Improved Code Summarization via a Graph Neural Network

Alexander LeClair
aleclair@nd.edu
University of Notre Dame
South Bend, IN

Lingfei Wu
wuli@us.ibm.com
IBM Research
Yorktown Heights, NY

Sakib Haque
shaque@nd.edu
University of Notre Dame
South Bend, IN

Collin McMillan
cmc@nd.edu
University of Notre Dame
South Bend, IN

ABSTRACT

Automatic source code summarization is the task of generating natural language descriptions for source code. Automatic code summarization is a rapidly expanding research area, especially as the community has taken greater advantage of advances in neural network and AI technologies. In general, source code summarization techniques use the source code as input and outputs a natural language description. Yet a strong consensus is developing that using structural information as input leads to improved performance. The first approaches to use structural information flattened the AST into a sequence. Recently, more complex approaches based on random AST paths or graph neural networks have improved on the models using flattened ASTs. However, the literature still does not describe the using a graph neural network together with source code sequence as separate inputs to a model. Therefore, in this paper, we present an approach that uses a graph-based neural architecture that better matches the default structure of the AST to generate these summaries. We evaluate our technique using a data set of 2.1 million Java method-comment pairs and show improvement over four baseline techniques, two from the software engineering literature, and two from machine learning literature.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**;

KEYWORDS

Automatic documentation, neural networks, deep learning, artificial intelligence

ACM Reference Format:
Alexander LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. 2020. Improved Code Summarization via a Graph Neural Network. In *28th International Conference on Program Comprehension (ICPC '20), October 5–6, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3387904.3389268

ICPC '20, October 5–6, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7958-8/20/05…$15.00
https://doi.org/10.1145/3387904.3389268

1 INTRODUCTION

Source code summarization is the task of writing brief natural language descriptions of code [15, 19, 29, 37]. These descriptions have long been the backbone of developer documentation such as JavaDocs [27]. The idea is that a short description allows a programmer to understand what a section of code does and that code's purpose in the overall program, without requiring the programmer to read the code itself. Summaries like "uploads log files to the backup server" or "formats decimal values as scientific notation" can give programmers a clear picture of what code does, saving them time from comprehending the details of that code.

Automatic code summarization is a rapidly expanding research area. Programmers are notorious for neglecting the manual effort of writing summaries themselves [12, 26, 45, 48], and automation has long been cited as a desirable alternative [17]. The term "source code summarization" was coined around ten years ago [19] and since that time the field has proliferated. At first, the dominant strategy was based on sentence templates and heuristics derived from empirical studies [15, 36, 40, 44, 50, 51]. Starting around 2016, data-driven strategies based on neural networks came to the forefront, leveraging gains from both the AI/NLP and mining software repositories research communities [3, 23, 25, 29].

These data-driven approaches were inspired by neural machine translation (NMT) from natural language processing. In NMT, a sentence in one language e.g. English is translated into another language e.g. Spanish. A dataset of millions of examples of English sentences paired with Spanish translations is required. A neural architecture based on the encoder-decoder model is used to learn the mapping between words and even the correct grammatical structure from one language to the other based on these examples. This works well because both input and output languages are sequences of roughly equal length, and mappings of words tend to exist across languages. The metaphor in code summarization is to treat source code as one language input and summaries as another. So code would be input to the same models' encoder, and summaries to the decoder. Advances in repository mining made it possible to gather large datasets of paired examples [33].

But evidence is accumulating that the metaphor to NMT has major limits [21]. Source code has far fewer words that map directly to summaries than the NMT use case [30]. Source code tends not to be of equal length to summaries; it is much longer [15, 39]. And crucially, source code is not merely a sequence of words. Code is a complex web of interacting components, with different classes,

routines, statements, and identifiers connected via different relationships. Software engineering researchers have long recognized that code is much more suited to graph or tree representations that tease out the nuances of these relationships [6, 42]. Yet, the typical application of NMT to code summarization treats code as a sequence to be fed into a recurrent neural network (RNN) or similar structure designed for sequential information.

The literature is beginning to recognize the limits to sequential representations of code for code summarization. Hu *et al.* [23] annotate the sequence with clues from the abstract syntax tree (AST). LeClair *et al.* [29] expand on this idea by separating the code and AST into two different inputs. Alon *et al.* [3] extract paths from the AST to aid summarization. Meanwhile, Allamanis *et al.* [2] propose using graph neural networks (GNNs) to learn representations of code (though for the problem of code generation, not summarization). These approaches all show how neural networks can be effective in extracting information from source code better in a graph or tree form than in a sequence of tokens, and using that information for downstream tasks such as summarization.

What is missing from the literature is a thorough examination of **how** graph neural networks improve representations of code based on the AST. There is evidence *that* GNN-based representations improve performance, but the degree of that improvement for code summarization has not been explored thoroughly, and the reasons for the improvement are not well understood.

In this paper, we present an approach for improving source code summarization using GNNs. Specifically, we target the problem of summarizing program subroutines. Our approach is based on the `graph2seq` model presented by Xu *et al.* [58], though with a few modifications to customize the model to a software engineering context. In short, we use the GNN-based encoder of `graph2seq` to model the AST of each subroutine, combined with the RNN-based encoder used by LeClair *et al.* [29] to model the subroutine as a sequence. We demonstrate a 4.6% BLEU score improvement for a large, published dataset [30] as compared to recent baselines. In an experiment, we use techniques from the literature on explainable AI to propose explanations for why the approach performs better and in which cases. We seek to provide insights to guide future researchers. We make all our data, implementations, and experimental framework available via our online appendix (Section 9).

2 PROBLEM, SIGNIFICANCE, SCOPE

We target the problem of automatically generating summaries of program subroutines. To be clear, the input is the source code of a subroutine, and the output is a short, natural language description of that subroutine. These summaries have several advantages when put into documentation such as decreased time to understand code [17], improved code comprehension [11, 54], and to making code more searchable [22]. Programmers are notorious for consuming high quality documentation for themselves, while neglecting to write and update it themselves [17]. Therefore, recent research has focused on automating the documentation process. Current research has had success generating summaries for a subset of methods that are generally shorter and use simpler language in both the code and reference comment (e.g. setters and getters), but have had a problem with methods that have more complex structures or language [29]. A similar situation for other SE research problems has been helped by various graph representations of code [2, 16], but using graph representations is only starting to be accepted for code summarization [3, 16]. Graph representations have the potential to improve code summarization because, instead of using only a sequence of code tokens as input, the model can access a rich variety of relationships among tokens.

Automatic documentation has a large potential impact on how software is developed and maintained. Not only would automatic documentation reduce the time and energy programmers spend reading and writing software, having a high level summary available has been shown to improve results in other SE tasks such as code categorization and code search [22, 28].

3 BACKGROUND AND RELATED WORK

This section discusses some of the previous work relevant to this work and source code summarization.

3.1 Source Code Summarization

Source code summarization research can be broadly categorized as either 1) heuristic/template-driven approaches or 2) more recent AI/Data-driven approaches. Heuristic-based approaches for source code summarization started to gain popularity in 2010 with work done by Haiduc *et al.* [20]. In their work, text retrieval techniques and latent semantic indexing (LSI) were used to pick important keywords out of source code, then those words are considered the summary. Early work done by Haiduc *et al.* and others have helped inspire other work using extractive summarization techniques based on TF-IDF, LSI, and LDA to create a summary. Heuristic-based approaches are less related to this work than data-driven approaches, so due to space limitations we direct readers to surveys by Song *et al.* [49] and Nazar *et al.* [41] for additional background on the topic.

