Toward Deep Learning Software Repositories

Martin White, Christopher Vendome, Mario Linares-Vásquez, and Denys Poshyvanyk
Department of Computer Science
College of William and Mary
Williamsburg, Virginia 23187–8795
Email: {mgwhite, cvendome, mlinarev, denys}@cs.wm.edu

***Abstract*—Deep learning subsumes algorithms that automatically learn compositional representations. The ability of these models to generalize well has ushered in tremendous advances in many fields such as natural language processing (NLP). Recent research in the software engineering (SE) community has demonstrated the usefulness of applying NLP techniques to software corpora. Hence, we motivate deep learning for software language modeling, highlighting fundamental differences between state-of-the-practice software language models and *connectionist models*. Our deep learning models are applicable to source code files (since they only require lexically analyzed source code written in any programming language) and other types of artifacts. We show how a particular deep learning model can remember its state to effectively model sequential data, e.g., streaming software tokens, and the state is shown to be much more expressive than discrete tokens in a prefix. Then we instantiate deep learning models and show that deep learning induces high-quality models compared to n-grams and cache-based n-grams on a corpus of Java projects. We experiment with two of the models' hyperparameters, which govern their capacity and the amount of context they use to inform predictions, before building several committees of software language models to aid generalization. Then we apply the deep learning models to code suggestion and demonstrate their effectiveness at a real SE task compared to state-of-the-practice models. Finally, we propose avenues for future work, where deep learning can be brought to bear to support model-based testing, improve software lexicons, and conceptualize software artifacts. Thus, our work serves as the *first* step toward deep learning software repositories.**

***Keywords*—Software repositories, machine learning, deep learning, software language models, n-grams, neural networks**

I. Introduction

The field of natural language processing (NLP) has developed many prominent techniques to support speech recognition [1] and statistical machine translation [2], among many other applications. One critical component to many of these techniques is a statistical language model, and the most prevalent class of statistical language models is simple Markov models called n-gram models (or "n-grams") [3]. n-grams are useful abstractions for modeling sequential data where there are dependencies among the terms in a sequence. A corpus can be regarded as a sequence of sequences, and corpus-based models such as n-grams learn conditional probability distributions from the order of terms in a corpus. Corpus-based models can be used for many different types of tasks like discriminating instances of data or generating new data that are characteristic of a domain.

The terms in a sequence can represent different entities depending on the domain. In software engineering (SE), sequential data emerge from countless artifacts, e.g., source code files and execution traces, where the terms (or "words") can be software tokens or method calls, and the sequences (or "sentences") can be lines of code or method call sequences. While software tokens and method calls characterize two different lexicons, statistical language models such as n-grams can be applied to corpora from each domain because the models represent simple *arrangements* of terms. Consequently, models like n-grams can be used to predict the next term in a sequence [4].

Recent research in the SE community has examined and successfully applied n-grams to formal languages, like programming languages, and SE artifacts [5]–[13]. The breadth of these applications in SE research and practice underscores the importance of the ability to effectively learn from sequential data in software repositories. *However, there is an apparent discrepancy between the representation power of models like n-grams for reaping information from repositories and the expressiveness that is produced and archived in repositories.* Consider the characteristics of modern software repositories and the requirements that these characteristics impose on models. Software repositories are massive depots of unstructured data, so good models require a lot of **capacity** to be able to learn from the voluminous scale rather than saturate after observing a fraction of the data that are available. Specifically, the kind of conceptual information that is buried in software repositories is very complex, requiring **expressive** models to manage this complexity. Moreover, software artifacts are laden with **semantics**, which means approaches that depend on matching lexemes are suboptimal. Finally, practical SE tasks require a lot of **context**—much more than short lists of the last two, three, and four terms in a sequence—whether the task is developing a feature or reproducing an issue. Capacity, expressiveness, semantics, and context are key concerns when mining sequential SE data and inducing software language models in particular. Nonetheless, n-grams have limited effective capacity [14]. They are not expressive, because they are simply smoothed counts of term co-occurrences [15]. They have trouble with semantics and generalizing beyond the explicit features observed in training [16]–[18]. Lastly, language models, including software language models, based on n-grams are quickly overwhelmed by the curse of dimensionality [16], so the effective amount of context is limited.

How can we improve the performance at SE tasks (e.g., code suggestion) based on software language models? In order to improve the quality of software language models, we must improve the **representation power** of the abstractions we use, so the goal of this paper is to marry deep learning and software language modeling. The purpose of applying deep learning to software language modeling is to improve the quality of the underlying abstractions for numerous SE tasks, viz. code suggestion [5], [12], deriving readable string test inputs to reduce human oracle cost [7], predicting programmer comments to improve search over code bases and code categorization [8], improving error reporting [10], generating feasible test cases to improve coverage [11], improving stylistic consistency to aid readability and maintainability [13], and code migration [19]–[21]. Thus, we make the following contributions:

- We introduce deep learning to SE research, specifically, software language modeling. Deep learning, a nascent field in machine learning, will provide the SE community with new ways to mine and analyze sequential data to support SE tasks.
- We motivate deep learning algorithms for software language modeling by clearly distinguishing them from state-of-the-practice software language models.
- We show that deep learning induces high-quality software language models compared to state-of-the-practice models using an intrinsic evaluation metric [4]. Then we demonstrate its effectiveness at a practical SE task.
- Our work is the first step in a new space of models and applications. While we focus on applying one deep architecture to one SE task, we believe deep learning is teeming with opportunities in SE research. We identify several avenues for future work, which highlight different ways that deep learning can be used to support practical SE tasks.

Sec. II will review background on software language modeling and deep learning for NLP. This section will define all the keywords (e.g., deep architecture, deep learning, deep software language model) and affirm the purpose of introducing these state-of-the-art approaches to SE research. Sec. III will pinpoint how this new class of software language models is poised to perform better at SE tasks by emphasizing their capacity and expressiveness, as well as their ability to model semantics and consider rich contexts. Sec. IV will use perplexity (PP), an intrinsic evaluation metric, to compare the quality of this new class of software language models to a state-of-the-practice baseline, and Sec. V will measure the models at a real SE task. Sec. VI will discuss threats to the validity of our work. Sec. VII will describe several avenues for future work. One avenue proposes using deep software language models to support objectives other than code coverage in model-based testing. Another avenue proposes using deep software language models to improve software lexicons. Sec. VIII concludes the paper.

II. Background and Related Work

In this section, we present background on statistical language models and preliminary research applying these models to software corpora. We focus on how current approaches to software language modeling can be improved, laying the foundation for Sec. III, where we show how deep learning can realize these improvements. Then we define all the keywords associated with *deep learning*, before presenting preliminary research applying deep learning to NLP.

A. Statistical Language Models

A statistical language model is a probability distribution over sentences in a language [4]. This ostensibly simple abstraction is remarkably effective for NLP tasks such as speech recognition and statistical machine translation. In statistical language modeling, our goal is to find a tractable representation of a sentence s by way of the joint distribution:

$$p(s) = \prod_{i=1}^{m} p(w_i|w_1^{i-1}) \approx \prod_{i=1}^{m} p(w_i|w_{i-n+1}^{i-1}). \quad (1)$$

In practice, we generalize the model's maximum likelihood estimates using one of many smoothing techniques [22]. Essentially, these probabilistic automata (i.e., n-grams) measure the degree of membership of every conceivable sentence in the language. Sentences frequently observed—in a generative sense—in the training corpus are judged to be more fluent than sentences observed less frequently (or not at all). In other words, we expect a good model to assign a high probability to a representative test document, or, equivalently, in-domain out-of-sample cases should have low cross entropy,

$$H_p(s) \approx -\frac{1}{m}\sum_{i=1}^{m} \log_2 p(w_i|w_{i-n+1}^{i-1}). \quad (2)$$

Cross entropy is an empirical estimate of how well a language model predicts terms in a sequence [9]. Likewise, PP $= 2^{H_p}$ estimates the average number of tokens at each point in the test document [9]. In language modeling, PP is a proxy for *quality*, and—as noted by Tu et al. [12]—good quality language models show great promise in SE applications. Our goal is to propose powerful abstractions novel to software language modeling using PP as empirical validation of their efficacy and capacity to support SE tasks.

