Fast and Memory-Efficient Neural Code Completion

Alexey Svyatkovskiy[*], Sebastian Lee[◇†], Anna Hadjitofi[◇‡],
Maik Riechert[§], Juliana Vicente Franco[§] and Miltiadis Allamanis[§]
[*]Microsoft, WA, USA Email: alsvyatk@microsoft.com
[†]University of Oxford, Oxford, UK Email: sebalexlee@gmail.com
[‡]Alan Turing Institute, London, UK Email: annahadjitofi@googlemail.com
[§]Microsoft Research, Cambridge, UK Email: {marieche,jufranc,miallama}@microsoft.com

***Abstract*—Code completion is one of the most widely used features of modern integrated development environments (IDEs). While deep learning has made significant progress in the statistical prediction of source code, state-of-the-art neural network models consume hundreds of megabytes of memory, bloating the development environment. We address this in two steps: first we present a modular neural framework for code completion. This allows us to explore the design space and evaluate different techniques. Second, within this framework we design a novel reranking neural completion model that combines static analysis with granular token encodings. The best neural reranking model consumes just 6 MB of RAM, — 19x less than previous models — computes a single completion in 8 ms, and achieves 90% accuracy in its top five suggestions.**

I. Introduction

Deep learning has a substantial impact on software engineering methods across a variety of tasks [1]. One early application of machine learning of source code has been code completion [2, 3, 4], *i.e.* the suggestion of code that a developer is about to type. Code completion is the most frequently used feature in IDEs [5, 6]. Early machine learning methods used feature-based models to predict the next function to be invoked [2, 7]. Other models, such as n-gram language models [3], do not require extraction of features, but instead use the code's tokens and — to some extent — can generalize to new code and APIs [8]. However, n-gram models have a prohibitive memory footprint consuming gigabytes of RAM [9].

Recently, neural machine learning methods have been found to be effective for code completion, achieving state-of-the-art accuracy [9, 10, 11]. Nevertheless, these models commonly have a large memory and computational footprint — consuming many hundreds of megabytes in RAM, which is prohibitive for any single component of a modern IDE with hundreds of features. Furthermore, despite the promising results from neural code completion models, they fail to capture rich information from static analyses (*e.g.* type information) that is directly useful to code completion.

In this work, we tackle these limitations by reformulating neural code completion from *generation* to *ranking*. We achieve this by taking advantage of the candidate completion suggestions generated by pre-existing static analyses. This significantly improves the predictive accuracy compared to baseline models and enables us to use fine-level encodings of code tokens, which reduce or completely remove the need for maintaining a memory-intensive vocabulary and embedding matrix, while achieving good accuracy trade-offs. We show that we can create efficient neural code completion models that consume just 6 MB of RAM — 19× less than the baselines — and execute in a few milliseconds while still achieving 90% accuracy in their top five suggestions for API completion. Such systems can support a wide variety of developer environments, including those that are significantly resource-constrained, and avoid introducing bloat to editors and IDEs. This is important as we seek to provide inclusive solutions to developers — including those that do not have access to high-end machines or Internet connections.

In brief, (a) we present a modular neural framework for code completion that allows us to explore a wide range of neural components that offer varying trade-offs in terms of memory, speed and accuracy (section III); (b) within this framework, we present a novel reranking neural architecture that combines static analysis and granular neural token encodings combining the best of static analysis-based code completion and neural-based code completion (section III); (c) we implement our framework for API completion and present an extensive and principled evaluation over the design space (section IV), testing 64 model configurations with multiple hyperparameter settings spending more than one GPU-year of computational effort. Our evaluation shows that our novel neural reranking models that have a memory footprint as little as 6 MB can achieve 90% completion accuracy on their top five suggestions in less than 8 ms.

II. Approach

We use the synthetic snippet in Figure 2 as a running example. A developer is currently declaring the `array_inner_product` variable. We call the current location of the cursor (shown in red) the *completion location*, which is where a code completion tool has the opportunity to serve *completion suggestions*. Commonly, an IDE will perform a static analysis (Figure 1; left) to determine the type of `array1` and return the list of *candidate completions* that are valid at the given location and the given *completion context* (declared variables, imported packages, *etc.*). Traditionally, IDEs did not employ learned components and instead yielded an alphabetically sorted list of type-correct suggestions. In this

[◇] Work performed as AI Residents in MSR Cambridge, UK.

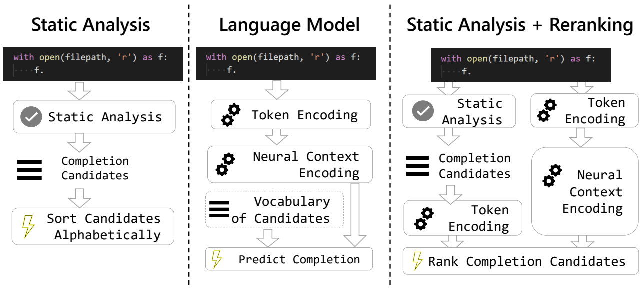


Fig. 1: Static Analysis-based code completion (left), Language model-based code completion (center), Neural rerank models with static analysis (this work; right).

```
import jax

array1 = jax.numpy.random.rand(10)
array2 = jax.numpy.random.rand(10)
array_inner_product = array1.
```

Candidate Completion	**Targets**
all	choose
any	clip
argmax	compress
argmin	conj
astype	...
base	**dot**
byteswap	...

Fig. 2: Motivating Example. A developer is using `jax` and is currently declaring `array_inner_product` using the `jax` API (left; cursor in red). IDEs completion tools will ask a completion system to yield a ranked list of *candidate completion targets* at the current *completion location* which are then shown to the user (right).

case, the list of suggestions is quite long (Figure 2; right). One approach, used by Bruch et al. [2], Proksch et al. [7], is to extract hard-coded features from the completion context and learn which candidate completion targets are relevant in the given context. This approach, however, misses the opportunity to learn richer features directly from data — an approach that has recently been made possible thanks to deep learning. For example, the name of the variable `array_inner_product` indicates that the developer is about to invoke the `dot` method to compute the inner product. Manually anticipating and extracting relevant features that cover as many cases is difficult. Neural methods aim to address this.

Furthermore, such approaches learn about individual APIs and cannot generalize to unseen ones. This requires sufficient example usages of an API to *learn* about those features. However, in many cases, this is not possible. In our example, `jax` is a relatively new machine learning library[1] that is under active development and relatively few public codebases used it. Most existing approaches to code completion would not generalize to `jax` or other previously unseen completion targets since they require sufficient training data of the API usage within real code.

Neural models alleviate this as they can generalize better to previously unseen code by automatically learning the aspects that are relevant for a given completion context. Such features include the structure of the code, but also the names of the variables and functions appearing in the context. For `jax` (Figure 2), which is built to match `numpy` in some aspects, neural models can recognize similarities in the numeric manipulations in Figure 2. An important source of information is contained in the names within the completion context (*e.g.* the subtokens in `array_inner_product` of Figure 2). Early models [3, 8, 12, 13, 14, 15] did *not* take into account the structure within names, which nevertheless contains valuable information. Only recently, Karampatsis et al. [9] introduced such techniques in code completion. To take this idea a step further, we devise a framework (section III) that allows us to test multiple techniques for learning from the internal structure of names and show how these techniques provide different trade-offs in terms of completion accuracy, memory requirements and computational cost.

Furthermore, all existing neural models treat code completion as a language modeling problem, *i.e.* the models are tasked with *generating* the full target completion from their internal knowledge (Figure 1; center). Considering the diverse nature of the completion targets, this is an unnecessarily hard task, since the language model needs to be aware of all valid completion targets (*e.g.* all the candidate completion targets in Figure 2), or be able to reconstruct them from scratch. Within our neural framework, we present a novel model that treats the neural code completion problem as the problem of *(re)ranking* the candidate completion targets returned from a static analysis (Figure 1; right). This improves the existing state-of-the-art allowing us to build and deploy memory-efficient code completion tools without sacrificing accuracy.

