Fast and Memory-Efficient Neural Code Completion

Alexey Svyatkovskiy
alsvyatk@microsoft.com
Microsoft

Sebastian Lee*
sebalexlee@gmail.com
University of Oxford, UK

Anna Hadjitofi*
annahadjitofi@googlemail.com
Alan Turing Institute, UK

Maik Riechert
marieche@microsoft.com
Microsoft

Juliana Franco
juliana.franco@microsoft.com
Microsoft

Miltiadis Allamanis
miltiadis.allamanis@microsoft.com
Microsoft

ABSTRACT

Code completion is one of the most widely used features of modern integrated development environments (IDEs). Deep learning has recently made significant progress in the statistical prediction of source code. However, state-of-the-art neural network models consume prohibitively large amounts of memory, causing computational burden to the development environment, especially when deployed in lightweight client devices.

In this work, we reframe neural code completion from a generation task to a task of learning to rank the valid completion suggestions computed from static analyses. By doing so, we are able to design and test a variety of deep neural network model configurations. One of our best models consumes 6 MB of RAM, computes a single suggestion in 8 ms, and achieves 90% recall in its top five suggestions. Our models outperform standard language modeling code completion techniques in terms of predictive performance, computational speed, and memory efficiency. Furthermore, they learn about code semantics from the natural language aspects of the code (*e.g.* identifier names) and can generalize better to previously unseen code.

1 INTRODUCTION

Deep learning has had a substantial impact on software engineering methods across a variety of tasks [3]. One early application of machine learning of source code has been code completion (or autocompletion) [11, 21, 38], *i.e.* the task of predicting the elements that the developer will type. Code completion is the most frequently used feature in IDEs [6, 37]. Early machine learning methods [11, 39] used feature-based models to predict the next method to be invoked. However, these models can only support suggestions on a limited, pre-defined set of APIs (Application Programming Interfaces) and do *not* have the capability to generalize to different APIs or different versions of the same API. Other models, such as *n*-gram language models [21], do not require extraction of specific features, but instead view the code as a sequence of tokens and — to some extent — can generalize to new code and APIs [19]. Recently, neural machine learning methods [26, 43] have been found to be effective for code completion, achieving state-of-the-art accuracy.

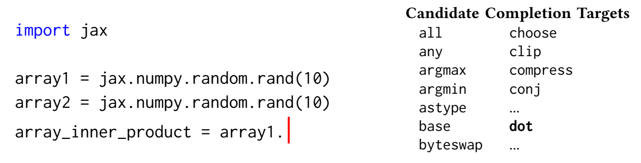
At a high level, the desiderata for a code completion system are:

- *high accuracy*: serve good, relevant suggestions to the user;
- *low memory*: consume small amounts of memory in the development environment;
- *high speed*: serve suggestions in real time, with a minimal computational burden;
- *high coverage*: make (accurate) suggestions in as many cases as possible, *e.g.* even for previously unseen APIs;
- *private*: no information about the developer's code should be moved outside of the code editor;
- *offline*: no network connection should be necessary to provide good completions.

While neural machine learning models on code completion tasks have been shown to achieve state-of-the-art accuracy, practical deployment is inhibited by issues pertaining to the desired attributes mentioned above. First, machine learning-based models traditionally treat code completion as a generation problem [16, 19, 26, 41] in which the model needs to generate the full completion. However, generating predictions requires learning about all potential code identifiers, which are significantly sparse, *i.e.* identifiers are extremely diverse and occur infrequently [5]. In neural language models, this is achieved either by learning "embeddings" for each possible token in a vocabulary of tokens [19, 43] or by learning to generate these tokens from parts [26]. There are three important problems with this approach that hinder deployment: ▹ poor *coverage*: models deal poorly with previously unseen (out-of-vocabulary) tokens, *e.g.* from new APIs; it is hard for any model to predict the name of a method it has never seen before. ▹ large *memory* footprint: these models are prohibitively large for practical deployment in lightweight clients or without introducing bloat in the developer environment. ▹ slow inference *speed*: suggestions need to be performed real-time on the developer's CPU. Some of the above problems can be alleviated by moving completion models to the cloud [45], sacrificing privacy and offline usage.

