CAST: Enhancing Code Summarization with Hierarchical Splitting and Reconstruction of Abstract Syntax Trees

Ensheng Shi[a,†] **Yanlin Wang**[b,†,§] **Lun Du**[b] **Hongyu Zhang**[c]
Shi Han[b] **Dongmei Zhang**[b] **Hongbin Sun**[a,§]
[a]Xi'an Jiaotong University [b]Microsoft Research [c]The University of Newcastle
s1530129650@stu.xjtu.edu.cn, hsun@mail.xjtu.edu.cn
{yanlwang, lun.du, shihan, dongmeiz}@microsoft.com
hongyu.zhang@newcastle.edu.au

Abstract

Code summarization aims to generate concise natural language descriptions of source code, which can help improve program comprehension and maintenance. Recent studies show that syntactic and structural information extracted from abstract syntax trees (ASTs) is conducive to summary generation. However, existing approaches fail to fully capture the rich information in ASTs because of the large size/depth of ASTs. In this paper, we propose a novel model CAST that hierarchically splits and reconstructs ASTs. First, we hierarchically split a large AST into a set of subtrees and utilize a recursive neural network to encode the subtrees. Then, we aggregate the embeddings of subtrees by reconstructing the split ASTs to get the representation of the complete AST. Finally, AST representation, together with source code embedding obtained by a vanilla code token encoder, is used for code summarization. Extensive experiments, including the ablation study and the human evaluation, on benchmarks have demonstrated the power of CAST. To facilitate reproducibility, our code and data are available at https://github.com/DeepSoftwareAnalytics/CAST.

1 Introduction

Code summaries are concise natural language descriptions of source code and they are important for program comprehension and software maintenance. However, it remains a labor-intensive and time-consuming task for developers to document code with good summaries manually.

Over the years, many code summarization methods have been proposed to automatically summarize program subroutines. Traditional approaches such as rule-based and information retrieval-based approaches regard source code as plain text (Haiduc et al., 2010a,b) without considering the complex grammar rules and syntactic structures exhibited in source code. Recently, abstract syntax trees (ASTs), which carry the syntax and structure information of code, are widely used to enhance code summarization techniques. For example, Hu et al. (2018a) propose the structure-based traversal (SBT) method to flatten ASTs and use LSTM to encode the SBT sequences into vectors. Hu et al. (2019) and LeClair et al. (2019) extend this idea by separating the code and AST into two input channels, demonstrating the effectiveness of leveraging AST information. Alon et al. (2019a,b) extract paths from an AST and represent a given code snippet as a set of sampled paths. Other works (Wan et al., 2018; Zhang et al., 2019; Mou et al., 2016) use tree-based models such as Tree-LSTM, Recursive Neural Network (RvNN), and Tree-based CNN to model ASTs and improve code summarization.

We have identified some limitations of the existing AST-based approaches, which lead to a slow training process and/or the loss of AST structural information. We now use an example shown in Fig. 1 to illustrate the limitations:

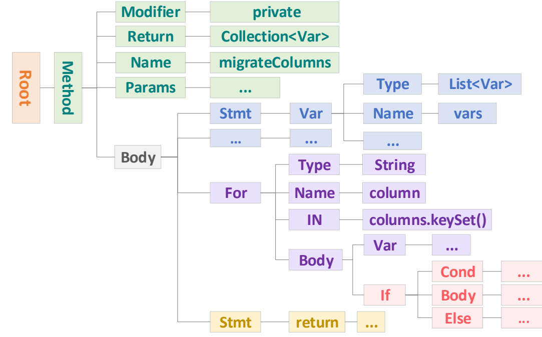
- Models that directly encode ASTs with tree-based neural networks suffer from long training time. HybridDrl (Wan et al., 2018) spends 21 hours each epoch on Funcom (LeClair et al., 2019). This is because ASTs are usually large and deep due to the complexity of programs, especially when there are nested program structures. For example, our statistics show that the maximal node number/depth of ASTs of methods in TL-CodeSum (Hu et al., 2018b) and Funcom are 6,165/74 and 550/32, respectively. Moreover, HybridDrl transforms ASTs into binary trees, leading to deeper trees and more loss of structural information. As shown in Fig. 1(e), the main semantics of the code in Fig. 1(a) are not fully captured by

[†]The first two authors contribute equally.
[§]Yanlin Wang and Hongbin Sun are the corresponding authors.

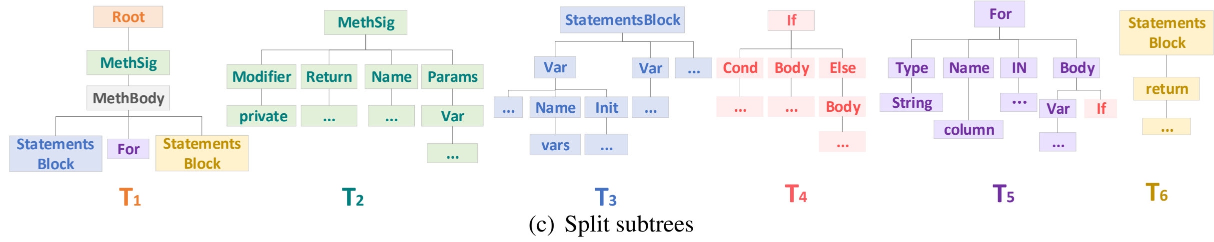
Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, pages 4053–4062
November 7–11, 2021. ©2021 Association for Computational Linguistics

```
1.  private Collection<Var> migrateColumns(SQLTable currentTable){
2.    List<Var> vars=new ArrayList<>();
3.    String tableType=currentTable.getEntityType();
4.    Map<String,ResourceType.DataType> columns=currentTable.getColumns();
5.    Map<String,String> foreignColumns=currentTable.getForeignKeyColumns();
6.    for (String column : columns.keySet()) {
7.      ResourceType.DataType columnType=columns.get(column);
8.      if (foreignColumns.containsKey(column)) {
9.        vars.addAll(migrateAsRelation(tableType,
          column,foreignColumns.get(column)));}
10.     else {
11.       vars.addAll(migrateAsResource(tableType,columnType,column));
12.     }
13.   }
14.   return vars;
15. }
```