This paper builds on the current work done with data-driven approaches in source code summarization which have dominated NLP and SE literature since around 2015. In Table 1, we divide recent work into two groups by their use of the AST as an input to the model. Then we further divide related work by the following six attributes:

(1) Src Code - A model uses the source code sequence as input, not as part of the AST.
(2) AST - A model uses the AST as input.
(3) API - A model uses API information.
(4) FlatAST - Using a flattened version of the AST as model input.
(5) GNN - The model uses a form of graph neural network for node/edge embedding.
(6) Paths - Using a path through the AST as input to the model.

A brief history of the related data-driven work starts with Iyer *et al* [25]. In their work they used stack overflow questions and responses where the title of the post was considered the high level summary, and the source code in the top rated response was used as the input. The model they developed was an attention based sequence to sequence model similar to those used in neural machine translation tasks [53]. To expand on this idea, Hu *et al.* [24] added API information as an additional input into the model. They found that the model was able to generate better responses if it had access to information provided by API calls in the source code.

Later, Hu *et al.* [23] developed a structure based traversal (SBT) method for flattening the AST into a sequence that keeps words in

	Src Code	AST	API	FlatAST	GNN	Paths
2016 Iyer *et al.* [25]	x					
2017 Loyola *et al.* [34]	x					
2017 Lu *et al.* [35]	x		x			
2018 Hu *et al.* [24]	x		x			
2018 Liang *et al.* [31]	x	x				
2018 Hu *et al.* [23]	x	x		x		
2018 Wan *et al.* [56]		x			x	
2019 LeClair *et al.* [29]	x	x		x		
2019 Alon *et al.* [3]		x		x		x
2019 Fernandes *et al.* [16]		x			x	

Table 1: Comparison of recent data-driven Source Code Summarization research categorized by the data, architectures, and approaches used. The approaches in the upper table use only the source code sequence as input to their models, while the bottom table approaches use the AST or a combination of AST and source code.

the code associated with their node type. The SBT sequence is a combination of source code tokens and AST structure which was then input into an off the shelf encoder/decoder model. LeClair *et al.* [29] built upon this work by creating a multi-input model that used the SBT sequence with all identifiers removed as the first input, and the source code tokens as the second. They found that if you decouple the structure of the code form the code itself that the model improved its ability to learn that structure.

More recently, Alon *et al.* [3] in 2018 proposed a source code summarization technique that would encode each pairwise path between nodes in the AST. They would then randomly select a subset of these paths for each iteration in training. These paths were then encoded and used as input to a standard encoder/decoder model. They found that encoding the AST paths allowed the model to generalize to unseen methods more easily, as well as providing a level of regularization by randomly selecting a subset of paths each training iteration.

Then in 2019 Fernandes *et al.* [16] developed a GNN based model that uses three graph representations of source code as input 1) next token, 2) AST, and 3) last lexical use. To represent these three graphs they used a shared node setup where each graph represented a different set of edges between source code tokens. Using this approach they observed a better "global" view of the method and had success with maintaining the central named entity from the method. Fernandes' observation is an important clue that there is additional information embedded in the source code beyond the sequence of tokens, this motivates the use of a GNN for the AST as a separate input in our work.

3.2 Neural Machine Translation

For the last six years work in neural machine translation (NMT) has been dominated by the encoder-decoder model architecture developed by Bahdanau *et al.* [5]. The encoder-decoder architecture can be thought of as two separate models, one to encode the input (e.g. English words) into a vector representation, and one to decode that representation into the desired output (e.g. German tokens).

Commonly, encoder-decoder models use a recurrent layer (RNN, GRU, LSTM, etc.) in both the encoder and decoder. Recurrent layers are effective at learning sequence information because for each token in a sequence, information is propagated through the layer [52]. This allows each token to affect the following tokens in the sequence. Some of the common recurrent layers such as the GRU and LSTM also can return state information at each time step of the input sequence. The state information output from the encoder is commonly used to seed the initial state of the decoder improving translation results [53].

Another more recent addition to many encoder-decoder models is the attention mechanism. The intuition behind attention is that not all tokens in a sequence are of equal importance to the final output prediction. What attention tries to do is to learn what words are important and map input tokens to output tokens. It does this by taking the input sequence at every time step and the predicted sequence at a time step and tries to determine which time step in the input will be most useful to predict the next token in the output.

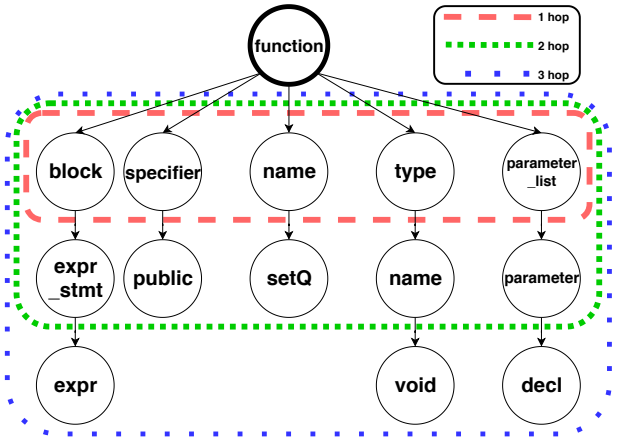
3.3 Graph Neural Networks

Graph Neural Networks are another key background technology to this paper. A recent survey by Wu et al. [57] categorizes GNNs into four groups:

(1) Recurrent Graph Neural Networks (RecGNNs)
(2) Convolutional Graph Neural Networks (ConvGNNs)
(3) Graph Autoencoders (GAEs)
(4) Spatial-temporal Graph Neural Networks (STGNNs)

We will focus on ConvGNNs in this section because they are well suited for this task, and it is what we use in this paper. ConvGNNs were developed after RecGNNs and were designed with the same idea of message passing between nodes. They have also been shown to encode spatial information better than RecGNNs and are able to be stacked, improving the ability to propagate information across nodes [57]. ConvGNNs take graph data and learn representations of nodes based on the initial node vector and its neighbors in the graph. The process of combining the information from neighboring nodes is called "aggregation." By aggregating information from neighboring nodes a model can learn representations based on arbitrary relationships. These relationships could be the hidden structures of a sentence, the parts of speech [8], dependency parsing trees [59], or the sequence of tokens [7]. ConvGNNs have been used for similar tasks before, such as in graph2seq for semantic parsing [58] and natural question generation [8].

ConvGNNs also allow nodes to get information from other nodes that are further than just a single edge or "hop" away. In the figure below we show an example partial AST and what 1, 2, and 3 hops



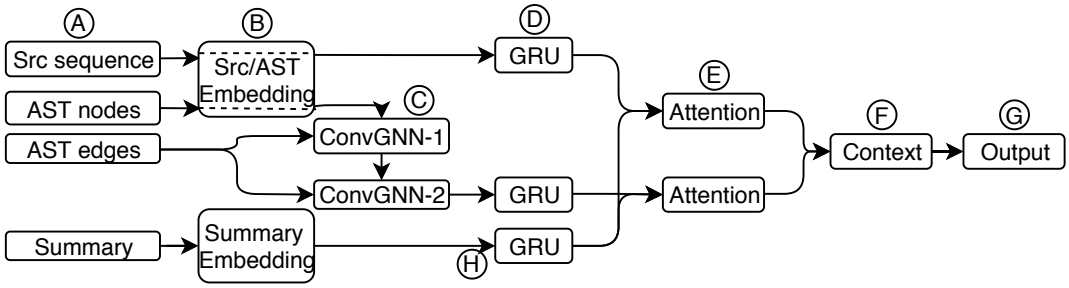


Figure 1: High level diagram of model architecture for 2-hop model

look like for the token 'function'. Each time a hop is performed, the node gets information from its neighboring nodes. So, on the first hop the token 'function' aggregates information from the nodes 'block', 'specifier', 'name', 'type', and 'parameter_list'. In the next hop that occurs, the 'function' node will still only combine information from its neighbors, but now each of those nodes will have aggregate information from their children. For example, the node 'block' will contain information from the 'expr_stmt' node. Then when the 'function' node aggregates the 'block' node, it has information from both 'block' and 'expr_stmt'.

