On the Embeddings of Variables in Recurrent Neural Networks for Source Code

Nadezhda Chirkova
HSE University[*]
Moscow, Russia

Abstract

Source code processing heavily relies on the methods widely used in natural language processing (NLP), but involves specifics that need to be taken into account to achieve higher quality. An example of this specificity is that the semantics of a variable is defined not only by its name but also by the contexts in which the variable occurs. In this work, we develop dynamic embeddings, a recurrent mechanism that adjusts the learned semantics of the variable when it obtains more information about the variable's role in the program. We show that using the proposed dynamic embeddings significantly improves the performance of the recurrent neural network, in code completion and bug fixing tasks.

1 Introduction

Deep learning is now actively being deployed in source code processing (SCP) for solving such tasks as code completion (Li et al., 2018), generating code comments (Alon et al., 2019), and fixing errors in code (Vasic et al., 2019a). Source code visually looks like a text, motivating the wide use of NLP architectures in SCP. A lot of modern SCP approaches are based on recurrent neural networks (Le et al., 2020), other popular architectures are transformers, and convolutional and graph neural networks.

Utilizing the specifics of source code as a data domain may potentially improve the quality of neural networks for SCP. These specifics include three main aspects. Firstly, the source code is strictly structured, i. e. the source code follows the syntactic rules of the programming language. Secondly, the vocabularies may be large or even potentially unlimited, i. e. a programmer is allowed to define the identifiers of the arbitrary complexity. Thirdly, the identifiers are invariant to renaming, i. e. renaming all the user-defined identifiers does not change the algorithm that the code snippet implements. The first two mentioned specifics have been extensively investigated in the literature. For example, the tree-based architectures, such as TreeLSTM (Chen et al., 2018) or TreeTransformer (Shiv and Quirk, 2019) allow for the utilization of the code's structure. On the other hand, using byte-pair encoding (Karampatsis et al., 2020; Sennrich et al., 2016) or the anonymization of out-of-vocabulary identifiers (Chirkova and Troshin, 2021) deals with the unlimited vocabulary problem. However, the property of source code being invariant to renaming user-defined identifiers has not been paid much attention to. In this work, we aim to close this gap for the recurrent neural networks (RNNs).

Let us take a closer look at the invariance property. In Fig. 1 (a) and (b), a code snippet implementing a simple mathematical calculation is presented with two different variable naming schemes. Both code snippets implement the same algorithm, i. e. are equivalent in the "program semantics" space, but have different text representations. Classic NLP approach implies using the embedding layer as the first layer in the network where learnable embeddings encode the global semantics of the input tokens. In example (a), the embeddings of variables `x` and `y` make sense, as these variables are usually used in mathematical calculations, but in example (b), the embeddings of variables `foo` and `foo2` do not reflect any semantics. Moreover, even in case (a), the semantics of identifier `y` reflected by its embedding are too broad, i. e. this identifier could be used in a lot of different calculations, while variable `y` has a much more specific role in the program, i. e. storing the result of the particular function. The key idea of this work is that *the embedding of a variable in the program should reflect the variable's particular role in this program and not only its name*. The name of the variable may act as the secondary

[*] The work was done while working at Samsung-HSE Laboratory, HSE University.

Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, pages 2679–2689
June 6–11, 2021.

source of information about the variable's role, but the main source of this information is the program itself, i. e. the contexts the variable is used in.

We develop the recurrent mechanism called *dynamic embeddings* that captures the representations of the variables in the program based on the contexts in which these variables are used. Being initialized before a program processing, the dynamic embedding of a variable is updated each time the variable has been used in the program, see the scheme in Fig. 1(g). We test the dynamic embedding approach in two settings: the standard setting with the full data, and the anonymized setting, when all variable names are replaced with unique placeholders `var1`, `var2`, `var3` etc. In the full data setting, we initialize the dynamic embeddings with standard embeddings, see Fig. 1(e), to implement the idea of the variable name being a secondary source of information about the variable's semantics. In the anonymized setting, we initialize the dynamic embeddings using a constant initial embedding, the same for all identifiers, see Fig. 1(f). In this setting, the variable names are not used at all, and the model detects the role of the variable purely based on the contexts in which the variable is used in the program. Although being less practically oriented, the anonymized setting is a conceptually interesting benchmark, as it highlights the capabilities of deep learning architectures to understand the pure program *structure*, that is actually the main goal of SCP, without relying on the *unstructured* textual information contained in variable names. In addition, the anonymized setting could be the case in practice, e. g. when processing the decompiled or obfuscated code (Lacomis et al., 2019).