In this work, we tackle the aforementioned limitations by reformulating the problem of code completion from *generation* to *ranking*. We achieve this by taking advantage of the candidate suggestions generated by pre-existing static analyses. This significantly improves the predictive accuracy and enables us to use finer 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 only 6 MB of RAM and execute in a few milliseconds while still achieving 90% recall in their top five suggestions. Such systems can support a wide variety of developer environments, including those that are significantly resource-constrained, and avoid introducing bloat to existing editors and IDEs. This is important for the software engineering community that seeks to provide inclusive

*Work performed while the authors were AI resident in the Microsoft Research AI Residency program in Cambridge, UK.



```
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   …
```

Figure 1: Motivating Example. The developer is employing `jax` and is currently trying to declare a `array_inner_product` using the `jax` API (left; cursor in red). Most modern IDEs will perform a static analysis to determine the set of valid code completions at the current *completion location*. The *candidate completion targets* (right) are then shown to the user.

solutions to developers — including those that do not have access to high-end machines.

In brief, (a) we present a modular neural architecture tailored to code completion that allows us to combine a multitude of neural components that offer varying trade-offs in terms of memory, speed and accuracy (section 3); (b) we implement our architecture for API completion and present an extensive and principled evaluation (section 4). We test multiple model configurations with multiple hyperparameter settings and show that neural models that have a memory footprint as little as 3 MB can achieve 90% completion accuracy in less than 10 ms; finally (c) we show how the best models can generalize to previously unseen libraries (subsection 4.3).

2 MOTIVATING EXAMPLE

We use the synthetic snippet shown in Figure 1 as a running example. The 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 to determine the type of `array1` and return the list of *candidate completion targets* 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 suggestions. In this case, the list of suggestions is quite long (Figure 1; right). One approach, used by Bruch et al. [11], Proksch et al. [39], 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 capturing such features that cover a large corpus of code is difficult.

Furthermore, such approaches learn about each individual API 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 in early 2019 very few, if any, public codebases used it. Many existing approaches to code completion would not generalize to `jax` or other previously unseen completion targets since they require sufficient training data of the usage of an API within existing code.

[1] At the time of writing, `jax` is at version 0.1.x indicating that it will rapidly evolve, potentially introducing new APIs and breaking old ones. For such APIs the available data will be scarce.

Neural models alleviate this issue as they can — in principle — 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` specifically (Figure 1), which is built to match `numpy` in many aspects, neural models can recognize similarities in the numeric manipulations in Figure 1. An important source of information is contained in the names within the completion context (*e.g.* the subtokens in `array_inner_product` of Figure 1). Early models [5, 9, 19, 21, 32, 46] did *not* take into account the structure within names, which nevertheless contains valuable (but noisy) information. Only recently, Karampatsis et al. [26] introduced such techniques in code completion. In this work, we take this idea a step further and 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.

However, 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. Considering the diverse nature of the completion targets, this is a (unnecessarily) hard task, since the language model needs to be aware of all possible completion targets (*e.g.* all the candidate completion targets in Figure 1), or be able to reconstruct them from scratch. In this work, we show that treating the neural code completion problem as the problem of *ranking* the candidate completion targets returned from a static analysis yields significantly improved results and allows us to build lightweight and accurate neural code completion models.

3 APPROACH

To address the code completion task, we present a general neural architecture (Figure 2). Designing neural networks for a specific task is an engineering effort that commonly involves combining different components (also referred to as "modules") in a suitable way 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 2):

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

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

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

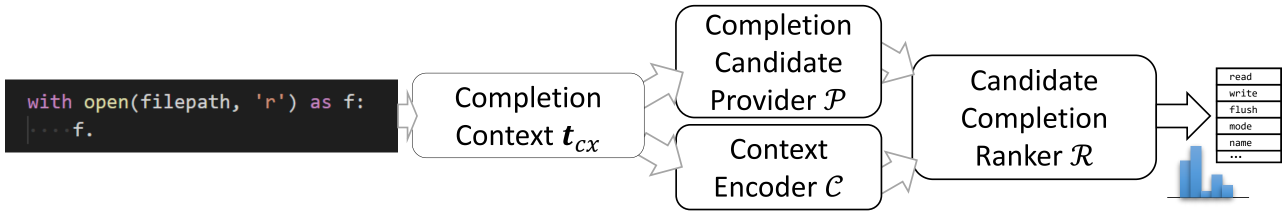

with open(filepath, 'r') as f:
f.
Completion Context t_{cx}
Completion Candidate Provider $\mathcal{P}$
Context Encoder $\mathcal{C}$
Candidate Completion Ranker $\mathcal{R}$
read
write
flush
mode
name
...

Figure 2: Our architecture for machine learning-based code completion systems. From the developer environment, the (partial) code context t_{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 completion 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 1), showing how different choices affect suggestion accuracy, model size and computational efficiency. All our neural models are trained end-to-end.

