Code Completion by Modeling Flattened Abstract Syntax Trees as Graphs

Yanlin Wang,[1] **Hui Li**[2*]
[1] Microsoft Research Asia
[2] School of Informatics, Xiamen University
yanlwang@microsoft.com, hui@xmu.edu.cn

Abstract

Code completion has become an essential component of integrated development environments. Contemporary code completion methods rely on the abstract syntax tree (AST) to generate syntactically correct code. However, they cannot fully capture the sequential and repetitive patterns of writing code and the structural information of the AST. To alleviate these problems, we propose a new code completion approach named CCAG, which models the flattened sequence of a partial AST as an AST graph. CCAG uses our proposed AST Graph Attention Block to capture different dependencies in the AST graph for representation learning in code completion. The sub-tasks of code completion are optimized via multi-task learning in CCAG, and the task balance is automatically achieved using uncertainty without the need to tune task weights. The experimental results show that CCAG has superior performance than state-of-the-art approaches and it is able to provide intelligent code completion.

1 Introduction

Code completion, which provides code suggestions for developers, is one of the most attractive features in integrated development environments (IDEs). According to the study of Murphy, Kersten, and Findlater (2006), users of Eclipse IDE used the code suggestion of Eclipse as much as the common editing commands (e.g., copy and paste) since it reduces the required amount of typing and eliminates typos.

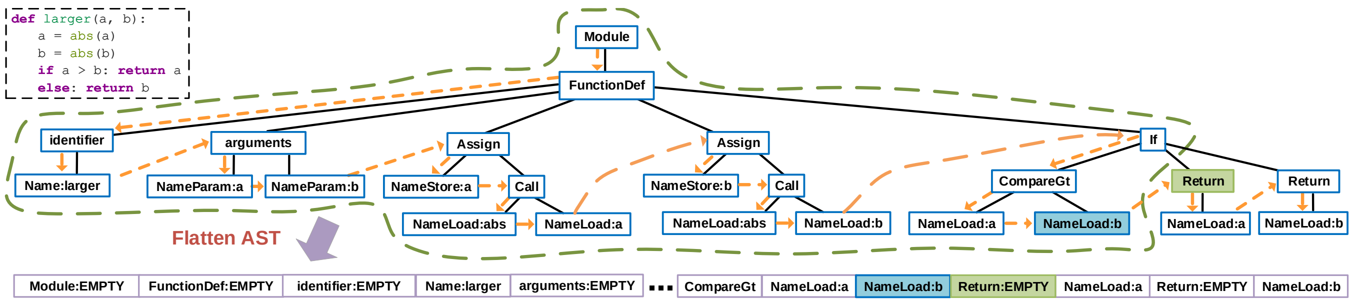
Hindle et al. (2012) firstly reduce code completion to a natural language processing (NLP) problem. Thereafter, many researchers leverage NLP techniques to design code completion engines (Allamanis et al. 2018; Le, Chen, and Babar 2020). Early works generate code as a sequence of code tokens (Hindle et al. 2012). However, directly modeling tokens of code sequences sometimes fails to produce syntactically correct code (Brockschmidt et al. 2019). Recent works (Liu et al. 2016; Li et al. 2018) alleviate this issue by using the target language's grammar to generate abstract syntax trees (ASTs) which are syntactically correct by construction. They show that AST based code completion, which contains value prediction and type prediction as sub-tasks, can provide more intelligent code suggestions, and it has been adopted in several IDEs (e.g., Visual Studio Code IDE (Svyatkovskiy et al. 2019)). In this paper, the term "code completion" refers to AST based code completion.

To model the tree structure of AST, most code completion methods opt to flatten the AST using pre-order depth-first traversal (Liu et al. 2016; Li et al. 2018). Then, powerful deep learning techniques (e.g., LSTMs and Transformer) can be adopted for learning the representation of the flattened AST sequence for later use in code completion. In addition to code completion, recent works on learning program representations (Allamanis, Brockschmidt, and Khademi 2018; Brockschmidt et al. 2019) for other downstream tasks (e.g., variable prediction and hole completion) have shed some light on the benefits of modeling an AST as a graph of which the representation can be learned using Graph Neural Network (GNN) (Wu et al. 2020).

However, neither of the two paradigms can fully model ASTs for the code completion task since they neglect the sequential and repetitive patterns of humans on writing code, and the structural information of AST: (1) Firstly, when writing code, the skeleton (i.e., function declaration) is often written first and other statements in the body of the function are written one by one just like the pre-order depth-first traversal of AST (Yang 2020). Such sequential information is important to code completion. (2) Moreover, code completions are surprisingly repetitive (Hellendoorn et al. 2019). For example, the study of Aye and Kaiser (2020) on a codebase from GitHub shows that there is a 59.8% probability that any keyword, identifier, or literal repeats one of the previous 100 tokens. Capturing such a repetitive pattern can enhance code completion. (3) Lastly, the structural information of AST provides strong indications on the dependencies between linked nodes which should be considered. Simple sequential modeling ignores the repetitive pattern and the structural information while the vanilla graph based modeling neglects the sequential and repetitive patterns. In addition to the problem of AST modeling, some code completion methods (Liu et al. 2016, 2020) model the sub-tasks (i.e., value prediction and type prediction) via multi-task learning (Vandenhende et al. 2020), but the task weights are manually set. Therefore, these methods suffer from the task imbalance which will impede proper training in multi-task learning (Chen et al. 2018; Vandenhende et al. 2020).

To address the problems mentioned above, we propose an

[*] Hui Li is the corresponding author.



```
def larger(a, b):
    a = abs(a)
    b = abs(b)
    if a > b: return a
    else: return b
```

Figure 1: The AST of a Python function. The orange dashed arrows show the traversal for the flattening. The green dashed line indicates a partial AST with `NameLoad:b` being the *right-most* node and `Return` (i.e., `Return:EMPTY` in the flattened AST) being the next node to predict.

effective Code Completion method by modeling flattened ASTs as Graphs (CCAG for short). Compared to previous methods, the contributions of our work are:

- CCAG models a flattened AST derived from the pre-order depth-first traversal of a partial AST as a graph, and it is tailored to include the sequential and repetitive patterns of writing code and the structural information of the AST.
- CCAG uses our proposed AST Graph Attention Block (ASTGab) comprised of three different attention based layers. Each layer captures differing dependencies among AST nodes. ASTGab is further enhanced with the residual connection so that multiple ASTGabs can be stacked to improve the performance.
- CCAG adopts an uncertainty based method to automatically balance the two sub-tasks of code completion in multi-task learning without the need to tune task weights.

We conduct extensive experiments on benchmark data for evaluating code completion. Results show that CCAG surpasses state-of-the-art code completion approaches.

2 Preliminary

Any programming language has an explicit context-free grammar (CFG), and it can be used to parse source code into an AST which represents the abstract syntactic structure of source code. An AST is a tree where each non-leaf node corresponds to a *non-terminal* in the CFG specifying structural information (e.g., `ForStatement` and `IfStatement`). Each leaf node corresponds to a *terminal* (e.g., variable names and operators) in the CFG encoding program text. An AST can be converted back into source code easily. Fig. 1 provides an example of the AST for a python function. We can see that each non-leaf node contains a type attribute (e.g., `Module`) and each leaf node contains a type attribute and a value attribute (e.g., `NameLoad:a` means that the type is `NameLoad` and the value is `a`).