In the experiments, we show that using the proposed dynamic embeddings significantly outperforms the model that uses the standard embeddings, called static embeddings in our work, in both described settings in two SCP tasks, namely code completion and bug fixing.

To sum up, our contributions are as follows:

- We propose the dynamic embeddings for capturing the semantics of the variable names in source code;
- To demonstrate the wide practical applicability of the proposed dynamic embeddings, we show that they outperform static embeddings in two different code processing tasks, namely code completion (generative task) and bug fixing (discriminative task), in the full data setting;
- We propose the version of the dynamic embeddings approach that does not use variable names at all, and show that it achieves high results in both tasks, sometimes even outperforming the standard model trained on full data (with variable names present in the data).

Our source code is available at `https://github.com/nadiinchi/dynamic_embeddings`.

2 Related Work

The possibility of improving deep learning models of source code by taking into account the invariance property of variable names has been superficially discussed in the literature. Ahmed et al. (2018) replace variables with their types, while Gupta et al. (2017) and Xu et al. (2019) use the static embeddings for anonymized variables. However, the existing SCP work did not consider developing a special architecture that dynamically updates the embeddings of the variables during processing a code snippet.

Our research is also related to the field of processing out-of-vocabulary (OOV) variable names. The commonly used approaches for dealing with OOV variables are using the pointer mechanism (Li et al., 2018) or replacing OOV variables with their types (Hu et al., 2018). As we show in our work, both methods may be successfully combined with the proposed dynamic embeddings.

In the context of NLP, Kobayashi et al. (2017) use a similar model with dynamic embeddings to process OOV and anonymized named entities in natural text. In contrast to their approach, we apply dynamic embeddings to the whole vocabulary of variable names, and incorporate dynamic embeddings into the model that relies on the syntactic structure of code. This results in more meaningful dynamic embeddings.

3 Proposed method

We firstly describe what format of the model input we use, i. e. the procedure of code preprocessing, and then describe our model. At the end of this section, we discuss how we use the proposed model in two code processing tasks.

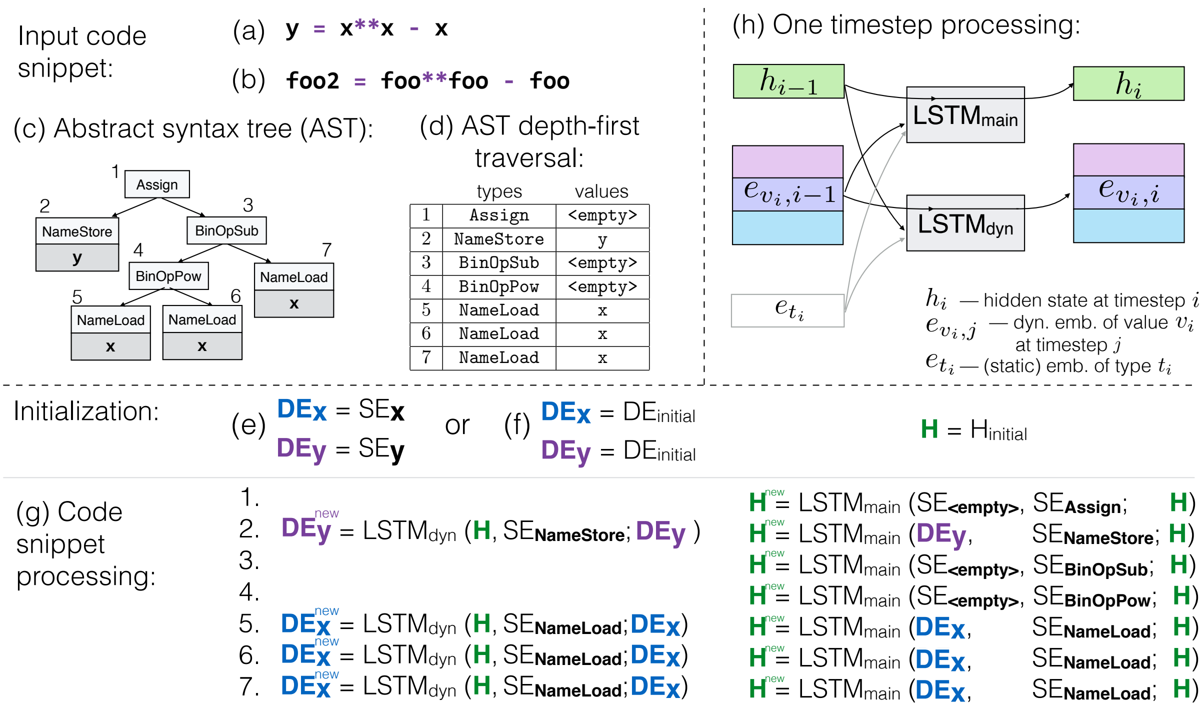
Input code snippet:
(a) y = x**x - x
(b) foo2 = foo**foo - foo
(c) Abstract syntax tree (AST):
1 Assign
2 NameStore y
3 BinOpSub
4 BinOpPow
5 NameLoad x
6 NameLoad x
7 NameLoad x
(d) AST depth-first traversal:
types values
1 Assign <empty>
2 NameStore y
3 BinOpSub <empty>
4 BinOpPow <empty>
5 NameLoad x
6 NameLoad x
7 NameLoad x
(h) One timestep processing:
h_{i-1}
h_i
LSTMmain
LSTMdyn
$e_{v_i,i-1}$
$e_{v_i,i}$
e_{t_i}
h_i — hidden state at timestep i
$e_{v_i,j}$ — dyn. emb. of value v_i at timestep j
e_{t_i} — (static) emb. of type t_i
Initialization:
(e) DEx = SEx DEy = SEy
or
(f) DEx = DEinitial DEy = DEinitial
H = Hinitial
(g) Code snippet processing:
1. H new = LSTMmain (SE<empty>, SEAssign; H)
2. DEy new = LSTMdyn (H, SENameStore; DEy) H new = LSTMmain (DEy, SENameStore; H)
3. H new = LSTMmain (SE<empty>, SEBinOpSub; H)
4. H new = LSTMmain (SE<empty>, SEBinOpPow; H)
5. DEx new = LSTMdyn (H, SENameLoad; DEx) H new = LSTMmain (DEx, SENameLoad; H)
6. DEx new = LSTMdyn (H, SENameLoad; DEx) H new = LSTMmain (DEx, SENameLoad; H)
7. DEx new = LSTMdyn (H, SENameLoad; DEx) H new = LSTMmain (DEx, SENameLoad; H)

