Deep Learning for Source Code Modeling and Generation: Models, Applications, and Challenges

TRIET H. M. LE, HAO CHEN, and MUHAMMAD ALI BABAR, The University of Adelaide

Deep Learning (DL) techniques for Natural Language Processing have been evolving remarkably fast. Recently, the DL advances in language modeling, machine translation, and paragraph understanding are so prominent that the potential of DL in Software Engineering cannot be overlooked, especially in the field of program learning. To facilitate further research and applications of DL in this field, we provide a comprehensive review to categorize and investigate existing DL methods for source code modeling and generation. To address the limitations of the traditional source code models, we formulate common program learning tasks under an encoder-decoder framework. After that, we introduce recent DL mechanisms suitable to solve such problems. Then, we present the state-of-the-art practices and discuss their challenges with some recommendations for practitioners and researchers as well.

CCS Concepts: • **General and reference** → **Surveys and overviews**; • **Computing methodologies** → **Neural networks**; *Natural language processing*; • **Software and its engineering** → **Software notations and tools**; **Source code generation**;

Additional Key Words and Phrases: Deep learning, big code, source code modeling, source code generation

ACM Reference format:
Triet H. M. Le, Hao Chen, and Muhammad Ali Babar. 2020. Deep Learning for Source Code Modeling and Generation: Models, Applications, and Challenges. *ACM Comput. Surv.* 53, 3, Article 62 (June 2020), 38 pages.
https://doi.org/10.1145/3383458

1 INTRODUCTION

Deep Learning (DL) has recently emerged as an important branch of Machine Learning (ML) because of its incredible performance in Computer Vision and Natural Language Processing (NLP) [75]. In the field of NLP, it has been shown that DL models can greatly improve the performance of many classic NLP tasks such as semantic role labeling [96], named entity recognition [210], machine translation [256], and question answering [175]. Similarly, source code is a special type of structured natural language written by programmers [101], which can be analyzed by DL models. Machine intelligence that understands and creates complex structure of software has a lot of applications in Software Engineering (SE). Specifically, Big Code is the research area that uses ML and DL for source code modeling.[1] To facilitate the building of ML models, software community offers valuable datasets such as online code corpora like GitHub, question answering

[1] http://science.dodlive.mil/2014/03/21/darpas-muse-mining-big-code/.

Authors' addresses: T. H. M. Le, H. Chen, and M. A. Babar, School of Computer Science, University of Adelaide, Adelaide, SA 5005, Australia; emails: {triet.h.le, hao.chen01, ali.babar}@adelaide.edu.au.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2020 Association for Computing Machinery.
0360-0300/2020/06-ART62 $15.00
https://doi.org/10.1145/3383458

```
try {
    open(file);
    // Do something that can raise exception
    close(file); // Error occurs when the exception is raised.
}
catch (Exception e) {
    // Handle exception
}
```

Listing 1. A buggy code snippet that involves long-term dependencies.

```
try {
    open(file);
    // Do something that can raise exception
}
catch (Exception e) {
    // Handle exception
}
finally {
    close(file); // Always close the file, which fixes the bug.
}
```

Listing 2. A clean code snippet that involves long-term dependencies.

forums like Stack Overflow, as well as documentation of various software and programming tools that are highly rich in content. There have been many applications of ML in SE thesaurus construction [40], language models for code [54], and information retrieval of answers, projects, and documentation [39, 271, 279]. Compared to other domains, DL for Big Code is still growing and requiring more research, tasks, and well-established datasets.

Among the Big Code tasks, source code generation is an important field to predict explicit code or program structure from multimodal data sources such as incomplete code, programs in another programming language, natural language descriptions, or execution examples [5]. Code generation tools can assist the development of automatic programming tools to improve programming productivity. Such models can also represent the temporal context and discrete logic of programs. However, traditional source code models were either inflexible [25, 220], time-consuming to design for specific languages/tasks [87, 92], or unable to capture long-term dependencies of code statements [9, 222]. Listings 1 and 2 show that the difference of buggy and clean versions of the same code snippet depends on the relative location of the close(file) function with respect to the open(file) function, which is an example of long-term dependency in source code. DL models can help address the aforementioned issues because of their superior ability in extracting features from various data formats (e.g., natural language and symbol sequences) and capturing both syntactic and semantic information at various scales [96].

There is an extensive review on ML for Big Code focusing on probabilistic models [5]. Compared to this previous work, here are our major differences:

- Extensive coverage of state-of-the-art DL models and their extension to source code modeling and generation;

- A systematic mapping of Big Code tasks based on their inputs (encoder) and outputs (decoder) for applying DL models;
- List of datasets for source code modeling and generation tasks;
- Challenges and future directions for deep source code models.

With such significant differences, our article gives a more holistic view of applying DL to source code modeling and generation as compared to Reference [5]. Recently, there has also been a review [213] on DL architectures and its applications in various domains such as computer vision, NLP, and social network analysis. Unlike the previous work, our article focuses more on many practical applications in source code modeling and generation, and then it illustrates how DL models and encoder-decoder framework can be used to address such problems. This work will be a useful guide for practitioners and researchers from both DL and SE fields when working with source code.

The remaining of this literature review is organized as follows: Section 2 presents existing language models for source code as well as their limitations, which motivates the use of DL models. Section 3 formulates source code modeling under encoder-decoder framework and describes important components of such framework. Section 4 highlights the recent practices for building deep source code models. Section 5 reviews DL-based applications for various Big Code tasks. Section 6 presents the available datasets for such tasks. Section 7 discusses the current challenges and proposes some future directions in using DL for source code modeling and generation. Finally, Section 8 summarizes the main contributions of this literature review.

2 TRADITIONAL SOURCE CODE MODELING APPROACHES AND THEIR LIMITATIONS

We describe four traditional approaches to handle the syntactic and semantic facets of source code including (*i*) domain-specific language guided models, (*ii*) probabilistic grammars, (*iii*) simple probabilistic language models (i.e., *n*-grams), and (*iv*) simple neural language models. Several serious issues still associate with these traditional approaches, which can be handled effectively using DL models. More details of each approach and its limitations are covered in this section.

2.1 Domain-specific Language Guided Models

Domain-Specific Languages (DSLs) are often used to define the parametrization rules and states for generating the structure of a program. DSL-based models create different grammar rules for common code statements (e.g., control flow, comments, and brackets). Compared to a general-purpose programming language, the grammar size of a DSL is usually smaller, which makes it more efficient for specific code generation tasks. DSL-based model has been studied by Gulwani et al. [87] and Jha et al. [115]. Gvero et al. [92] reduced the search space for Scala expression suggestion using a calculus of *succinct types* and designed a higher-order function for code completion. Since the generation process is human-interpretable, this type of model can be a promising approach in SE.

In program induction, a DSL specifies the space of candidate programs (*program template*), while the example input-output pairs are known as *specification*. Under this scenario, the problem is known as Inductive Logic Programming (ILP) [187]. Two classic families of solving ILP are the bottom-up approaches, constructing programs from example features, and the top-down approaches, testing examples from generations and adjusting the rules accordingly [64]. Given the precise and combinatorial nature of induction, the induction is commonly cast as a Constraint Satisfaction Problem (CSP) [242]. This belongs to the top-down family of ILP [64]. The formal definition of the problem can be found in the work of Pu et al. [215]. Such a CSP problem can be solved by a constraint solver such as Z3 [58]. The problem with this approach is that the solver is

always heuristic and its scalability is poor. Often these systems cannot handle noisy, erroneous, or ambiguous data.

DSL-guided models can capture the structure of a specific programming language; however, they require deep domain knowledge to generate the detailed syntactic and semantic rules. One possible way to increase flexibility is to represent a DSL with a probabilistic model.

2.2 Probabilistic Grammars

In formal languages, production rules define all possible generations of strings (code statements). Context-Free Grammars (CFGs) is a set of production rules that can be applied regardless of context, i.e., the left-hand side of a production rule only contains a single non-terminal symbol. CFG is a common way to define the structure of a programming language, which can then be used to convert its source code into Abstract Syntax Trees (ASTs) [103]. Probabilistic Context-Free Grammar (PCFG) is an extension to CFG, in which the production rules of a context-free language are associated with a probabilistic model. Bielik et al. [25] generalized the idea of PCFG to the task of code completion in other non-context-free languages like JavaScript. Later, Raychev et al. [220] extended the previous work on PCFGs by learning a decision tree to build a probabilistic model using ASTs of a proposed DSL, namely, TGen. These works achieved strong results for some code completion tasks.

Another extension of CFG, namely, Tree Substitution Grammar (TSG), uses tree fragments instead of a sequence of symbols for the production rules. Such rules of TSG are defined by a Tree Adjoining Grammar [117], which can create more flexibility and better represent complex linguistic structures [49]. To limit the model complexity and sparse grammar, researchers often use non-parametric Bayes to infer the distributions [8, 49]. These models are suitable for pattern mining, since their automatic model selection ability allows the discovery of more complex structures. However, non-parametric Bayesian methods are often extremely slow to compute and hard to scale. Although probabilistic grammars achieve high performance for domain-specific languages, they still require manually designed rules to model code locality and reuse.

2.3 *n*-gram Language Models

Besides the two syntactic models above, one simple yet effective statistical language model is *n*-gram. More specifically, this model assumes each token/word is conditionally dependent on the previous $n-1$ tokens/words as described in the following equation: $P(\mathbf{W}) = \prod_t P(w_t|w_{t-(n-1),\ldots,t-1})$, where $P(w_t|w_{t-(n-1),\ldots,t-1})$ can be simply computed by counting the occurrences of all *n*-grams in the training set. A direct advantage of *n*-grams over syntactic models (e.g., probabilistic grammars and DSL-guided models) is that it is easier to generalize, since the dependencies and rules of the programming languages are learned automatically from source code. Hindle et al. [101] took the initiative to utilize *n*-gram to build a language model for source code. Since then, besides code completion [194, 222], *n*-gram models have also been applied to other tasks such as idiom mining [7], syntax error detection [36], source code analysis [104], and code obfuscation [161]. However, simple language models like *n*-gram cannot capture high-level programming paradigms.

To address this issue, a line of work has enhanced language models to be more adaptable to local information. Bielik et al. [26] augmented a DSL-based model with *n*-gram and showed strong empirical results for programming language modeling. The resulting model was also more efficient in training and inference compared to neural models. Hellendoorn et al. [98] added a local *n*-gram cache and merged the predictions of local and global models. Both models were claimed to surpass DL counterparts (e.g., RNN and LSTM) at the time of their publication; thus, these two models would be good baselines for source code modeling.

It is noted that *n*-gram is not good at modeling long-term dependencies (cf. Listings 1 and 2) between tokens. The *n*-gram truncation discards long-term positional information. Other statistical models, such as Hidden Markov models, also fail to encode long-term history [247], since its state space becomes exponentially large when encoding several previous history tokens into one state. Besides, another limitation with an *n*-gram model is the sparsity of the vector representation of the word or code token, which is caused by the large vocabulary of source code. This sparsity issue can be solved using the distributed representation of neural language models.

2.4 Simple Neural Program Models

Instead of incorporating the frequency of each previous token explicitly, neural network embedding models with one hidden layer first convert one-hot encoding of a word into an intermediate word-embedding vector with a much shorter length (e.g., 100–1000) compared to the vocabulary size. This idea is also known as *distributed word representation*. In its original work [233], the word embeddings of up to *n* - 1 previous tokens with respect to the current word were fed to a fully connected neural network with one hidden layer. At the output layer, a softmax function was applied to calculate the probability of the next word. However, a serious drawback of this original model is the high computational cost of the hidden layer. Therefore, log-bilinear was presented [182] to address this challenge by replacing the non-linear activations with a context matrix to determine the context vector with respect to the current word. Then, the similarity between the context vector of the previous tokens and the current word was computed. Later, several methods were proposed to speed up the training and prediction time using hierarchical architecture [183]. In this previous work, the log-bilinear model was demonstrated to outperform the traditional fully connected neural network and *n*-gram language models for the task of modeling APNews.

Later, the idea of neural network embedding model was adopted by researchers for source code modeling, which can be referred to as *simple neural program models*. More specifically, Maddison et al. [164] combined log-bilinear with a tree depth-first search traversal technique (i.e., Log-bilinear Tree-Traversal models) to generate human-understandable source code. Allamanis et al. [9] extended Maddison's approach to retrieve source code snippets from natural language queries and vice versa. Allamanis et al. [4] also used a log-bilinear model to recommend method and class names for object-oriented programming in Java, and this model outperformed an *n*-gram model in both of the tasks. Similar to *n*-gram models, the knowledge of log-bilinear models is limited to the previous *n* - 1 tokens. Therefore, the above works needed to define the global and local context explicitly for log-bilinear models to capture the short-term, long-term dependencies and sequential property of source code. The list of contexts is still human-crafted and incomplete, thus limiting its applications in new domains. However, with their simplicity, simple neural program models are being used as (pre-trained) input features (cf. Section 3.2) for various Big Code tasks.

2.5 Advantages of Deep Learning Models over Traditional Approaches

Encoder-decoder framework [248] with DL models (cf. Section 3) can be used to effectively capture the dependencies and sequential property of an input sequence. To be more specific, DL models are suitable for code modeling and generation, since they are good at the following four important aspects: (*i*) automatic feature generation, (*ii*) capturing long-term dependencies as well as sequential property, (*iii*) end-to-end learning, and (*iv*) generalizability. Existing models have to trade off among these four properties. For example, *n*-gram models can automatically extract features from source code, but cannot capture long-term dependencies well due to the combinatorial explosion of terms. Similarly, simple neural program models (e.g., fully connected or log-bilinear models) still require human-designed rules to capture the dependencies and structure of source code, which limits its end-to-end training and generalizability. In contrast, with deep domain knowledge

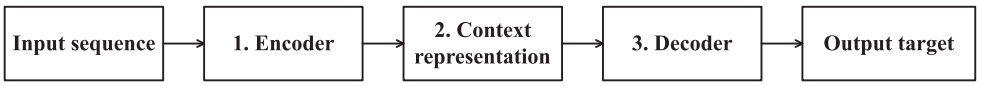

Fig. 1. Main steps of an encoder-decoder framework.

incorporated, DSL-guided models and probabilistic grammars can effectively capture the dependencies and sequential properties, but the strictly defined rules make the models harder to generalize and automate the learning process in new domains. Next, encoder-decoder framework using DL models for sequence/code modeling is presented.

3 DEEP SEQUENCE MODELING WITH ENCODER-DECODER FRAMEWORK

In NLP, the main objective is to process a large amount of natural language mostly in the form of text or human voice. Code—a means of communication for developers—is similar to natural language [5, 101], which has syntactic structures and semantic meaning. Specifically, there are various Big Code tasks (cf. Section 5) whose input is a sequence (e.g., source code and/or natural language) and prediction target can be either a sequence or just a simple numeric/categorical value. Inspired by the field of NLP, such Big Code tasks can be formulated under an *encoder-decoder framework*. If both the input and output are sequences, then encoder-decoder framework can also be called seq2seq [248]. The main steps of an encoder-decoder framework are illustrated in Figure 1.

The components for steps 1 and 3 are referred to as encoder and decoder, respectively, which are typically two DL models (cf. Section 3.1). In the original paper [248], the final internal state of the encoder is used as the context vector. In more recent works, step 2 is often handled by attention mechanism [16] (cf. Section 3.1.3) or external memories [265] (cf. Section 3.1.4) with a richer and position sensitive context sequence. In this section, three main elements of an encoder-decoder learning framework for sequence modeling, including (*i*) DL models, (*ii*) input embeddings, and (*iii*) stable training of such models, are covered. It should be noted that both sequential models (i.e., modeling word-by-word) and structural models (i.e., exploiting syntactic structure of a sentence/code snippet) can be used for sequence modeling. Although Section 3 mainly reviews different components of an encoder-decoder framework originally proposed for sequence modeling in NLP, this section is still important because of two reasons: (*i*) the presented deep models/techniques can be extended to source code and (*ii*) only a small portion of such models has been utilized for source code modeling. Based on this section, Section 4 subsequently presents the current practices of source code modeling and generation using encoder-decoder framework.

