DeepSumm - Deep Code Summaries using Neural Transformer Architecture

Vivek Gupta
vkgupta@stanford.edu
(Submitting under option 3)

Abstract

Source code summarizing is a task of writing short, natural language descriptions of source code behaviour during run time. Such summaries are extremely useful for software development and maintenance but are expensive to manually author, hence it is done for small fraction of the code that is produced and is often ignored. Automatic code documentation can possibly solve this at a low cost. This is thus an emerging research field with further applications to program comprehension, and software maintenance. Traditional methods often relied on cognitive models that were built in the form of templates and by heuristics and had varying degree of adoption by the developer community. But with recent advancements, end to end data-driven approaches based on neural techniques have largely overtaken the traditional techniques. Much of the current landscape employs neural translation based architectures with recurrence and attention which is resource and time intensive training procedure. In this paper, we employ neural techiques to solve the task of source code summarizing and specifically compare NMT based techniques to more simplified and appealing Transformer architecture on a dataset of Java methods and comments. We bring forth an argument to dispense the need of recurrence in the training procedure. To the best of our knowledge, transformer based models have not been used for the task before. With supervised samples of more than 2.1m comments and code, we reduce the training time by **more than 50%** and achieve the BLEU score of **17.99** for the test set of examples.

1 Introduction

A “summary” of source code is a brief natural language description of that section of source code. One of the most common targets for summarization are the subroutines or more generally known methods in a program; for example, the one-sentence descriptions of Java methods widely used in automatically-formatted documentation e.g. JavaDocs as shown below:

Method:

```
public void render(GameData data)
{
    setText(Message.render(data, type.getPattern(), attributes))
}
```

Comment:

```
Renders the message and updates the message text
```

These comments are useful because they help programmers understand the role that the method plays in a program. Empirical studies have repeatedly shown that the understanding the role of the methods in a program is a crucial step to understanding the program's behavior overall. Even a short summary of a method such as shown above can tell a programmer a lot about that method and the program as a whole.

A holy grail of software engineering research has long been to generate these summaries automatically. Forward *et al.*. [1] pointed out in 2002 that "software professionals value technologies that improve automation of the documentation process," and "that documentation tools should seek to better extract knowledge from core resources", such as source code [7]. Tools such as Resharper, JavaDoc and Doxygen automate the format and presentation of documentation, but still leave programmers with the most labor-intensive effort of writing the text and examples.

The research task of generating such summaries is also known as "source code summarization" with much emphasis on summarization of methods. For several years, significant progress was made based on content selection and sentence templates but all these techniques have lately given the way to AI based systems based on big data input.

The inspiration for a vast majority of efforts into AI-based code summarization originates in neural machine translation (NMT) from the natural language processing research community. It is typically thought of in terms of sequence to sequence(seq2seq) learning. In software engineering research, machine translation can be considered as a metaphor for source code summarization: the words and tokens in the body of a method are one sequence, while the desired natural language summary is the target sequence. This application of NMT to code summarization has shown strong benefits in a variety of applications. Much of NMT based methods used for summarization task uses attention based architecture which is based on the method to jointly align and translate - which is either based on recurrence or on convolutions. These underlying architectures have systematic problems in them such as exploding or vanishing gradient problem to address long range dependencies. They are computational complex and are not very interpretable. With this project we aim to solve such problem using Transformer based architecture and compare its performance with traditional recurrent, encoder-decoder based seq2seq model with attention.

2 Related Work

Early methods - Heuristic based/ Template based methods

Hauic *et al.* [2] are often credited with the first attempt to create textual summaries of the code (class or a method) and were the first one to coin the term 'source code summarization'. This early work applied TR methods like Latent Semantic Indexing and combined them with structural information of the code to make effective summaries of the code. A different work was developed by Sridhar *et al.* [3] presented a new method to automatically produce descriptive summary comments for software methods. Based on the method's signature and body, their comment generator found the content of the summary and produced the natural language text that summarizes the method. Moreno *et al.*.[3] suggested a new method to automatically produce natural language summaries for software classes by using stereotype of class.

