code2seq: Generating Sequences from Structured Representations of Code

Uri Alon
Technion
urialon@cs.technion.ac.il

Omer Levy
Facebook AI Research
omerlevy@gmail.com

Eran Yahav
Technion
yahave@cs.technion.ac.il

Abstract

The ability to generate natural language sequences from source code snippets can be used for code summarization, documentation, and retrieval. Sequence-to-sequence (seq2seq) models, adopted from neural machine translation (NMT), have achieved state-of-the-art performance on these tasks by treating source code as a sequence of tokens. We present CODE2SEQ: an alternative approach that leverages the syntactic structure of programming languages to better encode source code. Our model represents a code snippet as the set of paths in its abstract syntax tree (AST) and uses attention to select the relevant paths during decoding, much like contemporary NMT models. We demonstrate the effectiveness of our approach for two tasks, two programming languages, and four datasets of up to 16M examples. Our model significantly outperforms previous models that were specifically designed for programming languages, as well as general state-of-the-art NMT models.

1 Introduction

Modeling the relation between source code and natural language can be used for automatic code summarization (Allamanis et al., 2016), documentation (Iyer et al., 2016), retrieval (Allamanis et al., 2015b), and even generation (Balog et al., 2016; Rabinovich et al., 2017; Yin and Neubig, 2017; Devlin et al., 2017; Murali et al., 2017). In this work, we consider the general problem of generating a natural language sequence from a given snippet of source code.

A direct approach is to frame the problem as a machine translation problem, where the source sentence is the sequence of tokens in the code and the target sentence is a corresponding natural language sequence. This approach allows one to apply state-of-the-art neural machine translation (NMT) models from the sequence-to-sequence (seq2seq) paradigm (Sutskever et al., 2014; Cho et al., 2014; Bahdanau et al., 2014; Luong et al., 2015; Vaswani et al., 2017), yielding state-of-the-art performance on various code captioning and documentation benchmarks (Hindle et al., 2012; Iyer et al., 2016; Allamanis et al., 2016; Loyola et al., 2017).

We present an alternative approach for encoding source code that leverages the syntactic structure of programming languages: CODE2SEQ. Specifically, we represent a given code snippet as a set of compositional paths over its abstract syntax tree (AST), where each path is compressed to a fixed-length vector using LSTMs (Hochreiter and Schmidhuber, 1997). During decoding, CODE2SEQ attends over a different weighted sum of the path-vectors to produce each output token, much like NMT models attend over contextualized token representations in the source sentence.

Preprint. Work in progress.

Code summarization in Java:

```
int ________(Item item) {
  int position=0;
  for (Iterator<Item> i=item.getParent()
      .getItemIterator(); i.hasNext(); ) {
    Item child=i.next();
    if (item==child) {
      return position;
    }
    position++;
  }
  return -1;
}
```

find index of item

(a)

Code captioning in C#:

```
string myJson = "{'Username': 'myusername',"
  + "'Password':'pass'}";
using (var client = new HttpClient())
{
  var response = await client.PostAsync(
    "http://yourUrl",
    new StringContent(myJson,
      Encoding.UTF8, "application/json"));
}
```

how to send data in POST request using C#

(b)

Figure 1: Example of (a) code summarization for a Java code snippet, and (b) code captioning for a C# code snippet. The text boxes at the bottom of the figure are the predictions produced by our model. In code summarization we predict the method name (as a sequence of individual tokens). In code captioning we predict a natural language sentence.

We show the effectiveness of our code2seq model on two tasks: (1) code summarization (Figure 1a), where we predict a Java method's name given its body, and (2) code captioning (Figure 1b), where we predict a natural language sentence that describes a given C# snippet. On both tasks, our code2seq model outperforms models that were explicitly designed for code, such as Paths+CRFs (Alon et al., 2018b) and CodeNN (Iyer et al., 2016), as well as state-of-the-art NMT models (Luong et al., 2015; Vaswani et al., 2017). To examine the importance of each component of the model, we conduct a thorough ablation study. In particular, we show the importance of structural encoding of code, by showing how our model yields a significant improvement over an ablation that uses only token-level information without syntactic paths. To the best of our knowledge, this is the first work to leverage the syntactic structure of code for end-to-end generation of sequences.