III. A Neural Code Completion Framework

First, we present a general neural framework (Figure 3). Designing neural networks for a specific task is an engineering effort that commonly involves combining different components (also referred to as "modules") such that the neural network achieves sufficient accuracy while complying to other non-functional requirements, such as computational speed and memory consumption. Here, we design a framework that allows us to perform a principled exploration of a series of design decisions that can help us pick a good trade-off among the desired properties of a practical completion system. Our framework distinguishes the following components (Figure 3):

Token Encoder A neural network $\mathcal{E}$ encoding a code token t into a distributed vector representation (embedding) $\boldsymbol{r}_t$.

Context Encoder A neural network $\mathcal{C}$ that encodes (*i.e.* summarizes) the completion context $\boldsymbol{t}_{\mathrm{cx}}$, into a distributed vector representation (embedding) $\boldsymbol{c}_{\mathrm{cx}}$.

Candidate Provider $\mathcal{P}$ A component that accepts the completion context $\boldsymbol{t}_{\mathrm{cx}}$ and yields an (unordered) set of M candidate completion targets s_i, *i.e.* $\mathcal{P}(\boldsymbol{t}_{\mathrm{cx}}) = \{s_i\} = \{s_0, ..., s_M\}$.

Completion Ranker A neural component $\mathcal{R}$ that accepts the context encoding $\boldsymbol{c}_{\mathrm{cx}}$, along with a set of candidate completion targets $\{s_i\}$, and ranks them.

[1] At the time of writing, `jax` is at version 0.2.x indicating that it will rapidly evolve, potentially introducing new APIs and breaking old ones. For such APIs the available data will be scarce.

The components, their inputs/outputs, and concrete implementations are shown in Table I. We denote a concrete configuration with a tuple of the form $\langle \mathcal{E}, \mathcal{C}, \mathcal{P} \rangle$. For example, $\langle$SUBTOKEN, GRU, STAN$\rangle$ is an instantiation of the framework with a SUBTOKEN token encoder $\mathcal{E}$, a GRU context encoder $\mathcal{C}$ and a STAN completion provider $\mathcal{P}$. Our architecture is general and subsumes most neural language models [8, 9, 10, 16]. By selecting the concrete implementation of the components, we retrieve a range of neural code completion architectures. To restrict the explored space, we fix two aspects. Following current state-of-the-art models [9], we treat the completion context as a list of the N tokens before the completion location, *i.e.* $\boldsymbol{t}_{\mathrm{cx}} = [t_0, ..., t_{N-1}]$. For example, the last four elements of the completion context of Figure 2 are [`array_inner_product`, =, `array1`, `.`]. Although other representations of code have been explored [1] and could be used in our framework, token-based models have been shown to achieve good performance in code completion [9], with a small computational cost for extracting information from the code context. For example, the graph representations of Allamanis et al. [17] contains significantly more information, but the computational cost of processing these structures is prohibitive for real-time systems. We thus pass to the context encoder $\mathcal{C}$ the context $\boldsymbol{t}_{\mathrm{cx}}$ and the token encoder $\mathcal{E}$. The second design decision is to tie the token encoder within $\mathcal{C}$ and $\mathcal{R}$. This is a common architectural choice in deep learning and Inan et al. [18] showed that such an approach is theoretically principled and yields improved results, while reducing memory requirements. In the rest of the section, we discuss concrete implementations for each component.

A. Token Encoders $\mathcal{E}$

There is a wide literature about encoding tokens in source code and text. A key characteristic of source code tokens is that they tend to be extremely sparse, combining different words (commonly called “subtokens”) to create a single code token [12]. For example `array_inner_product` is a very rare name, but is made of three more common subtokens. We consider four commonly employed token encoders. All of the presented token encoders have been used in some form in previous works or in natural language processing and offer alternative trade-offs in terms of their ability to represent tokens, memory requirements, and computational cost.

Token-Unit Encoder (TOKEN) The simplest and most commonly used encoder that we consider is a token-unit encoder. TOKEN learns an embedding of dimension D for each token in a fixed vocabulary V_t. This requires learning and storing an embedding matrix with $|V_t| \times D$ parameters. TOKEN then performs a lookup, *i.e.*

$$\mathcal{E}_{\text{TOKEN}}(t) = \text{EMBEDDINGLOOKUP}\left(t, V_t\right), \tag{1}$$

where EMBEDDINGLOOKUP(t, V_t) returns the D-dimensional row of the embedding matrix that corresponds to t. If the lookup fails, then the learned embedding of a special unknown identifier (“UNK”) is returned. The vocabulary V_t is selected from the training data and contains the most frequent tokens and the UNK symbol. The size of the vocabulary $|V_t|$ is a hyperparameter that needs to be tuned: smaller vocabularies reduce memory requirement at the cost of failing to represent many tokens and thus yielding less accurate suggestions. Commonly, TOKEN has a large number of parameters: for practical vocabulary sizes and sufficiently expressive embedding dimension D, the number of parameters is in the order of 10^7, which is orders of magnitude more than the number of parameters typically required for context encoders and amounts to many megabytes which need to be stored in the RAM.

Subtoken Encoder (SUBTOKEN) Source code identifiers are often made up of smaller parts. For example, `array_inner_product` is made up of three subtokens (`array`, `inner`, `product`). SUBTOKEN learns to *compose* the meaning of an identifier from its subtokens into a single encoding and that allows to better capture the sparse nature of identifiers while simultaneously reducing the memory requirements compared to $\mathcal{E}$. The SUBTOKEN encoder sub-tokenizes identifiers deterministically by splitting on camelCase and pascal_case, lower-casing each subtoken, and using an embedding matrix with size $|V_s| \times D$, where V_s is the subtoken “vocabulary”. Since subtokens are less sparse than tokens, $|V_s|$ can be much smaller than $|V_t|$, and thus SUBTOKEN can afford a smaller embedding matrix. Obtaining a representation of a token t requires composing the representation from the subtoken embeddings that constitute the token, *i.e.*

$$\mathcal{E}_{\text{SUBTOKEN}}(t) = \bigoplus_{t_s \in \text{SPLIT}(t)} \text{EMBEDDINGLOOKUP}\left(t_s, V_s\right), \tag{2}$$

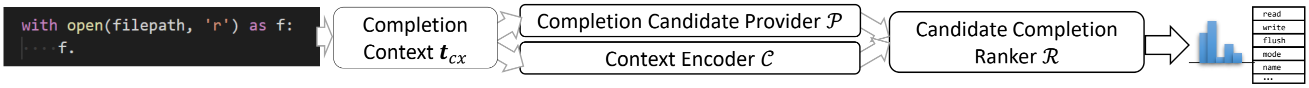
where EMBEDDINGLOOKUP$(\cdot)$ is defined analogously to the word-level case, SPLIT() is a function that subtokenizes its input and returns a set of subtokens, and $\oplus$ is an aggregation operator that “summarizes” the meaning of a single token from its subtokens. We tested three options for $\oplus$: element-wise summation, average, and maximum. In early experiments, all operators achieved comparable performance; for conciseness we only report results for the maximum operator.

BPE-Based Encoder (BPE) Byte-pair encoding is a method commonly used in natural language processing for dealing with rare words in an adaptive way [19] and has its origins in data compression. Specifically, BPE uses a preprocessing step to “learn” subtokens by combining commonly occurring consecutive characters. Then each token is represented as a sequence of those subtokens. Note that the way that a token is split depends on the training data and is “learned” during preprocessing. For example, a BPE splits `array_inner_product` into `array`, _, `in`, `ner`, _, `prod`, `uct`. Our BPE encoder is identical to SUBTOKEN but replaces SPLIT in Equation 2 with the BPE-based splitting.

