DOI:10.1145/3460348

Developer tools that use a neural machine learning model to make predictions about previously unseen code.

BY MICHAEL PRADEL AND SATISH CHANDRA

Neural Software Analysis

SOFTWARE IS INCREASINGLY dominating the world. The huge demand for more and better software is turning tools and techniques for software developers into an important factor toward a productive economy and strong society. Such tools aim at making developers more productive by supporting them through (partial) automation in various development tasks. For example, developer tools complete partially written code, warn about potential bugs and vulnerabilities, find code clones, or help developers search through huge code bases.

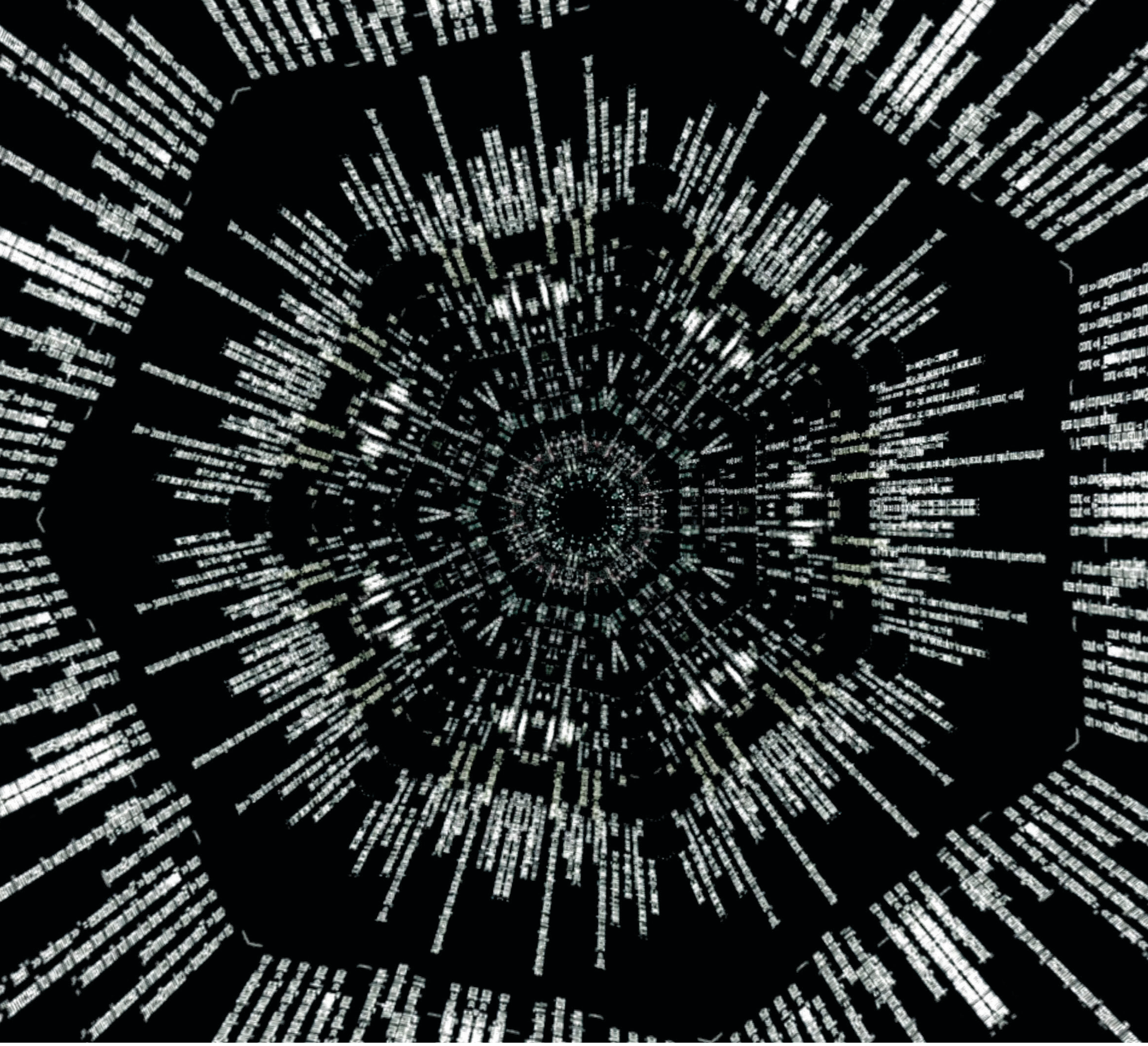
The conventional way of building developer tools is program analysis based on precise, logical reasoning. Such traditional program analysis is deployed in compilers and many other widely used tools. Despite its success, there are many problems that traditional program analysis can only partially address. The reason is that practically all interesting program analysis problems are undecidable, that is, giving answers guaranteed to be precise and correct is impossible for non-trivial programs. Instead, program analysis must approximate the behavior of the analyzed software, often with the help of carefully crafted heuristics.

Crafting effective heuristics is difficult, especially because the correct analysis result often depends on uncertain information, for example, natural language information or common coding conventions, that is not amenable to precise, logic-based reasoning. Fortunately, software is written by humans and hence follows regular patterns and coding idioms, similar to natural language.[16] For example, developers commonly call a loop variable `i` or `j`, and most developers prefer a `for-loop` over a `while-loop` when iterating through a sequential data structure. This "naturalness" of software has motivated research on machine learning-based software analysis that exploits the regularities and conventions of code.[1,31]

Over the past years, deep neural networks have emerged as a powerful technique to reason about uncertain data and to make probabilistic predictions. Can software be considered "data" for neural networks? This article answers the question with a confident "yes." We present a recent stream of research on what we call *neural software analysis*—an alternative take at program analy-

» **key insights**

- **Neural software analysis is a new way of creating tools for software developers, complementing, and for some problems, outperforming traditional program analysis.**
- **The learning-based approach naturally handles fuzzy information, such as coding conventions and natural language embedded in code, without relying on manually encoded heuristics.**
- **Neural software analyses address challenging software development problems, such as bug detection, type prediction, and code completion, and they are used in industrial practice.**



sis based on neural machine learning models that reason about software.

This article defines criteria for when to use neural software analysis based on when it is likely to complement or even outperform traditional program analysis. We then present a conceptual framework that shows how neural software analyses are typically built, and illustrate it with a series of examples, three of which we describe in more detail. The example analyses address common development problems, such as bug detection or code completion, and are already used by practitioners, despite the young age of the field. Finally, we discuss open challenges and give an outlook into promising directions for future work on neural software analysis.

When to Use Neural Software Analysis

In principle, practically all program analysis problems can be formulated in a traditional, logic reasoning-based way, as well as a data-driven, learning-based way, such as neural software analysis. The following describes conditions where neural software analysis is most suitable, and likely to outperform traditional, logic-based program analysis. See Figure 1 for a visual illustration.