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

The components, their inputs/outputs, and concrete implementations are shown in Table 1. We denote a concrete configuration with a tuple of the form $\langle \mathcal{E}, \mathcal{C}, \mathcal{P} \rangle$. For example, ⟨Subtoken, Gru, StAn⟩ is an instantiation of the framework with a Subtoken token encoder $\mathcal{E}$, a Gru context encoder $\mathcal{C}$ and a StAn completion ranker $\mathcal{P}$. Our architecture is general and subsumes neural language models [19, 26, 43]. By alternating the concrete implementation of the components, we retrieve a range of neural code completion architectures. To restrict the explored space, we fix two aspects. First, we treat the completion context as a list of the N tokens before the completion location, *i.e.* $t_{cx} = [t_0, ..., t_{N-1}]$. For example, the last few elements of the completion context of Figure 1 are [..., `array_inner_product`, =, `array1`, `.`]. Although more sophisticated representations of code have been explored [3] and can be used within the presented framework, token-level models have been shown to have great performance [26] while reducing the computational cost for extracting information from the code context. For example, although the graph representations of Allamanis et al. [4] would contain more information, the computational cost of processing such structures is prohibitive for real-time systems. We thus pass to the context encoder $\mathcal{C}$ the context t_{cx} and the token encoder $\mathcal{E}$. The second design decision is to use the same token encoder in $\mathcal{C}$ and $\mathcal{R}$. This simplifies the search space, while reducing the number of model parameters. Although this may constrain the model, Inan et al. [23] showed that such an approach is theoretically principled and yields competitive results. In the rest of the section, we discuss concrete implementations for each component.

3.1 Token Encoders $\mathcal{E}$

There is a wide literature of methods for 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 [5]. 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 that allow us to better capture the sparse nature of code identifiers while simultaneously allowing us to reduce the memory requirements of a neural code completion model. All of the presented token encoders have been used in some form in previous works 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.*

$$\mathcal{E}_{\text{Token}}(t) = \text{EmbeddingLookUp}(t, V_t), \quad (1)$$

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 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 MBs 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. 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.*

$$\mathcal{E}_{\text{Subtoken}}(t) = \bigoplus_{t_s \in \text{Split}(t)} \text{EmbeddingLookUp}(t_s, V_s), \quad (2)$$

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 $P(s_i\|\boldsymbol{c}_{\text{cx}})$	Dot

Table 1: Components of the architecture in Figure 2. By combining these components, we retrieve a concrete neural code completion model system. We denote a specific architecture using a tuple, *e.g.* ⟨Subtoken, Gru, StAn⟩ 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 $\mathcal{C}$.

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 concrete operations for $\oplus$: element-wise summation, average, and maximum. Since in our experiments all aggregation operators achieve similar results, we only report the results for the element-wise maximum.

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 [42] 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. We use the implementation in the `sentencepiece` library [29].

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 (a) the number of parameters commonly is significantly smaller compared to other encoders and (b) 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. [48] and Kim et al. [28]. We define an alphabet that consists of commonly used characters present 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.

3.2 Context Encoders $\mathcal{C}$

Context encoders 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 efficiency we only consider context encoders of the form

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

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

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

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

where $\boldsymbol{h}^{(\nu)}$ is the vector state at position ν, $\boldsymbol{t}_\nu$ 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 [8], 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 RNN, 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 single vector. There is a multitude of CNN configurations that are applicable to our setting, but all of them accept as an input an $N \times D$ matrix, and after a few layers of convolution, a pooling layer is used to compute the final H-dimensional context representation $\boldsymbol{c}_{\text{cx}}$. This architecture has resemblance to the natural language classification work of Kim [27].

Transformer Context Encoders. (Transformer) An alternative to RNN-based and CNN-based sequence models are transformers [47] which have recently shown exceptional performance in natural language processing, such as in BERT [13] and GPT-2 [40]. 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 and use as context representation the first vector of the output.

Completion Location-Annotated Encoders. We can provide additional information to any of the above context encoders C, *e.g.* information derived from analyzing the code. Here, we test adding some lightweight information that is useful for API completion. Specifically, we annotate the variable or namespace on which an API completion is performed. For example, in the code of Figure 1 we indicate to the context encoders all the tokens that refer to the receiver object `array1`. This may allow a context encoder, to recognize API patterns, *e.g.* if `foo.open()` was previously invoked then invoking `read()` on `foo` is likely. Of course, other annotations are also possible here, 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 to their name a diamond ($\diamond$), *e.g.* Gru$^\diamond$.