2 Related Work

The growing availability of open source repositories leads to a rise in the field of using machine learning for source code. Several works modeled code as a sequence of tokens (Hindle et al., 2012; Iyer et al., 2016; Allamanis et al., 2016; Loyola et al., 2017), characters (Bielik et al., 2017) and API calls (Raychev et al., 2014). While sometimes obtaining satisfying results, these works treat code as a sequence, rather than a tree. This forces these techniques to implicitly re-learn the (predefined) syntax of the programming language, wasting resources and decreasing accuracy.

Works that use syntactic information when learning code usually use it for relatively easier tasks which mainly focus on "filling gaps" in a given program (Alon et al., 2018b; Bielik et al., 2016; Raychev et al., 2016, 2015; Allamanis et al., 2018) or semantic classification of code snippets (Alon et al., 2018a). Moreover, none of the works using syntactic relations are compositional, and therefore the number of possible syntactic relations is fixed either before or after training, and often consume a lot of memory.

Alon et al. (2018b) use syntactic paths to represent program elements in learning models, as a plug-in to other end-to-end models. Their syntactic relations are represented *monolithically* and are therefore only those that are observed frequently enough during training. As a result, *they cannot represent unseen relations*. In contrast, by learning AST paths node-by-node using LSTMs, our model can represent and use *any* syntactic path in any unseen example.

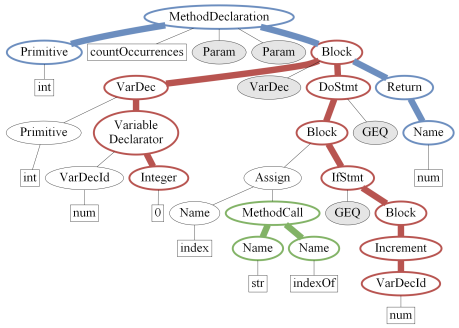
Allamanis et al. (2018) represented code with Gated Graph Neural Networks in which nodes in the graph represented identifiers, and edges represented syntactic and semantic relations in the code such as "ComputedFrom" and "LastWrite". These features were designed for the semantics of a specific programming language and required an expert to think of and implement the useful relations. In contrast, our model has minimal assumptions on the input language and is general enough not to require expert semantic knowledge nor to manually design features.

```java
int countOccurrences(String str, char ch) {
   int num = 0;
   int index = -1;
   do {
      index = str.indexOf(ch, index + 1);
      if (index >= 0) {
         num++;
      }
   } while (index >= 0);
   return num;
}
```

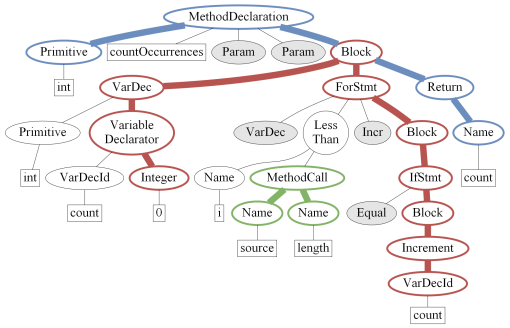
(a)

```java
int countOccurrences(String source, char value) {
   int count = 0;
   for (int i = 0; i < source.length(); i++) {
       if (source.charAt(i) == value) {
           count++;
       }
   }
   return count;
}
```

(b)



(c)



(d)

Figure 2: An example of two Java methods that have exactly the same functionality. Although having a different *sequential* (token-based) representation, considering syntactic patterns reveals recurring paths, which might differ only in single nodes (a `ForStmt` node instead of a `Do`-while node).

3 Representing Code as AST Paths

An Abstract Syntax Tree (AST) is a tree which uniquely represents a source code snippet in a given language and grammar. The leaves of the tree are called *terminals*, and usually refer to user-defined values which represent identifiers and names from the code. The non-leaf nodes are called *nonterminals* and represent a restricted set of elements in the language, e.g., loops, expressions, and variable declarations. For example, Figure 2c shows a partial AST for the code snippet of Figure 2a. Names (such as `num`) and types (such as `int`) are represented as values of terminals; syntactic structures such as variable declaration (`VarDec`) and `DoStmt` are represented as nonterminals.