Figure 1: The overview of the proposed approach. (a) and (b): two variants of the input code snippet, variant (a) is used in other illustration blocks; (c) abstract syntax tree (AST); (d) AST converted to a sequence that will be passed to the RNN; (e) the static-embedding-based initialization of dynamic embeddings; (f) the constant initialization of dynamic embeddings; (h) the scheme of updating dynamic embeddings and hidden states; (g) the scheme of one timestep processing. SE: static embedding, DE: dynamic embedding.

3.1 Code preprocessing

To capture the syntactic structure of an input code snippet, we convert it to an abstract syntax tree (AST), see Fig. 1(c) for the illustration. In order to process the code snippet with an RNN, we need to convert the AST into a sequence. We use the most popular approach that implies traversing the AST in the depth-first order (Li et al., 2018), see Fig. 1(d). Recent research shows that using the AST traversal may be even more effective than using specific tree-based architectures (Chirkova and Troshin, 2020).

Each node in the AST contains a *type*, reflecting the syntactic unit, e. g. `If` or `NameLoad`. Some nodes also contain *values*, e. g. a user-defined variable or a constant. We insert the `<EMPTY>` value in the nodes that do not have values so that the input snippet is represented as a sequence of (type, value) pairs: $I = [(t_1, v_1), \dots, (t_L, v_L)]$. Here L denotes the length of the sequence, $t_i \in T$ denotes the type and $v_i \in V$ denotes the value. The size of the type vocabulary T is small and determined by the programming language, while the size of the value vocabulary V may be potentially large, as it contains a lot of user-defined identifiers. Given sequence I, the RNN outputs a sequence of hidden states $[h_1, \dots, h_L]$, $h_i \in \mathbb{R}^{d_{\text{hid}}}$, $i = 1, \dots, L$. These hidden states are used to output the task-specific prediction as described in Section 3.3.

3.2 Dynamic embeddings

We use the standard *baseline* recurrent architecture that initializes the hidden state with a learnable predefined vector $h_{\text{init}} \in \mathbb{R}^{h_{\text{hid}}}$: $h_0 = h_{init}$, and then updates the hidden state at each timestep $i = 1, \dots, L$:

$$h_i = \text{LSTM}_{\text{main}}(e_{v_i}, e_{t_i}; h_{i-1}).$$

Here, $e_{v_i} \in \mathbb{R}^{d_{\text{val}}}$ and $e_{t_i} \in \mathbb{R}^{d_{\text{type}}}$ denote the embeddings of the value and the type correspondingly. Without loss of generality, we use the Long Short-Term Memory recurrent unit (LSTM) (Hochreiter and Schmidhuber, 1997). In this work, we replace value embeddings e_{v_i} with *dynamic* embeddings described below.