Code completion consumes a partial AST as the input:

Definition 1 (Partial AST (Liu et al. 2016))**.** Given a complete AST T, a partial AST is a subtree T' of T, such that for each node n in T', its left sequence $L_T(n)$ with respect to T is a subset of T', i.e., $L_T(n) \subseteq T'$. Here, the left sequence $L_T(n)$ is defined as all the nodes that are visited earlier than n in the pre-order depth-first traversal sequence of T.

For example, the green dashed line in Fig. 1 illustrates a partial AST. Following Li et al. (2018), we append `EMPTY` as the value to each non-leaf node when flattening the AST. The formal definition of code completion is as follows:

Definition 2 (Code Completion (Liu et al. 2016))**.** Each partial AST T' has one *right-most* node n_R, and all other nodes of T' form its left sequence $L_T(n_R)$. We call the next node after n_R in the pre-order depth-first traversal sequence of the complete AST T as the next node following T'. Given a partial AST T', the task of code completion is to predict the value and the type of the next node following T'.

For the partial AST shown in Fig. 1, the right-most node is `NameLoad:b` and the next node to predict is `Return` (i.e., `Return:EMPTY`). A successful model should give both the value `EMPTY` and the type `Return` as predictions.

The traversal order of ASTs in code completion[1] is consistent with the way that developers implement a function: the function declaration is often written first and then other statements are written one by one (Yang 2020).

3 Learning to Complete Code with CCAG

In this section, we will describe the details of CCAG. Fig. 2 provides an overview of CCAG.

3.1 Program Representation

CCAG models each flattened AST sequence of a partial AST as an AST graph. Fig. 3 shows how CCAG represents the partial AST in Fig. 1 as a graph. Duplicated nodes in the flattened AST are merged into one node in the graph. Each *node-node* edge in the graph indicates that the two linked nodes are adjacent in the flattened AST sequence. Node-node edges are undirected, allowing information propagation in both directions. The weight of a node-node edge is the frequency of the occurrence of the edge in the corresponding flattened AST sequence.

However, flattening a partial AST into a sequence may result in the information loss of the tree structure. Following Li

[1]Note that some works use in-order depth-first traversal to define the problem (Liu et al. 2016; Li et al. 2018; Liu et al. 2020). But the AST examples in their papers and the code completion task in their experiments use pre-order depth-first traversal.

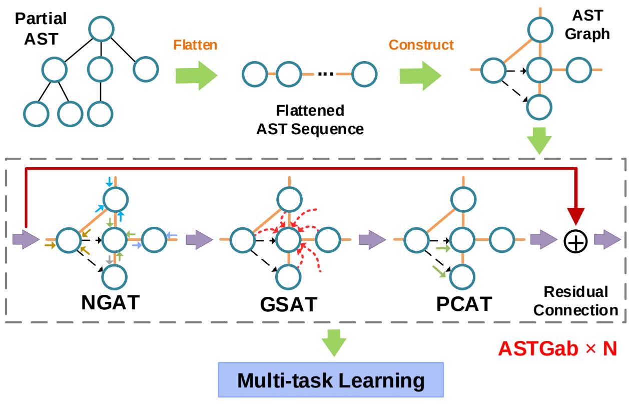


Figure 2: Overview of CCAG. In graphs, orange lines show the traversal for flattening the AST or the node-node edges, and black dashed lines are parent-child edges.

et al. (2018), we record the parent node of each node in original ASTs since parent-child information can help the model learn the hierarchical structure of the AST. Then, we add *parent-child* edges to the AST graph. Parent-child edges are directed (from parent to child) and unweighted to retain the structure. Each node may have more than one parent node.

One remaining issue is that the positional information of each node in the flattened AST sequence is missing in the graph, since repeated AST nodes are merged into one node in the graph. To remedy it, we record the distance between the last occurrence of each node and the right-most node in the flattened AST sequence as the positional encoding. For instance, assuming that the flattened AST sequence is $\{n_1, n_2, n_3, n_2\}$ with n_2 being the right-most node, the position embeddings $\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3$ for n_1, n_2 and n_3 are vectors with all dimensions being 3, 0 and 1, respectively. Note that the position embeddings are fixed and will not be updated.

For a node n_i, we embed its value and type into two separate spaces. $\mathbf{v}_i, \mathbf{t}_i \in \mathbb{R}^d$, $\mathbf{p}_i \in \mathbb{R}^{2d}$ are its value embedding vector, type embedding vector and position embedding vector, respectively. The representation $\mathbf{h}_i$ of n_i is as follows:

$$\mathbf{h}_i = ReLU\big(\mathbf{W}^{(p)}([\mathbf{t}_i \,||\, \mathbf{v}_i] + \mathbf{p}_i) + \mathbf{b}^{(p)}\big), \quad (1)$$

where $||$ indicates the concatenation operation, $\mathbf{W}^{(p)} \in \mathbb{R}^{d\times 2d}$ is a parameter matrix and $\mathbf{b}^{(p)}$ is the bias vector.

Since we first flatten the partial AST and then derive the AST graph from the flattened AST sequence, one may ask why using such indirect modeling. Here, we provide discussions on alternatives to justify the rationality of our design. In Sec. 4.2, we will also show that our design leads to superior performance than the alternatives.

- **Alternative 1: Directly modeling the original partial AST as a graph/tree**. In such a design, sequential information is missing, and repeated nodes may be located in different parts of the graph which makes it hard for the model to capture the repetitive pattern. A tree can be viewed as a special graph and has same issues as mentioned above.
- **Alternative 2: Directly modeling the flattened AST sequence.** Directly modeling the flattened AST sequence also makes capturing the repetitive pattern harder since repeated nodes can be physically distant from each other in the sequence. Additionally, simple sequential modeling will result in the loss of the structural information of the partial AST.

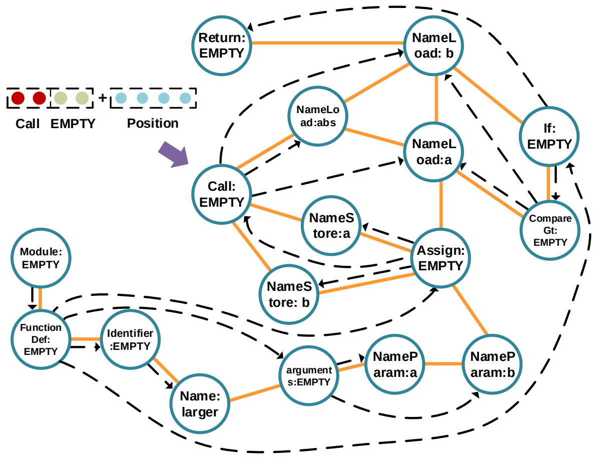


Figure 3: The AST graph in CCAG for the partial AST in Fig. 1. Orange lines show the node-node edges and black dashed lines indicate the parent-child edges.

Differently, in CCAG, the sequential pattern is incorporated by modeling the flattened AST sequence, the repetitive pattern can be captured via the merged repeated nodes and the weights of node-node edges, and the structural information is retained via the parent-child edges.

3.2 AST Graph Attention Block (ASTGab)

We adopt the idea of GNN to design an AST Graph Attention Block (ASTGab) for learning the representation of the AST graph. GNN is well-suited for learning AST graphs since it is designed to automatically extract features from the rich node connections in the graph data (Wu et al. 2020). The input to the ASTGab is the initial embeddings of all AST nodes in one AST graph derived from a partial AST. An ASTGab consists of three different layers which extract node features at different levels, as shown in Fig. 2. The output is handled by a residual connection to overcome the difficulty of training deep neural networks.