Character-Based Encoder (CHAR) Finally, we consider a character-level encoder. CHAR composes a representation of a token from its individual characters. The primary benefits of CHAR is that commonly its number of parameters is

Component	Signature	Returns	Implementations
Token Encoder	$\mathcal{E}(t)$	Token Embedding $\boldsymbol{r}_t \in \mathbb{R}^D$	Token, Subtoken, bpe, Char
Context Encoder	$\mathcal{C}(\boldsymbol{t}_{\text{cx}}, \mathcal{E})$	Context Embedding $\boldsymbol{c}_{\text{cx}} \in \mathbb{R}^H$	Gru, biGru, Transformer, Cnn, and annotated ($\diamond$) variants
Candidate Provider	$\mathcal{P}(\boldsymbol{t}_{\text{cx}})$	Candidate Completions Set $\{s_i\}$	Vocab, StAn
Completion Ranker	$\mathcal{R}(\mathcal{E}, \{s_i\}, \boldsymbol{c}_{\text{cx}})$	Ranked suggestions by $P(s_i \vert \boldsymbol{c}_{\text{cx}})$	Dot

TABLE I: Components of the framework in Figure 3. By combining these components, we retrieve a concrete neural code completion model system. We denote a specific architecture using a tuple, *e.g.* $\langle$Subtoken, Gru, StAn$\rangle$ is a model with a static analysis-based candidate provider $\mathcal{P}$, a subtoken-based token encoder $\mathcal{E}$, and an RNN-based context encoder $\mathcal{C}$.



```
with open(filepath, 'r') as f:
    f.
```

Completion Context $\boldsymbol{t}_{cx}$
Completion Candidate Provider $\mathcal{P}$
Context Encoder $\mathcal{C}$
Candidate Completion Ranker $\mathcal{R}$
read
write
flush
mode
name
...

Fig. 3: Our framework for machine learning-based code completion systems. From the developer environment, the (partial) code context $\boldsymbol{t}_{\text{cx}}$ is extracted and passed into the context encoder, $\mathcal{C}$, that "summarizes" the completion context. Simultaneously, a candidate provider $\mathcal{P}$ receives the code context and computes a set of candidate completions for the current location. Finally, a completion ranker $\mathcal{R}$ ranks the candidates given the summarized context and presents them to the user. In this work, we instantiate each component with various types of neural networks (Table I), showing how different choices affect suggestion accuracy, model size and computational cost. All neural models are trained end-to-end.

significantly smaller compared to other encoders and that it can represent arbitrary tokens as long as they are made from known characters. Representations of tokens are then *computed* without involving a lookup like in the encoders previously discussed. Thus, Char stores *only* the parameters of the network and has no vocabulary. The trade-off is that such networks commonly have a smaller representational capacity and are slightly more computationally expensive compared to the other encoders. For a token t,

$$\mathcal{E}_{\text{Char}}(t) = \text{1dCnn}(\text{GetChars}(t)), \tag{3}$$

where 1dCnn is a 1D convolutional neural network (CNN) and GetChars splits t into a list of characters. The 1dCnn is similar to the NLP work of Zhang et al. [20] and Kim et al. [21]. We define an alphabet of commonly used characters in our data. Each token is then represented as a matrix of one-hot columns in which the element corresponding to the relevant index in the alphabet is set to 1.

B. Context Encoders $\mathcal{C}$

Context encoders (Figure 3, Table I) are responsible for taking the completion context and encoding all information that is relevant for the current completion location into a single vector representation $\boldsymbol{c}_{\text{cx}}$. Again, there is a large set of design options. For computational efficiency we only consider context encoders of the form

$$\boldsymbol{c}_{\text{cx}} = \mathcal{C}\left(\boldsymbol{t}_{\text{cx}}, \mathcal{E}\right) = \mathcal{C}\left(\mathcal{E}(t_0), ..., \mathcal{E}(t_{N-1})\right), \tag{4}$$

i.e. context encoders that accept as input the N context tokens before the completion location and a token encoder. The output $\boldsymbol{c}_{\text{cx}}$ is an H-dimensional vector, where H is a hyperparameter. One benefit of encoders of this form at test-time is that the token encodings can be cached.

RNN-based Context Encoders (Gru) Recurrent neural networks (RNN) are a common neural module that summarize variable-length sequences. RNN encoders take the form

$$\boldsymbol{h}_\nu = \text{RnnCell}\left(\boldsymbol{h}_{\nu-1}, \mathcal{E}(t_{\nu-1})\right), \tag{5}$$

where $\boldsymbol{h}_\nu$ is the vector state at position ν, $\mathcal{E}(t_\nu)$ is the encoding of the input at time ν and RnnCell is a learnable function. We test two commonly used RnnCells, LSTMs [22] and GRUs [23], but given their similar performance, we only report results for GRUs. The output of the Gru context encoder is $\mathcal{C}_{\text{Gru}}(\boldsymbol{t}_{\text{cx}}, \mathcal{E}) = \boldsymbol{h}_N$. We also test a bi-directional GRU which we denote as biGru.

CNN Context Encoders (Cnn) Similar to the Char token encoder, 1D CNNs can be used to encode the context into a vector representation. There is a multitude of CNN configurations that are applicable to our setting. All of them accept as an input an $N \times D$ matrix, and after a few layers of convolution and pooling, a pooling layer is used to compute the final context representation $\boldsymbol{c}_{\text{cx}}$. This architecture has resemblance to the natural language classification of Kim [24].

Transformer Context Encoders (Transformer) An alternative to RNN-based and CNN-based sequence models are transformers [25] which have recently shown exceptional performance in natural language processing Devlin et al. [26], Radford et al. [27]. Although transformers can be parallelized efficiently, they have quadratic runtime memory requirements with respect to the sequence length. Here, we employ the standard transformer encoder architecture.

Completion Location-Annotated Encoders We can provide additional information to any of the above context encoders $\mathcal{C}$, *e.g.* information derived from analyzing the code. Here, we test adding lightweight information that is useful for API completion. Specifically, we annotate all occurrences of the variable or namespace on which an API completion is performed. For example, in Figure 2 we indicate to the context encoders all the tokens that refer to the object `array1`. This may allow a context encoder to better recognize API patterns, *e.g.* if `foo.open()` was previously invoked then invoking `read()` on `foo` is likely. Other annotations are also possible, but we do *not* test them in this work.

To capture this long-range context, we simply wrap the token encoder of Equation 4 such that it appends a 0/1 component to each token encoding $\boldsymbol{r}_t$. This bit is set to one for the tokens that are bound to the variable or namespace whose API is about to be completed. This provides additional information to the context encoders at a minimal cost. We denote such encoders by appending a diamond ($\diamond$) to their name, *e.g.* GRU$^\diamond$.

C. Candidate Providers $\mathcal{P}$

We consider two types of candidate providers. VOCAB providers — commonly used in language model-based completion engines — have a fixed list (vocabulary) of candidate completions. The vocabulary is compiled from the training data by taking the top most frequent target completions up to some size $\mathbb{V}_{max}$, which is a model hyperparameter. Commonly, the vocabulary is identical to V_t defined by the TOKEN encoder. The second candidate provider is a static analysis-based provider (STAN). Such providers are common in both typed and untyped languages where a static analysis tool can determine a set of plausible completions. For example, IntelliSense [28], PyCharm [29] and IntelliJ [30] return only all type-correct completions that can be used at a suggestion location.

These two providers offer different trade-offs: STAN providers yield much more precise and informative candidate completion targets compared to VOCAB providers. Such candidate providers are preferred by most IDEs [10] since they do *not* risk making suggestions that are invalid at the completion location which may confuse developers. Nevertheless, VOCAB providers can function in partial or incomplete contexts where static analyses cannot yield informative results. Although using a predefined vocabulary of completions simplifies the machine learning model, it does not allow for the model to provide candidates beyond those seen in the training data. This is a major limitation for suggesting and generalizing to rare, evolved or previously unseen APIs, such as the `jax` API in Figure 2. Although a VOCAB provider can use a static analysis-based post-processing step to remove some false positives it cannot generalize beyond its fixed vocabulary.