Dynamic embeddings. The general idea of dynamic embeddings is that the variable's embedding is updated in the RNN-like manner after each occurrence of the variable. We first describe the updating procedure and then discuss the initialization strategy. Since the dynamic embeddings change over timesteps, we use notation $e_{v,i}$ for the dynamic embeddings of value v at timestep i. For example, for the value located at the i-th position in the input sequence, v_i, the dynamic embedding after processing the i-th step is denoted as $e_{v_i,i}$, and its previous state is denoted as $e_{v_i,i-1}$. At each timestep $i = 1, \dots, L$, we update the dynamic embedding $e_{v_i,i}$ of the current value v_i and hidden state h_i using two LSTMs:

$$e_{v_i,i} = \mathrm{LSTM}_{\mathrm{dyn}}(h_{i-1}, e_{t_i}; e_{v_i,i-1}) \qquad (1)$$

$$e_{v,i} = e_{v,i-1}, \quad v \neq v_i \qquad (2)$$

$$h_i = \mathrm{LSTM}_{\mathrm{main}}(e_{v_i,i-1}, e_{t_i}; h_{i-1}) \qquad (3)$$

An illustration of this update procedure is given in Fig. 1(h), and the example scheme of processing a code snippet is given in Fig. 1(g). $\mathrm{LSTM}_{\mathrm{main}}$ implements the recurrence over the hidden state, while $\mathrm{LSTM}_{\mathrm{dyn}}$ implements the recurrence over dynamic embeddings, and the same $\mathrm{LSTM}_{\mathrm{dyn}}$ is used to update the dynamic embeddings of different values at different timesteps. We note that at timestep i, the dynamic embedding of only current value v_i is updated, while the dynamic embeddings of other values do not change, as stated in Eq. (2).

In practice, several dummy values, e. g. `<EMPTY>`, `<UNK>` and `<EOF>`, do not change their roles in different sequences. We use static embeddings for these values.

Initializing dynamic embeddings. The most reasonable strategy for initializing the dynamic embeddings is to use static embeddings: $e_{v,0} = e_v$ where e_v are the learnable embedding vectors of all values v in vocabulary V. In this case, the model utilizes all the available information about the variable: the variable's name introduced by the programmer that is supposed to somehow reflect the mission of the variable, and the contexts in which the variable occurs (captured by hidden states). In other words, the model firstly "understands" the loose role of the variable from its name and then "finetunes" this understanding, while learning more about what the variable is used for.

Another possible strategy is to ignore all the variable names and initialize all dynamic embeddings with a constant embedding: $e_{v,0} = e_{\mathrm{init}}$, $e_{\mathrm{init}} \in \mathbb{R}^{d_{\mathrm{val}}}$, $v \in V$. Although the initial embeddings of all values are the same, they will be updated differently, as different values occur in different locations in the program, and the dynamic embeddings will characterize these locations. Interestingly, the described strategy ensures that if we rename all the variables in the program, the output of the RNN does not change. Such a behaviour is consistent with the variable invariance property: renaming all user-defined variables does not change the underlying algorithm. The common sense is that the architecture for processing source code should be consistent with the variable invariance property, and dynamic embeddings with the constant initial embedding fulfill this conceptual requirement. On the other hand, commonly-used static embeddings are not consistent with the invariance property, i. e. renaming variables scheme results in using different embeddings and changes the predictions of the RNN.

As will be shown below, in practice, using both sources of information, namely variable names and variable occurrences in the program, performs better than relying on only one source of information. In other words, dynamic embeddings with static embedding initialization outperform both static embeddings in the full data setting (relying only on variable names) and dynamic embeddings with constant initialization (relying only on variable occurrences).

3.3 Task-specific prediction

We test the proposed dynamic embeddings in two SCP tasks: code completion and variable misuse prediction. Below, we describe how we make predictions in these tasks, using the output $[h_1, \dots, h_L]$ of the RNN.

Code completion. In code completion, the task is to predict the next type-value pair (t_{i+1}, v_{i+1}) given prefix $[(t_1, v_1), \dots, (t_1, v_i)]$ at each timestep $i = 1, \dots, L$. In our work, we focus on value prediction, as type prediction is a simple task usually solved with high quality in practice (Li et al., 2018). We rely on the setup and the architecture of Li et al. (2018).

