TreeCaps: Tree-Based Capsule Networks for Source Code Processing

How to cite:

For guidance on citations see FAQs.

© 2021 AAAI

https://creativecommons.org/licenses/by-nc-nd/4.0/

Version: Accepted Manuscript

oro.open.ac.uk
TreeCaps: Tree-Based Capsule Networks for Source Code Processing

Nghi D. Q. Bui 1, Yijun Yu 2, Lingxiao Jiang 1

1 School of Information Systems, Singapore Management University {dqnbui,lxjiang}@smu.edu.sg
2 School of Computing & Communications, The Open University, UK, y.yu@open.ac.uk

Abstract

Recently program learning techniques have been proposed to process source code based on syntactical structures (e.g., Abstract Syntax Trees) and/or semantic information (e.g., Dependency Graphs). While graphs may be better at capturing various viewpoints of code semantics than trees, constructing graph inputs from code needs static code semantic analysis that may not be accurate and introduces noise during learning. Although syntax trees are precisely defined according to the language grammar and easier to construct and process than graphs, previous tree-based learning techniques have not been able to learn semantic information from trees to achieve better accuracy than graph-based techniques. We propose a new learning technique, named TreeCaps, by fusing together capsule networks with tree-based convolutional neural networks, to achieve learning accuracy higher than existing graph-based techniques while it is based only on trees. TreeCaps introduces novel variable-to-static routing algorithms into the capsule networks to compensate for the loss of previous routing algorithms. Aside from accuracy, we also find that TreeCaps is the most robust to withstand those semantic-preserving program transformations that change code syntax without modifying the semantics. Evaluated on a large number of Java and C/C++ programs, TreeCaps models outperform prior deep learning models of program source code, in terms of both accuracy and robustness for program comprehension tasks such as code functionality classification and function name prediction. Our implementation is publicly available at: https://github.com/bdqnghi/treecaps.

Introduction

Software developers often spend the majority of their time in navigating existing program code bases to understand the functionality of existing source code before implementing new features or fixing bugs (Xia et al. 2018; Evans Data Corporation 2019; Britton et al. 2012). Learning a model of programs has been found useful for their tasks such as classifying the functionality of programs (Nix and Zhang 2017; Dahl et al. 2013; Pascanu et al. 2015; Rastogi, Chen, and Jiang 2013), predicting bugs (Yang et al. 2015; Li et al. 2017, 2018; Zhou et al. 2019), translating programs (Chen, Liu, and Song 2018; Gu et al. 2017; Bui, Jiang, and Yu 2018; Bui, Yu, and Jiang 2019; Nghi, Yu, and Jiang 2019), etc.

It is common that adding semantic descriptions (e.g., via code comments, visualizing code control flow graphs, etc.) may enhance human understanding of programs and ease machine learning. With the help of static code dependency analysis techniques (Nielson, Nielson, and Hankin 1999), for example, Gated Graph Neural Networks (GGNN) (Li et al. 2016; Fernandes, Allamanis, and Brockschmidt 2019; Allamanis, Brockschmidt, and Khademi 2018) learn code semantics via graphs where edges are added between the code syntax tree nodes to indicate various kinds of dependencies between the nodes. However, adding such edges requires extra processing of ASTs and may introduce noise for different learning tasks since there is no consensus on which types of edges are needed for which tasks.

There also exist deep learning techniques that process code syntax trees or abstract syntax trees (ASTs) (Mou et al. 2016; Alon et al. 2019b; Zhang et al. 2019). However, they are limited in how they represent and learn ASTs although ASTs entail all code semantics. Tree-Based Convolutional Neural Network (TBCNN) (Mou et al. 2016) shares the same computational principle as GGNN, i.e., information is accumulated from nearby children to parent nodes only, which limits the number of iterations for a node to accumulate information from its distant descendants. Code2vec (Alon et al. 2019b) decomposes trees into a bag of path-contexts for learning; ASTNN (Zhang et al. 2019) splits big trees for programs and functions into smaller subtrees for individual statements. They adapt recurrent neural network models to learn the path-contexts or flattened subtrees, but still likely miss code dependency information that is not represented in the decomposed paths and subtrees.