3.1 Deep Learning Models for Sequence Modeling

Two main classes of DL models for sequence modeling, namely, (*i*) recurrent and (*ii*) non-recurrent neural networks, are first presented. Then, three techniques to build more robust models including (*i*) attention mechanism, (*ii*) external memory, and (*iii*) beam search are covered. It is noted that Multi-Layer Perceptron (MLP) [227] (a.k.a. fully connected/feed-forward neural network) is not reviewed in this section, since it is an extension of the neural network described in Section 2.4 (i.e., with more hidden layers). Thus, MLP is still limited in capturing the dependencies and sequential property of a sequence and not widely used for sequence modeling unless combined with more advanced techniques such as attention mechanism (cf. Section 3.1.3).

3.1.1 Recurrent Neural Networks. Recurrent Neural Network (RNN) and its variants have been widely used for building language models [138, 171, 172]. RNN is a special type of deep neural network, in which a block of parameters is shared and repeated many times across different parts

of a sequence, resulting in a deep computational graph [75]. This architecture helps a network to learn with input/output of various lengths that MLPs cannot.

However, vanilla (plain) RNNs are hard to train [207] and are not good at keeping past information from different time scales. Gated RNNs, such as Long Short-Term Memory (LSTM) [102] and Gated Recurrent Units (GRUs) [46], model the keeping and forgetting mechanisms explicitly with sigmoid activations, namely, *gates*. An LSTM has three gates to control input, output, and forgetting, respectively. In addition, there is a memory cell state to generate the hidden states.

RNN units can be made deep to encode more complex transitions [206]. Highway layers [244] were introduced to stabilize the training gradients in Recurrent Highway Networks [289]. To capture the long-term dependencies in time series and represent hierarchical information, RNNs layers can be stacked with different update frequencies [137]. The gated feedback RNNs [47] allow the network to learn its own clock rates by using additional gates. As of the writing of this review, the state-of-the-art RNN for language model is the Fast-Slow RNN [188], which incorporates the strengths of both deep and multiscale RNNs.

3.1.2 Non-recurrent Neural Networks. Temporal convolution or one-dimensional convolution across time is another type of neural network that can capture long-term relations with hierarchical architecture [258]. It has been applied to sentiment analysis, sentence classification, machine translation, and meta-learning [178].

Convolutional Neural Networks (CNNs) have been used in several sentence modeling tasks. In 2013, Kalchbrenner and Blunsom [121] used a CNN as the encoder and an RNN as the decoder for dialogue generation. One year later, Blunsom et al. proposed Dynamic CNN [27] for sentence semantic modeling, where variable length and relation discovery were enabled by max pooling. However, these earlier works failed to achieve similar performance of LSTMs.

Recently, non-recurrent structures have emerged again with similar performance as RNN, but they are faster to compute. Masked convolution layers are used as the decoder in a neural machine translation system [122]. Gehring et al. [73] proposed ConvS2S that brought skip connection [95] and attention [16] (cf. Section 3.1.3) to sentence modeling and achieved the state-of-the-art translation performance. Combining recurrent and convolutional units is also useful. He et al. [97] strengthened the input-to-output correlation by adding *cross-layer convolutions* to stacked RNNs.

Vaswani et al. [256] proposed a multi-head attention model called Transformer relying on self-attention and positional encoding to compute the sequence representations. Transformer allows the decoder to attend to information arbitrarily far and reduced training time significantly without quality loss. Like CNN, the causal structure is held by masking later output for the autoregressive factorization. Recently, a deep language representation model, BERT [59], utilizing a novel bidirectional training of a Transformer has achieved the state-of-the-art results in 11 NLP tasks such as sentence classification/tagging, question answering, and named entity recognition. Later, BERT was extended to language modeling [53] and language generation [62] by addressing its limitations of fixed-length contexts and bidirectional nature, respectively. It is noted that Transformers can be leveraged for contextual (same token with varying usages) embeddings of source code.

Speeding up the sequential generation is another interesting direction. It should be noted that all autoregressive models only generate samples sequentially, since they use ancestral sampling. Thus, alternative architectures for rapid, parallel sample generation are required. Gu et al. [83] enabled non-autoregressive learning by sampling a latent variable representing the *fertilities*, i.e., the usage of each source word in decoding, which required to be supervised by an external alignment system. Inverse-Autoregressive Flows (IAFs) [130] could generate high-dimensional samples in parallel from latent variables. Oord et al. [198] incorporated WaveNet with IAF to create parallel WaveNet, which sampled with higher fidelity, but ran 3000 times faster in audio generation.

3.1.3 Attention Mechanism. One problem of the original encoder-decoder framework is that decoder can only access a single context vector. Human understands text lines by repeatedly attending to different parts of a sequence. To simulate this behavior, Bahdanau et al. [16] used a sequence as the context and proposed *attention mechanism* to adapt the weights of context associated with a certain output stage and impose an explicit alignment between input and output tokens. To be more specific, with an encoded sequence F, and at each step t, the hidden state $\mathbf{h}_t$ is computed using an RNN model with input source vector $\mathbf{c}_t$ generated by attention mechanism as additional input: $\mathbf{h}_t = \text{RNN}(\mathbf{h}_{t-1}, [\mathbf{e}_{t-1}; \mathbf{c}_t])$. This way of incorporating attention is known as *early binding*. Alternatively, attention can be considered just before generating the output token. A typical soft attention similar to Bahdanau et al.'s [16] can be computed as follows:

(1) With the previous hidden state $\mathbf{h}_{t-1}$, the *attention energy* is computed with a content-based scoring function $\mathbf{u}_t = \text{score}(F, \mathbf{h}_{t-1})$.
(2) Exponentiate and normalize u_t to 1: $\mathbf{a}_t = \text{softmax}(\mathbf{u}_t)$.
(3) Compute the *input source vector* $\mathbf{c}_t = F\mathbf{a}_t$.

The simplest way to define the score is dot-product, $\text{score}(F, \mathbf{h}_{t-1}) = F^T\mathbf{h}_{t-1}$. Or an *expected input embedding* V can be defined so that $\text{score}(F, \mathbf{h}_{t-1}) = F^T V\mathbf{h}_{t-1}$. In the original paper [16], the attention energy is computed with a Multi-Layer Perceptron (MLP) as follows:

$$\text{score}(F, \mathbf{h}_{t-1}) = \mathbf{v}^T \tanh(WF + V\mathbf{h}_{t-1}).$$

Content-based attention energy is computed by scoring each element separately, which makes it hard to discriminate the elements with similar content from different locations. Location-sensitive attention [45] broke through this limitation by generating attentions in an autoregressive fashion. However, unrolling through another RNN during backpropagation can greatly increase the computational time. Vaswani et al. [256] presented multi-head self-attention to represent the relevant context of each word in an input sequence at different locations. By combining sequence modeling and attention mechanism, the state-of-the-art results have been achieved for neural machine translation [16, 116, 144, 270].

3.1.4 Memory-augmented Neural Networks. Attention is closely related to external memory. Together, they have become important building blocks for a neural network. External memories are used as internal states, which can be updated by attention mechanism for selective reading and updating. The classic Memory Network (MemNN) [265] tried to mimic a Random Access Memory (RAM) and use soft attention as a differentiable version of addressing.

A memory network usually takes the following inputs:

- A *query* q is the last utterance the speaker said in a general dialogue setting, or the question in a Question Answering setting.
- A *memory* vector $\mathbf{m}$ is the dialogue history of the model. The knowledge can be as large as the whole codebase or documentation given the model is sufficiently powerful.

And this type of network has the following modules to handle the inputs:

- An *encoder* converts q into a vector using an RNN [44] or a simpler word embedding [174].
- A *memory module* M finds the best part of $\mathbf{m}$ related to q. This is the *addressing* stage.
- A *controller module* C sends q to M and reads back the relevant memory, adds that to the current state. In practice, we always cycle this process to enable complicated reasoning.
- A *decoder* generates output from the final states.

The training of MemNN is fully supervised, in which the label of the best part of memory is given at every stage of the memory addressing. Its follow-up End-to-End Memory Network [245] uses soft attention for memory addressing to train the attention in backpropagation and relax the supervision to only the output. It is noted that using one memory unit to match both the query and the state limits the expressiveness. By splitting the memory into a key-value pair, Millor et al. [177] encoded prior knowledge and obtained better results. Recurrent Entity Network [99] demonstrates how to enhance the memory unit by letting the agent learn to read and write the memory to track the facts. Weston [266] generalized this model to unsupervised learning by adding a new stage to generate answers and predict replies. This kind of model is evaluated on various tasks such as basic reasoning on 20 bAbI tasks [264], reading children's books [100], and understanding real dialogues from movies [60]. All of the tasks can be found on the website of the bAbI project.[2] Recently, many works have tried to make traditional memory paradigms differentiable so the models can be optimized with SGD. Such end-to-end trainable models have been used to handle algorithmic learning and reasoning tasks such as language understanding and program induction, which are covered in more detail in Section 4.3.

3.1.5 Beam Search. Searching for the best-decoded result with the highest probability is computationally intractable. In other words, there can be an exponentially large number of generated sentences in NLP or source code in Big Code. One solution would be to choose the word/token with the highest output probability after each time step during the decoding process. However, this greedy process will likely result in a sub-optimum result. Therefore, in machine translation, beam search is widely adopted as the heuristic search technique [248]. Instead of taking the next word with the highest possibility directly, a list of previous most likely partial translations is kept and used to extend the corresponding translation at the current step. After each step, we re-rank the generated sequences and keep only the most probable ones. The length of the list to keep at each time step is known as *beam size*. This method often improves the translation, but the performance is closely dependent on the beam size [134].

3.2 Input Embeddings of Deep Learning Models

In this section, the input representation for sequence modeling is presented. This is also important for code modeling, since keyword representations can vary hugely within the scopes of functions, classes, and projects. Such large vocabulary makes traditional representations such as one-hot encoding or *n*-grams form a very sparse embedding vector. As a result, in recent deep language models, input words are often converted into real-valued vectors with distributed representations [176]. Words in the embedded space demonstrate nice emergent properties such as semantic relationship that can be represented as vector arithmetic. Sharing the embedding weights with the softmax layer in the decoding process results in notable improvements for language models [107].

For NLP tasks, it is a common practice to use general-purpose word vectors pre-trained on large corpora such as word2vec [176], GloVe [208], and fastText [28]. The parameters can also be optimized for specific datasets and tasks together with the model. McCann et al. [168] performed the word vector training based on a machine translation task, which helped their word vectors (i.e., CoVe) to capture more complex contextual information. They showed that replacing traditional word vectors with CoVe could improve the performances of many tasks. The pre-trained word embeddings can also be fine-tuned to adapt to the Big Code task of interest. However, in the domain of source code modeling, the vocabulary size is much larger than that of NLP. As a result, re-training the word embeddings using initialization of the pre-trained parameters is also

[2] https://research.fb.com/projects/babi/.

worth considering. Some treatments of source code representation are presented in more detail in Section 4.1.

In Big Code applications, sub-word level inference is also substantially important. For example, code completion algorithms should be able to suggest incomplete function or variable names by seeing only a part of the word. To incorporate character-level information, the characters can be combined into word representation. Ling et al. [157] proposed C2W model, which uses Bidirectional LSTM to construct word embedding from character sequences. CNN can also be used for character-level embeddings [285].

3.3 Stable Training of Deep Learning Models

Recurrent models (e.g., RNN) for sequence modeling are hard to train [207] and, similar to other types of neural network, easily prone to overfitting. Like other DL architectures, RNN and its variants are usually trained using a special form of backpropagation, namely, *backpropagation through time*. There are several techniques for optimizing the loss function of a model. Among them, Stochastic Gradient Descent (SGD), or mini-batch gradient descent, is mostly adopted due to its efficient computation and parallelization on Graphics Processing Units [75]. Merity et al. [172] applied a non-monotonically triggered Averaged SGD [167] for language modeling and achieved superior results. It is noted that most of the recent papers on DL have reported their models trained by modern versions of SGD, namely, RMSProp [251] or Adam [129].

Model regularization also has a significant impact on the generalization performance. We review four common types of regularization for training DL models more effectively, including: (*i*) dropout, (*ii*) normalization, (*iii*) activation regularization, and (*iv*) structural regularization.

Dropout [243] randomly turns off several positions of the activation following a Bernoulli distribution. However, applying it to the RNN hidden states disrupts the model's ability to preserve long-term dependencies [283]. Two ways of retaining the information have been adopted. The first way is to limit the dropout rate of the hidden states by keeping previous information. Zoneout [139] randomly copies previous values of the activations rather than zeroing them out. Semeniuta et al. [235] applied dropout on the input gate to prevent memory loss. The second one is *locked dropout*, i.e., using the same dropout mask for a full forward pass. This method preserves the activation norms instead of gradually dropping information. Gal et al. [69] linked locked dropout with variational Bayes inference and used it for embedding dropout. In addition, locked DropConnect [259] on the hidden weights resulted in substantial improvements [172].

Normalization restricts the activations of different time steps to follow a stable distribution. Inspired by batch normalization [108], multiple normalization techniques customized for recurrent structures have been studied, such as recurrent batch normalization [50], weight normalization [230], and layer normalization [15]. There are also normalization techniques targeting the gradient stability. For example, spectral normalization [180] is designed to keep the gradient bounded by constraining the activations to be Lipschitz continuous.

Regularization can be applied to weights and activations of DL models. L_2 regularization on the weights is referred to as *weight decay*. *Activation regularization* penalizes activations with $\alpha L_2(m \cdot h_t)$, where α is a regularization term, m is a scaling factor, and h_t is the hidden state, respectively. Temporal activation regularization [173] penalizes the large changes in the hidden state of a neural model with $\beta L_2(h_t - h_{t-1})$, where β is a scaling factor, h_t and h_{t-1} are the hidden states at time t and $t-1$, respectively.

Structural regularization prevents exploding or vanishing gradients by restricting the model structure. Model structure restriction can be done by forcing the recurrent matrix to be unitary [13] or using element-wise interactions. Strongly typed RNNs [17] use type-consistent operations for the recurrent units. Other simplifications are Quasi-RNN [31] and Simple Recurrent Unit [148].

These regularization techniques are important to reduce the overfitting and improve the generalization performance of deep source code models, since the learned models can become overly complicated due to the need for representing various types of rules.

4 RECENT PRACTICES OF BUILDING DEEP LEARNING MODELS FOR SOURCE CODE MODELING AND GENERATION

This section focuses on the practices of developing DL models for source code under the encoder-decoder framework presented in Section 3. Specifically, we present the techniques for the following: (*i*) deep encoder models, (*ii*) deep decoder models, and (*iii*) deep controller models to better generalize the capabilities of DL models to source code modeling and generation.

4.1 Deep Encoder Models

For many deep source code tasks, the input takes the form of a sequence, such as code snippets, comments, or descriptions, where we rely on a deep module to capture the semantic and context of input for further processing. We call this kind of module deep encoder models. The most widely used encoder models are sequential models such as RNN and its variants. For general-purpose software repository mining, the effectiveness of RNNs has been tested [268]. However, sequential models may not be as effective for code modeling and generation due to the following limitations:

- Syntactic context is not well represented in a sequential model, which may lead to a violation of the grammar rules of a programming language.
- Large vocabulary of code leading to the out-of-vocabulary issue affects the generalizability of a deep code model.
- Recurrent models (e.g., RNNs) suffer from the hidden-state bottleneck, in which the size of hidden-state vector limits the information that a model can carry through time.

Many methods have been proposed to overcome these shortcomings. These are (*i*) structural (tree-/graph-based) representation, (*ii*) open vocabulary model, and (*iii*) attention mechanism.

Structural representation. Abstract Syntax Tree (AST) is a natural way to capture the syntactic structure of a program. In an AST, a program is parsed into a hierarchy of non-terminal and terminal (leaf) nodes based on the syntax of a programming language. To utilize AST for code representation, the simplest way would be to use depth-first search to convert an AST into a sequence [12, 54, 150]. Other studies proposed DL models (Recursive Neural Networks [267], Tree-LSTM [263], or CNN [186]) to work directly on the hierarchical structure of a parse tree. Recently, Zhang et al. [284] have shown that splitting an AST into code-statement subtrees can improve the performance of tree-based representations. Recent works (i.e., code2vec [11] and code2seq [10]) have also proposed to use AST paths as a representation for code, in which the extracted paths would be aggregated using an attention-based deep neural network. Recently, Allamanis et al. [6] have presented novel Gated Graph Neural Networks [151] to represent source code as a directed graph. Specifically, they incorporated data/control-flow information of variables into AST to capture the syntactic and semantic structures of source code more effectively. It is observed that structural representation of source code has witnessed a growing interest from the Big Code community. There is also a comprehensive review [43] on source code embeddings.