Dimension 1: Fuzziness of the available information. Traditional program analysis is based on drawing definite conclusions from exact input information. However, such precise, logical reasoning often fails to represent uncertainties. Neural software analysis instead can handle fuzzy inputs given to the analysis, for example, natural language embedded in code. The fuzzier the available information is, the more likely it is that neural software analysis is suitable. The reason is neural models identify patterns while allowing for an imprecise description of these patterns. For example, instead of relying on a strict rule of the form “If the code has property A, then B holds," as traditional program analysis would use, neural software analysis learns fuzzy rules of the form "If the code is like pattern A, then B is likely to hold."

Dimension 2: Well-defined correctness criterion. Some program analysis problems have a well-defined correctness criterion, or specification, which precisely describes when an answer offered by an analysis is what the human wants, without checking with the human. For example, this is the case for test-guided program synthesis, where the provided test cases specify when an answer is correct, or for an analysis that type checks a program, where the rules of a type system define when a program is guaranteed to be type-safe. In contrast, many other analysis problems do not offer the luxury of a well-defined correctness criterion. For these problems, a human developer ultimately decides whether the answer by the analysis fits the developer's needs, typically based on whether the analysis successfully imitates what a developer would do. For example, such problems include code search based on a natural language query, code completion based on partially written code, or predicting whether a code change risk causing bugs. Neural software analysis often outperforms traditional analysis for problems that lack a well-defined correctness criterion. The reason is that addressing such "specification-free" problems is ultimately a matter of finding suitable heuristics, a task at which learned models are very effective.

Dimension 3: Number of examples to learn from. Neural models are data-hungry, and hence, neural software analysis works best if there are plenty of examples to learn from. Typically, training an effective neural model requires at least several thousands of examples. These examples can come in various forms, for example, code snippets extracted from a large code corpus. Some neural software analyses do not only extract examples from the code as-is, but also modify the code to create examples of an otherwise underrepresented class, for example, buggy code.

Figure 1. Three dimensions to determine whether to use neural software analysis.

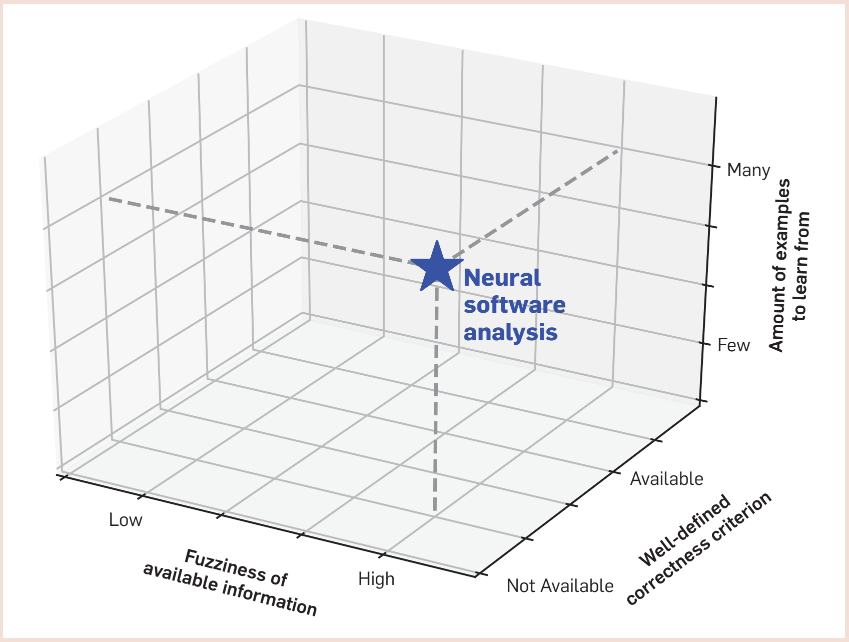
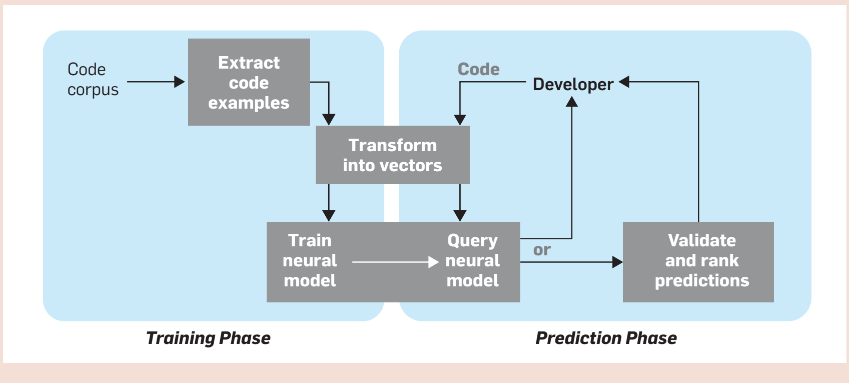


Figure 2. Typical components of a neural software analysis.



A Conceptual Framework for Neural Software Analysis

Many neural software analyses have an architecture that consists of five components (see Figure 2). Given a code corpus to learn from, the first component *extracts code examples* suitable for the problem the analysis tries to address. These code examples are transformed into vectors—either based on an intermediate representation known from compilers, such as token sequences or abstract syntax trees, or a novel code representation developed specifically for learning based analysis. Next, the examples serve as training data to *train* a neural model. The first three steps all happen during a training phase that is performed only once, before the analysis is deployed to developers. After training, the analy sis enters the prediction phase, where a developer *queries* the neural model with previously unseen code examples. The model yields predictions for the given query, which are either given directly to the developer or go through an optional *validation and ranking* component. Validation and ranking sometimes rely on a traditional program analysis, combining the strengths of both neural and logic-based reasoning. The remainder of this section discusses these components in more detail.

Extracting code examples. *Lightweight static analysis.* To extract code examples to learn from, most neural software analyses build on a lightweight static analysis. Such a static analysis reuses standard tools and libraries available for practically any programming language, for example, a tokenizer, which splits the program code into tokens, or a parser, which transforms the program code into an abstract syntax tree (AST). These tools are readily available as part of an IDE or a compiler. Building on such a lightweight, standard static analysis, instead of resorting to more sophisticated static analyses, is beneficial in two ways. First, it ensures the neural software analysis scales well to large code corpora, which are typically used for effective training. Second, it makes it easy to port a neural software analysis developed for one language to another language.

Obtaining labeled examples. Practically all existing neural software analyses use some form of supervised learning. They hence require labeled code examples, that is, code examples that come with the desired prediction result, so that the neural model can learn from it. How to obtain such labeled examples depends on the specific task an analysis is addressing. For example, an analysis that predicts types can learn from existing type annotations,[14,24,28] and an analysis that predicts code edits can learn from edit histories, for example, documented in a version control system.[10,36] Because large amounts of labeled examples are a prerequisite for effective supervised learning, what development tasks receive most attention by the neural software analysis community is partially driven by the availability of sufficiently large, annotated datasets.

Representing software as vectors. Since neural models reason about vectors of numbers, the most important design decision of a neural software analysis is how to turn the code examples extracted in the previous step into vectors. We discuss two aspects of this step: How to represent the basic building blocks of code, such as, individual code tokens. How to compose representations of the basic building blocks into representations of larger snippets of code, for example, statements or functions.