It is desirable to learn code models via ASTs because trees can be more efficiently and precisely constructed from code than graphs without the need of semantic analysis that may be expensive or inaccurate. Towards this goal, this paper proposes a novel architecture called TreeCaps by fusing capsule networks (Sabour, Frosst, and Hinton 2017) with TBCNN to build code models from trees, as a complement to graph-based models. TreeCaps first adapts TBCNN to take in trees and extract (local) node features with its convolution capability and converts the node features into capsules in its Primary Variable Capsule (PVC) layer where the number of capsules can change for different tree inputs. It then adapts CapsNet by introducing two methods to route the dynamic number of capsules in PVC to a static number of capsules in its Secondary Capsule (SC) layer. Our first method inherits the dynamic routing algorithm (Sabour, Frosst, and Hinton 2017) for static numbers of capsules; it shares a global transformation matrix across every pair of capsules between the layers (Yang et al. 2018; Zhang and Chen 2019). Our second method is a novel Variable-to-Static (VTS) routing algorithm that selects the capsules with the most promi-
ent outputs in the PVC layer and squeezes them into a fixed set of capsules. The method utilizes the common intuition that code semantics can often be determined by considering only a portion of code elements. Further, we apply a dynamic routing algorithm from the capsules in the SC layer to the final Code Capsule (CC) layer whose number of capsules is fixed according to a specific learning task, to get the vector embeddings of the trees for the task. Compared to the max-pooling method to combine node features in TBCCN, the pipeline of our routing methods (PVC \to SC \to CC) can learn more complex combinations of AST features.

Across codebases in C/C++ and Java with respect to commonly compared program comprehension tasks such as code functionality classification and function name prediction, our empirical evaluation shows that TreeCaps achieves better classification accuracy and better F1 score in prediction compared to other code learning techniques such as Code2vec, Code2seq, ASTNN, TBCNN, GGNN, GREAT and GNN-FILM. We have also applied three types of semantic-preserving transformations (Zhang et al. 2020; Wang and Su 2019) that transform programs into syntactically different but semantically equivalent code to attack the models. Evaluations also show that our TreeCaps models are the most robust, able to preserve its predictions for transformed programs more than other learning techniques.

Related Work

There has been huge interest in applying deep learning techniques for software engineering tasks such as program functionality classification (Mou et al. 2016; Zhang et al. 2019), function name prediction (Fernandes, Allamanis, and Brockschmidt 2019), bug localization (Pradel and Sen 2018; Gupta, Kanade, and Shevade 2019), code clone detection (Zhang et al. 2019), program refactoring (Hu et al. 2018), program translation (Chen, Liu, and Song 2018), and code synthesis (Brockschmidt et al. 2019). A model of source code can often be learned in two steps: (1) convert source code into suitable intermediate representations, and (2) design learning networks to process the representations.

Mou et al. (2016) parse code into ASTs and design Tree-Based Convolutional Neural Networks (TBCNNs) as the learning networks. Allamanis, Brockschmidt, and Khademi (2018) extend ASTs to graphs by adding a variety of code dependencies as edges among tree nodes, intended to represent code semantics, and apply Gated Graph Neural Networks (GGNN) (Li et al. 2016) to learn the graphs, which indeed enhances the performance of TBCNN (Mou et al. 2016) for certain tasks. GNN-FILM (Brockschmidt 2019) is also a graph-based model that explores by applying feature-wise linear modulation (FiLM) on Graph Neural Network (GNN). Hellendoorn et al. (2019) proposes a hybrid approach to combine sequence-based models (Recurrent Neural Networks, Transformer) and graph-based models (GNNs) into a model called Graph Relational Embedding Attention Transformer (GREAT) to address the major drawback of GGNN that can only capture local information of the source code. While the graph-based model extracts local features of source code, the sequence-based model captures global features, their combination improves the performance of GREAT over GNNs.

Code2vec (Alon et al. 2019b), Code2seq (Alon et al. 2019a), and ASTNN (Zhang et al. 2019) are designed based on splitting ASTs into smaller ones, either as a bag of path-contexts or as flattened subtrees representing individual statements, and use various kinds of Recurrent Neural Network (RNN) to learn such code representations. Inst2vec (Ben-Nun, Jakobovits, and Hoefler 2018) uses the RNN to model the Intermediate Representation of the binary code that is independent of the source programming language.

Capsule networks (CapsNet) (Sabour, Frosst, and Hinton 2017; Hinton, Sabour, and Frosst 2018) use dynamic routing to model spatial and hierarchical relations among objects in an image. The techniques have been successfully applied to image processing tasks such as image classification, character recognition, and text classification (Jayasundara et al. 2019; Rajasegaran et al. 2019; Yang et al. 2018; Li et al. 2019). However, none of the studies has considered complex tree data as input, which is however natural for programs. Capsule Graph Neural Networks (Zhang and Chen 2019) proposed to classify biological and social network graphs does not handle tree- or graph-based code syntax. To the best of our knowledge, we are the first to adopt capsule networks for program source code processing to learn code models on syntax trees directly, without the need for extra static program semantic analysis techniques that may be expensive or introduce inaccuracies (Nielson, Nielson, and Hankin 1999).