B. Applications of Statistical Software Language Models

Hindle et al. [5] demonstrated that language models over software corpora emit a "naturalness" in the sense that real programs written by real people have useful statistical properties, encapsulated in statistical language models, that can leverage SE tasks. This work was an important first step in applying natural language abstractions to software corpora, but n-grams are simple approaches that do not have the capacity to learn representations that reliably generalize beyond the explicit features in a training corpus [17]. Furthermore, these models build limited domain abstractions, and they are quickly overwhelmed by the curse of dimensionality [5], [16], [23]–[25]. The expectation in software language modeling research is that performance at SE tasks will improve with models more sophisticated than n-grams [5]. The purpose of our work is to introduce compositional representations that are designed to process data in stages in a complex architecture. Each stage transforms internal representations as information flows from one layer of the architecture to the next [26]. The feature spaces in a deep learning model are fundamentally different than the conditional probability tables that constitute an n-gram model, and the power lies in the fact that these representations generalize well [26], [27].

Allamanis and Sutton [9] estimated an n-gram from a software corpus with more than one billion tokens, but we regard the massive scale as an organic smoothing technique. The model's effectiveness is still subject to token distances in the corpus, where clues behind the n-gram's relatively short prefix (or "history") are elided from the model's context [16],

[25], [28]. Moreover, the massive scale does not truly solve the problem of considering tokens' semantic similarity [16], [25], [28]. The approach for software language modeling that we present in Sec. III is designed to consider an arbitrary number of levels of context, where *context* takes on a much deeper meaning than concatenated tokens in a prefix. In our work, the deep learning model encodes context in a continuous-valued state vector, encapsulating much richer semantics. Finally, Allamanis and Sutton [9] conducted experiments where they collapsed the vocabulary by having the tokenizer replace identifiers and literals with generic tokens, which was a novel way to measure the model's performance on structural aspects of the code. However, we regard this approach as feature engineering. In this case, the token types in the corpus are engineered to solve the specific problem of modeling syntax. But the essence of deep learning, which underpins our work, is to design approaches that can *automatically* discover these feature spaces [26], [29] to—for instance—capture regularities at the syntactic, type, scope, *and* semantic levels [5].

Although Allamanis' giga-token model over source code demonstrated improvements in quality, a drawback to estimating n-grams over a massive corpus is losing resolution in the model. Good resolution yields regularities "endemic" to particular granularities, e.g., methods, classes, or modules. Of course, if the training corpus is too small, then the language model will be brittle for any practical application [14], [30]. A cache-based language model [31], [32] is designed to solve this optimization problem by interpolating a static model with a dynamic cache component. Recently, Tu et al. [12] applied cache models to software corpora:

$$p(w_i|w_{i-n+1}^{i-1}, c) = \alpha p_N(w_i|w_{i-n+1}^{i-1}) + \beta p_C(w_i|w_{i-n+1}^{i-1}) \quad (3)$$

where $0 \leq \alpha, \beta$ and $\alpha + \beta = 1$, c is the list of n-grams that are stored in the cache, p_N is a static n-gram model, and p_C is a dynamic cache model. The cache component encapsulates endemic and specific patterns in source code. While the cache component is a mechanism for capturing local context, the context we recur in a deep learning model will be an expressive continuous-valued state vector, which is capable of characterizing domain concepts [16], [26] rather than simply storing auxiliary conditional probability tables like p_C. Consequently, cache-based language models still suffer from the inability to understand semantic similarity, because they are fundamentally look-up tables, whereas deep learning models induce similar representations for token types used in similar ways [15].

C. Artificial Neural Networks

Connectionism includes an expansive and deep body of knowledge that pervades artificial intelligence, cognitive psychology, neuroscience, and philosophy, and a rigorous treatment is well beyond the scope of this paper. **Connectionist models** comprise neuron-like processing units, and each unit has an activity level computed from its inputs [33]. In a feed-forward topology, information in the **artificial neural network** flows from input units through hidden units to output units along connections with adjustable weights. A neural network **architecture** specifies intrinsic characteristics such as the number of units in the input, hidden, and output layers as well as the number of hidden layers in the network. A **deep architecture** comprises many hidden layers. Supervised learning algorithms discriminatively train [34] the weights to achieve the desired input-output behavior [35], so the hidden units automatically learn to represent important features of the domain [33]. This process of training the weights in a deep architecture is known as **deep learning**, and we refer to software language models based on deep learning as **deep software language models**. *Accordingly, deep learning models, including deep software language models, comprise multiple levels of nonlinear transformations* [26]. The canonical learning algorithm for neural networks is the back-propagation procedure [35], which allows an arbitrarily connected neural network to develop internal representations of its environment [35]. These neural activation patterns, or *distributed representations* [36], harness formidable and efficient internal representations of domain concepts. Units can "participate" in the representation of more than one concept, which gives way to *representational efficiency* (where different pools of units encode different concepts) and aids generalization [26], [37].

A simple two-layer **feed-forward neural network**, with one hidden layer and one output layer, cannot reliably learn beyond first-order temporal dependencies [38]. This architecture can be augmented with a short-term memory by recurring the hidden layer, which encapsulates the network's state, back to the input layer. The directed cycle provides context for the current prediction, and this continuous-valued state vector is fundamentally different than a discrete token in an n-gram's history. We can provide more context by extending the recurrence and considering an arbitrary number of levels of context. *From a temporal perspective*, this **recurrent neural network** (RNN) can be viewed as a very deep neural network [23], [39]–[42], where **depth** is the length of the longest path from an input node to an output node, and the purpose of the depth in this case is to reliably model temporal dependencies. The depth of a RNN is evident when you unfold the recurrence in time and measure the path from any unit in the deepest state vector to any output unit. Deep architectures like RNNs lie at the forefront of machine learning and NLP, but we are not indiscriminately introducing complexity. We expect these approaches will yield tremendous advances in SE as they already have in other fields.

D. Applications of Neural Network Language Models

Connectionist models for NLP go back at least as far as Elman [43], who used them to represent lexical categories, and Miikkulainen and Dyer [44], who developed a mechanism for building distributed representations for communication in a parallel distributed processing network. Bengio et al. [16] proposed a statistical model of natural language based on neural networks to learn distributed representations for words to allay the curse of dimensonality: One training sentence increases the probability of a combinatorial number of similar sentences [16]. Sequences of words were modeled by agglutinating the word representations of consecutive words in the corpus into a single pattern to be presented to the network. Bengio also constructed model ensembles by combining a neural network language model with low-order n-grams and observed that mixing the neural network's posterior distribution with an interpolated trigram improved the performance. This work also measured the performance of the model after adding direct connections from nodes in the projection layer to output nodes, but the topology of this network does not constitute a deep architecture. This model represents history by presenting n-gram patterns to the network, whereas our work is based on a network which considers an arbitrary number of contextual levels to inform predictions.

Our primary related work is the work by Mikolov [25], who excised the projection layer in Bengio's architecture [16] and added recurrent connections [45] from the hidden layer back to the input layer to form a RNN. Representing context with recurrent connections rather than patterns of n-grams is what distinguishes Mikolov's recurrent architecture from Bengio's feed-forward architecture. Mikolov reported improvements using RNNs over feed-forward neural networks [25] and implemented a toolkit [46] for training, evaluating, and using RNN language models. The package implements several heuristics for controlling the computational complexity of training RNNs [47]. Recently, Raychev et al. [48] proposed a tool based in part on Mikolov's package, RNNs, and program analysis techniques for synthesizing API completions.

III. A Deep Software Language Model

In this section, we specify a deep architecture for software language modeling and pinpoint how this new class of models is poised to improve the performance at SE tasks that use language models. We begin with the ubiquitous two-layer feed-forward neural network. As noted in Sec. II, these models cannot reliably learn beyond first-order temporal dependencies, so Elman networks augment the architecture with a short-term memory mechanism [43]. RNNs extend Elman networks by considering an arbitrary number of levels of context. RNNs are state-of-the-art models for NLP, but they are expensive to train, so a number of heuristics have been developed to control the complexity [47]. One heuristic is designed to reduce the complexity of computing the posterior distribution for each training example by factorizing the output layer and organizing the tokens into classes [49], [50]. Another heuristic involves training a maximum entropy model with a RNN by implementing direct connections between input units and output units [47].

