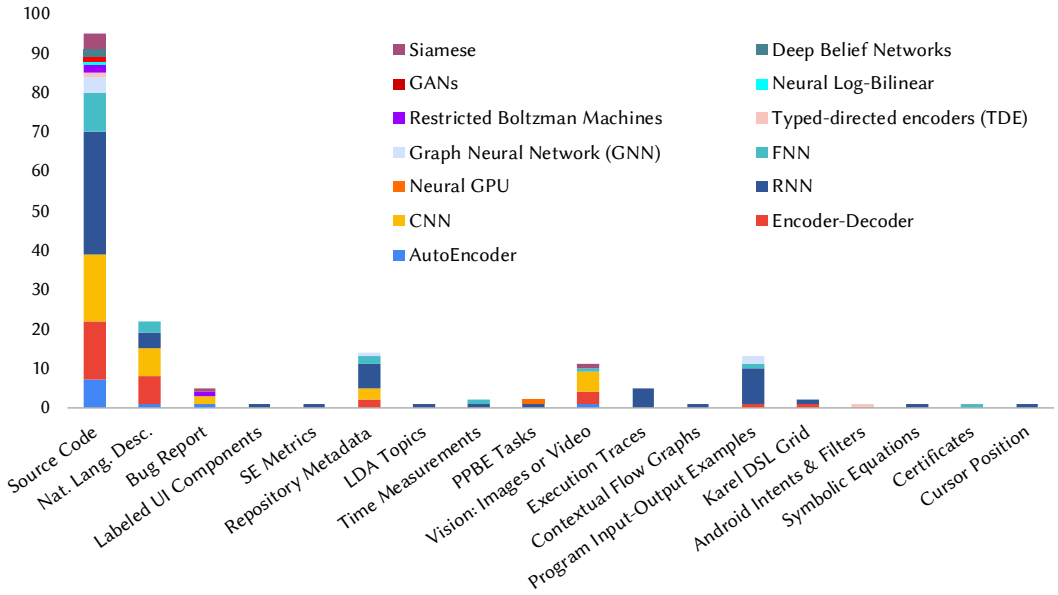

Fig. 12. DL Architectures by Data Type

prediction [165]. Here, a DBN is used to learn semantic features of token vectors from a program's AST graph to better predict defects. A DBN can be a useful architecture in this situation due to its ability to extract the necessary semantic features from a tokenized vector of source code tokens. Those features are then used within their prediction models to drastically increase performance.

6.1.1 Results of Exploratory Data Analysis. In our exploratory data analysis, we found that SE tasks greatly influence the architecture adopted in an approach. The mutual information value between the features of a SE task and a DL architecture is 1.11*B*. We also note that the SE research landscape has primarily focused on SE tasks that consist primarily of text-based data, including source code. This helps to explain why RNNs are used in $\approx$ 45% of the papers analyzed in this SLR. The encoder-decoder architecture was also seen frequently ($\approx$ 22% of papers), which generally makes use of RNNs.

6.1.2 Opportunities for Future Work. We were able to correlate different DL architectures with particular SE tasks and data types, primarily due to the fact that a given architecture is typically suited for a specific type of implicit feature engineering. However, there exists a fundamental problem in the ability of current research to validate and quantify these implicit features the model is extracting. This leads to decreased transparency in DL models, which in turn, can impact their practical applicability and deployment for real problems. Thus, there exists an open research problem related to being able to explain how a given model was capable of predicting the target function, *specifically* as it relates to SE data [12, 59, 174, 178, 182]. While interpretability is a broader issue for the DL community, insights into implicit feature engineering specifically for SE data would be beneficial for DL4SE work. It is necessary for developers to understand what complex hierarchical features are used by the model for this prediction. This could demystify their ability to correctly predict the output for a given input datum.

The ability to increase the interpretability of DL4SE solution also contributes toward the novel field of SE4DL, where SE methods are applied to the creation, sustainability and maintenance of DL software. The ability to interpret DL based solutions could help to create more complete testing suites for DL based software. This paradigm becomes even more important as new and innovative

DL architectures are being developed. The SE community could take inspiration from the recent success in the NLP community on developing benchmarks for explainability [138]. Peeling back the "black box" nature of DL models should allow for an analysis on the integrity of the learning algorithms and an ability to better understand and build usable tools around their predictions.

Summary of Results for RQ$_{3A}$:

Our analysis revealed seven major types of DL architectures that have been used in work on DL4SE including: *Recurrent Neural Networks (RNNs)* ($\approx$ 45%), *Encoder-Decoder Models* ($\approx$ 22%), *Convolutional Neural Networks (CNNs)* ($\approx$ 21%), *Feed-Forward Neural Networks (FNNs)* ($\approx$ 13%), *AutoEncoders* ($\approx$ 8%), *Siamese Neural Networks* ($\approx$ 5%), as well as a subset of other custom, highly tailored architectures. RNNs and Encoder-Decoder models were both the most prevalent architecture used in our surveyed studies and the most diverse in terms of their varying configurations. We also discovered strong correlations between particular DL architectures to data types. For example, we found that architectures capable of capturing temporal differences within sequential data are used to study source code, natural language, repository metadata and program input-output examples. Likewise, architectures capable of capturing spatial and structural features from data have been used to study images, bug reports and program structures (ASTs, CFGs, etc.).

6.2 *RQ$_{3B}$*: What learning algorithms and training processes are used in order to optimize the models?

In addition to the variety of DL models that can be used within a DL-based approach, the way in which the model is trained can also vary. To answer RQ$_{3B}$ we aimed to analyze the learning algorithms used in three primary ways: according to (i) the manner in which the weights of the model are updated, (ii) the overall error calculation, and (iii) by the optimization algorithm, which governs the parameters of the learning algorithm as training progresses. Learning algorithms that have been defined in ML/DL research are typically used in an "off-the-shelf" manner, without any alteration or adjustment, in the context of SE research. This is likely a result of researchers in SE being primarily interested in DL applications, rather than the intricacies of learning algorithms.

In terms of the process for adjusting weights, the most prevalent technique employed among our analyzed studies was the incorporation of the gradient descent algorithm. The breakdown of learning algorithms throughout our SLR are as follows: We found $\approx$ 76% of the primary studies used some version of gradient descent to train their DL model. The remaining studies used gradient ascent $\approx$ 2%, or policy based learning $\approx$ 2%. Other studies did not explicitly specify their learning algorithm in the paper $\approx$ 18%. Our exploratory data analysis revealed that papers published in recent years (2018 and 2019) have begun to employ learning algorithms that differ from gradient descent, such as reward policies or gradient ascent.

Our analysis reveled that there are a variety of ways that DL-based implementations calculate error. However, we did find that a majority of the papers we analyzed used cross entropy as their loss function $\approx$ 20%, which was most commonly paired with gradient descent algorithms. Other common loss functions that were used with gradient descent algorithms were negative log likelihood ($\approx$ 9%), maximum log likelihood ($\approx$ 9%), and cosine loss ($\approx$ 2%). There were a number of papers which did not provide any indication about the loss function within their learning algorithm ($\approx$ 42%). We did find that when the primary study was not using gradient descent as a way to adjust

the weights associated with the DL model, the error functions used became a lot more diverse. For example, the work done by Ellis *et al.* learned to infer graphical programs from deep learning hand-drawn images. They used gradient ascent rather than descent as their learning algorithm and also used surrogate likelihood function as a way to calculate the error of the model [48]. We found that approaches that implement reinforcement algorithms are based on a developed policy, which calculates the error associated with the action taken by the model and adjusts the weights.

Lastly, we examined the use of optimization algorithms to determine if there were any relevant patterns. We discovered that the choice of optimization algorithm is somewhat agnostic to the model, the weight adjustment algorithm and the error function. In many cases, the optimization algorithm was not reported within the primary study ($\approx$ 53% of the time). However, we did analyze the papers that provided this information and identified four major optimization algorithms: Adagrad (3) , AdaDelta (3), RMSprop (11), and Adam (30). Below, we briefly address each optimization algorithm in order to point out potential situations in which they should be used.

Adagrad is an algorithm that adapts the learning rate based on the impact that the parameters have on classification. When a particular parameter is frequently involved in classification across multiple inputs, the amount of adjustment to those parameters is lower. Likewise, when the parameter is only associated with infrequent features, then the adjustment to that parameter is relatively high [46]. A benefit of AdaGrad is that it removes the need for manual adjustment of the learning rates. However, the technique that AdaGrad calculates the degree by which it should adjust the parameters is using an accumulation the sum of the squared gradients. This can lead to summations of the gradient that are too large, often requiring an extremely small learning rate.

AdaDelta was formulated out of AdaGrad in order to combat the gradient size problem. Rather than consider all the sums of the past squared gradients, AdaDelta only considers the sum of the past squared gradients limited to a fixed size. Additionally, this optimization algorithm does not require a default learning rate as it is defined by an exponentially decaying average of the calculated squared gradients up to a fixed size [184].

RMSprop is the next optimization algorithm, however, this algorithm has not been published or subjected to peer review. This algorithm was developed by Hinton *et al.* and follows the similar logic of AdaDelta. The way in which RMSprop battles the diminishing learning rates that AdaGrad generates is by dividing the learning rate by the recent average of the squared gradients. The only difference is that AdaDelta uses the root means squared error in the numerator as a factor that contributes to the adjustment of the learning rate where RMSprop does not.

Adam, the last of our optimization algorithms discussed, also calculates and uses the exponentially decaying average of past squared gradients similar to AdaDelta and RMSprop. However, the optimization algorithm also calculates the exponentially decaying average of the past gradients. Keeping this average dependent on gradients rather than just the squared gradients allows Adam to introduce a term which mimics the momentum of how the learning rate is moving. It can increase the rate at which the learning rate is optimized [88].

6.2.1 Results of Exploratory Data Analysis. We found that the loss function is correlated to the chosen technique to combat overfitting with a mutual dependence of $1.00B$. However, the SE community omits reporting the loss function in $\approx$ 33% of the papers we analyzed. Additionally, the loss function is correlated to SE task with a mutual dependence of $1.14B$

6.2.2 Opportunities for Future Work. A consequential highlight of our analysis of employed learning algorithms was the lack of data available from the primary studies. However, we did find a strong correlation between certain loss functions paired to specific learning algorithms. One aspect we believe could provide vital insight into the DL process is an analysis regarding how learning algorithms affect the parameters of the model for certain types of data. It would not only be

important to study the type of data that learning algorithms and loss functions are associated with, but also what preprocessing techniques influence the learning algorithms and loss functions chosen. It is possible that some loss functions and learning algorithms are more efficient when applied to data that has been subjected to a particular preprocessing technique. Finding the optimal pairing of loss function and learning algorithm for an architecture/data pair remains an open problem.

Summary of Results for RQ_{3B}:

Our analysis revealed four different techniques for updating the weights of the DL models, with the large majority making use of gradient descent. We found four major techniques that were utilized for calculating error, including cross entropy ≈ 20%, negative log likelihood ≈ 9%, maximum log likelihood ≈ 9%, and cosine loss ≈ 2%– with cross entropy being the most prevalent. Finally, we observed the use of four major optimization algorithms, including Adagrad (3) , AdaDelta (3), RMSprop (11), and Adam (30).

6.3 RQ_{3C}: What methods are employed to combat over- and under-fitting?

Two potential problems associated with the use of any type of learning based approach, whether that be canonical machine learning or deep learning, are *overfitting* and *underfitting*. Both of these issues are related to the notion of generalization, *i.e.*, how well does a trained ML/DL model perform on unseen data. Overfitting is the process of a model learning to fit the training data extremely well, yet not being able to generalize to unseen data, and hence is a poor approximation of the actual target function to be learned [154]. Underfitting is typically described as the scenario in which a given model incurs a high error rate on a training set. This can occur when the model lacks the necessary complexity, is overly constrained, or has not had the sufficient training iterations to appropriately approximate the target function. For RQ_{3C}, we are primarily interested in the specific methods employed by researchers to combat these two problems in the context of SE tasks.

Figure 13 provides an overview of some general methods used to combat overfitting and underfitting[4]. The figure also addresses what parts of an ML/DL approach are affected by these techniques. As illustrated, there are three main types of regularization. The first regularizes the model, which includes things such as adding Dropout layers [147] or Batch Normalization [81]. The second regularizes the data itself, either through adding more data or cleaning the data already extracted. The third type of regularization is applied to the training process, which modifies the loss function with L1 regularization, L2 regularization or incorporates early stop training.

As outlined in [5], the use of a validation set is a commonly used method for detecting if a model is overfitting or underfitting to the data, which is why it is very common to split data into training, validation and evaluation sets. The splitting of data helps to ensure that the model is capable of classifying unseen data points. This can be done in parallel with a training procedure, to ensure that overfitting is not occurring. We see cross-validation in ≈ 11% papers we analyzed. However, other potentially more effective techniques were seen less frequently.

We aimed to determine if a given SE task had any relationship with the methods employed to prevent over/under-fitting. Figure 14 analyzes the relationship between DL approaches and

[4]Generated through an analysis of the following sources: *https://elitedatascience.com/overfitting-in-machine-learning*, *https://hackernoon.com/memorizing-is-not-learning-6-tricks-to-prevent-overfitting-in-machine-learning-820b091dc42*, *https://towardsdatascience.com/dont-overfit-how-to-prevent-overfitting-in-your-deep-learning-models-63274e552323*, *https://elitedatascience.com/bias-variance-tradeoff*

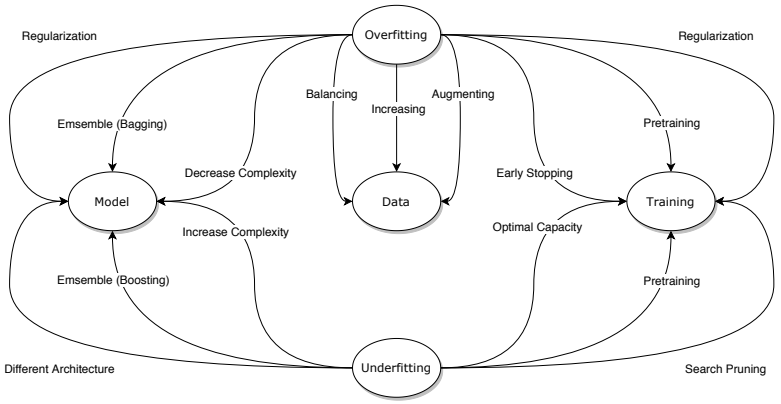

Fig. 13. Overfitting and Underfitting Overview

the techniques that combat overfitting. This figure shows that there are some techniques that are much more commonly applied to SE tasks than others. For example, *dropout* (≈ 32%) was the most commonly used regularization technique and is used in a variety of DL approaches that address different SE tasks, followed by *data cleaning* (≈ 14%), *L1/L2 regularization* (≈ 15%), and *early stopping* (≈ 13%). Dropout is one of the most popular regularization techniques because of its effectiveness and ease of implementation. Dropout randomly blocks signals from a node within a layer of the neural network with a certain probability determined by the researcher. This ensures that a single node doesn't overwhelmingly determine the classification of a given data point. We also observed a number of custom methods that were employed. These methods are configured to address the specific neural network architecture or data type being used. For example, in Sun et al. [149], they encourage diversity in the behavior of generated programs by giving a higher sampling rate to the perception primitives that have higher entropy over K different initial states. In Delvin et al. [42] they perform multiple techniques to combat overfitting which include the even sampling of the dataset during training and ensuring that each I/O grid of every example is unique. In addition to the aforementioned techniques, we found a subset of more unique approaches including the use of deep reinforcement learning instead of supervised learning [162], gradient clipping, lifelong learning [58], modification of the loss function [20], pretraining [146, 162], and ensemble modeling [82].