As in other research areas related to natural language generation, data-driven techniques have largely supplanted template-based techniques due to a much higher degree of flexibility and reduced human effort in template creation.

End-to-end data driven methods

Iyer *et al.* [4] presented an end to end natural language generation system called CODE-NN that jointly performs content selection using an attention mechanism, and surface realization using Long Short Term Memory (LSTM) networks. The system generates a summary one word at a time, guided by an attention mechanism over embeddings of the source code, and by context from previously generated words provided by a LSTM network. The simplicity of the model allows it to be learned from the training data without the burden of feature engineering (Angeli *et al.*., 2010 [5]) or the use of an expensive approximate decoding algorithm [6]. Much of our work is based on the initial setup done by Iyer *et al.* [4] but incorporating recent advancements in achieving the state of the art models. Of note is that the attentional encoder-decoder seq2seq model originally described by Bahdanau *et al.*. [7] is at the core of many of these papers, as it provides strong baseline performance even for many software engineering tasks.

Wei *et al.* [8] explored code summarization task along with code generation task holistically to produce a model that performs summarization task. They exploit the duality between these tow tasks and a propose a dual training framework to train the tow tasks simultaneously. They consider the dualities on probablity and attention weights, and design corresponding regularization terms to constrain the duality.

Hu *et al.* [9] explored the use of AST - Abstract Syntax Trees which capture structures and semantics of Java methods. ASTs are converted into sequences before they are fed into atypical encoder-decoder based seq2seq model. The model itself is an off-the-shelf encoder-decoder; the main advancement is the AST-annotated representation called Structurebased Traversal (SBT). SBT is essentially a technique for flattening the AST and ensuring that words in the code are associated with their AST node type. Alex *et al.*[10] further exploited this concept and combined this with an attentional encoder-decoder system, except with two encoders: one for code/text data and one for AST data. Alex *et al.* [11] also released the dataset which we use in our work.

3 Approach

In this work we mainly consider two set of model architectures. The first architecture, the Neural Machine Translation based encoder-decoder (seq2seq) architecture with attention and the secondly Transformer architecture. Seq2seq based models provides baseline results to compare and contrast with Transformer model.

3.1 Neural Machine Translation Architecture

The workhorse of most Neural Machine Translation (NMT) systems is the attentional encoder-decoder architecture [12] . This architecture originated in work by Bahdanau *et al.* [7] and is explained in great detail by a plethora of very highly regarded sources such as [12], [13]. In this section, we cover only the concepts necessary to understand our approach at a high level.

In an encoder-decoder architecture, there are a minimum of two recurrent neural networks (RNNs). The first, called the encoder, converts an arbitrary-length sequence into a single vector representation of a specified length. The second, called the decoder, converts the vector representation given by the encoder into another arbitrary-length sequence. Encoder generally as a bidirectional RNN while decoder is a unidirectional RNN. The sequence inputted to the encoder is one language e.g. English, and the sequence from the decoder is another language e.g. French. For our experiments, we modeled functions as the source sequence and comments as the target sequence to be fed to this recurrent and attention based encoder-decoder architecture. The final hidden states of the encoder are concatenated and set as the initial state of the decoder after a linear projection. With the encoder hidden states at each step and the decoders hidden state at each step we compute a multiplicative attention score. We compute a softmax over these scores and multiply the scores with the hidden states and accumulate them to form the attention score. Attention score is concatenated with the decoder hidden states and sent through a linear layer and we compute the softmax over the possible words of the output vocabulary to produce the next word prediction.

3.2 Our approach - Transformer architecture

Much of our work is based on the seminal work done by Vaswani *et al.* [14]. It is built by many multi-head self attentions blocks. We will delve into the details of this architecture in this section.