There are several aggregation strategies which have been shown to have different performance for different tasks [16, 57]. A common aggregation strategy is to sum a node vector with its neighbors and then apply an activation on that node, but there are many schemes that can be used to combine node information. Some other approaches to this are pooling, min, max, and mean. Xu *et al.* discuss different node and edge aggregation methods in their paper on creating sequences from graphs. They found that in most cases a mean aggregator out performed other types of aggregators, including one using an LSTM.

4 APPROACH

This section provides the details of our approach. Our model is based off the neural model proposed by LeClair *et al.* [29] and builds on that work by using ConvGNNs discussed in the previous section. In a nutshell, our approach works in 5 steps:

(1) Embed the source code sequence and the AST node tokens.
(2) Encode the embedding output with a recurrent layer for the source code token sequence and a ConvGNN for the AST nodes and edges.
(3) Use an attention mechanism to learn important tokens in the source code and AST.
(4) Decode the encoder outputs.
(5) Predict the next token in the sequence.

4.1 Model Overview

An overview of our model is in Figure 1. In a nutshell, what we did was modify the model on the multi-input encoder-decoder proposed by LeClair *et al.* to use a ConvGNN instead of a flattened AST. Notice in area A of Figure 1 that our model has four inputs 1) the sequence of source code tokens, 2) the nodes of the AST, 3) the edges of the AST, 4) the predicted sequence up to this point. Next, in area B, we embed the inputs using standard embedding layers. The source sequence and AST nodes share an embedding due to a large overlap in vocabulary. Then in area C of Figure 1 the AST nodes are input into the ConvGNN layers, the number of layers here depends on the hop size of the model, and then input into a GRU. The source code sequence goes into a GRU after the embedding in area D. For the decoder in area H, we have an embedding layer feeding into a GRU. We then do two attention mechanisms seen in area E, one between the source code and summary, and the other between the AST and summary. Then in areas F and G we combine the outputs of our attention creating a context vector which is flattened and used to predict the next token in the sequence.

Key Novel Component. The key novel component of this paper is in processing the AST using a ConvGNN and combining the output of the ConvGNN encoder with the output of the source code token encoder. In our approach, the ConvGNN allows the nodes of the AST to learn representations based on their neighboring nodes. Teaching the model information about the structure of the code, and how it relates to the tokens found in the source code sequence. Both the source and and AST encodings are input into separate attention mechanisms with the decoder and are then concatenated. This creates a context vector which we then use in a dense layer to predict the next token in the sequence.

Basically, what we do is combine the structure of the sequence (the AST) with the sequence itself (the source code). Combining the structure of a sequence and the sequence itself into a model has been shown to improve the quality of generated summaries in both SE and NLP literature [23, 29]. In this paper we aim to show that using a neural network architecture that is more suited to the structure of the data (graph vs sequence) we can further improve the models ability to learn complex relationships in the source code.

4.2 Model Details

In this section we will discuss specific model implementation details that we used for our best performing model to encourage reproducibility. We developed our proposed model using Keras [9] and Tensorflow [1]. We also provide our source code and data as an online appendix (details can be found in Section 9).

First, as mentioned in the previous section, our model is based on the encoder-decoder architecture and has four inputs 1) the source code sequence, 2/3) the AST as a collection of nodes along with an adjacency matrix with edge information and 4) the comment generated up to this point which is the input to the decoder. As seen in Figure 1, we use two embedding layers one for the source code and AST and one for the decoder. We use a single embedding layer for both the source code and AST node inputs because they have such a large overlap in vocabulary. The shared embedding

layer has a vocabulary size of 10908 and an embedding size of 100. The decoder embedding layer has a vocabulary size of 10000 and an embedding size of 100. So far, this follows the model proposed by LeClair *et al.* [29].

Next, the model has two encoders, one for the source code sequence and another for the AST. The source code sequence encoder is a single GRU layer with an output length of 256. We have the source code GRU return its hidden states to use as the initial state for the decoder GRU. The second encoder, the AST encoder, is a collection of ConvGNN layers followed by a GRU of length 256. The number of ConvGNN layers depends on the number of hops used, for our best model this was 2-hops as seen in Figure 1.

The ConvGNN that we use for the AST node embeddings takes the AST embedding layer output and the AST edge data as inputs. Then, for each node in the input it sums the current node vector with each of it's neighbors and multiplies that by a set of trainable weights and adds a trainable bias. In our best performing implementation we use a ConvGNN layer for each hop in the model as seen in Figure 1. We also test our model with different numbers of hops, which slightly changes the architecture of the AST encoder of the model by adding additional ConvGNN layers.

Next, we have two attention mechanisms 1) an attention between the decoder and the source code, and 2) between the decoder and the AST. These attention mechanisms learn which tokens from the source code/AST are important to the prediction of the next token in the decoder sequence given the current predicted sequence generated up to this point. The attention mechanisms are then concatenated together with the decoder to create a final context vector. Then, we apply a dense layer to each vector in our final context vector, which we then flatten and use to predict the next token in the sequence.

4.3 Data Preparation

The data set that we used for this project was provided by LeClair *et al.* in a paper on recommendations for source code summarization datasets [30]. LeClair *et al.* describe best practices for developing a dataset for source code summarization and also provide a dataset of 2.1 million Java method comment pairs. They provide their dataset in two versions, 1) a filtered version with the raw, unprocessed version of the methods and comments and 2) the tokenized version where text processing has already been applied. For our baseline comparisons, we use the tokenized version of the dataset provided by LeClair *et al.* allowing us to directly compare results with their work in source code summarization. The dataset did not include ASTs that were already parsed, so we use the SrcML library [10] to generate the associated ASTs from the raw source code.

4.4 Hardware Details

For training, validating, testing of our models we used a workstation with Xeon E1430v4 CPUs, 110GB RAM, a Titan RTX GPU, and a Quadro P5000 GPU. Software used include the following:

Ubuntu 18.04	Python 3.6	CUDA 10
Tensorflow 1.14	Keras 2.2	CuDNN 7

5 EXPERIMENT DESIGN

In this section we discuss the design of our experiments and discuss the methodology, baselines, and metrics used to obtain our results.

5.1 Research Questions

Our research objective is to determine if our proposed approach of using the source code sequence along with a graph based AST and ConvGNN outperform current baselines. We also want to determine why our proposed model may outperform current baselines based on the use of the AST graph and ConvGNN. We ask the following Research Questions (RQs) to explore these situations:

RQ$_1$ What is the performance of our approach compared to the baselines in Section 5.4 in terms of the metrics in Section 5.3?

RQ$_2$ What is the degree of difference in performance caused by the number of graph hops in terms of the metrics in Section 5.3?

RQ$_3$ Is there evidence that the performance differences are due to use of the ConvGNN?

The rationale for RQ$_1$ is to compare our approach with other approaches that use the AST as input. Previous work has already shown that the inclusion of the AST as an input to the model outperforms previous models where no AST information was provided [3, 16, 24, 29]. Some previous work provides the AST as a tree or graph [3, 16], but the source code sequence was not provided to the model. Our proposed model is a logical next step in source code summarization literature and we ask RQ$_1$ to evaluate our model against previous models.

The rationale for RQ$_2$ is to determine what affect (if any) the number of hops has on the generated summaries (a description of ConvGNN hops can be found in Section 3.3). Xu *et al.* [58] discuss the impact of hop size on their work with generating SQL queries. They found that for their test models, the number of hops did not affect model convergence. To test this they generate random directed graphs of sizes 100 and 1000 and trained a model to find the shortest path between nodes, but did not evaluate how hop size affects the task of source code summarization. Since ConvGNNs create a layer for each hop, it becomes computationally expensive to train models with an arbitrarily large number of hops. With RQ$_2$ we hope to build an intuition into how the number of hops affects ConvGNN learning specifically for source code summarization.