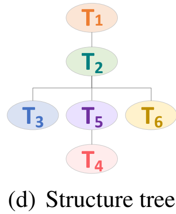
(a) Source code snippet



(b) Full AST



(c) Split subtrees



(d) Structure tree

	Reference	loop through each of the columns in the given table, migrating each as a resource or relation
AST-based models	**HybridDrl**	finds a set of rows in a table
	Astattgru	maps a list to the table
	Hdeepcom	loop total columns of of column the the , , , column having . column
	Code2seq	returns the list of the given
	Astnn	returns the list of the given list of the given list .
Others	**Attgru**	executes the function , including the java attributes from all columns rules found
	NCS	migrate the columns of a table in the given resource query
	CodeNN	returns a foreign - of columns the
Ours	**CAST**	loop through each of the columns in the table, migrating each as a resource or relation

(e) Summaries generated by various approaches.The first raw is the summaries wrote by human. Two sub-sentences in the reference summary are marked in different color.

Figure 1: A running example of code, AST, and generated summaries.

HybridDrl.

- Linearization methods that flatten ASTs into sequences (Hu et al., 2018a; Alon et al., 2019a,b), by nature, lose the hierarchical information of ASTs. ASTNN (Zhang et al., 2019) splits an AST into small statement trees to reduce the difficulty of large tree training. However, each subtree contains one statement only and subtrees are later linearized and fed into an RNN, also leading to the loss of hierarchical information. From Fig. 1(e), we can see that linearization methods Code2seq (Alon et al., 2019a), Astattgru (LeClair et al., 2019) and ASTNN (Zhang et al., 2019) fail to capture the main semantics, and HDeepcom (Hu et al., 2019) captures only partial semantics.

To overcome the above limitations, we propose a novel model CAST (Code summarization with hierarchical splitting and reconstruction of Abstract Syntax Trees). The key idea of our approach is to split an AST (Fig. 1(b)) into a set of subtrees (Fig. 1(c)) at a proper granularity and learn the representation of the complete AST by aggregating its subtrees' representation learned using tree-based neural models. First, we split a full AST in a hierarchical way using a set of carefully designed rules. Second, we use a tree-based neural model RvNN to learn each subtree's representation. Third, we reconstruct the split ASTs and combine all subtrees' representation by another RvNN to capture the full tree's structural and semantic information. Finally, the representation of the complete tree, together with source code embedding obtained by a vanilla code token encoder, is fed to a Transformer decoder to generate descriptive summaries. Take Fig. 1(a) for example again: there are two sub-sentences in

the reference summary. The `For` block (Lines 6, 7 and 13 in Fig. 1(a)) corresponds to the first sub-sentence *"loop through each of the columns in the given table"*, and the `If` block (Line 8-12) corresponds to the second sub-sentence *"migrating each as a resource or relation"*. The semantics of each block can be easily captured when the large and complex AST is split into five subtrees as shown in Fig. 1(c). After splitting, T_5 corresponds to first sub-sentence and T_4 corresponds to the second sub-sentence. When we reconstruct the split ASTs according to Fig. 1(d), it is easier for our approach to generate the summary with more comprehensive semantics.

Our method CAST has two-sided advantages: (1) Tree splitting reduces AST to a proper size to allow effective and affordable training of tree-based neural models. (2) Different from previous work, we not only split trees but also reconstruct the complete AST using split ASTs. This way, high-level hierarchical information of ASTs can be retained.

We conduct experiments on TL-CodeSum (Hu et al., 2018b) and Funcom (LeClair et al., 2019) datasets, and compare CAST with the state-of-the-art methods. The results show that our model outperforms the previous methods in four widely-used metrics Bleu-4, Rouge-L, Meteor and Cider, and significantly decreases the training time compared to HybridDrl. We summarize the main contributions of this paper as follows:

- We propose a novel AST representation learning method based on hierarchical tree splitting and reconstruction. The splitting rule specification and the tool implementation are provided for other researchers to use in AST relevant tasks.
- We design a new code summarization approach CAST, which incorporates the proposed AST representations and code token embeddings for generating code summaries.
- We perform extensive experiments, including the ablation study and the human evaluation, on CAST and state-of-the-art methods. The results demonstrate the power of CAST.