We also analyzed the relationships between techniques to combat over/under-fitting, and the underlying data type that a given model operated upon. We observed similar patterns in that there are a variety of techniques to combat overfitting regardless of the data type. The only exception to this pattern was seen when analyzing natural language, where L1/L2 regularization was predominately used. Figure 14 illustrates that the techniques used to combat overfitting do not have a strong association with the SE task. Therefore, we observe that a range of different techniques are applicable across many different contexts.

One of the more concerning trends that we observed is the number of papers categorized into the *Did Not Discuss* (≈ 19%) category. Given the importance of combating overfitting when applying a DL approach, it is troublesome that so many primary studies did not mention these techniques. We hope that our observation of this trend signals the importance of recording such information.

Combating underfitting is a more complex process, as there aren't a well-defined set of standard techniques that are typically applied. One method that can be used to combat underfitting is searching for the optimal capacity of a given model. The optimal capacity is the inflection point where the model starts to overfit to the training data and performs worse on the unseen validation set. One technique for achieving this optimal capacity include maximizing training time while

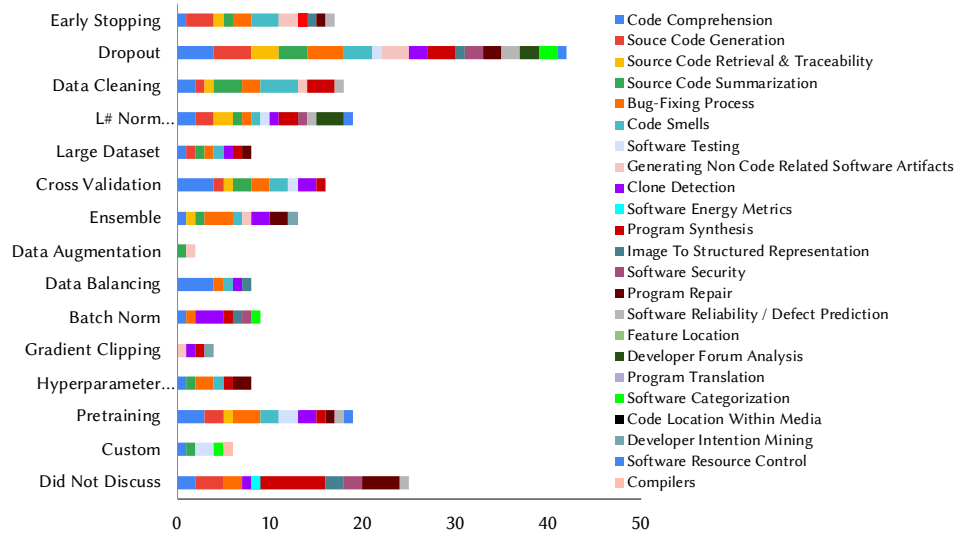

Fig. 14. Overfitting Techniques per Task Type

monitoring performance on validation data. Other techniques include the use of a more complex model or a model better suited for the target function, which can be determined by varying the number of neurons, varying the number of layers, using different DL architectures, pretraining the model, and pruning the search space. From our SLR, the most commonly used underfitting techniques applied were pruning the search space of the model [30, 83], curriculum training [30, 137, 186] and pretraining [146, 162]. We found that only 6/128 primary studies explicitly stated the implementation of an underfitting technique. This is a stark contrast to the number of studies implementing an overfitting technique, 97/128 .

Surprisingly, more than ≈ 19% of our studied papers did not discuss any techniques used to combat overfitting or underfitting. Combating this issue is a delicate balancing act, as attempting to prevent one can begin to cause the other if the processes are not carefully considered. For example, having a heavily regularized learning model to prevent overfitting to a noisy dataset can lead to an inability to learn the target function, thus causing underfitting of the model. This is also possible while attempting to prevent underfitting. An increase in the number of parameters within the architecture to increase the complexity of the model can cause the model to learn a target function that is too specific to the noise of the training data. Therefore, the incorporation of techniques to address over- and under-fitting is crucial to the generalizability of the DL approach.

6.3.1 Opportunities for Future Research. Given the relative lack of discussion of techniques to combat the over- and under-fitting observed in our studies, it is clear that additional work is needed in order to better understand different mitigation techniques in the context of SE tasks and datasets, culminating in a set of shared guidelines for the DL4SE community. In addition, more work needs to be done to analyze and understand specialized techniques for SE tasks, data types, and architectures. Similar to preprocessing data, the implementation of over- and underfitting techniques are subject to a set of variables or parameters that define how they work. An in-depth analysis on how these details and parameters change depending on the type of SE task, architecture or data, is beyond the scope of this review. However, it would be useful to the SE community to provide some intuition about what combination of over- and underfitting techniques to apply and what parameters inherent to those techniques will likely lead to beneficial results.

Summary of Results for RQ_{3c}:

Our analysis shows that *dropout* (≈ 32%) was the most commonly used method to combat over/under-fitting, followed by *data cleaning* (≈ 14%), *L1/L2 regularization* (≈ 15%), and *early stopping* (≈ 13%). Nearly 1/4 of papers did not discuss such techniques.

7 RQ4: HOW WELL DO DL TASKS PERFORM IN SUPPORTING VARIOUS SE TASKS?

In this RQ, we aim to explore the impact that DL4SE research has had through an examination of the effectiveness of the techniques proposed in our selected studies. we primarily analyze metrics on a per task basis and summarize the current state of benchmarks and baselines in DL4SE research.

7.1 RQ_{4A}: What "baseline" techniques are used to evaluate DL models and what benchmarks are used for these comparisons?

For RQ_{4A}, we examine the baseline techniques and evaluation metrics used for comparison in DL4SE work. In general, while we did observe the presence of some common benchmarks for specific SE tasks, we also found that a majority of papers self-generated their own benchmarks. We observed that baseline approaches are extremely individualized, even within the same SE task. Some DL4SE papers do not compare against any baseline approaches while others compare against 3-4 different models. Therefore, we included the listing of baselines that each paper compared against in our supplemental material [167, 168]. We found that many of the baseline approaches were canonical machine learning models or very simple neural networks. We suspect the reason for this is in part due to DL4SE being a relatively new field, meaning that there were not many available DL-based approaches to compare against. As the field of DL4SE begins to mature, we expect to see a transition to evaluations that include comparisons against previously implemented DL approaches.

One somewhat troubling pattern that we observed is that many model implementations do not include a publicly available implementation of a DL approach. This, in part, explains why there are so many highly individualized, baseline approaches. Since researchers do not have access to common baselines used for comparison, they are forced to implement their own version of a baseline. The robustness of the results of such papers may suffer from the fact that many papers did not include any information about the baselines themselves. Additionally, a unique implementation of the same baselines could lead to confounding results when attempting to examine purported improvements. While we expect that the set of existing, publicly available baselines will continue to improve over time, we also acknowledge the need for well-documented and publicly available baselines, and guidelines that dictate their proper dissemination.

Our online appendix [167, 168] includes a list of all the benchmarks and baselines used for each paper within our SLR. The diversity and size of this list of benchmarks prohibited its inclusion to the text of this manuscript. However, we recorded the number of primary studies that used a previously curated benchmark as opposed to ones that curated their own benchmark. We noted that there is an overwhelming number of self-generated benchmarks. Additionally, we classified self-generated benchmarks into those that are publicly available and those that are not. Unfortunately, we found a majority of self-generated benchmarks may not be available for public use. The full breakdown of benchmarks used in the primary studies can be seen in Figure 15. This trend within DL4SE is worrying as there are few instances where DL approaches can appropriately compare against one another with available benchmarks. We hope that our online repository aids researchers by providing them with an understanding about which benchmarks are available for an evaluation

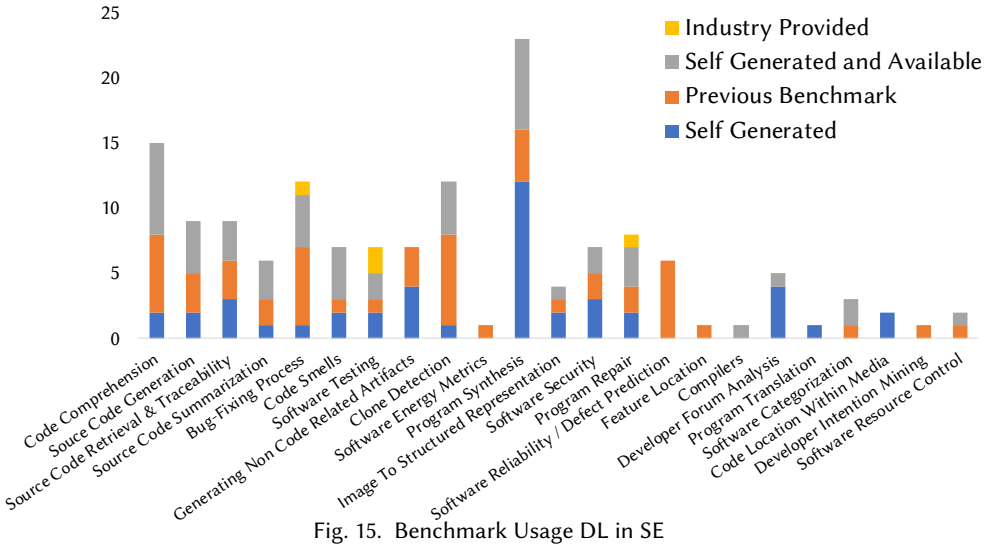

Fig. 15. Benchmark Usage DL in SE

of their approach within a specific SE task. Additionally, we urge future researchers to make self-generated benchmarks publicly available, which will provide a much needed resource not only for comparisons between approaches, but also for available data applicable to DL techniques.

Although the use of previously established benchmarks was not common among our studies, we did observe a subset of benchmarks that were used multiple times within our primary studies. For the SE task of clone detection, we found that the dataset BigCloneBench [152] was used frequently to test the quality of the DL frameworks. Also, for the task of defect prediction, we saw uses of the PROMISE dataset [142] as a way to compare previous DL approaches that addressed defect prediction in software.

7.1.1 Opportunities for Future Research. The use of baselines and benchmarks in DL4SE studies, for the purpose of evaluation, is developing into a standard practice. However, there exists a need for replicable, standardized, baseline approaches that can be used for comparison when applying a new DL approach to a given SE task. The baseline implementations should be optimized for the benchmark used as data for a non-biased evaluation. This requires a thorough and detailed analysis of each published benchmark, within a specific SE task, for high quality data that does not suffer from sampling bias, class imbalance, etc. Many of the primary studies used a comparative approach for their evaluation, however, with a lack of standardized baselines the evaluations are dependent on how optimized the baseline is for a particular dataset. This can lead to confusing or conflicting results across SE tasks. We have started to see recent progress in the derivation and sharing of large-scale datasets with efforts such as the CodeXGlue dataset from Microsoft [112].

Summary of Results for RQ$_{4A}$:

Our analysis revealed a general lack of well-documented, reusable baselines or benchmarks for work on DL4SE. A majority of the baseline techniques utilized in the evaluations of our studied techniques were self-generated, and many are not publicly available or reusable. While a subset of benchmark datasets do exist for certain tasks, there is a need for well-documented and vetted benchmarks.

Table 4. Metrics Used for Evaluation

Measurement Type	Metrics	Studies
Alignment Scores	Rouge-L	[162]
	BLEU Score	[26, 29, 57, 63, 72, 77, 82, 95, 151, 162]
	METEOR Score	[29, 162]
Classification Measures	Precision	[9, 13, 32, 33, 41, 62, 65, 71, 73, 92, 94, 99, 106, 140, 159, 165, 172, 176] [18, 39, 49, 64, 78, 80, 101, 103, 108, 117, 119, 120, 126, 129, 155, 158, 166, 170, 175, 179, 181, 185, 190, 191]
	Recall	[9, 13, 32, 33, 39, 41, 65, 71, 73, 78, 94, 99, 103, 104, 106, 140, 165, 170, 176] [18, 27, 64, 64, 101, 108, 117, 119, 126, 129, 133, 155, 158, 166, 175, 179–181, 185, 190, 191]
	Confusion Matrix	[120, 133]
	Accuracy	[16, 17, 20, 22, 28, 30, 40–42, 44, 58, 71, 74, 83, 96, 97, 99, 105, 122, 123, 130, 137, 149, 176, 186] [10, 13, 19, 26, 31, 34, 37, 48, 56, 68, 69, 72, 73, 75, 78, 94, 102, 103, 111, 121, 124, 126, 133, 140, 145, 151, 157, 163, 171, 180, 195]
	ROC/AUC	[21, 32, 33, 38, 39, 56, 76, 111, 140, 144, 170, 170, 179, 193]
	F-Score	[9, 11, 18, 32, 33, 38, 39, 49, 64, 71, 78, 93, 94, 101, 106, 108, 117, 129, 144, 155, 158, 165, 166, 170, 175, 176, 181, 185, 190, 191, 193]
	Matthews Correlation	[33, 155]
	Scott-Knott Test	[111]
	Exam-Metric	[189]
	Clustering-Based	[52]
Coverage & Proportions	Rate or Percentages	[25, 36, 49, 62, 67, 110, 164, 180, 192]
	Coverage-Based	[60, 109, 110, 128, 166, 187]
	Solved Tasks	[15, 47, 67, 146, 170]
	Cost-Effectiveness	[111, 171]
	Total Energy or Memory Consumption	[139]
Distance Based	CIDER	[162, 195]
	Cross Entropy	[75]
	Jaccard Distance	[123]
	Model Perplexity	[38, 86, 173]
	Edit Distance	[57, 86]
	Exact Match	[57]
	Likert Scale	[82]
Approximation Error	Mean Absolute Error	[33, 34]
	Minimum Absolute Difference	[123]
	Macro-averaged Mean Absolute Error	[32, 33]
	Root Mean Squared Error	[143]
	Median Absolute Error	[34]
	Macro-averaged Mean Cost Error	[32]
Ranking	F-Rank	[62]
	Top K - Based	[80, 107, 120, 136, 145]
	Spearmans Rank	[158]
	MRR	[27, 29, 35, 62, 74, 80, 84, 92]
	Kruskal's γ	[193]
Timing	Time	[14, 47, 48, 69, 172]

7.2 RQ_{4B}: How is the impact or automatization of DL approaches measured and in what way do these models promote generalizability?