Representing code tokens. Any technique for representing code as vectors faces the question of how to map the basic building blocks of the programming language into vectors. Most neural software analyses address the token representation challenge in one of two ways. One approach is to abstract away all non-standard tokens in the code, for example, by abstracting variable names into `var1`, `var2`, and so on.[13,36] While this approach effectively reduces the vocabulary size, it also discards potentially useful information. The other approach maps each token into an embedding vector of a fixed size. The goal here is to represent semantically similar tokens, for example, the two identifiers `len` and `size`, with similar vectors.[38] To obtain such an embedding, some analyses train a token embedding before training the neural model that addresses the main task, and then map each token to a vector using the pre-trained embedding.[17,30] Alternatively, some analyses learn an embedding function jointly with the overall neural model, essentially making the task of handling the many different identifiers part of the overall optimization task that the machine learning model addresses.[2]

Neural models of code are almost exclusively classification models, which is motivated by the fact that most information associated with programs is discrete.

A key challenge is the fact the vocabulary of identifiers that developers can freely choose, for example, variable and function names, grows in an apparently linear fashion when new projects are added to a code corpus.[18] The reason is that developers come up with new terminology and conventions for different applications and application domains, leading to multiple millions of different identifiers in a corpus of only a few thousand projects. The simplest approach to handle the vocabulary problem is to fix the vocabulary to the, say, 10,000 most common tokens, while representing all other, out-of-vocabulary tokens with as a special "unknown" vector. More sophisticated techniques split tokens into subwords, for example, `writeFile` into "write" and "file," represent each subword individually, and then compose subword vectors into the representation of a full token. To split tokens into subwords, neural software analyses can rely on conventions[3] or compression algorithms that compute a fixed-size set of those subwords that occur most frequently.[18]

Representing snippets of code. How to turn snippets of source code, for example, the code in a statement or function, into a vector representation has received lots of attention by researchers and practitioners recently. The many proposed techniques can be roughly summarized into two groups. Both of them rely on some way of mapping the most elementary building blocks of code into vectors, as described earlier. On the one hand, there are techniques that turn a code snippet into one or more sequences of vectors. The simplest, yet quite popular and effective, technique[13,14,36] starts from the sequence of code tokens and maps each token into a vector. For example, a code snippet `x=true;` would be mapped into a sequence of four vectors that represent "`x`", "`=`", "`true`", and "`;`", respectively. Instead of viewing code as a flat sequence of tokens, other techniques leverage the fact that code, in contrast to, for example, natural language, has a well-defined and unambiguous structure.[5,26] Such techniques typically start from the AST of a code snippet and extract one or more paths through the tree, mapping each node in a path to a vector. For example, the popular code2vec technique[5] extracts many such AST paths, each connecting two leaves in the tree.

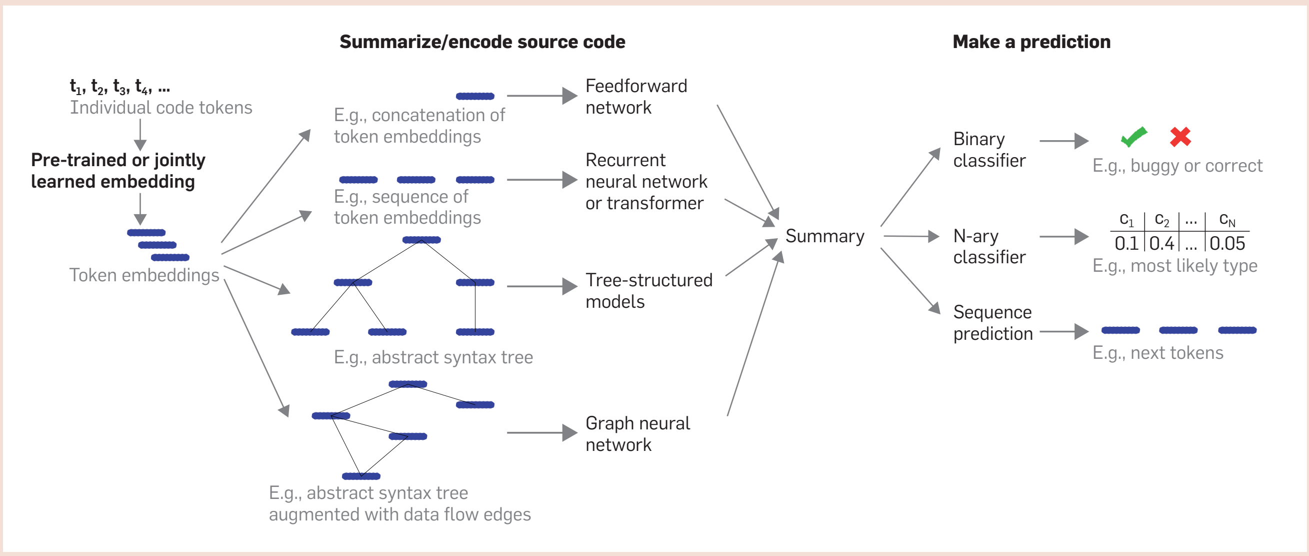
On the other hand, several techniques represent code snippets as graphs of vectors.[2,10,39] These graphs are typically based on ASTs, possibly augmented with additional edges that represent data flow, control flow, and other relations between code elements that can be computed by a traditional static analysis. Given such a graph, these techniques map each node into a vector, yielding a graph of vectors. The main advantage of graph-based vector representations of code is that they provide the rich structural and semantic information available for code to the neural model. On the downside, rich graph representations of code are less portable across programming languages, and graph-based neural models tend to be computationally more expensive than sequence-based models.

Neural models of software. Once the source code is represented as vectors, the next step is to feed these vectors into a machine learning model. Neural models of software typically consist of two parts. One part summarizes (or encodes) the given source code into a more compact representation, for example, a single vector. Given this summary, the other part then makes a prediction about the code. Figure 3 shows popular neural components used in these two parts, which we discuss in more detail next. Each neural component essentially corresponds to a learned function that maps some input to some output.

Summarizing the code source. The simplest way of summarizing code is to concatenate the vectors that describe it and map the concatenation into a shorter vector using a feedforward network. For code represented as a sequence of vectors, for example, each representing a token, recurrent neural networks or transformers[37] can summarize them. While the former traverses the sequence once, the latter iteratively pays attention to specific elements of the sequence, where the decision what to pay attention to is learned. Another common neural component are tree-structured models that summarize a tree representation of code, for example, the AST of a code snippet.[5,26] Finally, the perhaps most sophisticated of today's techniques are graph neural networks (GNNs),[2] which operate on a graph representation of the code. Given a graph of initial vector representations of each node, a GNN repeatedly updates nodes based on the current representations of the neighboring nodes, effectively propagating information across code elements.