3.3 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 [33], PyCharm [25] and IntelliJ [24] 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 IDEs [43] 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. 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 particular case is "structural prediction" providers that learn to predict completion targets by composing them from individual subunits, such as subtokens [2], BPE [26] or characters. We do not test such providers for two reasons. First, the computational cost for the structural prediction of completion targets imposes significant burden to the developer's environment. Second, the generation of a full candidate completion target requires taking multiple steps (*e.g.* one per subtoken for a subtoken-level provider). The error of such machine learning models compounds for each step, making the generation of valid completions hard.

3.4 Completion Ranker $\mathcal{R}$

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

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

i.e. the softmax over the dot product of the token encodings of candidate suggestions with a linearly transformed context encoding $\boldsymbol{c}_{\text{cx}}$. Here, W is a linear layer of size $H \times D$, which is learned along with the rest of the model parameters, and 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 language model, with the difference that this equation is evaluated only at specific completion locations and *not* for each source code token. However, when the candidate provider yields a variable set of candidates, Equation 6 can be thought of as a ranking model of the candidates in $\{s_i\}$. At test-time, Equation 6 needs to be fully computed only if the exact probabilities need to be presented to the user. If only a ranked list is required, then ranking the candidates by $(W\boldsymbol{c}_{\text{cx}})^\top \mathcal{E}(s_k) + b_{s_k}$ is sufficient, since softmax is monotonic. Although this reduces the computational cost of the completion ranker $\mathcal{R}$ at test time, filtering the suggestions as discussed in section 4.2 is not possible, and therefore we do *not* employ this trick.

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 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[2] 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.

3.5 Composing Components: Model Zoo

Table 1 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

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

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 $N = 80$.

4 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: the completion of method invocations and field accesses (*i.e.* API completion). API completion is the task of suggesting the next method that a developer will call at a given function 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 a 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 [20]. In this section, we focus on this one. However, our framework is more general; as long as there is a candidate provider $\mathcal{P}$ for a given completion context, our approach is still applicable.

To evaluate the API completion performance, we follow the evaluation methods of previous work: we assume that code is written sequentially and from left to right. This is a strong assumption that is rarely true in practice. Nevertheless, there is no reason to believe that improvements measured using this assumption hurt the performance of real-life systems. The necessity to preserve the confidentiality and privacy of developers and their code makes it impossible to obtain large-scale data on how developers write code in practice and thus this evaluation method is a reasonable strategy for offline evaluation. Hellendoorn et al. [20] explore in detail how real-life code completion differs to standard assumptions.

Data Collection. The training, validation and test data used for the experiments came from the 2700 top-starred Python source code repositories on GitHub. Scraped dataset may contain a large amount of duplicates [1, 31], which may skew the evaluation results. To avoid this, we deduplicate our dataset using the tool of Allamanis [1]. Our dataset contains libraries from a diverse set of domains including scientific computing, machine learning, dataflow programming, and web development. To prepare the data, we use the type inference engine currently in Visual Studio Code (PTVS) to collect all completion locations where an API completion suggestion would be emitted by PTVS, similar to Svyatkovskiy et al. [43]. We extract the previous $N = 80$ tokens preceding the method invocation ($\boldsymbol{t}_{\mathrm{cx}}$), and information about the receiver object or namespace, which we use to recreate a Python StAn candidate completion provider. The final dataset contains approximately 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. The most common completed APIs are from the packages `numpy`, `os`, `str`, `list` and `os.path`.

Evaluation Metrics. We measure three aspects of the completion models: accuracy, model size and suggestion speed. To measure *model size*, we compute the total number of parameters of each model. This number correlates well with the actual random access memory (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 evaluation methodology is similar in nature to the one used by Proksch et al. [39].

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 on 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.

Finally, we are interested in evaluating the *predictive accuracy* of each model. We use two well established metrics. First, "recall at top k" (denoted as "Recall@k") measures the percentage 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 possible score, and is defined as $\frac{1}{N}\sum_{i=1}^{N}\frac{1}{r_i}$ where r_i is the rank of each sample.

Each of our experiments runs on a standalone virtual machine equipped with single 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 [12], 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.

4.1 Multiobjective Evaluation

Although predictive performance is an important factor, in this work we additionally need to consider the memory and computational cost trade-offs to offer the best experience to developers. Improving on the last two quantities usually reduces predictive accuracy. Therefore, we treat the problem as multiobjective.

To achieve this, we run a search across multiple model configurations and hyperparameters. 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 3 plots the Pareto fronts across our metrics for some of the model configurations, as retrieved by our search. Each line represents the Pareto optimal options for a given configuration. Table 2 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 represents 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 size for a plugin in a modern IDE or editor.