Other Candidate Providers Beyond VOCAB and STAN, other candidate providers can be used. One commonly used case is "structural prediction" providers that learn to predict completion targets by composing them from individual subunits, such as subtokens [31], BPE [9] or characters, allowing for an unbounded set of possible candidates. This is currently used in state-of-the-art work of Karampatsis et al. [9] and Aye and Kaiser [11]. We do not test such providers since STAN-based providers have access to strictly more information and thus will always perform better in comparison with structural prediction providers. Furthermore, the generation of a full candidate completion target requires multiple steps (*e.g.* beam search which needs one step per subtoken/character) which imposes additional computational burden and makes the generation of invalid completions more probable.

D. Completion Ranker $\mathcal{R}$

We test a single, commonly used, target completion candidate ranker, DOT. DOT ranks a set of candidate completion targets, $\{s_i\}$, according to the probability distribution

$$P(s_k|\boldsymbol{c}_{\mathrm{cx}}, \{s_i\}, \mathcal{E}) = \frac{\exp\left((W\boldsymbol{c}_{\mathrm{cx}})^\top \mathcal{E}(s_k) + b_{s_k}\right)}{\sum_{s_j \in \{s_i\}} \exp\left((W\boldsymbol{c}_{\mathrm{cx}})^\top \mathcal{E}(s_j) + b_{s_j}\right)}, \tag{6}$$

i.e. the standard softmax equation over the dot product of the token encodings of candidate suggestions with a linearly transformed context encoding $\boldsymbol{c}_{\mathrm{cx}}$. Here, W is a linear layer of size $H \times D$, which is learned along with the rest of the model parameters, and linearly maps the H-dimensional vector into a D-dimensional one. The vector $\boldsymbol{b}$ is a learned bias signifying the "popularity" of a given method independent from its context. Since we want our model to generalize to APIs that were previously unseen, we set $\boldsymbol{b} = \boldsymbol{0}$ for all non-VOCAB-based models.

When Equation 6 is used in conjunction with a VOCAB-based candidate provider and a TOKEN encoder, Equation 6 reduces to a standard token-level language model. However, when the candidate provider yields a context-dependent, variable set of candidates, (as all STAN models do) Equation 6 becomes a ranking model of the candidates in $\{s_i\}$. As we will explore in the next sections ranking a smaller set of valid candidates is a simpler problem than predicting a target from a large vocabulary, which leads to improved performance with better memory footprint.

E. Composing Components: Model Zoo

Table I summarizes the components and implementations discussed. To create a full code completion system, one needs to pick an implementation for each component, and instantiate a single neural network. This leads to 64 model combinations with varying accuracy, memory requirements, and computational cost. Additionally, each component has its own hyperparameters, yielding a large search space. For a given configuration and hyperparameters these neural networks are trained by jointly optimizing all their parameters (embedding matrices, layer weights, *etc.*) end-to-end with the objective to minimize the cross entropy loss of Equation 6. In our training, we additionally employ early stopping and set the context size to $N = 80$ tokens, which yields a reasonable trade-off between the amount of context and computation/memory concerns.

IV. EVALUATION

In the previous section, we discussed a general framework for neural models of code completion. To evaluate the multitude of configurations, we focus on a particular instance of code completion, API completion. This is the suggestion of method invocations and field accesses that a developer will use at a given invocation site. In many languages such as Python, Java, and C# an API member of a receiver object or namespace `foo` is accessed by using a single dot. For example, a developer will write `np.array` to access the `array` API from the popular `numpy` package. IDEs will commonly offer a list of candidate suggestions when a developer presses the "`.`" key. Although this is just one form of code completion it is one of the most valuable and one of the hardest to predict [32]. In this section, we focus on this one. First, we explore how various architectural decisions affect code completion models. Then, we contrast our models to those of other publications, and finally discuss how the best models generalize to new APIs. Note that our framework and models are more general; whenever a candidate provider $\mathcal{P}$ for a given completion context can be provided, our approach will be applicable.

To evaluate the API completion performance, we follow the established evaluation methodology within this area [9]: we assume that code is written sequentially and from left to right and measure if the models predict the method that the user intended to invoke, directly reflecting the use case of code completion systems. It is known that such an "offline" evaluation does not fully reflect real-life ("online") code completion setting [32]. However, thanks to extensive data collection within Facebook, Aye et al. [16] empirically observe a strong correlation between online and offline settings: although the accuracy of online completions is lower than those predicted offline, models that perform better offline, also perform better in the online setting. This is an encouraging result for research where due to privacy and confidentiality it is unlikely that we can monitor code completions of realistic and diverse developer environments at scale across different organizations. We also deploy our code completion system to a small set of about 2k users and reconfirm these observations.

Data The training, validation and test data used for the experiments came from the 2700 top-starred Python source code repositories on GitHub. Scraped datasets commonly contain a large amount of duplicates [33, 34], which may skew the evaluation results. To avoid this, we deduplicate our dataset using the tool of Allamanis [34]. Our dataset contains libraries from a diverse domains including scientific computing, machine learning, dataflow programming, and web development. To prepare the data, we use the Visual Studio Code type inference engine (PTVS) to collect all completion locations where an API completion suggestion would be emitted by PTVS, similar to Svyatkovskiy et al. [10]. We extract the previous $N = 80$ tokens preceding the method invocation ($\boldsymbol{t}_{\text{cx}}$), and information about the receiver object or namespace, which we use to create a Python STAN candidate completion provider. The final dataset contains 255k files and a total of 7.4 million API completion instances. We split the data per-file into training-validation-test at 60-20-20 proportions.

Evaluation Metrics We measure three aspects: accuracy, model size and suggestion speed that directly relate to the user experience of code completion. To measure *model size*, we compute the total number of parameters of each neural model. This number correlates well with the RAM consumption across machines and is not affected by noise (*e.g.* garbage collection). Given that every parameter is a `float32` we can compute the (uncompressed) size of the parameters of each neural model. This approach is similar in nature to the one used by Proksch et al. [7].

To measure the *computational cost* per-suggestion, we compute the average time needed for our neural network to compute a single suggestion on a CPU. Although we train our models on a GPU we cannot expect that the developer environment has GPU hardware and therefore only CPU time is relevant. Furthermore, to match a realistic running environment, we do *not* batch the computation of suggestions when calculating these statistics. Note here that the measurement is performed directly with the PyTorch code. In practice, the neural network computation would be statically compiled (*e.g.* via ONNX, TorchScript, TFX). Again, we expect such methods to yield similar improvements across configurations and do not test them. Note that this time excludes any computation needed for a static analysis, which is orthogonal to our models and is commonly amortized across editing time. The experience and user studies of the Visual Studio product team suggests that the computational budget should be at most a few tens of milliseconds even on relatively old machines [10].

Finally, we are interested in evaluating the *predictive accuracy* of each model, *i.e.* how well each completion model ranks the intended candidate higher. This directly relates to the user experience and the usefulness of any code completion system: the higher a relevant result is ranked, the better the accuracy of the model. We use two well established metrics. First, "recall at top k"[2] (denoted as "Recall@k") measures the proportion of examples where the correct completion is in the top k suggestions. We report $k = 1$ and $k = 5$ as we do not expect users to look at the suggestions beyond that point [35]. We also report the mean reciprocal rank (MRR), which is commonly used for evaluating ranking methods. MRR takes values in $[0, 1]$, with 1 being the best score, and is defined as $\frac{1}{N}\sum_{i=1}^{N}\frac{1}{r_i}$ where r_i is the rank of each target suggestion.

Each of our experiments runs on a virtual machine equipped with one NVIDIA Tesla V100 GPU and an Intel Xeon E5-2690 v4 (Broadwell) CPU. The results presented in this section stem from more than 1 year of a single GPU-time (across multiple machines). It should be noted that various techniques, such as model quantization [36], can be applied. However, these methods commonly provide similar speed-ups and memory reductions across all models retaining their relative ordering. We thus ignore these techniques at this stage.

A. Multitarget Evaluation

Although predictive performance is an important factor, it is important to consider the memory and computational cost trade-offs to offer the best experience to developers. These factors have been mostly overlooked by the current literature.

[2] The terminology derives from information retrieval domain. Some papers refer to this metric as "accuracy at top k" instead.

Improving on the last two quantities usually reduces predictive accuracy. Therefore, we treat the evaluation as multitarget.