Given the AST of a code snippet, we consider all pairwise paths between terminals, and represent them as sequences of terminal and nonterminal nodes. We then use these paths with their terminals' values to represent the code snippet itself.

For example, consider the two Java methods of Figure 2. Both of these methods count occurrences of a character in a string. They have exactly the same functionality, although a different implementation, and therefore different surface forms. Encoding these snippets of code as sequences of tokens might not expose recurring patterns that suggest the common method name. However, a structural observation reveals syntactic paths that are common to both methods, and differ only in a single node of a `Do`-while statement versus a `For` statement. This example shows the effectiveness of a syntactic encoding of code. Such an encoder can generalize much better to unseen examples because the AST normalizes a lot of the surface form variance. Since our encoding is compositional, the encoder can generalize even if the syntactic paths are not identical (e.g., a `For` node in one path and a `While` in the other).

Since a code snippet can contain an arbitrary number of such paths, we sample k paths as the representation of the code snippet. To avoid bias, k new paths are sampled afresh on every training iteration. In Section 6 we show that this runtime-sampling significantly improves results compared to sampling k paths for each example in advance.

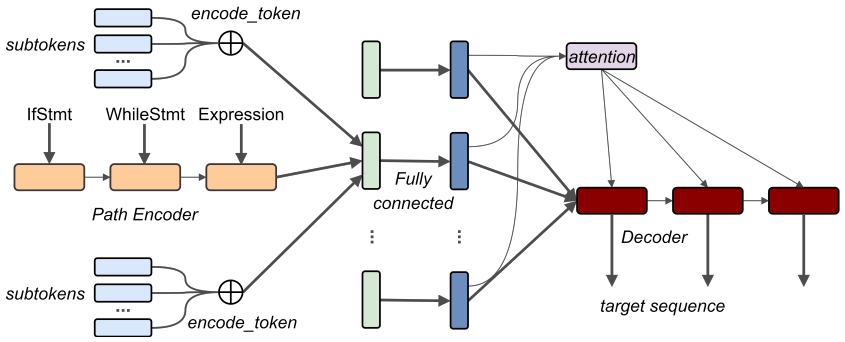


Figure 3: Our model encodes each path as a sequence of AST nodes, and averages the produced input vectors as the initial state of the decoder. The decoder generates an output sequence while attending over the encoded paths.

Formally, we denote a given snippet of code $\mathcal{C}$ and the hyperparameter k. On every training iteration k pairs of terminals are uniformly sampled from within the terminals of the AST of $\mathcal{C}$. Each pair of terminals $\left(v_1^i, v_{l_i}^i\right)$ implies a single path between them: $v_1^i v_2^i ... v_{l_i}^i$. Finally, the input code example is represented as a set of these k random AST paths: $\left\{\left(v_1^1 v_2^1 ... v_{l_1}^1\right), ..., \left(v_1^k v_2^k ... v_{l_k}^k\right)\right\}$, where l_j is the length of the jth path.

4 Model Architecture

Our model follows the standard encoder-decoder architecture for NMT (Section 4.1), with the significant difference that *the encoder does not read the input as a flat sequence of tokens*. Instead, the encoder creates a vector representation for each AST path separately (Section 4.2). The decoder then attends over the encoded AST paths (rather than the encoded tokens) while generating the target sequence. An illustration of our model is shown in Figure 3.

4.1 Encoder-Decoder Framework

Most of today's NMT models are based on an encoder-decoder architecture (Cho et al., 2014; Sutskever et al., 2014; Luong et al., 2015; Bahdanau et al., 2014), where the encoder maps an input sequence of tokens $(x_1, ..., x_n)$ to a sequence of continuous representations $\boldsymbol{z} = (z_1, ..., z_n)$. Given $\boldsymbol{z}$, the decoder then generates a sequence of output tokens $(y_1, ..., y_m)$ one token at a time, hence models the conditional probability: $p\left(y_1, ..., y_m | x_1, ..., x_n\right)$.