Making a prediction. Neural models of code are almost exclusively classification models, which is motivated by the fact that most information associated with programs is discrete. One common prediction component is a binary classifier, for example, to predict whether a piece of code is correct or buggy.[23,30] In a similar vein, an *N*-ary classifier predicts which class(es) out of a fixed set of *N* classes the code belongs to, for example, predicting the

Figure 3. Neural components popular for analyzing software. In principle, any of the summarization components on the left can be freely combined with any of the prediction components on the right.



type of a variable out of a set of common types.[14,24,28] Such classifiers output a probability distribution over the available classes, which the model obtains by passing the unscaled outputs (*logits*) of earlier layers of the network through the softmax function. Beyond flat classification, some neural code analyses predict sequences of vectors, for example, how to edit a given piece of code,[36] or a natural language description of the code.[3] A common neural model for such tasks is an encoder-decoder model, which combines a sequence encoder with a decoder that predicts a sequence. In these models, the decoder's output at each step is a probability distribution, for example, across possible source-level tokens, similar to a flat classifier. For next-token prediction,[19] which is a classic language model learning task, a decoder-only model suffices, because the sequence encoding is implicit in the state of the decoder.

Training and querying. Training and querying neural models of software works just like in any other domain: The parameters—called weights and biases—of the various functions that control a model's behavior are optimized to fit examples of inputs and expected outputs. For example, for an *N*-ary classification model, training will compare the predicted probability distribution to the desired distribution via a loss function, for example, cross entropy, and then try to minimize the loss through stochastic gradient descent. Once successfully trained, the model makes predictions about previously unseen software by generalizing from the examples seen during training.

For ranking, one option is to use the numeric vectors predicted by the model to identify the predictions the model is most confident about. In a classification model that uses the softmax function, many approaches interpret the predicted likelihood of classes as a probability distribution, and then rank the classes by their predicted probability.[14,28] In an encoder-decoder model, beam search is commonly used to obtain the k most likely predicted outputs by keeping the k overall most likely predictions while decoding a complex output, such as a sequence of code tokens.[36]

Example of Neural Software Analysis Tools

Given the conceptual framework for neural software analysis, we now discuss concrete example analyses. As a broader overview, Table 1 shows selected analysis problems that are suitable for neural software analysis, along with some representative tools addressing these problems. Allamanis et al.[1] provide an extensive survey of many more analyses. The tools address a diverse set of problems, ranging from classification tasks, such as bug detection or type prediction, over generation tasks, such as code completion and code captioning, to retrieval tasks, such as code search.

The remainder of this section discusses three concrete examples of neural software analysis in some more detail. To illustrate the example analyses, we use a small Python code example (see Figure 4). The example is part of an implementation of a tic-tac-toe game and centers around a class Board that represents the 3x3 grid the game is played on. The main loop of the game (lines 19 to 28) implements the turns players take. In each turn, a player marks a point on the grid, until one of the players marks three points in a row or until the grid is filled.

Table 1. Examples of neural software analyses.

Analysis problem	Neural software analyses
Bug detection	DeepBugs[30]
	VarMisuse using GGNN[2]
	VulDeePecker[2]3
	GREAT[15]
Program repair	DeepFix[13]
	SequenceR[7]
	Hoppity[10]
	Graph2Diff[35]
Code captioning	Code summarization[3]
	Code2Seq[4]
Type prediction	DeepTyper[14]
	NL2Type[24]
	LambdaNet[39]
	TypeWriter[28]
Code synthesis	RobustFill[9]
	Autopandas[6]
Code completion	Li et al.[21]
	IntelliCompose[34]
	Karampatsis et al.[18]
	TravTrans[19]
Clone detection	White et al.[40]
Reverse engineering	DIRE[20]
	David et al.[8]
Code search	Neural code search[33]
	Deep code search[11]
Code-comment matching	Panthaplackel et al.[27]

Figure 4. Python implementation of a tic-tac-toe game.

```
1   class Board:
2      TypeWriter infers the function signature
       (Board, int, int, str) -> Bool
4     def mark_point(self, x, y, player_name):
5       """
6         Marks the given point on the board
7         as chosen by the given player.
8         Returns whether the move gives the player
9         three marked fields in a row.
10      """
11      self.field[x][y] = player_name
12      has_three_in_a_row = False
13      ... # compute whether the player has won
14      return has_three_in_a_row
15
16    def show_winner(self, player_name):
17      ...
18
19  while not game_done:
20     active_player = ...
21     x = ...
22     y = ...
23      DeepBugs warns about a bug here:
24      has_won = board.mark_point(y, x, active_player)
25      if has_won:
26        # notify player
27        game_done = True
28        board.??? Neural model suggests completions here
```

Table 2. Three neural software analyses and how they map onto the conceptual framework in Figure 2.

Component of conceptual framework	Neural software analyses: Bug detection (DeepBugs)	Type prediction (TypeWriter)	Code completion
Code corpus	JavaScript (68M lines of open-source code)	Python (2.7M lines of open-source code and a larger commercial corpus)	Python (16M lines of open source code)
Extraction of code examples	Code snippets as-is and with artificially introduced bugs	Functions with their parameter and return types	Code token sequences, offset by one for next token prediction
Transformation into vectors	Concatenation of token embeddings and context information	Token embeddings for code, word em- beddings for comments	End-to-end learned token embeddings for code
Neural model	Simple feedforward model	Hierarchical model built from several recurrent neural networks	Encoder (bi-directional LSTM) and decoder (LSTM)
Validation and ranking	Rank warnings by predicted probability that code is buggy	Search and validate correct types with type checker	Rank output tokens by probability that it is the next token

We use this example to illustrate three neural analyses summarized in Table 2. The analyses are useful for finding a bug in the example explored next, followed by predicting the type of a function and completing the example's code, respectively.

Learning to find bugs. DeepBugs[30] is a neural software analysis that tackles a continuously important problem in software development—the problem of finding bugs. While there is a tremendous amount of work on traditional, logic-based analyses to find bugs, learning-based bug detection has emerged only recently. One example of a learning-based bug detector is DeepBugs, which exploits a kind of information typically ignored by program analyses: the implicit information encoded in natural language identifiers. Due to the inherent fuzziness of this information (see Dimension 1) and the fact that determining whether a piece of code is correct is often impossible without a human (Dimension 2), a learning-based approach is a good fit for this problem. DeepBugs formulates bug detection as a classification problem, that is, it predicts for a given code snippet whether the code is correct or buggy.

Extracting code examples. To gather training data, DeepBugs extracts code examples that focus on specific kinds of statements and bug patterns that may arise in these statements. One of these bug patterns is illustrated at line 24 of Figure 4, where the arguments `y` and `x` given to a function have been swapped accidentally. To find such swapped argument bugs, the analysis extracts all function calls with at least two arguments. Based on the common assumption that most code is correct, the extracted calls serve as examples of correct code. In contrast to the many correct examples one can extract this way, it is not obvious how to gather large amounts of incorrect code examples (Dimension 3). DeepBugs addresses this challenge by artificially introducing bugs into the extracted code examples. For example, creating swapped argument bugs amounts to simply swapping the arguments of calls found in the code corpus, which is likely to yield incorrect code. DeepBugs is a generic framework that supports other bug patterns beyond swapped arguments, which are elided here for brevity.