Tree-based Capsule Networks

An overview of the TreeCaps architecture is shown in Fig. 1. The code snippet in the training data is parsed into an AST and vectorized. The node vectors are fed into the TBCNN to extract node features to be used as the input for the Primary Variable Capsule (PVC) layer where the number of capsules can change dynamically according to the input tree size. The capsules in the PVC layer are then routed and reduced to a fixed number of capsules in the Secondary Capsule (SC) layer. The outputs of the SC layer are routed to the final Code Capsule (CC) layer where capsules can be seen as the vector representations for the input code, and can be trained with respect to various code comprehension tasks, such as code functionality classification and function name prediction.

Tree-based Convolutional Neural Networks

We briefly introduce the Tree-based Convolutional Neural Networks (TBCNN, (Mou et al. 2016)) for processing tree-structured inputs used in TreeCaps.

A tree $T = (V, E, X)$ consists of a set of nodes $V$, a set of node features $X$, and a set of edges $E$. An edge in a tree connects a node and its children. Each node in an AST also contains its corresponding texts (or tokens) and its type (e.g., operator types, statement types, function types, etc.) from the underlying code. Initially, we annotate each node $v \in V$ with a $D$-dimensional real-valued vector $x_v \in \mathbb{R}^D$ representing the features of the node. We associate every node $v$ with a hidden state vector $h_v$, initialized from the feature embedding $x_v$, which can be computed from a simple concatenation of the embeddings of its texts and type (Allamanis, Brockschmidt, and Khademi 2018). The embedding
matrices for the texts and types can be learned in the whole model training pipeline.

In TBCNN, a convolution window over an AST is emulated via a binary tree, where the weight matrix for each node is a weighted sum of three fixed matrices $W^t$, $W^l$, $W^r \in \mathbb{R}^{D \times D}$ (for the “top”, “left”, and “right” node respectively) and a bias term $b \in \mathbb{R}^D$. Hence, for a convolution window of depth $d$ in the original AST containing $K = 2^d - 1$ nodes (including the parent node) with vectors $[x_1, \ldots, x_K]$, where $x_i \in \mathbb{R}^D$, the convolutional output $y$ of the window is defined as follows:

$$ y = \tanh\left( \sum_{i=1}^K \eta_i^t W^t + \eta_i^l W^l + \eta_i^r W^r |x_i + b\right),$$

where $\eta_i^t$, $\eta_i^l$, $\eta_i^r$ are weights calculated corresponding to the depth and the position of the nodes. A TBCNN model usually stacks $m$ such convolutional layers together to generate the final node embeddings, where the output at layer $m$ will be used as the input for the next, i.e. the $m + 1$-th layer. Each layer has its own $W^t$, $W^l$, $W^r \in \mathbb{R}^{D \times D}$ and the bias term $b \in \mathbb{R}^D$ with different initialization.

The Primary Variable Capsule Layer (PVC)

In the PVC layer, we use $m$ tree-based CNN layers with different random initializations for $W^t$, $W^l$, $W^r$ and $b$. We group outputs of the convolutional layers together to form $N_{pvc} = |V| \times D$ sets of capsules with outputs $c_i \in \mathbb{R}^{D_{pvc}}, i \in [1, N_{pvc})$, where $D_{pvc} = m$ is the dimension of the capsules in the PVC layer. We apply a non-linear squash function (Sabour, Frosst, and Hinton 2017) to a capsule to calculate the final node embeddings, where the output at layer $m$ will be used as the input for the next, i.e. the $m + 1$-th layer. Each layer has its own $W^t$, $W^l$, $W^r \in \mathbb{R}^{D \times D}$ and the bias term $b \in \mathbb{R}^D$ with different initialization.

Sharing Weights across Child Capsules with Dynamic Routing (DRSW)

Since the number of capsules in the PVC is dynamic, a global transformation matrix cannot be defined in practice with variable dimensions. The solution for this problem is to define a shared transformation matrix $W_s \in \mathbb{R}^{N_{pvc} \times D_{pvc} \times D_{sc}}$ across the child capsules, where $N_{pvc}$ is the number of capsules in the PVC layer (Yang et al. 2018), $D_{sc}$ is the dimension of the capsules in the SC layer, and a dynamic algorithm routes the capsules (as summarized in Algo.1).