3.2.1 Encoder and Decoder Stacks

The encoder is composed of a stack of N identical layer. N is a parameter that can be tuned. Each stack of encoder has a multi-head attention layer followed by position wise feed forward network. A residual connection is applied follwoed by layer normalization [15] as shown in the figure 2

Similarly, the decoder is composed of N identical layers and has same set of layers as an encoder but in between the two layers is inserted a masked multihead attention layer that attends to the output of the final encoder layer.

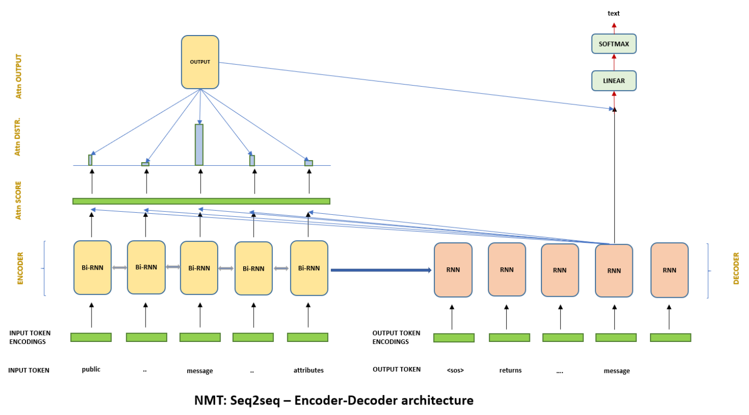

Figure 1: NMT:Seq2seq architecture

3.2.2 Attention functions

An attention function can be described as mapping a query and a set of key-value pairs to an output, where the query, keys, values, and output are all vectors. The output is computed as a weighted sum of the values. In scaled dot product attentions, the input consists of queries and keys of dimension d_k, and the values of dimension d_v. We compute dot products of the query with all the keys, scale down the results by a factor of $\sqrt{d_k}$, and apply a softmax to obtain the weights on the values. While implementing it in vectorized form, we compute this by packing all the queries in a matrix, Q. The keys are also packed in the matrices K and V. Since this is computed on the same sequence, we call this function as a Self Attention. Thus,

$$\text{Self-Attention}(Q,K,V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V \tag{1}$$

Instead of performing a single self attention function, Vaswani *et al.* [14] found beneficial to to have multiple instances of these self attention functions that operate on multiple (h times) projections of the Q, K and V. The outputs are then concatenated and once again projected, resulting in the final values.

$$\text{MultiHead Attention}(Q,K,V) = \text{Concat}(\text{head}_1,\ldots,\text{head}_h)W^O \tag{2}$$

where each head$_i$

$$\text{head}_i = \text{Self-Attention}(QW_i{}^Q, KW_i{}^K, VW_i{}^V) \tag{3}$$

Where the projections are parameter matrices

- $\text{W}_i{}^Q \in \mathbb{R}^{d_{model} \times d_k}$
- $\text{W}_i{}^K \in \mathbb{R}^{d_{model} \times d_k}$
- $\text{W}_i{}^V \in \mathbb{R}^{d_{model} \times d_v}$
- $\text{W}^O \in \mathbb{R}^{hd_v \times d_{model}}$

Using our experiments, we tune for h, d_k , d_v , d_{model}

The overall flow of tensors is shown in the figure 2:

3.3 Threats to the approach

We acknowledge the following issues or threats to the approach:

- The results are very much biased to the Java dataset that is used to train the model. With different dataset, the results could be different.
- We were not able to exploit structural composition of the methods and solely relied on the resulting tokens and ignored casing. In production scenario, both these are very important considerations to the comments that go with the code. Our main objective, however is to compare recurrent versus transformer based sequence trasduction procedure.