Transformation into vectors. DeepBugs represents each code example as a concatenation of several pieces of information. Most importantly, the representation includes the natural language identifiers involved in the code snippet. For the example bug in Figure 4, the analysis extracts the name of the called function, `mark_point`, and the names of the arguments, particularly, the two swapped arguments `y` and `x`. Beyond identifier names, the analysis also considers contextual information about a code snippet, for example, the ancestor nodes of the code in the AST or operators involved in an expression. These pieces of information are represented as vectors based on pre-trained embeddings. To represent identifiers, DeepBugs pre-trains a Word2vec model[25] on token sequences of source code, which enables the analysis to generalize across similar identifiers. For our running example, this generalization allows DeepBugs to understand that when giving arguments named similarly to `x` and `y` to a function named similarly to `mark_point`, one typically passes `x` as the first of the two arguments.

Neural model. The neural model that classifies a given piece of code as buggy or correct is a simple feedforward neural network. Figure 5 (top-left) illustrates the model with an example. The model concatenates the embedding vectors of all inputs given to the model and then predicts the probability p that the code is buggy. For all code examples that are taken from the code corpus without modification, that is, supposedly correct code, the model is trained to predict $p = 0.0$, whereas it is trained to predict $p = 1.0$ for the artificially injected bugs. Once trained, the DeepBugs model can predict for previously unseen code how likely it is that this code is buggy. To this end, the analysis extracts exactly the same kind of information as during training and queries the classification model with them. If the model predicts p above some configurable threshold, the analysis reports a warning to the developer.

Validation and ranking. DeepBugs does not further validate potential bugs before reporting them as warnings to developers. However, to prioritize warnings, DeepBugs ranks potentially buggy code by the predicted probability p. Developers can then inspect all potential bugs with a p above some threshold and go down the list starting from the most likely bug.

DeepBugs-inspired code analysis tools are available for various JetBrains IDEs. Plugins that analyze JavaScript[a] and Python code have already been downloaded by thousands of developers.

a https://opensourcelibs.com/lib/deepbugsplugin.

Learning to predict types. TypeWriter[28] is a neural software analysis to predict type annotations in dynamically typed languages, such as Python or JavaScript. While not required in these languages, type annotations are often sought for when projects are growing, as they help ensure correctness, facilitate maintenance, and improve the IDE support available to developers. To illustrate the problem, consider the `mark_point` function in Figure 4. Since the function does not have any type annotations, the problem is to infer the type of its arguments and its return type. Type prediction is a prime target for neural software analysis because it fits all three dimensions discussed earlier. Source code provides various hints about the type of a variable or function, many of which are fuzzy, such as the name of a variable, the documentation that comes with a function, or how a variable is used (Dimension 1). Because there sometimes is more than one correct type annotation that a developer could reasonably choose, there is no well-defined criterion to automatically check whether a human will agree with a predicted type (Dimension 2). Finally, large amounts of code have already been annotated with types, providing sufficient training data for neural models (Dimension 3).

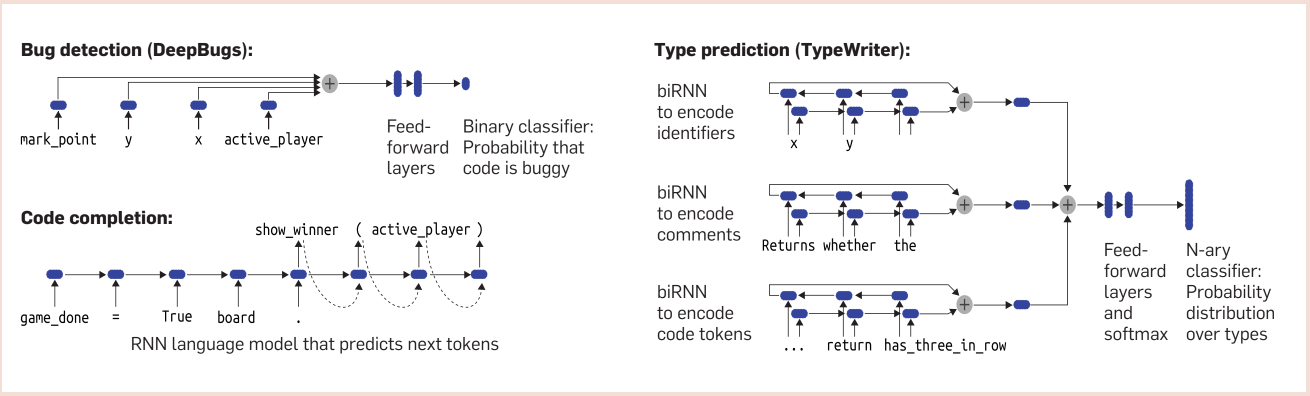
Extracting code examples. To predict the types of function arguments and return types, TypeWriter extracts two kinds of information from Python code. On the one hand, the analysis extracts natural language information associated with each possibly type-annotated program element, such as the name of a function argument or a comment associated with the function. On the other hand, the analysis extracts programming language information, such as how the code element is used. For example, consider the return type of the `mark_point` function. TypeWriter extracts the return statement at line 14, which includes a variable name (`has_...`) and a comment associated with the function ("… Returns whether…"), which both hint at the return type being Boolean. Such information is extracted for each function in each code corpus. For functions that already have type annotations, the analysis also extracts the existing annotations, which will serve as the ground truth to learn from.

Transformation into vectors. TypeWriter represents the extracted information as several sequences of vectors based on pre-trained embeddings. All code tokens and identifier names associated with a type are mapped to vectors using a code token embedding. The code-token embedding is pre-trained on all the Python code that the analysis learns from. The comments associated with a type are represented as a sequence of words, that each is represented as a vector obtained via a word embedding. The word embedding is pre-trained on all comments extracted from the given code corpus to ensure it is well suited for the vocabulary that typically appears in Python comments.

Neural model. The neural model of TypeWriter (Figure 5, right) summarizes the given vector sequences using multiple recurrent neural networks: one for all identifier names associated with the to-be-typed program element, one for all code tokens related to it, and one for all related natural language words. These three recurrent neural networks each result in a vector, and the concatenation of these vectors then serves as input to a feedforward network that acts as a classifier. The classifier outputs a probability distribution over a fixed set of types, for example, the 1,000 most common types in the corpus. TypeWriter interprets the probabilities predicted by the neural model as a ranked list of possible types for the given program element, with the type that has the highest probability at the top of the list.

Validation and ranking. The topmost predicted type may or may not be correct. For example, the neural model may predict `int` as the type of `player_name`, while it should be `string`. To be useful in practice, a type prediction tool must ensure that adding the types it suggests does not introduce any type errors. TypeWriter uses a gradual type checker to validate the types predicted by the neural model and to find a set of new type annotations that are consistent with each other and with any previously existing annotations. To this end, the approach turns the problem of assigning one of the predicted types to each of the not yet annotated program elements into a combinatorial search problem. Guided by the number of type errors that the gradual type checker reports, TypeWriter tries to add as many missing types as possible, without introducing any new type errors. The type checker-based validation combines the strengths of neural

Figure 5. Neural models used in the three example analyses.