To predict the next value v_{i+1} based on $[h_1, \dots, h_i]$, we firstly apply the standard attention mechanism (Bahdanau et al., 2015), obtaining the context vector $c_i = \sum_{j=1}^{i} \alpha_j h_j$, α_j denote attention weights, and then combine all available representations using a fully-connected layer:

$$\hat{h}_i = W^1 h_i + W^2 c_i + W^3 h_{\text{parent}},$$

where h_{parent} is the hidden state of the parent node. For computing the logit $y_{v,i} \in \mathbb{R}$ of each value v, we reuse the dynamic embeddings $e_{v,i}$ of the input layer, as well as the static embeddings of several dummy values: $y_{v,i} = e_{v,i}^T \hat{h}_i$, and apply Softmax on top of $y_{v,i}$ to predict the probability distribution $P_i^{vals} \in \mathbb{R}^{|V|}$ over next value v. Finally, we use the pointer mechanism to improve the prediction of rare values. We reuse attention scores $[\alpha_1, \dots, \alpha_i]$, $\sum_{j=1}^{i} \alpha_j = 1$, $\alpha_j \geqslant 0$ as a distribution over previous positions $P_i^{pos} \in \mathbb{R}^i$, and use switcher $s = \sigma(w^{swit,1} h_i + w^{swit,2} c_i) \in (0, 1)$ to gather two distributions into one: $R_i = [sP_i^{vals}, (1-s)P_i^{pos}]$. To make the prediction, we select the largest element of vector R_i; if it corresponds to the value from the vocabulary, we output this value, if it corresponds to the position, we copy the value from that position. To train the model, we optimize the cross-entropy loss, using as ground truth the values in the vocabulary for in-vocabulary values and the last occurrence of the value (if any) for out-of-vocabulary values.

Variable misuse prediction. The variable misuse task implies outputting two pointers: the first one points to the location i in which the wrong value v_i is used and the second one points to the location j that can be used to repair the bug by copying its value v_j. If there is no bug, the first pointer selects a special no-bug location. In this task, we rely on the approach of Vasic et al. (2019b) and its implementation of Hellendoorn et al. (2020). In addition, we change the format of the model input and use the depth-first AST traversal (Li et al., 2018).

We use the bidirectional LSTM, with each of the two LSTMs being equipped with its own dynamic embeddings. As a result, we have two sequences of hidden states: $[h_1^{\text{fw}}, \dots, h_L^{\text{fw}}]$ and $[h_1^{\text{bw}}, \dots, h_L^{\text{bw}}]$. To make the prediction, we firstly combine two representations using a fully-connected layer:

$$h_i = tanh(W^1 h_i^{\text{fw}} + W^2 h_i^{\text{bw}}), \;\; i = 1, \dots, L$$

and then use two other fully-connected layers to obtain logits $y_i^{\text{bug}} \in \mathbb{R}$ and $y_i^{\text{fix}} \in \mathbb{R}$ of each position i: $y_i^{\text{bug}} = (w^{\text{bug}})^T h_i$, $y_i^{\text{fix}} = (w^{\text{fix}})^T h_i$. Finally, we apply Softmax over $[y_1^{\text{bug}}, \dots, y_L^{\text{bug}}, y^{\text{nobug}}]$ and over $[y_i^{\text{fix}}]_{i=1}^{L}$ to obtain two distributions over positions. Here, learnable $y^{\text{nobug}} \in \mathbb{R}$ corresponds to a no-bug position. The model is trained using the cross-entropy loss.

4 Experimental setup

Data and preprocessing. We conduct experiments on Python150k (Raychev et al., 2016a) and JavaScript150k (Raychev et al., 2016b) datasets. Both datasets are commonly used in SCP and were obtained by downloading repositories from GitHub. However, the train / test split released by the authors of the dataset does not follow the best practices of splitting data in SCP (Allamanis, 2019; LeClair and McMillan, 2019), so we use another train / test split released by Chirkova and Troshin (2020). This split is based on the repositories, i. e. all files from one repository go either to train or test, and was deduplicated using the tools provided by Allamanis (2019), i. e. code files in the test set that are duplicated in the train set were removed; this is a common case in source code downloaded from GitHub. In addition, the Python dataset includes only redistributable code (Kanade et al., 2020). Splitting by repository and deduplication are highly important in SCP to avoid a percentage of testing accuracy being provided by the examples the model saw during training. With the described new split, the results in our tables are not directly comparable to the results reported in other works. To validate our implementation, we compared the quality of baseline models trained in our implementation with the quality reported in the papers describing these baselines, and observed that the numbers are close to each other (see details in Section 5.4).