Open vocabulary model. The vocabulary of source code is open, rather than fixed. It is impractical to train a classifier using the whole vocabulary, so it is more common to truncate it by keeping only the most frequent 1K or 10K terms and replacing the others with an Out-of-Vocabulary (OoV) token (i.e., <unk>). The drawback of this truncation is that the OoV tokens (*neologisms* [4]) from a testing set cannot be predicted. To solve the OoV problem, Karampatsis et al. [124] have proposed a novel open-vocabulary neural language model for source code modeling. This work used GRU to

build a neural language model on top of sub-word units (character subsequences of code tokens) generated by the Byte pair encoding algorithm [67]. The large-scale experiments showed that this proposed sub-word neural program model was better than the state-of-the-art n-gram model and also more robust against the OoV problem across different programming languages and projects. Character-based DL models [125, 128] are also an alternative to subword for addressing the OoV problem. Recently, Cvitkovic et al. [52] extended the graph-based code representation [6] to incorporate open vocabulary using a Graph-Structured Cache, in which novel words/tokens would be added as *cached* [78] nodes into an existing AST.

Attention mechanism. Attention mechanism can be used for addressing both the OoV issue and the hidden-state bottleneck of RNNs. Bhoopchand et al. [24] employed a pointer network to copy OoV tokens from the recent past during code completion. The pointer network is a soft attention over previous input embeddings. A controller produces a scalar to decide whether to select from the copying position or language model distribution. Li et al. [150] recently proposed a similar model but with attention over the previous hidden states. Similar to attention layers of decoder networks, the attention output and the input are concatenated. The attention used in these models is computed with an MLP [16]. The drawback of this approach is that the modification of hidden-state cache and computation of attention are very computationally intensive. We have also observed that adding token copying can lead to a precision drop of token predictions comparing to no pointer network. Splitting the hidden states into separate parts for pointer network and context encoding solves this problem. Attention mechanism is also used by some non-recurrent models, e.g., Das and Shah [56] employed a gated unit over the word embeddings for a feed-forward neural network. This model was used to represent some commonly occurring value types such as iterator variable names for contextual code completion.

4.2 Deep Decoder Models

Given the embedded features produced by an encoder model, a decoder model generates output for a target domain (e.g., code and natural language). Unlike natural languages, source code must adhere to the target syntax of a programming language. Xu et al. [275] used a *sketch* for SQL generation template and trained neural networks to copy certain parts to these slots with a column attention mechanism. Furthermore, the decoding template can be enriched by introducing a DSL. By training a model with direct syntactic information such as ASTs, much better results can be obtained for code generation. Dong and Lapata [61] proposed the Seq2Tree model for transferring language into logical forms. A tree structure is decoded by predicting a "nonterminal" token as a root of a subtree. Parent information is fed by incorporating attention mechanism with all previous states. This model makes it easier to keep structural information, but this simple modification only supports limited grammar rules and has no guarantee of syntactic correctness.

Yin et al. [281] designed and implemented a syntax-driven neural model to transform natural language statements into corresponding Python ASTs. This model got 10+ BLEU and decent accuracy on several code generation tasks. Instead of directly generating source code c, a probabilistic grammar model g representing the distribution of an AST y given input natural language description x is defined so the syntax structure is automatically captured. The task is to select the best possible AST $\hat{y} = \arg\max_y g(y|x)$. Then given the AST, code can be inferred in a definite process.

A finite set of production rules $r \in R$ is the main component of a formal grammar specification. Using a depth-first traversal from left to right, the AST for these rules can be generated with a sequence of two types of actions a on the current AST y:

- Apply a specific production rule r;
- Add a terminal token v (e.g., \n).

Now, we can map the AST generation to a traditional seq2seq learning task. Given an input sequence $\mathbf{x}$, a sequence of action $\mathbf{a}$ can be generated as follows: $p(\mathbf{a}|\mathbf{x}) = \prod_{t=1}^{T} p(a_t|\mathbf{x}, a_{<t})$, where a_t is the action taken at time t.

In fact, generating highly accurate general-purpose programs is a very challenging task. The performance of program generation relies heavily on the development of code generative models, which needs to satisfy the following three major requirements:

- **Input representation** of a program should be able to handle OoV tokens, which usually requires learning embedding vector from both word-level and character-level tokens. For example, it is helpful to use compositional word representation such as C2W [157] and copying mechanism [257] to represent newly appeared OoV names in code. Syntactic information is also important, as mentioned in Section 4.1. To reconstruct grammatically correct programs from generation, encoders and decoders often accept and yield AST tokens.
- **Addressing mechanisms** (e.g., attention [16] and memory [79]) are important in guiding the decoding process and also used to copy from cache and recent history. However, content-based attention [16] and its extension, writable memory [79], are both not very suitable for implementing this copying mechanism. Another structure to keep recent history is associative memory [14]. The memory is kept in an associative matrix and updated at every time step. Some variants of associative memory are also incorporated with language models [55].
- **Discrete structure learning** is the key to representing function logic and control flows of a program. It is important for neural program models to learn a discrete structure and make deterministic decisions based on such structure.

We further discuss the solutions and challenges in designing these components in Section 7.

4.3 Deep Controller Models

Theoretically, a neural network itself is capable of learning a program [51, 239]. Therefore, to solve more complex problems, deep neural network can be used as a controller to learn the next instruction/operation to execute directly from the input-output examples without a DSL. This class of models is also referred to as *neural abstract machines*. Grefenstette et al. (2015) [81] connected an LSTM with soft-differentiable stack-, queue-, and deque-based memories to generalize the ability of RNN in machine translation. Neural Turing Machine (NTM) [79] and Differentiable Neural Computer (DNC) [80] use a recurrent model (RNN or LSTM) to read and write to an external memory matrix in a dynamic manner to simulate the execution of Turing computers. NTM uses a content-based soft attention to control reading/writing and enables access to every memory cell and gradient flow from these units. DNC defines a differentiable free list to track the usage of each memory location to address temporal orders of memory. Yang [278] generalized Turing machines to the continuous setting by storing memory on manifolds and controlled memory addressing using Lie group actions that are differentiable. Besides NTM and DNC, there are other neural abstract machines:

- Neural Programmer [190];
- Neural Programmer-Interpreter [224];
- Neural RAM [140];
- Neural Stack [118];
- Neural Program Lattices [149];
- Neural GPU [120].

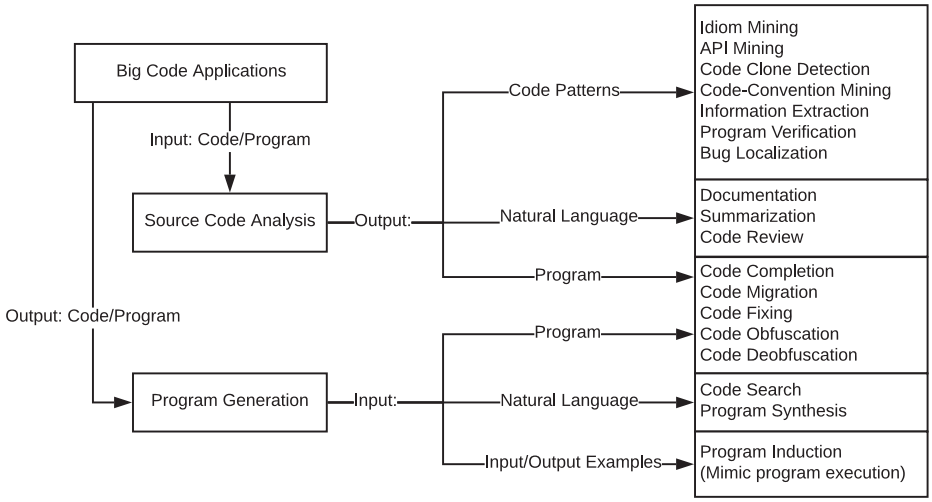

Fig. 2. A taxonomy of Big Code applications based on their inputs and outputs.

Neural abstract machine is a flexible and powerful class of DL models that is capable of inferring implicit graph structure, which can be used to simulate modern program executions. Instead of constructing an explicit program representation, they learn the model weights to describe the latent procedure and then simulate the dataflow of program execution. Although neural abstract machines behave similarly to programs and they are also flexible enough to emulate program execution, there is not yet a clear way of extracting program representations directly from these models. Besides program learning, this type of model has also been used in other domains such as question answering [80, 218], finding the shortest path [80], navigation in 3D simulation [205], querying a database for entity relations [80], and one-shot recognition [218].

5 DEEP LEARNING FOR BIG CODE APPLICATIONS

To review DL for Big Code applications, we categorize the input and output formats of different tasks into **source code analysis** and **program generation** as shown in Figure 2. For **source code analysis**, the input is source code and the output can take on many forms, such as natural language, code fragments/pattern, or a whole program. For **program generation**, all tasks have the same output in the format of source code, but different inputs (e.g., code, natural language, or input/output examples). The inputs of both source code analysis and program generation in our taxonomy are sequences, which can be modeled by DL models under encoder-decoder framework, as introduced in Sections 3 and 4. In addition, this taxonomy covers most of input and output types so any new Big Code tasks can be categorized easily. Then, suitable DL models can be selected accordingly to solve such tasks. DL for Big Code is growing very fast; thus, it is nearly impossible to include every single work for each presented application. Also, direct one-to-one performance comparison between the reviewed models of each application may not be possible due to different datasets used and/or lack of reported results in the original works. However, we still believe that this review is a good starting point for practitioners and researchers working in this emerging area.

5.1 Source Code Analysis

Source code analysis tasks take source code as context and generate outputs in another format. Source code analysis utilizes the distribution learned from large code corpus and performs various kinds of predictions. First, the case when the outputs are code patterns/elements is presented.

Idiom mining extracts the code segments that reappear across projects. Most common code idioms often describe important programming concepts and can be reused across projects [7, 8]. A related task to idiom mining is predicting class/method names based on their context/bodies [4], which can be generalized to source code classification. One of the first DL works on programming language processing was presented by Mou et al. [186]. This study proposed a tree-based CNN (TBCNN) with dynamic pooling learned directly from an AST of a program. The authors demonstrated that their learned feature vectors of program tokens with TBCNN could be grouped together in terms of functionality. Such representations were also demonstrated to be more effective than *n*-gram (Bag-of-words) methods for identifying programming tasks and detecting bubble sort patterns. Later, several studies utilizing different structural information of code (AST paths [11] or data-/control-flow information [6]) achieved strong performance for these tasks.

Application Programming Interface (API) mining and **Code clone detection** witness many uses of DL models. More particularly, DeepAPI [85] was devised to learn a distributed representation using a deep RNN seq2seq model for both user queries and associated APIs. This work was found to perform better than the bag-of-word approach for API generation task. After that, many DL works modeling ASTs of source code (e.g., RtNN [267], Tree-LSTM [263], and ASTNN [284]) also obtained high detection rates for code clones.

Code-convention mining finds the code practices recommended by a specific programming language but not enforced by compilers. These coding conventions (e.g., indentation, naming conventions, and white space) help improve the readability and maintainability of source code. A pioneering work using ML in this direction was proposed by Allamanis et al. [3] in 2014 using *n*-gram features combined with Support Vector Machine model. Nevertheless, after that, no other work has been reported to directly use DL for this code-convention mining. However, it is observed that recent DL models that effectively capture the usage contexts of source code (e.g., References [6, 52]) can be investigated for this task.

Information extraction aims to identify the existence of some code snippets, program or software-related artifacts from natural language [37, 237], images, or videos [212, 276]. With recent advances of DL models (e.g., CNN) in computer vision, more works [200, 201] are emerging in this direction to detect code/software elements from image and videos. Parallel to CNN, RNN has also been utilized to separate code fragments from natural language on Stack Overflow forum [280].

Program verification predicts whether there exist bugs or security issues in a program. **Bug localization** is closely related to program verification, except that besides code, the locations of specific types of bugs are also identified. Besides formal methods [166] and traditional ML approaches [38, 145, 146, 261], DL has been shown to perform quite well for this task. DeepBugs [214] represented more than 150K JavaScript source code files using word2vec and then trained a feed-forward neural network to distinguish between buggy and non-buggy code. This approach was found to achieve an accuracy of more than 90% and have the potential to perform in real-time. Another study [272] using a word-embedding CNN architecture for both bug reports and source code files outperformed the existing DL works [142, 273] for bug localization. Additionally, VulDeePecker [152] utilized a bidirectional LSTM (bi-LSTM) model with *code gadgets* as input to identify multiple types of vulnerabilities in source code. There is also a comprehensive review [74] on vulnerability analysis and discovery from source code using ML and DL techniques.

The reviewed DL models for this case have either used sequential and/or structural (i.e., ASTs) code representation. Most of the DL models were better than the non-DL ones, while the structural models seemed to have stronger performance than the sequential counterparts.

In other scenarios of source code analysis, the outputs can be natural language, which leads to the following tasks:

Documentation is an important artifact embedded in a program to help annotate the requirements, operation, and uses of source code as well as make software maintenance easier. Inspired by Neural Machine Translation [16], Barone et al. [19] utilized the same model to create the baseline for documentation generation. Later, Hu et al. [105] proposed an LSTM model with attention, DeepCom, to automatically generate the documentation for Java code. Recently, a novel model, code2seq [10], with an attentional decoder to select optimal compositional paths of an AST, has been proposed to represent code snippets. Code2seq has achieved better performance than DeepCom for code documentation.

Summarization is a sub-task of documentation, in which the main functionality of source code or a function is briefly described. In 2015, Rush et al. [229] proposed SUM-NN (i.e., a neural attention model) for code summarization and outperformed existing information retrieval method (i.e., minimizing cosine distance between code and the corresponding summary) and phrase-based systems [133]. However, this model tended to generate short descriptions. To overcome this issue, Iyer et al. [110] introduced an improved neural attention model, namely, CODE-NN, by replacing feed-forward neural network with LSTM for the decoder in a seq2seq model. Chen et al. [41] presented a bimodal using two Variational AutoEncoders (i.e., one for natural language and one for source code) to support code retrieval and summarization for C# and SQL languages. To leverage the knowledge of API for summarizing source code, Hu et al. [106] combined the representation of API sequences learned from related API summarization tasks with code sequences in a seq2seq model with attention. Like code documentation, the code2seq structural DL model [10] has recently outperformed other existing works for code summarization task.

Code review is an important step in software development, since it helps identify bad coding practices that may lead to software defects. However, this step is mostly carried out manually, which is time-consuming and prone to error. To automate this process, DeepCodeReviewer [88] aimed to find relevant reviews for the current code snippets/program. Specifically, four separate LSTM models with word2vec embeddings were trained for different parts of source code and reviews. Then, the results were combined using a feed-forward neural network to determine the relevance of the current review. We found that there is still not much DL work in this area, which can stimulate future efforts to improve the performance of deep code-review models.

We see many modern DL algorithms from neural language processing applied to these tasks, which demonstrates the similarity between these two fields. Some recent techniques such as structural embeddings and attention mechanisms (cf. Section 4.1) have also been integrated into DL models, which enables the models to learn from massive previous knowledge and to be more expressive to capture the underlying structure of the given data. The case when both input and output are code is discussed in the next section along with other *program generation* tasks.

5.2 Program Generation

Program generation tasks require the inference about the program code or code structure. We classify program generation applications into three categories based on their inputs: (*i*) unconditional program generation, (*ii*) program transduction, and (*iii*) multimodal program generation.

In **unconditional program generation**, the input consists of only code corpus. The goal is to generate the next most likely tokens or draw samples similar to the current input. Code completion/suggestion is a typical task in this category.

Code completion is a useful and common feature that many code editors offer. Although code completion tools incorporated into the Integrated Development Environments (IDEs) are mostly rule-based,[3] learning-based code completion has also been an active field of study. Completion

[3] https://www.jetbrains.com/help/idea/auto-completing-code.html.