2 Related Work

2.1 Source Code Representation

Previous work suggests various representations of source code for follow-up analysis. Allamanis et al. (2015) and Iyer et al. (2016) consider source code as plain text and use traditional token-based methods to capture lexical information. Gu et al. (2016) use the Seq2Seq model to learn intermediate vector representations of queries in natural language to predict relevant API sequences. Mou et al. (2016) propose a tree-based convolutional neural network to learn program representations. Alon et al. (2019b; 2019a) represent a code snippet as a set of compositional paths in the abstract syntax tree. Zhang et al. (2019) propose an AST-based Neural Network (ASTNN) that splits each large AST into a sequence of small statement trees and encodes them to vectors by capturing the lexical and syntactical knowledge. Shin et al. (2019) represent idioms as AST segments using probabilistic tree substitution grammars for two tasks: idiom mining and code generation. (LeClair et al., 2020; Wang and Li, 2021) utilize GNNs to model ASTs. There are also works that utilize ensemble model (Du et al., 2021) or pre-trained models (Feng et al., 2020a; Guo et al., 2021; Bui et al., 2021) to model source code.

2.2 Source Code Summarization

Apart from the works mentioned above, researchers have proposed many approaches to source code summarization over the years. For example, Allamanis et al. (2015) create the neural logbilinear context model for suggesting method and class names by embedding them in a high dimensional continuous space. Allamanis et al. (2016) also suggest a convolutional model for the summary generation that uses attention over a sliding window of tokens. They summarize code snippets into extreme, descriptive function name-like summaries.

Neural Machine Translation based models are also widely used for code summarization (Iyer et al., 2016; Haije, 2016; Hu et al., 2018a,b; Wan et al., 2018; Hu et al., 2019; LeClair et al., 2019; Ahmad et al., 2020; Yu et al., 2020). **CodeNN** (Iyer et al., 2016) is the first neural approach for code summarization. It is a classical encoder-decoder framework that encodes code to context vectors with an attention mechanism and then generates summaries in the decoder. **NCS** (Ahmad et al., 2020) models code using Transformer to capture the long-range dependencies. **HybridDrl** (Wan et al., 2018) uses hybrid code representations (with ASTs) and deep reinforcement learning. It encodes the sequential and structural content of code

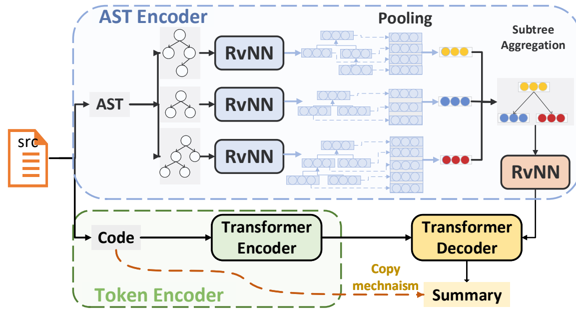


Figure 2: CAST model structure.

by LSTMs and tree-based LSTMs and uses a hybrid attention layer to get an integrated representation. **HDeepcom** (Hu et al., 2019), **Astattgru**, and **Attgru** (LeClair et al., 2019) are essentially encoder-decoder network using RNNs with attention. Astattgru and HDeepcom utilize a multi-encoder neural model that encodes both code and AST. **Code2seq** (Alon et al., 2019a) represents a code snippet as a set of AST paths and uses attention to select the relevant paths while decoding. When using neural networks to represent large and deep ASTs, the above work will encounter problems such as gradient vanishing and slow training. CAST can alleviate these problems by introducing a more efficient AST representation to generate better code summarie.

3 CAST: Code Summarization with AST Splitting and Reconstruction

This section presents the details of our model. The architecture of CAST (Fig. 2) follows the general Seq2Seq framework and includes three major components: an AST encoder, a code token encoder, and a summary decoder. Given an input method, the AST encoder captures the semantic and structural information of its AST. The code token encoder encodes the lexical information of the method. The decoder integrates the multi-channel representations from the two encoders and incorporates a copy mechanism (See et al., 2017) to generate the code summary.

3.1 AST Encoder

3.1.1 AST Splitting and Reconstruction

Given a code fragment, we build its AST and visit it by preorder traversal. Each time a composite structure (i.e. `If`, `While`, etc.) is encountered, a placeholder node is inserted. The subtree rooted at this statement is split out to form the next level

```
Input AST =  Root [Method [
                    Modifier [t*],
                    Return [t],
                    Name [t],
                    (Params [(Var [Type[t], Name [t]])+] ,)?
                    Body [StmtT*]]
StmtT     =  SimpleT | BlockT
OvT       =  Root [MethSig [MethBody [(StmtsT_r | BlockT_r)* ]]]
SigT      =  MethSig [Modifier [t*],
                     Return [t],
                     Name [t],
                     (Params [(Var [Type[t], Name [t]])+])?]
StmtsT    =  StatementsBlock [SimpleT+]
BlockT    =  IfT | ForT | WhileT | DoWhileT | ... | TryT
IfT       =  If [Cond [...], Body [...] (, Else [...])?]
SimpleT   =  VariableDeclaration | Expression | ReturnStatement
StmtsT_r  =  rootOf(StmtsT)
BlockT_r  =  rootOf(BlockT)
t         =  identifier
Subtrees  =  {OvT} U {SigT} U {StmtsT*} U {BlockT*}
```

Figure 3: AST splitting rule specification. Input is a full `AST` and output is the `Subtrees` set. We use brackets to denote subtree relation instead of the visualized trees to make the rules compact. ***Bold and italic*** stands for components that can be further reduced. Underline stands for nodes created in the splitting process.

tree, whose semantics will be finally stuffed back to the placeholder. In this way, a large AST is decomposed into a set of small subtrees with the composite structures retained.