Table 4 describes the distribution of metrics found in this SLR. In our analysis of utilized metrics within work on DL4SE, we observed that the metrics chosen are often related to the type of learning. Therefore, many of the supervised learning methods have metrics that analyze the resulting hypothesis, such as the accuracy (≈ 46%), precision (≈ 35%), recall (≈ 33%), or F1 measure (≈ 26%). In fact, classification metrics are reported in ≈ 74% of the papers. These metrics are used to compare the supervised learning algorithms with the outputs representing the target hypothesis. Intuitively, the type of metric chosen to evaluate the DL-based approach is dependent upon the data type and architecture employed by the approach. The "other" category illustrated in Figure 4 is comprised of less popular metrics including: *likert scale*, *screen coverage*, *total energy consumption*, *coverage of edges*, *ROUGE*, *Jaccard similarity*, *minimum absolute difference*, *cross entropy*, *F-rank*, *top-k generalization*, *top-k model-guided search accuracy*, *Spearman's rank correlation coefficient*, and *confusion matrices*. In addition to the use of these metrics, we found a limited number of statistical tests to support the comparison between two approaches. These statistical tests included: *Kruskal's γ*, *macro-averaged mean cost-error*, *Matthew's correlation coefficient*, and *median absolute error*. Surprisingly, only approximately 5% of papers made use of statistical tests.

We classified each primary study into seven categories, which represents the major contribution of the work. The result of this inquiry can be seen in Figure 16. We found three primary objectives that the implementation of a DL model is meant to address: (i) in ≈ 43% of papers observed, a DL approach was implemented with the main goal of increasing automation efficiency; (ii) in ≈ 24% of the papers observed, a DL approach was implemented with the main goal of advancing

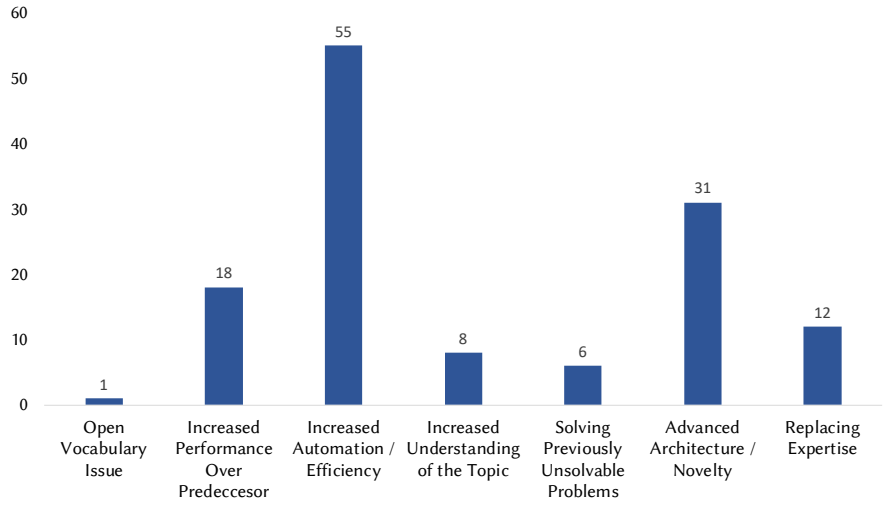

Fig. 16. Impact of DL4SE

or introducing a novel architecture; (iii) in ≈ 14% of the papers observed, a DL approach was implemented with the main goal of increasing performance over a prior technique.

In addition to the primary studies major objectives, we also observed that many papers did not analyze the complexity or generalizability of their implemented models. Thus to examine this further, we analyzed our primary studies through the lends of Occam's Razor and model efficiency. A valid question for many proposed DL techniques applied to SE tasks is whether the complexity of the model is worth the gains in effectiveness or automation for a given task, as recent research has illustrated [55]. This concept is captured in a notion known as *Occam's Razor*. Occam's Razor is defined by two major viewpoints: 1) "Given two models with the same generalization error, the simpler one should be preferred because simplicity is desirable" [45], 2) "Given two models with the same training-set error, the simpler one should be preferred because it is likely to have lower generalization error" [45]. In the context of our SLR, we aimed to investigate the concept of Occam's Razor through analyzing whether authors considered technically "simpler" baseline techniques in evaluating their approaches. In Figure 17 we break the primary studies into four groups: 1) those that compare against less complex models and analyze the results; 2) those that manipulate the complexity of their own model by testing a variety of layers or nodes per layer; 3) those that perform both; 4) those that did not have any Occam's Razor consideration. Note that these are overlapping groupings and so the sum of papers exceeds the number of papers in our SLR.

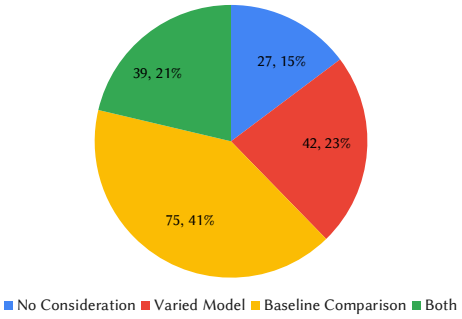

Fig. 17. Evidence of Occam's Razor

Although a majority of the primary studies do consider Occam's Razor, there are still ≈ 16% of DL4SE studies that do not consider the principle. Without a consideration of Occam's Razor, it is possible that a canonical machine learning model or a simple statistical based approach could yield an optimal solution. This idea coincides with the findings mentioned by Fu et al. [55], who discovered that by applying a simpler optimizer to fine tune an SVM they were able to outperform a DL model applied to the same task. Fu *et al.* warn

against the blind use of DL models without a thorough evaluation regarding whether the DL technology is a necessary fit for the problem [55]. Interestingly, in ≈ 23% of the primary studies, the author's considered Occam's Razor by adjusting the complexity of the model being evaluated. This is done by varying the number of layers, the number of nodes, the size of embeddings, etc. The downside to this method is that there is no way to determine if the extraction of complex hierarchical features is more complex than what is necessary to address the SE task. The only way to properly answer this question is to compare against baseline approaches that are not as complex. In our DL4SE studies, this often took the form of a comparison to a canonical ML technique.

7.2.1 Results of Exploratory Data Analysis. Our exploratory data analysis revealed papers that combat overfitting, excluding data augmentation, omit ROC or AUC evaluations with a confidence level of ≈ 0.95. This metric is a common means by which comparisons to baseline approaches can be performed. Our exploratory data analysis of this RQ revealed that the automation impact is correlated to the SE task deduced from a mutual information of $0.71B$. This means that there is a subtle association between the SE task and the claimed automation impact of the approach.

7.2.2 Opportunities for Future Research. Throughout our analysis regarding the evaluation of DL4SE studies, it became apparent that there is a troubling lack of consistency of analysis, even within a given application to an SE task. Thus, there is an opportunity to develop guidelines and supporting evaluation infrastructure for metrics and approach comparisons. Such work would allow for clearer and more concise evaluations of new approaches, solidifying claims made from the results of a given evaluation. DL models are evaluated on their ability to be generalizable, this is normally accomplished through the use of a testing set, which the model has not been trained on. However, these testing sets can suffer from under representing certain class of data that can be found in the real world. More work is needed on evaluating the quality of testing sets and determining how representative they are when being used for evaluation. Having the limitations of DL approaches well document will create a greater opportunity for these DL solutions to be applied in the context of industry and real software development scenarios. Lastly, it would be advantageous for the research community to develop a methodology that could demonstrate the *need* for the complexity that DL offers when addressing a particular problem.

Summary of Results for RQ$_{4b}$:

Our analysis illustrates that a variety of metrics have been used to evaluate DL4SE techniques, with *accuracy* (≈ 46%), *precision* (≈ 35%), *recall* (≈ 33%), and *F1-measure* (≈ 26%) being the most prominent. In terms of claimed impact of our primary studies, the most claimed was *increased automation or efficiency*, followed by *advancing a DL architecture*, and *replacing human expertise.* We also found that most studies did consider the concept of Occam's Razor and offered a comparison to a conceptually simpler learning model.

8 RQ$_5$: WHAT COMMON FACTORS CONTRIBUTE TO THE DIFFICULTY WHEN REPRODUCING OR REPLICATING DL4SE STUDIES?

DL models carry with them significant complexity, thus even small, seemingly nuanced changes can lead to drastic affects in the approach's performance. Such changes could encompass the model,

the extraction and preprocessing of the data, the learning algorithm, the training process, or the hyperparameters. In this RQ, we synthesize important details related to reporting a *complete* set of elements of computation learning in DL4SE. We examined through the lens of *replicability*, or the ability to reproduce a given described approach using the same experimental setup or author-provided artifacts, and *reproducibility*, or the ability to reproduce a given approach in a different experimental setup by independently developing the approach as described in the paper [131].

In terms of replicability, we found that ≈ 14% of the primary studies provided enough information through an online repository to reasonably replicate the approach (derived via a thorough open coding with four authors). This means that the vast majority of DL4SE studies either did not provide the implementation of the approach or did not provide the dataset to train and test the approach. In terms of reproducibility, we found that ≈ 49% of the studies we analyzed provided enough detail in the publication regarding the various elements of computational learning such that a given technique could be reasonably expected to be reproduced from the text itself (according to a thorough open coding procedure between four authors). Out of the 63 studies that can be reproduced, only 11 of those studies were also replicable. We found that there were ten major factors that contributed to the lack of reproducibility. The breakdown of this analysis for the primary studies are show in in Figure 18.

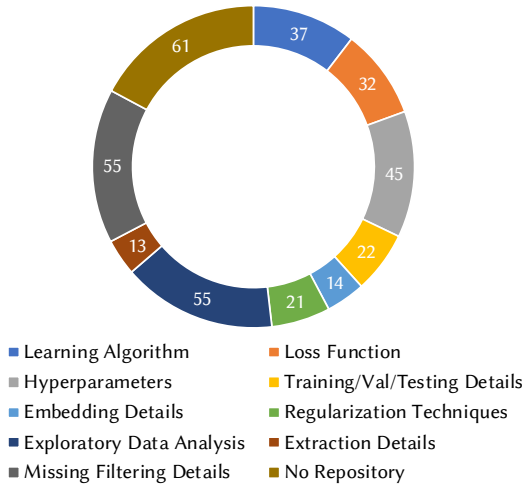

Fig. 18. Non-Reproducibility Factors

In Figure 18 we show areas where DL approaches in SE may be lacking the necessary details to reproduce or reimplement a given approach. The first two areas that contribute to the difficult of reproducibility pertain to the *learning algorithm* (37 papers) and the *hyperparameters* (45 papers). We found that there were missing details pertaining to either the method of tuning the weights, the manner in which the error was calculated or the optimization algorithm. All three aspects of the learning algorithm are important for an accurate reproduction of a described technique, and omission of these details can jeopardize the reproduction process. In addition to the learning algorithm, the hyperparameters also serve a crucial role in reproducibility. Consequently, if the hyperparameters are not reported, then it is impossible to know how many parameters contributed to the estimation of the target function, since hyperparameters control the number of layers and the number of nodes per layer. Additionally, the manner in which the learning rate is adjusted ultimately controls the parameters that estimate the target function. An incorrect learning rate can lead to incorrect parameters, which in can in turn lead to modeling of a different target function.

Additional details we often found omitted from papers pertained to *the data* and the way it was extracted, filtered, and formatted. DL models are data-driven, meaning that they extract features from the data without any human intervention. In order for the study to be reproducible, three pieces of information need to be accessible. First is the *extraction details* (13 papers) . In order for a study to be reproduced, the data must be accessible which means either the dataset must be readily available or the details about how the data was extracted need to be reported. The second piece of information that is needed is the *preprocessing details* (14 papers) . Once the data is extracted, it needs to be formatted into a representation that the DL model can accept as input. The manner in which the data is represented within the model, at least partially, controls the features that are able to be extracted. Thus, if the data is represented differently from an original study, then the

results of the reproduced study may be invalid. The last attribute of the data that is required is the *filtering details* (55 papers) . Data can be inherently noisy and authors will frequently need to filter out noise in order for the DL model to learn an effective relationship between an input and a target. This process typically involves the removal of certain data points, or an entire section of the original dataset, based upon a criteria that the authors stipulate. We discovered that 55 primary studies are missing crucial details about the filtering of their data. Reporting these details related to the filtering of data should not only include the filtering criteria, but should also explain the steps and methodology taken to remove this data from the dataset.

8.0.1 Opportunities for Future Research. Our analysis highlights the importance of open source approach implementations and datasets. This is not only the best mechanism for replicability and reproducibility, but allows for future research to more effectively build upon past techniques. We have mentioned before that DL models are extremely data-intensive. However, there are relatively few publicly available SE-related datasets tailored for DL techniques. This not only inhibits the use of DL in SE, but it also makes comparative studies with different DL models difficult. This lack of data hurts the evaluation of DL-based approaches because there are very few baselines to compare a newly synthesized model to. This can lead to claims regarding the effectiveness of a DL implementation that can be difficult to refute or verify. Therefore, we encourage future researchers to make the datasets and DL models publicly available. We believe that this will drive a greater quality of research and allow for verifiable comparisons of DL-based approaches in SE.

This SLR also highlights a number of factors that contribute to the difficulty in reproducing some of the DL4SE studies. Based on two of those factors, details regarding the exploratory data analysis and data filtering, there exists an opportunity to generate guidelines dedicated toward the preprocessing of data for specific DL algorithms. In particular, filtering steps can have large implications on overfitting of the model and lead to a reduced capacity to generalize to unseen data. One interesting research direction could be to analyze how impactful unique filtering steps can be within a specific SE task. This could provide an indication on the trade-off between generalizability and the model's performance.

Summary of Results for RQ_5:

Our analysis illustrates that only 19 of our primary studies could be conceivably labeled as *replicable*, whereas only 63 studies could be reasonably *reproduced* based upon the description given in the study. In addition to a lack of published open source implementations and datasets, the major contributing factors to these issues were mainly due to the *missing data filtering details* (55 papers) and a lack of description of *hyperparameters* (45 papers).