Feed-forward networks. A pattern is presented to a feed-forward neural network by setting the value of each unit in the network's input layer x to the pattern's corresponding value. For instance, given a software corpus $\mathcal{C}$ of lexically analyzed tokens, we can represent a token w in the vocabulary $\mathcal{V}_{\mathcal{C}}$ using one-hot encoding[1] and set $w_i = x_i$. The token is projected onto a feature space $\mathcal{F}$ by an affine transformation $p_j = a_{ji}x_i + b_j$. *This transformation (or "pre-activation") is a fundamental point of divergence from models like n-grams.* Then each p_j is transformed by a differentiable, nonlinear function f such that $z_j = f(p_j)$ where z_j are the units that comprise the hidden layer z. The size of z (i.e., $|z|$) is an example of a "hyperparameter" [26], and adjusting this hyperparameter will regulate the model's **capacity** such that models with larger hidden layers yield more capacity [51], [52]. Practical choices for f include the logistic sigmoid, the hyperbolic tangent, and the rectifier [53]. These "activation" functions enable highly nonlinear and supremely **expressive** models [23].

After learning weights from x to z, when a fresh token is presented to the network, the units z_j will fire with varying intensities—analyzing the learned features—and ascribe a point in $\mathcal{F}$ to the token, effectively inducing *clusters* of examples in $\mathcal{F}$. These clusters enable a connectionist software language model to generalize beyond simple Markov chains in $\mathcal{C}$ like n-grams and model **semantics**. The hidden units are transformed $q_k = \beta_{kj}z_j$ (omitting all bias terms going forward) and activated by a function g in the output layer y such that $y_k = g(q_k)$. For multinomial classification problems, such as predicting the next token in source code, the softmax function activates values in the output layer such that $p(y_k|w) = g(q_k)$. In software language modeling, propagating a token $w(t)$ from x through z to y yields a posterior distribution over $\mathcal{V}_{\mathcal{C}}$, and the model predicts the next token $w(t+1)$ in a sequence:

$$\hat{w}(t+1) = \operatorname*{argmax}_k p(y_k|w(t)). \tag{4}$$

We require an algorithm for learning $\theta = \{a, \beta\}$ from $\mathcal{C}$, i.e., maximizing the likelihood function,

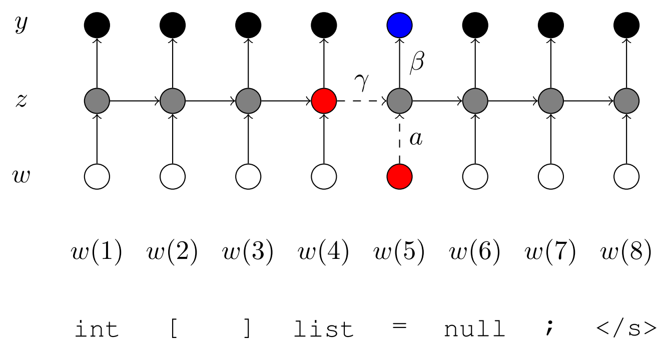
$$\mathcal{L}(\theta) = \prod_{t=1}^{|\mathcal{C}|} p(w(t+1)|w(t), \theta). \tag{5}$$

Equivalently, we can minimize the negative log-likelihood by training θ using stochastic gradient descent [54]. For each $w \in \mathcal{C}$, we compute the gradient of the error in the output layer, using a cross entropy criterion, and propagate this error back through the network [35], using the chain rule to evaluate partial derivatives of the error function with respect to the weights, before updating the weights. Overfitting is a concern since these models have the capacity to learn very complex representations, so θ is typically regularized [16], [18], [25].

Elman networks. The immediate concern with the model (Eq. (4)) is the inability to reliably learn beyond first-order temporal dependencies. n-grams encode "temporal" dependencies by learning tables of smoothed conditional probability distributions over a large number of prefixes. On the other hand, connectionist models can represent histories much more compactly. Specifically, an Elman network [43] augments a feed-forward network with a simple short-term memory mechanism. The short-term memory is realized by copying the hidden state $z(t-1)$ back to the input layer $x(t)$ and learning more weights γ to influence the hidden activations. This recurrence provides **context** for the current prediction. Thus, the input layer x in an Elman network is essentially a concatenation of $w(t)$ and $z(t-1)$, i.e., $x_i(t) = (w(t), z(t-1))_i$. In a feed-forward network, the pre-activation in z took the form $p_j = a_{ji}x_i$, but after recurring the state vector, the pre-activation in z takes the form $p_j(t) = a_{ji}w_i(t) + \gamma_{jj}z_j(t-1) = \alpha_{ji}x_i(t)$, where α is simply a concatenation of a_{ji} and γ_{jj}, and the model becomes $\theta = \{\alpha, \beta\}$. The cost of learning an additional $\mathcal{O}(m^2)$ parameters, where $m = |z|$, is met with improved **representation power** over sequences of software tokens.

Recurrent networks. A RNN (Fig. 1) extends the memory bank in an Elman network for an arbitrary number of levels of context—though there may be practical limits to the depth of the recurrence [55]. Therefore, because of the gradient problem, we typically truncate the back-propagation through time procedure [56]. Parenthetically, there are other ways

[1]In a **one-hot encoded** vector, one component is equal to one, and every other component is equal to zero, e.g., $w = (0, \ldots, 0, 1, 0, \ldots, 0)$.



y
z
w
γ
β
a
$w(1)$ $w(2)$ $w(3)$ $w(4)$ $w(5)$ $w(6)$ $w(7)$ $w(8)$
int [ ] list = null ; </s>

Fig. 1. RNN UNFOLDED IN TIME. The depth of a RNN is evident when the recurrence is unfolded in time. Time steps correspond to software tokens $w(t)$ in a corpus, where $w(0) =$ <s>. Each node in the figure represents a vector of units. White nodes are one-hot token representations; gray nodes are continuous-valued, hidden states; black nodes represent posterior distributions over the vocabulary. Units in the state vectors (gray nodes) compute their activation as a function of the current token and the previous state. Regarding the depth in this notional model, $y(8)$ is a function of $w(8), z(7)$, yet $z(7)$ is a function of $w(7), z(6)$, etc. Hence, predictions are informed by processing data in the past using *multiple levels of nonlinear transformations*.

to control this problem [25], [57], [58], but we omit these implementation details here. As the error is back-propagated through time in a RNN, each level of temporal context has an abating amount of influence on training γ, which is shared across time. This weight sharing yields an efficient representation compared to n-grams—which are hampered by the curse of dimensionality as they try to encode deeper contexts—and persisting a sequence of state vectors is much more expressive, in terms of discriminative power, than hard-coded prefixes. Interestingly, if τ is the number of time steps the error is back-propagated through time, the network is still capable of learning information longer than τ steps [25]. However, while a RNN is capable of learning powerful representations, it has some computationally expensive components, e.g., the posterior distribution in the output layer. Massive software repositories have a daunting challenge that arguably far exceeds the same problem in natural languages (including highly inflective natural languages), which is the size of the vocabulary. Computing the softmax function over extremely large vocabularies $|\mathcal{C}| \times \varepsilon$ times, where ε is the number of training epochs, is nontrivial. One solution to controlling this complexity is to factorize the output layer [3], [17], [50], [59]. Using class-based output layers has been shown to yield 15–30 times speed-up [25]. Direct connections are another implementation detail designed to improve performance. Bengio et al. [16] implemented direct connections from units in the projection layer to output units. The authors reported the connections did not help the model generalize from their relatively small corpus, but the connections did help reduce the training time. Mikolov [47] proposed direct connections from input units to output units and cast these connections as a maximum entropy model which can be trained with the neural network using stochastic gradient descent. The only change to the model specification is the addition of a term in the output pre-activation to account for the connections. The direct connections are reported to have yielded significant performance gains with respect to PP and word error rate [25].

Now, we are ready to present a deep architecture for software language modeling, specified in Eq. (6)–(8), without implementation details like class-based output layers and direct connections for clarity:

$$x_i(t) = (w(t), z(t-1))_i \tag{6}$$
$$z_j(t) = f(\alpha_{ji} x_i(t)) \tag{7}$$
$$y_k(t) = g(\beta_{kj} z_j(t)) \tag{8}$$

where $\alpha = \text{concatenate}(a, \gamma)$, $f(u_j) = \text{sigmoid}(u_j)$, and $g(u_k) = \text{softmax}(u_k)$. The model is similar to Eq. (4), except we present more than the current token to the network, i.e., $x(t)$ rather than simply $w(t)$. For example, in Fig. 1, the input layer $x(5)$ comprises the two red nodes (Eq. (6)), α concatenates the linear transformations represented by the dashed arrows, and the network computes (Eq. (7)–(8)) a posterior distribution (blue node). The argument of the maximum of this distribution is the network's prediction, which (in this case) should be the Java literal `null`.