Before presenting the formal tree splitting rules, we provide an illustrative example in Fig. 1. The full AST[1] (Fig. 1(b)) of the given code snippet (Fig. 1(a)) is split to six subtrees T_1 to T_6 in Fig. 1(c). T_1 is the overview tree with non-terminal nodes `Root`, `MethSig`, `MethBody`, and three terminal nodes `StatementsBlock` (blue), `For`, and `StatementsBlock` (yellow) corresponding to the three main segments with `Line2-5`, `Line6-13`, and `Line14` in Fig. 1(a), respectively. The `StatementsBlock` (blue) node corresponds to T_3 which contains 4 initialization statements. The `For` node corresponds to T_5 and the `StatementsBlock` (yellow) node corresponds to T_6 which consists of a return statement. Note that each subtree reveals one-level abstraction, meaning that nested structures are abstracted out. Therefore, the `If` statement nested in the `For` loop is split out to the subtree T_4, leaving a placeholder `If` node in T_5.

We give the formal definition[2] of subtrees in

[1] The full AST is omitted due to space limit, it can be found in Appendix.

[2] We only present the top-down skeleton and partial rules

Fig. 3. The goal is to split a given AST to the subtree set $Subtrees$. In general, all subtrees are generated by mapping reduction rules on the input AST (similar to mapping language grammar rules to a token sequence) and the output $Subtrees$ collects four kinds of subtrees: OvT, $SigT$, $StmtsT$, and $BlockT$. OvT is the method overview tree, providing the big picture of the method and $SigT$ gives the method signature information. To avoid too many scattered simple statements, we combine sequential statements to form a statements' block $StmtsT$. We drill down each terminal node in OvT to a subtree T being a $StmtT$ or $BlockT$, providing detailed semantics of nodes in the overview tree. In the same way, subtrees corresponding to nested structures (such as `For` or `If`) will be split out to form new subtrees. We split out nested blocks one level at a time until there is no nested block. Finally, we obtain a set of these block-level subtrees. Also, a *structure tree* (e.g., Fig. 1(d)) that represents the ancestor-descendant relationships between the subtrees is maintained.

3.1.2 AST Encoding

We design a two-phase AST encoder module according to the characteristics of subtrees. In the first phase, a tree-based Recursive Neural Network (RvNN) followed by a max-pooling layer is applied to encode each subtree. In the second phase, we use another RvNN with different parameters to model the hierarchical relationship among the subtrees.

A subtree T_t is defined as (V_t, E_t) where V_t is the node set and E_t is the edge set. The forward propagation of RvNN to encode the subtree T_t is formulated as:

$$\mathbf{h_i^{(t)}} = \tanh\left(\mathbf{W^C}\mathbf{c_i^{(t)}} + \frac{1}{|Ch^t(v_i)|}\sum_{v_j \in Ch^t(v_i)} \mathbf{W^A}\mathbf{h_j^{(t)}}\right), \quad (1)$$

where $\mathbf{W^C}$ and $\mathbf{W^A}$ are learnable weight matrices, $\mathbf{h_i^{(t)}}$, $\mathbf{c_i^{(t)}}$, $Ch(v_i)$ are the hidden state, token embedding, and child set of the node v_i, respectively. Particularly, $\mathbf{h_i^{(t)}}$ equals to $\mathbf{W^C}\mathbf{c_i^{(t)}}$ for the leaf node v_i.

Intuitively, this computation is the procedure where each node in the AST aggregates information from its children nodes. After this bottom-up aggregation, each node has its corresponding hidden states. Finally, the hidden states of all nodes

due to space limitation. The full set of rules, the splitting algorithm, and tool implementation are provided in Appendix.

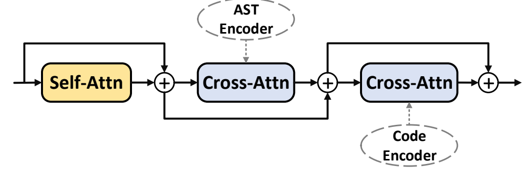


Figure 4: The serial strategy for integrating two encoding sources in the decoder.

are aggregated to a vector $\mathbf{s_t}$ through dimension-wise max-pooling operation, which will be used as the embedding for the whole subtree T_t:

$$\mathbf{s_t} = \text{maxpooling}\left(\cup\mathbf{h_i^{(t)}}\right), \forall v_i \in V_t. \quad (2)$$

After obtaining the embeddings of all subtrees, we further encode the descendant relationships among the subtrees. These relationships are represented in the *structure tree* (e.g., Fig. 1(d)) T, thus we apply another RvNN model on T:

$$\mathbf{h_t^{(a)}} = \tanh(\mathbf{W^S}\mathbf{s_t} + \frac{1}{|Ch(v_t)|}\sum_{v_k \in Ch(v_t)} \mathbf{W^B}\mathbf{h_k^{(a)}}). \quad (3)$$

There are two main advantages of our AST encoder design. First, it enhances the ability to capture semantic information in multiple subtrees of a program by the first layer RvNN, because the tree splitting technique leads to subtrees that contain semantic information from different modules. In addition, to obtain more important features of the node vectors, we sample all nodes through max pooling. The second layer RvNN can further aggregate information of subtrees according to their relative positions in the hierarchy. Second, tree sizes are decreased significantly after splitting, thus the gradient vanishing and explosion problems are alleviated. Also, after tree splitting, the depth of each subtree is well controlled, leading to more stable model training.