Neighbor Graph Attention Layer (NGAT). The initial embeddings are first fed into a NGAT which extracts features from the *first-order* neighborhood along the node-node edges in the AST graph. Strong dependencies typically exist between first-order neighbors. For instance, the comparison operators are normally followed by loading the variable. In the AST graph, this indicates a node-node edge between two AST nodes (e.g., the node-node edge between `CompareGt:EMPTY` and `NameLoad:a` in Fig. 3).

Inspired by Velickovic et al. (2018), we perform self-attention on every pairs of the first-order neighbors connected by the node-node edges in the AST graph and compute the first-order neighbor attention coefficient in NGAT:

$$e^{(n)}_{i,j} = a(\mathbf{W}^{(n)}\mathbf{h}_i, \mathbf{W}^{(n)}\mathbf{h}_j, w_{i,j}), \quad (2)$$

where $e_{i,j}^{(n)}$ shows the attention coefficient between first-order neighbors i and j, a is an attention mechanism, $\mathbf{W}^{(n)} \in \mathbb{R}^{d\times d}$ is the shared weight matrix for all first-order node pairs, and $w_{i,j}$ is the weight of the node-node edge between i and j. The first-order neighbor attention coefficients are then normalized across all the first-order neighbors of an AST node using softmax. There are many choices for the design of a. We adopt a single-layer feedforward neural network followed by a LeakyReLU non-linearity unit with a negative input slope of 0.2 as Velickovic et al. (2018). This is equivalent to the following expression:

$$\alpha_{i,j}^{(n)} = \frac{\exp\Big(LeakyReLU\Big(\mathbf{a}^{(n)T}\left[\mathbf{W}^{(n)}\mathbf{h}_i \,||\, \mathbf{W}^{(n)}\mathbf{h}_j \,||\, w_{i,j}\right]\Big)\Big)}{\sum_{k\in\mathcal{N}_i}\exp\Big(LeakyReLU\Big(\mathbf{a}^{(n)T}\left[\mathbf{W}^{(n)}\mathbf{h}_i \,||\, \mathbf{W}^{(n)}\mathbf{h}_k \,||\, w_{i,k}\right]\Big)\Big)}, \tag{3}$$

where $\mathcal{N}_i$ indicates the set of first-order neighbors of node i along node-node edges, and $\mathbf{a}^{(n)} \in \mathbb{R}^{2d+1}$ and $\mathbf{W}^{(n)} \in \mathbb{R}^{d\times d}$ are parameter vector and matrix, respectively. Then, the node feature for each AST node is updated as:

$$\mathbf{h}_i^{(n)} = ReLU\big(\sum_{j\in\mathcal{N}_i}\alpha_{i,j}^{(n)}\mathbf{W}^{(a)}\mathbf{h}_j\big), \tag{4}$$

where $\mathbf{W}^{(a)} \in \mathbb{R}^{d\times d}$ is the weight matrix.

As suggested by Velickovic et al. (2018), we employ multi-head attention (Vaswani et al. 2017) in order to stabilize the learning of NGAT. Specifically, we perform M independent transformations in Eq. 4 and use the average of the M results as the representation of each AST node:

$$\mathbf{h}_i^{(n)} = ReLU\big(\frac{1}{M}\sum_{m=1}^{M}\sum_{j\in\mathcal{N}_i}\alpha_{i,j,m}^{(n)}\mathbf{W}_m^{(a)}\mathbf{h}\big), \tag{5}$$

where $\mathbf{W}_m^{(a)} \in \mathbb{R}^{d\times d}$ is the parameter matrix for m-th head, and $\alpha_{i,j,m}^{(n)}$ is the normalized attention coefficient obtained by the m-th attention mechanism in Eq. 4.

Global Self-attention Layer (GSAT). The second component GSAT captures the importance of each node to other nodes in the graph. Compared to NGAT, which focuses on the local neighborhood, GSAT brings a global vision of the AST graph to CCAG. Since the AST graph only contains unique AST nodes of which the number is much smaller than the original AST, we can feed the entire AST graph into GSAT, and then the self-attention mechanism is adopted to draw the global dependencies:

$$\mathbf{H}^{(g)} = softmax\big(\frac{(\mathbf{W}^{(k)}\mathbf{H}^{(n)})^T(\mathbf{W}^{(q)}\mathbf{H}^{(n)})}{\sqrt{d}}\big)\big(\mathbf{W}^{(v)}\mathbf{H}^{(n)}\big)^T, \tag{6}$$

where $\mathbf{H}^{(n)}$ are the representations of AST nodes from NGAT, and $\mathbf{W}^{(k)}, \mathbf{W}^{(q)}, \mathbf{W}^{(v)} \in \mathbb{R}^{d\times d}$ are parameters.

Parent-child Attention Layer (PCAT). The third layer PCAT captures the structural information from parent-child edges and refine the node features from GSAT. Precisely, we adopt a two-layer attention mechanism in PCAT:

$$\begin{aligned}\mathbf{p}_i &= ReLU\big(\mathbf{W}_1^{(p1)}\mathbf{h}_i^{(g)} + \frac{1}{j\in|\mathcal{P}_i|}\mathbf{W}_2^{(p1)}\sum_{j\in\mathcal{P}_i}\mathbf{h}_j^{(g)} + \mathbf{b}^{(p1)}\big)\\ \mathbf{h}_i^{(p)} &= ReLU\big(\mathbf{W}^{(p2)}\mathbf{p}_i + \mathbf{b}^{(p2)}\big)\end{aligned} \tag{7}$$

where $\mathbf{h}_i^{(p)}$ is the representation for node i outputted by PCAT and it incorporates the feature(s) from its parent node(s). $\mathcal{P}_i$ indicates the set of parent nodes of node i, $|\mathcal{P}_i|$ is the set size of $\mathcal{P}_i$, $\mathbf{h}_i^{(g)} \in \mathbf{H}^{(g)}$ is the representation for node i from GSAT, $\mathbf{W}_1^{(p1)}, \mathbf{W}_2^{(p1)}, \mathbf{W}^{(p2)} \in \mathbb{R}^{d\times d}$ are learnable parameter matrices, and $\mathbf{b}^{(p1)}, \mathbf{b}^{(p2)}$ are bias vectors.

Residual Connection. As shown in Fig. 2, we can stack multiple ASTGabs to increase the non-linearities of CCAG and the input to each ASTGab is the output from the previous ASTGab. However, deeper neural networks face the difficulty of training. Thus, we add a residual connection (He et al. 2016) to each ASTGab to ease the difficulty:

$$\begin{aligned}\mathbf{r}_i &= ReLU\big(\mathbf{W}^{(r1)}\mathbf{h}_i^{(p)} + \mathbf{b}^{(r1)}\big)\\ \mathbf{h}_i^{(r)} &= \mathbf{W}^{(r2)}\mathbf{r}_i + \mathbf{b}^{(r2)} + \mathbf{h}_i\end{aligned} \tag{8}$$

where $\mathbf{W}^{(r1)}, \mathbf{W}^{(r2)} \in \mathbb{R}^{d\times d}$ are learnable weight matrices, and $\mathbf{b}^{(r1)}, \mathbf{b}^{(r2)}$ are bias vectors. $\mathbf{h}_i^{(r)}$ is the final output of one ASTGab for the AST node i.