At each decoding step, the probability of the next target token depends on the previously generated token, and is therefore factorized as:

$$p\left(y_1, ..., y_m | x_1, ..., x_n\right) = \prod_{j=1}^{m} p\left(y_j | y_{<j}, z_1, ..., z_n\right) \tag{1}$$

In attention-based models, at each time step t in the decoding phase, a context vector c_t is computed by attending over the elements in z using the decoding state h_t, typically computed by an LSTM.

$$\boldsymbol{\alpha}^t = softmax\left(h_t W_a \boldsymbol{z}\right) \tag{2}$$

$$c_t = \sum_{i}^{n} \alpha_i^t z_i \tag{3}$$

The context vector c_t and the decoding state h_t are then combined to predict the current target token y_t. Previous work differs in the way the context vector is computed and in the way it is combined

with the current decoding state. A standard approach (Luong et al., 2015) is to pass c_t and h_t through a multi-layer perceptron (MLP) and then predict the probability of the next token using softmax:

$$p\left(y_t|y_{<t}, z_1, ..., z_n\right) = softmax\left(W_s tanh\left(W_c\left[c_t; h_t\right]\right)\right) \tag{4}$$

4.2 AST Encoder

Given a set of AST paths $(x_1, ..., x_k)$, our goal is to create a vector representation z_i for each path $x_i = v_1^i v_2^i ... v_{l_i}^i$. We represent each path separately by using a bi-directional LSTM to encode the path and sub-token embeddings to capture the compositional nature of the terminals' values (the tokens).

Path Representation Each AST path is composed of nodes and their child indices from a fairly limited vocabulary of 364 symbols in Java and 279 in C#. We represent each node using a learned embedding matrix E^{nodes}, and then encode the entire sequence using the final states of a bi-directional LSTM:

$$h_1, ..., h_l = LSTM(E_{v_1}^{nodes}, ..., E_{v_l}^{nodes}) \tag{5}$$

$$encode\_path(v_1 ... v_l) = [h_l^{\rightarrow}; h_1^{\leftarrow}] \tag{6}$$

Token Representation The first and last node of an AST path are non-terminals whose values are tokens in the code. We split code tokens into *sub*tokens, as was suggested by previous work (Allamanis et al., 2015a, 2016); for example, a token with the value `ArrayList`, will be decomposed into `Array` and `List`. This is somewhat analogous to the recent use of subword information in NMT (e.g., byte-pair encoding (Sennrich et al., 2016)), although in the case of programming languages, coding conventions such as camel notation provide us with an explicit partition of each token. Specifically, we use a learned embedding matrix $E^{subtokens}$ to represent each subtoken, and then sum the subtoken vectors to represent the full token:

$$encode\_token(w) = \sum_{s \in split(w)} E_s^{subtokens} \tag{7}$$

The decoder may also use subtokens when generating the output (e.g. when predicting method names). However, the target subtokens will be generated sequentially via an LSTM, and the subtoken embedding matrix will be different.

Combined Representation To represent the entire path $x = v_1 ... v_l$, we concatenate the path's representation with each of the token representation of each terminal node, and apply a fully-connected layer:

$$z = tanh\left(W_{in}\left[encode\_path(v_1 ... v_l); encode\_token(value(v_1)); encode\_token(value(v_l))\right]\right) \tag{8}$$

where $value$ is the mapping of a terminal node to its associated value, and W_{in} is a $(2d_{path} + 2d_{token}) \times d_{hidden}$ matrix.

Decoder Start State To provide the decoder with an initial state, we average the combined representations of all the paths:

$$h_0 = \frac{1}{k}\sum_{i=1}^{k} z_i \tag{9}$$

Unlike typical encoder-decoder models, the order of the input random paths is not taken into account. Each path is encoded separately and the combined representations are aggregated with mean pooling to initialize the decoder's state. This expresses our intent of representing code as a *set* of random paths.