TABLE I. STATISTICS ON THE CORPORA USED FOR THE STUDY

Corpus	Projects	Tokens	Unique
Training	732	4,979,346	90,415
Development	125	1,000,581	19,816
Testing	173	1,364,515	32,124
Total	1,030	7,344,442	125,181

IV. EMPIRICAL VALIDATION

The goal of our empirical study was to evaluate the effectiveness and expressiveness of deep software language models with the purpose of providing better abstractions for software language modeling and its associated SE tasks. We used PP, an intrinsic evaluation metric that estimates the average number of tokens to choose from at each point in a sequence, as the criterion. We began by computing the PP of several different n-gram configurations by varying the order n and adding a dynamic cache component to establish a state-of-the-practice baseline in software language modeling. Then we instantiated several deep software language models and computed the PP of these models with varying amounts of capacity and context over the same software corpus. Next, we selected the most performant architectures and interpolated several model instances to aid generalization and assess the performance of committees of software language models. Finally, deep software language models are capable online learners, so we also measured the performance of models that learned as they tested on out-of-domain samples.

A. Methodology and Data Collection

To build the corpora for our study, we used the JFlex scanner generator [60], which was packaged with a full Java lexical analyzer, to tokenize the source code in a repository of 16,221 Java projects cloned from GitHub. We augmented the production rules in the lexer since it originally did not support Java annotations. After tokenizing the files in each project, we sampled projects from the repository without replacement, querying enough projects to gather over seven million tokens. Then we randomly partitioned the projects into **mutually exclusive** training, development, and testing sets where approximately five million tokens were allotted for training, one million tokens for development, and one

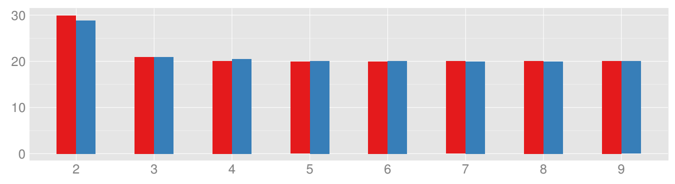


Fig. 2. PP V. ORDER. PP of back-off (red) and interpolated (blue) models at different orders (from two to nine). The 5-gram and 8-gram were the top performing back-off and interpolated models, respectively.

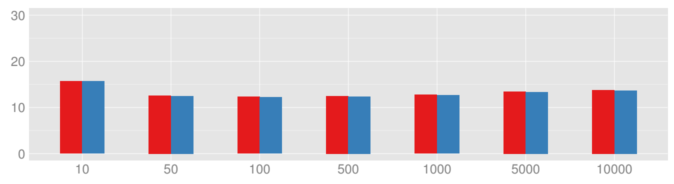


Fig. 3. PP V. CACHE SIZE. PP of 5-gram back-off (red) and interpolated 8-gram interpolated (blue) models with unigram caches varying in size from 10 to 10,000. The 100-token cache yielded the best result for each model type.

million tokens for testing. The purpose of a training set is to learn a useful representation of the domain. For example, a high-quality software language model is *useful*, because it can effectively predict the next token in a sequence. In a supervised setting, the training set couples input and its corresponding target to guide the form of the representation. To learn a good model, the supervised learning algorithm presents a token $w(t)$ to the model and, in the case of deep software language models, the back-propagation through time algorithm (with a gradient descent step) trains the model using the next token $w(t+1)$ in the sequence. Generally, the purpose of a development set is to govern hyperparameters such as the learning rate in gradient searches. The model is evaluated on the development set after each training epoch, and its relative performance on the development set can be used to judge convergence. It is important to note that data in the development set are not used to learn any of the model's parameters. For example, in the case of deep software language models, none of the weights are modified as the model is evaluated on the development set. Once the model is trained, it can be evaluated on a testing set. Notably, by partitioning the projects into mutually exclusive training, development, and testing sets, all of our experiments simulated new project settings (or "greenfield development") [12]. *Training and testing on distinct domains presents unique challenges for corpus-based models like software language models* [12].

For each set of projects, we removed blank lines from the files and randomly agglutinated the source files to form a training corpus, a development corpus, and a testing corpus. Tab. I lists summary statistics for each corpus, including the number of projects used to form each corpus, the total number of tokens in each corpus, and the number of unique tokens in each corpus. The total unique tokens denotes the number of unique tokens in a concatenated corpus of all three corpora. From these corpora, we used standard text normalization techniques that are used in the NLP community [25]. We used regular expressions to replace integers, real numbers, exponential notation, and hexadecimal numbers with a generic `<num>` token. After replacing numbers, we replaced singletons in each corpus as well as every token in the development and testing corpora that did not appear in the training corpus with `<unk>` to build an open vocabulary system [4] with a vocabulary size of 71,293.

B. State-of-the-Practice Software Language Models

In practice, n-grams' maximum likelihood estimates are discounted [22], and the probability mass gleaned from the observed n-grams is redistributed using either back-off [61] or interpolation [62]. We used SRILM [63] to estimate back-off and interpolated n-grams, from our training corpus, varying the order from two to nine. Each model was smoothed using modified Kneser-Ney [22] with an unknown token and no cutoffs. Fig. 2 plots PP versus order for each model. The models' results on the test corpus are virtually indistinguishable for this dataset, and both models appear to saturate near order five. This saturation is consistent with other studies on similar corpora [5]. With respect to PP, the most performant back-off model was the 5-gram (PP = 19.8911) and the most performant interpolated model was the 8-gram (PP = 19.9815). We augmented each of the models with a unigram cache, varying the size of the cache from 10 to 10,000. The dynamic unigram cache model was linearly interpolated with the static n-gram model using a mixing coefficient of 0.05. Fig. 3 plots PP versus unigram cache size for both models. The 100-token unigram cache component effectively improves the performance for both the 5-gram back-off model (PP = 12.3170) and the interpolated 8-gram model (PP = 12.2209). These performance gains from using a dynamic cache component are consistent with previous empirical studies [12].

We used the interpolated 8-gram model with a 100-token unigram cache (PP = 12.2209) as the baseline.

C. State-of-the-Art Software Language Models

After computing a baseline using state-of-the-practice software language models, we configured a RNN. These models have expansive design spaces spanned by several hyperparameters. We chose to measure the performance by varying the size m of the hidden layer and the number of steps τ in the truncated back-propagation through time algorithm. To train and test RNNs, we used the RNNLM Toolkit [46]. We instantiated 10 models with the same random seed, but we varied m from 50 to 500 units with sigmoid activations.[2] For each model, we truncated the back-propagation through time algorithm at $\tau = 10$ levels, updating the context every 10 iterations, and factorized the output layer into 268 classes. We used the default starting learning rate of 0.1 and the default ℓ_2 regularization parameter of 10^{-6}. The learning rate was annealed during training by monitoring the model's performance on the development set: After each training epoch, PP on the development set was computed to govern the learning rate schedule [46]. Finally, to determine the number of direct connections from input nodes to output nodes, we built a frequency distribution of the token types in the training

[2]While the RNNLM Toolkit uses sigmoid activations, the toolkit implements a simple mechanism to control the gradient problem by limiting the maximum size of gradients of errors that get accumulated in the hidden units [25].

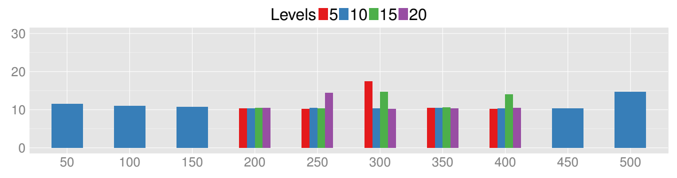


Fig. 4. PP V. HIDDEN SIZE AND DEPTH. PP of RNNs with hidden layers varying in size. Initially, we fixed the number of levels of context at 10. Then, for each $200 \leq m \leq 400$, we varied the number of levels of context.

corpus. We found that 995 token types covered 80.0% of the tokens in the training corpus, so we set the number of direct connections equal to 1,000. Fig. 4 plots PP versus m, where the deep models are shown to outperform—without a dynamic auxiliary component like a cache during testing—the baseline on this dataset, with the best results between 200 and 400 units. Next, we selected the five models with m between 200 and 400. For each model, we varied the number of levels of context $\tau = 5, 15, 20$, keeping the same configuration for every other parameter. Fig. 4 plots PP for each $200 \leq m \leq 400$ at four different values for τ. For our dataset, the most performant models in our (m, τ) design space were (300, 20) (PP = 10.1960) and (400, 5) (PP = 10.1721).

Deep learning (PP = 10.1721) beat the baseline.