In Algo.1, for each capsule $i$ in the $l$-th PVC layer and each capsule $j$ in the $l + 1$-th SC layer, we multiply the output of the PVC layer $u_i$ by the shared transformation matrix $W_s$ to produce the prediction vectors $\hat{u}_{ij} = W_su_i$. The “prediction vectors” are responsible for predicting the strength of each capsule in the PVC layer, then a weighted sum over all “prediction vectors” $\hat{u}_{ij}$ will produce the capsule $j$ in the SC layer. The trainable shared transformation matrix learns the part-whole relationships between the primary capsules and secondary capsules, while effectively transforms $u_i$’s into the same dimensionality as $v_j$ where each $v_j$ denotes the capsule output of the SC layer. The coupling coefficients $\beta_{ij}$ between capsule $i$ and all the capsules in the SC layer sum to 1 and are determined by a “routing softmax” whose initial logits $\alpha_{ij}$ are the log prior probabilities that capsule $i$ in PVC layer should be coupled to capsule $j$ in the SC layer. Then we use $r$ iterations to refine $\beta_{ij}$ based on the agreements between the prediction vectors $\hat{u}_{ij}$ and the secondary capsule outputs $v_j$ where $v_j = squash(\sum_i \beta_{ij}\hat{u}_{ij})$.

Algorithm 1 Dynamic Routing

1: procedure ROUTING($\hat{u}_{ij}$, $r$, $l$)
2: Initialize $\forall i \in [1, l], \forall j \in [1, l + 1], \alpha_{ij} \leftarrow 0$
3: for $r$ iterations do
4: $\forall i \in [1, l], \beta_i \leftarrow softmax(\alpha_i)$
5: $\forall j \in [1, l + 1], v_j \leftarrow squash(\sum_i \beta_{ij}\hat{u}_{ij})$
6: $\forall i \in [1, l], \forall j \in [1, l + 1], \alpha_{ij} \leftarrow \alpha_{ij} + \hat{u}_{ij} \cdot v_j$
7: end for
8: return $v_j$
9: end procedure

Variable-to-Static Routing (VTS)

Sharing the transformation matrix reduces the ability to learn different features because each pair of capsules is supposed to have its transformation matrix. Due to this limitation, we offer the second solution to route the variable number of capsules in the
This way, rare capsules, when they have prominent outputs, to-static routing algorithm. It is the capsules with the most the SC layer are determined by iterative runs of the variable-initialization

we only consider the entities with the highest existence prob-

Hence, the outputs of the PVC layer, a

are used to initialize the capsules in SC layer directly.

The major difference between the VTS algo-

of capsules by choosing only the most important capsules in

Non-essential entities, and only a portion of all entities deter-

in practice, not every node of the AST contributes towards

PVC layer. It is based on an observation of source code that,

us to initialize the capsules in SC layer directly.

We initialize the outputs of the SC layer with the outputs of the a capsules with the highest L2 norms in the PVC layer. Hence, the outputs of the PVC layer, \([u_1, \ldots, u_{\alpha_{\text{max}}}]\), are first ordered by their L2 norms to obtain \(U_{\text{sorted}}\), and then the first a vectors of \(U_{\text{sorted}}\) are assigned as \(v_j\), \(j \leq a\).

Since the probability of the existence of an entity is de-

noted by the length of the capsule output vector (L2 norm), we only consider the entities with the highest existence prob-

abilities for initialization (in other words, highest activation) following the aforementioned intuition. It should be noted that the capsules with the a-highest norms are used only for the initialization; the actual outputs of the static capsules in the SC layer are determined by iterative runs of the variable-
to-static routing algorithm. It is the capsules with the most prominent outputs along with the capsules of the highest vector similarities to them that get routed to the next layer. In this way, rare capsules, when they have prominent outputs, are still preserved and routed to the next layer.

Next, we route all b capsules in the PVC layer based on the similarity among them and the static capsule layer outputs. We initialize the routing coefficients as \(\alpha_{ij} = 0\), equally to the b capsules in the PVC layer. Subsequently, they are iteratively refined based on the agreement between the current SC layer outputs \(v_j\) and the PVC layer outputs \(u_i\). The agreement, in this case, is measured by the dot prod-

product, \(f_{ij} \leftarrow u_i \cdot v_j\), and the routing coefficients are adjusted with \(f_{ij}\) accordingly. If a capsule \(u\) in the PVC layer has a strong agreement with a capsule \(j\) in the SC layer, then \(f_{ij}\) will be positively large; whereas if there is strong dis-

agreement, then \(f_{ij}\) will be negatively large. Subsequently, the sum of vectors \(u\), is weighted by the updated \(\beta_{ij}\) to cal-

culate \(s_j\), which is then squashed to update \(v_j\).