Table 1. Deep Code Completion Models

Model	Code representation	Study
LSTM	word	Dam et al. 2016 [127]
	character	Karpathy 2015 [125]
	AST node	Liu et al. 2016 [159]
LSTM & Pointer Net	word	Bhoopchand et al. 2016 [24]
	AST node	Li et al. 2017 [150]
RNN	word	White et al. 2015 [268]
MLP with attention	word	Das et al. 2015 [56]

Notes: MLP: Multi-Layer Perceptron, RNN: Recurrent Neural Network, LSTM: Long Short-Term Memory.

algorithms have been developed for different languages with high accuracy and flexibility [25, 92, 223]. These algorithms help enhance IDEs with better code suggestion, API calls, or components based on the current context. *Program generative models* is an extreme case for completion where no input is given and tokens are sampled from the first one to the end as a Markov chain [164, 192, 195]. Table 1 presents several representative works for neural code completion. The deep code completion models have utilized the recent practices presented in Section 4.1 including (*i*) structural information (e.g., AST [150, 159]), (*ii*) open vocabulary (e.g., character-level features [125]), and (*iii*) attention mechanism (e.g., pointer network [24]). It is noted that these models can also be used as deep encoder models (cf. Section 4.1) for other program generation tasks.

In **program transduction**, the goal is to convert source code into another form of code. The following applications fall into this category:

Code migration helps developers to port projects in one language to another [2]. It is a common case to upgrade source code to a higher version of language or framework. Automatic transducers to update APIs and code structure are very useful for development and deployment. One such API migration from Java to C# was carried out using a seq2seq model, namely, DeepAM [86]. A similar motivation was proposed by Nguyen et al. [193] to migrate APIs from Java to C#, yet still preserving the semantic information by learning the word2vec embeddings of pair-wise APIs.

Code fixing or **Code repair** is the next step after program verification and bug localization, in which bugs need to be fixed. In 2016, sk_p model using LSTM-based seq2seq framework was proposed to correct seven Python assignments in Massive open online courses, namely, MITx, and this model achieved an average 29% of accuracy for error correcting. SynFix [23] and DeepFix [90] with LSTM and GRUs, respectively, were trained on student assignments to fix syntax errors, which obtained roughly a complete fix rate of 30%. Another work [231] trained a combined model of LSTM and *n*-gram on a large Java corpus from GitHub, which interestingly reported that 10-gram model slightly outperformed the LSTM counterpart for fixing syntax errors. This finding suggests that a more sophisticated (e.g., incorporating syntactic code structure) and better fine-tuned DL model can be utilized to further improve the result. There is also an extensive review on this topic [184].

Code obfuscation prevents unauthorized people from analyzing and stealing source code, and thus protects the intellectual properties. There are some off-the-shelf obfuscated code generators such as Allatori[4] and ProGuard.[5] Statistical ML language model was also used for this task [161], but DL models have not been much explored, except for some simple cases [163]. However, this task

[4]http://www.allatori.com/.
[5]https://www.guardsquare.com/en/products/proguard.

is a good fit for encoder-decoder framework, since the input is code sequence and the output is obfuscated text sequence of such input code. A different but related task, identifying obfuscated code, has witnessed more applications of DL models [240, 262]. The automatically generated features of such works can give some insights into designing effective (DL) methods for code obfuscation.

Code deobfuscation is opposite to obfuscation, in which deobfuscation recovers the original version of source code from the obfuscated one. There are deobfuscation tools for JavaScript [1, 221, 255] and Android apps.[6] A DL-based approach [20], namely, Context2Name, was proposed to recover natural variable names for minified code. It first used a deep sequential auto-encoder to extract embeddings for identifiers from their usage contexts. Such embeddings were then fed into an RNN to infer natural variable names. This approach was demonstrated to outperform the state-of-the-art JavaScript deobfuscation tools such as JSNice[7] and JSNaughty.[8] Further research on DL models for code deobfuscation can be done, since DL for image deblurring, demosaicing, and inpainting is being actively investigated [169].

As mentioned in Section 4.1, to get more accurate generation of programs, code completion usually takes AST node sequences as input. Attention and copying mechanism also improve the performance of code generative models. However, high-quality code transduction is hard to achieve by sentence-by-sentence transferring, since the differences in programming language properties and designs may require code structure to be changed. Back-translation and generative models can be helpful in these cases, which is similar to image transferring [288].

A more challenging category is **multimodal program generation**, where the input type is not restricted. Natural language (e.g., documentation and comments), GUI screenshots [21], and speech can all be used. Related applications are listed as follows:

Code search returns the best matching snippets of existing source code based on natural language queries. A log-bilinear neural language model was used to enable retrieving source code with natural language and vice versa [9]. As explained in Section 2.4, log-bilinear model alone, however, is unable to capture long code dependencies, which is essential for code search, especially for long code segments. Later, a deep code search model, CODEnn (i.e., bi-LSTM units combined with max-pooling layer), was proposed by Gu et al. [84]. This model was shown to outperform the previous state-of-the-art results of CodeHow [162] (an extended Boolean model) for code search.

Program synthesis extends code completion by generating code based on many forms of information such as natural language, images, and speech. Previously, applications were limited to DSL search [9]. With DL, general-purpose program synthesis for various tasks can be now tackled as shown in Table 2. As mentioned in Section 4.2, to generate syntactically correct programs, many studies (e.g., References [61, 204, 217, 281]) have proposed to utilize AST-based instead of sequential decoder. Such decoders predicted AST nodes sequentially using RNNs (e.g., LSTM), which could be computationally expensive. Later, Sun et al. [246] proposed a grammar-based structural CNN with attention mechanisms to replace RNN in the decoder for generating code from natural language description. The structural CNN-based decoder generated a grammar rule (sequence of tokens) per step instead of token-by-token, which makes the decoding process more compact and efficient. This approach was also shown to achieve the state-of-the-art results for Python code generation using the HeartStone benchmark dataset (cf. Section 6.1). Recently, Brockschmidt et al. [32] extended the graph-based code representation [6] to code generation by augmenting the AST with *inherited* and *synthesized* nodes to capture the attribute grammars [132] during the decoding process. The Graph2Graph model of this work generated more accurate C# code samples compared to

[6] http://apk-deguard.com/.
[7] http://www.jsnice.org/.
[8] http://jsnaughty.org/.

Table 2. Deep Program Synthesis Models

Model	Generating Target	Study
Seq2seq	Bash shell	Lin et al. 2017 [155]
	CSS and HTML	Beltramelli 2017 [21]
Seq2seq with grammar constraint	Simple C code	Amodio et al. 2017 [12]
Seq2AST	Domain-specific language	Dong and Lapata 2016 [61]
	Card-game code	Rabinovich et al. 2017 [217]
	General Python code	Yin and Neubig 2017 [281]
Seq2Set	SQL queries	Xu et al. 2017 [275]
Graph2Graph	General C# code	Brockschmidt et al. 2018 [32]
Pointer Network	Card-game code	Ling et al. 2016 [156]
STNs and MLPs	LaTeX graphs	Ellis et al. 2017 [63]
Reinforcement learning	SQL queries	Zhong et al. 2017 [286]

Notes: MLP: Multi-Layer Perceptron, LSTM: Long Short-Term Memory, STN: Spatial Transformer Network [113].

existing methods. With CNNs, graphics programs can be inferred from their output drawings. Ellis et al. [63] trained a hierarchical neural net to convert simple hand-drawings into a DSL, which is then converted into LaTeX code with a bias-optimal search algorithm [232]. Beltramelli et al. [21] used an encoder-decoder architecture to generate front-end code from GUI screenshots.

Program induction seeks to fit a given pair of input/output examples to mimic program execution, in which the execution correctness is more important than the readability of source code. Because of the complexity of the problem space, targeted programs are often limited to certain forms of DSL or only some simple problems. Balog et al. [18] used a seq2seq model, namely, DeepCoder, to predict the probable DSL functions required to map the given inputs to outputs, which reduced the search space and made the program induction process 10× faster than the corresponding search-based counterparts. Another class of models is *differentiable interpreters*, in which pre-defined grammars of a DSL are continuously parameterized with neural networks. Such parametrization allows a model to search the program for a given input-output pair more efficiently. Evans and Grefenstette [64] made inductive logic programming differentiable by using a neural network to perform inference given generated clauses, their weights, and valuation of the axions. The model is end-to-end trainable and generalizable well on small datasets. However, this method is very memory-intensive. Riedel et al. [226] proposed a differentiable interpreter for Forth programming language to use input-output examples to create a complete program. Differentiable approaches often perform worse than the search-based methods for low-level programming languages (e.g., Assembly) [72]. Also, differentiable interpreters are still limited to solving only simple problems (e.g., accessing, sorting, or copying array elements) [65]. Neural abstract machines (cf. Section 4.3) are also suitable for program induction. For example, by learning directly from recursions, Cai et al. [35] extended the ability of Neural Programmer-Interpreter [224] and provided a generalization proof about the overall system behavior. There is also a recent review [189] on program induction.

We observe that most recent works claiming the state-of-the-art results on various Big Code generation tasks have used some variants of RNN (e.g., LSTM/bi-LSTM/GRU with attention mechanism). Also, embeddings extracted from code tokens and ASTs are common choices for the input of these DL models. However, the proposed DL models have only been investigated for a few applications. There is still lack of extensive ablation studies to test the generalizability of these models for many different Big Code tasks. Next, in Section 6, various datasets are presented to facilitate the building of deep source code models.

6 DATASETS FOR BIG CODE APPLICATIONS

There is a great need of large datasets for DL in general and deep code models in particular to exhibit their power [75]. In this section, two types of corpora are presented for source code analysis and program generation (cf. Figure 2). This section is mainly devoted to highlight the potential, but not yet established development of datasets for Big Code tasks.

6.1 Datasets for Source Code Analysis

There is a growing curated list for various code analysis tasks and datasets.[9] Currently, there are many unlabeled and large-scale open-source code corpora such as SourceForge[10] and GitHub. Such datasets are widely used for building a probabilistic model on source code. Some large code corpora and their usefulness for various tasks including code completion and code pattern mining are presented hereafter. These datasets can be utilized for other source code analysis tasks as well.

- Karpathy et al.[11] used the Linux Kernel repository on GitHub[12] to demonstrate the expressive ability of RNNs. Later, this dataset has also been used to evaluate code completion tasks.
- Java projects on SourceForge and GitHub are utilized for various code mining tasks [192].
- Another large Java corpus containing all forked projects crawled from GitHub with more than 1B code tokens was also used to design new code complexity metrics and enhance the performance of code suggestion tasks [7].
- Instead of working directly on source code, code can be first converted to ASTs, upon which a code language model is built. More particularly, Bielik et al. [25] and Raychev et al. [220] have used 100,000+ JavaScript[13] and Python[14] programs, respectively, to build their language models for code completion tasks.

Although the aforementioned corpora are useful for building a robust probabilistic model on source code, there are two major problems with these unlabeled datasets. First, without further annotations, only unsupervised learning can be applied. Second, there is no official benchmark dedicated to these tasks; researchers often train their models on their own datasets, which leads to a challenge when comparing the performance between approaches.

There have been some efforts to create datasets with labels for bug identification and code clone detection tasks. For these tasks, issue tracking systems such as JIRA or BugZilla and version control systems like Git provide a huge amount of information about software development activities.

- Lamkanfi et al. [143] proposed a bug dataset that combined data spanning over the whole bug-triage cycle from both defect tracking systems JIRA and BugZilla into a structured format. Such dataset motivates not only bug analyses but also reproducibility and comparison of the bug detection models in Eclipse and Mozilla.
- Just et al. [119] prepared a bug dataset, namely, Defects4J,[15] which contains both buggy and fixed source code along with their commit messages for 395 bugs in six Java projects.

[9] http://learnbigcode.github.io/.
[10] https://sourceforge.net/.
[11] http://karpathy.github.io/2015/05/21/rnn-effectiveness/.
[12] https://github.com/torvalds/linux.
[13] http://www.srl.inf.ethz.ch/js150.php.
[14] http://www.srl.inf.ethz.ch/py150.
[15] https://github.com/rjust/defects4j.

- A large-scale and continuously growing dataset [252] of failed builds and successful fixes was automatically curated from GitHub and Travis-CI.[16]
- BigCloneBench [249] is a similar effort for code clone detection tasks. This dataset[17] consists of the clones detected from IJaDataset containing 25K open-source Java projects. BigCloneBench has been carefully verified by experts to provide a reliable evaluation benchmark for future research on detecting various types of code clones.

6.2 Datasets for Program Generation

For *code completion*, there have been several useful datasets mentioned in the previous section. For other program generation tasks, natural language annotation (or other forms of information) is also required as input. Existing works [9, 110, 158, 216] have shown that online forums such as Stack Overflow, IFTTT[18] can provide an abundant amount of code and related discussions. However, since these sources are user-generated, they are noisy and unaligned with any format. Other existing datasets are carefully annotated manually, but most of them are still limited in quantity. Some examples of such datasets for program synthesis are reviewed here.

- Project Euler[19] is an open platform containing over 600 programming and mathematical problems. New problems are constantly added every two weeks. The description, pseudocode, and some sample inputs and outputs for each problem are included. This dataset can serve as a great source for program synthesis. Although most of the problems have been solved, there is still no public repository that contains completely tested and well-documented code for those problems. Oda et al. [196] utilized this dataset to perform translation between Python and Japanese languages, but unfortunately, the detailed code was not disclosed.
- Another large-scale and perpetual dataset [33], namely, Blackbox, of code edits and IDE usages have been collected from BlueJ Java IDE. This dataset can be used for various program generation tasks such as code completion, code fixing, and program synthesis.
- Card2code[20] is another high-quality dataset for program generation [156]. The authors collected the card descriptions of two trading card games (i.e., Magic the Gathering and Hearthstone) and their corresponding open-source Python implementations to perform program generation. However, this dataset is too domain-specific, and thus limits its applicability.
- Later, Oda et al. [19] presented a parallel Django dataset to address the limitations of the previous datasets. This dataset includes all source code of Django web framework with line-by-line English annotation, including functions for a wide variety of real-world use cases such as I/O operations, string manipulation, and exception handling.
- Barone et al. [19] presented a large-scale dataset[21] containing two main corpora. The first corpus contains more than 150,370 triplets of Python function declarations, bodies, and their corresponding docstrings for code documentation generation. The second corpus consists of than 161,630 pairs of function declarations and their bodies to facilitate code generation.
- WikiSQL [286] is another crowd-sourced dataset[22] for developing natural language interfaces for relational databases from users' queries.

[16] https://travis-ci.org/.
[17] https://github.com/clonebench/BigCloneBench.
[18] https://ifttt.com.
[19] https://projecteuler.net/.
[20] https://github.com/deepmind/card2code.
[21] https://github.com/EdinburghNLP/code-docstring-corpus.
[22] https://github.com/salesforce/WikiSQL.

- Yin et al. [280] introduced a dataset using Stack Overflow posts, namely, CoNaLa, which maps natural language intents of developers and the corresponding code snippets.

Besides program synthesis, program induction is another (special) form of program generation, in which the program datasets can always be generative with predefined rules and languages such as arithmetic, lists, group-theory, and tree relations. It is noted that the dataset for program induction is still very limited. One of the few related works is the question dataset[23] collected by Rothe et al. [228]. The dataset contains 605 natural language questions querying about a board-game situation, from which the algorithm/instructions can be learned by a machine to recover the original input-output pairs.

7 CHALLENGES AND DIRECTIONS OF DEEP CODE MODELING AND GENERATION

Most source code models are limited in their applications, since they usually perform significantly worse on corpora different from the training set, especially for complex program generation problems. The models are usually constrained to solve very specific tasks and are not flexible enough to generate code distribution for general purposes. Moreover, a real-world programming problem is often specified in a multimodal way, i.e., described with natural language and demonstrated with example inputs and outputs in text, graphics, or action sequences. In this case, a model has to solve program synthesis and induction problems simultaneously. Although it is natural for a human to consider both aspects while programming, there is no existing study using examples and descriptions together to build a neural program model that generates a program non-existent in the database. Tackling a real programming challenge requires a combination of program synthesis and induction models and also depends on the development of representation learning and goal-oriented dialogue. However, a proper architecture for this task has yet to come. A model with such complex logic and power is hard to create/train because of the following challenges:

- Collecting labeled data for this complex task requires massive annotation work. Therefore, the data must be used effectively during the training process.
- DL models with differentiable variables are not very good at representing discrete features, which is crucial to learn the programming rules and control flows.
- A proper evaluation metric that can incorporate both semantic meaning and grammatical and execution correctness is important. However, there is little work in that direction.
- Deep source code models tend to be overly complex, since a large proportion of parameters are dedicated to solving problems in which real programmers have no difficulty. For example, extensive computation has to be done in memory networks to learn how to copy variable names from previous occurrences. Therefore, we should learn from human experience to devise ways to simplify such procedures.