5 Experiments

We evaluate our model on two code-to-sequence tasks: summarization (Section 5.1), in which we predict Java methods' names from their bodies, and captioning (Section 5.2), where we generate natural language descriptions of C# code snippets. We thus demonstrate that our approach can produce both method names and natural language outputs, and can encode a code snippet in any language for which an AST can be constructed (i.e., a parser exists).

Setup The values of all of the parameters are initialized using the initialization heuristic of Glorot and Bengio (2010). We optimize the cross-entropy loss (Rubinstein, 1999, 2001) with Nesterov momentum (Nesterov, 1983) with value of 0.95, an initial learning rate of 0.01, decayed by a factor of 0.95 every epoch. We apply dropout (Srivastava et al., 2014) of 0.25 on the input vectors x_j, and a recurrent dropout of 0.5 on the LSTM that encodes the AST paths. We used $d_{tokens} = d_{nodes} = d_{hidden} = d_{target} = 128$. Each LSTM which encodes the AST paths had 128 cells and the decoder LSTM had 320 cells. We used $k = 200$ as the number of random paths on each example.

5.1 Code Summarization

In this task, we predict a Java method's name given its body. Code summarization is a good benchmark because a method name in open-source Java projects tends to be succinct and precise, and a method body is usually a complete logical unit. We predict the target method name as a sequence of subtokens; e.g., `setMaxConnectionsPerServer` is predicted as the sequence "set max connections per server". We adopt the measure used by Allamanis et al. (2016), who measured precision, recall and F1 score over the target sequence, case insensitive.

Data We experiment with this task across three datsets:

Java-small – Contains 11 relatively large Java projects, which were originally used for 11 distinct models for training and predicting within the scope of the same project (Allamanis et al., 2016). We use the same data, but train and predict across projects: we took 9 projects for training, 1 project for validation and 1 project as our test set. This dataset contains about 700K examples.

Java-med – A new dataset of the 1000 top-starred Java projects from GitHub. We randomly select 800 projects for training, 100 for testing and the rest for validation. This dataset contains about 4M examples.

Java-large – A new dataset of the 9500 top-starred Java projects which were created since January 2007. We randomly select 9000 projects for training, 300 for testing and 200 for validation. This dataset contains about 16M examples.

Baselines We compare CODE2SEQ to the following baselines: Allamanis et al. (2016) who used a convolutional attention network to predict method names, and syntactic paths with Conditional Random Fields (CRFs) (Alon et al., 2018b). In addition, we compared to two NMT baselines that read the input source code as a stream of tokens: a 2-layer bidirectional encoder-decoder LSTMs with global attention (Luong et al., 2015), and the Transformer (Vaswani et al., 2017) which uses attention mechanisms instead of a recurrent encoder. We put significant effort to strengthen the NMT baselines in order to provide a fair comparison: (1) we split tokens to subtokens, as in our model (e.g., `HashMap` $\rightarrow$ `Hash Map`), (2) we deliberately kept the original casing of the source tokens, since we found it to improve their results a little, and (3) during inference, generated UNK tokens were replaced with the source tokens that were given the highest attention. For the 2-layer BiLSTM we used embeddings of size 512, each of the encoder and decoder had 512 cells, and the default hyperparameters of OpenNMT (Klein et al.). For the Transformer, we used their original hyperparameters (Vaswani et al., 2017).

Performance Table 1 shows the results for the code summarization task. Our model significantly outperforms the baselines in both precision and recall, demonstrating that there is added value in leveraging ASTs to encode source code. The precision of the LSTM baseline was almost as good as of our model on the Java-large dataset, but their recall and F1 score were significantly worse. As a result, our model outperforms the best of the baselines by up to 7.82 points (22% relative improvement).