The Code Capsules Layer (CC)

The CC layer outputs the vector embeddings for the code \(X_{\text{cc}} \in \mathbb{R}^{N_{\text{cc}} \times D_{\text{cc}}}\), where \(D_{\text{cc}}\) is the dimensionality of each code capsule and \(N_{\text{cc}}\) is fixed with respect to a specific code learning task. Note in the outputs of the SC layer \(X_{\text{sc}} \in \mathbb{R}^{N_{\text{sc}} \times D_{\text{sc}}}\), \(N_{\text{sc}}\) is also fixed,

The following subsections explain how we set \(N_{\text{cc}}\) and train the TreeCaps models for different code learning tasks.

Code (Functionality) Classification This task is to, given a piece of code, classify the functionality class it belongs to. We want \(N_{\text{cc}}\) capsules in the CC layer, each of which corresponds to a functionality class of code that appeared in the training data. As such, we let \(N_{\text{cc}} = \kappa\), where \(\kappa\) is the number of functionality classes. We calculate the probabil-

ity of the existence of each class by obtaining L2 norm of each capsule output vector. We use the margin loss (Sabour, Froston, and Hinton 2017) as the loss function during training.

Function (Method) Name Prediction This task is to, given a piece of code (without its function header), predict a meaningful name that reflects the functionality of the code. For this task, following Alon et al. (2019b)’s prediction approach, we let \(N_{\text{cc}}\) of the CC layer be 1, and the output of the only capsule represent the vector for the given piece of code. In this case, the output capsules of the CC layer has the shape of \(X_{\text{cc}} \in \mathbb{R}^{1 \times D_{\text{cc}}}\), which is also the code vector that represents for the code snippet \(C\), denoted as \(v_C\).

The vector embeddings of the function are learn-able parameters, formally defined as \(\text{functions\_vocab} \in \mathbb{R}^{L \times D_{\text{cc}}}\), where \(L\) is the set of function names found in the training corpus. The embedding of \(\text{functions\_vocab}\) is row \(i\) of \(\text{functions\_vocab}\). The predicted distribution of the model \(q(l)\) is computed as the (softmax-normalized) dot product between the context vector \(v_C\) and each of the function embeddings:

\[
q(l) = \frac{\exp(v_C^T \cdot \text{functions\_vocab}_{l})}{\sum_{l \in L} \exp(v_C^T \cdot \text{functions\_vocab}_{l})}
\]  

where \(q(l)\) is the normalized dot product between the vector of \(l\), and the code vector \(v_C\), i.e., the probability that a function name \(l\) should be assigned to the given code snippet \(C\). We choose \(l\) that gives the maximum probability for the snippet \(v_C\). For training the network, we use cross-entropy as the loss function.

Empirical Evaluation

General Settings. We use fAST, an efficient parser (Yu 2019) to parse code into ASTs in a binary format equiva-

lent to SrcML (Collard, Decker, and Maletic 2013);\(^1\) we also

\(^1\)https://www.srcml.org/, 400+ node types for multiple program-

languages. We chose SrcML because (1) it provides unified AST representations for various languages such as C++/Java, and (2) it has an extension SrcSlice (https://github.com/srcML/srcSlice) to help identify dependencies and construct the graphs needed for GGNN, which is an evaluation baseline.
use another parser PycParser\(^2\) used by TBCNN and ASTNN for a fairer comparison and evaluate the effects of parser choices. For the parameters in our TBCNN layer, we follow Mou et al. (2016) to set the size of type embeddings to 128, the size of text embeddings to 128, and the number of convolutional steps \(m\) to 8. For the capsule layers, we set \(N_c = 100, D_{sc} = 16, D_{cc} = 16\) and routing iterations \(r = 3\). We use Tensorflow libraries to implement TreeCaps. To train the models, we use the Rectified Adam (RAdam) optimizer (Liu et al. 2019) with an initial learning rate of 0.001 subjected to decay on an Nvidia Tesla P100 GPU.