For the code completion task, we use the entire code files as training objects, filtering out exceptionally long files, i. e. files longer than $3 \cdot 10^4$ characters. The resulting training / testing set consists of 76K / 39K files for Python and of 69K / 41K for JavaScript. The mean length of the code files in 567 / 669 AST nodes for Python / JavaScript.

For the variable misuse task, we select all top-level functions, including functions inside classes from all files, and filter out functions longer than 250 AST nodes, and functions with fewer than three positions containing user-defined variables or less than three distinct user-defined variables. The resulting training / testing set consists of 417K / 231K functions for Python and 202K / 108K functions for JavaScript. One function may occur in the dataset up to 6 times: 3 times with a synthetically generated bug and 3 times without bug. The buggy examples are generated synthetically by choosing random bug and positions from positions containing user-defined variables. The described strategy for injecting synthetic bugs is the same as in (Hellendoorn et al., 2020).

In both tasks, the size of the node type vocabulary is 330 / 44 for Python / JavaScript, the vocabulary of node values is limited to 50K of the most frequent values.

Metrics. Following Li et al. (2018), we use accuracy to measure model quality in the code completion task, counting all predictions of `<UNK>` as wrong. Following Hellendoorn et al. (2020), to measure the quality in the variable misuse task, we use the joint localization and repair accuracy (what portion of buggy values is correctly located and fixed).

Details. In all our models, node type embeddings have 300 units, node value embeddings have 1200 units (for static embeddings), and the one-layer LSTM's hidden state has 1500 units. The described model size matches the configuration of the model of Li et al. (2018). The proposed dynamic embeddings of values have 500 units in all experiments to show that they outperform the static embeddings with much less dimension. In the code completion task, we split the input AST traversals into the chunks, each chunk has the length of 50 AST nodes, and apply attention and pointer only over the last 50 positions. In the variable misuse task, we pass the entire function's AST traversal to the model. In code completion / variable misuse tasks, we train all models for 10 epochs with AdamW (Loshchilov and Hutter, 2019) / Adam (Kingma and Ba, 2015) with an initial learning rate of 0.001 / 0.0001, a learning rate decay of 0.6 after each epoch, a batch size of 128 / 32, and using weight decay of 0.01 / 0. We also use early stopping for the variable misuse task. For code completion, all hyperparameters are the same as in (Li et al., 2018). We tuned hyperparameters to achieve convergence on the training set, for variable misuse. We use the same hyperparameters for static and dynamic embedding models. Both used datasets are large, which helps to avoid overfitting, thus the regularization is not much needed.

		Full-data	Anonymized
1	Assign	<empty>	<empty>
2	NameStore	y	<var3>
3	BinOpSub	<empty>	<empty>
4	BinOpPow	<empty>	<empty>
5	NameLoad	x	<var9>
6	NameLoad	x	<var9>
7	NameLoad	x	<var9>

Figure 2: The visualisation of the model input in two settings considered in the paper: full data setting and anonymized setting, for the example code snippet from Figure 1(a). The leftmost column represents types (the same for both settings), two other columns visualize values for two settings.

5 Experiments

5.1 Anonymized setting

We firstly test the proposed dynamic embeddings in the setting without using the user-defined variable names, stored in node values. Directly omitting values results in losing much information, this can be seen as replacing *all* the variables in a code snippet with the *same* variable `var`. To save the information about whether two AST nodes store the same value or not, we *anonymize* values, i. e. we map the set of all node values in the program (except dummy values, e. g. `<EMPTY>`) to the random subset of anonymized values `var1...varK`, K is a size of the anonymized value vocabulary, we use $K = 1000$. For example, code snippet `sum = sum + lst[i]` may be transformed into `var3 = var3 + var8[var1]`, and `stat = [sum / n; sum]` — into `var1 = [var5 / var2; var5]`. All occurrences of the same value in the program, e. g. `sum`, are replaced with one anonymized value, but value `sum` may be replaced with different anonymized values in different programs. Fig. 2 visualizes how the anonymization is applied to AST. Although being not so practically oriented, the anonymized setting highlights the capabilities of the deep learning models to capture *pure* syntactic information from the AST, without relying on the unstructured text information laid in variable names. In our opinion, this setting should become a must for the future testing of syntax-based SCP models, and the proposed dynamic embeddings could be used as a first layer in such models to capture an equal-not-equal relationship between values.