7.1 Data Efficiency

Real-world programs have a wide range of functionalities, which are difficult to cover with sets of annotated examples. Program context also changes rapidly, e.g., the meaning of local variables changes between function scopes. Thus, we have to find ways to make the best use of data, which in turn asks for a good model and training method that can learn fast and generalize well.

7.1.1 Unsupervised Learning. To learn translation without access to a parallel corpus, Lample et al. [144] tried to circumvent such lack of parallel datasets in machine translation with adversarial training. They used Denoising Auto-encoder to learn sentence embedding and

[23] https://github.com/anselmrothe/question_dataset.

adversarial training [70] on the encoded domain to classify source and target embeddings. We have seen some early signs of using adversarial learning for program repair [93]. However, extraction of generic representations from unsupervised learning is still not the dominant approach [199].

Reinforcement Learning (RL) is also helpful when the labeled data are scarce. It has been adopted in image captioning to generate more natural photo descriptions. An interesting crossover field for RL and programming is natural language interfaces for system control [141, 209]. Automatic analysis of multimodal instructions has the potential to change the way how we develop and use software. Instead of using a rule-based control system, we can use ML and DL to train agents on examples or simulations. Guu et al. [91] trained an agent to parse natural instructions to generate code in a stack-based language in SCONE [61]. RL also enables intelligent systems to model continuous rewards or policies and interact with its environment to gain new knowledge, which enables grammar induction on even infinite search space.

The optimization target for an RL agent is very flexible. It can be integrated into the state-of-the-art conversation [287] and captioning models [225]. This learning scheme has the potential to be used in source code modeling to improve the generation quality or even guide a model to explore the context of the program. Misra et al. [179] designed a learning model by asking the system for image understanding. By swapping the question selection module with a test selection, an automatic testing framework can be built. A few investigations of RL with DL models have also been conducted for code summarization [260] and code fixing [89].

7.1.2 Weakly and Semi-supervised Training. The problem of utilizing partially labeled data is called semi-supervised learning. A very closely related concept is weakly supervised learning, which originally means using fewer training samples by self-training. Recently, this term often refers to the cases of using noisy or partially labeled data; for example, enhancing semantic segmentation with only image-level labeled samples [238]. A widely adopted semi-supervised learning architecture is student-teacher framework [219]. In the state-of-the-art method, Tarvainen et al. [250] constructed a stronger teacher model by taking the exponential moving average of the students. Such models can also be customized for program generation tasks when the labeled dataset is limited.

7.1.3 Active Learning. Another way of dealing with a large quantity of unlabeled data is active learning, which aims to design an oracle that lets the agent select which samples should be labeled [236]. Konyushkova and Raphael [136] demonstrated that their Learning Active Learning algorithm is capable of solving real-world problems in a wide range of domains. Active learning problem can also be considered as a special case of multi-armed bandit [71]. The restriction is that the labeling process cannot be abandoned completely. These active learning schemes can be applied in Big Code data collections.

7.2 Discrete and Symbolic Representations

Different from conventional DL models for sequence modeling, a program generation model should pay more attention to representing control flows and discrete rules.

7.2.1 Representation Learning. Many Big Code tasks require a rapid understanding of source code context and generalization to other source code projects given the limited training data. Attempts to extract structural information from programs date back to grammatical inference in the '90s. Grammatical inference is the process of inducing, learning, or inferring grammars [57], which generates Finite-State Automata (FSA) of various types. In this field, researchers try to associate complexity theory, formal logic, and discrete structures with learning. This field has been connected with many scientific disciplines, including bioinformatics, computational linguistics, and

lossless compression. Different from the statistical construction of language models [22, 77], where the probability distribution of transition states of a Markov model [68] is learned, grammar inference determines an explicit representation of a programming language. However, these methods have limited expressive ability and difficulty handling real-world continuous space.

RNNs are also considered powerful enough to learn complex representations. There are many FSA extraction algorithms from RNN, called *RNN Rule Extraction* (RNN-RE) based on searching and sampling [111]. Sodsong et al. [241] trained an RNN with program paths from OpenJDK to detect caller-sensitive method vulnerabilities [48] and showed that grammar could be learned with CrySSMEx [112]. But the generation process relied on very aggressive quantization, and the trained complex RNN structure was hard to understand. RNN-RE has also been criticized to be "Fool's gold" [135]. Recent deep code models [6, 52] have captured control-/data-flow information of code using graph-based neural networks; however, their usage contexts are still pre-defined and limited to statically typed programming languages.

Deep generative models such as Generative Adversarial Nets [76], Variational AutoEncoders (VAEs) [131], and Autoregressive models [197] have been widely used to learn various kinds of representations. Recent advances in generative modeling of images [76, 82, 130, 254], audio [170, 197], and videos [66, 123] have yielded impressive samples and applications [109, 126, 147]. Such generative models can be utilized to address the *representation learning* issues in program generation. Deep generative models can be used to learn the probabilistic distributions of unlabeled code corpus. Vector representations can then be extracted from the distributions of these models.

7.2.2 Discrete Representation and Addressing. For language and code, more concise and discrete representations are required to conduct complex reasoning or predictions. Discrete learning can also simulate computer operations and improve model interpretability [199]. Recent advances in discrete representation techniques are presented in this section for future adaptation to program modeling and generation.

Sometimes the representation has a desired explicit symbolic form. In this case, we can define the target of representation learning as predicting attributes to fill in the predefined structure. For example, Yang et al. [277] extracted knowledge tuples from question answering datasets with a model similar to *Neural Symbolic Machine* [153]. However, it is more common that the structure is complex and unknown, where the algorithm has to infer the structure by itself.

We have seen some exciting signs of progress in discrete representation in other domains. For example, learning discrete hidden latent vectors in generative models enables interesting applications such as speaker transferring and video prediction. Oord et al. [199] proposed VQ-VAE, which used vector quantization to prevent posterior collapse and model discrete representation. A strong autoregressive model is used as the decoder for unsupervised speaker conversion tasks. VQ-VAE gives a great example of how to represent discrete rules with only continuous models and some non-linearity. Also, this model is trained self-supervised and has many potential applications in source code modeling and generation.

Besides representation, deterministic programming logic can be simulated by discrete addressing in memory-augmented networks. Xu et al. 2015 [274] proposed stochastic hard attention for image captioning by sampling just a column. During training, one would sample a set of sequences and compute gradient with an estimation similar to REINFORCE [269] used for policy-gradient RL. This technique is also used for other memory-augmented models. Zaremba et al. [282] modified Neural Turing Machine to use discrete addressing and trained the model with REINFORCE. Bornschein et al. [30] proposed a memory-augmented generative model that used a variational approximation for discrete addressing. They combined discrete memory addressing

with continuous latent variables for generative few-shot learning. Discrete addressing has the advantage of definite logic representation; thus, it can model the logic of source code better.

7.2.3 Training with Discrete Variables. Training a discrete latent variable model turns out to be not easy. The gradient estimation based on random sampling such as REINFORCE always introduces high variance. In general, many approaches rely on control variates to reduce the variance.

There are several gradient estimators for discrete latent variable models, such as IWAE [34], NVIL [181], and RWS [29]. Tucker et al. [253] proposed a low-variance, unbiased gradient estimates for discrete variables using control variates and Gumbel-Softmax [114, 165].

7.3 Real-world Application and Evaluation

One major limitation of previous neural code completion models is that they did not provide a complete solution for real-world problems. More details are covered hereafter.

- The model is trained on a fixed vocabulary. Thus, to generate new OoV values, the model has to be retrained. In addition, word-level code completion can only predict the next token after the typing of a complete word.
- Incomplete code leads to ambiguous parsing results for languages with complex grammars rather than LL(1). Thus, the current state cannot be mapped to a definite training input and this can make the prediction inaccurate.

More practical and engineering-friendly frameworks for rapid new concept learning and code generation are required. The following techniques can be adopted to deal with these limitations:

- Open-vocabulary learning in Section 4.1 can address the OoV issue and partial-word code completion. There is a large vocabulary in code and much less regularity compared to natural language. However, code tokens still share similarities on the character and subword levels, since variable/function names can contain the same parts of existing words in different order.
- PCFG is a natural fit for ambiguous parsing, and it could be integrated into a system. The evaluation process should be changed accordingly, since the probabilistic parsing also introduces errors. To better account for such error in training, one can connect PCFG parameters to the learning objective and train the model end-to-end.

Furthermore, the evaluation methods being used for the code generation tasks are not perfect. AST node prediction accuracy is not a proper evaluation metric. Although syntactic information is better represented by AST node sequences, this is not the natural order of typing and the precision does not directly reflect the productivity gain of a tool. BLEU [203] scores are very sensitive to tokenization. To deal with this problem, there is a standard evaluation code for comparable BLEU scores.[24] In addition, ROUGE [154] and BLEU often do not capture linguistic fluency and coherence [160]. This problem gets even worse when evaluating code generations. Bielik et al. [25] used precision and log probability for their probabilistic models. Practically, under a code completion setting, developers would find a tool useful if some of the predicted top-k items hit [192].

A code-in-code-out system for generation and evaluation is more suitable for this situation. Even predicting AST sequences, code completion algorithms should include a converter between AST sequence and incomplete code. The evaluations then can be carried out by measuring the code-level accuracy. Ideally, execution correctness should be used for evaluating code generation, but it

[24] https://github.com/awslabs/sockeye/tree/master/contrib/sacrebleu.

is hard to quantify in most real cases. And the resulting error cannot be directly passed through gradient to the model parameters. Static code analysis and RL techniques can also be adopted.

7.4 Human-like Programming

Current neural program models are still quite distant from mimicking real programmers. Specifically, developers do not usually implement everything from scratch, but they rather try to adapt existing code to suit their needs. One way is to divide complicated tasks into simple ones for neural program models to speed up the training and improve performance. For example, Hashimoto et al. [94] proposed to solve code generation by retrieving related examples and applying modifications. Other ways to achieve human-like programming would be to use copying mechanism [257] to reuse code as well as utilize both description and examples for program generation.

7.4.1 Copying Mechanism. Training a model to attend to the right part of information could be tricky. Sometimes, attention tends to repeat itself, and thus generates long and meaningless sequences. To deal with this problem, we have to add a structural constraint on the attention mechanism. In most cases, an almost diagonal attention matrix is what we want. However, complex addressing mechanisms like external memory networks do not perform well on language modeling tasks. Also, these models usually cannot be efficiently trained. The complexity of program generation can be greatly reduced if human-like behavior is considered. See et al. [234] proposed a technique to copy words from input sequences to generate rare words. Such copying mechanism was adopted by a recent work [42] to better handle OoV code tokens and enhance the robustness of automated program repair.

7.4.2 Programming by Description and Examples. In many real-world scenarios such as competitive coding,[25] programmers write code by comprehending descriptions in natural language and then test their code with some input/output pairs. There are some search-based methods for program generation with both description and examples as context [211]. However, most models are not end-to-end trainable and they also have very limited capacity, which requires further research.

8 CONCLUSIONS

Artificial Intelligence (AI) in general and Deep Learning (DL) in particular have been increasingly leveraged for source code modeling. To facilitate more DL uses for practitioners and researchers in this area, our literature review first presented the limitations of existing source code models. We highlighted the potential of DL models to address these challenges and provided a more general solution using encoder-decoder framework for a wide range of problems. We then described important elements of such framework using DL models. We gave recommendations on applying encoder-decoder framework with DL models to source code modeling and generation. Various Big Code applications (i.e., source code analysis and program generation) following encoder-decoder framework were then presented. We also identified the gaps between the state-of-the-art DL models and their applicability in source code modeling and generation. For some of these gaps, we proposed corresponding suggestions on how to address them in future research.

We also want to provide some final thoughts about AI safety for source code modeling and generation. Recently, another emerging trend of research points out the vulnerability in the DL models that may result in bad consequences, namely, *adversarial DL*. For instance, an image can be perturbed to change the output of a DL model completely [185, 191], but a human cannot distinguish such a subtle change. Later, the idea of *adversarial DL* was also adapted to models used for sequence modeling (e.g., RNN) [202]. Such finding indicates that it is totally possible to fool

[25] https://code.google.com/codejam/.

source code models, e.g., turning a predicted vulnerable code snippet into a benign one. Therefore, besides aiming for high performance, practitioners and researchers in this area should also be aware of the robustness of a DL source code model to input data changes.

ACKNOWLEDGMENT

The authors would like to thank the editors and anonymous reviewers for their constructive feedback to improve our study.

REFERENCES

[1] Simon Aebersold, Krzysztof Kryszczuk, Sergio Paganoni, Bernhard Tellenbach, and Timothy Trowbridge. 2016. Detecting obfuscated JavaScripts using machine learning. In *Proceedings of the 11th International Conference on Internet Monitoring and Protection (ICIMP'16)*. IARIA.

[2] Karan Aggarwal, Mohammad Salameh, and Abram Hindle. 2015. *Using Machine Translation for Converting Python 2 to Python 3 Code*. Technical Report. PeerJ PrePrints. University of Alberta, Edmonton, Canada.

[3] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. 2014. Learning natural coding conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 281–293.

[4] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. 2015. Suggesting accurate method and class names. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*. ACM, 38–49.

[5] Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Comput. Surv.* 51, 4, Article 81 (July 2018).

[6] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to represent programs with graphs. In *Proceedings of the International Conference on Learning Representations (ICLR'18)*.

[7] Miltiadis Allamanis and Charles Sutton. 2013. Mining source code repositories at massive scale using language modeling. In *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Press, 207–216.

[8] Miltiadis Allamanis and Charles Sutton. 2014. Mining idioms from source code. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 472–483.

[9] Miltos Allamanis, Daniel Tarlow, Andrew Gordon, and Yi Wei. 2015. Bimodal modelling of source code and natural language. In *Proceedings of the 32nd International Conference on Machine Learning (ICML'15)*. 2123–2132.

[10] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2018. code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400* (2018).

[11] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *Proc. ACM Progr. Lang.* 3 (2019), 40.

[12] Matthew Amodio, Swarat Chaudhuri, and Thomas Reps. 2017. Neural attribute machines for program generation. *arXiv preprint arXiv:1705.09231* (2017).

[13] Martin Arjovsky, Amar Shah, and Yoshua Bengio. 2016. Unitary evolution recurrent neural networks. In *Proceedings of the International Conference on Machine Learning*. 1120–1128.

[14] Jimmy Ba, Geoffrey E. Hinton, Volodymyr Mnih, Joel Z. Leibo, and Catalin Ionescu. 2016. Using fast weights to attend to the recent past. In *Proceedings of the Conference on Advances in Neural Information Processing Systems*. 4331–4339.

[15] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. 2016. Layer normalization. *arXiv preprint arXiv:1607.06450* (2016).

[16] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473* (2014).

[17] David Balduzzi and Muhammad Ghifary. 2016. Strongly-typed recurrent neural networks. *arXiv preprint arXiv:1602.02218* (2016).

[18] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2016. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989* (2016).

[19] Antonio Valerio Miceli Barone and Rico Sennrich. 2017. A parallel corpus of Python functions and documentation strings for automated code documentation and code generation. *arXiv preprint arXiv:1707.02275* (2017).

[20] Rohan Bavishi, Michael Pradel, and Koushik Sen. 2018. Context2Name: A deep learning-based approach to infer natural variable names from usage contexts. *arXiv preprint arXiv:1809.05193* (2018).

[21] Tony Beltramelli. 2017. pix2code: Generating code from a graphical user interface screenshot. *arXiv preprint arXiv:1705.07962* (2017).

[22] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. 2003. A neural probabilistic language model. *J. Mach. Learn. Res.* 3, Feb. (2003), 1137–1155.