D. Committees of Deep Software Language Models

Neural network language models are initialized with small random weights. Different instantiations will likely lead to models finding different local minima on the error surface, and models converging to different local minima may have different perspectives on the task. Therefore, we can construct committees of software language models by simply averaging $p(y|x)$ for each model instance [64]. Bengio et al. [16] reported performance gains by combining a neural network language model with an interpolated trigram, and the authors noted the performance gains suggest that the models make errors in different places. Likewise, Schwenk and Gauvain [18] interpolated neural network language models with back-off models to improve the performance in a speech recognition system. Mikolov [25] reported performance gains by combining several RNN language models. We instantiated five RNN-(300, 20) models and five RNN-(400, 5) models with different random seeds. Tab. II lists the results of combining several software language models on our dataset, e.g., RNN-(300, 20)-1,2 denotes the linear interpolation of two RNN-(300, 20) models—one model instantiated with random seed 1 and the another instantiated with random seed 2—where the coefficients in the mixture are 0.50. $\mathcal{N}$ denotes an interpolated 8-gram model with a 100-token unigram cache. So, RNN-(300, 20)-1,2,3,4,5,$\mathcal{N}$ represents the combination of five deep models and an interpolated n-gram model, where the combination of deep models has a weight of 0.60 in the mixture and the n-gram has a weight of 0.40. The top performing committee, RNN-(300, 20)-1,2,3,4,5,$\mathcal{N}$, achieves PP = 7.8512, which is equivalent to a cross-entropy score of 2.9729 bits. Recall these performance scores are computed using a training corpus of 732 *randomly* chosen projects, and the test corpus is another *random* collection of 173 out-of-domain projects. The committee improves the performance as compared to instances of each model, e.g., RNN-(300, 20)-1 (PP = 10.1960) and $\mathcal{N}$ (PP = 12.2209).

TABLE II. COMMITTEES OF SOFTWARE LANGUAGE MODELS

Committee	Coefficients	PP
RNN-(300, 20)-1,2	0.50	9.6467
RNN-(300, 20)-1,2,3	0.33	9.5060
RNN-(300, 20)-1,2,3,4	0.25	9.4549
RNN-(300, 20)-1,2,3,4,5	0.20	9.3534
RNN-(300, 20)-1,2,3,4,5,$\mathcal{N}$	0.60	7.8512
RNN-(400, 5)-1,2	0.50	9.5775
RNN-(400, 5)-1,2,3	0.33	9.9305
RNN-(400, 5)-1,2,3,4	0.25	9.6265
RNN-(400, 5)-1,2,3,4,5	0.20	9.5326
RNN-(400, 5)-1,2,3,4,5,$\mathcal{N}$	0.60	7.9346

Constructing committees of deep software language models can aid generalization and improve performance.

E. Deep Software Language Models Online

Cache-based language models separate static concerns from dynamic concerns using a convex combination of components, where a large static model is interpolated with a small dynamic model. In software language modeling, this small dynamic model has been used to capture local patterns in source code [12]. Neural network language models are capable of learning online by back-propagating the error for each test document and performing a gradient search thereby enabling adaptation. Tab. III lists the results of evaluating models online on our dataset. RNNs denotes a static model; RNNd denotes a dynamic model; RNNc denotes a committee of static and dynamic models using the corresponding mixture coefficient. $\mathcal{N}$ denotes an interpolated 8-gram model with a 100-token unigram cache as above. For example, RNNs-(300, 20)-5 denotes a static model with 300 hidden units and 20 levels of context instantiated with random seed 5, and RNNd-(300, 20)-5 denotes a similarly configured model that learns as it tests. RNNc-(300, 20)-5 denotes a committee comprising the static and dynamic models whose votes are weighted by the coefficients. The static and dynamic models were equally weighted in our experiments. Evaluating models online, where the deep software models can learn as they test, significantly improved the performance on our dataset.

For deep software models online (PP = 3.5958), the cross entropy scores are on the order of *two bits*.

When deep software language models are online, they can be incrementally trained. Thus, in new project settings, online learners are able to automatically adapt as the project is being developed. In the committee of static and dynamic models, the static component can be regarded as weak prior knowledge of the domain in new project settings, and the dynamic component acts as an incremental learner, which adapts as the project is being developed. Although the dynamic models performed noticeably better on our dataset, one potential

TABLE III. ONLINE MODELS

Model	Coefficients	PP
RNNs-(300, 20)-5	-	10.1686
RNNd-(300, 20)-5	-	3.6518
RNNc-(300, 20)-5	0.50	3.9856
RNNs-(400, 5)-1	-	10.1712
RNNd-(400, 5)-1	-	3.5958
RNNc-(400, 5)-1	0.50	3.7480

benefit of sacrificing some of the performance gain by using a committee is that static models can serve as anchors and help prevent the model from being "poisoned," i.e., degenerated by learning unreliable information online.

V. CASE STUDY: CODE SUGGESTION

In Sec. IV, our intrinsic evaluation compared the quality of deep software language models to state-of-the-practice models. In this section, we conduct an extrinsic evaluation [4] to measure the performance of deep software language models at a real SE task, code suggestion, and *we show that deep learning improves the performance at an SE task based on software language models*. A **code suggestion** engine recommends the next token given the context [5], [12]. The goal of the study was to measure the accuracy of deep software language models for code suggestion with the purpose of providing better tools and automated techniques to aid software development and maintenance. The context of the study consisted of the same corpora listed in Tab. I. The quality focus concerned code suggestion of tokens in the testing corpus. We examined the following research questions:

RQ1 Do deep learning models (Sec. III) significantly outperform state-of-the-practice models (Sec. II) at code suggestion on our dataset?

RQ2 Are there any distinguishing characteristics of the test documents on which the deep learning models achieve considerably better performance as compared to state-of-the-practice models?

RQ1. We used Top-k accuracy to compare deep software language models to state-of-the-practice models at code suggestion. Top-k accuracy has been used in previous code suggestion studies [5], [12]. Tab. IV lists our Top-k results, where $k = 1, 5, 10$, for the most performant static and dynamic models of each model type. The deep learning models appear to outperform the n-grams at each level, so we designed comparative experiments to measure the statistical significance of our results. The **treatments** in our experimental design were the language models. The **experimental units** were the sentences in the test corpus, and the **responses** were the Top-k scores. The **null hypothesis** stated there was no difference in performance. The (two-tailed) **research hypothesis** stated there was a difference in performance. We tested these hypotheses at $\alpha = 0.05$ using the Wilcoxon test [65], a nonparameteric test, to determine whether the reported differences were statistically significant. Comparing the best deep software language model (RNNd-(300, 20)-5) to the best n-gram model (interpolated 8-gram), we found $p \leq 2.2 \times 10^{-16} < 0.05 = \alpha$ in all three cases (i.e., Top-1, Top-5, and Top-10); therefore, we rejected the null hypothesis, suggesting that a statistically

TABLE IV. TOP-K ACCURACY (%)

Model	Top-1	Top-5	Top-10
Interpolated 8-gram	49.7	71.3	78.1
Interpolated 8-gram 100-cache	4.8	69.5	78.5
RNNs-(400, 5)-1	61.1	78.4	81.4
RNNd-(300, 20)-5	72.2	88.4	92.0

significant difference existed. We interpreted the difference as deep learning realizing an improvement at the code suggestion task. Regarding the effect size (Cliff's δ), we observed a medium effect size for Top-1, a large effect size for Top-5, and a medium effect size for Top-10.

Deep learning significantly outperformed n-grams at code suggestion on our dataset.

RQ2. After assessing the significance of applying deep learning to a real SE task, we conducted an exploratory study on the performance results. We began by sorting all the sentences in the test corpus by their Top-10% (according to the n-gram), i.e., the ratio of the number of tokens (including `</s>`) in the sentence suggested in the Top-10 divided by the total number of tokens in the sentence. We observed that the sentences at the top of the list with low Top-10 scores were relatively short in length. Some of these sentences only comprised annotations (e.g., @Before and @Test) and others only comprised keywords (e.g., else, try, and finally). Given the poor performance of n-grams on these test documents, we were interested in comparing the performance of deep software language models on these sentences. We designed another set of experiments to compare the performance of the two models; however, in these experiments, the **experimental units** were sentences of length one, two, or three, respectively. Each experiment compared the models' Top-k performances at each sentence length. The **null hypothesis** for each comparative experiment stated there was no difference in performance. The (two-tailed) **research hypothesis** stated there was a difference in performance. We tested these hypotheses as above. All comparisons yielded statistically significant differences, where $p \leq 2.2 \times 10^{-16} < 0.05 = \alpha$; therefore, we rejected the null hypothesis (for each comparison) and interpreted the difference as improved performance. Regarding the effect size (Cliff's δ), we only found large effect sizes for Top-5, where the sentence length was equal to two, and for Top-10, where the sentence length was equal to two or three.