We 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 ⟨Token, —, StAn⟩ model. The performance of the baseline is worse than all neural models in terms of recall and MRR, although it is fast and small in size (Table 2). This is not surprising, since this model does *not* take into account any context information beyond what is available to the StAn provider.

StAn vs. *Vocab.* Table 2 shows that Vocab-based models underperform StAn-based models. These models are similar to the language models presented by Hellendoorn and Devanbu [19], Karampatsis et al. [26] and Svyatkovskiy et al. [43] except that our models are trained only on API completion locations, and not for predicting arbitrary code tokens[3]. We do *not* plot the Pareto fronts of Vocab models in Figure 3a since they are worse than all other fronts and would require increasing the scale of the plot. All Vocab-based models have multiple shortcomings. First, they need to generate the target completion from a long list of all possible APIs, without any explicit knowledge of the correct choices, and thus have multiple opportunities for making errors. In contrast, StAn models have strictly more information. 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 model performs worse. The difference is more pronounced for models with a smaller vocabulary, which fail to represent the sparsity of code tokens. In subsection 4.3 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 (<1 MB), however even 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.

[3]Pretraining the Vocab or StAn-based models may improve predictive performance on our task [14] but given space and time limitations, we leave this to future work.

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 (⋄) context encoders variants provide a small but consistent improvement thanks to the additional information.

Model Size vs. *Computational Cost.* Figure 3c 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 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.

4.2 Additional Improvements

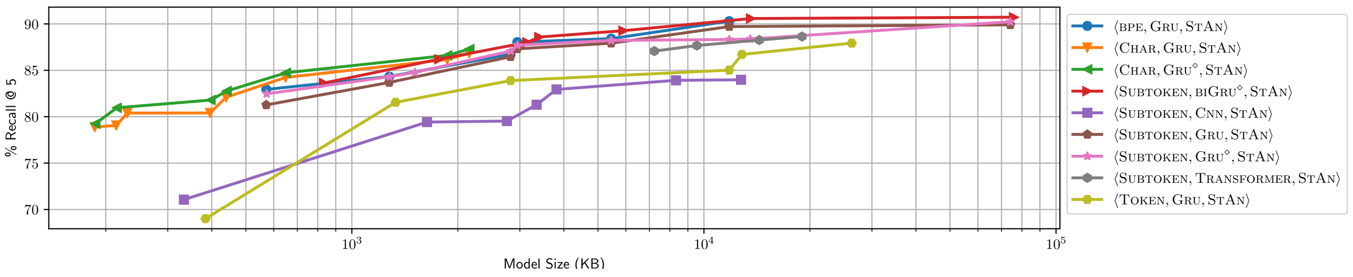
The analysis presented so far compared multiple model combinations with respect to the design decisions of the neural model. We showed that StAn-based models outperform Vocab-based models, that a Gru context encoder provides a reasonable way to summarize the context, and that Subtoken and bpe provide a good trade-off between accuracy and model size. We now explore potential ways to further improve ⟨□, biGru, StAn⟩ models.

When to Suggest. Code completion systems — like all machine learning methods — are prone to mistakes. Additionally, the code completion task is ill-defined since it requires inferring the latent intent of the developer from a partial context. Even a human expert watching a developer write code would not be able to predict every target completion correctly without additional information. Furthermore, showing to the user incorrect suggestions is distracting.

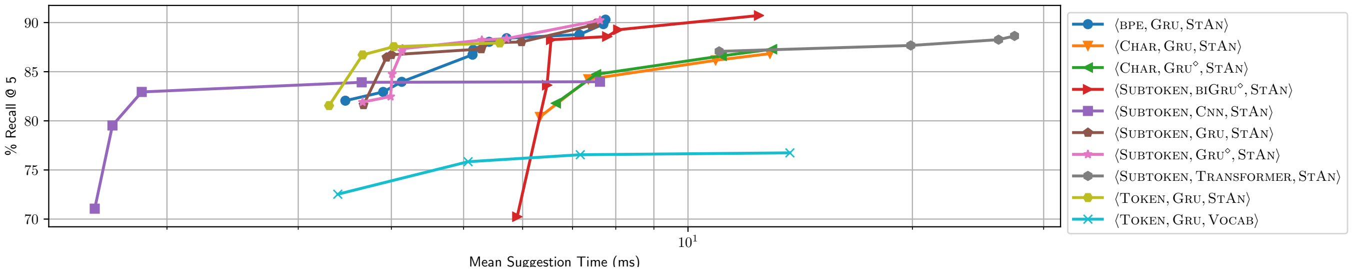
Fortunately, the models discussed in this work give a probabilistic estimate over each target completion (Equation 6). We can use this estimate to limit code completion models to confident suggestions at the cost of showing fewer suggestions. Figure 4 shows the Recall@1 for two of our best-performing models. The plot can be used as a tool for setting an appropriate threshold on the probability of the suggestions. We can increase the predictive performance of a model at the cost of making suggestions at fewer completion locations. For example, if we want the top suggestion to be correct 80% of the time, the confidence threshold needs to be set to 40% for the models shown in Figure 4.