3.2 Code Token Encoder

The code snippets are the raw data source to provide lexical information for the code summarization task. Following (Ahmad et al., 2020), we adopt the code token encoder using Transformer that is composed of a multi-head self-attention module and a relative position embedding module. In each attention head, the sequence of code token embeddings $c = (c_1, ..., c_n)$ are transformed into output vector $o = (o_1, ..., o_n)$

$$o_i = \sum_{j=1}^{n} \alpha_{ij}\left(\mathbf{W^V}c_j + a_{ij}^V\right), e_{ij} = \frac{(\mathbf{W^Q}c_i)^T\left(\mathbf{W^K}c_j + a_{ij}^K\right)}{\sqrt{d_k}} \quad (4)$$

where $\alpha_{ij} = \frac{\exp e_{ij}}{\sum_{k=1}^{n} \exp e_{ik}}$, $\mathbf{W^Q}$, $\mathbf{W^K}$ and $\mathbf{W^V}$ are trainable matrices for queries, keys and values; d_k is the dimension of queries and keys; a_{ij}^K and a_{ij}^V are relative positional representations for positions i and j.

3.3 Decoder with Copy Mechanism

Similar to the code token encoder, we adopt Transformer as the backbone of the decoder. Unlike the original decoder module in (Vaswani et al., 2017), we need to integrate two encoding sources from code and AST encoders. The serial strategy (Libovický et al., 2018) is adopted, which is to compute the encoder-decoder attention one by one for each input encoder (Fig. 4). In each cross-attention layer, the encoding of ASTs ($h^{(a)} = (h_1^{(a)}, ..., h_l^{(a)})$ flatted by preorder traversal) or codes ($o = (o_1, ..., o_n)$) is queried by the output of the preceding summary self-attention $s = (s_1, ..., s_m)$.

$$
\begin{aligned}
z_i &= \sum_{j=1}^{n} \alpha_{ij} \left(\mathbf{W^V} h_j^{(a)} \right), \alpha_{ij} = \frac{\exp e_{ij}^{ast}}{\sum_{k=1}^{n} \exp e_{ik}^{ast}} \\
y_i &= \sum_{j=1}^{n} \alpha_{ij} \left(\mathbf{W^V} o_j \right), \alpha_{ij} = \frac{\exp e_{ij}^{code}}{\sum_{k=1}^{n} \exp e_{ik}^{code}} \\
e_{ij}^{ast} &= \frac{(\mathbf{W_d^Q} s_i)^T \left(\mathbf{W_d^K} h_j^{(a)} \right)}{\sqrt{d_k}}, e_{ij}^{code} = \frac{(\mathbf{W_d^Q} z_i)^T \left(\mathbf{W_d^K} o_j \right)}{\sqrt{d_k}}
\end{aligned} \tag{5}
$$

where $\mathbf{W_d^Q}$, $\mathbf{W_d^K}$ and $\mathbf{W_d^V}$ are trainable projection matrices for queries, keys and values. $\mathbf{l}$ is the number of subtrees. $\mathbf{m}$ and $\mathbf{n}$ are the length of code and summary tokens, respectively. Following (Vaswani et al., 2017), we adopt a multi-head attention mechanism in the self-attention and cross-attention layers of the decoder. After stacking several decoder layers, we add a softmax operator to obtain the generation probability $P_t^{(g)}$ of each summary token.

We further incorporate the copy mechanism (See et al., 2017) to enable the decoder to copy rare tokens directly from the input codes. This is motivated by the fact that many tokens (about 28% in the Funcom dataset) are directly copied from the source code (e.g., function names and variable names) in the summary. Specifically, we learn a copy probability through an attention layer:

$$
P_t^{(c)}(i) = \frac{\exp(< \mathbf{W^{cp}} \mathbf{h}_i^{(c)}, \mathbf{h}_t^{(s)} >)}{\sum_{k=1}^{T_c} \exp(< \mathbf{W^{cp}} \mathbf{h}_k^{(c)}, \mathbf{h}_t^{(s)} >)}, \tag{6}
$$

where $P_t^{(c)}(i)$ is the probability for choosing the i-th token from source code in the summary position t, $\mathbf{h}_i^{(c)}$ is the encoding vector of the i-th code token, $\mathbf{h}_t^{(s)}$ is the decoding vector of the t-th summary token, $\mathbf{W^{cp}}$ is a learnable projection matrix to map $h_i^{(c)}$ to the space of $h_i^{(s)}$, and T_c is the code length. The final probability for selecting the token w as t-th summary token is defined as:

$$
P_t(w) = \gamma_t P_t^{(g)}(w) + (1 - \gamma_t) \sum_{i: w_i^{(c)} = w} P_t^{(c)}(i), \tag{7}
$$

where $w_i^{(c)}$ is the i-th code token and γ_t is a learned combination probability defined as $\gamma_t = \text{sigmoid}(\Gamma(\mathbf{h}_t^{(s)}))$, where Γ is a feed forward neural network. Finally, we use Maximum Likelihood Estimation as the objective function and apply AdamW for optimization.