3.3 Predicting Next AST Nodes and Optimization

As the right-most node has the most useful information of the next node, we use a soft-attention mechanism to incorporate the relevance of each node to the right-most node when generating the global representation of the AST graph:

$$\begin{aligned}\beta_{i,n_j} &= \mathbf{z}^{(t1)}\sigma\Big(\mathbf{W}_1^{(t1)}\mathbf{h}_i^{(r)} + \mathbf{W}_2^{(t1)}\mathbf{h}_{n_j}^{(r)} + \mathbf{b}^{(t1)}\Big)\\ \mathbf{s}_j &= \sum_{i\in\mathcal{G}_j}\beta_{i,n_j}\mathbf{h}_i^{(r)}\end{aligned} \tag{9}$$

where $\mathcal{G}_j$ is the set of all unique AST nodes in the AST graph j, and $\mathbf{h}_{n_j}^{(r)}$ is the output from ASTGab(s) for the right-most node of the AST graph j. $\mathbf{W}_1^{(t1)}, \mathbf{W}_2^{(t1)} \in \mathbb{R}^{d\times d}$ are weight matrices, $\mathbf{b}^{(t1)}$ is a bias vector, σ is the sigmoid function, $\mathbf{z}^{(t1)} \in \mathbb{R}^d$ is a learnable parameter vector, and $\mathbf{s}_j$ indicates the global representation of the AST graph j.

The global representation $\mathbf{s}_j$ is then projected to the value/type vocabulary space followed by softmax to generate the value/type probability distribution for the next node:

$$\begin{aligned}\bar{\mathbf{y}}_j^{(v)} &= \mathbf{W}^{(v)}\mathbf{s}_j + \mathbf{b}^{(v)},\ \hat{\mathbf{y}}_j^{(v)} = softmax\big(\bar{\mathbf{y}}_j^{(v)}\big)\\ \bar{\mathbf{y}}_j^{(t)} &= \mathbf{W}^{(t)}\mathbf{s}_j + \mathbf{b}^{(t)},\ \hat{\mathbf{y}}_j^{(t)} = softmax\big(\bar{\mathbf{y}}_j^{(t)}\big)\end{aligned} \tag{10}$$

where $\mathbf{W}^{(v)} \in \mathbb{R}^{V\times d}, \mathbf{W}^{(t)} \in \mathbb{R}^{T\times d}$ are parameters, $\mathbf{b}^{(t)} \in \mathbb{R}^V$ and $\mathbf{b}^{(t)} \in \mathbb{R}^T$ are bias vectors, and V/T is the vocabulary size of node value/type. $\hat{\mathbf{y}}_j^{(v)} \in \mathbb{R}^V$ and $\hat{\mathbf{y}}_j^{(t)} \in \mathbb{R}^T$ are the predicted value and type probability distributions of the next node to the partial AST corresponding to j.

Cross-entropy loss can be used for the optimization:

$$\mathcal{L}_v = -\sum_{j\in\mathcal{AG}}\mathbf{y}_j^{(v)}\log(\hat{\mathbf{y}}_j^{(v)}),\quad \mathcal{L}_t = -\sum_{j\in\mathcal{AG}}\mathbf{y}_j^{(t)}\log(\hat{\mathbf{y}}_j^{(t)}), \tag{11}$$

where $\mathcal{L}_v$ and $\mathcal{L}_t$ are the loss functions for value prediction and type prediction, respectively. $\mathcal{AG}$ is the set of all AST graphs generated from the data (i.e., one AST graph for one partial AST). $\mathbf{y}_j^{(v)}$ and $\mathbf{y}_j^{(t)}$ are two one-hot encodings of the ground-truth value and type for the next node, respectively.

There are two prediction tasks in code completion, and two paradigms of training exist: (1) Train two models independently (Li et al. 2018). One uses $\mathcal{L}_v$ for value prediction and the other uses $\mathcal{L}_t$ for type prediction. (2) Leverage multi-task learning (Vandenhende et al. 2020), and train one model with a global loss function (Liu et al. 2016, 2020): $\mathcal{L} = w_v\mathcal{L}_v + w_t\mathcal{L}_t$, where w_v and w_t are task weights. We choose the second training method. The reason is that value and type are two related attributes in code completion where the type can serve as a constraint to the value, and vice versa (Liu et al. 2020). Training two relevant tasks jointly via multi-task learning can benefit both tasks. Nevertheless, previous multi-task learning based code completion methods (Liu et al. 2016, 2020) treat two tasks equally (i.e., fix $w_v = w_t = 1$) during optimization, while the two tasks may have different and changing converge speeds in multi-task learning (Chen et al. 2018). Task imbalance will impede proper training because they manifest as imbalances between backpropagated gradients. However, the cost for manually tuning and updating task weights is unaffordable. To alleviate this issue, we use the idea of *uncertainty* (Kendall, Gal, and Cipolla 2018) to automatically weigh two tasks during optimization without the need to tune task weights. As a result, the joint loss[2] used by CCAG is:

$$\begin{aligned}\mathcal{L} &\approx \frac{1}{\theta^2}\mathcal{L}_v + \frac{1}{\tau^2}\mathcal{L}_t + \log\theta + \log\tau \\ &= \exp(-2\theta')\cdot\mathcal{L}_v + \exp(-2\tau')\cdot\mathcal{L}_t + \theta' + \tau',\end{aligned} \tag{12}$$

where θ and τ are learnable scalars. Their magnitudes indicate how "uniform" the discrete distributions are, which is related to the *uncertainty* as measured in entropy (Kendall, Gal, and Cipolla 2018). We let $\theta' = \log\theta, \tau' = \log\tau$, and train CCAG to learn θ' and τ' instead of unconstrained θ and τ for the numerical stability. The reason is that $\frac{1}{\theta^2}$ and $\frac{1}{\tau^2}$ may encounter the overflow error for very small θ and τ, and $\log\theta$ and $\log\tau$ will have the math domain error for nonpositive θ and τ. θ' and τ' are automatically learned parameters and can be interpreted as the task weights in multi-task learning (Kendall, Gal, and Cipolla 2018). This way, we avoid manually setting the task weights and the two tasks are automatically balanced in multi-task learning so that CCAG can provide accurate predictions for both tasks. All the parameters of CCAG including θ' and τ' can be updated by gradient descent based methods.

4 Experiments

4.1 Experimental Setup

Data. We choose two benchmark datasets[3] JavaScript (JS) and Python (PY) used in previous studies (Li et al. 2018; Liu et al. 2020). Each dataset contains 150,000 program files and their corresponding ASTs. We use the official train/test split, i.e., 100,000 in the training set and 50,000 in the test set.

[2]Its derivation can be found in the appendix.

[3]https://www.sri.inf.ethz.ch/research/plml

We follow the method of Li et al. (2018) for data preprocessing. We first flatten ASTs into sequences using pre-order depth-first traversal. JS and PY have 95 and 330 types, respectively. We then divide each program into segments consisting of 50 consecutive AST nodes. Since the number of values is too large, we keep K most frequent values in the training data to construct the value vocabulary where K={1,000, 10,000, 50,000}. Values which are not included in the value vocabulary are replaced with `UNK`. For non-leaf nodes, `EMPTY` is used as the value. This way, we have 6 datasets: JS1k, JS10k, JS50k, PY1k, PY10k and PY50k. For each flattened sequence with l AST nodes, we predict the value/type of the r-th ($2 \leq r \leq l$) AST node based on its preceding $r - 1$ nodes (Li et al. 2018; Liu et al. 2020).