The rationale for RQ$_3$ is that discovering *why* a model learned to generate certain summaries can be just as important as evaluation metrics [4, 13, 14, 38, 47, 55]. Doshie *et al.* [14] discuss what interpretability means and offers guidelines to researchers on what they can do to make their models more explainable, while Roscher *et al.* state that *"...explainability is a prerequisite to ensure the scientific value of the outcome"*[46]. As models are developed many factors change, and it is often times not an easy task to determine which factors had the greatest impact on performance. In their work on explainable AI, Arras *et al.* and Samek *et al.* show how you can use visualizations to aid in the process of explainability for text based modeling tasks [4, 47]. With RQ$_3$ we aim to explain what impact the inclusion of the AST as a graph and ConvGNN had on generated summaries.

5.2 Methodology

To answer RQ$_1$, we follow established methodology and evaluation metrics that have become standard in both source code summarization work and neural machine translation from NLP [32, 43]. To start, we use a large well-documented data set from the literature to allow us to easily compare results and baselines. We use the data handling guidelines outlined in LeClair *et al.* [30] so that we do not

Baseline		BLEU-A	BLEU-1	BLEU-2	BLEU-3	BLEU-4	ROUGE-LCS F1
ast-attendgru		18.69	37.13	21.11	14.27	10.90	49.75
graph2seq		18.61	37.56	21.27	14.13	10.63	49.69
code2seq		18.84	37.49	21.36	14.37	10.95	49.69
BiLSTM+GNN->LSTM		19.05	37.70	21.53	14.59	11.11	55.74
ConvGNN Models	# of hops	BLEU-A	BLEU-1	BLEU-2	BLEU-3	BLEU-4	ROUGE-LCS F1
code+gnn+dense	2	19.46	38.71	22.04	14.86	11.31	56.07
code+gnn+BiLSTM	2	**19.93**	**39.14**	**22.49**	**15.31**	11.70	56.08
code+gnn+GRU	1	19.70	38.15	22.12	15.22	**11.73**	**57.15**
code+gnn+GRU	2	19.89	39.01	22.42	15.28	11.70	55.78
code+gnn+GRU	3	19.58	38.48	22.09	15.01	11.52	56.14
code+gnn+GRU	5	19.68	38.89	22.30	15.09	11.46	55.81
code+gnn+GRU	10	19.34	38.68	21.94	14.73	11.20	55.10

Table 2: BLEU and ROUGE-LCS scores for the baselines and our proposed models

have data leakage between our training, validation, and testing sets. Next, we train our models for ten epochs and choose the model with the highest validation accuracy score for our comparisons. Choosing the model with the best validation performance out of ten epochs is a training strategy that has been successfully used in other related work [29]. For RQ_1 we evaluate the best performing model using automated evaluation techniques to compare against our baselines and report in this paper.

For RQ_2 we train five ConvGNN models with all hyper-parameters frozen except for hop size. We test our model using hop sizes of 1,2,3,5, and 10 in line with other related work [58]. We used the model configuration outlined in Section 4 that uses the source code and AST input, as well as a GRU layer directly after the ConvGNN layers. We chose this model because of its performance and its faster training speed compared with the BiLSTM model. To report our results we use the same "best of ten" technique that we use to answer RQ_1, that is, we train each model for ten epochs and report the results on the model with the highest validation accuracy.

For RQ_3 we use a combination of automated tools and metrics such as BLEU [43], ROUGE [32], and visualizations from model weights. Visualizing model weights and parameters has become a popular way to help explain what deep learning models are doing, and possibly give insight into why they generate the output that they do. To help us answer RQ_3 we use concepts similar to those outlined in Samek *et al.* [47] for model visualizations.

5.3 Metrics

For our quantitative metrics we use both BLEU [43] and ROUGE [32] to evaluate our model performance. BLEU scores are a standard evaluation metric in the source code summarization literature [24, 25, 29]. BLEU is a text similarity metric that compares overlapping n-grams between two given texts. While BLEU can be thought of as a precision score: how much of the generated text appears in the reference text. In contrast, ROUGE can be thought of as a recall score: how much of the reference appears in the generated text.

ROUGE is used primarily in text summarization tasks in the NLP literature due to the score allowing multiple references since there may be multiple correct summaries of a text [32]. In our work we do not have multiple reference texts per method, but ROUGE gives us additional information about the performance of our models that BLEU scores alone do not provide. In this paper we report a composite BLEU score, $BLEU_1$ through $BLEU_4$ (n-grams of length 1 to length 4), and ROUGE-LCS (longest common sub-sequence) to have a well rounded set of automated evaluation metrics.

5.4 Baselines

We compare our model against four baselines. These baselines are all from recent work that is directly relevant to this paper. We chose these baselines because they provide comparison for three categories in source code summarization using the AST: 1) flattened AST, 2) using paths through the AST, and 3) using a graph neural network to encode the AST.

Each of these baselines uses AST information as input to the model with different schemes. They also cover a variety of model architectures and configurations. Due to space limitations we do not list all relevant details for each model, but have a more in depth overview in Section 3.1 and in Table 1.

- **ast-attendgru**: In this model LeClair *et al.* [29] use a standard encoder-decoder model and add an additional encoder for the AST. They flatten the AST using the SBT technique outline in Hu *et al.* [23]. Then both the source code tokens and the flattened AST are provided as input into the model. For encoding these inputs they use recurrent layers and then use a decoder with a recurrent layer to generate the predictions. This approach is representative of other approaches that flatten the AST into a sequence.
- **graph2seq**: Xu *et al.* [58] developed a general graph to sequence neural model that generates both node and graph embeddings. In their work they use an SQL query and generate a natural language query based on the SQL. Their implementation propagates both forward and backwards over the graph, and includes node level attention. They achieved state of the art results on an SQL->natural language task using BLEU-4 as a metric. They also evaluate how the number of hops affected the performance of the model finding that any number of hops still converged to similar results, but specific models could perform just as well with less hops lowering the amount of computation needed.
- **code2seq**: Alon *et al* [3] use random pairwise paths through the AST as model input which we discuss more in depth in Section 3.1. They used C# code to generate summaries, while we use Java. They had a variety of configurations that they test, due to this we did a good-faith re-implementation of their base model in an attempt to capture the major contributions of their approach.

- **BILSTM+GNN**: Fernandes *et al.* [16] proposed a model using a BILSTM and GNN trained with a C# data set for code summarization. We reproduced a model using the information outlined in their paper. We trained the model using the Java data set from LeClair *et al.* to create a comparison for our work. In their paper they report results on a variety of model architectures and setups, we include comparison results with a model based on their best performing configuration.

These baselines are not an exhaustive list of relevant work, but they cover recent techniques used for source code summarization. Some other work that we chose not to use for baselines include Hu *et al.* [23], and Wan *et al.* [56]. We chose not to include Hu *et al.* in our baselines because the work done by LeClair *et al.* built upon their work and was shows to have higher performance, and is much closer to our proposed work in this paper. In our proposed model we use the technique outlined in LeClair *et al.* of separating the source code sequence tokens from the AST.

Wan *et al.* [56] is another potential baseline, but we found it unsuitable for comparison in this paper for three reasons: 1) the approach combines an AST+code encoding with Reinforcement Learning (RL), and the RL component adds many experimental variables with effects difficult to distinguish from the AST+code component, 2) the AST+code encoding technique has now been superceded by other techniques which we already use as baselines, and 3) we were unable to reproduce the results in the paper. An interesting question for future work is to study the effects of the RL component in a separate experiment: the RL component of Wan *et al.* is supportive of, rather than a competitor with, the AST+code encoding. We also do not compare against heuristic based approaches. Most data-driven approaches outperform heuristic based approaches in all of the automated metrics, and previous work has already reported the comparisons.