9 THREATS TO VALIDITY

Our systematic literature review was conducted according to the guidelines set forth by Kitchenham et al. [89]. However, as with any SLR our review does exhibit certain limitations primarily related to our search methodology and our data extraction process employed to build our paper taxonomy.

9.0.1 External Validity. Issues related to external validity typically concern the generalization of the conclusions drawn by a given study. A potential threat to the external validity to our systematic literature review is the search string and filtering process used to identify meaningful DL4SE

studies. It is possible that our search string missed studies that should have been included in our review. This could be due to a missed term or combination of terms that may have returned more significant results. We mitigated this threat by testing a variety of DL and SE terms such as:

(1) (“Deep Learning” OR “Neural Network”)
(2) (“Learning”) AND (“Neural” OR “Automatic” OR “Autoencoder” OR “Represent”)
(3) (“Learning”) AND (“Supervised” OR “Reinforcement” OR “Unsupervised” OR “Semi-supervised”)
(4) (“Learning” OR “Deep” OR “Neural” OR “Network”)
(5) (“Learning” OR “Deep” OR “Neural”)
(6) (“Artificial Intelligence” OR “Learning” OR “Representational” OR “Neural” OR “Network”)

We evaluated these potential search strings through an iterative process as outlined by Kitchenham et al. The utilized search string "Deep" `OR` "Learning" `OR` "Neural" returned the greatest number of DL4SE studies. This search string was also chosen to limit selection bias since it “cast the widest net” in order to bolster completeness and limit potential biases introduced by more restrictive search strings. However, the trade-off was that it required a much more substantial effort to remove studies that were not applicable to DL4SE.

We also face potential selection bias of the studies to be included into our SLR. We attempt to mitigate this threat through the use of inclusion and exclusion criteria, which is predefined before the filtering process begins, and which we have listed in Sec. 14. This criteria is also helpful in reducing the manual effort of filtering papers given our broad search string. We also perform snowballing as a means to mitigate selection bias. In this method, we collect all the references from the primary studies that passed our inclusion and exclusion criteria and determine if any of those references should be considered for the SLR.

Additionally, to further illustrate the generalizability of our paper sampling methodology, we perform a probability sampling to determine if we capture a significant proportion of DL4SE papers. We found that our expert sampling strategy captures a statistically significant number of studies, such that we are confident in our taxonomy’s representation. Therefore, we feel that the trends highlighted in this review can be generalized to the entire body of DL4SE work. We discuss more details pertaining to our statistical sampling of studies in Sec. 3.

Another potential threat to our systematic literature review consists of the venues chosen for consideration. For our review, we included the top SE, PL, and AI related conferences and journals. We included venues with at least a C CORE ranking [1], which helped us to determine venue impact. Although it is possible that not considering other conferences and journals caused us to miss some pertinent studies, we wanted to capture trends as seen in top SE, PL, and AI venues. Furthermore, we provide our current taxonomy and list of venues on our website, and welcome contributions from the broader research community. We intend for our online appendix to serve as a "living" document that continues to survey and categorize DL4SE research.

9.0.2 Internal Validity. A major contribution of this paper lies in our derived taxonomy that characterizes the field of DL4SE. To mitigate any mistakes in our taxonomy, we followed a process inspired by open coding in constructivist grounded theory [24] where each attribute classification of a primary study within our SLR was reviewed by at least three authors. However, while multiple evaluators limit the potential bias of this process, the classifications are still potentially affected by the collective views and opinions of the authors. Therefore, in effort to provide transparency into our process and bolster the integrity of our taxonomy, we have released all data extraction and classifications in our online repository [167, 168]. In releasing this information, authors of the works included in the SLR can review our classifications.

9.0.3 Construct Validity. One point of construct validity is the conclusions we draw at the end of each research question. In order to draw these conclusions, we performed an exploratory data analysis using rule association mining. In this analysis, we mine associations between attributes of DL solutions to SE tasks, which provides inspiration to further research why certain attributes show a strong or weak correlation. Additional details surrounding our correlation discovery and association rule learning processes can be found in Sec. 3.

Another threat to construct validity is our methodology for data synthesis and taxonomy derivation. To mitigate this threat we followed a systematic and reproducible process for analyzing the primary studies and creating a resulting taxonomy. To reduce the potential bias of data extraction, the authors developed and agreed upon a data extraction form to apply to each study. For our taxonomy, primary studies were categorized by three authors and refined by one additional authors. Through this process, we limit the number of individual mistakes in extracting the data and synthesizing the taxonomy.

10 GUIDELINES FOR FUTURE WORK ON DL4SE

In this section, we provide guidelines for conducting future work on DL4SE based upon the findings of our SLR. As illustrated in Figure 19, we synthesized a checklist with five prescribed steps that should aid in guiding researchers through the process of applying DL in SE. We do not intend for this to serve as a *comprehensive* guide, as each implementation of these complex models come with their own nuances. However, we attempt to cover many of the *essential* aspects of applying DL to SE in order to help guide future researchers.

Step 1: Preprocessing and Exploring Software Data: This step focuses on building a proper data pipeline. As part of this step, one must determine whether deep learning techniques are a viable solution for representing the data associated with a given software engineering task. This involves determining if enough data is available for learning, performing data preprocessing such as filtering, implementing an exploratory data analysis (EDA) study to better understanding the data, and performing standard data splitting to avoid data snooping.

What to report - Steps within the data formatting process should be explicitly explained to the point where replication is possible; this would include any abstractions, filters, vocabulary limitations or normalizations that the data is subjected to. Lastly, researchers should consider any potential limitations or biases uncovered as part of an exploratory data analysis together with the steps taken to mitigate these factors.

Step 2: Perform Feature Engineering: While one of the advantages of Deep Learning models is that they provide some degree of automated feature engineering, it is still important for researchers to decide upon *what information* should be used to provide a meaningful signal to their model. As such, the process of feature engineering involves identifying important features from data and accounting for different modalities (*e.g.*, textual, structured, or visual) of such data. Additionally, this step will require defining the target function that the model will attempt to learn. This requires knowing whether the model will be trained in a supervised manner, *i.e.*, with labels, an unsupervised manner, *i.e.*, without labels, or through reinforcement learning. Lastly, the data processing to convert the preprocessed data into a format that can be fed directly into a model should be performed. Similarly to the last step, all steps should be thoroughly documented in order to support replication.

What to report - For this step researchers should report their definition of their target function, the explicit features they used (e.g., code, natural language, snippets of execution traces, etc.) as well as reasoning that explains why they believe that the features and learning function chosen map well to their given problem domain. In addition, the researchers should report how they vectorize the resulting features, which includes the format of the vector or matrix subjected to the DL model.

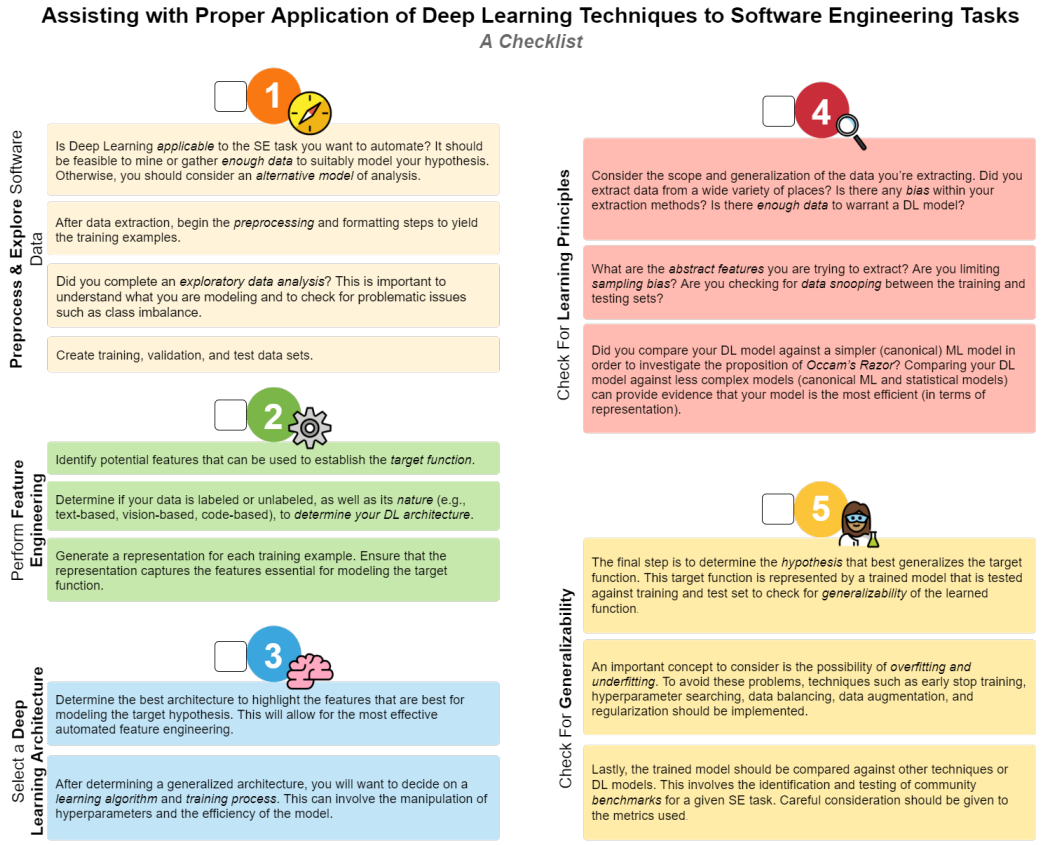

Fig. 19. Guidelines for Applying DL to SE Research

Step 3: Select a Deep Learning Architecture: This step comprises the determination of the learning algorithm, DL architecture, hyper-parameterization, optimization methods, and the training process. Researchers should carefully consider various aspects of existing architectures and determine whether a given architecture could be adapted for their task or whether the creation of a new architecture should be considered. In general these decisions can be guided by the successes and failures of past work.

What to report - Particular attention to the learning algorithm is required as this can significantly affect the results of the model. In addition to the learning algorithm, a full and thorough explanation of the optimization algorithm should be included, since the optimization algorithm describes the process of adjusting the parameters of the model to enable effective learning. When researchers report their approach, they should include a exhaustive description regarding the DL architecture used in their approach, which might include aspects such as the number of nodes, number of layers, type of layers, activation function(s), etc. Without all the aforementioned information pertaining to the architecture, future researchers will be forced to make assumptions which would likely affect reproducibility. Lastly, authors should communicate the hyper-parameters employed and training process designed. The hyper-parameters directly affect the performance of the model as they control

the ability to the model to "learn". Hyper-parameters can be determined by researchers or through empirical analysis on a reserved portion of the dataset. Therefore, the process of finding the optimal set of hyper-parameters should be explicitly stated. Similarly, researchers should communicate details pertaining to the training process of the DL model, this commonly includes the number of training iterations, the time taken to train, and any techniques employed to combat overfitting.

Step 4: Check for Learning Principles: This step is for assessing whether the basic learning principles are being considered to avoid biases, data snooping, or unnecessarily complex models (*i.e.*, Occam's Razor). Specifically, it is relevant to ensure your *dataset* is diverse, representative, and sufficient. Moreover, it is relevant to ensure your *model* considers less complex approaches than deep learning architectures. When considering Occam's Razor through the comparison of simpler (canonical) ML models, it is important to report the type of models and a brief overview of how those models were turned. This allows for additional reproducibility within the evaluation of the approach, it also substantiates the claim that the implemented DL model is appropriate for the problem being addressed.

What to report - For this step in the process, researchers should explicitly state which learning principles they considered in their analysis. This might include techniques to check for bias in a dataset, comparisons against simpler techniques, or techniques for examining whether data snooping was occurring in practice.

Step 5: Check for Generalizability: The final step involves properly evaluating a DL model. This requires carefully choosing metrics for the task, testing the model against a held out test set, ensure the model has reached optimum capacity, and using standards (or creating standards) through the usage of benchmarks. This step demonstrates a meaningful attempt to accurately represent how the model would perform in a "real world" scenario. Therefore, researchers should provide evidence of generalizability that the model has achieved and describe which benchmarks were used to support those claims.

What to report - Here researchers should strive both include the details of their evaluation plan, as well as provide rationale for the choices made. For example, this may include detailed descriptions of various metrics and justifications as to why those metrics are an appropriate measure of model efficacy for a given SE task.

In addition to the above steps steps, our findings corroborate many of the issues discussed in the study by Humbatova et al. [79] This study analyzed real faults discovered in DL systems from GitHub. They found three major areas of error when implementing DL systems: errors in the input, the model, and the training process. They also interviewed developers to determine the severity and effort required to address many of these pitfalls. They found that developers thought the most severe errors related to (i) the proper implementation of the optimizer, (ii) deriving correct input data, and (iii) correctly implementing the models with the proper number of layers. However, they also learned that developers typically found that such errors require a relatively low amount of effort to fix [79].

Our research demonstrates the components of learning within DL4SE work that are often not discussed or properly accounted for. For example, the selection and appropriate values of hyper-parameters can be an impactful when using a DL model. However, developers rated this issue to be the third highest in the amount of effort needed to address it. Similarly, properly preparing the training data and training process correct are ranked number two and number one, respectively, for the most amount of effort required to address the issue.

All details mentioned in the above guidelines should be accessible in any DL4SE publication in order to facilitate reproduction and replication of the experiments done. We also encourage authors to promote transparency of their approach by making all datasets, models, and implementation

scripts available via an online repository. This should lead to increased quality of future DL research in SE and allow for more meaningful comparisons of different DL approaches addressing SE tasks.

11 FRINGE RESEARCH NOT INCLUDED IN THIS REVIEW

In the midst of performing this SLR, we encountered studies which passed our initial inclusion criteria, but were eventually excluded based on their lack of a DL implementation. This SLR maintains the definition that deep learning models must automatically extract complex, hierarchical features from the data it is given. This implies that the data must be subjected to multiple, nonlinear transformations by passing it through multiple hidden layers within a neural network. This type of model would exclude certain algorithms that represent more canonical machine learning techniques. This hierarchy and definition of deep learning is shared by Goodfellow *et al.* [61] in a widely recognized textbook.

There were two common types of papers we encountered when performing our SLR that we felt deserved a brief introduction and explanation as to why they were not included. The first is primary studies which use Word2Vec or some variation of Word2Vec in order to embed some type of sequential data. We frequently observed Word2Vec used as a pre-processing or embedding approach in order to draw relationships between textual software artifacts using some form of similarity measure. However, we contend that such techniques do not fall within the scope of this SLR, as Word2Vec does not constitute a sufficiently "deep" architecture due to it only having a single embedding layer making it unable to model more complex representations that are normally associated with "deep" architecture, and are often used in conjunction with classical machine learning algorithms. Thus, including such works in our review would have significantly diluted the body of work that applies true DL techniques to SE problems.