Model	PY			JS		
	LSTM	LSTM+at	LSTM+pt	LSTM	LSTM+at	LSTM+pt
Stat. emb. (an. data)	55.76	59.74	60.28	51.80	56.26	57.67
Dyn. emb. (an. data)	**66.35**	**66.79**	**66.90**	**61.69**	**62.86**	**62.85**
Stat. emb. (full data)	61.62	63.73	64.69	62.03	64.28	65.05

Table 1: *Anonymized* setting, code completion task, accuracy (%) of the proposed dynamic embedding model and the baseline static embedding model on Python150k (Py) and JavaScript150k (JS) datasets. All standard deviations over three runs are less than 0.05%. The last row represents the conventionally used model trained on the full data (Li et al., 2018) (this model uses more information during training than the models in the first three rows). Columns list the three variants of the base architecture: LSTM, attentional LSTM (LSTM+at), and attentional LSTM with pointer (LSTM+pt).

Model	PY	JS
Stat. emb. (an. data)	25.17	13.16
Dyn. emb. (an. data)	**63.64**	**53.53**
Stat. emb. (full data)	54.78	35.06

Table 2: *Anonymized* setting, variable misuse task, joint accuracy (%) of the proposed dynamic embedding model and the baseline static embedding model on Python150k (Py) and JavaScript150k (JS) datasets. All standard deviations over three runs are less than 0.1%. The last row represents the conventionally used model trained on the full data (Vasic et al., 2019b) (this model uses more information during training than the models in the first three rows).

In the described *anonymized* setting, we compare the proposed dynamic embeddings (constant initialization) with the static embeddings, i. e. learning the static embeddings of `var1..varK`.

Results for the code completion task. In the code completion task, we consider three variants of the architecture: plain LSTM, and attentional LSTM with and without pointer. We note that our goal is to compare the dynamic embeddings with the baseline in three setups, i. e. using three base architectures. We do not pursue the goal of comparing base architectures. Table 1 lists the results.

For all base architectures, the proposed dynamic embeddings outperform static embeddings by a large margin. We note that the number of parameters in both architectures is approximately the same. In the first two setups, with plain and attentional LSTMs, the models can only predict values by generating them from the vocabulary (no pointer), relying on the input and output embeddings of the values. In these setups, the difference between static and dynamic embeddings is large, indicating that dynamic embeddings capture the semantics of the variables significantly better. In the setup with pointer LSTM as a base architecture, the static embeddings win back some percent of correct predictions by relying on the pointer mechanism. Still, the gap between them and dynamic embeddings is large. The portion of correct predictions made using the pointer is 25% for static embeddings and only 0.01% for dynamic embeddings. This shows that dynamic embeddings actually replace the pointer mechanism, performing better. This also explains why the difference in quality of dynamic embeddings between attentional LSTM and pointer LSTM is very small.

Model (full data)	Code compl.		Var. misuse	
	PY	JS	PY	JS
Stat. emb.	64.69	65.05	54.78	35.06
Dyn. emb.	**68.61**	**65.67**	**68.59**	**53.74**

Table 3: *Full data* setting, two tasks, Python150k (Py) and JavaScript150k (JS) datasets. Accuracy (%) of LSTM with pointer (code compeltion), joint accuracy (%) of BiLSTM (variable misuse). All standard deviations are less than 0.05% for code completion and 0.1% for the variable misuse task. Comparing the conventionally used model (static embeddings) and the proposed dynamic embeddings (static initialization). The conventionally used model is a model of Li et al. (2018) for the code completion task and of Vasic et al. (2019b) for the variable misuse task. Note that we use custom data split, see details in Sec. 4.

Interestingly, on the Python dataset, the model with dynamic embeddings trained in the anonymized setting outperforms the conventionally used static embedding model trained in the full data setting, although the first model uses much less information during training. The explanation is that the first model predicts rare values much better than the second model: the accuracy of rare[1] val-

[1] By rare values, we mean values outside top-1000 frequent values.

ues prediction is 27% for the first model and 11% for the second, for the pointer LSTM model. On the contrary, frequent values are easier to predict with static embeddings: the accuracy of predicting frequent values is 53% for the first model and 57% for the second model. The total frequencies of rare and frequent values are approximately the same and equal to 25% (the rest 50% are `EMPTY` values, they are predicted with similar quality with both models). As a result, when counting accuracy over all values, the first model outperforms the second one.