[23] Sahil Bhatia and Rishabh Singh. 2016. Automated correction for syntax errors in programming assignments using recurrent neural networks. *arXiv preprint arXiv:1603.06129* (2016).
[24] Avishkar Bhoopchand, Tim Rocktäschel, Earl Barr, and Sebastian Riedel. 2016. Learning Python code suggestion with a sparse pointer network. *arXiv preprint arXiv:1611.08307* (2016).
[25] Pavol Bielik, Veselin Raychev, and Martin Vechev. 2016. PHOG: Probabilistic model for code. In *Proceedings of the International Conference on Machine Learning*. 2933–2942.
[26] Pavol Bielik, Veselin Raychev, and Martin Vechev. 2016. Program synthesis for character level language modeling. In *Proceedings of the International Conference on Learning Representations (ICLR'16)*.
[27] Phil Blunsom, Edward Grefenstette, and Nal Kalchbrenner. 2014. A convolutional neural network for modelling sentences. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics*.
[28] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. 2016. Enriching word vectors with subword information. *arXiv preprint arXiv:1607.04606* (2016).
[29] Jörg Bornschein and Yoshua Bengio. 2014. Reweighted wake-sleep. *arXiv preprint arXiv:1406.2751* (2014).
[30] Jörg Bornschein, Andriy Mnih, Daniel Zoran, and Danilo Jimenez Rezende. 2017. Variational memory addressing in generative models. In *Proceedings of the Conference on Advances in Neural Information Processing Systems*. 3921–3930.
[31] James Bradbury, Stephen Merity, Caiming Xiong, and Richard Socher. 2016. Quasi-recurrent neural networks. *arXiv preprint arXiv:1611.01576* (2016).
[32] Marc Brockschmidt, Miltiadis Allamanis, Alexander L. Gaunt, and Oleksandr Polozov. 2018. Generative code modeling with graphs. *arXiv preprint arXiv:1805.08490* (2018).
[33] Neil Christopher Charles Brown, Michael Kölling, Davin McCall, and Ian Utting. 2014. Blackbox: A large scale repository of novice programmers' activity. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*. ACM, 223–228.
[34] Yuri Burda, Roger Grosse, and Ruslan Salakhutdinov. 2015. Importance weighted autoencoders. *arXiv preprint arXiv:1509.00519* (2015).
[35] Jonathon Cai, Richard Shin, and Dawn Song. 2017. Making neural programming architectures generalize via recursion. *arXiv preprint arXiv:1704.06611* (2017).
[36] Joshua Charles Campbell, Abram Hindle, and José Nelson Amaral. 2014. Syntax errors just aren't natural: Improving error reporting with language models. In *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 252–261.
[37] Luigi Cerulo, Michele Ceccarelli, Massimiliano Di Penta, and Gerardo Canfora. 2013. A hidden Markov model to detect coded information islands in free text. In *Proceedings of the IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM'13)*. IEEE, 157–166.
[38] Tantithamthavorn Chakkrit. 2016. *Towards a Better Understanding of the Impact of Experimental Components on Defect Prediction Models*. Ph.D. NARA Institute of Science and Technology.
[39] Chunyang Chen, Zhenchang Xing, and Lei Han. 2016. Techland: Assisting technology landscape inquiries with insights from stack overflow. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME'16)*. IEEE, 356–366.
[40] Chunyang Chen, Zhenchang Xing, and Ximing Wang. 2017. Unsupervised software-specific morphological forms inference from informal discussions. In *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 450–461.
[41] 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*. ACM, 826–831.
[42] Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2018. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *arXiv preprint arXiv:1901.01808* (2018).
[43] Zimin Chen and Martin Monperrus. 2019. A literature study of embeddings on source code. *arXiv preprint arXiv:1904.03061* (2019).
[44] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078* (2014).
[45] Jan K. Chorowski, Dzmitry Bahdanau, Dmitriy Serdyuk, Kyunghyun Cho, and Yoshua Bengio. 2015. Attention-based models for speech recognition. In *Proceedings of the Conference on Advances in Neural Information Processing Systems*. 577–585.
[46] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. 2014. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555* (2014).
[47] Junyoung Chung, Caglar Gulcehre, Kyunghyun Cho, and Yoshua Bengio. 2015. Gated feedback recurrent neural networks. In *Proceedings of the International Conference on Machine Learning*. 2067–2075.

[48] Cristina Cifuentes, Andrew Gross, and Nathan Keynes. 2015. Understanding caller-sensitive method vulnerabilities: A class of access control vulnerabilities in the Java platform. In *Proceedings of the 4th ACM SIGPLAN International Workshop on State of the Art in Program Analysis*. ACM, 7–12.
[49] Trevor Cohn, Phil Blunsom, and Sharon Goldwater. 2010. Inducing tree-substitution grammars. *J. Mach. Learn. Res.* 11, Nov. (2010), 3053–3096.
[50] Tim Cooijmans, Nicolas Ballas, César Laurent, Çağlar Gülçehre, and Aaron Courville. 2016. Recurrent batch normalization. *arXiv preprint arXiv:1603.09025* (2016).
[51] Balázs Csanád Csáji. 2001. *Approximation with Artificial Neural Networks*. Master's thesis. Faculty of Sciences, Etvs Lornd University, Hungary 24 (2001), 48.
[52] Milan Cvitkovic, Badal Singh, and Anima Anandkumar. 2018. Open vocabulary learning on source code with a graph-structured cache. *arXiv preprint arXiv:1810.08305* (2018).
[53] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V. Le, and Ruslan Salakhutdinov. 2019. Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv preprint arXiv:1901.02860* (2019).
[54] Hoa Khanh Dam, Truyen Tran, and Trang Pham. 2016. A deep language model for software code. *arXiv preprint arXiv:1608.02715* (2016).
[55] Rumen Dangovski, Li Jing, and Marin Soljacic. 2017. Rotational unit of memory. *arXiv preprint arXiv:1710.09537* (2017).
[56] Subhasis Das and Chinmayee Shah. 2015. *Contextual Code Completion Using Machine Learning*. Technical Report. Stanford University, CA, USA.
[57] Colin De la Higuera. 2010. *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press.
[58] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.
[59] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).
[60] Jesse Dodge, Andreea Gane, Xiang Zhang, Antoine Bordes, Sumit Chopra, Alexander Miller, Arthur Szlam, and Jason Weston. 2015. Evaluating prerequisite qualities for learning end-to-end dialog systems. *arXiv preprint arXiv:1511.06931* (2015).
[61] Li Dong and Mirella Lapata. 2016. Language to logical form with neural attention. *arXiv preprint arXiv:1601.01280* (2016).
[62] Li Dong, Nan Yang, Wenhui Wang, Furu Wei, Xiaodong Liu, Yu Wang, Jianfeng Gao, Ming Zhou, and Hsiao-Wuen Hon. 2019. Unified language model pre-training for natural language understanding and generation. In *Proceedings of the Conference on Advances in Neural Information Processing Systems*. 13042–13054.
[63] Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Joshua B. Tenenbaum. 2017. Learning to infer graphics programs from hand-drawn images. *arXiv preprint arXiv:1707.09627* (2017).
[64] Richard Evans and Edward Grefenstette. 2017. Learning explanatory rules from noisy data. *arXiv preprint arXiv:1711.04574* (2017).
[65] John K. Feser, Marc Brockschmidt, Alexander L. Gaunt, and Daniel Tarlow. 2016. Differentiable functional program interpreters. *arXiv preprint arXiv:1611.01988* (2016).
[66] Chelsea Finn, Ian Goodfellow, and Sergey Levine. 2016. Unsupervised learning for physical interaction through video prediction. In *Proceedings of the Conference on Advances in Neural Information Processing Systems*. 64–72.
[67] Philip Gage. 1994. A new algorithm for data compression. *C Users J.* 12, 2 (1994), 23–38.
[68] Paul A. Gagniuc. 2017. *Markov Chains: From Theory to Implementation and Experimentation*. John Wiley & Sons.
[69] Yarin Gal and Zoubin Ghahramani. 2016. A theoretically grounded application of dropout in recurrent neural networks. In *Proceedings of the Conference on Advances in Neural Information Processing Systems*. 1019–1027.
[70] Yaroslav Ganin, Evgeniya Ustinova, Hana Ajakan, Pascal Germain, Hugo Larochelle, François Laviolette, Mario Marchand, and Victor Lempitsky. 2016. Domain-adversarial training of neural networks. *J. Mach. Learn. Res.* 17, 59 (2016), 1–35.
[71] Ravi Ganti and Alexander G. Gray. 2013. Building bridges: Viewing active learning from the multi-armed bandit lens. *arXiv preprint arXiv:1309.6830* (2013).
[72] Alexander L. Gaunt, Marc Brockschmidt, Rishabh Singh, Nate Kushman, Pushmeet Kohli, Jonathan Taylor, and Daniel Tarlow. 2016. Terpret: A probabilistic programming language for program induction. *arXiv preprint arXiv:1608.04428* (2016).
[73] Jonas Gehring, Michael Auli, David Grangier, Denis Yarats, and Yann N. Dauphin. 2017. Convolutional sequence to sequence learning. *arXiv preprint arXiv:1705.03122* (2017).
[74] Seyed Mohammad Ghaffarian and Hamid Reza Shahriari. 2017. Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. *ACM Comput. Surv.* 50, 4 (2017), 56.

[75] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. 2016. *Deep Learning*. Vol. 1. The MIT Press, Cambridge, MA.
[76] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative adversarial nets. In *Proceedings of the Conference on Advances in Neural Information Processing Systems*. 2672–2680.
[77] Joshua T. Goodman. 2001. A bit of progress in language modeling. *Comput. Speech Lang.* 15, 4 (2001), 403–434.
[78] Edouard Grave, Moustapha M. Cisse, and Armand Joulin. 2017. Unbounded cache model for online language modeling with open vocabulary. In *Proceedings of the Conference on Advances in Neural Information Processing Systems*. 6042–6052.
[79] Alex Graves, Greg Wayne, and Ivo Danihelka. 2014. Neural turing machines. *arXiv preprint arXiv:1410.5401* (2014).
[80] Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gómez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, et al. 2016. Hybrid computing using a neural network with dynamic external memory. *Nature* 538, 7626 (2016), 471–476.
[81] Edward Grefenstette, Karl Moritz Hermann, Mustafa Suleyman, and Phil Blunsom. 2015. Learning to transduce with unbounded memory. In *Proceedings of the Conference on Advances in Neural Information Processing Systems*. 1828–1836.
[82] Karol Gregor, Frederic Besse, Danilo Jimenez Rezende, Ivo Danihelka, and Daan Wierstra. 2016. Towards conceptual compression. In *Proceedings of the Conference on Advances in Neural Information Processing Systems*. 3549–3557.
[83] Jiatao Gu, James Bradbury, Caiming Xiong, Victor O. K. Li, and Richard Socher. 2017. Non-autoregressive neural machine translation. *arXiv preprint arXiv:1711.02281* (2017).
[84] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 933–944.
[85] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API learning. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 631–642.
[86] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2017. DeepAM: Migrate APIs with multi-modal sequence to sequence learning. *arXiv preprint arXiv:1704.07734* (2017).
[87] Sumit Gulwani. 2010. Dimensions in program synthesis. In *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*. ACM, 13–24.
[88] Anshul Gupta and Neel Sundaresan. 2018. Intelligent code reviews using deep learning. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'18) Deep Learning Day*.
[89] Rahul Gupta, Aditya Kanade, and Shirish Shevade. 2018. Deep reinforcement learning for programming language correction. *arXiv preprint arXiv:1801.10467* (2018).
[90] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. DeepFix: Fixing common C language errors by deep learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*. 1345–1351.
[91] Kelvin Guu, Panupong Pasupat, Evan Zheran Liu, and Percy Liang. 2017. From language to programs: Bridging reinforcement learning and maximum marginal likelihood. *arXiv preprint arXiv:1704.07926* (2017).
[92] Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. 2013. Complete completion using types and weights. *ACM SIGPLAN Not.*, Vol. 48. ACM, 27–38.
[93] Jacob Harer, Onur Ozdemir, Tomo Lazovich, Christopher Reale, Rebecca Russell, Louis Kim, et al. 2018. Learning to repair software vulnerabilities with generative adversarial networks. In *Proceedings of the Conference on Advances in Neural Information Processing Systems*. 7933–7943.
[94] Tatsunori B. Hashimoto, Kelvin Guu, Yonatan Oren, and Percy S. Liang. 2018. A retrieve-and-edit framework for predicting structured outputs. In *Proceedings of the Conference on Advances in Neural Information Processing Systems*. 10073–10083.
[95] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 770–778.
[96] Luheng He, Kenton Lee, Mike Lewis, and Luke Zettlemoyer. 2017. Deep semantic role labeling: What works and what's next. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*.
[97] Zhen He, Shaobing Gao, Liang Xiao, and David Barber. 2017. Wider and deeper, cheaper and faster: Tensorized LSTMs for sequence learning. In *Proceedings of the Conference on Advances in Neural Information Processing Systems*. 1–11.
[98] Vincent J. Hellendoorn and Premkumar Devanbu. 2017. Are deep neural networks the best choice for modeling source code? In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*. ACM, 763–773.
[99] Mikael Henaff, Jason Weston, Arthur Szlam, Antoine Bordes, and Yann LeCun. 2016. Tracking the world state with recurrent entity networks. *arXiv preprint arXiv:1612.03969* (2016).

[100] Felix Hill, Antoine Bordes, Sumit Chopra, and Jason Weston. 2015. The Goldilocks principle: Reading children's books with explicit memory representations. *arXiv preprint arXiv:1511.02301* (2015).
[101] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*. IEEE, 837–847.
[102] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neur. Comput.* 9, 8 (1997), 1735–1780.
[103] John E. Hopcroft. 2008. *Introduction to Automata Theory, Languages, and Computation*. Pearson Education India. 77–106.
[104] Chun-Hung Hsiao, Michael Cafarella, and Satish Narayanasamy. 2014. Using web corpus statistics for program analysis. *ACM SIGPLAN Not.*, Vol. 49. ACM, 49–65.
[105] 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*. ACM, 200–210.
[106] Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. 2018. Summarizing source code with transferred API knowledge. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence (IJCAI'18)*. International Joint Conferences on Artificial Intelligence Organization, 2269–2275.
[107] Hakan Inan, Khashayar Khosravi, and Richard Socher. 2016. Tying word vectors and word classifiers: A loss framework for language modeling. *arXiv preprint arXiv:1611.01462* (2016).
[108] Sergey Ioffe and Christian Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the International Conference on Machine Learning*. 448–456.
[109] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A. Efros. 2016. Image-to-image translation with conditional adversarial networks. *arXiv preprint arXiv:1611.07004* (2016).
[110] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Vol. 1. 2073–2083.
[111] Henrik Jacobsson. 2005. Rule extraction from recurrent neural networks: A taxonomy and review. *Neur. Comput.* 17, 6 (2005), 1223–1263.
[112] Henrik Jacobsson. 2006. The crystallizing substochastic sequential machine extractor: CrySSMEx. *Neur. Comput.* 18, 9 (2006), 2211–2255.
[113] Max Jaderberg, Karen Simonyan, Andrew Zisserman, et al. 2015. Spatial transformer networks. In *Proceedings of the Conference on Advances in Neural Information Processing Systems*. 2017–2025.
[114] Eric Jang, Shixiang Gu, and Ben Poole. 2016. Categorical reparameterization with Gumbel-Softmax. *arXiv preprint arXiv:1611.01144* (2016).
[115] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, Vol. 1*. ACM, 215–224.
[116] Melvin Johnson, Mike Schuster, Quoc V. Le, Maxim Krikun, Yonghui Wu, Zhifeng Chen, Nikhil Thorat, Fernanda Viégas, Martin Wattenberg, Greg Corrado, et al. 2016. Google's multilingual neural machine translation system: Enabling zero-shot translation. *arXiv preprint arXiv:1611.04558* (2016).
[117] Aravind Joshi and Owen Rambow. 2003. A formalism for dependency grammar based on tree adjoining grammar. In *Proceedings of the Conference on Meaning-text Theory*. 207–216.
[118] Armand Joulin and Tomas Mikolov. 2015. Inferring algorithmic patterns with stack-augmented recurrent nets. In *Proceedings of the Conference on Advances in Neural Information Processing Systems*. 190–198.
[119] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the International Symposium on Software Testing and Analysis*. ACM, 437–440.
[120] Łukasz Kaiser and Ilya Sutskever. 2016. Neural GPUs learn algorithms. In *Proceedings of the International Conference on Learning Representations (ICLR'16)*.
[121] Nal Kalchbrenner and Phil Blunsom. 2013. Recurrent continuous translation models. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, Vol. 3. 413.
[122] Nal Kalchbrenner, Lasse Espeholt, Karen Simonyan, Aaron van den Oord, Alex Graves, and Koray Kavukcuoglu. 2016. Neural machine translation in linear time. *arXiv preprint arXiv:1610.10099* (2016).
[123] Nal Kalchbrenner, Aaron van den Oord, Karen Simonyan, Ivo Danihelka, Oriol Vinyals, Alex Graves, and Koray Kavukcuoglu. 2016. Video pixel networks. *arXiv preprint arXiv:1610.00527* (2016).
[124] Rafael-Michael Karampatsis and Charles Sutton. 2019. Maybe deep neural networks are the best choice for modeling source code. *arXiv preprint arXiv:1903.05734* (2019).
[125] Andrej Karpathy. 2016. The unreasonable effectiveness of recurrent neural networks. Retrieved from http://karpathy.github.io/2015/05/21/rnn-effectiveness (2016).