We also identified a new field of research that has gained popularity known as SE4DL where the concepts learned from software testing and maintenance are being applied to the development of software based on DL algorithms. The process of our SLR captured some of these primary studies that have worked in applying SE ideals to the models generated by DL algorithms. These works focus on problems such as concolic testing for deep neural networks, addressing the lack of interoperability of DL systems by developing multi-granularity testing criteria, generating unique testing data through the use of generative adversarial networks, detecting bugs in the implementations of these DL models, seeding mutation of input data to create new testing data, detecting bugs when using certain ML frameworks such as TensorFlow, and detecting erroneous behaviors in autonomous vehicles powered by deep neural networks [50, 51, 66, 87, 113, 115, 118, 150, 156, 183, 187, 188, 194]. These pieces of work are only the beginning of a growing field that will attempt to understand and interpret the inner-workings of DL models.

12 RELATED STUDIES AND LITERATURE REVIEWS

In this literature review, we systematically collect, analyze, and report on DL approaches applied to a variety of SE tasks. While the focus of our work is specifically on Deep Learning, given its increasing popularity, there also exists a body of work which looks to analyze related concepts of the applications of ML more generally in SE. These studies vary in terms of scope and level of detail in their analysis of the implementations of the components of (deep) learning. Previous to our approach, a number of systematic literature reviews analyzed the applications of Machine Learning and Artificial Intelligence to the field of SE [2, 125, 169]. These works focus on implementations of ML models that do not contain the complexity or hierarchical feature extraction that DL possesses. In contrast, our work solely focuses on the technology of DL applications involving the extraction of complex, hierarchical features from the training data.

More recent works have analyzed the use of DL strategies within the scope of a specific SE task. For example, there have been a variety of SLRs performed for the use of DL applied to defect prediction or anomaly detection [7, 23, 91, 177]. These studies take a detailed look at the application of DL only within the scope of defect prediction or anomaly detection. However, our SLR examines the application of DL to a multitude of SE tasks and analyzes the variety of issues and correlations found generally when applying DL to any field of SE.

The most highly related works to the work presented in this study also looks at the application of DL to a variety of SE tasks. The two most closely related papers in this space are non-peer reviewed literature reviews hosted on arXiv by Li *et al.* [100] and Ferreira *et al.* [54], which we briefly discuss here for completeness. Li *et al.*'s study analyzes 98 DL studies for general trends and applications in DL. The primary findings of this work discusses the SE tasks addressed and the most common architectures employed. The main discussion points of this survey revolve around issues that are inherent to the implementation of DL methods. This includes problems with efficiency, understandability and testability of DL approaches. This SLR also mentions the difference between the application of DL in research and industry, which could be a result of not having a suitable way to apply DL toward SE tasks in practice.

Similarly, Ferreira *et al.* [54] provide similar analysis to the paper presented by Li et al. However, Ferreira *et al.* provides a brief description of the works they studied as well as highlights some of their strengths in regards to the specific tasks they addressed. They also perform a general survey of the type of DL architectures implemented. This study's discussions involve highlighting the strengths and weakness of DL applications such as the lack of need for feature engineering, the ability to represent unstructured data, high computational costs, and large numbers of parameters.

Our work differs significantly as we implement a much more detailed and methodological analysis than both of these SLRs. Our analysis is rooted in an exploration of the *components of learning*, offering a much more comprehensive view of the entirety of applied DL approaches. Additionally, we carry out an exploratory data analysis to examine trends in the attributes we extracted from our primary studies. In addition to discussing common architectures and SE tasks, our SLR goes further by discussing additional factors such as data types, preprocessing techniques, exploratory data analysis methods, learning algorithms, loss functions, hyperparamter tuning methods, techniques to prevent over/under - fitting, baseline techniques used in evaluations, benchmarks used in evaluations, consideration of the principle of Occam's Razor, metrics used in evaluations and reproducibility concerns, all of which are omitted from the previous two studies. In addition to a thorough discussion of these aspects, we also provide a correlation analysis between these different attributes of DL approaches applied to SE. We outline many meaningful relationships that can aid future researchers in their development of new DL solutions. Along with our analysis, we synthesized common pitfalls and difficulties that exist when applying DL-based solutions, and provide guidelines for avoiding these. Lastly, we include actionable next steps for future researchers to continue exploring and improving on various aspects of DL techniques applied to SE.

13 CONCLUSIONS

In this paper, we present a systematic literature review on the primary studies related to DL4SE from the top software engineering research venues. Our work heavily relied on the guidelines laid out by Kitchenham *et al.* for performing systematic literature reviews in software engineering. We began by establishing a set of research questions that we wanted to answer pertaining to applications of DL models to SE tasks. We then empirically developed a search string to extract the relevant primary studies to the research questions we wanted to answer. We supplemented our searching process with snowballing and manual additions of papers that were not captured by our systematic approach but were relevant to our study. We then classified the relevant pieces

of work using a set of agreed upon inclusion and exclusion criteria. After distilling out a set of relevant papers, we extracted the necessary information from those papers to answer our research questions. Through the extraction process and the nature of our research questions, we inherently generated a taxonomy which pertains to different aspects of applying a DL-based approach to a SE task. Our hope is that this SLR provides future SE researchers with the necessary information and intuitions for applying DL in new and interesting ways within the field of SE. The concepts described in this review should aid researchers and developers in understanding where DL can be applied and necessary considerations for applying these complex models to automate SE tasks.

ACKNOWLEDGMENT

This material is supported by the NSF grants CCF-1927679, CCF-1955853, CCF-2007246, and CCF-1815186. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

REFERENCES

[1] [n.d.]. https://www.core.edu.au/home

[2] [n.d.]. Literature Reviews on Applying Artificial Intelligence/Machine Learning to Software Engineering Research Problems: Preliminary. http://ceur-ws.org/Vol-2506/Paper5-seed2019.pdf

[3] 2019. ACM Artifact Review Policies https://www.acm.org/publications/policies/artifact-review-badging.

[4] 2020. Best Data Science & Machine Learning Platform. https://rapidminer.com/

[5] Yaser S. Abu-Mostafa, Malik Magdon-Ismail, and Hsuan-Tien Lin. 2012. *Learning from data: a short course.* AML-book.com.

[6] Toufique Ahmed, V. Hellendoorn, and Premkumar T. Devanbu. 2021. Learning lenient parsing & typing via indirect supervision. *Empirical Software Engineering* 26 (2021), 1–31.

[7] Feidu Akmel, Ermiyas Birhanu, and Bahir Siraj. 2018. A Literature Review Study of Software Defect Prediction using Machine Learning Techniques. *International Journal of Emerging Research in Management and Technology* 6 (06 2018), 300. https://doi.org/10.23956/ijermt.v6i6.286

[8] Miltiadis Allamanis. 2018. The Adverse Effects of Code Duplication in Machine Learning Models of Code. *CoRR* abs/1812.06469 (2018). arXiv:1812.06469 http://arxiv.org/abs/1812.06469

[9] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. 2015. Suggesting Accurate Method and Class Names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. ACM, New York, NY, USA, 38–49. https://doi.org/10.1145/2786805.2786849

[10] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to Represent Programs with Graphs. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=BJOFETxR-

[11] Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A Convolutional Attention Network for Extreme Summarization of Source Code. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016 (JMLR Workshop and Conference Proceedings)*, Maria-Florina Balcan and Kilian Q. Weinberger (Eds.), Vol. 48. JMLR.org, 2091–2100. http://proceedings.mlr.press/v48/allamanis16.html

[12] Plamen Angelov and Eduardo Soares. 2020. Towards explainable deep neural networks (xDNN). *Neural Networks* 130 (2020), 185–194. https://doi.org/10.1016/j.neunet.2020.07.010

[13] Forough Arabshahi, Sameer Singh, and Animashree Anandkumar. 2018. Combining Symbolic Expressions and Black-box Function Evaluations in Neural Programs. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=Hksj2WWAW

[14] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2017. DeepCoder: Learning to Write Programs. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=ByldLrqlx

[15] Rohan Bavishi, Caroline Lemieux, Roy Fox, Koushik Sen, and Ion Stoica. 2019. AutoPandas: Neural-Backed Generators for Program Synthesis. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 168 (Oct. 2019), 27 pages. https://doi.org/10.1145/3360594

[16] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. 2018. Neural Code Comprehension: A Learnable Representation of Code Semantics. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, Samy Bengio,

Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett (Eds.). 3589–3601. https://proceedings.neurips.cc/paper/2018/hash/17c3433fecc21b57000debdf7ad5c930-Abstract.html

[17] Sahil Bhatia, Pushmeet Kohli, and Rishabh Singh. 2018. Neuro-symbolic Program Corrector for Introductory Programming Assignments. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. ACM, New York, NY, USA, 60–70. https://doi.org/10.1145/3180155.3180219

[18] Nghi D. Q. Bui, Lingxiao Jiang, and Yijun Yu. 2018. Cross-Language Learning for Program Classification Using Bilateral Tree-Based Convolutional Neural Networks. In *The Workshops of the The Thirty-Second AAAI Conference on Artificial Intelligence, New Orleans, Louisiana, USA, February 2-7, 2018 (AAAI Workshops)*, Vol. WS-18. AAAI Press, 758–761. https://aaai.org/ocs/index.php/WS/AAAIW18/paper/view/17338

[19] Nghi D. Q. Bui, Yijun Yu, and Lingxiao Jiang. 2019. Bilateral Dependency Neural Networks for Cross-Language Algorithm Classification. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 422–433. https://doi.org/10.1109/SANER.2019.8667995

[20] Rudy Bunel, Matthew J. Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. 2018. Leveraging Grammar and Reinforcement Learning for Neural Program Synthesis. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=H1Xw62kRZ

[21] Lutz Büch and Artur Andrzejak. 2019. Learning-Based Recursive Aggregation of Abstract Syntax Trees for Code Clone Detection. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 95–104. https://doi.org/10.1109/SANER.2019.8668039

[22] Jonathon Cai, Richard Shin, and Dawn Song. 2017. Making Neural Programming Architectures Generalize via Recursion. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=BkbY4psgg

[23] Raghavendra Chalapathy and Sanjay Chawla. 2019. Deep Learning for Anomaly Detection: A Survey. *CoRR* abs/1901.03407 (2019). arXiv:1901.03407 http://arxiv.org/abs/1901.03407

[24] K. Charmaz. 2006. *Constructing Grounded Theory*. SAGE Publications Inc.

[25] Chao Chen, Wenrui Diao, Yingpei Zeng, Shanqing Guo, and Chengyu Hu. 2018. DRLgencert: Deep Learning-Based Automated Testing of Certificate Verification in SSL/TLS Implementations. In *2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23-29, 2018*. IEEE Computer Society, 48–58. https://doi.org/10.1109/ICSME.2018.00014

[26] Chunyang Chen, Ting Su, Guozhu Meng, Zhenchang Xing, and Yang Liu. 2018. From UI Design Image to GUI Skeleton: A Neural Machine Translator to Bootstrap Mobile GUI Implementation. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. ACM, New York, NY, USA, 665–676. https://doi.org/10.1145/3180155.3180240

[27] C. Chen, Z. Xing, Y. Liu, and K. L. X. Ong. 2019. Mining Likely Analogical APIs across Third-Party Libraries via Large-Scale Unsupervised API Semantics Embedding. *IEEE Transactions on Software Engineering* (2019), 1–1. https://doi.org/10.1109/TSE.2019.2896123

[28] G. Chen, C. Chen, Z. Xing, and B. Xu. 2016. Learning a dual-language vector space for domain-specific cross-lingual question retrieval. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 744–755.

[29] Qingying Chen and Minghui Zhou. 2018. A Neural Framework for Retrieval and Summarization of Source Code. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE 2018)*. ACM, New York, NY, USA, 826–831. https://doi.org/10.1145/3238147.3240471

[30] Xinyun Chen, Chang Liu, and Dawn Song. 2018. Towards Synthesizing Complex Programs From Input-Output Examples. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=Skp1ESxRZ

[31] Xinyun Chen, Chang Liu, and Dawn Song. 2018. Tree-to-tree Neural Networks for Program Translation. In *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.). Curran Associates, Inc., 2547–2557. http://papers.nips.cc/paper/7521-tree-to-tree-neural-networks-for-program-translation.pdf

[32] Morakot Choetkiertikul, Hoa Khanh Dam, Truyen Tran, and Aditya Ghose. 2017. Predicting the delay of issues with due dates in software projects. *Empirical Software Engineering* 22, 3 (01 Jun 2017), 1223–1263. https://doi.org/10.1007/s10664-016-9496-7

[33] M. Choetkiertikul, H. K. Dam, T. Tran, A. Ghose, and J. Grundy. 2018. Predicting Delivery Capability in Iterative Software Development. *IEEE Transactions on Software Engineering* 44, 6 (June 2018), 551–573. https://doi.org/10.1109/TSE.2017.2693989

[34] M. Choetkiertikul, H. K. Dam, T. Tran, T. Pham, A. Ghose, and T. Menzies. 2019. A Deep Learning Model for Estimating Story Points. *IEEE Transactions on Software Engineering* 45, 7 (July 2019), 637–656. https://doi.org/10.1109/TSE.2018.2792473

[35] C. S. Corley, K. Damevski, and N. A. Kraft. 2015. Exploring the Use of Deep Learning for Feature Location. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME) (ICSME'15)*. 556–560. https://doi.org/10.1109/ICSM.2015.7332513 ISSN:.

[36] Chris Cummins, Pavlos Petoumenos, Alastair Murray, and Hugh Leather. 2018. Compiler Fuzzing Through Deep Learning. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands) *(ISSTA 2018)*. ACM, New York, NY, USA, 95–105. https://doi.org/10.1145/3213846.3213848

[37] Milan Cvitkovic, Badal Singh, and Animashree Anandkumar. 2019. Open Vocabulary Learning on Source Code with a Graph-Structured Cache. In *Proceedings of the 36th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.), Vol. 97. PMLR, 1475–1485. http://proceedings.mlr.press/v97/cvitkovic19b.html

[38] Hoa Khanh Dam, Trang Pham, Shien Wee Ng, Truyen Tran, John Grundy, Aditya Ghose, Taeksu Kim, and Chul-Joo Kim. 2019. Lessons Learned from Using a Deep Tree-Based Model for Software Defect Prediction in Practice. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 46–57. https://doi.org/10.1109/MSR.2019.00017

[39] H. K. Dam, T. Tran, T. T. M. Pham, S. W. Ng, J. Grundy, and A. Ghose. 2018. Automatic feature learning for predicting vulnerable software components. *IEEE Transactions on Software Engineering* (2018), 1–1. https://doi.org/10.1109/TSE.2018.2881961

[40] Yuntian Deng, Anssi Kanervisto, Jeffrey Ling, and Alexander M. Rush. 2017. Image-to-Markup Generation with Coarse-to-Fine Attention. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70* (Sydney, NSW, Australia) *(ICML'17)*. JMLR.org, 980–989.