Baselines. We compare CCAG with several state-of-the-art code completion methods. It is worth pointing out that some of them are simultaneously proposed in different papers with a slight difference in the design of the prediction layer. We group methods with similar designs as one baseline and describe the prediction layer we use. These baselines include:

- **VanillaLSTM** (Liu et al. 2016; Li et al. 2018; Svyatkovskiy et al. 2019) adopts LSTM to extract AST node features from flattened AST sequences. The inputs to each LSTM cell are previous hidden state and the concatenation of the type and value embeddings of the current AST node. The last output hidden state is fed to a prediction layer consisting of a single-layer feedforward neural network followed by softmax.
- **ParentLSTM** (Li et al. 2018) is an improved version of VanillaLSTM. It computes the attention weights between the current hidden state and all the previous hidden states within a context window to produce a context vector. The input to the prediction layer is the concatenation of the current hidden state, the context vector and the hidden state of the parent node of the current node.
- **PointerMixtureNet** (Bhoopchand et al. 2016; Li et al. 2018) improves ParentLSTM using Pointer Network (Vinyals, Fortunato, and Jaitly 2015). When predicting, it chooses the value/type for the next node from either a predefined global vocabulary or the context window according to the produced probability from the model.
- **Transformer** (Kim et al. 2020; Svyatkovskiy et al. 2020a) improves VanillaLSTM by replacing LSTM with Transformer (Vaswani et al. 2017).
- **Transformer-XL** (Liu et al. 2020) adopts an improved Transformer architecture (Dai et al. 2019) to model the flattened AST sequence and the path from the predicting node to the root in the AST is fed to a BiLSTM to capture the structural information. The naive multi-task learning is used, and task weights are fixed to be equal.

Transformer-XL and CCAG (and its variants) adopt multi-task learning for training. For other methods, value prediction and type prediction are trained independently.

Hyper-parameters. For a fair comparison, we use 128 as the embedding size, hidden size and batch size for all meth-

	JS1k		JS10k		JS50k		PY1k		PY10k		PY50k	
	value	type	value	type	value	type	value	type	value	type	value	type
VanillaLSTM	53.19%	69.52%	58.04%	71.16%	59.70%	72.08%	49.99%	68.08%	52.67%	68.86%	53.66%	69.09%
ParentLSTM	56.45%	71.99%	61.54%	73.46%	63.39%	74.24%	52.57%	70.10%	55.87%	76.25%	56.93%	71.00%
PointerMixtureNet	56.49%	71.95%	62.33%	74.28%	64.14%	76.01%	52.98%	69.98%	56.91%	**76.94%**	57.22%	70.91%
Transformer	58.40%	**73.29%**	**63.93%**	**74.78%**	65.31%	75.89%	53.49%	70.63%	57.52%	71.45%	59.05%	71.91%
Transformer-XL	**59.23%**	72.11%	62.82%	74.09%	**66.41%**	**76.23%**	**55.13%**	**72.45%**	**58.21%**	73.19%	**60.00%**	**72.42%**
CCAG	**62.79% (6.01%)**	**75.72% (3.32%)**	**66.69% (4.32%)**	**78.55% (5.04%)**	**68.19% (2.68%)**	**80.14% (5.13%)**	**61.92% (12.32%)**	**76.71% (5.88%)**	**63.24% (8.64%)**	**80.90% (5.15%)**	**64.22% (7.03%)**	**75.31% (3.99%)**

Table 1: Results on six datasets. Results of CCAG and the best baselines are in bold. The percentages in brackets indicate the improvements of CCAG over the best baselines.

	JS50k		PY50k	
	value	type	value	type
CCAG$_g$	66.20%	75.97%	62.05%	73.22%
CCAG$_n$	67.94%	78.69%	63.15%	74.20%
CCAG$_b$	65.27%	75.10%	59.22%	71.45%
CCAG$_p$	66.73%	76.11%	63.05%	74.00%
CCAG$_r$	66.76%	76.31%	62.35%	73.64%
CCAG$_{ng}$	37.36%	49.66%	30.36%	45.60%
CCAG$_{gs}$	67.44%	78.23%	62.45%	73.79%
CCAG$_{pe}$	67.98%	79.90%	62.01%	73.17%
CCAG	**68.19%**	**80.14%**	**64.22%**	**75.31%**

Table 2: Comparisons among variants of CCAG.

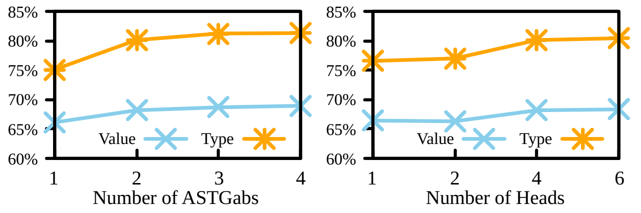


Figure 4: Impacts of hyper-parameters on CCAG for JS50k. Left: Varying numbers of ASTGabs when 4 heads are used. Right: Varying numbers of heads when 2 ASTGabs are used.

ods. All methods are optimized with Adam (Kingma and Ba 2015) using an initial learning rate of 0.001. For VanillaLSTM, ParentLSTM and PointerMixtureNet, the learning rate is multiplied by 0.6 after each epoch, the gradient norm is clipped to 5, and the size of context windows is set to 50 as suggested by Li et al. (2018). For Transformer based methods, we search the settings of heads and layer number in 1-6 and 1-8, respectively. Then, their best results are reported. Following Li et al. (2018), for methods using parent-child relations, the hidden state of one parent node will be replaced with the hidden state of the immediate predecessor to the current node, if the parent node and the current node are not in the same segment. By default, we use 2 ASTGabs and 4 heads in CCAG and its variants, but we also report the impacts of varying these hyper-parameters in Sec.4.2.

Metrics. We use accuracy as the evaluation metric for the code completion task (Li et al. 2018). It shows the proportion of correctly predicted values/types for next AST nodes.

4.2 Experimental Results

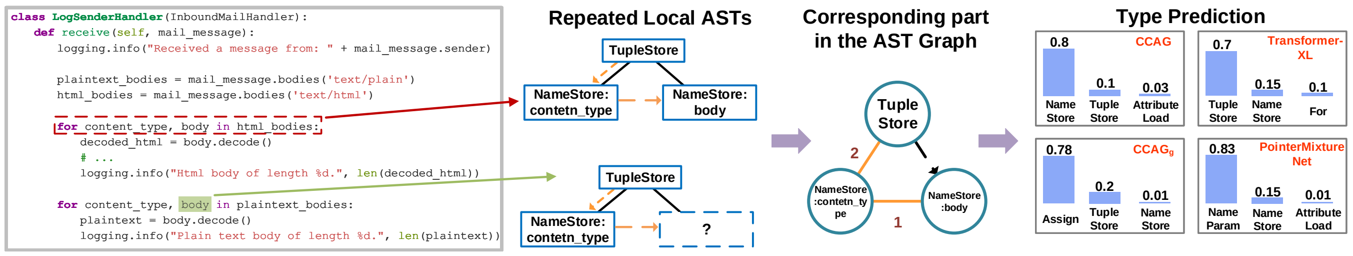
We run each method 3 times and report the average results. We conduct the Wilcoxon signed-rank test to test whether the improvements of CCAG are statistically significant and all the p-values are less than 0.01. We now analyze the results to answer several research questions:

RQ1: How does CCAG perform compared to baselines? Tab. 1 shows all methods' performance of both value prediction and type prediction on 6 datasets. We can see that CCAG consistently surpasses the best baselines, by 2.68%-12.32% in value prediction and 3.32%-5.88% in type prediction. CCAG also outperforms VanillaLSTM, which is used in Visual Studio Code IDE (Svyatkovskiy et al. 2019), by 7.03%-23.86% in value prediction and 8.92%-17.48% in type prediction. This demonstrates CCAG's effectiveness.