4 Experimental Setup

4.1 Dataset and Preprocessing

In our experiment, we adopt the public Java datasets TL-CodeSum (Hu et al., 2018b) and Funcom (LeClair et al., 2019), which are widely used in previous studies (Ahmad et al., 2020; Hu et al., 2018a, 2019, 2018b; LeClair et al., 2020, 2019; Zhang et al., 2020; Wei et al., 2020). The partitioning of train/validation/test sets follows the original datasets. We split code tokens by camel case and snake case, replace numerals and string literals with the generic tokens `<NUM>` and `<STRING>`, and set all to lower case. We extract the first sentence of the method's Javadoc description as the ground truth summary. Code that cannot be parsed by the Antlr parser (Parr, 2013) is removed. At last, we obtain $83,661$ and $2,111,230$ pairs of source code and summaries on TL-CodeSum and Funcom, respectively.

4.2 Experiment Settings

We implement our approach based on the open-source project OpenNMT (Klein et al., 2017). The vocabulary sizes are $10,000$, $30,000$ and $50,000$ for AST, code, and summary, respectively. The batch size is set to 128 and the maximum number of epochs is 200/40 for TL-CodeSum and Funcom. For optimizer, we use the AdamW (Loshchilov and Hutter, 2019) with the learning rate 10^{-4}. To alleviate overfitting, we adopt early stopping with patience 20. The experiments are conducted on a server with 4 GPUs of NVIDIA Tesla V100 and it takes about 10 and 40 minutes each epoch for TL-CodeSum and Funcom, respectively. Detailed

hyper-parameter settings and training time can be found in Appendix.

4.3 Evaluation Metrics

Similar to previous work (Iyer et al., 2016; Wan et al., 2018; Zhang et al., 2020), we evaluate the performance of our proposed model based on four widely-used metrics, including BLEU (Papineni et al., 2002), Meteor (Banerjee and Lavie, 2005), Rouge-L (Lin, 2004) and Cider (Vedantam et al., 2015). These metrics are prevalent metrics in machine translation, text summarization, and image captioning. Note that we report the scores of BLEU, Meteor (Met. for short), and Rouge-L (Rouge for short) in percentages since they are in the range of $[0, 1]$. As Cider scores are in the range of $[0, 10]$, we display them in real values. In addition, we notice that the related work on code summarization uses different BLEU implementations, such as BLEU-ncs, BLEU-M2, BLEU-CN, BLEU-FC, etc (named by (Gros et al., 2020)). And there are subtle differences in the way the BLEUs are calculated (Gros et al., 2020). We choose the widely used BLEU-CN (Iyer et al., 2016; Alon et al., 2019a; Feng et al., 2020b; Wang et al., 2020) as the BLEU metric in this work. Detailed metrics description can be found in Appendix.

5 Experimental Results

5.1 The Effectivness of CAST

We evaluate the effectiveness of CAST by comparing it to the recent DNN-based code summarization models introduced in Sec. 2.2: CodeNN, HybridDrl, HDeepcom, Attgru, Astattgru, Code2seq, and NCS. To make a fair comparison, we extend ASTNN to CodeAstnn, with an additional code token encoder as ours, so that the only difference between them is the AST representation.

From the results in Table 1, we can see that CAST outperforms all the baselines on both datasets. CodeNN, Code2seq, Attgru, and NCS only use code or AST information. Among them, NCS performs better because it applies a transformer to capture the long-range dependencies among code tokens. Astattgru and CodeAstnn outperform Attgru because of the addition of AST channels. Note that our model outperforms other baselines even without the copy mechanism or aggregation. This is because we split an AST into block-level subtrees and each subtree contains relatively complete semantics. On the contrary, related work such as ASTNN splits an AST into statement-level subtrees, which only represent a single statement and relatively fragmented semantics.

5.2 Comparison of Different AST Representations

We evaluate the performance of different AST representations by comparing CAST with Code2seq, HybridDrl, Astattgru, and CodeAstnn. Table 1 shows that CAST performs the best among them. As linearization-based methods, Astattgru flattens an AST to a sequence and Code2seq obtains a set of paths from an AST, both losing some hierarchical information of ASTs naturally. As tree-based methods, HybridDrl transforms ASTs to binary trees and trains on the full ASTs with tree-based models. This leads to AST structural information loss, gradient vanishing problem, and slow training process (21 hours each epoch in Funcom)[3]. Both CodeAstnn and CAST perform better than HybridDrl, Code2seq, and Astattgru because they split a large AST into a set of small subtrees, which can alleviate the gradient vanishing problem. Our CAST achieves the best performance and we further explain it from two aspects: splitting granularities of ASTs. and the AST representation learning.

For splitting granularities of ASTs, CodeAstnn is statement-level splitting, leading to subtrees 71% smaller than ours on TL-CodeSum[4]. Therefore, it may not be able to capture the syntactical information and semantic information. In terms of AST representation learning, CodeAstnn and CAST all use RvNN and Max-pooling to learn the representation of subtrees but different ways to aggregate them. The former applies a RNN-based model to aggregate the subtrees. It only captures the sequential structure and the convergence becomes worse as the number of subtrees increases (Bengio et al., 1993). The latter applies RvNN to aggregate all subtrees together according to their relative positions in the hierarchy, which can combine the semantics of subtrees well.

5.3 Ablation Study

To investigate the usfulness of the subtree aggregation (Sec. 3.1.2) and the copy mechanism (Sec. 3.3), we conduct ablation studies on two variants of CAST. The results of the ablation study are given in the bottom of Table 1.