5.5 Threats to Validity

The primary threat to validity for this paper is that the automated metrics we use to score and rate out models may not be representative of human judgement. BLEU and ROUGE metrics can give us a good indication how our model performs compared to the reference text and other models, but there are instances where the model may generate a valid summary that does not align with the reference text. On the other hand, there is no evaluation as to whether a reference comment for a given method is a good summary. The benefit of these automated metrics is that they are fast and have wide use among the source code summarization community. To mitigate the potential pitfalls that using automated metrics may involve, we include an in depth discussion and evaluation of specific examples from our model to help interpret what our model has learned when compared to baselines.

The dataset we use is also another potential threat to validity. While other data sets do exist with other programming languages, for example C# or Python, many of these data sets lack the size and scope of the data set provided by LeClair *et al.*. For example the C# dataset used Fernandes *et al.* has 23 projects, with 55,635 methods having associated documentation. Another common pitfall of datasets described in LeClair *et al.* is that many datasets split data on the function level instead of the project level. This means that functions from the same project can appear in both the training and testing sets causing potential data leakage. Using Java is beneficial due to its widespread use in many different types of programs and its adoption in industry. Java also has a well defined commenting standard with JavaDocs that creates easily parsable documentation.

One other threat to validity is that we were unable to perform extensive hyper-parameter optimizations on our models due to hardware limitations. It could be the case that some of our models or baselines outlined in Table 2 could be heavily impacted by certain hyper-parameters (e.g., input sequence length, learning rate, and vocabulary size), giving different scores and rankings. This is a common issue with deep learning projects, and this affects nearly all similar types of experiments. We try to mitigate the impact of this issue by being consistent with our hyper-parameter values. We also take great care when reproducing work for our baselines, making sure the experimental set ups are reasonable and match them as closely as we can to their descriptions.

6 EXPERIMENT RESULTS

This section provides the experiment results for the research questions we ask in Section 5.1. To answer RQ_1 we use a combination of automated metrics and discuss its performance compared to other models in the context of these metrics. For RQ_2 we test a series of models with different hop sizes and compare them to our model as well as our baselines. To answer RQ_3 we provide a set of examples comparing our model using the graph AST and ConvGNN with a flattened AST model and show how the addition of the ConvGNN contributes to the overall summary.

6.1 RQ_1: Quantitative Evaluation

For our experimental results we tested three model configurations. In labeling our models we use code+gnn to represent the models that use an encoder for the source code tokens, a ConvGNN encoder for the AST, and then we use a +{layer_name} format to show the layer that was used on the output of the ConvGNN.

We found that model code+gnn+BiLSTM was the highest performing approach obtaining a BLEU-A score of 19.93 and ROUGE-LCS score of 56.08, as seen in Table 2. The code+gnn+BiLSTM model outperformed the nearest graph-based baseline by 4.6% BLEU-A and 0.06% ROUGE-LCS. The code+gnn+BiLSTM model also outperformed the flattened AST baseline by 5.7% BLEU-A and 12.72% ROUGE-LCS. We attribute this increase in performance to the use of the ConvGNN as an encoding for the AST. Adding the ConvGNN allows the model to learn better AST node representations than it can with only a sequence model. We go into more depth into how the ConvGNN may be boosting performance in Section 6.3. We attribute our performance improvement over other graph-based approaches to the use of the source code token sequence as a separate additional encoder. We found that using both the source code sequence and the AST allows the model to learn when to copy tokens directly from the source code, serving a purpose similar to a 'copy mechanism' as described by Gu *et al.* [18]. Copy mechanisms are used to copy words directly from the input text to the output, primarily used to improve performance with rare or unknown tokens. In this case, the model has learned to copy tokens directly from the source code. This works well for source code summarization because of the large overlap in source code and summary vocabulary (over 94%). In Section 6.3 example 1 we show how models that use

both source code and AST input utilize the source code attention like a copy mechanism.

We also see a noticeable effect on model performance based on the recurrent layer after the ConvGNNs. LeClair *et al.* achieved 18.69 BLEU-A using only a recurrent layer to encode the flattened AST sequence, and without a recurrent layer the code+gnn+dense model achieves a 19.46 BLEU-A. In an effort to see how different recurrent layers affect the models performance, we trained a two hop model using GRU and another model using a BiLSTM as shown in Table 2. We found that code+gnn+BiLSTM outperformed code+gnn+GRU by 0.05 BLEU-A and 0.3 ROUGE-LCS. The improved score of the BiLSTM layer is likely due to the increased complexity of the layer over the GRU. We find in many cases that the BiLSTM architecture outperforms other recurrent layers, but at a significantly increased computational cost. For this reason, and because the code+gnn+BiLSTM model only outperformed the code+gnn+GRU model by 0.05 BLEU-A (0.2%), we chose to conduct our other tests using the code+gnn+GRU architecture.

6.2 RQ$_2$: Hop size analysis

In Table 2 we compare the number of hops in the ConvGNN layers and how it affects the performance of the model. We found that for the AST two hops had the best overall performance. With having two hops, a node will get information from nodes up to two edges away. As outlined in Section 4, our model implementation creates a separate ConvGNN layer for each hop in series. One explanation for why two hops had the best performance is that, because we are generating summaries at the method level, the ASTs in the dataset are not very deep. Another possibility could be that the other nodes most important to a specific node are its neighbors, and dealing with smaller clusters of node data is sufficient for learning. Lastly, even though the number of hops directly influences how far and quickly information will propagate through the ConvGNN, every iteration the neighboring nodes are now an aggregate of their neighbors n hops away. In other words, after one iteration with two hops, a nodes neighbor is now an aggregate of nodes three hops away, so after enough iterations each node should be affected by every other node, with closer nodes having a larger effect.

While using two hops reported the best BLEU score for the code+gnn+GRU models, it only performed 1.5% better than using three hops and 2.8% better than using 10. Also notice that using five hops outperformed three and ten hops, this could be due to the random initialization or other minor factors. Because the difference in overall BLEU score is relatively small between hop sizes, we believe that the number of hops is less important than other hyper-parameters. It could be that the number of hops will be more important when summarizing larger selections of code where nodes are farther apart. For example, if the task were to summarize an entire program it may be beneficial to have more hops in your encoder to allow information to propagate farther.

6.3 RQ$_3$: Graph AST Contribution

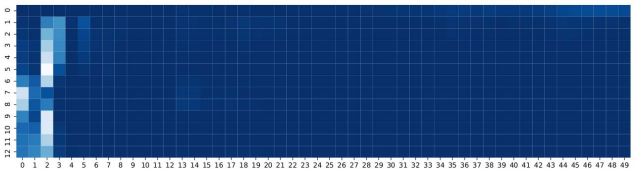
We provide three in-depth examples of the GNN's contribution to our model. In these examples we also compare directly with the ast-attendgru model proposed by LeClair *et al.* [29]. We chose to compare with ast-attendgru because it represents a collection of work using flattened ASTs to summarize source code as well as

Example 1, Method ID 20477616

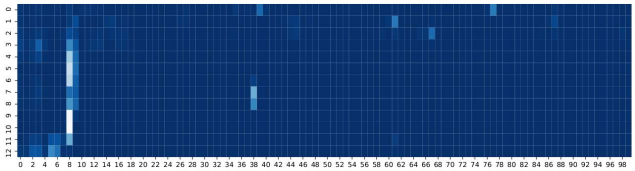
summaries	
reference	sends a guess to the server
code+gnn+GRU	sends a guess to the socket
ast-attendgru	attempts to initiate a <UNK> guess

source code

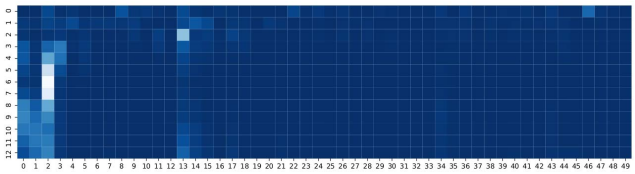
```
public void sendGuess(String guess) {
  if( isConnected() ) {
    gui.statusBarInfo("Querying...", false);
    try {
      os.write( (guess + "\\r\\n").getBytes() );
      os.flush();
    } catch (IOException e) {
gui.statusBarInfo(
    "Failed to send guess.IOException",true
    );
System.err.println(
    "IOException during send guess to server"
    );
    }
  }
}
```