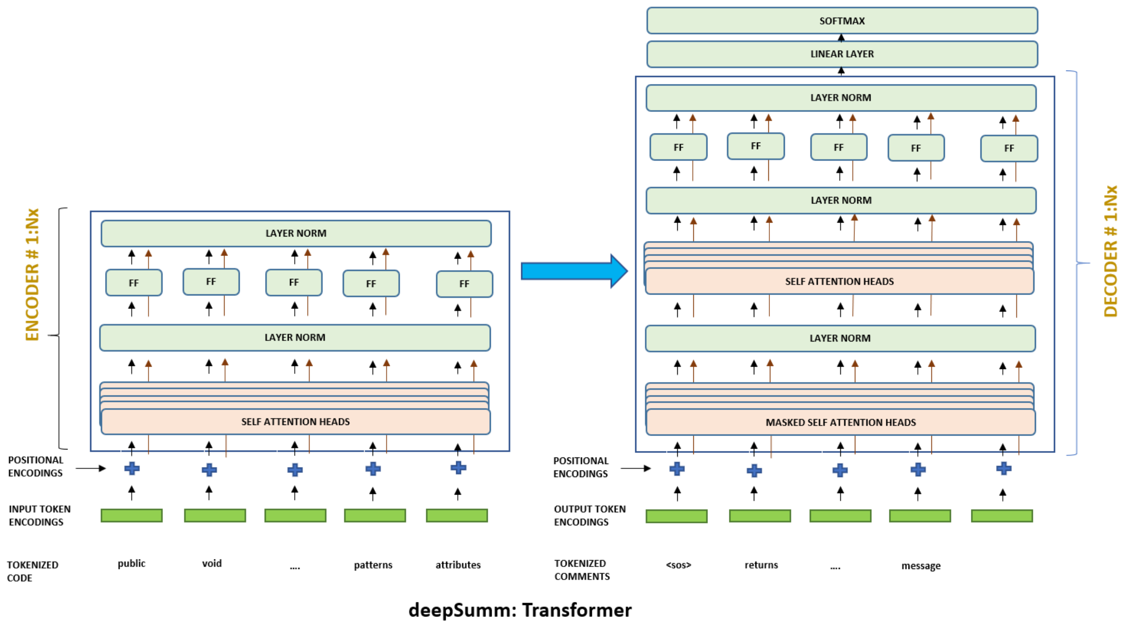

Figure 2: Transformer: Flow of tensors

4 Experiments

4.1 Data

The dataset we use in this paper is based on the dataset provided by LeClair *et al.* [10]. We used this dataset because it is both the largest and most recent in source code summarization. LeClair *et al.* provided the dataset after minimal initial processing that filtered for Java methods with JavaDoc comments in English, and removed methods over 100 words long and comments >13 and <3 words. The result is a dataset of 2.1m Java methods and associated comments. Furthermore, LeClair *et al.* provided 3 groups of data,

- Raw data set: 51m Java methods along with original source files.
- Filtered data set: 2.1m Java method and comments with unprocessed source code and unprocessed comments.
- Tokenized data set: 2.1m Java method and comments. Preprocessed source code with special characters removed, camel case split, lowercased. Comments are the first line of the javadoc lowercased with special characters removed. Our Train, validation and test sets were formed from this group of the data.

Due to the limited resources for the work, we did not use the entire dataset but only a partial set of it for the large runs of experiments. We randomly selected three training sets from the original dataset which are described in the below table and ran various experiments on them to test and validate both the approaches.

Table 1: Supervised Data set

Set Label	Training	Validation	Testing
Small Set	100,000	3000	3000
Medium Set	1m	5000	5000
Large Set	2.1m	10,000	10,000

We target the problem of source code summarization of methods – automatic generation of natural language descriptions of methods. Specifically, we target summarization of Java methods, with the objective of creating method summaries like those used in JavaDocs. While we limit the scope of the experiments in this paper to Java dataset, in principle the techniques described in this paper are applicable to any programming language.

4.2 Evaluation method

To evaluate the performance of the model in train and validation sets, we use perplexity measure and to evaluate a model in the test set we use BLEU score [16].

Perplexity is the measure of how well a probability distribution or a model such as ours, in a train and validation setting, predicts a sample. It is used to compare probabilistic natural language models . a low perplexity indicates the probability distribution is good at predicting the sample. The benefit of using perplexity is that it is easier to calculate.