RQ2: Does each component of CCAG contribute to the improvement of accuracy? We conduct an ablation study to investigate whether each component of CCAG contributes to its performance. Tab. 2 reports the results of the following variants of CCAG on JS50k and PY50k:

- **CCAG$_g$** models the original partial AST (instead of the flattened AST) as a graph, following the idea of Allamanis, Brockschmidt, and Khademi (2018); Brockschmidt et al. (2019). The representation of each AST node is the concatenation of its type and value embeddings.
- **CCAG$_n$** fixes task weights in multi-task learning.
- **CCAG$_b$** directly models the original partial AST as a graph, fixes task weights in multi-task learning and does not include PCAT and the residual connection.
- **CCAG$_p$**, **CCAG$_r$**, **CCAG$_{ng}$**, **CCAG$_{gs}$** and **CCAG$_{pe}$** do not include PCAT, the residual connection, NGAT, GSAT and the position embeddings, respectively.

Except the above differences, other parts of these variants are the same as CCAG. From Tab. 2, we can see that CCAG$_g$ does not perform as well as CCAG, which confirms the superiority of our design to construct AST graphs based on flattened ASTs instead of original partial ASTs. CCAG$_n$ has worse accuracy than CCAG, showing that using the uncertainty based method to automatically weigh the two tasks in multi-task learning can provide better results. CCAG$_b$ only includes the core parts NGAT and GSAT, of which the idea comes from GNN (e.g., GAT (Velickovic et al. 2018)). CCAG$_b$ has comparable performance to the best baselines in Tab. 1. Nevertheless, it performs worse than other variants with more components, showing that using more sophisticated neural networks like GNN is not the only reason for the superiority of CCAG. All of CCAG$_p$, CCAG$_r$, CCAG$_{ng}$, CCAG$_{gs}$ and CCAG$_{pe}$ perform worse than CCAG, which



```
class LogSenderHandler(InboundMailHandler):
    def receive(self, mail_message):
        logging.info("Received a message from: " + mail_message.sender)

        plaintext_bodies = mail_message.bodies('text/plain')
        html_bodies = mail_message.bodies('text/html')

        for content_type, body in html_bodies:
            decoded_html = body.decode()
            # ...
            logging.info("Html body of length %d.", len(decoded_html))

        for content_type, body in plaintext_bodies:
            plaintext = body.decode()
            logging.info("Plain text body of length %d.", len(plaintext))
```

Figure 5: A code completion example. The type of the next AST node `NameStore:body` is predicted by four methods. The orange lines illustrate the traversal for flattening the AST or the node-node edges. The black lines show parent-child edges.

verifies that all components contribute to the effectiveness of CCAG. CCAG$_{ng}$ shows the worst result among all variants. The reason is that ASTGab is based on GNN which updates node representations using information from the neighborhood. Hence, NGAT which aggregates information from neighbors is the most important component in ASTGab to implement GNN, and ASTGab without NGAT (i.e., CCAG$_{ng}$) does not work at all.

RQ3: What are the impacts when changing the hyper-parameters of CCAG? We conduct a study of the impacts of hyper-parameters on CCAG. We show part of the results on JS50k in Fig. 4. For other settings and datasets, similar trends exist. Specifically, increasing the number of ASTGabs improves accuracy, but using more than 2 ASTGabs will not bring a significant gain. Besides, using more heads instead of a single head brings a noticeable improvement of accuracy. However, employing more than 4 heads does not improve accuracy too much.

Case Study. Fig. 5 shows a program file in GitHub which is used to illustrate how the incoming email is handled in Google Cloud. It is also contained in the test set of PY. `NameStore:body` highlighted with green is the next node to predict. We provide the probabilities of the top-3 types predicted by four methods. CCAG$_g$ gives high probabilities to the types of the first-order neighbors of the next node in the AST graph based on the original partial AST. Transformer-XL mainly gives high probabilities to the types of nodes along the path from the predicting node to the root in the original partial AST, and PointerMixtureNet gives high probabilities to the frequent types. Differently, CCAG successfully captures the repetitive pattern (the repeated local ASTs) via the corresponding part in our AST graph which contains the repetitive pattern in edges and edge weights. Given the key part in our designed AST graph, it is easy for CCAG to predict the next type `NameStore`.

5 Related Work

The pioneering work for code completion (Bruch, Monperrus, and Mezini 2009) adopts Best Matching Neighbor algorithm, which is later improved by Proksch, Lerch, and Mezini (2015) with Bayesian Networks. Hindle et al. (2012) find that most software can be processed by NLP techniques, and they use n-gram language model for code completion.

Recently, deep learning techniques such as RNNs (Raychev, Vechev, and Yahav 2014), GRUs (Aye and Kaiser 2020), LSTMs (Liu et al. 2016; Svyatkovskiy et al. 2019; Yang 2020), Transformer and its variants (Kim et al. 2020; Svyatkovskiy et al. 2020a; Liu et al. 2020), and multi-task learning (Liu et al. 2016, 2020) have been introduced to code completion and significantly improved the accuracy of code completion engines. Besides, some works consider modeling more context information. Bhoopchand et al. (2016); Li et al. (2018) adopt Pointer Network (Vinyals, Fortunato, and Jaitly 2015) and keep the context representations of previous tokens to enhance code completion. Li et al. (2018) consider the parent-child relation in AST to capture the structure of AST. Liu et al. (2020) model the path from the next node to the root in the AST to improve code completion. Svyatkovskiy et al. (2020b) explore different neural architectures (e.g., CNNs) for better context encoding.

In addition to code completion, there are some works studying learning general program representations for different downstream tasks (Allamanis et al. 2018; Le, Chen, and Babar 2020). Some of them extract AST-node-level (Wang, Liu, and Tan 2016) or AST-path-level features (Alon et al. 2018, 2019a,b). Other works model ASTs as trees (Mou et al. 2016; Jayasundara et al. 2019; Zhang et al. 2019) or graphs (Nguyen and Nguyen 2015; LeClair et al. 2020; Allamanis, Brockschmidt, and Khademi 2018; Brockschmidt et al. 2019) and generate program representations.

6 Conclusion

This paper illustrates how to model the flattened AST as an AST graph to provide intelligent code completion. Our proposed CCAG uses ASTGab to capture different dependencies in the AST graph and the sub-tasks of code completion are automatically balanced in optimization using uncertainty. Experimental results on benchmark data have demonstrated the effectiveness of CCAG. In the future, we plan to explore the potential of modeling other representations of code (e.g., data flow graphs) to improve CCAG further.

Acknowledgments

Hui Li was partially supported by the National Natural Science Foundation of China (No. 62002303, 62036004), the Natural Science Foundation of Fujian Province of China (No. 2020J05001) and China Fundamental Research Funds for the Central Universities (No. 20720200089).

Appendix

Derivation of Eq. 12

Existing code completion methods, which use multi-task learning, trains one model with a global loss function (Liu et al. 2016, 2020):

$$\mathcal{L} = w_v \mathcal{L}_v + w_t \mathcal{L}_t, \tag{13}$$

where $\mathcal{L}_v$ and $\mathcal{L}_t$ are defined in Eq. 11 of our submission. w_v and w_t are task weights for value prediction and type prediction, respectively. w_v and w_t are fixed and set to be equal (Liu et al. 2020). However, the two tasks may have different and changing converge speeds during multi-task learning (Chen et al. 2018; Vandenhende et al. 2020). Task imbalances will impede proper training because they manifest as imbalances between backpropagated gradients. However, the cost for manually tuning and updating task weights is unaffordable. To alleviate this issue, we use the idea of *uncertainty* (Kendall, Gal, and Cipolla 2018) and automatically weigh the two tasks during optimization without the need to tune task weights.