Table 2: 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 (—).

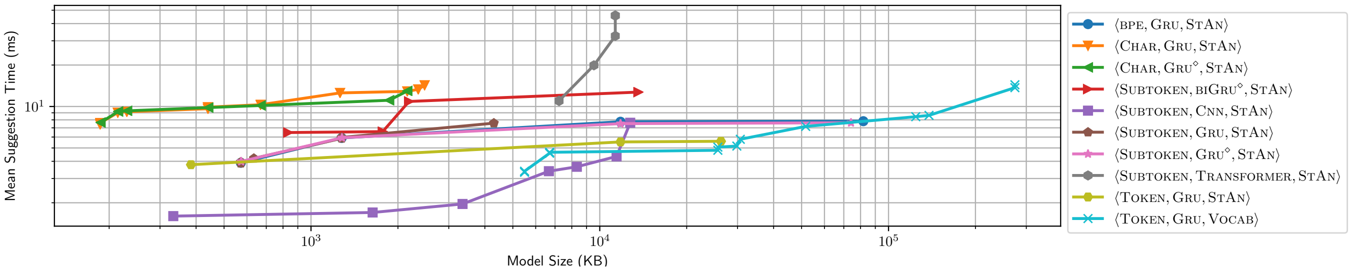
$\mathcal{E}$	$\mathcal{C}$	$\mathcal{P}$	Best for size ~3 MB				Best for size ~50 MB			
			Recall@1	Recall@5	MRR	Time (ms)	Recall@1	Recall@5	MRR	Time (ms)
Most Frequently Used (Popularity)			41.75	72.04	0.5470	0.02 ± 0.01	—	—	—	—
Token	Gru	Vocab	53.01	72.53	0.6140	3.39 ± 1.00	55.87	76.55	0.6477	7.17 ± 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.

Figure 3: 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.

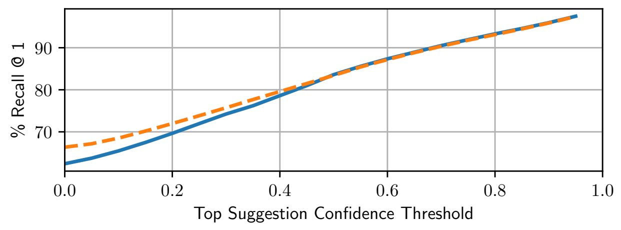


Figure 4: Recall@1 for the best ⟨SUBTOKEN, GRU, STAN⟩ (solid line) and ⟨BPE, GRU, STAN⟩ (dashed line) models with size below 3 MB. When limiting the model to increasingly confident suggestions, predictive accuracy improves.

Training without a Static Analyzer. To train a model with a StAn provider requires that a static analysis tool is able to produce the training data. However, getting a static analyzer to run for a sufficiently large codebase is hard and does not easily scale. Inspired by techniques such as negative sampling [34] and NCE [18], we may be able to overcome this challenge by using a proxy candidate provider that yields random distractor candidate completion targets. Furthermore, for computational efficiency, we can use the target completions of other samples in each minibatch as distractors.

We test this on the setting of the best ⟨bpe, Gru, StAn⟩ configuration that is approximately 50 MB. The results show some degradation in performance. For example, Recall@5 falls to 85.5% (from 89.9%) and MRR falls to 0.729 (from 0.786). This suggests that using a proxy static analyzer during training reduces the predictive accuracy of models. 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 limitation to this evaluation. Our testset contains samples where we could use — at batch — a static analyzer to retrieve the candidate suggestions. Thus, our testset is somewhat 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 before it can provide any results. However, our testset contains a diverse set of APIs. This gives us sufficient confidence that these results hold more generally. A related validation of this concept is found in subsection 4.3.

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 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 data structure.