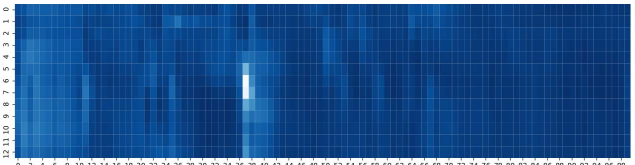
(b) *code+gnn+GRU*: Source attention



(c) *code+gnn+GRU*: AST attention



(d) *ast-attendgru*: Source attention



(e) *ast-attendgru*: AST attention

Example 1: Visualization of source code and AST attention for code+gnn+gru and ast-attendgru

having separate encoders for the source code sequence and AST. We feel that comparing against this model allows us to isolate the contribution that the ConvGNN is making to the generated summaries. It should be noted however that these two models process the AST differently, they both use SrcML [10] to generate ASTs, but ast-attendgru takes another step and additionally processes the AST using the SBT technique developed by Hu *et al.*. More details about how LeClair *et al.* process the AST can be found in Section 5.4.

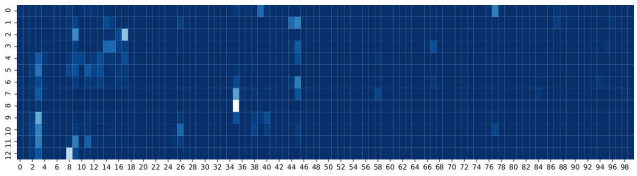
Our first example visualized in Example 1 shows an instance where the reference summary contains tokens that also appear in the source code. We use this example to showcase how the source code and AST attentions work together as a copy mechanism to generate summaries. The visualizations in Example 1 are a snapshot of the attention outputs for both the source code and AST when the models are predicting the first token in the sequence, which is 'sends' in the reference. We can see in Example 1 (a) that the code+gnn+GRU model and (c) the ast-attendgru model attend to the third token in the input source code sequence (column 2), which in this case is the token 'send'. Where these two models differ, however, is what their respective ASTs are attending to (seen in (b) and (d)). In the case of code+gnn+GRU (b), the model is attending to the token 'send' in column eight and the token 'status' in column thirty-eight. On the other hand, The ast-attendgru (d) model is attending to column thirty-seven, which is the token 'sexpr_stmt'.

One explanation for this is that code+gnn+GRU is able to combine structural and code elements better than models that don't utilize a ConvGNN. In the context of this example what this means is that the AST token 'send' in the code+gnn+GRU is a learned combination of the 'send' node and its neighbors, which in the AST are nodes 'name', 'status', 'bar', and 'info'. The ast-attendgru model only sees the AST as a sequence of tokens, so when it attends to the token 'sexpr_stmt', its neighboring tokens are 'sblock' and 'sexpr'. Another observation from Example 1 (d) is that, generally, the ast-attendgru model activates more on the AST sequence than it does on the source code sequence, while code+gnn+GRU activates similarly on both attentions and focuses more on specific tokens. This could be due to the ConvGNN learning more specific structure information from the AST.

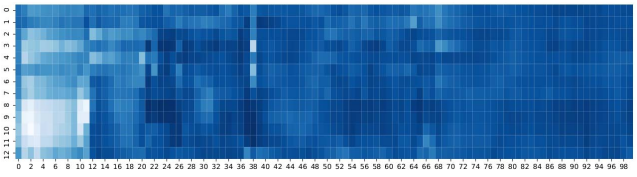
We also found that because the AST attention is more fine grained than the ast-attendgru attention, it learns whether to copy words directly from the source code better than the other model. In this case, because both the source code and AST attention focus on the same token, 'send', it determined that 'send' or a word very close to it (in this case 'sends') should be the predicted token. If the source code and AST attention differ then the model will often times predict tokens that are not in the source code or AST.

If we look at later tokens in the predicted sequence, we see that code+gnn+GRU predicts the correct tokens until the last one. For the final token the reference token is 'server' and code+gnn+GRU predicted 'socket'. What is also interesting here is that ast-attendgru predicted the token 'guess' which is in the reference summary and source code sequence. If we look at Example 2 we see the output of the AST attention mechanisms for both the code+gnn+GRU and ast-attendgru models during their prediction of the final token in the sequence. What this means is that code+gnn+GRU has the input sequence [sends, a, guess, to, the] and predicts 'socket' and ast-attendgru has the sequence [attempts, to, initiate, a, <UNK>] and predicts 'guess'. We do not include the source code visualization here because they were very similar to the visualizations in Example 1 (a) and (c). So, in the source code attention both models attend to the tokens 'send' and 'guess', but as we can see in the AST visualization, code+gnn+GRU is attending to the token in column forty-six - 'querying'; and ast-attendgru is attending to a large, non-specific area in the structure of the AST. The code+gnn+GRU model has learned that the combination of the tokens 'send', 'guess', and the AST token 'querying' lead to the prediction of the token 'socket'. While this prediction was incorrect, the token 'socket' is closely related to the term 'server' in this context. Notice that the token 'querying' is also in the source code, but neither model attends to it. As stated above, the source code attention is acting more as a copy mechanism and is attending to tokens that it believes should be the next predicted token, then it relies on the AST attention to add additional information for the final prediction.

Example 3 is a case where the code+gnn+GRU model correctly predicts the sixth token in the sequence, 'first', but ast-attendgru predicts the token 'specified'. For this prediction both models have the same predicted token sequence input to the decoder. If we look at Example 3, we can see the the attention visualizations for our models for their prediction of the sixth token in the sequence. In Example 3 (a) in the sixth row, the code+gnn+GRU model is attending to the fifth column, which is the token 'o'. In this piece of code 'o' is the identifier for the input parameter for the method. The ast-attendgru model source attention (Example 3 (c)) is attending to columns seventeen and thirty-four which are both the token 'game'. Looking at the code+gnn+GRU AST attention (Example 3 (b)), we see that it is attending to column sixty-three, which is also the token 'game'. So, in this example the ast-attengru model's source attention is attending to 'game' and the code+gnn+GRU model's attention is also attending to the token 'game' in the AST. This is important because it shows both models have learned that this token is important to the prediction, but in different contexts. Looking at the visualizations, we see again that the code+gnn+GRU model is able to focus on specific, important tokens in both the source code and the AST, while the ast-attendgru model attends to larger portions of the AST.



(a) *code+gnn+GRU*: AST attention



(b) *ast-attendgru*: AST attention

Example 2: Visualization of AST attention mechanisms for code+gnn+gru and ast-attendgru when predicting the final token in the sequence.

7 DISCUSSION AND FUTURE WORK

The major take-aways from the work outlined in this paper are:

- Using the AST as a graph with ConvGNN layers outperforms a flattened version of the AST
- Including the source code sequence as a separate encoder allows the model to learn to use the source code and AST as a copy mechanism.
- The improved node embeddings from the ConvGNN allow the model to learn better token representations where representation of tokens in the AST are a combination of structure elements.