[41] J. Deshmukh, A. K. M, S. Podder, S. Sengupta, and N. Dubash. 2017. Towards Accurate Duplicate Bug Retrieval Using Deep Learning Techniques. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 115–124. https://doi.org/10.1109/ICSME.2017.69

[42] Jacob Devlin, Rudy Bunel, Rishabh Singh, Matthew J. Hausknecht, and Pushmeet Kohli. 2017. Neural Program Meta-Induction. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.). 2080–2088. https://proceedings.neurips.cc/paper/2017/hash/3bf55bbad370a8fcad1d09b005e278c2-Abstract.html

[43] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *CoRR* abs/1810.04805 (2018). arXiv:1810.04805 http://arxiv.org/abs/1810.04805

[44] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. 2017. RobustFill: Neural Program Learning under Noisy I/O. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017 (Proceedings of Machine Learning Research)*, Doina Precup and Yee Whye Teh (Eds.), Vol. 70. PMLR, 990–998. http://proceedings.mlr.press/v70/devlin17a.html

[45] Pedro Domingos. 1998. Occam's Two Razors: The Sharp and the Blunt. In *Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining* (New York, NY) *(KDD'98)*. AAAI Press, 37–43.

[46] John Duchi, Elad Hazan, and Yoram Singer. 2011. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *J. Mach. Learn. Res.* 12 (July 2011), 2121–2159. http://dl.acm.org/citation.cfm?id=1953048.2021068

[47] Kevin Ellis, Lucas Morales, Mathias Sablé-Meyer, Armando Solar-Lezama, and Josh Tenenbaum. 2018. Learning Libraries of Subroutines for Neurally–Guided Bayesian Program Induction. In *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.). Curran Associates, Inc., 7805–7815. http://papers.nips.cc/paper/8006-learning-libraries-of-subroutines-for-neurallyguided-bayesian-program-induction.pdf

[48] Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Josh Tenenbaum. 2018. Learning to Infer Graphics Programs from Hand-Drawn Images. In *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.). Curran Associates, Inc., 6059–6068. http://papers.nips.cc/paper/7845-learning-to-infer-graphics-programs-from-hand-drawn-images.pdf

[49] Sarah Fakhoury, Venera Arnaoudova, Cedric Noiseux, Foutse Khomh, and Giuliano Antoniol. 2018. Keep it simple: Is deep learning good for linguistic smell detection?. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 602–611. https://doi.org/10.1109/SANER.2018.8330265

[50] Fabio Falcini, Giuseppe Lami, and Alessandra Costanza. 2017. Deep Learning in Automotive Software. *IEEE Software* 34 (05 2017), 56–63. https://doi.org/10.1109/MS.2017.79

[51] F. Falcini, G. Lami, and A. Mitidieri. 2017. Yet Another Challenge for the Automotive Software: Deep Learning. *IEEE Software* (2017), 1–1. https://doi.org/10.1109/MS.2017.265101102

[52] Ming Fan, Xiapu Luo, Jun Liu, Meng Wang, Chunyin Nong, Qinghua Zheng, and Ting Liu. 2019. Graph Embedding Based Familial Analysis of Android Malware using Unsupervised Learning. In *2019 IEEE/ACM 41st International*

Conference on Software Engineering (ICSE). 771–782. https://doi.org/10.1109/ICSE.2019.00085

[53] Usama Fayyad, Gregory Piatetsky-Shapiro, and Padhraic Smyth. 1996. From Data Mining to Knowledge Discovery in Databases. *AI Magazine* 17, 3 (Mar. 1996), 37. https://doi.org/10.1609/aimag.v17i3.1230

[54] Fabio Ferreira, Luciana Lourdes Silva, and Marco Tulio Valente. 2019. Software Engineering Meets Deep Learning: A Literature Review. arXiv:1909.11436 [cs.SE]

[55] Wei Fu and Tim Menzies. 2017. Easy over Hard: A Case Study on Deep Learning. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 49–60. https://doi.org/10.1145/3106237.3106256

[56] Jian Gao, Xin Yang, Ying Fu, Yu Jiang, and Jiaguang Sun. 2018. VulSeeker: A Semantic Learning Based Vulnerability Seeker for Cross-platform Binary. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE 2018)*. ACM, New York, NY, USA, 896–899. https://doi.org/10.1145/3238147.3240480

[57] Sa Gao, Chunyang Chen, Zhenchang Xing, Yukun Ma, Wen Song, and Shang-Wei Lin. 2019. A Neural Model for Method Name Generation from Functional Description. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 414–421. https://doi.org/10.1109/SANER.2019.8667994

[58] Alexander L. Gaunt, Marc Brockschmidt, Nate Kushman, and Daniel Tarlow. 2017. Differentiable Programs with Neural Libraries. In *Proceedings of the 34th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Doina Precup and Yee Whye Teh (Eds.), Vol. 70. PMLR, International Convention Centre, Sydney, Australia, 1213–1222. http://proceedings.mlr.press/v70/gaunt17a.html

[59] Leilani H. Gilpin, David Bau, Ben Z. Yuan, Ayesha Bajwa, Michael Specter, and Lalana Kagal. 2019. Explaining Explanations: An Overview of Interpretability of Machine Learning. arXiv:1806.00069 [cs.AI]

[60] Patrice Godefroid, Hila Peleg, and Rishabh Singh. 2017. Learn&Fuzz: Machine Learning for Input Fuzzing. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering* (Urbana-Champaign, IL, USA) *(ASE 2017)*. IEEE Press, Piscataway, NJ, USA, 50–59. http://dl.acm.org.proxy.wm.edu/citation.cfm?id=3155562.3155573

[61] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. The MIT Press.

[62] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep Code Search. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. ACM, New York, NY, USA, 933–944. https://doi.org/10.1145/3180155.3180167

[63] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API Learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. ACM, New York, NY, USA, 631–642. https://doi.org/10.1145/2950290.2950334

[64] Chenkai Guo, Dengrong Huang, Naipeng Dong, Quanqi Ye, Jing Xu, Yaqing Fan, Hui Yang, and Yifan Xu. 2019. Deep Review Sharing. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 61–72. https://doi.org/10.1109/SANER.2019.8668037

[65] Jin Guo, Jinghui Cheng, and Jane Cleland-Huang. 2017. Semantically Enhanced Software Traceability Using Deep Learning Techniques. In *Proceedings of the 39th International Conference on Software Engineering* (Buenos Aires, Argentina) *(ICSE '17)*. IEEE Press, Piscataway, NJ, USA, 3–14. https://doi.org/10.1109/ICSE.2017.9

[66] Jianmin Guo, Yu Jiang, Yue Zhao, Quan Chen, and Jiaguang Sun. 2018. DLFuzz: Differential Fuzzing Testing of Deep Learning Systems. *CoRR* abs/1808.09413 (2018). arXiv:1808.09413 http://arxiv.org/abs/1808.09413

[67] Rahul Gupta, Aditya Kanade, and Shirish Shevade. 2019. Deep Reinforcement Learning for Syntactic Error Repair in Student Programs. *Proceedings of the AAAI Conference on Artificial Intelligence* 33 (07 2019), 930–937. https://doi.org/10.1609/aaai.v33i01.3301930

[68] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. DeepFix: Fixing Common C Language Errors by Deep Learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence* (San Francisco, California, USA) *(AAAI'17)*. AAAI Press, 1345–1351.

[69] Huong Ha and Hongyu Zhang. 2019. DeepPerf: Performance Prediction for Configurable Software with Deep Sparse Neural Network. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 1095–1106. https://doi.org/10.1109/ICSE.2019.00113

[70] Jiawei Han, Jian Pei, and Yiwen Yin. 2000. Mining Frequent Patterns without Candidate Generation. ACM Press, 1–12.

[71] Z. Han, X. Li, Z. Xing, H. Liu, and Z. Feng. 2017. Learning to Predict Severity of Software Vulnerability Using Only Vulnerability Description. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 125–136. https://doi.org/10.1109/ICSME.2017.52

[72] Jacob Harer, Onur Ozdemir, Tomo Lazovich, Christopher Reale, Rebecca Russell, Louis Kim, and peter chin. 2018. Learning to Repair Software Vulnerabilities with Generative Adversarial Networks. In *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.).

Curran Associates, Inc., 7933–7943. http://papers.nips.cc/paper/8018-learning-to-repair-software-vulnerabilities-with-generative-adversarial-networks.pdf

[73] Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. 2018. Deep Learning Type Inference. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. ACM, New York, NY, USA, 152–162. https://doi.org/10.1145/3236024.3236051

[74] Vincent J. Hellendoorn and Premkumar Devanbu. 2017. Are Deep Neural Networks the Best Choice for Modeling Source Code?. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. ACM, New York, NY, USA, 763–773. https://doi.org/10.1145/3106237.3106290

[75] Vincent J. Hellendoorn, Premkumar T. Devanbu, and Mohammad Amin Alipour. 2018. On the Naturalness of Proofs. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. ACM, New York, NY, USA, 724–728. https://doi.org/10.1145/3236024.3264832

[76] Thong Hoang, Hoa Khanh Dam, Yasutaka Kamei, David Lo, and Naoyasu Ubayashi. 2019. DeepJIT: An End-to-End Deep Learning Framework for Just-in-Time Defect Prediction. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 34–45. https://doi.org/10.1109/MSR.2019.00016

[77] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep Code Comment Generation. In *Proceedings of the 26th Conference on Program Comprehension* (Gothenburg, Sweden) *(ICPC '18)*. Association for Computing Machinery, New York, NY, USA, 200–210. https://doi.org/10.1145/3196321.3196334

[78] Q. Huang, X. Xia, D. Lo, and G. C. Murphy. 2018. Automating Intention Mining. *IEEE Transactions on Software Engineering* (2018), 1–1. https://doi.org/10.1109/TSE.2018.2876340

[79] Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. 2019. Taxonomy of Real Faults in Deep Learning Systems. arXiv:1910.11015 [cs.SE]

[80] Xuan Huo, Ferdian Thung, Ming Li, David Lo, and Shu-Ting Shi. 2021. Deep Transfer Bug Localization. *IEEE Transactions on Software Engineering* 47, 7 (2021), 1368–1380. https://doi.org/10.1109/TSE.2019.2920771

[81] Sergey Ioffe and Christian Szegedy. 2015. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *CoRR* abs/1502.03167 (2015). arXiv:1502.03167 http://arxiv.org/abs/1502.03167

[82] Siyuan Jiang, Ameer Armaly, and Collin McMillan. 2017. Automatically Generating Commit Messages from Diffs Using Neural Machine Translation. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering* (Urbana-Champaign, IL, USA) *(ASE 2017)*. IEEE Press, Piscataway, NJ, USA, 135–146. http://dl.acm.org.proxy.wm.edu/citation.cfm?id=3155562.3155583

[83] Ashwin Kalyan, Abhishek Mohta, Oleksandr Polozov, Dhruv Batra, Prateek Jain, and Sumit Gulwani. 2018. Neural-Guided Deductive Search for Real-Time Program Synthesis from Examples. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=rywDjg-RW

[84] Rafael-Michael Karampatsis and Charles Sutton. 2019. Maybe Deep Neural Networks are the Best Choice for Modeling Source Code. *CoRR* abs/1903.05734 (2019). arXiv:1903.05734 http://arxiv.org/abs/1903.05734

[85] Rafael-Michael Karampatsis, Hlib Babii, R. Robbes, Charles Sutton, and A. Janes. 2020. Big Code != Big Vocabulary: Open-Vocabulary Models for Source Code. *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)* (2020), 1073–1085.

[86] Deborah S. Katz, Jason Ruchti, and Eric Schulte. 2018. Using recurrent neural networks for decompilation. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 346–356. https://doi.org/10.1109/SANER.2018.8330222

[87] Pieter-Jan Kindermans, Kristof T. Schütt, Maximilian Alber, Klaus-Robert Müller, Dumitru Erhan, Been Kim, and Sven Dähne. 2017. Learning how to explain neural networks: PatternNet and PatternAttribution. arXiv:1705.05598 [stat.ML]

[88] Diederik P. Kingma and Jimmy Ba. 2014. Adam: A Method for Stochastic Optimization. arXiv:1412.6980 [cs.LG]

[89] B. Kitchenham and S Charters. 2007. Guidelines for performing Systematic Literature Reviews in Software Engineering.

[90] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger (Eds.). Curran Associates, Inc., 1097–1105. http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[91] Donghwoon Kwon, H. Kim, Jinoh Kim, S. Suh, I. Kim, and K. J. Kim. 2017. A survey of deep learning-based network anomaly detection. *Cluster Computing* 22 (2017), 949–961.

[92] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. 2015. Combining Deep Learning with Information Retrieval to Localize Buggy Files for Bug Reports (N). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 476–481. https://doi.org/10.1109/ASE.2015.73

[93] Tien-Duy B. Le, Lingfeng Bao, and David Lo. 2018. DSM: A Specification Mining Tool Using Recurrent Neural Network Based Language Model. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. ACM, New York, NY, USA, 896–899. https://doi.org/10.1145/3236024.3264597

[94] Tien-Duy B. Le and David Lo. 2018. Deep Specification Mining. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands) *(ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 106–117. https://doi.org/10.1145/3213846.3213876

[95] Alexander LeClair, Siyuan Jiang, and Collin McMillan. 2019. A Neural Model for Generating Natural Language Summaries of Program Subroutines. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, 795–806. https://doi.org/10.1109/ICSE.2019.00087

[96] Sun-Ro Lee, Min-Jae Heo, Chan-Gun Lee, Milhan Kim, and Gaeul Jeong. 2017. Applying Deep Learning Based Automatic Bug Triager to Industrial Projects. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. ACM, New York, NY, USA, 926–931. https://doi.org/10.1145/3106237.3117776

[97] Dor Levy and Lior Wolf. 2017. Learning to Align the Source Code to the Compiled Object Code. In *Proceedings of the 34th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Doina Precup and Yee Whye Teh (Eds.), Vol. 70. PMLR, International Convention Centre, Sydney, Australia, 2043–2051. http://proceedings.mlr.press/v70/levy17a.html

[98] D. Li, Z. Wang, and Y. Xue. 2018. Fine-grained Android Malware Detection based on Deep Learning. In *2018 IEEE Conference on Communications and Network Security (CNS)*. 1–2. https://doi.org/10.1109/CNS.2018.8433204

[99] Liuqing Li, He Feng, Wenjie Zhuang, Na Meng, and Barbara G. Ryder. 2017. CCLearner: A Deep Learning-Based Clone Detection Approach. *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (2017), 249–260.