Feature hashing refers to a common trick for vectorizing features [7, 36] 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. Of course — as with all hashing methods — this introduces collisions as multiple features will 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 ⟨Subtoken, Gru, StAn⟩ model. For each subtoken, we compute the MD5 hash of its string representation and map it to an integer in $\{0 \ldots |V| - 1\}$ where $|V|$ is the size of the "vocabulary", *i.e.* $\phi(s) = \text{MD5}(s) \mod |V|$. 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.

4.3 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 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 3 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 4.1) but is the only reasonable choice for this scenario, since we have no prior information about the APIs. Table 2 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 StAn-based models with granular token encodings (*e.g.* Subtoken, bpe) are also preferable from a generalization perspective.

5 RELATED WORK

Our work is related to a large set of literature in natural language processing and machine learning, particularly deep learning. We

Table 3: Performance on unseen libraries for models with size ∼ 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
Token	Gru	StAn	50.83	0.3486	80.12	0.5652	66.67	0.5180
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
Char	Gru	StAn	63.21	0.4334	74.22	0.4910	72.65	0.5616
Random Choice		StAn	38.80	0.2512	40.37	0.2628	68.17	0.4901

refer the interested reader to the book of Goodfellow et al. [17] 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. [11], 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 [15] and was — to our knowledge — the first data-driven code completion system to be deployed. However, as of 2019 this project has been discontinued [15]. Following similar principles, Proksch et al. [39] 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. [11] and Proksch et al. [39] rely heavily on manually extracting features that are relevant to the completion task. For example, Proksch et al. [39] 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 [3]. 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. [21] who create a token-level n-gram language model completion models. This was arguably the first model that performed unconstrained code completion and was shown to work well. A variety of language model flavors have since been explored [5, 9, 38]. Further improvements to language models (and therefore completion systems) were made by Tu et al. [46] 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 [30], performance could be improved. Hellendoorn and Devanbu [19] 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.

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. [26] 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 [19]. Despite this, the model of Karampatsis et al. [26] can have a large memory footprint and treats completion as a generation problem over BPE subtokens. We showed in our evaluation that such models have characteristics that are prohibitive for real-life deployment. Simultaneously, Svyatkovskiy et al. [43] presented a neural model that corresponds to our ⟨Token, Gru, Vocab⟩ model. Although other, more structured, language models of code have been researched [9, 10, 32], 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. As far as we are aware, this work is the first attempt to use ranking together with neural models for code completion.

6 DISCUSSION & OPEN CHALLENGES

In this work, we presented a principled exploration of the design space of practical neural code completion models. We showed how to design and 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 ranking models strike the best trade-off among predictive performance, model size and computational speed across the tested models. The subtoken, StAn-based models also exhibit good generalization to unseen APIs. Such models can be practically deployed in IDEs. This is especially important to the software engineering community, which seeks to provide inclusive solutions to all developers, even those that do not have access to state-of-the-art equipment.

Future work. Speed and memory improvements may also be possible with techniques such as quantization [12, 43] and knowledge distillation [44]. Providing that these techniques do not result in a significant drop in recall, they could be an additional step towards making our neural models even more computationally and memory efficient.

REFERENCES

[1] Miltiadis Allamanis. 2019. 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*. 143–153.

[2] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. 2015. 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)*.

[3] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)* 51, 4 (2018), 81.

[4] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to Represent Programs with Graphs. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

[5] Miltiadis Allamanis and Charles Sutton. 2013. Mining source code repositories at massive scale using language modeling. In *Proceedings of the Working Conference on Mining Software Repositories (MSR)*.

[6] Sven Amann, Sebastian Proksch, Sarah Nadi, and Mira Mezini. 2016. A study of Visual Studio usage in practice. In *Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering (SANER)*.

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

[8] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural machine translation by jointly learning to align and translate. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

[9] Pavol Bielik, Veselin Raychev, and Martin Vechev. 2016. PHOG: Probabilistic Model for Code. In *Proceedings of the International Conference on Machine Learning (ICML)*.

[10] Marc Brockschmidt, Miltiadis Allamanis, Alexander L Gaunt, and Oleksandr Polozov. 2019. Generative code modeling with graphs. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

[11] Marcel Bruch, Martin Monperrus, and Mira Mezini. 2009. 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)*.

[12] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. 2014. Training deep neural networks with low precision multiplications. *arXiv preprint arXiv:1412.7024* (2014).

[13] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[14] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. *arXiv preprint arXiv:2002.08155* (2020).

[15] Eclipse Foundation. [n. d.]. Code Recommenders. www.eclipse.org/recommenders. Visited Mar 2020.