To be specific, we adapt the *scaled* version of the value prediction for the next node of the partial AST j:

$$Pr\big(y_j^{(v)} \big| \bar{\mathbf{y}}_j^{(v)}, \theta\big) = softmax\Big(\frac{1}{\theta^2}\bar{\mathbf{y}}_j^{(v)}\Big), \tag{14}$$

where $\bar{\mathbf{y}}_j^{(v)}$ is the unnormalized probability distribution for the value of the next AST node as defined in Eq. 10 of Sec. 3.3. θ is a positive scalar. Eq. 14 can be interpreted as a Boltzmann distribution (also called Gibbs distribution) (Kendall, Gal, and Cipolla 2018). The magnitude of θ indicates how "uniform" the discrete distribution is, which is related to its *uncertainty* as measured in entropy (Kendall, Gal, and Cipolla 2018).

Taking the negative logarithm of the likelihood of next node value being value c, we have:

$$\begin{aligned}&\mathcal{L}_v(y_j^{(v)} = c) = -\log softmax\big(y_j^{(v)} = c, \bar{\mathbf{y}}_j^{(v)}\big)\\ &= -\bar{\mathbf{y}}_j^{(v)}(c) + \log \sum_{k\in\mathcal{V}, k\neq c} \exp\big(\bar{\mathbf{y}}_j^{(v)}(k)\big),\end{aligned} \tag{15}$$

where $\mathcal{V}$ is the vocabulary of AST node value, and $\bar{\mathbf{y}}_j^{(v)}(c)$ is the c-th dimension of $\bar{\mathbf{y}}_j^{(v)}$. Using Eq. 15, the log likelihood for the output being node value c in Eq. 14 can be derived as:

$$\begin{aligned}&\log Pr\big(y_j^{(v)} = c \big| \bar{\mathbf{y}}_j^{(v)}, \theta\big)\\ &= \frac{1}{\theta^2}\bar{\mathbf{y}}_j^{(v)}(c) - \log \sum_{k\in\mathcal{V}, k\neq c} \exp\big(\frac{1}{\theta^2}\bar{\mathbf{y}}_j^{(v)}(k)\big)\\ &= -\frac{1}{\theta^2}\mathcal{L}_v(y_t = c) + \log\Bigg(\sum_{k\in\mathcal{V}, k\neq c} \exp\big(\bar{\mathbf{y}}_j^{(v)}(k)\big)\Bigg)^{\frac{1}{\theta^2}}\\ &\quad - \log \sum_{k\in\mathcal{V}, k\neq c} \exp\big(\frac{1}{\theta^2}\bar{\mathbf{y}}_j^{(v)}(k)\big).\end{aligned} \tag{16}$$

Applying the approximation proposed by Kendall, Gal, and Cipolla (2018):

$$\frac{1}{\theta}\sum_{k\in\mathcal{V}, k\neq c} \exp\Big(\frac{1}{\theta^2}\bar{\mathbf{y}}_j^{(v)}(k)\Big) \approx \Bigg(\sum_{k\in\mathcal{V}, k\neq c}\Big(\exp\big(\bar{\mathbf{y}}_j^{(v)}(k)\big)\Big)\Bigg)^{\frac{1}{\theta^2}}, \tag{17}$$

we have:

$$\begin{aligned}&\log Pr\big(y_j^{(j)} = c \big| \bar{\mathbf{y}}_j^{(v)}, \theta\big)\\ &= -\frac{1}{\theta^2}\mathcal{L}_v(y_j = c) - \log \frac{\sum_{k\in\mathcal{V}, k\neq c}\exp\big(\frac{1}{\theta^2}\bar{\mathbf{y}}_j^{(v)}(k)\big)}{\Big(\sum_{k\in\mathcal{V}, k\neq c}\exp\big(\bar{\mathbf{y}}_j^{(v)}(k)\big)\Big)^{\frac{1}{\theta^2}}}\\ &\approx -\frac{1}{\theta^2}\mathcal{L}_v(y_j^{(v)} = c) - \log\theta.\end{aligned} \tag{18}$$

Similarly, for type prediction, we can get:

$$\log Pr\big(y_j^{(t)} = e \big| \bar{\mathbf{y}}_j^{(t)}, \tau\big) \approx -\frac{1}{\tau^2}\mathcal{L}_t(y_j^{(t)} = e) - \log\tau, \tag{19}$$

where τ is a positive scalar, e is a type, and $\bar{\mathbf{y}}_j^{(t)}$ is the unnormalized probability distribution for the type of the next AST node as defined in Eq. 10 of our submission.

Given Eq. 18 and Eq. 19, the joint loss with ground-truth value and type for the next node being c and e can be derived as:

$$\begin{aligned}&\mathcal{L}(y_j^{(v)} = c,\, y_j^{(t)} = e,\, \theta,\, \tau) =\\ &= -\log\Big(Pr(y_j^{(v)} = c\big|\bar{\mathbf{y}}_j^{(v)}, \theta)\cdot Pr(y_j^{(t)} = e\big|\bar{\mathbf{y}}_j^{(t)}, \tau)\Big)\\ &\approx \frac{1}{\theta^2}\mathcal{L}_v(y_j^{(v)} = c) + \frac{1}{\tau^2}\mathcal{L}_t(y_j^{(t)} = e) + \log\theta + \log\tau.\\ &= \exp(-2\theta')\cdot\mathcal{L}_v(y_j^{(v)} = c) + \exp(-2\tau')\cdot\mathcal{L}_t(y_j^{(t)} = e)\\ &\quad + \theta' + \tau'.\end{aligned} \tag{20}$$

Then, taking all the next nodes into consideration, we have the join loss (i.e., Eq. 12 in Sec. 3.3):

$$\begin{aligned}\mathcal{L} &\approx \frac{1}{\theta^2}\mathcal{L}_v + \frac{1}{\tau^2}\mathcal{L}_t + \log\theta + \log\tau\\ &= \exp(-2\theta')\cdot\mathcal{L}_v + \exp(-2\tau')\cdot\mathcal{L}_t + \theta' + \tau'.\end{aligned} \tag{21}$$

In the last transition of Eqs. 20 and 21, we let $\theta' = \log\theta$ and $\tau' = \log\tau$, and train the model to learn $\theta'$and τ' instead of the unconstrained scalars θ and τ. This is for the numerical stability since $\frac{1}{\theta^2}$ and $\frac{1}{\tau^2}$ may encounter the overflow error for very small θ and τ, and $\log\theta$ and $\log\tau$ will have the math domain error for nonpositive θ and τ.

θ' and τ' are automatically learned parameters and can be interpreted as the task weights in multi-task learning (Kendall, Gal, and Cipolla 2018). They are regularized by the last two terms in Eq. 20 to prevent overfitting. This way, we avoid manually setting the task weights and the two tasks are automatically balanced in multi-task learning so that CCAG can provide accurate predictions for both tasks. All the parameters of CCAG including θ' and τ' can be updated by gradient descent based methods.