### Baselines
We choose a few recent code modeling techniques to compare with TreeCaps: Code2vec (Alon et al. 2019b), Code2seq (Alon et al. 2019a), TBCNN (Mou et al. 2016), ASTNN (Zhang et al. 2019), GGNN (Allamanis, Brockschmidt, and Khademi 2018), GREAT (Hellendoorn et al. 2019), GNN-FiLM (Brockschmidt 2019). We also include a token-based baseline by treating source code as sequences of tokens and using a neural machine translation (NMT) baseline, which is a 2-layer Bi-LSTM, to process the token sequences. A common setting used among all these techniques is that they all utilize both node type and token information to initialize a node in ASTs. We set both the dimensionality of type embeddings and text embeddings to 128. Note that we try our best to make the baselines as strong as possible by choosing the hyper-parameters above as the “optimal settings” according to their papers or code.\(^3\)

We use different baselines for the two tasks since not all the models were designed for both tasks. For the graph-based models (GGNN, GREAT, GNN-FiLM), there is no publicly available tool to generate the needed graph representations of code by adding semantic edges into the ASTs as presented in (Allamanis, Brockschmidt, and Khademi 2018), so we have implemented a tool by ourselves to represent the code as graphs with the assistance of SrcSlice and SrcML. We include as many edges for these edge types. For Code2vec, we follow the settings suggested in their latest Code2seq paper as well as the implementation in the official software artifacts to reproduce their results.

### Setups for Code Classification
#### Datasets, Metrics, and Models
We use datasets in two different programming languages. The first Sorting Algorithms (SA) dataset is from Ngh, Yu, and Jiang (2019), which contains 10 algorithm classes of 1000 sorting programs written in Java. The second OJ dataset is from Mou et al. (2016), which contains 52000 C programs of 104 classes. We split each dataset into training, testing, and validation sets by the ratios of 70/20/10. We use the same classification accuracy metric as Mou et al. (2016) for comparing classification results.

We compare TreeCaps with other techniques that have been applied to the code classification task, such as TBCNN (Mou et al. 2016), ASTNN (Zhang et al. 2019), Code2vec (Alon et al. 2019b), GGNNs (Allamanis, Brockschmidt, and Khademi 2018; Fernandes, Allamanis, and Brockschmidt 2019). Since TBCNN (Mou et al. 2016) and ASTNN (Zhang et al. 2019) use PycParser to parse code into AST, we also compare TreeCaps with all the baselines by using both PycParser and SrcML. We also include an ablation study to measure the impact of different combinations of node initialization and representation.

#### Code Classification Results
As shown in Table 1, TreeCaps models, especially TreeCaps-VTS, have the highest classification accuracy when combining node type and node token information, for both of the SA and OJ datasets. When node token information is used, the simpler 2-layer Bi-LSTM models may achieve higher accuracy. The OJ dataset also shows that the choice of a parser affects the performance significantly. The models using PycParser all achieve higher accuracy than the models using SrcML. This is due to the reason that ASTs generated by PycParser have only around 50 node types, while SrcML has more than 400 node types, which makes it harder for the networks to learn. Across the datasets, The TreeCaps-VTS performs consistently the best in terms of the F1 measure among the baselines under different settings.

### Setups for Function Name Prediction
#### Datasets, Metrics, and Models
We use the datasets from Code2seq (Alon et al. 2019a) containing three sets of Java programs: Java-Small (700k samples), Java-Med (4M samples), and Java-Large (16M samples). These datasets have been split into training/testing/validation by projects. We measure prediction performance using precision (P), recall (R), and F1 scores over the sub-words in generated names, following the metrics used by Alon et al. (2019b); Fernandes, Allamanis, and Brockschmidt (2019). For example, a predicted name \texttt{result\_compute} is considered to be an exact match of the actual name \texttt{compute\_Result}; predicted \texttt{compute} has full precision but only 50% recall; and predicted \texttt{compute\_model\_result} has full recall but only 67% precision.

We use these baselines for the function name prediction task: Code2vec, TBCNN, Code2seq, GGNN, the 2-layer Bi-LSTM, and two additional ones: GNN-FiLM (Brockschmidt 2019), a graph-based model, and GREAT (Hellendoorn et al. 2019), a hybrid model mixing sequence-based and graph-based techniques. The inputs for these two baselines are graph representations of code, similar to GGNN, and we have adapted these two baselines into the function name prediction task. ASTNN separates ASTs for individual states and is not designed for the function name prediction task, so we exclude it in this evaluation. We train each of the models for 50 epochs for each of the three datasets. We also

---

\(^2\)https://github.com/eliben/pycparser/, 50+ node types for C.