[16] Christine Franks, Zhaopeng Tu, Premkumar Devanbu, and Vincent Hellendoorn. 2015. Cacheca: A cache language model based code suggestion tool. In *Proceedings of the International Conference on Software Engineering (ICSE)*.

[17] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press. www.deeplearningbook.org.

[18] Michael U Gutmann and Aapo Hyvärinen. 2012. Noise-contrastive estimation of unnormalized statistical models, with applications to natural image statistics. *Journal of Machine Learning Research (JMLR)* (2012).

[19] Vincent J Hellendoorn and Premkumar Devanbu. 2017. Are deep neural networks the best choice for modeling source code?. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*.

[20] Vincent J Hellendoorn, Sebastian Proksch, Harald C Gall, and Alberto Bacchelli. 2019. When code completion fails: A case study on real-world completions. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 960–970.

[21] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the naturalness of software. In *Proceedings of the International Conference on Software Engineering (ICSE)*.

[22] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural Computation* (1997).

[23] Hakan Inan, Khashayar Khosravi, and Richard Socher. 2016. Tying word vectors and word classifiers: A loss framework for language modeling. *arXiv preprint arXiv:1611.01462* (2016).

[24] JetBrains. 2020. IntelliJ Code Completion. https://www.jetbrains.com/help/idea/auto-completing-code.html. https://www.jetbrains.com/help/idea/auto-completing-code.html

[25] JetBrains. 2020. PyCharm Code Completion. https://www.jetbrains.com/help/pycharm/auto-completing-code.html. https://www.jetbrains.com/help/pycharm/auto-completing-code.html

[26] Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. 2020. Big Code!= Big Vocabulary: Open-Vocabulary Models for Source Code. In *Proceedings of the International Conference on Software Engineering (ICSE)*.

[27] Yoon Kim. 2014. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882* (2014).

[28] Yoon Kim, Yacine Jernite, David Sontag, and Alexander M Rush. 2016. Character-aware neural language models. In *Thirtieth AAAI Conference on Artificial Intelligence*.

[29] Taku Kudo and John Richardson. 2018. SentencePiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. In *Proceedings of the Conference on Empirical Methods for Natural Language Processing (EMNLP)*.

[30] Roland Kuhn and Renato De Mori. 1990. A cache-based natural language model for speech recognition. *Pattern Analysis and Machine Intelligence, IEEE Transactions on* (1990).

[31] Cristina V Lopes, Petr Maj, Pedro Martins, Vaibhav Saini, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek. 2017. DéjàVu: a map of code duplicates on GitHub. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 84.

[32] Chris Maddison and Daniel Tarlow. 2014. Structured Generative Models of Natural Source Code. In *Proceedings of the International Conference on Machine Learning (ICML)*.

[33] Microsoft. 2020. IntelliSense. https://code.visualstudio.com/docs/editor/intellisense. https://code.visualstudio.com/docs/editor/intellisense

[34] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*. 3111–3119.

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

[36] John Moody. 1989. Fast learning in multi-resolution hierarchies. In *Advances in neural information processing systems*. 29–39.

[37] Gail C Murphy, Mik Kersten, and Leah Findlater. 2006. How are Java software developers using the Eclipse IDE? *IEEE software* 23, 4 (2006), 76–83.

[38] Tung Thanh Nguyen, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. 2013. A statistical semantic language model for source code. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

[39] Sebastian Proksch, Johannes Lerch, and Mira Mezini. 2015. Intelligent code completion with Bayesian networks. *ACM Transactions on Software Engineering and Methodology (TOSEM)* (2015).

[40] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners. (2019).

[41] Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *Proceedings of the Symposium on Programming Language Design and Implementation (PLDI)*.

[42] Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural machine translation of rare words with subword units. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL)*.

[43] Alexey Svyatkovskiy, Ying Zhao, Shengyu Fu, and Neel Sundaresan. 2019. Pythia: AI-assisted Code Completion System. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2727–2735.

[44] Raphael Tang, Yao Lu, Linqing Liu, Lili Mou, Olga Vechtomova, and Jimmy Lin. 2019. Distilling task-specific knowledge from BERT into simple neural networks. *arXiv preprint arXiv:1903.12136* (2019).

[45] TabNine Team. 2020. TabNine. https://tabnine.com/. Visited Mar 2020.

[46] Zhaopeng Tu, Zhendong Su, and Premkumar Devanbu. 2014. On the localness of software. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*.

[47] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*. 5998–6008.

[48] Xiang Zhang, Junbo Zhao, and Yann LeCun. 2015. Character-level convolutional networks for text classification. In *Advances in neural information processing systems*. 649–657.