Model	Java-small			Java-med			Java-large		
	Prec	Rec	F1	Prec	Rec	F1	Prec	Rec	F1
ConvAttention (Allamanis et al., 2016)	50.25	24.62	33.05	60.82	26.75	37.16	60.71	27.60	37.95
Paths+CRFs (Alon et al., 2018b)	8.39	5.63	6.74	32.56	20.37	25.06	22.71	19.83	21.12
2-layer BiLSTM (full tokens)	32.40	20.40	25.03	48.37	30.29	37.25	58.02	37.73	45.73
2-layer BiLSTM (split tokens)	42.63	29.97	35.20	55.15	41.75	47.52	63.53	48.77	55.18
Transformer (Vaswani et al., 2017) (split tokens)	38.13	26.70	31.41	50.11	35.01	41.22	59.13	40.58	48.13
code2seq	**50.64**	**37.40**	**43.02**	**61.24**	**47.07**	**53.23**	**64.03**	**55.02**	**59.19**

Table 1: Our model significantly outperforms previous PL-oriented and NMT models.

Model	BLEU
MOSES† (Koehn et al., 2007)	11.57
IR†	13.66
SUM-NN† (Rush et al., 2015)	19.31
2-layer BiLSTM (split tokens)	19.78
Transformer (Vaswani et al., 2017)	19.68
CodeNN† (Iyer et al., 2016)	20.53
code2seq	**23.04**

Table 2: Our model outperforms previous works in the code captioning task in C#. †Results previously reported by Iyer et al. (2016), and reproduced by us.

Our model outperforms ConvAttention (Allamanis et al., 2016), which was designed specifically for this task, and also outperforms Paths+CRFs (Alon et al., 2018b) which used syntactic features.

Data Efficiency These results show the data efficiency of our architecture. The F1 score on the Java-small dataset, which contains about 5% of the number of examples in the Java-large dataset, is more than 70% of the score on the large dataset. In contrast, the score of Paths+CRFs (Alon et al., 2018b) drops from 21.12 on the large dataset to only 6.74 on the small dataset (about 68% drop). This can be explained by the monolithic nature of their input representations, which is very data consuming, in contrast to our compositional representation.

5.2 Code Captioning

For this task we consider predicting a full natural language sentence given a short C# code snippet. We used the dataset of CodeNN (Iyer et al., 2016), which was collected as 66,015 pairs of filtered questions and answers from StackOverflow. They used a semi-supervised classifier to filter irrelevant examples and asked human annotators to provide two additional titles for the examples in the test set, making a total of three reference titles for each code snippet. For this task we used BLEU score with smoothing, using exactly the same evaluation code of Iyer et al. (2016).

This task is more difficult to evaluate since two captions can be of equal quality but very different linguistically. This is partially addressed by using three references for each test example. Moreover, StackOverflow code snippets are typically short, incomplete at times, and aim to provide an answer to a very specific question. Despite that, we gain an improvement of 2.51 BLEU points over CodeNN on their original dataset.

Baselines We present results compared to CodeNN, 2-layer bidirectional encoder-decoder LSTMs, and the Transformer. As before, we provide a fair comparison by splitting tokens to subtokens, and replacing UNK during inference. We also include numbers from other baselines used by Iyer et al. (2016).

Results Table 2 summarizes the results for the code captioning task. Our model achieves a BLEU score of 23.04, which improves by 2.51 points over CodeNN who introduced this dataset, and over all the other baselines including BiLSTMs and the Transformer.

These results show that when the training examples are short and incomplete code snippets, our model generalizes better to unseen examples than a shallow textual token-level approach, thanks to its

Model	Precision	Recall	F1
No AST nodes	55.51	43.11	48.53
Single prediction (no decoder)	47.99	28.96	36.12
Full-tokens (no subtoken-split)	48.53	34.80	40.53
No tokens (only AST nodes)	33.78	21.23	26.07
No attention	57.00	41.89	48.29
No random (sample k paths in advance)	59.08	44.07	50.49
code2seq (original model)	**60.93**	**45.77**	**52.27**

Table 3: Variations on the code2seq model. These experiments were performed for the code summarization task, on the dev set of Java-med.

syntactic observation of the data. The results of this task show how our model is able to generate both sequences of tokens which compose a method name (Section 5.1) as well as longer natural language sentences (Section 5.2).

6 Ablation Study

To better understand the strengths and weaknesses of our model, we conducted an ablation study.