To achieve this, we run a search across multiple model configurations and hyperparameters[3]. These — among other parameters — include the size of the vocabularies (V_t for TOKEN and V_s for SUBTOKEN), the context encoder hidden dimension H, and the size D of the token embeddings which have the biggest effect on a model's size. Reporting the results for all configurations would significantly reduce the clarity of our evaluation. Therefore, we gradually bisect the design space and discuss the results. Figure 4 plots the Pareto fronts of the multitarget evaluation across our evaluation metrics for some of the model configurations, as computed by our search. Each line represents the Pareto optimal options for a given configuration. Table II shows the evaluation metrics for a selected subset of model configurations. For each configuration, we present the metrics of the best models that are approximately 3 MB and 50 MB in size. We pick these sizes because they represent two realistic points. A 3 MB model can be deployed even in severely restricted environments (*e.g.* low-end hardware used to teach students to code). A 50 MB model is a reasonable upper bound size for a plugin in a modern IDE or editor. Note that although most modern computers have a few gigabytes of RAM, modern IDEs — such as Visual Studio Code — contain a number of components which individually amount to a few megabytes of RAM but collectively can consume a significant proportion of a computer's RAM. Real-life experience from the Visual Studio teams deploying IDEs to thousands of users, dictates that the memory consumption of code completion systems be as minimal as realistically possible.

We additionally compare our models to a simple baseline ("Popularity") that yields the most frequently used target completion for a given API in our training set. This can be thought as a non-neural $\langle$TOKEN, $\varnothing$, STAN$\rangle$ model. The predictive performance of the baseline is worse than all neural models, although it is faster and smaller in size (Table II). This is unsurprising, since it does *not* take into account any context information beyond what is available to the STAN provider.

STAN vs. VOCAB Table II shows that VOCAB-based models underperform significantly to our novel STAN-based models. The VOCAB models are similar to the language models presented by Hellendoorn and Devanbu [8] except that our models are trained only on API completion locations, and not for predicting arbitrary code tokens. We do *not* plot the Pareto fronts of VOCAB models in Figure 4a since they are so much worse that would require increasing the scale of the plot. This is not surprising since STAN models have strictly more information. All VOCAB-based models have multiple shortcomings. First, they need to predict the target completion from a long list of plausible candidates, without any explicit knowledge of the correct choices, and thus have multiple opportunities for making errors: making a correct choice from a longer list is harder than from a short list. Moreover, VOCAB-based models are larger and slower. For a VOCAB-based model to have good recall, its vocabulary must be sufficiently large to contain the majority of suggestions it may need to make. In turn, a large vocabulary implies a large embedding matrix, which substantially increases memory requirements. The relatively slow speed of this model stems from the need to compute Equation 6 over the whole vocabulary V_t, whereas STAN-based models can compute it over only a smaller set of candidate completions. These observations are common across all VOCAB models, even those not explicitly presented here. For this reason we will not consider them any further.

***Token Encoders* $\mathcal{E}$** Having established that STAN-based models are preferable to VOCAB-based models, we now discuss the different token encoders using the STAN candidate provider and the GRU context encoder. The performance differences are small, but generally the TOKEN models perform worse. The difference is more pronounced for models with a smaller vocabulary, which fail to represent the sparsity of code tokens. In subsection IV-C we show that TOKEN-level encoders also tend to generalize worse compared to other token encoders. The SUBTOKEN, BPE, and CHAR provide competitive results at a higher computational cost (needed for composing the representation of each token). CHAR models perform best for small model sizes ($\leq$1 MB), however with increased model capacity, they fail to scale up. In light of these results, we consider SUBTOKEN or BPE based models to be reasonable options. These observations about BPE are consistent with those of Karampatsis et al. [9] and Aye and Kaiser [11].

***Context Encoders* $\mathcal{C}$** Finally, we turn our attention to the context encoders. We select the SUBTOKEN encoder as it provides a reasonable trade-off between memory consumption and predictive performance. Both CNN and TRANSFORMER underperform compared to GRU encoders. Additionally, TRANSFORMER-based models are significantly more computationally expensive. BIGRU encoders improve marginally over GRU, but at an increased computational cost. Finally, completion-location annotated ($\diamond$) context encoders variants provide a small but consistent improvement thanks to the additional information.

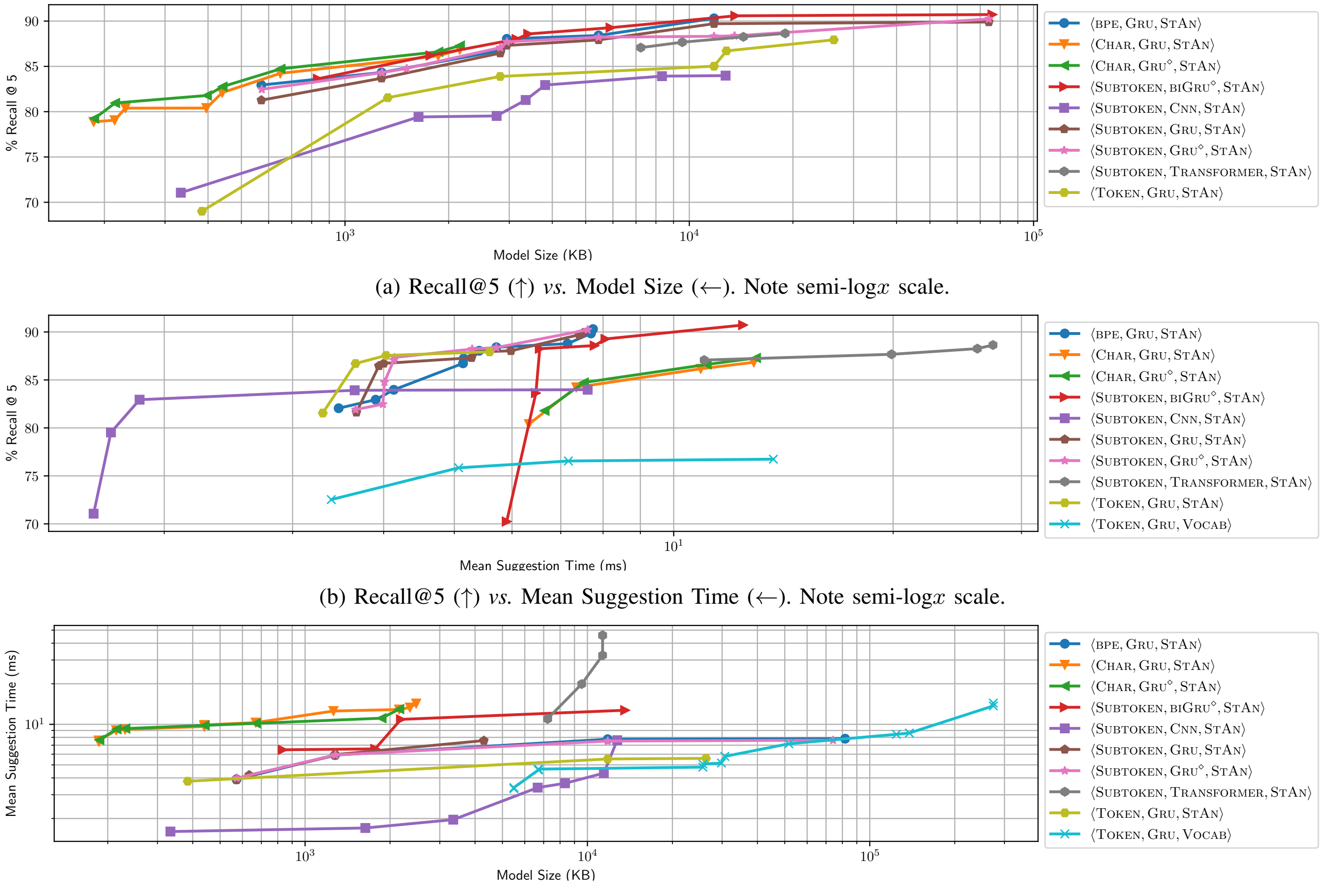
Model Size* vs. *Computational Cost Figure 4c shows the log-log Pareto fronts between the computational time needed for computing a single suggestion *vs.* the model size. As a general principle, the larger a model (*i.e.* more parameters it contains) the slower the computation of the completion suggestions. Furthermore, different configurations have different scaling behaviors. TRANSFORMER-based models are — unsurprisingly — the slowest, whereas the model size of other models seems to have a smaller but noticeable effect. CHAR-based models tend to be slower compared to SUBTOKEN or TOKEN-based models, since they trade-off memory with computation. Overall, most models make predictions in under 20 ms which makes them eligible for real-time code completion systems.