[126] Tero Karras, Timo Aila, Samuli Laine, and Jaakko Lehtinen. 2017. Progressive growing of GANs for improved quality, stability, and variation. *arXiv preprint arXiv:1710.10196* (2017).
[127] Hoa Khanh Dam, Truyen Tran, and Trang Pham. 2016. A deep language model for software code. *arXiv preprint arXiv:1608.02715* (2016).
[128] Yoon Kim, Yacine Jernite, David Sontag, and Alexander M. Rush. 2016. Character-aware neural language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*. 2741–2749.
[129] Diederik P. Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).
[130] Diederik P. Kingma, Tim Salimans, Rafal Jozefowicz, Xi Chen, Ilya Sutskever, and Max Welling. 2016. Improved variational inference with inverse autoregressive flow. In *Proceedings of the Conference on Advances in Neural Information Processing Systems*. 4743–4751.
[131] Diederik P. Kingma and Max Welling. 2013. Auto-encoding variational Bayes. *arXiv preprint arXiv:1312.6114* (2013).
[132] Donald E. Knuth. 1968. Semantics of context-free languages. *Math. Syst. Theor.* 2, 2 (1968), 127–145.
[133] Philipp Koehn, Hieu Hoang, Alexandra Birch, Chris Callison-Burch, Marcello Federico, Nicola Bertoldi, Brooke Cowan, Wade Shen, Christine Moran, Richard Zens, et al. 2007. Moses: Open source toolkit for statistical machine translation. In *Proceedings of the 45th Annual Meeting of the ACL on Interactive Poster and Demonstration Sessions*. Association for Computational Linguistics, 177–180.
[134] Philipp Koehn and Rebecca Knowles. 2017. Six challenges for neural machine translation. *arXiv preprint arXiv:1706.03872* (2017).
[135] John F. Kolen. 1994. Fool's gold: Extracting finite state machines from recurrent network dynamics. In *Proceedings of the Conference on Advances in Neural Information Processing Systems*. 501–508.
[136] Ksenia Konyushkova, Raphael Sznitman, and Pascal Fua. 2017. Learning active learning from data. In *Proceedings of the Conference on Advances in Neural Information Processing Systems*. 4226–4236.
[137] Jan Koutnik, Klaus Greff, Faustino Gomez, and Juergen Schmidhuber. 2014. A clockwork RNN. In *Proceedings of the International Conference on Machine Learning*. 1863–1871.
[138] Ben Krause, Emmanuel Kahembwe, Iain Murray, and Steve Renals. 2017. Dynamic evaluation of neural sequence models. *arXiv preprint arXiv:1709.07432* (2017).
[139] David Krueger, Tegan Maharaj, János Kramár, Mohammad Pezeshki, Nicolas Ballas, Nan Rosemary Ke, Anirudh Goyal, Yoshua Bengio, Aaron Courville, and Chris Pal. 2016. Zoneout: Regularizing RNNs by randomly preserving hidden activations. *arXiv preprint arXiv:1606.01305* (2016).
[140] Karol Kurach, Marcin Andrychowicz, and Ilya Sutskever. 2015. Neural random-access machines. *arXiv preprint arXiv:1511.06392* (2015).
[141] Thomas Laengle, Tim C. Lueth, Eva Stopp, Gerd Herzog, and Gjertrud Kamstrup. 1995. KANTRA-A natural language interface for intelligent robots. In *Proceedings of the Conference on Intelligent Autonomous Systems (IAS'95)*. 357–364.
[142] An Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. 2017. Bug localization with combination of deep learning and information retrieval. In *Proceedings of the IEEE/ACM 25th International Conference on Program Comprehension (ICPC'17)*. IEEE, 218–229.
[143] Ahmed Lamkanfi, Javier Pérez, and Serge Demeyer. 2013. The Eclipse and Mozilla defect tracking dataset: A genuine dataset for mining bug information. In *Proceedings of the 10th IEEE Working Conference on Mining Software Repositories (MSR'13)*. IEEE, 203–206.
[144] Guillaume Lample, Ludovic Denoyer, and Marc'Aurelio Ranzato. 2017. Unsupervised machine translation using monolingual corpora only. *arXiv preprint arXiv:1711.00043* (2017).
[145] Tien-Duy B. Le, Richard J. Oentaryo, and David Lo. 2015. Information retrieval and spectrum based bug localization: Better together. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*. ACM, 579–590.
[146] Triet Huynh Minh Le, Bushra Sabir, and Muhammad Ali Babar. 2019. Automated software vulnerability assessment with concept drift. In *Proceedings of the IEEE/ACM 16th International Conference on Mining Software Repositories (MSR'19)*. IEEE, 371–382.
[147] Christian Ledig, Lucas Theis, Ferenc Huszár, Jose Caballero, Andrew Cunningham, Alejandro Acosta, Andrew Aitken, Alykhan Tejani, Johannes Totz, Zehan Wang, et al. 2016. Photo-realistic single image super-resolution using a generative adversarial network. *arXiv preprint arXiv:1609.04802* (2016).
[148] Tao Lei and Yu Zhang. 2017. Training RNNs as fast as CNNs. *arXiv preprint arXiv:1709.02755* (2017).
[149] Chengtao Li, Daniel Tarlow, Alexander L. Gaunt, Marc Brockschmidt, and Nate Kushman. 2017. Neural program lattices. In *Proceedings of the International Conference on Learning Representations (ICLR'17)*.
[150] Jian Li, Yue Wang, Irwin King, and Michael R. Lyu. 2017. Code completion with neural attention and pointer networks. *arXiv preprint arXiv:1711.09573* (2017).
[151] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. 2015. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493* (2015).

[152] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. VulDeePecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681* (2018).

[153] Chen Liang, Jonathan Berant, Quoc Le, Kenneth D. Forbus, and Ni Lao. 2016. Neural symbolic machines: Learning semantic parsers on freebase with weak supervision. *arXiv preprint arXiv:1611.00020* (2016).

[154] Chin-Yew Lin. 2004. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches out: Proceedings of the ACL-04 Workshop*, Vol. 8.

[155] Xi Victoria Lin, Chenglong Wang, Deric Pang, Kevin Vu, and Michael D. Ernst. 2017. *Program Synthesis from Natural Language Using Recurrent Neural Networks*. Technical Report UW-CSE-17-03-01, University of Washington Department of Computer Science and Engineering, Seattle, WA.

[156] Wang Ling, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiskỳ, Andrew Senior, Fumin Wang, and Phil Blunsom. 2016. Latent predictor networks for code generation. *arXiv preprint arXiv:1603.06744* (2016).

[157] Wang Ling, Tiago Luís, Luís Marujo, Ramón Fernandez Astudillo, Silvio Amir, Chris Dyer, Alan W. Black, and Isabel Trancoso. 2015. Finding function in form: Compositional character models for open vocabulary word representation. *arXiv preprint arXiv:1508.02096* (2015).

[158] Chang Liu, Xinyun Chen, Eui Chul Shin, Mingcheng Chen, and Dawn Song. 2016. Latent attention for if-then program synthesis. In *Proceedings of the Conference on Advances in Neural Information Processing Systems*. 4574–4582.

[159] Chang Liu, Xin Wang, Richard Shin, Joseph E. Gonzalez, and Dawn Song. 2016. Neural code completion. *OpenReview*. Retrieved from: https://openreview.net/pdf?id=rJbPBt9lg.

[160] Chia-Wei Liu, Ryan Lowe, Iulian V. Serban, Michael Noseworthy, Laurent Charlin, and Joelle Pineau. 2016. How NOT to evaluate your dialogue system: An empirical study of unsupervised evaluation metrics for dialogue response generation. *arXiv preprint arXiv:1603.08023* (2016).

[161] Han Liu. 2016. Towards better program obfuscation: Optimization via language models. In *Proceedings of the 38th International Conference on Software Engineering Companion*. ACM, 680–682.

[162] Fei Lv, Hongyu Zhang, Jian-guang Lou, Shaowei Wang, Dongmei Zhang, and Jianjun Zhao. 2015. Codehow: Effective code search based on API understanding and extended Boolean model (E). In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE'15)*. IEEE, 260–270.

[163] Haoyu Ma, Xinjie Ma, Weijie Liu, Zhipeng Huang, Debin Gao, and Chunfu Jia. 2014. Control flow obfuscation using neural network to fight concolic testing. In *Proceedings of the International Conference on Security and Privacy in Communication Networks (SecureComm'14)*.

[164] Chris Maddison and Daniel Tarlow. 2014. Structured generative models of natural source code. In *Proceedings of the 31st International Conference on Machine Learning (ICML'14)*. 649–657.

[165] Chris J. Maddison, Andriy Mnih, and Yee Whye Teh. 2016. The concrete distribution: A continuous relaxation of discrete random variables. *arXiv preprint arXiv:1611.00712* (2016).

[166] Magnus Madsen, Ondřej Lhoták, and Frank Tip. 2017. A model for reasoning about JavaScript promises. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 86 (Oct. 2017), 24 pages.

[167] Stephan Mandt, Matthew D. Hoffman, and David M. Blei. 2017. Stochastic gradient descent as approximate Bayesian inference. *arXiv preprint arXiv:1704.04289* (2017).

[168] Bryan McCann, James Bradbury, Caiming Xiong, and Richard Socher. 2017. Learned in translation: Contextualized word vectors. *arXiv preprint arXiv:1708.00107* (2017).

[169] Richard McPherson, Reza Shokri, and Vitaly Shmatikov. 2016. Defeating image obfuscation with deep learning. *arXiv preprint arXiv:1609.00408* (2016).

[170] Soroush Mehri, Kundan Kumar, Ishaan Gulrajani, Rithesh Kumar, Shubham Jain, Jose Sotelo, Aaron Courville, and Yoshua Bengio. 2016. SampleRNN: An unconditional end-to-end neural audio generation model. *arXiv preprint arXiv:1612.07837* (2016).

[171] Gábor Melis, Chris Dyer, and Phil Blunsom. 2017. On the state of the art of evaluation in neural language models. *arXiv preprint arXiv:1707.05589* (2017).

[172] Stephen Merity, Nitish Shirish Keskar, and Richard Socher. 2017. Regularizing and optimizing LSTM language models. *arXiv preprint arXiv:1708.02182* (2017).

[173] Stephen Merity, Bryan McCann, and Richard Socher. 2017. Revisiting activation regularization for language RNNs. *arXiv preprint arXiv:1708.01009* (2017).

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

[175] Tomas Mikolov, Armand Joulin, and Marco Baroni. 2015. A roadmap towards machine intelligence. *arXiv preprint arXiv:1511.08130* (2015).

[176] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S. Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Proceedings of the Conference on Advances in Neural Information Processing Systems*. 3111–3119.

[177] Alexander Miller, Adam Fisch, Jesse Dodge, Amir-Hossein Karimi, Antoine Bordes, and Jason Weston. 2016. Key-value memory networks for directly reading documents. *arXiv preprint arXiv:1606.03126* (2016).

[178] Nikhil Mishra, Mostafa Rohaninejad, Xi Chen, and Pieter Abbeel. 2017. Meta-learning with temporal convolutions. *arXiv preprint arXiv:1707.03141* (2017).

[179] Ishan Misra, Ross Girshick, Rob Fergus, Martial Hebert, Abhinav Gupta, and Laurens van der Maaten. 2017. Learning by asking questions. *arXiv preprint arXiv:1712.01238* (2017).

[180] Takeru Miyato, Toshiki Kataoka, Masanori Koyama, and Yuichi Yoshida. 2018. Spectral normalization for generative adversarial networks. *arXiv preprint arXiv:1802.05957* (2018).

[181] Andriy Mnih and Karol Gregor. 2014. Neural variational inference and learning in belief networks. *arXiv preprint arXiv:1402.0030* (2014).

[182] Andriy Mnih and Geoffrey Hinton. 2007. Three new graphical models for statistical language modelling. In *Proceedings of the 24th International Conference on Machine Learning*. ACM, 641–648.

[183] Andriy Mnih and Geoffrey E. Hinton. 2009. A scalable hierarchical distributed language model. In *Proceedings of the Conference on Advances in Neural Information Processing Systems*. 1081–1088.

[184] Martin Monperrus. 2018. Automatic software repair: A bibliography. *ACM Comput. Surv.* 51, 1 (2018), 17.

[185] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. 2016. DeepFool: A simple and accurate method to fool deep neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2574–2582.

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

[187] Stephen Muggleton and Luc De Raedt. 1994. Inductive logic programming: Theory and methods. *J Logic Progr* 19 (1994), 629–679.

[188] Asier Mujika, Florian Meier, and Angelika Steger. 2017. Fast-slow recurrent neural networks. *arXiv preprint arXiv:1705.08639* (2017).

[189] Kant Neel. 2018. Recent advances in neural program synthesis. *arXiv preprint arXiv:1802.02353* (2018).

[190] Arvind Neelakantan, Quoc V. Le, and Ilya Sutskever. 2015. Neural programmer: Inducing latent programs with gradient descent. *arXiv preprint arXiv:1511.04834* (2015).

[191] Anh Nguyen, Jason Yosinski, and Jeff Clune. 2015. Deep neural networks are easily fooled: High confidence predictions for unrecognizable images. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 427–436.

[192] Anh Tuan Nguyen and Tien N. Nguyen. 2015. Graph-based statistical language model for code. In *Proceedings of the IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE'15)*, Vol. 1. IEEE, 858–868.

[193] Trong Duc Nguyen, Anh Tuan Nguyen, Hung Dang Phan, and Tien N. Nguyen. 2017. Exploring API embedding for API usages and applications. In *Proceedings of the IEEE/ACM 39th International Conference on Software Engineering (ICSE'17)*. IEEE, 438–449.

[194] Tung Thanh Nguyen, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. 2013. A statistical semantic language model for source code. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*. ACM, New York, NY, 532–542.

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

[196] Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. 2015. Learning to generate pseudo-code from source code using statistical machine translation (T). In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE'15)*. IEEE, 574–584.

[197] Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. 2016. WaveNet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499* (2016).

[198] Aaron van den Oord, Yazhe Li, Igor Babuschkin, Karen Simonyan, Oriol Vinyals, and Koray Kavukcuoglu. 2017. Parallel WaveNet: Fast high-fidelity speech synthesis. *arXiv preprint arXiv:1711.10433* (2017).

[199] Aaron van den Oord, Oriol Vinyals, and Koray Kavukcuoglu. 2017. Neural discrete representation learning. *arXiv preprint arXiv:1711.00937* (2017).

[200] Jordan Ott, Abigail Atchison, Paul Harnack, Adrienne Bergh, and Erik Linstead. 2018. A deep learning approach to identifying source code in images and video. In *Proceedings of the 15th International Conference on Mining Software Repositories (MSR'18)*. ACM, New York, NY, 376–386.

[201] Jordan Ott, Abigail Atchison, Paul Harnack, Natalie Best, Haley Anderson, Cristiano Firmani, and Erik Linstead. 2018. Learning lexical features of programming languages from imagery using convolutional neural networks. In *Proceedings of the 26th Conference on Program Comprehension (ICPC'18)*. ACM, New York, NY, 336–339.