To measure the performance of the model we used BLEU score or BiLingual Evaluation Understudy [16]. BLEU is a measure of the number of matching Ngrams between the machine's output and a reference translation. BLEU has been the traditional standard for MT systems and is considered to be the correspondence between a machine's output and that of a human. Technically speaking we used Pytorch's, Torchtext implementation of BLEU [17], `torchtext.data.metrics.bleu_score`

4.3 Experimental details

To get results greedily, we performed a screening test on the hyperparameters and model configurations. To accomplish this we used Small dataset in the table 1 and ran many experiments to get most likelihood set of hyper parameters that might work better. We further screened the parameters obtained in the Medium dataset in table 1. We finally consumed these hyperparameters and configurations in the model with Large dataset.

We used grid search on learning rate, encoder, decoder layer(s), encoder, decoder heads and dimensionality of the model. For the final set of results, we fixed learning rate to the best and the stable rate from the experiments we conducted.

The baselines used for the model was from the work done by LeClair *et al.* [10] in producing models namely **ast-attendgru** and **attendgru**. LeClair *et al.* exploited the structural composition of the tokens in the methods along with the tokens themselves in ast-attendgru model, and for attendgru they used a vanilla off the shelf attention based seq2seq model. Since we wanted to experience the difference between recurrent based and a transformer based architecture, deepsumm-attention from section 3.1 on small dataset was our another baseline. deepsumm-transformer from section 3.2 and figure 2 was the model under observation for the 3 different classes of our dataset.

4.4 Results

4.4.1 Baseline

Table 2: Baselines

Model	Dataset	BLEU
ast-attendgru	Large	19.6
attendgru	Large	19.4
deepsumm-attention	Small	10.70
deepsumm-transformer	Small	11.86
deepsumm-transformer	Medium	16.95
deepsumm-transformer	Large	17.99

4.4.2 Detailed results

N: Encoder, Decoder layers, d_{model}: Input dimension of the token embedding, ***batch***: Batch size, ***h***: encoder, decoder heads, ***Size***: Number of parameters(x10^6),

Table 3: deepSumm - Transformer

Dataset	***N***	d_{model}	***batch***	***h***	d_k,d_v	***Epochs***	PPL_{test}	***BLEU***	***Size***
Small	3	256	128	8	512	10	33.102	11.43	17.5
	2						30.937	11.8	16.3
	4						43.732	9.45	***18.7***
	1						32.563	***11.86***	14.8
	2	128					35.088	11.63	7.7
	1						35.854	11.35	7.2
	1	512					30.47	11.84	31.1
	2						***30.42***	11.59	35.3
	2			4			31.186	11.78	16.3
Medium	3	256	128	8	512	5	18.003	16.41	***53.5***
	2						***17.928***	16.8	52.2
	1						19.415	16.32	50.9
	2			4			18.376	***16.95***	52.2
Large	3	256	256	8	512	5	***16.152***	17.91	***76.6***
	2						16.431	17.81	75.3
	1						17.419	17.81	75.3
	2			4			16.539	***17.99***	75.2

We expected and observed a drop in training time of the deepsumm transformer based models from its attention based recurrent architectures by atleast half. This enabled us to run the experiment on the larger dataset. This is inline with what we learnt from the work done by Vaswani *et al.* [14]. We also hoped to deepsumm transformer model to outperform the recurrent based baseline in both processing time and the performance metric. The results obtained were in line.

We were however not able to come close to the work done and results obtained by LeClair on the same dataset [10]. We suspect not having structural composition embedded into the model is contributing to the loss. We also strongly suspect that our training time and exploration of the gradient space via a static learning rate is not optimal and may also have come strongly contributed to the shortcoming. We were only able to train four models for the larger dataset and for only 5 epochs.

5 Analysis

We take a look at 2 examples in this section to perform inference on the generated natural language summaries of the methods. The examples are handpicked for illustration purposes only.

1. Original Code:

```
public PartVO getChild(){
        return child;
    }
```

   Tokenized input to the model: "public part vo get child return child".

   Reference comment: "get the child part".