References

Allamanis, M.; Barr, E. T.; Devanbu, P. T.; and Sutton, C. A. 2018. A Survey of Machine Learning for Big Code and Naturalness. *ACM Comput. Surv.* 51(4): 81:1–81:37.

Allamanis, M.; Brockschmidt, M.; and Khademi, M. 2018. Learning to Represent Programs with Graphs. In *ICLR*.

Alon, U.; Brody, S.; Levy, O.; and Yahav, E. 2019a. code2seq: Generating Sequences from Structured Representations of Code. In *ICLR*.

Alon, U.; Zilberstein, M.; Levy, O.; and Yahav, E. 2018. A general path-based representation for predicting program properties. In *PLDI*, 404–419.

Alon, U.; Zilberstein, M.; Levy, O.; and Yahav, E. 2019b. code2vec: learning distributed representations of code. *Proc. ACM Program. Lang.* 3(POPL): 40:1–40:29.

Aye, G. A.; and Kaiser, G. E. 2020. Sequence Model Design for Code Completion in the Modern IDE. *Arxiv Preprint* URL https://arxiv.org/abs/2004.05249.

Bhoopchand, A.; Rocktäschel, T.; Barr, E. T.; and Riedel, S. 2016. Learning Python Code Suggestion with a Sparse Pointer Network. *Arxiv Preprint* URL https://arxiv.org/abs/1611.08307.

Brockschmidt, M.; Allamanis, M.; Gaunt, A. L.; and Polozov, O. 2019. Generative Code Modeling with Graphs. In *ICLR*.

Bruch, M.; Monperrus, M.; and Mezini, M. 2009. Learning from examples to improve code completion systems. In *ESEC/FSE*, 213–222.

Chen, Z.; Badrinarayanan, V.; Lee, C.; and Rabinovich, A. 2018. GradNorm: Gradient Normalization for Adaptive Loss Balancing in Deep Multitask Networks. In *ICML*, volume 80, 793–802.

Dai, Z.; Yang, Z.; Yang, Y.; Carbonell, J. G.; Le, Q. V.; and Salakhutdinov, R. 2019. Transformer-XL: Attentive Language Models beyond a Fixed-Length Context. In *ACL*, 2978–2988.

He, K.; Zhang, X.; Ren, S.; and Sun, J. 2016. Deep Residual Learning for Image Recognition. In *CVPR*, 770–778.

Hellendoorn, V. J.; Proksch, S.; Gall, H. C.; and Bacchelli, A. 2019. When code completion fails: a case study on real-world completions. In *ICSE*, 960–970.

Hindle, A.; Barr, E. T.; Su, Z.; Gabel, M.; and Devanbu, P. T. 2012. On the naturalness of software. In *ICSE*, 837–847.

Jayasundara, V.; Bui, N. D. Q.; Jiang, L.; and Lo, D. 2019. TreeCaps: Tree-Structured Capsule Networks for Program Source Code Processing. In *NeurIPS Workshop on ML for Systems*.

Kendall, A.; Gal, Y.; and Cipolla, R. 2018. Multi-Task Learning Using Uncertainty to Weigh Losses for Scene Geometry and Semantics. In *CVPR*, 7482–7491.

Kim, S.; Zhao, J.; Tian, Y.; and Chandra, S. 2020. Code Prediction by Feeding Trees to Transformers. *Arxiv Preprint* URL https://arxiv.org/abs/2003.13848.

Kingma, D. P.; and Ba, J. 2015. Adam: A Method for Stochastic Optimization. In *ICLR*.

Le, T. H. M.; Chen, H.; and Babar, M. A. 2020. Deep Learning for Source Code Modeling and Generation: Models, Applications, and Challenges. *ACM Comput. Surv.* 53(3): 62:1–62:38.

LeClair, A.; Haque, S.; Wu, L.; and McMillan, C. 2020. Improved Code Summarization via a Graph Neural Network. In *ICPC*.

Li, J.; Wang, Y.; Lyu, M. R.; and King, I. 2018. Code Completion with Neural Attention and Pointer Networks. In *IJCAI*, 4159–4165.

Liu, C.; Wang, X.; Shin, R.; Gonzalez, J. E.; and Song, D. 2016. Neural Code Completion. *OpenReview.net* URL https://openreview.net/pdf?id=rJbPBt9lg.

Liu, F.; Li, G.; Wei, B.; Xia, X.; Fu, Z.; and Jin, Z. 2020. A Self-Attentional Neural Architecture for Code Completion with Multi-Task Learning. In *ICPC*, 37–47.

Mou, L.; Li, G.; Zhang, L.; Wang, T.; and Jin, Z. 2016. Convolutional Neural Networks over Tree Structures for Programming Language Processing. In *AAAI*, 1287–1293.

Murphy, G. C.; Kersten, M.; and Findlater, L. 2006. How Are Java Software Developers Using the Eclipse IDE? *IEEE Softw.* 23(4): 76–83.

Nguyen, A. T.; and Nguyen, T. N. 2015. Graph-Based Statistical Language Model for Code. In *ICSE*, volume 1, 858–868.

Proksch, S.; Lerch, J.; and Mezini, M. 2015. Intelligent Code Completion with Bayesian Networks. *ACM Trans. Softw. Eng. Methodol.* 25(1): 3:1–3:31.

Raychev, V.; Vechev, M. T.; and Yahav, E. 2014. Code completion with statistical language models. In *PLDI*, 419–428.

Svyatkovskiy, A.; Deng, S. K.; Fu, S.; and Sundaresan, N. 2020a. IntelliCode Compose: Code Generation Using Transformer. In *ESEC/FSE*, 1433–1443.

Svyatkovskiy, A.; Lee, S.; Hadjitofi, A.; Riechert, M.; Franco, J.; and Allamanis, M. 2020b. Fast and Memory-Efficient Neural Code Completion. *Arxiv Preprint* URL https://arxiv.org/abs/1611.08307.

Svyatkovskiy, A.; Zhao, Y.; Fu, S.; and Sundaresan, N. 2019. Pythia: AI-assisted Code Completion System. In *KDD*, 2727–2735.

Vandenhende, S.; Georgoulis, S.; Proesmans, M.; Dai, D.; and Gool, L. V. 2020. Revisiting Multi-Task Learning in the Deep Learning Era. *Arxiv Preprint* URL https://arxiv.org/abs/2004.13379.

Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A. N.; Kaiser, L.; and Polosukhin, I. 2017. Attention is All you Need. In *NIPS*, 5998–6008.

Velickovic, P.; Cucurull, G.; Casanova, A.; Romero, A.; Liò, P.; and Bengio, Y. 2018. Graph Attention Networks. In *ICLR*.

Vinyals, O.; Fortunato, M.; and Jaitly, N. 2015. Pointer Networks. In *NIPS*, 2692–2700.

Wang, S.; Liu, T.; and Tan, L. 2016. Automatically learning semantic features for defect prediction. In *ICSE*, 297–308.

Wu, Z.; Pan, S.; Chen, F.; Long, G.; Zhang, C.; and Yu, P. S. 2020. A Comprehensive Survey on Graph Neural Networks. *IEEE Trans. Knowl. Data Eng.* .

Yang, Y. 2020. Improving the Robustness to Data Inconsistency between Training and Testing for Code Completion by Hierarchical Language Model. *Arxiv Preprint* URL https://arxiv.org/abs/2003.08080.

Zhang, J.; Wang, X.; Zhang, H.; Sun, H.; Wang, K.; and Liu, X. 2019. A novel neural source code representation based on abstract syntax tree. In *ICSE*, 783–794.