Bug detection (DeepBugs):
mark_point
y
x
active_player
Feed-forward layers
Binary classifier: Probability that code is buggy
Code completion:
show_winner
(
active_player
)
game_done
=
True
board
.
RNN language model that predicts next tokens
Type prediction (TypeWriter):
biRNN to encode identifiers
x
y
biRNN to encode comments
Returns
whether
the
biRNN to encode code tokens
...
return
has_three_in_row
Feed-forward layers and softmax
N-ary classifier: Probability distribution over types

software analysis and traditional program analysis, by using the latter as a validation for the predictions made by the first.

TypeWriter has been developed at Facebook to add type annotations to Python code. It has already added several thousands of types to software used by billions of people.

Learning to complete partial code. As a third example, we describe a neural code completion technique. As a developer is typing code in, the technique predicts the next token at a cursor position. The neural model produces a probability distribution over potential output tokens, and typically, the top five or ten most likely tokens will be shown to the developer. The next-token prediction problem is highly suited to neural software analysis: Regarding Dimension 1, the information is fuzzy because there is no easy way to articulate rules that dictate which token should appear next.[b] Regarding Dimension 2, there is no well-defined correctness criterion, except that the program should continue to compile. Moreover, the interactivity requirements rule out an expensive validation at each point of prediction. Finally, regarding Dimension 3, there are copious amounts of training data available, as virtually all code in a given programming language is fair game as training data.

Extracting code examples. The code example extraction is trivial: For any given context—which means the code tokens up to the cursor position—the token that comes immediately after the cursor in a training corpus is the correct prediction. For the example in Figure 4, suppose a developer requests code completion at line 28 with the cursor at the location marked with ???. The analysis extracts the sequence of preceding tokens, that is, ⟨`game_done`, `=`, `True`, `board`, `.`⟩, and the expected prediction here would be `show_winner`.

Transformation into vectors. Tokens are represented through a fixed-size vocabulary, with out-of-vocabulary tokens being represented by the special "unknown" token. For example, the input above may be rewritten to ⟨42, 233, 8976, 10000, 5⟩, where the numbers are indices into the vocabulary, and board is an out-of-vocabulary token presented by index 10000. Also, all input output examples are typically padded up to be of the same length using an additional padding token.

Neural model. While predicting the next token seems like a flat classification problem, we want to condition the prediction on the preceding token sequence and not just the last token. Recurrent neural networks (RNN) are naturally suited to this task (see Figure 5, bottom-left). At each step, this network receives a summary of the previous token sequence via a hidden vector, which is depicted in the diagram by the edge coming into each node from the left. It also receives the embedding of the current token. The embedding is learned during training in an end-to-end manner, rather than separately pre-trained as in the previous analyses.

Given the hidden vector, and the embedding of the current token, the decoder produces an output token, which is expected to equal the input sequence, except shifted by one. In this way, each step of decoding is expected to produce the next token, considering the context up to that point. (Each step of decoding also produces a hidden vector to be passed to the next step.) More precisely, the decoder produces at each step a probability distribution over the vocabulary, that is, the token with maximum probability will be considered as the top-most prediction. During training, the loss is taken pointwise with respect to the ideal decoder output using negative log likelihood loss.

As an alternative to the RNN-based model, the recently proposed transformer architecture[37] can also be employed for code prediction.[19]

Validation and ranking. Code completion in an IDE often shows a ranked list of the most likely next tokens. Given the probability distribution that the model produces for each token, an IDE can show, for example, the top five most likely tokens. Local heuristics, for example, based on APIs commonly used within the project, may further tweak this ranked list before it is shown to the user. If developers are interested in predicting more than one token, beam search (noted previously) can predict multiple likely sequences.

Neural code completion has been an active topic of interest in industry. For example, it is available in TabNine, studied for internal usage at companies such as Facebook, and for widely used IDEs, such as IntelliJ from JetBrains and Visual Studio's IntelliCode from Microsoft.[34] Recent advances address three problems not considered here. First, out-of-vocabulary tokens are a crucial problem for code completion because the token to be predicted may not have been seen during training. The problem can be addressed by splitting complex identifier names into simpler constituent names[18] or by copying tokens from the context using a learned attention mechanism.[21] Second, RNNs are limited in how much of the code context they remember. Recent work[19,34] addresses this problem through transformer-based architectures, for example, using a GPT-2 transformer model that reasons about a depth-first traversal of the parse tree.[19] Third, the need to predict multiple tokens at a time, for example, to complete the entire line of code, can be addressed through beam search.[34]

Outlook and Open Challenges

Neural software analysis is a recent idea, and researchers and practitioners have just started to explore it. The following discusses open challenges and gives an outlook into how the field may evolve.

More analysis tasks. The perhaps most obvious direction for future work is to target more analysis tasks with neural approaches. In principle, every analysis task can be formulated as a learning problem. Yet, we see the biggest potential for problems that fit the three dimensions noted at the outset of this article. Some tasks that so far have received very little or no attention from the neural software analysis community include the prediction of performance properties of software, automated test input generation, and automated fault injection.

Better ways to gather data. Most current work focuses on problems for which it is relatively easy to obtain large amounts of high-quality train-

b Type information can guide which tokens cannot appear next.

ing data. Once such "low-hanging fruits" are harvested, we envision the community to shift attention to more sophisticated ways of obtaining data. One promising direction is to neurally analyze software based on runtime information. So far, almost all existing work focuses on static neural software analysis.

Better models. A core concern of every neural software analysis is how to represent software as vectors that enable a neural model to reason about the software. Learned representations of code are an active research field with promising results.[5,15,26] Driven by the observation that code provides a similar degree of "naturalness" and regularity as natural language,[16] the community is often driven by techniques that are successful in natural language processing. Yet, since software and natural language documents are clearly not the same, we envision the focus to move even more toward models specifically designed for software.

Semi-supervised and unsupervised learning. A promising direction for avoiding the need to obtain labeled training data for supervised learning is semi-supervised and unsupervised learning. The basic idea is to pre-train a model on some "pseudo-task," for which it is easy to obtain large amounts of labeled data, for example, a language model or an auto-encoder,[32] and to then use the pre-trained model on a related task for which few or even no labeled training data is available. Such few-to-zero-shot learning shows impressive results on natural language tasks and is likely to get adopted to software in the future.

Interpretability. Neural models often suffer from a lack of interpretability. This general problem affects neural software analysis, particularly because the "consumers" of these analyses typically are developers, that is, human users. Future research should investigate how to communicate to developers the reasons why a model makes a prediction, and how to give developers more confidence in following the predictions of a neural software analysis. Work on attributing predictions by a model to specific code lines is a promising first step in this direction.[12]

Integration with traditional program analysis. Neural and traditional analysis techniques have complementary strengths and weaknesses, and for many problems, combining both may be more effective than each of them individually. One way of integrating both kinds of analyses could be a continuous feedback loop, where a neural and a traditional analysis repeatedly augment the program in a way that enables the other analysis to cover more cases. Another idea is to apply traditional program slicing before passing code to a neural analysis, instead of simply passing all code as is typically done today. TypeWriter shows an early example of integrating neural and traditional analyses, where the latter validates the predictions made by the former.

Neural software analysis is an ambitious idea to address the challenges of a software-dominated world.