Comparison with Models from Other Publications We

[3] We use an automatic hyperparameter tuning method to perform a random search over ranges of all hyperparameters. For some hyperparameters, such as learning rate, the search is uniform random at a log scale.

TABLE II: Detailed evaluation for a selected subset of model configurations. We show results for the best performing model closest to two model sizes. Computational time ranges denote standard deviation. For some configurations, there is no model of ∼ 50 MB that outperforms a model of ∼ 3 MB. This is denoted with a dash (—).

$\mathcal{E}$	$\mathcal{C}$	$\mathcal{P}$	Best for size ∼3 MB				Best for size ∼50 MB			
			Recall@1	Recall@5	MRR	Time (ms)	Recall@1	Recall@5	MRR	Time (ms)
Most Frequently Used (Popularity)			41.75	72.04	0.5470	0.02 ± 0.01	—	—	—	—
Token	Gru	Vocab	53.01	72.53	0.6140	3.39 ± 1.00	55.87	76.55	0.6477	7.17 ± 1.43
Token	Transformer	Vocab	24.16	40.52	0.3103	10.46 ± 3.00	55.48	74.26	0.6354	36.73 ± 5.01
Token	Gru	StAn	63.78	83.89	0.7245	3.71 ± 0.98	68.78	87.93	0.7703	5.59 ± 1.28
Subtoken	Gru	StAn	63.40	87.31	0.7369	5.28 ± 1.13	67.98	89.90	0.7744	7.51 ± 1.78
BPE	Gru	StAn	**66.35**	88.04	**0.7567**	5.41 ± 1.50	**70.09**	89.84	**0.7861**	7.70 ± 1.85
Char	Gru	StAn	64.30	86.84	0.7396	12.88 ± 3.25	—	—	—	—
Subtoken	Gru ◇	StAn	63.76	87.71	0.7408	5.40 ± 1.23	66.57	90.23	0.7681	7.63 ± 1.97
Subtoken	biGru ◇	StAn	64.25	**88.58**	0.7428	7.79 ± 1.37	67.17	**90.58**	0.7736	12.72 ± 2.42
Subtoken	Cnn	StAn	55.66	81.29	0.6652	1.96 ± 0.89	57.19	83.98	0.6834	7.62 ± 1.77
Subtoken	Transformer	StAn	61.81	87.07	0.7241	11.01± 2.18	65.36	87.36	0.7504	25.85± 3.91



(a) Recall@5 (↑) *vs.* Model Size (←). Note semi-log$x$ scale.

(b) Recall@5 (↑) *vs.* Mean Suggestion Time (←). Note semi-log$x$ scale.

(c) Mean Suggestion Time (↓) *vs.* Model Size (←). Note log-log scale.

Fig. 4: Pareto fronts (Recall@5 *vs.* mean suggestion time *vs.* model size) across some model configurations over our hyperparameter search. Note some axes are in log-scale. Arrows (↑, ↓, ←, →) denote which direction improves the given axis. When a front stops early, it means that the hyperparameter search did not find settings that were Pareto optimal beyond that point. We do *not* plot the Pareto fronts of Vocab models since they are so much worse that would require increasing the scale of the plot.

select the best 3MB and 50MB $\langle \text{bpe}, \text{Gru}, \text{StAn} \rangle$ as our models of choice which achieve 66% (resp. 70%) recall@1 with about 6ms (resp. 8ms) of average suggestion time. Direct comparison with models discussed in other publications in terms of predictive accuracy is not possible due to differences in datasets and programming languages; none of the preexisting datasets can be used since we cannot run the necessary static analyses on them (for the StAn provider) and often the datasets are not available.

Nevertheless, we can reason from first principles to clearly show why StAn-based models improve upon existing models, while retaining all the advances from the methods of Karampatsis et al. [9] and Aye and Kaiser [11]. First, Karampatsis et al. [9] definitively show that n-gram language models have worse predictive performance than neural models due to their better generalization over unseen (sub)tokens. More importantly, reasonably well-performing n-gram models tend to have unacceptably large sizes that are between 5-8.5GB [9] ($\times 100$ to $\times 1700$ more than our models). The BPE-based neural language model of Karampatsis et al. [9] has a size of 115MB which is about $\times 2$–$\times 38$ larger than our models, while similar to our $\langle \text{bpe}, \text{Gru}, \text{Vocab} \rangle$ models. Aye and Kaiser [11] train a language model similar to Karampatsis et al. [9] with a size of 402.6 MB ($\times 8$–$\times 134$ larger than our models).

As discussed earlier, predictive performance comparison with models with structural prediction providers (*e.g.* the beam search over BPE tokens of Karampatsis et al. [9], Aye and Kaiser [11]) would be pointless. Reasoning from first principles, StAn models have access to strictly more information (from the static analysis) and need to just pick suggestion from a short list of candidates. In contrast, structural providers need to predict a valid API token out of an *unbounded* vocabulary of subtokens. Thus, by construction, StAn-based models have an advantage (access to more information) and will always perform better to similar non-StAn models. Essentially, StAn providers allows us to maintain all the novel advances of BPE-based language models of Karampatsis et al. [9], Aye and Kaiser [11] without any of the disadvantages of the structural prediction. Furthermore, since the target task is simpler in StAn-based models, models with similar predictive performance can be smaller — and thus faster. Finally, structural completion models perform a computationally costly beam search, which further increases the computational cost.

B. Deployment

We deploy our 50MB $(\text{Subtoken}, \text{Gru}, \text{StAn})$ completion model to about 2k users within AnonCompany. The model is converted into ONNX and "glued" to the company's IDE code. Due to strict privacy and confidentiality concerns we *cannot* observe any specifics about the code completions (*e.g.* the code context) but we can measure the performance of the suggestions served. We observe 23k code completion events, with a mean serving time of 20 ms (end-to-end; including the static analysis and UI rendering). Similarly to Aye et al. [16], we observe a reduced predictive performance compared to the one measured in the offline with Recall@5 of 54.20% (compared to 90.23% in the offline experiments). This substantially improves upon the previous non-neural model deployed [10] both in terms of predictive accuracy (+20%) and serving speed. We posit that this is due to differences between the open-source code used during training and the internal code used in Microsoft, along with the differences of partial code to commited code found on GitHub.

C. Generalization to Unseen APIs

So far, we observed the performance of the code completion models when the target completion APIs were seen during training. However, APIs evolve and code completion systems are asked to complete code from previously unseen libraries or new user-defined code. In all these cases, we cannot expect to have training data to train accurate models. Instead, we hope that models generalize.

To test these scenarios, we held out from our dataset API completions for three libraries. Table III shows the results of how the neural completion models generalize to completions of the unseen libraries. Our baseline (bottom line) is a completion system that randomly ranks the candidate code completions. This is a weaker baseline than the "most frequently used" baseline (in subsection IV-A) but is the only reasonable choice for this scenario, since we have no prior information about the APIs. Table III shows that all models, except the Vocab-based ones, perform better than the baselines, which indicates that all neural models generalize to some extent. The bad performance of the Vocab model is expected as it has to generate the names of the target completion candidates from its vocabulary. Many of those names have *not* been previously seen and therefore these models fail to generalize. Additionally, the Token-based model performs consistently poor. This can be attributed to the fact that Token-based models cannot generalize easily to previously unseen tokens. In contrast, the models that encode tokens in a more granular way tend to perform better, although there is not a clear "winner". The observed differences may be attributed to different naming conventions of the tested libraries. The results presented here suggest that our novel StAn-based models with granular bpe token encodings are also preferable from a generalization perspective.