[100] Xiaochen Li, He Jiang, Zhilei Ren, Ge Li, and Jingxuan Zhang. 2018. Deep Learning in Software Engineering. *CoRR* abs/1805.04825 (2018). arXiv:1805.04825 http://arxiv.org/abs/1805.04825

[101] Yi Li, Shaohua Wang, Tien N. Nguyen, and Son Van Nguyen. 2019. Improving Bug Detection via Context-Based Code Representation Learning and Attention-Based Neural Networks. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 162 (Oct. 2019), 30 pages. https://doi.org/10.1145/3360588

[102] Chen Liang, Mohammad Norouzi, Jonathan Berant, Quoc V Le, and Ni Lao. 2018. Memory Augmented Policy Optimization for Program Synthesis and Semantic Parsing. In *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.). Curran Associates, Inc., 9994–10006. http://papers.nips.cc/paper/8204-memory-augmented-policy-optimization-for-program-synthesis-and-semantic-parsing.pdf

[103] B. Lin, F. Zampetti, G. Bavota, M. Di Penta, M. Lanza, and R. Oliveto. 2018. Sentiment Analysis for Software Engineering: How Far Can We Go?. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. 94–104. https://doi.org/10.1145/3180155.3180195

[104] Bingchang Liu, Wei Huo, Chao Zhang, Wenchao Li, Feng Li, Aihua Piao, and Wei Zou. 2018. αDiff: Cross-version Binary Code Similarity Detection with DNN. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE 2018)*. ACM, New York, NY, USA, 667–678. https://doi.org/10.1145/3238147.3238199

[105] Chang Liu, Xinyun Chen, Richard Shin, Mingcheng Chen, and Dawn Song. 2016. Latent Attention For If-Then Program Synthesis. In *Advances in Neural Information Processing Systems 29*, D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett (Eds.). Curran Associates, Inc., 4574–4582. http://papers.nips.cc/paper/6284-latent-attention-for-if-then-program-synthesis.pdf

[106] Hui Liu, Zhifeng Xu, and Yanzhen Zou. 2018. Deep Learning Based Feature Envy Detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE 2018)*. ACM, New York, NY, USA, 385–396. https://doi.org/10.1145/3238147.3238166

[107] K. Liu, D. Kim, T. F. Bissyande, S. Yoo, and Y. Le Traon. 2018. Mining Fix Patterns for FindBugs Violations. *IEEE Transactions on Software Engineering* (2018), 1–1. https://doi.org/10.1109/TSE.2018.2884955

[108] Kui Liu, Dongsun Kim, Tegawendé F. Bissyandé, Taeyoung Kim, Kisub Kim, Anil Koyuncu, Suntae Kim, and Yves Le Traon. 2019. Learning to Spot and Refactor Inconsistent Method Names. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 1–12. https://doi.org/10.1109/ICSE.2019.00019

[109] P. Liu, X. Zhang, M. Pistoia, Y. Zheng, M. Marques, and L. Zeng. 2017. Automatic Text Input Generation for Mobile Testing. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 643–653. https://doi.org/10.1109/ICSE.2017.65

[110] Xiao Liu, Xiaoting Li, Rupesh Prajapati, and Dinghao Wu. 2019. DeepFuzz: Automatic Generation of Syntax Valid C Programs for Fuzz Testing. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First*

Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019. AAAI Press, 1044–1051. https://doi.org/10.1609/aaai.v33i01.33011044

[111] Yibin Liu, Yanhui Li, Jianbo Guo, Yuming Zhou, and Baowen Xu. 2018. Connecting software metrics across versions to predict defects. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 232–243. https://doi.org/10.1109/SANER.2018.8330212

[112] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. arXiv:2102.04664 [cs.SE]

[113] Lei Ma, Felix Juefei-Xu, Jiyuan Sun, Chunyang Chen, Ting Su, Fuyuan Zhang, Minhui Xue, Bo Li, Li Li, Yang Liu, Jianjun Zhao, and Yadong Wang. 2018. DeepGauge: Comprehensive and Multi-Granularity Testing Criteria for Gauging the Robustness of Deep Learning Systems. *CoRR* abs/1803.07519 (2018). arXiv:1803.07519 http://arxiv.org/abs/1803.07519

[114] Lei Ma, Felix Juefei-Xu, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Chunyang Chen, Ting Su, Li Li, Yang Liu, Jianjun Zhao, and Yadong Wang. 2018. DeepGauge: Multi-granularity Testing Criteria for Deep Learning Systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE 2018)*. ACM, New York, NY, USA, 120–131. https://doi.org/10.1145/3238147.3238202

[115] Shiqing Ma, Yingqi Liu, Wen-Chuan Lee, Xiangyu Zhang, and Ananth Grama. 2018. MODE: Automated Neural Network Model Debugging via State Differential Analysis and Input Selection. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. ACM, New York, NY, USA, 175–186. https://doi.org/10.1145/3236024.3236082

[116] David J. C. MacKay. 2002. *Information Theory, Inference & Learning Algorithms*. Cambridge University Press, USA.

[117] Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. 2019. NL2Type: Inferring JavaScript Function Types from Natural Language Information. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 304–315. https://doi.org/10.1109/ICSE.2019.00045

[118] Matthew Mirman, Timon Gehr, and Martin Vechev. 2018. Differentiable Abstract Interpretation for Provably Robust Neural Networks. In *Proceedings of the 35th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Jennifer Dy and Andreas Krause (Eds.), Vol. 80. PMLR, Stockholmsmässan, Stockholm Sweden, 3578–3586. http://proceedings.mlr.press/v80/mirman18b.html

[119] Facundo Molina, Renzo Degiovanni, Pablo Ponzio, Germán Regis, Nazareno Aguirre, and Marcelo Frias. 2019. Training Binary Classifiers as Data Structure Invariants. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, 759–770. https://doi.org/10.1109/ICSE.2019.00084

[120] K. P. Moran, C. Bernal-Cárdenas, M. Curcio, R. Bonett, and D. Poshyvanyk. 2018. Machine Learning-Based Prototyping of Graphical User Interfaces for Mobile Apps. *IEEE Transactions on Software Engineering* (2018), 1–1. https://doi.org/10.1109/TSE.2018.2844788

[121] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional Neural Networks over Tree Structures for Programming Language Processing. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence* (Phoenix, Arizona) *(AAAI'16)*. AAAI Press, 1287–1293.

[122] Vijayaraghavan Murali, Swarat Chaudhuri, and Chris Jermaine. 2017. Bayesian Specification Learning for Finding API Usage Errors. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. ACM, New York, NY, USA, 151–162. https://doi.org/10.1145/3106237.3106284

[123] Vijayaraghavan Murali, Letao Qi, Swarat Chaudhuri, and Chris Jermaine. 2018. Neural Sketch Learning for Conditional Program Generation. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=HkfXMz-Ab

[124] Anh Tuan Nguyen, Trong Duc Nguyen, Hung Dang Phan, and Tien N. Nguyen. 2018. A deep neural network language model with contexts for source code. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 323–334. https://doi.org/10.1109/SANER.2018.8330220

[125] Sunday Oke. 2008. A literature review on artificial intelligence. *International Journal of Information and Management Sciences* 19 (12 2008), 535–570.

[126] J. Ott, A. Atchison, P. Harnack, A. Bergh, and E. Linstead. 2018. A Deep Learning Approach to Identifying Source Code in Images and Video. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. 376–386.

[127] Stefano Panzeri, Cesare Magri, and Ludovico Carraro. [n.d.]. Sampling bias. http://www.scholarpedia.org/article/Sampling_bias

[128] Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. 2017. Neuro-Symbolic Program Synthesis. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=rJ0JwFcex

[129] Daniel Perez and Shigeru Chiba. 2019. Cross-Language Clone Detection by Learning Over Abstract Syntax Trees. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 518–528. https://doi.org/10.1109/MSR.2019.00078

[130] Chris Piech, Jonathan Huang, Andy Nguyen, Mike Phulsuksombati, Mehran Sahami, and Leonidas Guibas. 2015. Learning Program Embeddings to Propagate Feedback on Student Code. In *Proceedings of the 32nd International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Francis Bach and David Blei (Eds.), Vol. 37. PMLR, Lille, France, 1093–1102. http://proceedings.mlr.press/v37/piech15.html

[131] Hans Plesser. 2018. Reproducibility vs. Replicability: A Brief History of a Confused Terminology. *Frontiers in Neuroinformatics* 11 (01 2018). https://doi.org/10.3389/fninf.2017.00076

[132] M. Pradel, Georgios Gousios, J. Liu, and S. Chandra. 2020. TypeWriter: neural type prediction with search-based validation. *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2020).

[133] Michael Pradel and Koushik Sen. 2018. DeepBugs: A Learning Approach to Name-Based Bug Detection. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 147 (Oct. 2018), 25 pages. https://doi.org/10.1145/3276517

[134] Devanbu Prem, Matthew Dwyer, Sebastian Elbaum, Michael Lowry, Kevin Moran, Denys Poshyvanyk, Baishakhi Ray, Rishabh Singh, and Xiangyu Zhang. 2019. Deep Learning & Software Engineering: State of Research and Future Directions. In *Proceedings of the 2019 NSF Workshop on Deep Learning and Software Engineering*.

[135] Carl Edward Rasmussen and Zoubin Ghahramani. 2001. Occam's Razor. In *Advances in Neural Information Processing Systems 13*, T. K. Leen, T. G. Dietterich, and V. Tresp (Eds.). MIT Press, 294–300. http://papers.nips.cc/paper/1925-occams-razor.pdf

[136] Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code Completion with Statistical Language Models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 419–428. https://doi.org/10.1145/2594291.2594321

[137] Scott E. Reed and Nando de Freitas. 2016. Neural Programmer-Interpreters. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1511.06279

[138] Marco Tulio Ribeiro, Tongshuang Wu, Carlos Guestrin, and Sameer Singh. 2020. Beyond Accuracy: Behavioral Testing of NLP Models with CheckList. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Online, 4902–4912. https://doi.org/10.18653/v1/2020.acl-main.442

[139] S. Romansky, N. C. Borle, S. Chowdhury, A. Hindle, and R. Greiner. 2017. Deep Green: Modelling Time-Series of Software Energy Consumption. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 273–283. https://doi.org/10.1109/ICSME.2017.79

[140] Vaibhav Saini, Farima Farmahinifarahani, Yadong Lu, Pierre Baldi, and Cristina V. Lopes. 2018. Oreo: detection of clones in the twilight zone. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu (Eds.). ACM, 354–365. https://doi.org/10.1145/3236024.3236026

[141] Neil Salkind. 2010. Encyclopedia of Research Design. *CoRR* 10.4135/9781412961288 (2010). arXiv:10.4135 https://methods.sagepub.com/reference/encyc-of-research-design/n102.xml

[142] J. Sayyad Shirabad and T.J. Menzies. 2005. The PROMISE Repository of Software Engineering Databases. School of Information Technology and Engineering, University of Ottawa, Canada. http://promise.site.uottawa.ca/SERepository

[143] Jan Schroeder, Christian Berger, Alessia Knauss, Harri Preenja, Mohammad Ali, Miroslaw Staron, and Thomas Herpel. 2017. Predicting and Evaluating Software Model Growth in the Automotive Industry. In *2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017*. IEEE Computer Society, 584–593. https://doi.org/10.1109/ICSME.2017.41

[144] Shu-Ting Shi, Ming Li, David Lo, Ferdian Thung, and Xuan Huo. 2019. Automatic Code Review by Learning the Revision of Source Code. *Proceedings of the AAAI Conference on Artificial Intelligence* 33 (07 2019), 4910–4917. https://doi.org/10.1609/aaai.v33i01.33014910

[145] Richard Shin, Illia Polosukhin, and Dawn Song. 2018. Improving Neural Program Synthesis with Inferred Execution Traces. In *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.). Curran Associates, Inc., 8917–8926. http://papers.nips.cc/paper/8107-improving-neural-program-synthesis-with-inferred-execution-traces.pdf

[146] Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. 2018. Learning Loop Invariants for Program Verification. In *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.). Curran Associates, Inc., 7751–7762. http://papers.nips.cc/paper/8001-learning-loop-invariants-for-program-verification.pdf

[147] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research* 15, 56 (2014), 1929–1958. http://jmlr.org/papers/v15/srivastava14a.html

[148] Klaas-Jan Stol, Paul Ralph, and Brian Fitzgerald. 2016. Grounded Theory in Software Engineering Research: A Critical Review and Guidelines. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. ACM, New York, NY, USA, 120–131. https://doi.org/10.1145/2884781.2884833

[149] Shao-Hua Sun, Hyeonwoo Noh, Sriram Somasundaram, and Joseph Lim. 2018. Neural Program Synthesis from Diverse Demonstration Videos. In *Proceedings of the 35th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Jennifer Dy and Andreas Krause (Eds.), Vol. 80. PMLR, Stockholmsmässan, Stockholm Sweden, 4790–4799. http://proceedings.mlr.press/v80/sun18a.html

[150] Youcheng Sun, Min Wu, Wenjie Ruan, Xiaowei Huang, Marta Kwiatkowska, and Daniel Kroening. 2018. Concolic Testing for Deep Neural Networks. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE 2018)*. ACM, New York, NY, USA, 109–119. https://doi.org/10.1145/3238147.3238172

[151] Zeyu Sun, Qihao Zhu, Lili Mou, Yingfei Xiong, Ge Li, and Lu Zhang. 2019. A Grammar-Based Structural CNN Decoder for Code Generation. *Proceedings of the AAAI Conference on Artificial Intelligence* 33 (07 2019), 7055–7062. https://doi.org/10.1609/aaai.v33i01.33017055

[152] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia. 2014. Towards a Big Data Curated Benchmark of Inter-project Code Clones. In *2014 IEEE International Conference on Software Maintenance and Evolution*. 476–480.

[153] Daniel Tarlow, Subhodeep Moitra, A. Rice, Zimin Chen, Pierre-Antoine Manzagol, Charles Sutton, and E. Aftandilian. 2020. Learning to Fix Build Errors with Graph2Diff Neural Networks. *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops* (2020).

[154] I. Tetko, D. Livingstone, and A. I. Luik. 1995. Neural network studies, 1. Comparison of overfitting and overtraining. *J. Chem. Inf. Comput. Sci.* 35 (1995), 826–833.