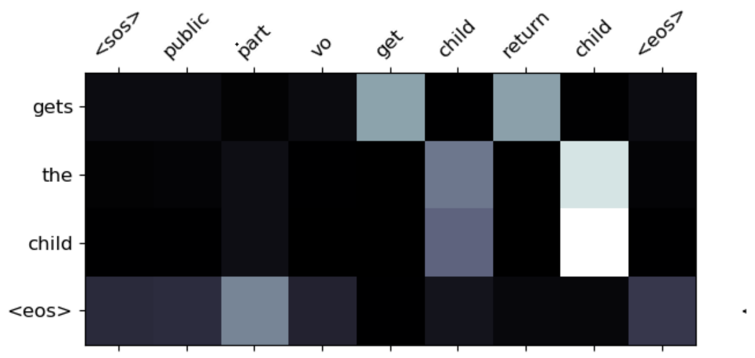

Figure 3: Attention distribution for the first example

deepSumm prediction: "gets the child".

We see this example as an acceptable generation of the summary as compared to the reference. The weights on the attention distribution shows that to generate "gets" the decoder is paying attention to "get" and "return" tokens from the source sequence which is justifiable. It follows the suit for the predicting "child".

2. Original Code:

```
public double getOxygenConsumptionRate() {
        return getValueAsDouble(OXYGEN_CONSUMPTION_RATE);
    }
```

Tokenized input to the model: "public double get oxygen consumption rate return get value as double oxygen consumption rate".

Reference comment: "gets the oxygen consumption rate".

deepSumm prediction: "returns the water consumption rate for this object".

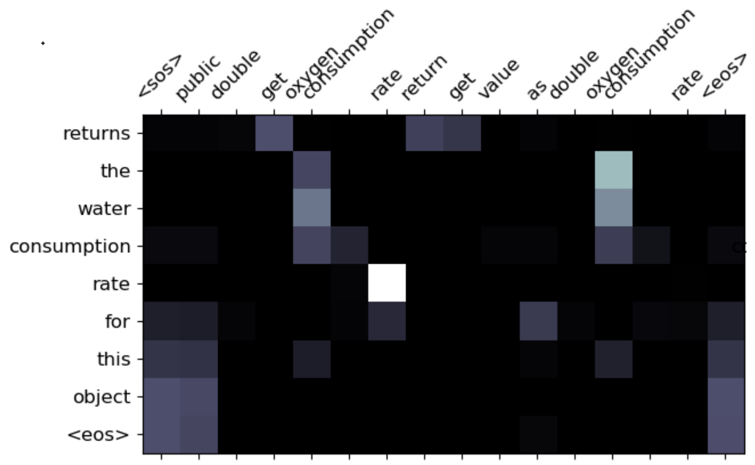

Figure 4: Attention distribution for the 2nd example

We see this example as a problematic generation of the summary as compared to the reference. Looking at the weights, the model while decoding the third token, paid attention to the consumption and predicted water. We think that the greedy algorithm paid the price of the faulty search here.

3. Original Code:

```
public void setMaximumColorDepth(String value) {
        this.maximumColorDepth = value;
    }
```

Tokenized input to the model: "public double get oxygen consumption rate return get value as double oxygen consumption rate".

Reference comment: "gets the oxygen consumption rate".

deepSumm prediction: "returns the water consumption rate for this object".

Recurrent attention(Small model) prediction: "sets the maximum color depth of the color" We see this example as a good generation of the summary as compared to the reference and