D. Training without a Static Analyzer

To train a StAn-based models, we needed to build a dataset making sure that a static analysis tool is able to resolve the developer environment (*e.g.* libraries, other dependencies). However, getting a static analyzer to run for a sufficiently large codebase is hard and does not easily scale, a known issue in the area of "big code" [1]. Inspired by techniques such as NCE [37], we hypothesize that we may be able to overcome this challenge by using a proxy candidate provider that yields random distractor candidate completion targets. Furthermore, for training efficiency, we can use the target completions of other samples in each minibatch as distractors.

We test this on the setting of the best $\langle \text{bpe}, \text{Gru}, \text{StAn} \rangle$ configuration of approximately 50 MB. The results show some

TABLE III: Performance on unseen libraries for models with size $\sim$ 50 MB.

$\mathcal{E}$	$\mathcal{C}$	$\mathcal{P}$	`jax`		`horovod`		`pyspark`	
			Recall@5	MRR	Recall@5	MRR	Recall@5	MRR
Token	Gru	Vocab	12.04	0.0931	4.35	0.2526	1.60	0.0121
Subtoken	Gru	StAn	56.52	0.4184	**88.82**	**0.6051**	**72.55**	0.5410
bpe	Gru	StAn	**72.55**	**0.5444**	82.30	0.5430	71.76	**0.5574**
Random Choice		StAn	38.80	0.2512	40.37	0.2628	68.17	0.4901

degradation in performance. For example, Recall@5 falls to 85.5% (from 89.9%) and MRR falls to 0.729 (from 0.786). This disproves our hypothesis that a proxy static analyzer can be used. This degradation can be attributed to the fact that model capacity is spent for learning to avoid distractor candidate target completions that will never appear at test time.

Note that there is a threat to the validity to this observation. Our testset contains samples where we could use — at batch — a static analyzer to retrieve the candidate suggestions. Thus, our testset is slightly biased towards completion locations where a static analyzer was available during the dataset construction and provides no indication about the performance at completion locations where a static analyzer needs to be manually configured. However, our testset contains a diverse set of APIs. This gives us sufficient confidence that these results hold more generally.

Removing the Explicit Subtoken Vocabulary So far, we have measured the size of a model by counting the parameters of the neural model. However, there is an additional cost for all non-Char models. We need to store in memory a mapping from the string representations of tokens in the vocabulary to their unique ids. This mapping (commonly implemented as a hash map) is consulted when performing an embedding lookup (*e.g.* in the $\textsc{EmbeddingLookUp}(t, V_t)$ of Equation 1 and Equation 2) and maps each (sub)token into a unique index in the embedding matrix. However, this data structure consumes memory. For example, a Subtoken-model with a vocabulary of 10k subtokens has a hash map that consumes about 1.1 MB of RAM. This additional cost of storing the necessary metadata *cannot* be avoided on bpe-based models where this metadata is necessary or in Vocab-based models where the target completion needs to be generated. However, for non-bpe StAn models we can employ feature hashing to eliminate the hash map.

Feature hashing refers to a common trick for vectorizing features [38, 39] and is a form of dimensionality reduction. The core idea is that a good hash function ϕ can provide a reasonable, deterministic mapping of features to ids. As with all hashing methods, this introduces collisions as multiple features may have the same hash, which nevertheless the machine learning models can learn to overcome to some extent. We use this technique for subtokens in the $\langle\textsc{Subtoken}, \textsc{Gru}, \textsc{StAn}\rangle$ model. For each subtoken, we compute the MD5 hash of its string representation and map it to an integer in $\{0...|V|-1\}$ where $|V|$ is the size of the "vocabulary", *i.e.* $\phi(s) = \textsc{Md5}(s) \mod |V|$. The larger $|V|$, the fewer hash collisions at the cost of a larger embedding matrix. Experimentally, we observe minor differences compared to the original models. Specifically, a model with $|V| = 2500$, sees a reduction of MRR of about 1%, whereas the performance of models with larger $|V|$ remains unchanged. Altogether, the results suggest that we can further reduce the memory consumption of Subtoken models by eliminating the stored hash map, without any impact on predictive performance.

V. Related Work

Our work is related to a large set of literature in natural language processing and machine learning, particularly deep learning. We refer the interested reader to the book of Goodfellow et al. [40] for a detailed treatise of deep learning methods and focus the rest of this section on code completion.

The first to propose learning code completions from data were arguably Bruch et al. [2], who tested three methods: association rule mining, frequency-based models and nearest-neighbor-based methods. That stream of work evolved into the Eclipse Code Recommenders [41] and was — to our knowledge — the first data-driven code completion system to be deployed widely to a popular IDE. However, as of 2019 this project has been discontinued [41]. Following similar principles, Proksch et al. [7] presented a Bayesian network for code completion. Similar to our StAn-based models, both these works focus on code completion within specific contexts, *e.g.* when the developer creates a method invocation. However, in contrast to our work, the methods of Bruch et al. [2] and Proksch et al. [7] rely heavily on manually extracting features that are relevant to the completion task. For example, Proksch et al. [7] extract features such as the direct supertype of the enclosing class, the kind of the definition, *etc.*. In contrast, our models require minimal manual feature extraction and can instead employ information that is readily available using a compiler (lexemes, variable bindings, *etc.*).

Our work is centered around the area of machine learning models for source code [1]. The core principle is to avoid extracting hand-coded features and instead rely on (possibly) structured representation of code (lexemes, ASTs, dataflow, *etc.*) and learn directly a task over those representations. "Feature-less" machine learning models of code completion were first studied by Hindle et al. [3] who create a token-level n-gram language model completion models. This was arguably the first model that performed unconstrained code completion. A variety of language models have since been explored [4, 12, 14]. Further improvements to language models (and therefore completion systems) were made by Tu et al.

[13] who noticed that source code tends to have a localness property, *i.e.* tokens tend to be repeated locally. The authors showed that by introducing a cache-based language model [42], performance could be improved. Hellendoorn and Devanbu [8] then set to test neural models of code along with n-gram language models for the task. Their evaluation showed that carefully tuning an n-gram language model, with a hierarchically scoped cache, outperforms some neural models. Nguyen et al. [43] used static analyses to extract features and build a templated n-gram language model for code completion which nevertheless cannot learn generalize to previous unseen APIs. To our knowledge, none of the n-gram language models has been deployed in a real-life IDE, due to the size constraints.

Deep learning methods are central to "feature-less" models and eliminate the need for hand-coded features across domains (*e.g.* image recognition) and can directly learn from raw data. Within this context, a multitude of deep learning models have been researched for source code. Recently, Karampatsis et al. [9] showed that an appropriately designed neural model that uses byte-pair encoding (BPE) yields superior results to non-deep learning models including those of Hellendoorn and Devanbu [8]. We replicated the advantages of BPE in this work. Despite this, the model of Karampatsis et al. [9] treats completion as a generation problem over BPE subtokens instead of taking advantage of readily available information from static analyses, which simplifies the task and offers improved results. Simultaneously, Svyatkovskiy et al. [10] presented a neural model that corresponds to our $\langle \textsc{Token}, \textsc{Gru}, \textsc{Vocab} \rangle$ model, which our $\langle \textsc{Subtoken}, \textsc{Gru}, \textsc{StAn} \rangle$ model outperform. Although other, more structured, language models of code have been researched [14, 15, 44], none of those have been tested for practical code completion systems. This is because of the complexity of the used code representations: since completion contexts are commonly incomplete (*e.g.* they do not parse), sophisticated methods are required to extract the structured representations needed by these models.