Our results show that deep learning improves the performance at a SE task based on software language models. Moreover, there may be interesting cases in software corpora where deep learning outperforms models like n-grams. Sentences of length one or two tokens are arguably more germane to software corpora than natural language corpora.

VI. THREATS TO VALIDITY

Threats to **construct validity** concern the relationship between theory and observation and relate to possible measurement imprecisions when extracting data used in a study. In mining the Git repositories and collecting the projects for our analysis, we relied on both the GitHub API and the `git` utility. These tools are under active development with a community supporting them. Additionally, the GitHub API is the primary interface to extract project information. We cannot exclude imprecisions due to the implementation of such an API.

Threats to **internal validity** can be related to confounding factors internal to a study that could have affected the results. In our study, we relied on RNNLM to train and evaluate deep software language models. While RNNLM is a reliable implementation that has been used in a number of NLP experiments [46], [47], [49], it is still an evolving project. However, our results and trends are in line with those that have been obtained in the field of NLP; thus, we are confident that the results are reliable.

Threats to **external validity** represent the ability to generalize the observations in a study. We do not claim that the obtained results can be observed across other repositories or projects, especially projects written in other programming languages. Additionally, our dataset is representative of only repositories hosted on GitHub, so we do not claim that the results generalize to all Java projects. GitHub's exponential growth and popularity as a public forge indicates that it represents a large portion of the open source community. While GitHub contains a large number of repositories, it may not necessarily be a comprehensive set of all open source projects or even all Java projects. However, we analyzed the diversity of the projects from the proposed metrics in Nagappan et al. [66] and compared our dataset to the projects available on Boa [67] and found 1,556 projects out of the 16,221 projects. We also analyzed the diversity of the 1,030 tokenized projects in our training, development, and test corpora, and we were able to match 128 projects. Our entire dataset had a diversity score of 0.3455, and the subset that we used to conduct our language modeling experiments had a diversity score of 0.2208. According to our dimensions, these values suggest that approximately 10% of our entire dataset covers one-third of the open source projects, and approximately 10% of our corpus covers one-fifth of open source projects. In our diversity analysis, we considered six metrics: programming languages, developers, project age, number of committers, number of revisions, and number of programming languages. For the entire dataset, we had scores of 0.45, 0.99, 1.00, 0.99, 0.96, and 0.99, respectively. For the study corpora, we had scores of 0.38, 0.98, 1.00, 0.98, 0.92, and 1.00, respectively. These results indicate that both our dataset and our corpora have high-dimensional diversity coverage for the relevant dimensions to our study. Since we consider only Java projects, it is expected that our representativeness would be rather low in the programming languages dimension. Thus, our results are representative of a proportion of the open source community; in particular, we have high coverage within our key dimensions. Further evaluation of projects across other open source repositories and other programming languages would be necessary to validate our observations in a more general context. It is also important to note that we only consider open source projects.

VII. Avenues for Future Work

There are two principal research components in our future work on deep software language modeling and, generally, using deep learning to mine sequential SE data. One research component examines **extensions** of the models. One set of extensions involves search problems, such as hyperparameter optimization, designed to improve deep learning-based approaches (e.g., our deep software language models) for mining sequential SE data. Another set of extensions involves entirely new architectures and models, such as stacked RNNs [57] and recursive neural networks [68], for mining sequential SE data. The other research component examines **applications** of deep architectures to SE tasks. We present three of these applications, informally organized according to *features*—from (concrete) token types to (abstract) conceptualizations. The first application, model-based testing, is an example of an SE task that can benefit from our deep software language models, where the *raw features* do not have to be words per se. As we noted in Sec. I, the nature of the terms depends on the domain, and this application supports that claim. The next application, software lexicon, shows that deep software language models are not simply useful for their high-quality output. We can also use their internal representations and *feature detectors* to support SE tasks. Thus, the same model that serves as a code suggestion engine can also be used to improve the software lexicon. Finally, while RNNs are deep in time, our last application, conceptualizing software artifacts, suggests that deep learning can be used to learn *hierarchies of features* to gradually abstract SE artifacts from token streams to useful concepts to support software maintenance and evolution.

Hyperparameter Optimization. Deep architectures comprise multiple levels of nonlinear transformations. Different subspaces in a deep architecture are trained as information flows forward and supervision propagates back through the network, but models like RNNs entail a considerable number of hyperparameters for governing different facets of the architecture, e.g., the size of the hidden layer, the number of levels before truncating the back-propagation through time algorithm, the number of classes for partitioning token types in the output layer, the learning rate, and the amount of regularization. We believe this translates to an expansive design space with new and important search problems for SE research to optimize these complex architectures over SE datasets for SE tasks. SE research has examined similar problems in different contexts [69]. Additionally, recent research in the machine learning community has proposed methodologies for automatically configuring the optimal set of hyperparameters [70], [71], but these approaches have not been measured using SE datasets in the context of SE tasks.

Model-based Testing. Tonella et al. [11] demonstrated how interpolating a spectrum of low-order Markov models inferred from an event log can be used to improve code coverage, since "backing off" increases the likelihood of deriving feasible test cases. Conceptually, the work by Tonella et al. uses "naturalness" (Sec. II) at different scales to improve code coverage, but we envision much more opportunity in model-based testing by exploiting the posterior distribution in the output layer of a deep software language model. While the posterior can specify natural event sequences, we can also infer *un*natural event sequences from this model. Our work will segment the posterior's domain into a natural space and an unnatural space. In this sense, we propose a novel interpretation of deep software language models as **natural bits** to support other aspects of software testing, e.g., destructive testing and regression testing. Moreover, the natural space in this model is

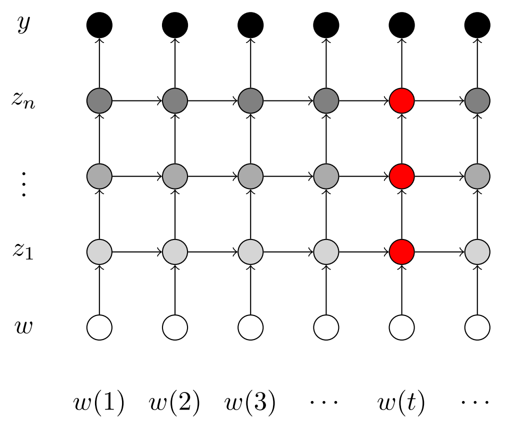


Fig. 5. STACKED RNN UNFOLDED IN TIME. While RNNs are deep in time, stacking RNNs yield architectures that are deep in both time and space. The depth in space (e.g., red nodes at time t) enables hierarchical processing which potentially learns information across multiple time scales [40], [57].

not fixed. We envision a framework for adapting this space by dynamically toggling event sequences as natural or unnatural depending on the evidence to *steer* the model online according to specific objectives (other than code coverage).

Software Lexicon. Software maintenance is hard. Since program comprehension is one contributing factor, improving the software lexicon is one way to support critical maintenance tasks. We envision a novel **lexicon broker** to negotiate commits with the expressed goal of supporting program comprehension by consolidating concepts and, to this end, serving as a recommendation engine in cases where developers' implementations can be improved. How can we enable this broker? While a deep software language model can effectively support many different SE tasks, the architecture's components may be used for other pertinent SE tasks such as learning software synonyms in massive repositories [72]–[76]. Recall the deep software language model embeds a token vector in a low-dimensional subspace using a linear projection a (Sec. III). This is perhaps best understood by thinking of the vector $a_{ji}w_i$ as a linear combination of the columns of a [77], i.e., $a_{ji}w_i = \sum_{j=1}^{n} w_j a_{\cdot j} = a_{\cdot N} \in \mathbb{R}^{m\times 1}$ where $1 \leq N \leq n$, $m = |z|$, and the last equality is because w is one-hot encoded. So, each column in a represents one token in the vocabulary. These are contextualized feature vectors which can leverage intelligent recommendations on improving the lexicon. After conducting our empirical validation (Sec. IV) and observing how well the deep software language models performed on our dataset, we conducted a cursory study of the models' internal representations. The purpose of this small exploratory study was to begin to understand how these models can be analyzed to address other SE concerns, e.g., token similarity [75]. We extracted the token embeddings $a_{\cdot j} \in \mathbb{R}^{300\times 1}$ from our RNNs-(300, 20)-1 model (Sec. IV). Tab. V lists the two closest tokens, using Euclidean distance, for three distinct queries. Of the 71,293 token types in the vocabulary (Sec. IV), the two closest tokens to getX were two other getter methods that appear to be related to position or size. Again, of the 71,293 token types, the closest tokens to transient were two other Java keywords. Finally, the two closest tokens to @BeforeClass were two other Java annotations. While these are intriguing anecdotes, this is very preliminary work in this space, but we believe these cursory observations warrant a deeper and much more rigorous examination in different SE contexts. Finally, it is important to note that architectures, e.g. stacked RNNs (Fig. 5), that are deep in both time *and* space, provide more opportunity for using components for other SE tasks.