To evaluate the importance of different components of our model, we vary our model in different ways and measure the change in performance. These experiments were performed for the code summarization task, on the Java-med dataset. We examine several alternative designs:

1. *No AST nodes* - instead of encoding an AST path using an LSTM, take only the first and last terminal values for constructing an input vector
2. *Single prediction* - no sequential decoding, and instead predict the target sequence as a single symbol using a single softmax layer instead of a decoder. This experiment examines whether method names are frequent enough not to require a sequential decoding. We used all the target names that occurred at least 5 times in the training set.
3. *Full-tokens* - no subtoken encoding, and instead use the full tokens values.
4. *No tokens* - use only the AST nodes without using the values associated with the terminals.
5. *No attention* - decode the target sequence given the initial decoder state, without attention.
6. *No random* - no re-sampling of k paths in each iteration, and instead sample in advance and use the same k paths for each example throughout the whole training.

Table 3 shows the results of these alternatives. As seen, *not encoding AST nodes* had a negative effect, showing the importance of learning the syntactic regularities directly. Using a *single prediction* instead of a decoder hurt the results significantly. This shows that the method name prediction task should be addressed as a sequential prediction, despite the methods' relatively short names compared to the full descriptions of the code captioning task. Using *full-tokens* or *no tokens* at all drastically hurt the results, showing the significance of encoding both subtokens and syntactic paths. The *no attention* experiment shows the contribution of attention in our model, which is consistent the constribution of attention in sequence-to-sequence models (Luong et al., 2015; Bahdanau et al., 2014). The *no random* experiment shows the positive contribution of sampling k different paths afresh on every training iteration, instead of using the same sample of paths from each example during whole training.

7 Conclusion

We presented a novel code-to-sequence model which considers the unique syntactic structure of source code with a sequential modeling of natural language. The core idea is to sample paths in the Abstract Syntax Tree of a code snippet, encode those paths with an LSTM and attend to them while generating the target sequence.

We demonstrate our approach by using it to predict method names across three datasets of varying sizes, predict natural language captions given partial and short code snippets, in two programming

languages. Our model performs significantly better than previous programming-language-oriented works and state of the art NMT models applied in our settings.

We believe that the principles presented in this paper can serve as a basis for a wide range of tasks which involve source code and natural language, and can be extended to other kinds of generated outputs. To serve this purpose, we will make all of our trained models and datasets publicly available.

References

Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 38–49, New York, NY, USA, 2015a. ACM. ISBN 978-1-4503-3675-8. doi:10.1145/2786805.2786849. URL http://doi.acm.org/10.1145/2786805.2786849.

Miltiadis Allamanis, Daniel Tarlow, Andrew D. Gordon, and Yi Wei. Bimodal Modelling of Source Code and Natural Language. In *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *JMLR Proceedings*, pages 2123–2132. JMLR.org, 2015b.

Miltiadis Allamanis, Hao Peng, and Charles A. Sutton. A convolutional attention network for extreme summarization of source code. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, pages 2091–2100, 2016. URL http://jmlr.org/proceedings/papers/v48/allamanis16.html.

Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. In *ICLR*, 2018.

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *arXiv preprint arXiv:1803.09473*, 2018a.

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. A general path-based representation for predicting program properties. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 404–419, New York, NY, USA, 2018b. ACM. ISBN 978-1-4503-5698-5. doi:10.1145/3192366.3192412. URL http://doi.acm.org/10.1145/3192366.3192412.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473, 2014. URL http://arxiv.org/abs/1409.0473.

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

Pavol Bielik, Veselin Raychev, and Martin T. Vechev. PHOG: probabilistic model for code. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, pages 2933–2942, 2016. URL http://jmlr.org/proceedings/papers/v48/bielik16.html.

Pavol Bielik, Veselin Raychev, and Martin Vechev. Program synthesis for character level language modeling. 2017.

Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel-rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy i/o. In *International Conference on Machine Learning*, pages 990–998, 2017.

Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pages 249–256, 2010.

Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 837–847, Piscataway, NJ, USA, 2012. IEEE Press. ISBN 978-1-4673-1067-3. URL http://dl.acm.org/citation.cfm?id=2337223.2337322.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997. ISSN 0899-7667. doi:10.1162/neco.1997.9.8.1735. URL http://dx.doi.org/10.1162/neco.1997.9.8.1735.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*, 2016. URL http://aclweb.org/anthology/P/P16/P16-1195.pdf.