VI. Discussion & Open Challenges

We presented an exploration of the design space of practical neural code completion and showed how to combine different deep learning components to retrieve a range of trade-offs beyond reusing existing neural architectures. Within this framework, we implemented and evaluated a number of neural code completion models aiming to improve their performance characteristics. The subtoken-level static analysis-based models strike the best trade-off among predictive performance, model size and computational speed across the tested models. Such models can be practically deployed in IDEs with an acceptable memory and computation footprint. This is an important step towards providing inclusive tools to all developers, even those that cannot access state-of-the-art equipment.

Future work Speed and memory improvements may also be possible with techniques such as quantization [10, 36] and knowledge distillation [45]. Providing that these techniques do not result in a significant drop in predictive performance, they could be an additional step towards making our neural models even more computationally and memory efficient.

References

[1] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, p. 81, 2018.

[2] M. Bruch, M. Monperrus, and M. Mezini, "Learning from examples to improve code completion systems," in *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2009.

[3] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2012.

[4] 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 Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2013.

[5] S. Amann, S. Proksch, S. Nadi, and M. Mezini, "A study of Visual Studio usage in practice," in *Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2016.

[6] G. C. Murphy, M. Kersten, and L. Findlater, "How are Java software developers using the Eclipse IDE?" *IEEE software*, vol. 23, no. 4, pp. 76–83, 2006.

[7] S. Proksch, J. Lerch, and M. Mezini, "Intelligent code completion with Bayesian networks," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2015.

[8] V. J. Hellendoorn and P. Devanbu, "Are deep neural networks the best choice for modeling source code?" in *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, 2017.

[9] R.-M. Karampatsis, H. Babii, R. Robbes, C. Sutton, and A. Janes, "Big code!= big vocabulary: Open-vocabulary models for source code," in *Proceedings of the International Conference on Software Engineering (ICSE)*, 2020.

[10] A. Svyatkovskiy, Y. Zhao, S. Fu, and N. Sundaresan, "Pythia: AI-assisted code completion system," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2019, pp. 2727–2735.

[11] G. A. Aye and G. E. Kaiser, "Sequence model design for code completion in the modern IDE," *arXiv preprint arXiv:2004.05249*, 2020.

[12] M. Allamanis and C. Sutton, "Mining source code repositories at massive scale using language modeling," in *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, 2013.

[13] Z. Tu, Z. Su, and P. Devanbu, “On the localness of software,” in *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, 2014.

[14] P. Bielik, V. Raychev, and M. Vechev, “PHOG: Probabilistic model for code,” in *Proceedings of the International Conference on Machine Learning (ICML)*, 2016.

[15] C. Maddison and D. Tarlow, “Structured generative models of natural source code,” in *Proceedings of the International Conference on Machine Learning (ICML)*, 2014.

[16] G. A. Aye, S. Kim, and H. Li, “Learning autocompletion from real-world datasets,” *arXiv preprint arXiv:2011.04542*, 2020.

[17] M. Allamanis, M. Brockschmidt, and M. Khademi, “Learning to represent programs with graphs,” in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2018.

[18] H. Inan, K. Khosravi, and R. Socher, “Tying word vectors and word classifiers: A loss framework for language modeling,” *arXiv preprint arXiv:1611.01462*, 2016.

[19] R. Sennrich, B. Haddow, and A. Birch, “Neural machine translation of rare words with subword units,” in *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*, 2016.

[20] X. Zhang, J. Zhao, and Y. LeCun, “Character-level convolutional networks for text classification,” in *Advances in neural information processing systems*, 2015, pp. 649–657.

[21] Y. Kim, Y. Jernite, D. Sontag, and A. M. Rush, “Character-aware neural language models,” in *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[22] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural Computation*, 1997.

[23] D. Bahdanau, K. Cho, and Y. Bengio, “Neural machine translation by jointly learning to align and translate,” in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2015.

[24] Y. Kim, “Convolutional neural networks for sentence classification,” *arXiv preprint arXiv:1408.5882*, 2014.

[25] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, “Attention is all you need,” in *Advances in Neural Information Processing Systems*, 2017, pp. 5998–6008.

[26] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of deep bidirectional transformers for language understanding,” *arXiv preprint arXiv:1810.04805*, 2018.

[27] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, “Language models are unsupervised multitask learners,” 2019.

[28] Microsoft, “IntelliSense,” https://code.visualstudio.com/docs/editor/intellisense, 2020. [Online]. Available: https://code.visualstudio.com/docs/editor/intellisense

[29] JetBrains, “PyCharm code completion,” https://www.jetbrains.com/help/pycharm/auto-completing-code.html, 2020. [Online]. Available: https://www.jetbrains.com/help/pycharm/auto-completing-code.html

[30] ——, “IntelliJ code completion,” https://www.jetbrains.com/help/idea/auto-completing-code.html, 2020. [Online]. Available: https://www.jetbrains.com/help/idea/auto-completing-code.html

[31] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, “Suggesting accurate method and class names,” in *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2015.

[32] V. J. Hellendoorn, S. Proksch, H. C. Gall, and A. Bacchelli, “When code completion fails: A case study on real-world completions,” in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 960–970.

[33] C. V. Lopes, P. Maj, P. Martins, V. Saini, D. Yang, J. Zitny, H. Sajnani, and J. Vitek, “Déjàvu: a map of code duplicates on github,” *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, p. 84, 2017.

[34] M. Allamanis, “The adverse effects of code duplication in machine learning models of code,” in *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, 2019, pp. 143–153.

[35] G. A. Miller, “The magical number seven, plus or minus two: some limits on our capacity for processing information.” *Psychological Review*, 1956.

[36] M. Courbariaux, Y. Bengio, and J.-P. David, “Training deep neural networks with low precision multiplications,” *arXiv preprint arXiv:1412.7024*, 2014.

[37] M. U. Gutmann and A. Hyvärinen, “Noise-contrastive estimation of unnormalized statistical models, with applications to natural image statistics,” *Journal of Machine Learning Research (JMLR)*, 2012.

[38] J. Attenberg, A. Dasgupta, J. Langford, A. Smola, and K. Weinberger, “Feature hashing for large scale multitask learning,” in *Proceedings of the International Conference of Machine Learning (ICML)*, 2009.

[39] J. Moody, “Fast learning in multi-resolution hierarchies,” in *Advances in neural information processing systems*, 1989, pp. 29–39.

[40] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, www.deeplearningbook.org.

[41] E. Foundation, “Code Recommenders,” www.eclipse.org/recommenders, visited Mar 2020.

[42] R. Kuhn and R. De Mori, “A cache-based natural language model for speech recognition,” *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 1990.

[43] S. Nguyen, T. Nguyen, Y. Li, and S. Wang, “Combining program analysis and statistical language model for code statement completion,” in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 710–721.

[44] M. Brockschmidt, M. Allamanis, A. L. Gaunt, and O. Polozov, “Generative code modeling with graphs,” in

Proceedings of the International Conference on Learning Representations (ICLR), 2019.

[45] R. Tang, Y. Lu, L. Liu, L. Mou, O. Vechtomova, and J. Lin, “Distilling task-specific knowledge from BERT into simple neural networks,” *arXiv preprint arXiv:1903.12136*, 2019.

Appendix A
Implementation Concerns

During training, we want to compute representations of the candidate completion targets provided by $\mathcal{P}$ for each sample in a minibatch and then score these representations against the output of $\mathcal{C}$ for each of the corresponding contexts. One problem we encounter when using the static analysis-based candidate provider StAn is that the number of candidate targets, *i.e.* $|\mathcal{P}(\boldsymbol{t}_{\text{cx}})|$, varies widely for different contexts $\boldsymbol{t}_{\text{cx}}$. One option would be to pad up to a maximum number of suggestions, but given the severe skew in the distribution of the number of suggestions from $\mathcal{P}$ for different completion contexts, this would lead to wasted computational effort. We overcome this by flattening the suggestions along the batch dimension, feeding them into the token encoder together with a complimentary tensor that encodes the origin index of each suggestion. This allows us to perform distributed scatter-style operations[4] and efficiently compute Equation 6 across the examples in the training minibatch without padding. At test time, this problem vanishes since no batching is required.

[4] For example `unsorted_segment_sum` in TensorFlow, and `scatter_add_` in PyTorch.