[155] Hannes Thaller, Lukas Linsbauer, and Alexander Egyed. 2019. Feature Maps: A Comprehensible Software Representation for Design Pattern Detection. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 207–217. https://doi.org/10.1109/SANER.2019.8667978

[156] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. DeepTest: Automated Testing of Deep-neural-network-driven Autonomous Cars. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. ACM, New York, NY, USA, 303–314. https://doi.org/10.1145/3180155.3180220

[157] Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. 2019. On Learning Meaningful Code Changes via Neural Machine Translation. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, 25–36. https://doi.org/10.1109/ICSE.2019.00021

[158] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2018. Deep Learning Similarities from Different Representations of Source Code. In *Proceedings of the 15th International Conference on Mining Software Repositories* (Gothenburg, Sweden) *(MSR '18)*. ACM, New York, NY, USA, 542–553. https://doi.org/10.1145/3196398.3196431

[159] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2018. An Empirical Investigation into Learning Bug-fixing Patches in the Wild via Neural Machine Translation. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE 2018)*. ACM, New York, NY, USA, 832–837. https://doi.org/10.1145/3238147.3240732

[160] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2018. An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation. *CoRR* abs/1812.08693 (2018). arXiv:1812.08693 http://arxiv.org/abs/1812.08693

[161] Anne-Wil Harzing Sat 6 Feb 2016 16:10 (updated Thu 5 Sep 2019 10:36). [n.d.]. Publish or Perish. https://harzing.com/resources/publish-or-perish

[162] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S. Yu. 2018. Improving Automatic Source Code Summarization via Deep Reinforcement Learning. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE 2018)*. ACM, New York, NY, USA, 397–407. https://doi.org/10.1145/3238147.3238206

[163] Ke Wang, Rishabh Singh, and Zhendong Su. 2018. Dynamic Neural Program Embeddings for Program Repair. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=BJuWrGW0Z

[164] Song Wang, Taiyue Liu, Jaechang Nam, and Lin Tan. 2020. Deep Semantic Feature Learning for Software Defect Prediction. *IEEE Transactions on Software Engineering* 46, 12 (2020), 1267–1293. https://doi.org/10.1109/TSE.2018.2877612

[165] S. Wang, T. Liu, and L. Tan. 2016. Automatically Learning Semantic Features for Defect Prediction. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. 297–308. https://doi.org/10.1145/2884781.2884804

[166] Shaohua Wang, NhatHai Phan, Yan Wang, and Yong Zhao. 2019. Extracting API Tips from Developer Question and Answer Websites. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 321–332. https://doi.org/10.1109/MSR.2019.00058

[167] Cody Watson, Nathan Cooper, David Nader, Kevin Moran, and Denys Poshyvanyk. 2021. *Data Analysis for the Systematic Literature Review of DL4SE*. https://doi.org/10.5281/zenodo.4768587

[168] Cody Watson, David Palacio, Nathan Cooper, Kevin Moran, and Denys Poshyvanyk. [n.d.]. Data Analysis for the Systematic Literature Review of DL4SE. https://wm-semeru.github.io/dl4se/

[169] Jianfeng Wen, Shixian Li, Zhiyong Lin, Yong Hu, and Changqin Huang. 2012. Systematic literature review of machine learning based software development effort estimation models. *Information and Software Technology* 54, 1 (2012), 41 – 59. https://doi.org/10.1016/j.infsof.2011.09.002

[170] M. Wen, R. Wu, and S. C. Cheung. 2018. How Well Do Change Sequences Predict Defects? Sequence Learning from Software Changes. *IEEE Transactions on Software Engineering* (2018), 1–1. https://doi.org/10.1109/TSE.2018.2876256

[171] Martin White, Michele Tufano, Matías Martínez, Martin Monperrus, and Denys Poshyvanyk. 2019. Sorting and Transforming Program Repair Ingredients via Deep Learning Code Similarities. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 479–490. https://doi.org/10.1109/SANER.2019.8668043

[172] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk. 2016. Deep Learning Code Fragments for Code Clone Detection. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE) (ASE'16)*. 87–98. ISSN:.

[173] Martin White, Christopher Vendome, Mario Linares-Vásquez, and Denys Poshyvanyk. 2015. Toward Deep Learning Software Repositories. In *Proceedings of the 12th Working Conference on Mining Software Repositories* (Florence, Italy) *(MSR '15)*. IEEE Press, Piscataway, NJ, USA, 334–345. http://dl.acm.org/citation.cfm?id=2820518.2820559

[174] Ning Xie, Gabrielle Ras, Marcel van Gerven, and Derek Doran. 2020. Explainable Deep Learning: A Field Guide for the Uninitiated. arXiv:2004.14545 [cs.LG]

[175] Rui Xie, Long Chen, Wei Ye, Zhiyu Li, Tianxiang Hu, Dongdong Du, and Shikun Zhang. 2019. DeepLink: A Code Knowledge Graph Based Deep Learning Approach for Issue-Commit Link Recovery. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 434–444. https://doi.org/10.1109/SANER.2019.8667969

[176] B. Xu, D. Ye, Z. Xing, X. Xia, G. Chen, and S. Li. 2016. Predicting Semantically Linkable Knowledge in Developer Online Forums via Convolutional Neural Network. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE) (ASE'16)*. 51–62. ISSN:.

[177] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun. 2015. Deep Learning for Just-in-Time Defect Prediction. In *2015 IEEE International Conference on Software Quality, Reliability and Security*. 17–26.

[178] Zebin Yang, Aijun Zhang, and Agus Sudjianto. 2019. Enhancing Explainability of Neural Networks through Architecture Constraints. arXiv:1901.03838 [stat.ML]

[179] Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to Mine Aligned Code and Natural Language Pairs from Stack Overflow. In *Proceedings of the 15th International Conference on Mining Software Repositories* (Gothenburg, Sweden) *(MSR '18)*. Association for Computing Machinery, New York, NY, USA, 476–486. https://doi.org/10.1145/3196398.3196408

[180] Pengcheng Yin, Graham Neubig, Miltiadis Allamanis, Marc Brockschmidt, and Alexander L. Gaunt. 2019. Learning to Represent Edits. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net. https://openreview.net/forum?id=BJl6AjC5F7

[181] Hao Yu, Wing Lam, Long Chen, Ge Li, Tao Xie, and Qianxiang Wang. 2019. Neural Detection of Semantic Code Clones via Tree-Based Convolution. In *Proceedings of the 27th International Conference on Program Comprehension* (Montreal, Quebec, Canada) *(ICPC '19)*. IEEE Press, 70–80. https://doi.org/10.1109/ICPC.2019.00021

[182] Hao Yuan, Haiyang Yu, Shurui Gui, and Shuiwang Ji. 2021. Explainability in Graph Neural Networks: A Taxonomic Survey. arXiv:2012.15445 [cs.LG]

[183] Tom Zahavy, Nir Ben-Zrihem, and Shie Mannor. 2016. Graying the black box: Understanding DQNs. *CoRR* abs/1602.02658 (2016). arXiv:1602.02658 http://arxiv.org/abs/1602.02658

[184] Matthew D. Zeiler. 2012. ADADELTA: An Adaptive Learning Rate Method. arXiv:1212.5701 [cs.LG]

[185] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A Novel Neural Source Code Representation Based on Abstract Syntax Tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 783–794. https://doi.org/10.1109/ICSE.2019.00086

[186] Lisa Zhang, Gregory Rosenblatt, Ethan Fetaya, Renjie Liao, William Byrd, Matthew Might, Raquel Urtasun, and Richard Zemel. 2018. Neural Guided Constraint Logic Programming for Program Synthesis. In *Advances in Neural Information*

Processing Systems 31, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.). Curran Associates, Inc., 1737–1746. http://papers.nips.cc/paper/7445-neural-guided-constraint-logic-programming-for-program-synthesis.pdf

[187] Mengshi Zhang, Yuqun Zhang, Lingming Zhang, Cong Liu, and Sarfraz Khurshid. 2018. DeepRoad: GAN-based Metamorphic Testing and Input Validation Framework for Autonomous Driving Systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE 2018)*. ACM, New York, NY, USA, 132–142. https://doi.org/10.1145/3238147.3238187

[188] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. 2018. An Empirical Study on TensorFlow Program Bugs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands) *(ISSTA 2018)*. ACM, New York, NY, USA, 129–140. https://doi.org/10.1145/3213846.3213866

[189] Zhuo Zhang, Yan Lei, Xiaoguang Mao, and Panpan Li. 2019. CNN-FL: An Effective Approach for Localizing Faults using Convolutional Neural Networks. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 445–455. https://doi.org/10.1109/SANER.2019.8668002

[190] Dehai Zhao, Zhenchang Xing, Chunyang Chen, Xin Xia, and Guoqiang Li. 2019. ActionNet: Vision-Based Workflow Action Recognition From Programming Screencasts. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 350–361. https://doi.org/10.1109/ICSE.2019.00049

[191] Gang Zhao and Jeff Huang. 2018. DeepSim: Deep Learning Code Functional Similarity. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 141–151. https://doi.org/10.1145/3236024.3236068

[192] Hui Zhao, Zhihui Li, Hansheng Wei, Jianqi Shi, and Yanhong Huang. 2019. SeqFuzzer: An Industrial Protocol Fuzzing Framework from a Deep Learning Perspective. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. 59–67. https://doi.org/10.1109/ICST.2019.00016

[193] Jinman Zhao, Aws Albarghouthi, Vaibhav Rastogi, Somesh Jha, and Damien Octeau. 2018. Neural-augmented static analysis of Android communication. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu (Eds.). ACM, 342–353. https://doi.org/10.1145/3236024.3236066

[194] Luisa M. Zintgraf, Taco S. Cohen, Tameem Adel, and Max Welling. 2017. Visualizing Deep Neural Network Decisions: Prediction Difference Analysis. *CoRR* abs/1702.04595 (2017). arXiv:1702.04595 http://arxiv.org/abs/1702.04595

[195] Amit Zohar and Lior Wolf. 2018. Automatic Program Synthesis of Long Programs with a Learned Garbage Collector. In *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.). Curran Associates, Inc., 2094–2103. http://papers.nips.cc/paper/7479-automatic-program-synthesis-of-long-programs-with-a-learned-garbage-collector.pdf

14 APPENDIX - SUPPLEMENTAL INFORMATION

Table 5. Software Terms used as Additional Search Parameters for DL/ML Venues

Agile software	Apps
Autonomic systems	Cloud computing
Component-based software	Configuration management
Crowd sourced software	Cyber physical systems
Debugging	Fault localization
Repair	Distributed software
Embedded software	Empirical software
End-user software	Formal methods
Human software	Green software
Human-computer interaction	Middleware
Frameworks	APIs
Mining software	Mobile applications
Model-driven	Parallel systems
Distributed systems	Concurrent systems
Performance	Program analysis
Program comprehension	Program Synthesis
Programming languages	Recommendation systems
Refactoring	Requirements engineering
Reverse engineering	Search-based software
Secure software	Privacy software
Software architecture	Software economics
Software metrics	Software evolution
Software maintenance	Software modeling
Software design	Software performance
Software process	Software product lines
Software reuse	Software services
Software testing	Software visualization
Modeling languages	Software tools
Traceability	Ubiquitous software
Validation	Verification

Table 6. SLR Inclusion Criteria

Study is published in the year range January 1, 2009 - June 1, 2019.
Study clearly defines a SE task.
Study was published in the predefined venues.
Study must identify and address a DL based approach.
Study must contain one of the terms learning, deep, or neural in the full body of the text.
Study was manually added by authors.

Table 7. SLR Exclusion Criteria

Study was published before January 1, 2009 or after June 1, 2019.
Study does not address a SE task.
Study only applied DL in the evaluation as a baseline for comparison.
Study is outside the scope of software engineering or is published in an excluded venue.
Study does not fully implement a DL based approach.
Study does not evaluate a DL based approach.
Study is an extended abstract.
Study is solely based on a representational learning approach (word2vec, doc2vec, etc.).

Table 8. Top Search Strings Tested

(“Deep Learning” OR “Neural Network”)
(“Learning”) AND (“Neural” OR “Automatic” OR “Autoencoder” OR “Represent”)
(“Learning”) AND (“Supervised” OR “Reinforcement” OR “Unsupervised” OR “Semi-supervised”)
(“Learning” OR “Deep” OR “Neural” OR “Network”)
(“Learning” OR “Deep” OR “Neural”)
(“Artificial Intelligence” OR “Learning” OR “Representational” OR “Neural” OR “Network”)

Table 9. Search String Terms Considered

Represent	Autoencoder
Learning	Artificial
Engineering	Automated
Recurrent	Context
Training	Layers
Representational	Feature
Neural	Network
Recurrent	Convolution
Machine	Deep
Intelligence	Back-propagation
Gradient	Hyper-parameters

Table 10. Column Discriptions for SE Tasks

Code Comprehension	Research focused on the understanding of source code or program functionality.
Source Code Generation	Research focused on the creation or automatic synthesis of source code.
Source Code Retrieval & Traceability	Research focused on the location of source code within software projects.
Source Code Summarization	Research focused on the summarization of source code or projects, also includes comment generation.
Bug-Fixing Process	Research focused on the location, understanding, or patching of bugs found in source code.
Code Smells	Research focused on locating, fixing or better understanding source code smells.
Software Testing	Research focused on the synthesis, improvement or implementation of software tests. This includes both traditional software and mobile software testing.
Generating Non Code Artifacts	Research focused on generating non code artifacts found in software repositories such as commit messages, story points, issue tracking, etc.
Clone Detection	Research focused on the ability to find and classify source code clones.
Software Energy Metrics	Research focused on making software more energy efficient.
Program Synthesis	Research focused on the generation of programs or improving the synthesis process.
Image To Structured Representation	Research focused on taking images or sketches and translating them to structured source code.
Software Security	Research focused on the privacy or security of software.
Program Repair	Research focused on program repair which focus exclusively on automatic patch generation.
Software Reliability / Defect Prediction	Research focused on predicting the reliability of software or the potential for defects within software.
Feature Location	Research focused on the activity of identifying an initial location in the source code that implements functionality in a software system.
Developer Forum Analysis	Research focused on mining developer forums to better understand the software engineering life cycle or the development process.
Program Translation	Research focused on translating programs from one language to another.
Software Categorization	Research focused on classifying different software projects into discrete categories, specifically for software within an app store.
Code Location Within Media	Research focused on identifying and extracting source code snippets in videos or images.
Developer Intention Mining	Research focused on developer communication and extracting developers intentions for a software project based on these communications.
Software Resource Control	Research focused on improving or optimizing the number of resources needed for a particular software project at run time.