\(^3\)The settings for each of the baselines and parameter analyses can be looked up in the supplementary materials.
Table 1: Performance in Code Functionality Classification compared. A ‘-' means that the model is not suited to use the relevant node representation or the parser and thus not evaluated.

<table>
<thead>
<tr>
<th>Model</th>
<th>SA Dataset (1000 samples)</th>
<th>OJ Dataset (52000 samples)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Parser</td>
<td>SrcML</td>
</tr>
<tr>
<td></td>
<td>Initial Info</td>
<td></td>
</tr>
<tr>
<td>2-layer Bi-LSTM</td>
<td>-</td>
<td>81.83</td>
</tr>
<tr>
<td>Code2vec</td>
<td>-</td>
<td>80.44</td>
</tr>
<tr>
<td>TBCNN</td>
<td>78.09</td>
<td>71.23</td>
</tr>
<tr>
<td>ASTNN</td>
<td>-</td>
<td>84.32</td>
</tr>
<tr>
<td>GGNN</td>
<td>82.12</td>
<td>74.25</td>
</tr>
<tr>
<td>TreeCaps-DRSW</td>
<td>83.15</td>
<td>74.56</td>
</tr>
<tr>
<td>TreeCaps-VTS</td>
<td>84.60</td>
<td>78.15</td>
</tr>
</tbody>
</table>

measure the training time for each of the models.

**Function Name Prediction Results.** As shown in Table 2, TreeCaps-VTS outperforms all other baselines for most of the settings. TreeCaps-DRSW also performs well but still worse than TreeCaps-VTS. Both TreeCaps-VTS and TreeCaps-DRSW are better than the graph-based models (GGNN, GNN-FilM, GREAT) and path-based model (Code2seq, Code2vec) without the need for additional code dependency analysis for constructing graphs. Regarding the training time, GGNN is the longest. The training time of both TreeCaps models is comparable to GREAT, the state-of-the-art graph-based technique to model source code, while TreeCaps-VTS is slightly faster.

**Model Analysis**

To better understand the importance of different components of our approach, we evaluate the effect of various aspects of the TreeCaps models. This subsection provides a robustness analysis and a comparison between DRSW algorithm and VTS algorithm.

**Robustness of Models** We measure the robustness of each model by applying the semantically-preserving program transformations to the Java-large’s test set for the function name prediction task. We follow Wang and Su (2019); Rabin et al. (2020) to transform programs in three ways that change code syntax but preserve code functionality: (1) Variable Renaming (VN), a refactoring transformation that renames a variable in code, where the new name of the variable is taken randomly from a set of variable vocabulary in the training set; (2) Unused Statement (US), inserting an unused string declaration to a randomly selected basic block in the code; and (3) Permute Statement (PS), swapping two independent statements (i.e., with no dependence) in a basic block in the code.

The Java-large test set is thus transformed into a new test set. We then examine if the models make the same predictions for the programs after transformation as the prior predictions for the original programs. We use percentage of predictions changed (PPC) as the metric used by (Zhang et al. 2020; Wang and Su 2019) to measure the robustness of the code models. Formally, suppose $P$ denotes a set of test programs, a semantic-preserving program transformation $T$ that transforms $P$ into a set of transformed programs $P' = \{p' = T(p)| p \in P\}$, and a source code model $M$ that can make predictions for any program $p$: $M(p) = l$, where $l \in L$ denotes a predicted label for $p$ according to a set of labels $L$ learned by $M$, we compute the percentage of predictions changed as:

$$PPC = \frac{\left| \{ p' \in P' | M(p) \neq M(p') \} \right|}{\left| P' \right|} \times 100$$

Lower PPC values for $M$ suggest higher robustness as they can maintain more of correct predictions with respect to the transformation. As shown in Table 3, TreeCaps-VTS is the most robust model against the program transformations. Although more kinds of program transformations could be applied to evaluate model robustness in our future work, the current analysis gives us the confidence that TreeCaps can be more robust against attacks via adversarial examples (Ramakrishnan et al. 2020; Bielik and Vechev 2020).

**Comparison between the Two Routing Algorithms** Figure 2 shows the comparisons between the Dynamic Routing algorithm with Shared Weights (DRSW) and Variable-to-Static Routing algorithm (VTS) for the code classification task on the OJ Dataset. There are two main observations: (1) when DRSW is used, the loss decreases slower than when VTS is used (in the right plot); and (2) VTS improves validation accuracy faster than DRSW (in the left chart). A reason is that DRSW has to learn an additional shared transformation matrix $W_x$, resulting in slower convergence due to a larger number of parameters to be learned.