TABLE V. QUERYING SIMILAR TOKENS USING TOKEN EMBEDDINGS

Query	Closest Tokens
getX	getY, getWidth
transient	native, volatile
@BeforeClass	@AfterClass, @Extension

Language Models for Software Evolution. Sutskever et al. [23] trained a type of RNN for character-level language modeling and found the model learned to balance parentheses and quotes over distances as long as 30 characters. They note that a character-level n-gram language model could only do this by modeling 31-grams. We believe this has some particularly important implications for modeling software corpora because of software concerns like scope and encapsulation. If a software language model is capable of balancing { and } in source code, then perhaps deep architectures can be designed with enough capacity to reliably predict the next word in a sequence "in time," yet—at a higher level of abstraction—the architecture is capable of representing invariant features of the evolution of the software system "in space." We see these feature hierarchies as gateways to entirely novel methods for classifying software systems in many different software maintenance and evolution contexts.

VIII. CONCLUSION

State-of-the-practice software language models are bound to the n-gram features that are apparent by simply scanning a corpus and aggregating counts of specific and discrete token sequences (Sec. II). On the other hand, deep learning uses expressive, continuous-valued representations that are capable of learning more robust models (Sec. III). We propose that SE research, with a wealth of unstructured data, is a unique opportunity to employ these state-of-the-art approaches. By empirically demonstrating that a relatively simple RNN configuration can outperform n-grams and cache-based n-grams with respect to PP on a Java corpus, deep software language models are shown to be high-quality software language models (Sec. IV), capable of showing great promise in SE applications [12]. We also demonstrate an improvement in performance at an SE task (Sec. V). Finally, we identify avenues for future work using deep software language models to conduct model-based testing, improve software lexicons, and support software maintenance and evolution (Sec. VII). Computer vision, speech recognition, and other fields have occupied the attention of the approaches we propose in this paper. Our work is one step—the *first* step—toward deep learning software repositories.

ACKNOWLEDGMENT

We would like to thank Abram Hindle from the University of Alberta for sharing his tools and data from his empirical study [5]. This work is supported in part by the NSF CCF-1218129 and NSF-1253837 grants. Any opinions, findings, and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

References

[1] F. Jelinek, *Statistical Methods for Speech Recognition*. Cambridge, MA, USA: MIT Press, 1997.

[2] P. Koehn, *Statistical Machine Translation*, 1st ed. New York, NY, USA: Cambridge University Press, 2010.

[3] J. Goodman, "Classes for fast maximum entropy training," *CoRR*, vol. cs.CL/0108006, 2001.

[4] D. Jurafsky and J. H. Martin, *Speech and Language Processing*, 2nd ed. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2009.

[5] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 837–847.

[6] T. T. Nguyen, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "A statistical semantic language model for source code," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE '13. New York, NY, USA: ACM, 2013, pp. 532–542.

[7] S. Afshan, P. McMinn, and M. Stevenson, "Evolving readable string test inputs using a natural language model to reduce human oracle cost," in *Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, ser. ICST '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 352–361.

[8] D. Movshovitz-Attias and W. W. Cohen, "Natural language models for predicting programming comments," in *ACL*. Sofia, Bulgaria: Association for Computational Linguistics, August 2013.

[9] M. Allamanis and C. A. Sutton, "Mining source code repositories at massive scale using language modeling," in *MSR*, 2013, pp. 207–216.

[10] J. C. Campbell, A. Hindle, and J. N. Amaral, "Syntax errors just aren't natural: Improving error reporting with language models," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR '14. New York, NY, USA: ACM, 2014, pp. 252–261.

[11] P. Tonella, R. Tiella, and D. C. Nguyen, "Interpolated n-grams for model based testing," in *ICSE*, 2014, pp. 562–572.

[12] Z. Tu, Z. Su, and P. Devanbu, "On the localness of software," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE '14. New York, NY, USA: ACM, 2014, pp. 269–280.

[13] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Learning natural coding conventions," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE '14. New York, NY, USA: ACM, 2014, pp. 281–293.

[14] R. Rosenfeld, "Two decades of statistical language modeling: Where do we go from here?" in *Proceedings of the IEEE*, vol. 88, 2000, pp. 1270–1278.

[15] A. Mnih and Y. W. Teh, "A fast and simple algorithm for training neural probabilistic language models," in *Proceedings of the 29th International Conference on Machine Learning*, 2012, pp. 1751–1758.

[16] Y. Bengio, R. Ducharme, P. Vincent, and C. Janvin, "A neural probabilistic language model," *J. Mach. Learn. Res.*, vol. 3, pp. 1137–1155, Mar. 2003.

[17] F. Morin and Y. Bengio, "Hierarchical probabilistic neural network language model," in *Proceedings of the Tenth International Workshop on Artificial Intelligence and Statistics*. Society for Artificial Intelligence and Statistics, 2005, pp. 246–252.

[18] H. Schwenk and J.-L. Gauvain, "Training neural network language models on very large corpora," in *Proceedings of the Conference on Human Language Technology and Empirical Methods in Natural Language Processing*, ser. HLT '05. Stroudsburg, PA, USA: Association for Computational Linguistics, 2005, pp. 201–208.

[19] A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Lexical statistical machine translation for language migration," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE '13. New York, NY, USA: ACM, 2013, pp. 651–654.

[20] ——, "Migrating code with statistical machine translation," in *Companion Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE Companion '14. New York, NY, USA: ACM, 2014, pp. 544–547.

[21] A. T. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Statistical learning approach for mining API usage mappings for code migration," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14. New York, NY, USA: ACM, 2014, pp. 457–468.

[22] S. F. Chen and J. Goodman, "An empirical study of smoothing techniques for language modeling," in *Proceedings of the 34th Annual Meeting on Association for Computational Linguistics*, ser. ACL '96. Stroudsburg, PA, USA: Association for Computational Linguistics, 1996, pp. 310–318.

[23] I. Sutskever, J. Martens, and G. Hinton, "Generating text with recurrent neural networks," in *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, ser. ICML '11. New York, NY, USA: ACM, June 2011, pp. 1017–1024.

[24] E. Arisoy, T. N. Sainath, B. Kingsbury, and B. Ramabhadran, "Deep neural network language models," in *Proceedings of the NAACL-HLT 2012 Workshop: Will We Ever Really Replace the N-gram Model? On the Future of Language Modeling for HLT*. Montréal, Canada: Association for Computational Linguistics, June 2012, pp. 20–28.

[25] T. Mikolov, "Statistical language models based on neural networks," Ph.D. dissertation, 2012.

[26] Y. Bengio, "Learning deep architectures for AI," *Found. Trends Mach. Learn.*, vol. 2, no. 1, pp. 1–127, Jan. 2009.

[27] T. Mikolov, W. tau Yih, and G. Zweig, "Linguistic regularities in continuous space word representations." in *HLT-NAACL*. The Association for Computational Linguistics, 2013, pp. 746–751.

[28] R. Rosenfeld, "A maximum entropy approach to adaptive statistical language modeling," *Computer, Speech and Language*, vol. 10, pp. 187–228, 1996.

[29] Y. Bengio, "Deep learning of representations: Looking forward," in *Proceedings of the First International Conference on Statistical Language and Speech Processing*, ser. SLSP '13. Berlin, Heidelberg: Springer-Verlag, 2013, pp. 1–37.

[30] B.-J. P. Hsu, "Language modeling for limited-data domains," Ph.D. dissertation, Cambridge, MA, USA, 2009.

[31] R. Kuhn and R. De Mori, "A cache-based natural language model for speech recognition," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 12, no. 6, pp. 570–583, Jun. 1990.