The three examples that we show are situations where the addition of the ConvGNN allowed the model to learn better node representations than using a flattened sequence for the AST. When both the source code attention and the AST attention align on a specific token, the model treats this like a copy mechanism, directly copying the input source token to the output. When the source and AST attention do not agree, we see the model relying more on the AST to predict the next token in the sequence. When we compare this to a model with a flattened AST input, we see a large difference in how the AST is being attended to, generally the flattened AST model looks at larger structure areas instead of specific tokens.

As an avenue for future work, models such as these have been shown to improve performance when ensembled. LeClair *et al.* showed that a model without AST information outperformed a model using AST information on specific types of summaries [29]. This could lead to interesting results, potentially showing that bringing in different features from the source code allows the models to learn to generate better summaries for specific types of methods.

8 CONCLUSION

In this work we have presented a new neural model architecture that utilizes a sequence of source code tokens along with ConvGNNs to encode the AST of a Java method and generate natural language summaries. We provide background and insights into why using a graph based neural network to encode the AST improves performance, along with providing a comparison of our results against relevant baselines. We conclude that the combination of source code tokens along with the AST and ConvGNNs allows the model to better learn when to directly copy tokens from the source code, as well as create better representations of tokens in the AST. We show that that the use of the ConvGNN to encode the AST improves aggregate BLEU scores (BLEU-A) by over 4.6% over other graph-based approaches and 5.7% improvement over flattened AST approaches. We also provide an in dept analysis of how the ConvGNN layers attribute to this increase in performance, and speculate on how these insights can be used for future work.

9 REPODUCIBILITY

All of our models, source code, and data used in this work can be found in our online repository at https://go.aws/2tPXV2R.

10 ACKNOWLEDGMENTS

This work is supported in part by NSF CCF-1452959 and CCF-1717607. Any opinions, findings, and conclusions expressed herein are the authors and do not necessarily reflect those of the sponsors.

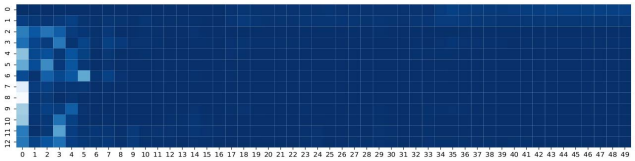
Example 3, Method ID 25584536

summaries

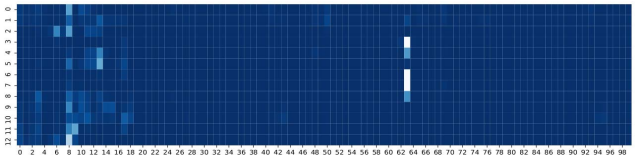
reference	returns the index of the first occurrence of the specified element
code+gnn+GRU	returns the index of the first occurrence of the specified element
ast-attendgru	returns the index of the specified object in the list

source code

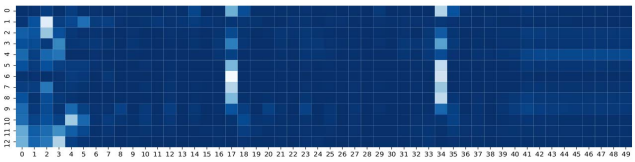
```
public int indexOf(Object o) {
  if (o == null) {
    for (int i = 0; i < size; i++) {
      if (gameObjects[i] == null) {
        return i;
      }
    }
    } else {
        for (int i = 0; i < size; i++) {
            if (o.equals(gameObjects[i])) {
                return i;
            }
        }
    }
    return -1;
}
```



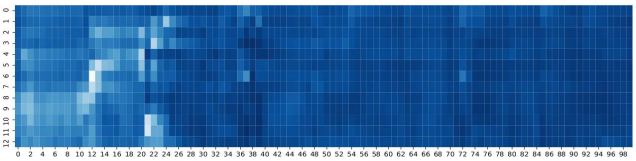
(b) *code+gnn+GRU*: Source attention



(c) *code+gnn+GRU*: AST attention



(d) *ast-attendgru*: Source attention



(e) *ast-attendgru*: AST attention

Example 3: Visualization of source code and AST attention for code+gnn+GRU and ast-attendgru

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.

[2] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to represent programs with graphs. *International Conference on Learning Representations* (2018).

[3] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019. code2seq: Generating sequences from structured representations of code. *International Conference on Learning Representations* (2019).

[4] Leila Arras, Franziska Horn, Grégoire Montavon, Klaus-Robert Müller, and Wojciech Samek. 2017. "What is relevant in a text document?": An interpretable machine learning approach. *PLOS ONE* 12, 8 (Aug 2017), e0181142. https://doi.org/10.1371/journal.pone.0181142

[5] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473* (2014).

[6] David Binkley. 2007. Source code analysis: A road map. In *2007 Future of Software Engineering*. IEEE Computer Society, 104–119.

[7] Huadong Chen, Shujian Huang, David Chiang, and Jiajun Chen. 2017. Improved Neural Machine Translation with a Syntax-Aware Encoder and Decoder. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Vancouver, Canada, 1936–1945. https://doi.org/10.18653/v1/P17-1177

[8] Yu Chen, Lingfei Wu, and Mohammed J Zaki. 2020. Reinforcement learning based graph-to-sequence model for natural question generation. *International Conference on Learning Representations* (2020).

[9] François Chollet et al. 2015. Keras. https://github.com/fchollet/keras.

[10] Michael L Collard, Michael J Decker, and Jonathan I Maletic. 2011. Lightweight transformation and fact extraction with the srcML toolkit. In *Source Code Analysis and Manipulation (SCAM), 2011 11th IEEE International Working Conference on*. IEEE, 173–184.

[11] Bas Cornelissen, Andy Zaidman, Arie Van Deursen, Leon Moonen, and Rainer Koschke. 2009. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering* 35, 5 (2009), 684–702.

[12] Sergio Cozzetti B. de Souza, Nicolas Anquetil, and Káthia M. de Oliveira. 2005. A study of the documentation essential to software maintenance. In *Proceedings of the 23rd annual international conference on Design of communication: documenting & designing for pervasive information (SIGDOC '05)*. ACM, New York, NY, USA, 68–75. https://doi.org/10.1145/1085313.1085331

[13] Derek Doran, Sarah Schulz, and Tarek R. Besold. 2017. What Does Explainable AI Really Mean? A New Conceptualization of Perspectives. *CoRR* abs/1710.00794 (2017). arXiv:1710.00794 http://arxiv.org/abs/1710.00794

[14] Finale Doshi-Velez and Been Kim. 2017. Towards A Rigorous Science of Interpretable Machine Learning. *arXiv e-prints*, Article arXiv:1702.08608 (Feb 2017), arXiv:1702.08608 pages. arXiv:stat.ML/1702.08608

[15] Brian P Eddy, Jeffrey A Robinson, Nicholas A Kraft, and Jeffrey C Carver. 2013. Evaluating source code summarization techniques: Replication and expansion. In *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*. IEEE, 13–22.

[16] Patrick Fernandes, Miltiadis Allamanis, and Marc Brockschmidt. 2018. Structured Neural Summarization. *CoRR* abs/1811.01824 (2018). arXiv:1811.01824 http://arxiv.org/abs/1811.01824

[17] Andrew Forward and Timothy C. Lethbridge. 2002. The relevance of software documentation, tools and technologies: a survey. In *Proceedings of the 2002 ACM symposium on Document engineering (DocEng '02)*. ACM, New York, NY, USA, 26–33. https://doi.org/10.1145/585058.585065

[18] Jiatao Gu, Zhengdong Lu, Hang Li, and Victor O.K. Li. 2016. Incorporating Copying Mechanism in Sequence-to-Sequence Learning. *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (2016). https://doi.org/10.18653/v1/p16-1154

[19] Sonia Haiduc, Jairo Aponte, Laura Moreno, and Andrian Marcus. 2010. On the use of automated text summarization techniques for summarizing source code. In *Reverse Engineering (WCRE), 2010 17th Working Conference on*. IEEE, 35–44.