G. Klein, Y. Kim, Y. Deng, J. Senellart, and A. M. Rush. OpenNMT: Open-Source Toolkit for Neural Machine Translation. *ArXiv e-prints*.

Philipp Koehn, Hieu Hoang, Alexandra Birch, Chris Callison-Burch, Marcello Federico, Nicola Bertoldi, Brooke Cowan, Wade Shen, Christine Moran, Richard Zens, Chris Dyer, Ondřej Bojar, Alexandra Constantin, and Evan Herbst. Moses: Open source toolkit for statistical machine translation. In *Proceedings of the 45th Annual Meeting of the ACL on Interactive Poster and Demonstration Sessions*, ACL '07, pages 177–180, Stroudsburg, PA, USA, 2007. Association for Computational Linguistics. URL http://dl.acm.org/citation.cfm?id=1557769.1557821.

Pablo Loyola, Edison Marrese-Taylor, and Yutaka Matsuo. A neural architecture for generating natural language descriptions from source code changes. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 287–292. Association for Computational Linguistics, 2017. doi:10.18653/v1/P17-2045. URL http://www.aclweb.org/anthology/P17-2045.

Thang Luong, Hieu Pham, and Christopher D. Manning. Effective approaches to attention-based neural machine translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing, EMNLP 2015, Lisbon, Portugal, September 17-21, 2015*, pages 1412–1421, 2015. URL http://aclweb.org/anthology/D/D15/D15-1166.pdf.

Vijayaraghavan Murali, Swarat Chaudhuri, and Chris Jermaine. Bayesian sketch learning for program synthesis. *CoRR*, abs/1703.05698, 2017. URL http://arxiv.org/abs/1703.05698.

Yurii E Nesterov. A method for solving the convex programming problem with convergence rate o (1/kˆ 2). In *Dokl. Akad. Nauk SSSR*, volume 269, pages 543–547, 1983.

Maxim Rabinovich, Mitchell Stern, and Dan Klein. Abstract syntax networks for code generation and semantic parsing. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1139–1149. Association for Computational Linguistics, 2017. doi:10.18653/v1/P17-1105. URL http://www.aclweb.org/anthology/P17-1105.

Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. *SIGPLAN Not.*, 49(6):419–428, June 2014. ISSN 0362-1340. doi:10.1145/2666356.2594321. URL http://doi.acm.org/10.1145/2666356.2594321.

Veselin Raychev, Martin Vechev, and Andreas Krause. Predicting program properties from "big code". In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 111–124, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3300-9. doi:10.1145/2676726.2677009. URL http://doi.acm.org/10.1145/2676726.2677009.

Veselin Raychev, Pavol Bielik, and Martin Vechev. Probabilistic model for code with decision trees. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 731–747, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4444-9. doi:10.1145/2983990.2984041. URL http://doi.acm.org/10.1145/2983990.2984041.

Reuven Rubinstein. The cross-entropy method for combinatorial and continuous optimization. *Methodology and computing in applied probability*, 1(2):127–190, 1999.

Reuven Y Rubinstein. Combinatorial optimization, cross-entropy, ants and rare events. *Stochastic optimization: algorithms and applications*, 54:303–363, 2001.

Alexander M. Rush, Sumit Chopra, and Jason Weston. A neural attention model for abstractive sentence summarization. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing, EMNLP 2015, Lisbon, Portugal, September 17-21, 2015*, pages 379–389, 2015. URL http://aclweb.org/anthology/D/D15/D15-1044.pdf.

Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1715–1725, Berlin, Germany, August 2016. Association for Computational Linguistics. URL http://www.aclweb.org/anthology/P16-1162.

Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of machine learning research*, 15(1): 1929–1958, 2014.

Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 6000–6010, 2017.

Pengcheng Yin and Graham Neubig. A syntactic neural model for general-purpose code generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 440–450. Association for Computational Linguistics, 2017. doi:10.18653/v1/P17-1041. URL `http://www.aclweb.org/anthology/P17-1041`.