Scalability. Most neural software analyses proposed so far focus on small code snippets, which typically are not larger than a single function. Future work is likely to tackle the question of how to make predictions about larger pieces of code, for example, entire modules or applications. Scaling up neural software analysis will require novel ways of decomposing the reasoning performed by an analysis and of propagating predictions made for one part of a program into another part of the program.

Software-related artifacts beyond the software itself. This article and most of the neural software analysis work done so far focuses on the software itself. There are many other artifacts associated with software that could also benefit from neural analysis. Such artifacts include texts that mix natural language with code elements, for example, in the communication that happens during code review or in bug reports, and software-generated artifacts, such as crash traces or logs.[29]

Conclusion

Neural software analysis is an ambitious idea to address the challenges of a software-dominated world. The idea has already shown promising results on a variety of development tasks, including work adopted by practitioners. From a fundamental point of view, neural software analysis provides a powerful and elegant way to reason about the uncertain nature of software. This uncertainty relates both to the fact that program analysis problems are typically undecidable and to the "fuzzy" information, such as natural language and coding conventions, that is embedded into programs. From a pragmatic point of view, neural software analysis can help in significantly reducing the effort required to produce a software analysis. While traditional, logic-based analyses are usually built by program analysis experts, of which only a few hundred exist in the world, neural software analyses are learned from data. This data-driven approach enables millions of developers to, perhaps unconsciously, contribute to the success of neural software analyses by the mere fact that they produce software. That is, neural software analysis uses the root cause of the demand for developer tools—the ever-increasing amount, complexity, and diversity of software—to respond to this demand. C

Acknowledgments

The first author was supported by the European Research Council (ERC, grant agreement 851895), and by the German Research Foundation within the ConcSys and Perf4JS projects.