**Discussion**

**Choice of Node Feature Extractor** For the step to extract the node features, we chose TBCNN because it was designed to process ASTs that usually contain deeper and larger numbers of nodes per code snippet than natural language parse trees per sentence, and it has been shown to outperform TreeLSTM in software engineering tasks such as code classification (Mou et al. 2016) and NLP tasks such as natural language inference (Mou et al. 2015).
Table 2: Performance of TreeCaps and the baselines for Function (Method) Name Prediction

<table>
<thead>
<tr>
<th>Model</th>
<th>java-small (700k Samples)</th>
<th>java-med (4M Samples)</th>
<th>java-large (16M Samples)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P</td>
<td>R</td>
<td>F1</td>
</tr>
<tr>
<td>2-layer Bi-LSTM</td>
<td>40.02</td>
<td>31.84</td>
<td>35.46</td>
</tr>
<tr>
<td>TBCNN</td>
<td>40.89</td>
<td>27.67</td>
<td>32.24</td>
</tr>
<tr>
<td>Code2vec</td>
<td>23.35</td>
<td>22.01</td>
<td>23.16</td>
</tr>
<tr>
<td>Code2seq</td>
<td>50.42</td>
<td>35.43</td>
<td>37.56</td>
</tr>
<tr>
<td>GCNN</td>
<td>40.25</td>
<td>35.25</td>
<td>36.86</td>
</tr>
<tr>
<td>GNN-FiLM</td>
<td>43.12</td>
<td>36.25</td>
<td>39.39</td>
</tr>
<tr>
<td>GREAT</td>
<td>47.25</td>
<td>39.97</td>
<td>43.56</td>
</tr>
<tr>
<td>TreeCaps-DRSW</td>
<td>45.19</td>
<td>39.49</td>
<td>42.89</td>
</tr>
<tr>
<td>TreeCaps-VTS</td>
<td>52.62</td>
<td>41.36</td>
<td>46.78</td>
</tr>
</tbody>
</table>

Figure 2: Comparisons between the Two Routing Algorithms

Figure 2: Comparisons between the Two Routing Algorithms

Relationship with Global Model of Source Code GREAT (Hellendoorn et al. 2019) is a hybrid approach to combine sequence-based and graph-based model to better capture both local and global features of code. TreeCaps shares the same synergy but with a different approach. The feature extraction step is to extract local features of the source code and the routing mechanism of the capsules is to combine the global features of the source code. We have shown in our evaluation that our capsule-based method performs better than GREAT, both in terms of F1 score and training time. Further research is needed to explore how different types of features are captured inside the capsules.

Conclusion

We propose TreeCaps, a novel neural network architecture that incorporates tree-based convolutional neural networks (TBCNN) into capsule networks for better learning of code on abstract syntax trees. To handle dynamic numbers of capsules produced from TBCNN, we propose two methods to route the capsules in the Primary Variable Capsule layer to a fixed number of capsules in the Secondary Capsule layer. We are the first to re-purpose capsule networks over syntax trees to learn code without the need for explicit semantics analysis. Our empirical evaluations have shown that TreeCaps can outperform existing code learning models (e.g., Code2vec, TBCNN, ASTNN, GGNN, GREAT, GNN-FiLM) for two different program comprehension tasks (e.g., code functionality classification and function name prediction) on C/C++/Java programs. It is our belief that the new method can be applied to other software engineering tasks such as bug localization and clone detection.

A limitation of TreeCaps is similar to the original capsule networks and many other neural networks: it still lacks explainability. Software developers may require additional evidence before accepting the prediction results, which suggests future work that relating TreeCaps outputs to certain visible patterns in code could help explain the predictions.

Acknowledgments

This research is supported by the Singapore Ministry of Education (MOE) Academic Research Fund (AcRF) Tier 1 grant and RISE Lab Operational Fund from SIS at SMU, Singapore MOE AcRF Tier 2 Award No. MOE2019-T2-1-193, Royal Society International Collaboration projects (Big Code Forensic Analytics in Secure SE IES/R1/191138, IES/R3/193175), EU H2020 EngageKTN project on Safer Drone Flights (https://droneidentity.eu), EPSRC STRIDE (Socio-technical resilience in software development) project (EP/T017465/1), the Huawei Trustworthy SE and Open-Source (TTE) Lab. We also thank the anonymous reviewers for their insightful comments and suggestions, and thank the authors of related work for sharing data.

References