[3]See training time details in Appendix Table 2 and 3.

[4]See dataset statistics in Appendix Table 5 to 8.

Model	Funcom				TL-CodeSum			
	BLEU	Met.	Rouge	Cider	BLEU	Met.	Rouge	Cider
CodeNN	20.93	11.44	29.09	0.90	22.22	14.08	33.14	1.67
HDeepcom	25.71	15.59	36.07	1.42	23.32	13.76	33.94	1.74
Attgru	27.82	18.10	39.20	1.84	29.72	17.03	38.49	2.35
NCS	29.18	19.94	40.09	2.15	40.63	24.85	52.00	3.47
Code2seq	23.84	13.84	33.65	1.31	16.09	8.94	24.21	0.66
HybridDrl	23.25	12.55	32.04	1.11	23.51	15.38	33.86	1.55
Astattgru	28.17	18.43	39.56	1.90	30.78	17.35	39.94	2.31
CodeAstnn	28.27	18.86	40.34	1.94	41.08	24.95	51.67	3.49
$CAST_A$	30.56	20.96	42.46	2.30	43.76	27.15	54.09	3.84
$CAST_C$	30.35	20.65	42.22	2.24	43.81	26.95	53.53	3.82
CAST	**30.83**	**20.96**	**42.71**	**2.31**	**45.19**	**27.88**	**55.08**	**3.95**

Table 1: Comparison with baselines.

Model	Informativeness	Naturalness	Similarity
CAST	**2.74**(1.29)	**3.08**(1.23)	**2.66**(1.29)
Astattgru	2.26(1.05)	2.46(1.31)	2.02(1.09)
NCS	2.39(1.10)	2.78(1.19)	2.17(1.14)
CodeAstnn	2.44(1.08)	3.00(1.13)	2.20(1.14)

Table 2: Results of human evaluation (standard deviation in parentheses).

- $CAST_A$: CAST without subtree aggregation, which directly uses the subtree vectors obtained by Eq. (2) as AST representation. Our results show that the performance of $CAST_A$ drops compared to CAST (except for Met. in Funcom), demonstrating that it is beneficial to reconstruct and aggregate information from subtrees.
- $CAST_C$: CAST without copy mechanism. Our results show that CAST outperforms $CAST_C$, confirming that the copy mechanism can copy tokens (especially the out-of-vocabulary ones) from input code to improve the performance of summarization.

5.4 Human Evaluation

Besides textual similarity based metrics, we also conduct a human evaluation by following the previous work (Iyer et al., 2016; Liu et al., 2019; Hu et al., 2019; Wei et al., 2020) to evaluate semantic similarity of the summaries generated by CAST, Astattgru, NCS and CodeAstnn. We randomly choose 50 Java methods from the testing sets (25 from TL-CodeSum and 25 from Funcom) and their summaries generate by four approaches. Specially, we invite 10 volunteers with more than 3 years of software development experience and excellent English ability. Each volunteer is asked to assign scores from 0 to 4 (the higher the better) to the generated summary from the three aspects: **similarity** of the generated summary and the ground truth summary, **naturalness** (grammaticality and fluency), and **informativeness** (the amount of content carried over from the *input code* to the generated summary, ignoring fluency). Each summary is evaluated by four volunteers, and the final score is the average of them.

Table 2 shows that CAST outperforms others in all three aspects. Our approach is better than other approaches in Informative, which means that our approach tends to generate summaries with comprehensive semantics. In addition, we confirm the superiority of our approach using Wilcoxon signed-rank tests (Wilcoxon et al., 1970) for the human evaluation. And the results[5] reflect that the improvement of CAST over other approaches is statistically significant with all p-values smaller than 0.05 at 95% confidence level (except for CodeAstnn on Naturalness).

6 Threats to Validity

There are three main threats to validity. First, we evaluate and compare our work only on a Java dataset, although in principle, the model should generalize to other languages, experiments are needed to validate it. Also, AST splitting algorithm need to be implemented for other languages by implementing a visitor to AST.

[5] See Appendix Table 9

Second, in neural network model design, there are many orthogonal aspects such as different token embeddings, whether to use beam search, teacher forcing. When showing the generality of CAST, we have done the experiments in a controlled way. A future work might be to do all experiments in a more controlled way and the performance of CAST could rise further when combined with all other orthogonal techniques.

Third, summaries in the datasets are collected by extracting the first sentences of Javadoc. Although this is a common practice to place a method's summary at the first sentence of Javadoc, there might still be some mismatch summaries. A higher quality dataset with better summaries collecting techniques is needed in the future.

7 Conclusion

In this paper, we propose a new model CAST that splits the AST of source code into several subtrees, embeds each subtree, and aggregates subtrees' information back to form the full AST representation. This representation, along with code token sequence information, is then fed into a decoder to generate code summaries. Experimental results have demonstrated the effectiveness of CAST and confirmed the usefulness of the abstraction technique. We believe our work sheds some light on future research by pointing out that there are better ways to represent source code for intelligent code understanding.

8 Acknowledgement

We thank reviewers for their valuable comments on this work. This research was supported by National Key R&D Program of China (No. 2017YFA0700800). We would like to thank Chin-Yew Lin, Jian-Guang Lou, Jiaqi Guo and Qian Liu for their valuable suggestions and feedback during the paper writing process. We also thank the participants of our human evaluation for their time.

References

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A transformer-based approach for source code summarization. In *ACL*.

Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles A. Sutton. 2015. Suggesting accurate method and class names. In *FSE*.

Miltiadis Allamanis, Hao Peng, and Charles A. Sutton. 2016. A convolutional attention network for extreme summarization of source code. In *ICML*.

Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019a. code2seq: Generating sequences from structured representations of code. In *ICLR*.

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019b. code2vec: learning distributed representations of code. In *POPL*.

Satanjeev Banerjee and Alon Lavie. 2005. METEOR: an automatic metric for MT evaluation with improved correlation with human judgments. In *IEEvaluation@ACL*.

Yoshua Bengio, Paolo Frasconi, and Patrice Y. Simard. 1993. The problem of learning long-term dependencies in recurrent networks. In *ICNN*.

Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. 2021. Infercode: Self-supervised learning of code representations by predicting subtrees. In *ICSE*.

Lun Du, Xiaozhou Shi, Yanlin Wang, Ensheng Shi, Shi Han, and Dongmei Zhang. 2021. Is a single model enough? mucos: A multi-model ensemble learning for semantic code search. In *CIKM*.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020a. Codebert: A pre-trained model for programming and natural languages. *arXiv Preprint*.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020b. Codebert: A pre-trained model for programming and natural languages. In *EMNLP (Findings)*.

David Gros, Hariharan Sezhiyan, Prem Devanbu, and Zhou Yu. 2020. Code to comment "translation": Data, metrics, baselining & evaluation. In *ASE*.

Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API learning. In *FSE*.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2021. Graphcodebert: Pre-training code representations with data flow. In *ICLR*.

Sonia Haiduc, Jairo Aponte, and Andrian Marcus. 2010a. Supporting program comprehension with source code summarization. In *ICSE*.

Sonia Haiduc, Jairo Aponte, Laura Moreno, and Andrian Marcus. 2010b. On the use of automated text summarization techniques for summarizing source code. In *WCRE*.

Tjalling Haije. 2016. *Automatic comment generation using a neural translation model*. Bachelor's thesis, University of Amsterdam.

Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018a. Deep code comment generation. In *ICPC*.

Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2019. Deep code comment generation with hybrid lexical and syntactical information. *Empirical Software Engineering*.

Xing Hu, Ge Li, Xin Xia, David Lo, Shuai Lu, and Zhi Jin. 2018b. Summarizing source code with transferred API knowledge. In *IJCAI*.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *ACL*.

Guillaume Klein, Yoon Kim, Yuntian Deng, Jean Senellart, and Alexander M Rush. 2017. Opennmt: Open-source toolkit for neural machine translation. *arXiv*.

Alexander LeClair, Sakib Haque, Linfgei Wu, and Collin McMillan. 2020. Improved code summarization via a graph neural network. In *ICPC*.

Alexander LeClair, Siyuan Jiang, and Collin McMillan. 2019. A neural model for generating natural language summaries of program subroutines. In *ICSE*.

Jindřich Libovický, Jindřich Helcl, and David Mareček. 2018. Input combination strategies for multi-source transformer decoder. In *WMT*.

Chin-Yew Lin. 2004. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*.

Zhongxin Liu, Xin Xia, Christoph Treude, David Lo, and Shanping Li. 2019. Automatic generation of pull request descriptions. In *ASE*.

Ilya Loshchilov and Frank Hutter. 2019. Decoupled weight decay regularization. In *ICLR*.

Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional neural networks over tree structures for programming language processing. In *AAAI*.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *ACL*.

Terence Parr. 2013. *The definitive ANTLR 4 reference (2 ed.)*. Pragmatic Bookshelf.

Abigail See, Peter J. Liu, and Christopher D. Manning. 2017. Get to the point: Summarization with pointer-generator networks. In *ACL*.

Eui Chul Richard Shin, Miltiadis Allamanis, Marc Brockschmidt, and Alex Polozov. 2019. Program synthesis and semantic parsing with learned code idioms. In *NeurIPS*, pages 10824–10834.

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

Ramakrishna Vedantam, C. Lawrence Zitnick, and Devi Parikh. 2015. Cider: Consensus-based image description evaluation. In *CVPR*.

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 *ASE*.

Yanlin Wang, Lun Du, Ensheng Shi, Yuxuan Hu, Shi Han, and Dongmei Zhang. 2020. Cocogum: Contextual code summarization with multi-relational gnn on umls. Technical report, Microsoft, Tech. Rep. MSR-TR-2020-16, May 2020.[Online].

Yanlin Wang and Hui Li. 2021. Code completion by modeling flattened abstract syntax trees as graphs. In *AAAI*.

Bolin Wei, Yongmin Li, Ge Li, Xin Xia, and Zhi Jin. 2020. Retrieve and refine: Exemplar-based neural comment generation. In *ASE*, pages 349–360. IEEE.

Frank Wilcoxon, SK Katti, and Roberta A Wilcox. 1970. Critical values and probability levels for the wilcoxon rank sum test and the wilcoxon signed rank test. *Selected tables in mathematical statistics*, 1:171–259.

Xiaohan Yu, Quzhe Huang, Zheng Wang, Yansong Feng, and Dongyan Zhao. 2020. Towards context-aware code comment generation. In *EMNLP (Findings)*, pages 3938–3947. Association for Computational Linguistics.

Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2020. Retrieval-based neural source code summarization. In *ICSE*.

Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In *ICSE*.