[202] Nicolas Papernot, Patrick McDaniel, Ananthram Swami, and Richard Harang. 2016. Crafting adversarial input sequences for recurrent neural networks. In *Proceedings of the Military Communications Conference (MILCOM'16)*. IEEE, 49–54.

[203] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: A method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*. Association for Computational Linguistics, 311–318.

[204] Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. 2016. Neuro-symbolic program synthesis. *arXiv preprint arXiv:1611.01855* (2016).

[205] Emilio Parisotto and Ruslan Salakhutdinov. 2017. Neural map: Structured memory for deep reinforcement learning. *arXiv preprint arXiv:1702.08360* (2017).

[206] Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, and Yoshua Bengio. 2013. How to construct deep recurrent neural networks. *arXiv preprint arXiv:1312.6026* (2013).

[207] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. 2013. On the difficulty of training recurrent neural networks. In *Proceedings of the International Conference on Machine Learning*. 1310–1318.

[208] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. GloVe: Global vectors for word representation. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP'14)*. 1532–1543.

[209] Dennis Perzanowski, Alan C. Schultz, William Adams, Elaine Marsh, and Magda Bugajska. 2001. Building a multimodal human-robot interface. *IEEE Intell. Syst.* 16, 1 (2001), 16–21.

[210] Matthew E. Peters, Waleed Ammar, Chandra Bhagavatula, and Russell Power. 2017. Semi-supervised sequence tagging with bidirectional language models. *arXiv preprint arXiv:1705.00108* (2017).

[211] Illia Polosukhin and Alexander Skidanov. 2018. Neural program search: Solving programming tasks from description and examples. *arXiv preprint arXiv:1802.04335* (2018).

[212] Luca Ponzanelli, Gabriele Bavota, Andrea Mocci, Massimiliano Di Penta, Rocco Oliveto, Barbara Russo, Sonia Haiduc, and Michele Lanza. 2016. CodeTube: Extracting relevant fragments from software development video tutorials. In *Proceedings of the 38th International Conference on Software Engineering Companion*. ACM, 645–648.

[213] Samira Pouyanfar, Saad Sadiq, Yilin Yan, Haiman Tian, Yudong Tao, Maria Presa Reyes, Mei-Ling Shyu, Shu-Ching Chen, and S. S. Iyengar. 2018. A survey on deep learning: Algorithms, techniques, and applications. *ACM Comput. Surv.* 51, 5 (2018), 92.

[214] Michael Pradel and Koushik Sen. 2018. DeepBugs: A learning approach to name-based bug detection. *arXiv preprint arXiv:1805.11683* (2018).

[215] Yewen Pu, Zachery Miranda, Armando Solar-Lezama, and Leslie Pack Kaelbling. 2017. Learning to select examples for program synthesis. *arXiv preprint arXiv:1711.03243* (2017).

[216] Chris Quirk, Raymond Mooney, and Michel Galley. 2015. Language to code: Learning semantic parsers for if-this-then-that recipes. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, Vol. 1. 878–888.

[217] Maxim Rabinovich, Mitchell Stern, and Dan Klein. 2017. Abstract syntax networks for code generation and semantic parsing. *arXiv preprint arXiv:1704.07535* (2017).

[218] Jack Rae, Jonathan J. Hunt, Ivo Danihelka, Timothy Harley, Andrew W. Senior, Gregory Wayne, Alex Graves, and Tim Lillicrap. 2016. Scaling memory-augmented neural networks with sparse reads and writes. In *Proceedings of the Conference on Advances in Neural Information Processing Systems*. 3621–3629.

[219] Antti Rasmus, Mathias Berglund, Mikko Honkala, Harri Valpola, and Tapani Raiko. 2015. Semi-supervised learning with ladder networks. In *Proceedings of the Conference on Advances in Neural Information Processing Systems*. 3546–3554.

[220] Veselin Raychev, Pavol Bielik, and Martin Vechev. 2016. Probabilistic model for code with decision trees. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 731–747.

[221] Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting program properties from big code. *ACM SIGPLAN Not.*, Vol. 50. ACM, 111–124.

[222] 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 (PLDI'14)*. ACM, New York, NY, 419–428.

[223] Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code completion with statistical language models. *ACM SIGPLAN Not.*, Vol. 49. ACM, 419–428.

[224] Scott Reed and Nando De Freitas. 2015. Neural programmer-interpreters. *arXiv preprint arXiv:1511.06279* (2015).

[225] Zhou Ren, Xiaoyu Wang, Ning Zhang, Xutao Lv, and Li-Jia Li. 2017. Deep reinforcement learning-based image captioning with embedding reward. *arXiv preprint arXiv:1704.03899* (2017).

[226] Sebastian Riedel, Matko Bosnjak, and Tim Rocktäschel. 2016. Programming with a differentiable forth interpreter. *arXiv preprint arXiv:1605.06640* (2016).

[227] Frank Rosenblatt. 1958. The perceptron: A probabilistic model for information storage and organization in the brain.*Psychol. Rev.* 65, 6 (1958), 386.

[228] Anselm Rothe, Brenden M. Lake, and Todd Gureckis. 2017. Question asking as program generation. In *Proceedings of the Conference on Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.). Curran Associates, Inc., 1046–1055.

[229] Alexander M. Rush, Sumit Chopra, and Jason Weston. 2015. A neural attention model for abstractive sentence summarization. *arXiv preprint arXiv:1509.00685* (2015).

[230] Tim Salimans and Diederik P. Kingma. 2016. Weight normalization: A simple reparameterization to accelerate training of deep neural networks. In *Proceedings of the Conference on Advances in Neural Information Processing Systems*. 901–909.

[231] Eddie Antonio Santos, Joshua Charles Campbell, Dhvani Patel, Abram Hindle, and José Nelson Amaral. 2018. Syntax and sensibility: Using language models to detect and correct syntax errors. In *Proceedings of the IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER'18)*. IEEE, 311–322.

[232] Jürgen Schmidhuber. 2004. Optimal ordered problem solver. *Mach. Learn.* 54, 3 (2004), 211–254.

[233] Holger Schwenk and Jean-Luc Gauvain. 2002. Connectionist language modeling for large vocabulary continuous speech recognition. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP'02)*, Vol. 1. IEEE, I–765.

[234] Abigail See, Peter J. Liu, and Christopher D. Manning. 2017. Get to the point: Summarization with pointer-generator networks. *arXiv preprint arXiv:1704.04368* (2017).

[235] Stanislau Semeniuta, Aliaksei Severyn, and Erhardt Barth. 2016. Recurrent dropout without memory loss. *arXiv preprint arXiv:1603.05118* (2016).

[236] Burr Settles. 2010. Active Learning Literature Survey. Technical Report 1648. University of Wisconsin, Madison 52, 55–66 (2010), 11.

[237] Abhishek Sharma, Yuan Tian, and David Lo. 2015. Nirmal: Automatic identification of software relevant tweets leveraging language model. In *Proceedings of the IEEE 22nd International Conference on Software Analysis, Evolution and Reengineering (SANER'15)*. IEEE, 449–458.

[238] Tong Shen, Guosheng Lin, Lingqiao Liu, Chunhua Shen, and Ian Reid. 2017. Weakly supervised semantic segmentation based on web image co-segmentation. *arXiv preprint arXiv:1705.09052* (2017).

[239] Hava T. Siegelmann and Eduardo D. Sontag. 1992. On the computational power of neural nets. In *Proceedings of the 5th Annual Workshop on Computational Learning Theory*. ACM, 440–449.

[240] Philippe Skolka, Cristian-Alexandru Staicu, and Michael Pradel. 2019. Anything to hide? Studying minified and obfuscated code in the web. In *Proceedings of the World Wide Web Conference*. ACM, 1735–1746.

[241] Wasuwee Sodsong, Bernhard Scholz, and Sanjay Chawla. 2017. SPARK: Static program analysis reasoning and retrieving knowledge. *arXiv preprint arXiv:1711.01024* (2017).

[242] Armando Solar-Lezama. 2013. Program sketching. *Int. J. Softw. Tools Technol. Transf.* 15, 5–6 (2013), 475–495.

[243] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.* 15, 1 (2014), 1929–1958.

[244] Rupesh Kumar Srivastava, Klaus Greff, and Jürgen Schmidhuber. 2015. Highway networks. *arXiv preprint arXiv:1505.00387* (2015).

[245] Sainbayar Sukhbaatar, Jason Weston, Rob Fergus, et al. 2015. End-to-end memory networks. In *Proceedings of the Conference on Advances in Neural Information Processing Systems*. 2440–2448.

[246] Zeyu Sun, Qihao Zhu, Lili Mou, Yingfei Xiong, Ge Li, and Lu Zhang. 2019. A grammar-based structural CNN decoder for code generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 7055–7062.

[247] Ilya Sutskever, Geoffrey E. Hinton, and Graham W. Taylor. 2009. The recurrent temporal restricted Boltzmann machine. In *Proceedings of the Conference on Advances in Neural Information Processing Systems*. 1601–1608.

[248] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to sequence learning with neural networks. In *Proceedings of the Conference on Advances in Neural Information Processing Systems*. 3104–3112.

[249] Jeffrey Svajlenko, Judith F. Islam, Iman Keivanloo, Chanchal K. Roy, and Mohammad Mamun Mia. 2014. Towards a big data curated benchmark of inter-project code clones. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME'14)*. IEEE, 476–480.

[250] Antti Tarvainen and Harri Valpola. 2017. Mean teachers are better role models: Weight-averaged consistency targets improve semi-supervised deep learning results. In *Proceedings of the Conference on Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.). Curran Associates, Inc., 1195–1204.

[251] Tijmen Tieleman and Geoffrey Hinton. 2012. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neur. Netw. Mach. Learn.* 4, 2 (2012), 26–31.

[252] David A. Tomassi, Naji Dmeiri, Yichen Wang, Antara Bhowmick, Yen-Chuan Liu, Premkumar T. Devanbu, Bogdan Vasilescu, and Cindy Rubio-González. 2019. BugSwarm: Mining and continuously growing a dataset of reproducible failures and fixes. In *Proceedings of the IEEE/ACM 41st International Conference on Software Engineering (ICSE'19)*. IEEE, 339–349.

[253] George Tucker, Andriy Mnih, Chris J. Maddison, John Lawson, and Jascha Sohl-Dickstein. 2017. REBAR: Low-variance, unbiased gradient estimates for discrete latent variable models. In *Proceedings of the Conference on Advances in Neural Information Processing Systems*. 2624–2633.

[254] Aaron van den Oord, Nal Kalchbrenner, Lasse Espeholt, Oriol Vinyals, Alex Graves, et al. 2016. Conditional image generation with PixelCNN decoders. In *Proceedings of the Conference on Advances in Neural Information Processing Systems*. 4790–4798.

[255] Bogdan Vasilescu, Casey Casalnuovo, and Premkumar Devanbu. 2017. Recovering clear, natural identifiers from obfuscated JS names. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*. ACM, 683–693.

[256] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *arXiv preprint arXiv:1706.03762* (2017).

[257] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. 2015. Pointer networks. In *Proceedings of the Conference on Advances in Neural Information Processing Systems*. 2692–2700.

[258] Alexander Waibel, Toshiyuki Hanazawa, Geoffrey Hinton, Kiyohiro Shikano, and K. Lang. 1988. Phoneme recognition: Neural networks vs. hidden Markov models vs. hidden Markov models. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP'88)*. IEEE, 107–110.

[259] Li Wan, Matthew Zeiler, Sixin Zhang, Yann Le Cun, and Rob Fergus. 2013. Regularization of neural networks using dropconnect. In *Proceedings of the International Conference on Machine Learning*. 1058–1066.

[260] 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*. ACM, 397–407.

[261] Shaowei Wang, David Lo, and Julia Lawall. 2014. Compositional vector space models for improved bug localization. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME'14)*. IEEE, 171–180.

[262] Yao Wang, Wan-dong Cai, and Peng-cheng Wei. 2016. A deep learning approach for detecting malicious JavaScript code. *Sec. Commun. Netw.* 9, 11 (2016), 1520–1534.

[263] Huihui Wei and Ming Li. 2017. Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. In *Proceedings of the International Joint Conference on Artificial Intelligence*. 3034–3040.

[264] Jason Weston, Antoine Bordes, Sumit Chopra, Alexander M. Rush, Bart van Merriënboer, Armand Joulin, and Tomas Mikolov. 2015. Towards AI-complete question answering: A set of prerequisite toy tasks. *arXiv preprint arXiv:1502.05698* (2015).

[265] Jason Weston, Sumit Chopra, and Antoine Bordes. 2014. Memory networks. *arXiv preprint arXiv:1410.3916* (2014).

[266] Jason E. Weston. 2016. Dialog-based language learning. In *Proceedings of the Conference on Advances in Neural Information Processing Systems*. 829–837.

[267] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 87–98.

[268] 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 (MSR'15)*. IEEE Press, Piscataway, NJ, 334–345.

[269] Ronald J. Williams. 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Mach. Learn.* 8, 3–4 (1992), 229–256.

[270] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. 2016. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144* (2016).

[271] Xin Xia, David Lo, Xinyu Wang, and Xiaohu Yang. 2015. Who should review this change?: Putting text and file location analyses together for more accurate recommendations. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME'15)*. IEEE, 261–270.

[272] Yan Xiao, Jacky Keung, Kwabena E. Bennin, and Qing Mi. 2019. Improving bug localization with word embedding and enhanced convolutional neural networks. *Inf. Softw. Technol.* 105 (2019), 17–29.

[273] Yan Xiao, Jacky Keung, Qing Mi, and Kwabena E. Bennin. 2017. Improving bug localization with an enhanced convolutional neural network. In *Proceedings of the 24th Asia-Pacific Software Engineering Conference (APSEC'17)*. IEEE, 338–347.

[274] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhudinov, Rich Zemel, and Yoshua Bengio. 2015. Show, attend and tell: Neural image caption generation with visual attention. In *Proceedings of the International Conference on Machine Learning*. 2048–2057.

[275] Xiaojun Xu, Chang Liu, and Dawn Song. 2017. SQLNet: Generating structured queries from natural language without reinforcement learning. *arXiv preprint arXiv:1711.04436* (2017).

[276] Shir Yadid and Eran Yahav. 2016. Extracting code from programming tutorial videos. In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. ACM, 98–111.

[277] Fan Yang, Jiazhong Nie, William W. Cohen, and Ni Lao. 2017. Learning to organize knowledge with N-gram machines. *arXiv preprint arXiv:1711.06744* (2017).

[278] Greg Yang. 2016. Lie access neural Turing machine. *arXiv preprint arXiv:1602.08671* (2016).

[279] Deheng Ye, Zhenchang Xing, Chee Yong Foo, Jing Li, and Nachiket Kapre. 2016. Learning to extract API mentions from informal natural language discussions. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME'16)*. IEEE, 389–399.

[280] Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to mine aligned code and natural language pairs from stack overflow. *arXiv preprint arXiv:1805.08949* (2018).

[281] Pengcheng Yin and Graham Neubig. 2017. A syntactic neural model for general-purpose code generation. *arXiv preprint arXiv:1704.01696* (2017).

[282] Wojciech Zaremba and Ilya Sutskever. 2015. Reinforcement learning neural Turing machines-revised. *arXiv preprint arXiv:1505.00521* (2015).

[283] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. 2014. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329* (2014).

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

[285] Xiang Zhang, Junbo Zhao, and Yann LeCun. 2015. Character-level convolutional networks for text classification. In *Proceedings of the Conference on Advances in Neural Information Processing Systems*. 649–657.

[286] Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2SQL: Generating structured queries from natural language using reinforcement learning. *arXiv preprint arXiv:1709.00103* (2017).

[287] Li Zhou, Kevin Small, Oleg Rokhlenko, and Charles Elkan. 2017. End-to-end offline goal-oriented dialog policy learning via policy gradient. *arXiv preprint arXiv:1712.02838* (2017).

[288] Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A. Efros. 2017. Unpaired image-to-image translation using cycle-consistent adversarial networks. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV'17)*.

[289] Julian Georg Zilly, Rupesh Kumar Srivastava, Jan Koutník, and Jürgen Schmidhuber. 2016. Recurrent highway networks. *arXiv preprint arXiv:1607.03474* (2016).

Received February 2019; revised February 2020; accepted February 2020