[32] P. Clarkson and A. Robinson, "Language model adaptation using mixtures and an exponentially decaying cache," in *In Proceedings of ICASSP-97*, 1997, pp. 799–802.

[33] G. E. Hinton, "Connectionist learning procedures," *Artif. Intell.*, vol. 40, no. 1-3, pp. 185–234, 1989.

[34] C. M. Bishop and J. Lasserre, "Generative or discrimative? Getting the best of both worlds," in *Bayesian Statistics 8*, International Society for Bayesian Analysis. Oxford University Pres, 2007, pp. 3–24.

[35] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Neurocomputing: Foundations of research." Cambridge, MA, USA: MIT Press, 1988, ch. Learning Representations by Back-propagating Errors, pp. 696–699.

[36] G. E. Hinton, "Learning distributed representations of concepts," in *Proceedings of the Eighth Annual Conference of the Cognitive Science Society*. Hillsdale, NJ: Erlbaum, 1986, pp. 1–12.

[37] G. E. Hinton, J. L. McClelland, and D. E. Rumelhart, "Distributed representations," in *Parallel Distributed Processing. Volume 1: Foundations*. Cambridge, MA: MIT Press, 1986, ch. 3, pp. 77–109.

[38] N. K. Sinha and M. M. Gupta, *Soft Computing and Intelligent Systems: Theory and Applications*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999.

[39] I. Sutskever, J. Martens, G. E. Dahl, and G. E. Hinton, "On the importance of initialization and momentum in deep learning," in *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, vol. 28, no. 3. JMLR Workshop and Conference Proceedings, May 2013, pp. 1139–1147.

[40] M. Hermans and B. Schrauwen, "Training and analysing deep recurrent neural networks," in *Advances in Neural Information Processing Systems 26*. Curran Associates, Inc., 2013, pp. 190–198.

[41] R. Pascanu, Ç. Gülçehre, K. Cho, and Y. Bengio, "How to construct deep recurrent neural networks," *CoRR*, vol. abs/1312.6026, 2013.

[42] O. İrsoy and C. Cardie, “Opinion mining with deep recurrent neural networks,” in *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, 2014, pp. 720–728.

[43] J. L. Elman, “Finding structure in time,” *COGNITIVE SCIENCE*, vol. 14, no. 2, pp. 179–211, 1990.

[44] R. Miikkulainen and M. G. Dyer, “Natural language processing with modular neural networks and distributed lexicon,” *Cognitive Science*, vol. 15, pp. 343–399, 1991.

[45] M. I. Jordan, “Serial order: A parallel distributed processing approach,” Institute for Cognitive Science, University of California, San Diego, Tech. Rep. ICS Report 8604, 1986.

[46] T. Mikolov, S. Kombrink, A. Deoras, L. Burget, and J. Cernocký, “RNNLM - Recurrent neural network language modeling toolkit,” in *Proceedings of ASRU 2011*. IEEE Signal Processing Society, 2011, pp. 1–4.

[47] T. Mikolov, A. Deoras, D. Povey, L. Burget, and J. Cernocký, “Strategies for training large scale neural network language models,” in *ASRU*, 2011, pp. 196–201.

[48] V. Raychev, M. Vechev, and E. Yahav, “Code completion with statistical language models,” in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’14. New York, NY, USA: ACM, 2014, pp. 419–428.

[49] T. Mikolov, S. Kombrink, L. Burget, J. Cernocký, and S. Khudanpur, “Extensions of recurrent neural network language model,” in *Proceedings of the 2011 IEEE International Conference on Acoustics, Speech, and Signal Processing, ICASSP 2011*. IEEE Signal Processing Society, 2011, pp. 5528–5531.

[50] Y. Shi, W. Zhang, J. Liu, and M. T. Johnson, “RNN language model with word clustering and class-based output layer,” *EURASIP J. Audio, Speech and Music Processing*, vol. 2013, p. 22, 2013.

[51] Y. Bengio, “Practical recommendations for gradient-based training of deep architectures,” *CoRR*, vol. abs/1206.5533, 2012.

[52] Y. Bengio, A. C. Courville, and P. Vincent, “Unsupervised feature learning and deep learning: A review and new perspectives,” *CoRR*, vol. abs/1206.5538, 2012.

[53] X. Glorot, A. Bordes, and Y. Bengio, “Deep sparse rectifier neural networks,” in *JMLR W&CP: Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics (AISTATS 2011)*, Apr. 2011.

[54] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, “Backpropagation applied to handwritten zip code recognition,” *Neural Comput.*, vol. 1, no. 4, pp. 541–551, Dec. 1989.

[55] Y. Bengio, P. Simard, and P. Frasconi, “Learning long-term dependencies with gradient descent is difficult,” *Trans. Neur. Netw.*, vol. 5, no. 2, pp. 157–166, Mar. 1994.

[56] P. Werbos, “Backpropagation through time: what does it do and how to do it,” in *Proceedings of IEEE*, vol. 78, no. 10, 1990, pp. 1550–1560.

[57] J. Schmidhuber, “Learning complex, extended sequences using the principle of history compression,” *Neural Comput.*, vol. 4, no. 2, pp. 234–242, Mar. 1992.

[58] R. Pascanu, T. Mikolov, and Y. Bengio, “On the difficulty of training recurrent neural networks,” in *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013*, 2013, pp. 1310–1318.

[59] A. Mnih and G. Hinton, “Three new graphical models for statistical language modelling,” in *Proceedings of the 24th International Conference on Machine Learning*, ser. ICML ’07. New York, NY, USA: ACM, 2007, pp. 641–648.

[60] (2014) JFlex. [Online]. Available: http://jflex.de/

[61] S. M. Katz, “Estimation of probabilities from sparse data for the language model component of a speech recognizer,” in *IEEE Transactions on Acoustics, Speech and Signal Processing*, 1987, pp. 400–401.

[62] F. Jelinek and R. L. Mercer, “Interpolated estimation of markov source parameters from sparse data,” in *In Proceedings of the Workshop on Pattern Recognition in Practice*, May 1980, pp. 381–397.

[63] A. Stolcke, “SRILM - an extensible language modeling toolkit.” in *INTERSPEECH*. ISCA, 2002.

[64] C. M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.

[65] D. Sheskin, *Handbook of Parametric and Nonparametric Statistical Procedures.*, 2nd ed. Chapman & Hall/CRC, 2000.

[66] M. Nagappan, T. Zimmermann, and C. Bird, “Diversity in software engineering research,” ser. ESEC/FSE ’13, 2013, pp. 466–476.

[67] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, “Boa: A language and infrastructure for analyzing ultra-large-scale software repositories,” ser. ICSE ’13, 2013, pp. 422–431.

[68] R. Socher, C. C.-Y. Lin, A. Y. Ng, and C. D. Manning, “Parsing natural scenes and natural language with recursive neural networks.” in *ICML*. Omnipress, 2011, pp. 129–136.

[69] A. Panichella, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia, “How to effectively use topic models for software engineering tasks? An approach based on genetic algorithms,” in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE ’13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 522–531.

[70] J. Bergstra and Y. Bengio, “Random search for hyper-parameter optimization,” *J. Mach. Learn. Res.*, vol. 13, no. 1, pp. 281–305, Feb. 2012.

[71] J. Snoek, H. Larochelle, and R. P. Adams, “Practical bayesian optimization of machine learning algorithms,” in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. Burges, L. Bottou, and K. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 2951–2959.

[72] J. Yang and L. Tan, “Inferring semantically related words from software context,” in *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, ser. MSR ’12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 161–170.

[73] M. J. Howard, S. Gupta, L. Pollock, and K. Vijay-Shanker, “Automatically mining software-based, semantically-similar words from comment-code mappings,” in *Proceedings of the 10th Working Conference on Mining Software Repositories*, ser. MSR ’13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 377–386.

[74] Y. Tian, D. Lo, and J. L. Lawall, “Automated construction of a software-specific word similarity database,” in *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering, CSMR-WCRE 2014, Antwerp, Belgium, February 3-6, 2014*, 2014, pp. 44–53.

[75] Y. Tian, D. Lo, and J. Lawall, “Sewordsim: Software-specific word similarity database,” in *Companion Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE Companion ’14. New York, NY, USA: ACM, 2014, pp. 568–571.

[76] J. Yang and L. Tan, “Swordnet: Inferring semantically related words from software context,” *Empirical Softw. Engg.*, vol. 19, no. 6, pp. 1856–1886, Dec. 2014.

[77] L. N. Trefethen and D. Bau, *Numerical Linear Algebra*. SIAM, 1997.