References

1. Allamanis, M., Barr, E., Devanbu, P., and Sutton, C. A survey of machine learning for big code and naturalness. *ACM Computing Surveys 51*, 4 (2018), 81.
2. Allamanis, M., Brockschmidt, M., and Khademi, M. Learning to represent programs with graphs. In *Proceedings of the 6th Intern. Conf. on Learning Representations* (Vancouver, BC, Canada, Apr. 30–May 3, 2018); https://openreview.net/forum?id=BJOFETxR-
3. Allamanis, M., Peng, H., and Sutton, C. A convolutional attention network for extreme summarization of source code. In *Proceedings of the Intern. Conf. on Learning Representations*, 2016, 2091– 2100.
4. Alon, U., Brody, S., Levy, O., and Yahav, E. code2seq: Generating sequences from structured representations of code. In *Proceedings of the 7th Intern. Conf. on Learning Representations* (New Orleans, LA, USA. May 6–9, 2019); https://openreview.net/forum?id=H1gKYo09tX
5. Alon, U., Zilberstein, M., Levy, O., and Yahav, E. code2vec: Learning distributed representations of code. In *Proceedings of ACM Program. Lang. 3*, 2019, 40:1–40:29; https://doi.org/10.1145/3290353
6. Bavishi, R., Lemieux, C., Fox, R., Sen, K., and Stoica, I. AutoPandas: neural-backed generators for program synthesis. In *Proceedings of ACM Program. Lang.* 2019, 168:1–168:27. https://doi.org/10.1145/3360594
7. Chen, Z., Kommrusch, S., Tufano, M., Pouchet, L., Poshyvanyk, D., and Monperrus, M. SequenceR: Sequence-to-sequence learning for end-to-end program repair. *IEEE TSE* (2019).
8. David, Y., Alon, U., and Yahav, E. Neural reverse engineering of stripped binaries using augmented control flow graphs. In *Proceedings of ACM Program. Lang. 4*, 2020, 225:1–225:28; https://doi.org/10.1145/3428293
9. Devlin, J., Uesato, J., Bhupatiraju, S., Singh, R., Mohamed, A., and Kohli, P. RobustFill: Neural program learning under noisy I/O. In *Proceedings of the 34th Intern. Conf. on Machine Learning* (Sydney, Australia, Aug. 6–11, 2017). D. Precup and Y.W. The, Eds. PMLR, 990–998; http://proceedings.mlr.press/v70/devlin17a.html
10. Dinella, E., Dai, H., Li, Z., Naik, M., Song, L., and Wang, K. Hoppity: Learning graph transformations to detect and fix bugs in programs. In *Proceedings of the 8th Intern. Conf. on Learning Representations* (Addis Ababa, Ethiopia, Apr. 26–30, 2020); https://openreview.net/forum?id=SJeqs6EFvB
11. Gu, X., Zhang, H., and Kim, S. Deep code search. In *Proceedings of the 40th Intern. Conf. on Softw. Eng.* (Gothenburg, Sweden, May 27–June 03, 2018). M. Chaudron, I. Crnkovic, M. Chechik, and M. Harman, Eds. ACM, 933–944; https://doi.org/10.1145/3180155.3180167
12. Gupta, R., Kanade, A., and Shevade, S. Neural attribution for semantic bug-localization in student programs. In *Proceedings of the Annual Conf. on Neural Information Processing Systems* (Vancouver, BC, Canada, Dec. 8–14, 2019). H.M. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E.B. Fox, and R. Garnett, Eds. 11861–11871; https://proceedings.neurips.cc/paper/2019/hash/f29a179746902e331572c483c45e5086-Abstract.html
13. Gupta, R., Pal, S., Kanade, A., and Shevade, S. DeepFix: fixing common C language errors by deep learning. In *Proceedings of the 31st AAAI Conf. on Artificial Intelligence* (San Francisco, CA, USA, Feb. 4–9, 2017). Satinder P. Singh and Shaul Markovitch, Eds. AAAI Press, 1345–1351; http://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14603
14. Hellendoorn, V., Bird, C., Barr, E., and Allamanis, M. Deep learning type inference. In *Proceedings of the 2018 ACM Joint Meeting on European Softw. Eng. Conf. and Symp. Foundations of Softw. Eng.* (Lake Buena Vista, FL, USA, Nov. 4–9, 2018). G.T. Leavens, A. Garcia, and C.S. Pasareanu, Eds. ACM, 152–162; https://doi.org/10.1145/3236024.3236051
15. Hellendoorn, V., Sutton, C., Singh, R., Maniatis, P., and Bieber, D. Global relational models of source code. In *Proceedings of the 8th Intern. Conf. on Learning Representations*, (Addis Ababa, Ethiopia, Apr. 26–30, 2020). OpenReview.net; https://openreview.net/forum?id=B1lnbRNtwr
16. Hindle, A., Barr, E., Su, Z., Gabel, M., and Devanbu, P. On the naturalness of software. In *Proceedings of the 34th Intern. Conf. on Softw. Eng.* (Zurich, Switzerland), June 2–9, 2012), 837–847.
17. Kanade, A., Maniatis, P., Balakrishnan, G., and Shi, K. Learning and evaluating contextual embedding of source code. In *Proceedings of the 37th Intern. Conf. on Machine Learning*, (Virtual event, July 13–18, 2020), PMLR, 5110–5121; http://proceedings.mlr.press/v119/kanade20a.html
18. Karampatsis, R., Babii, H., Robbes, R., Sutton, C., and Janes, A. Big code = big vocabulary: Open-vocabulary models for source code. In *Proceedings of 42nd Intern. Conf. on Softw. Eng.* (Seoul, South Korea, June 19–27, 2020). G. Rothermel and D-H Bae, Eds. ACM, 1073–1085; https://doi.org/10.1145/3377811.3380342
19. Kim, S., Zhao, J., Tian, Y., and Chandra, S. Code prediction by feeding trees to transformers. In *Proceedings of IEEE/ACM Intern. Conf. on Softw. Eng.*
20. Lacomis, J., Yin, P., Schwartz, E., Allamanis, M., Goues, C., Neubig, G., and Vasilescu, B. DIRE: A Neural Approach to Decompiled Identifier Naming. *ASE*, 2019.
21. Li, J., Wang, Y., Lyu, M., and King, I. Code completion with neural attention and pointer networks. In *Proceedings of the 27th Intern. Joint Conf. on Artificial Intelligence* (Stockholm, Sweden, 2018). AAAI Press, 4159–25.
22. Li, Y., Wang, S., and Nguyen, T. DLFix: context-based code transformation learning for automated program repair. *ICSE*, 2020.
23. Li, Z., et al. VulDeePecker: A deep learning-based system for vulnerability detection. *NDSS*, 2018.
24. Malik, R., Patra, J., and Pradel, M. NL2Type: Inferring JavaScript function types from natural language information. In *Proceedings of the 41st Intern. Conf. on Softw. Eng.* (Montreal, QC, Canada, May 25–31, 2019), 304–315; https://doi.org/10.1109/ICSE.2019.00045
25. Mikolov, T., Sutskever, I., Chen, K., Corrado, G., and Dean, J. Distributed representations of words and phrases and their compositionality. In *Proceedings of the 27th Annual Conf. on Neural Information Processing Systems* (Lake Tahoe, NV, USA, Dec. 5–8, 2013), 3111–3119.
26. Mou, L., Li, G., Zhang, L., Wang, T., and Jin, Z. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the 30th AAAI Conf. on Artificial Intelligence*, (Phoenix, AZ, USA, Feb. 12–17, 2016), 1287–1293.
27. Panthaplackel, S., Gligoric, M., Mooney, R., and Li, J. Associating natural language comment and source code entities. In *Proceedings of the 34th AAAI Conf. on Artificial Intelligence*; the *32nd Innovative Applications of Artificial Intelligence Conf*; and the *10th AAAI Symp. Educational Advances in Artificial Intelligence*, (New York, NY, USA, Feb. 7–12, 2020). AAAI Press, 8592–8599; https://aaai.org/ojs/index.php/AAAI/article/view/6382
28. Pradel, M., Gousios, G., Liu, J., and Chandra, S. Typewriter: neural type prediction with search-based validation. In *Proceedings of the 28th ACM Joint European Softw. Eng. Conf. and Symp. Foundations of Softw. Eng.* (Virtual Event, USA, Nov. 8–13, 2020), 209–220; https://doi.org/10.1145/3368089.3409715
29. Pradel, M., Murali, V., Qian, R., Machalica, M., Meijer, E., and Chandra, S. Scaffle: bug localization on millions of files. In *Proceedings of the 29th ACM SIGSOFT Intern. Symp. Softw. Testing and Analysis* (Virtual Event, USA, July 18–22, 2020). S. Khurshid and C.S. Pasareanu, Eds. ACM, 225–236; https://doi.org/10.1145/3395363.3397356
30. Pradel, M. and Sen, K. DeepBugs: A learning approach to name-based bug detection. *PACMPL 2 OOPSLA* (2018), 147:1–147:25; https: //doi.org/10.1145/3276517
31. Raychev, V., Vechev, M.T. and Krause, A. Predicting program properties from "Big Code." *Principles of Programming Languages*, (2015), 111–124.
32. Rozière, B., Lachaux, M., Chanussot, L., and Lample, G. Unsupervised translation of programming languages. In *Proceedings of the 2020 Conf. on Neural Information Processing Systems.* (Virtual Event, Dec. 6–12, 2020). H. Larochelle, M'A Ranzato, R. Hadsell, M-F Balcan, and H-Tien Lin, Eds; https://proceedings.neurips.cc/paper/2020/hash/ed23fbf18c2cd35f8c7f8de44f85c08d-Abstract.html
33. Sachdev, S., Li, H., Luan, S., Kim, S., Sen, K., and Chandra, S. Retrieval on source code: A neural code search. In *Proceedings of the 2nd ACM SIGPLAN Intern.Workshop on Machine Learning and Programming Languages*. ACM, 31–41.
34. Svyatkovskiy, A., Deng, S., Fu, S., and Sundaresan, N. IntelliCode compose: Code generation using transformer. In *Proceedings of the 28th ACM Joint European Softw. Eng. Conf. and Symp. Foundations of Softw. Eng.* (Virtual Event, USA, Nov. 8–13, 2020). P. Devanbu, M.B. Cohen, and T. Zimmermann, Eds.ACM, 1433–1443; https://doi.org/10.1145/3368089.3417058
35. Tarlow, D. et al. Learning to fix build errors with Graph2Diff neural networks. In *Proceedings of the 42nd Intern. Conf. on Softw. Eng. Workshops* (Seoul, Republic of Korea, June 29–July 19, 2020). ACM, 19–20; https://doi.org/10.1145/3387940.3392181
36. Tufano, M., Pantiuchina, J., Watson, C., Bavota, G., and Poshyvanyk, D. On learning meaningful code changes via neural machine translation. In *Proceedings of the 41st Intern. Conf. on Softw. Eng.* (Montreal, QC, Canada, May 25–31, 2019), 25–36; https://dl.acm.org/citation.cfm?id=3339509
37. Vaswani, A. et al. Attention is all you need. In *Proceedings of the 2017 Conf. on Neural Information Processing Systems* (Long Beach, CA, USA, Dec. 4–9, 2017), 6000–6010; http://papers.nips.cc/paper/7181-attention-is-all-you-need
38. Wainakh, Y., Rauf, M., and Pradel, M. IdBench: Evaluating semantic representations of identifier names in source code. In *Proceedings of the 43rd IEEE/ACM Intern. Conf. on Softw. Eng.* (Madrid, Spain, May 22–30 May 2021), 562–573.
39. Wei, J., Goyal, M., Durrett, G., and Dillig, I. LambdaNet: probabilistic type inference using graph neural networks. In *Proceedings of the 8th Intern. Conf. on Learning Representations* (Addis Ababa, Ethiopia, Apr. 26–30, 2020). OpenReview.net; https://openreview.net/forum?id=Hkx6hANtwH
40. White, M., Tufano, M., Vendome, C., and Poshyvanyk, D. Deep learning code fragments for code clone detection. *ASE*. 87–98.

Michael Pradel is a professor in the Computer Science Department at the University of Stuttgart, Germany.

Satish Chandra is a software engineer at Facebook, Menlo Park, CA, USA, where he also leads the Big Code group.