[20] S. Haiduc and A. Marcus. 2008. On the Use of Domain Terms in Source Code. In *16th IEEE International Conference on Program Comprehension (ICPC'08)*. Amsterdam, The Netherlands, 113–122.

[21] Vincent J Hellendoorn and Premkumar Devanbu. 2017. Are deep neural networks the best choice for modeling source code?. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 763–773.

[22] M. J. Howard, S. Gupta, L. Pollock, and K. Vijay-Shanker. 2013. Automatically mining software-based, semantically-similar words from comment-code mappings. In *2013 10th Working Conference on Mining Software Repositories (MSR)*. 377–386. https://doi.org/10.1109/MSR.2013.6624052

[23] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *Proceedings of the 26th International Conference on Program Comprehension*. ACM, 200–210.

[24] Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. 2018. Summarizing Source Code with Transferred API Knowledge.. In *IJCAI*. 2269–2275.

[25] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. 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)*, Vol. 1. 2073–2083.

[26] Mira Kajko-Mattsson. 2005. A Survey of Documentation Practice within Corrective Maintenance. *Empirical Softw. Engg.* 10, 1 (Jan. 2005), 31–55. https://doi.org/10.1023/B:LIDA.0000048322.42751.ca

[27] Douglas Kramer. 1999. API documentation from source code comments: a case study of Javadoc. In *Proceedings of the 17th annual international conference on Computer documentation*. ACM, 147–153.

[28] A. LeClair, Z. Eberhart, and C. McMillan. 2018. Adapting Neural Text Classification for Improved Software Categorization. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 461–472. https://doi.org/10.1109/ICSME.2018.00056

[29] Alexander LeClair, Siyuan Jiang, and Collin McMillan. 2019. A neural model for generating natural language summaries of program subroutines. In *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, 795–806.

[30] Alexander LeClair and Collin McMillan. 2019. Recommendations for Datasets for Source Code Summarization. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. 3931–3937.

[31] Yuding Liang and Kenny Q. Zhu. 2018. Automatic Generation of Text Descriptive Comments for Code Blocks. *CoRR* abs/1808.06880 (2018). arXiv:1808.06880 http://arxiv.org/abs/1808.06880

[32] Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. *Text Summarization Branches Out* (2004).

[33] C. Lopes, S. Bajracharya, J. Ossher, and P. Baldi. 2010. UCI Source Code Data Sets. http://www.ics.uci.edu/$\sim$lopes/datasets/

[34] Pablo Loyola, Edison Marrese-Taylor, and Yutaka Matsuo. 2017. A Neural Architecture for Generating Natural Language Descriptions from Source Code Changes. In *ACL*.

[35] Yangyang Lu, Zelong Zhao, Ge Li, and Zhi Jin. 2019. Learning to Generate Comments for API-Based Code Snippets. In *Software Engineering and Methodology for Emerging Domains*, Zheng Li, He Jiang, Ge Li, Minghui Zhou, and Ming Li (Eds.). Springer Singapore, Singapore, 3–14.

[36] Paul W McBurney, Cheng Liu, and Collin McMillan. 2016. Automated feature discovery via sentence selection and source code summarization. *Journal of Software: Evolution and Process* 28, 2 (2016), 120–145.

[37] Paul W McBurney and Collin McMillan. 2016. Automatic source code summarization of context for java methods. *IEEE Transactions on Software Engineering* 42, 2 (2016), 103–119.

[38] Tim Miller. 2019. Explanation in artificial intelligence: Insights from the social sciences. *Artificial Intelligence* 267 (2019), 1 – 38. https://doi.org/10.1016/j.artint.2018.07.007

[39] Laura Moreno and Jairo Aponte. 2012. On the analysis of human and automatic summaries of source code. *CLEI Electronic Journal* 15, 2 (2012), 2–2.

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

[41] Najam Nazar, Yan Hu, and He Jiang. 2016. Summarizing software artifacts: A literature review. *Journal of Computer Science and Technology* 31, 5 (2016), 883–909.

[42] Karl J Ottenstein and Linda M Ottenstein. 1984. The program dependence graph in a software development environment. *ACM SIGSOFT Software Engineering Notes* 9, 3 (1984), 177–184.

[43] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*. Association for Computational Linguistics, 311–318.

[44] Paige Rodeghero, Cheng Liu, Paul W McBurney, and Collin McMillan. 2015. An eye-tracking study of java programmers and application to source code summarization. *IEEE Transactions on Software Engineering* 41, 11 (2015), 1038–1054.

[45] Tobias Roehm, Rebecca Tiarks, Rainer Koschke, and Walid Maalej. 2012. How do professional developers comprehend software?. In *Proceedings of the 2012 International Conference on Software Engineering (ICSE 2012)*. IEEE Press, Piscataway, NJ, USA, 255–265. http://dl.acm.org/citation.cfm?id=2337223.2337254

[46] Ribana Roscher, Bastian Bohn, Marco F. Duarte, and Jochen Garcke. 2019. Explainable Machine Learning for Scientific Insights and Discoveries. arXiv:cs.LG/1905.08883

[47] Wojciech Samek, Thomas Wiegand, and Klaus-Robert Müller. 2017. Explainable artificial intelligence: Understanding, visualizing and interpreting deep learning models. *arXiv preprint arXiv:1708.08296* (2017).

[48] Lin Shi, Hao Zhong, Tao Xie, and Mingshu Li. 2011. An empirical study on evolution of API documentation. In *Proceedings of the 14th international conference on Fundamental approaches to software engineering: part of the joint European conferences on theory and practice of software (FASE'11/ETAPS'11)*. Springer-Verlag, Berlin, Heidelberg, 416–431. http://dl.acm.org/citation.cfm?id=1987434.1987473

[49] Xiaotao Song, Hailong Sun, Xu Wang, and Jiafei Yan. 2019. A Survey of Automatic Generation of Source Code Comments: Algorithms and Techniques. *IEEE Access* (2019).

[50] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K Vijay-Shanker. 2010. Towards automatically generating summary comments for java methods. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*. ACM, 43–52.

[51] Giriprasad Sridhara, Lori Pollock, and K Vijay-Shanker. 2011. Automatically detecting and describing high level actions within methods. In *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 101–110.

[52] Ilya Sutskever, James Martens, and Geoffrey E Hinton. 2011. Generating text with recurrent neural networks. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*. 1017–1024.

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

[54] Anneliese Von Mayrhauser and A Marie Vans. 1995. Program comprehension during software maintenance and evolution. *Computer* 8 (1995), 44–55.

[55] Laura von Rueden, Sebastian Mayer, Jochen Garcke, Christian Bauckhage, and Jannis Schuecker. 2019. Informed Machine Learning - Towards a Taxonomy of Explicit Integration of Knowledge into Machine Learning. arXiv:stat.ML/1903.12394

[56] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S. Yu. 2018. Improving Automatic Source Code Summarization via Deep Reinforcement Learning. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018)*. Association for Computing Machinery, New York, NY, USA, 397–407. https://doi.org/10.1145/3238147.3238206

[57] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. 2019. A Comprehensive Survey on Graph Neural Networks. *CoRR* abs/1901.00596 (2019). arXiv:1901.00596 http://arxiv.org/abs/1901.00596

[58] Kun Xu, Lingfei Wu, Zhiguo Wang, Yansong Feng, Michael Witbrock, and Vadim Sheinin. 2018. Graph2seq: Graph to sequence learning with attention-based neural networks. *arXiv preprint arXiv:1804.00823* (2018).

[59] Kun Xu, Lingfei Wu, Zhiguo Wang, Mo Yu, Liwei Chen, and Vadim Sheinin. 2018. Exploiting rich syntactic information for semantic parsing with graph-to-sequence model. *Conference on Empirical Methods in Natural Language Processing* (2018).