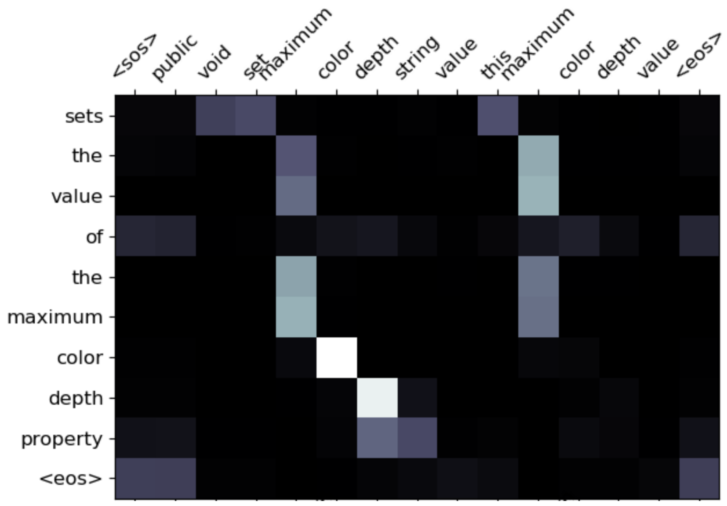

Figure 5: Attention distribution for the 3rd example

as compared tot he attentive RNN model which had a poor prediction.

6 Conclusion and Future work

With this work, we presented a model that effectively generates natural language summaries from program methods written in Java. Furthermore, we were able to create two contrasting models, one that uses Recurrence and other that uses state of the art, Transformer architecture. We observed the benefits of Transformer as cited by Vaswani *et al.* [14] first hand by running various experiments on both these models. Our transformer model was able to outperform recurrence based models by more that a BLEU score in the same dataset. The training time also improved. Empirically with this dataset by a factor of half when compared with recurrence. While we were not able to achieve state of the art results in the task but we think that employing effective structural information along with programming constructs in representing the methods will come handy along with more efficient strategy in exploring the gradient space and training the network for longer time. Furthermore, exploring ensemble methods in NLP as done by LeClair *et al.* [10] will also be a great strategy. We think, that these will be helpful next steps to take the next step in this line of work.

References

[1] Andrew Forward and Timothy C. Lethbridge. The relevance of software documentation, tools and technologies: A survey. In *Proceedings of the 2002 ACM Symposium on Document Engineering*, DocEng '02, page 26–33, New York, NY, USA, 2002. Association for Computing Machinery.

[2] Sonia Haiduc, Jairo Aponte, Laura Moreno, and Andrian Marcus. On the use of automated text summarization techniques for summarizing source code. In Giuliano Antoniol, Martin Pinzger, and Elliot J. Chikofsky, editors, *WCRE*, pages 35–44. IEEE Computer Society, 2010.

[3] Laura Moreno, Jairo Aponte, Giriprasad Sridhara, Andrian Marcus, Lori L. Pollock, and K. Vijay-Shanker. Automatic generation of natural language summaries for java classes. *2013 21st International Conference on Program Comprehension (ICPC)*, pages 23–32, 2013.

[4] 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 (Volume 1: Long Papers)*, pages 2073–2083, Berlin, Germany, August 2016. Association for Computational Linguistics.

[5] Gabor Angeli, Percy Liang, and Dan Klein. A simple domain-independent probabilistic approach to generation. In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*, pages 502–512, Cambridge, MA, October 2010. Association for Computational Linguistics.

[6] Ioannis Konstas and Mirella Lapata. A global model for concept-to-text generation. *J. Artif. Int. Res.*, 48(1):305–346, October 2013.

[7] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate, 2014.

[8] Bolin Wei, Ge Li, Xin Xia, Zhiyi Fu, and Zhi Jin. Code generation as a dual task of code summarization, 2019.

[9] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension*, ICPC '18, page 200–210, New York, NY, USA, 2018. Association for Computing Machinery.

[10] Alexander LeClair, Siyuan Jiang, and Collin McMillan. A neural model for generating natural language summaries of program subroutines, 2019.

[11] Alexander LeClair and Collin McMillan. Recommendations for datasets for source code summarization, 2019.

[12] Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. Effective approaches to attention-based neural machine translation, 2015.

[13] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks, 2014.

[14] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.

[15] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization, 2016.

[16] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: A method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ACL '02, page 311–318, USA, 2002. Association for Computational Linguistics.

[17] Torch Contributors. Torchtext.data.metrics, 2018.