However, on the JavaScript dataset, the first model does not outperform the second one. We analysed the example predictions of both models on both datasets and found that in JavaScript, there are a lot of short code snippets commonly used in different projects. This is expected since JavaScript is mostly used for one purpose, web development, while Python is used for a lot of different purposes. As a result, for JavaScript, the total frequency of top-1000 values is 32% (higher than for Python), while the total frequency of rare values is 22% (less than for Python). The commonly used code snippets are easy to predict in the full data setting but hard to predict in the anonymized setting: the accuracy of predicting frequent values is only 44% for the first model and 54% for the second model. The rare values are still better predicted with dynamic embeddings, but with the gap smaller than for Python: the accuracy of rare values prediction is 23% for the first model and 17% for the second one. The gap is smaller since rare values also occur in the commonly used code snippets which improves the performance of the second model on rare values. When counting accuracy over all values, the second model outperforms the first one.

Results for the variable misuse task. Table 2 lists the joint accuracies of the proposed model and the baseline in the anonymized setting. Again the dynamic embeddings outperform static embeddings by a large margin. Moreover, the dynamic embeddings outperform even the commonly used static embedding model trained on the full data, for both datasets. We think the reason is that we use the dynamic embeddings in two layers of bi-directional LSTMs and these bi-directional dynamic embeddings provide a rich representation of the input code snippet.

5.2 Full data setting

We now test the proposed dynamic embeddings in the full data setting, i. e. we compare a commonly used model with static embeddings and the proposed model with dynamic embeddings (static embedding initialization). The initialization of dynamic embeddings was discussed in Sec. 3.2. Both models process the full data (see illustration in Fig. 2.

The results for both tasks are presented in Table 3 and show that the dynamic embeddings outperform the static embedding model in all cases. We note that dynamic embeddings could be easily incorporated into any recurrent SCP architecture. In our experiments we incorporate them into the base models of Li et al. (2018) and Vasic et al. (2019b) and show that the dynamic embeddings significantly improve these base models. We also note that we use the dynamic embeddings of 500 units while static embeddings have 1200 units. The number of parameters in the dynamic embedding layer, 2.6M, is much smaller than that of the main LSTM layer, 13.8M, and two orders smaller than the number of parameters in the embedding layer, 134M (the numbers are given for the code completion task).

5.3 Example predictions

Figure 3 visualizes the predictions of different models for three example code snippets in Python. We highlighted three scenarios when the dynamic embedding model outperforms the static embedding model in the full data setting: 1) capturing the specific role of the variable, e. g. variable `qual` indexes sequence in the list comprehension in the left example; 2) associating variables with each other, e. g. in the central example, variable `name` always goes with `0`, and variable `post` always goes with `1`; 3) repeating variables when they occur in the similar context they have already been used, e. g. `zeros` in the right example. In all these examples, the proposed dynamic model makes correct predictions, while the static embedding model makes mistakes, in the full data setting. In the anonymized setting, all models tend to predict previously used variables, and again the dynamic embedding model captures the described relationships, and the static embedding model tends to simply predict the most frequent previously used variable.

```
total = sum([qual for ? in \
             quals if qual > 0])
```

```
lengths = [[], []]
contents = [[], []]
for name, post in data:
    lengths[0].append(len(name))
    lengths[1].append(len(post))
    contents[0].append(name)
    contents[1].append(?
```

```
import torch
mask = torch.zeros(len(tokens))
ids = torch.?(len(tokens))
```

	Ground truth: qual	**Ground truth**: post	**Ground truth**: zeros
Full data	**Static emb.**: <empty> **Dynamic emb.**: qual	**Static emb.**: <empty> **Dynamic emb.**: post	**Static emb.**: resize **Dynamic emb.**: zeros
Anonymized data	**Static emb.**: qual **Dynamic emb.**: qual	**Static emb.**: name **Dynamic emb.**: post	**Static emb.**: torch **Dynamic emb.**: zeros

Figure 3: Example predictions for code completion task on Python language. Row 1: ground truth; rows 2 and 3: model trained in the full data setting; rows 4 and 5: models trained in the anonymized setting (these models observe data in a different way, see Fig. 2). The model predicts one next token based on the prefix and does not see gray-colored code.

5.4 Validating the implementation

In our experiments, we use the setup of Li et al. (2018) in the code completion task and of Hellendoorn et al. (2020) in the variable misuse task, but with our custom data split, see details in Section 4. To maintain the possibility of comparing our results to these works, we trained the static embedding models in the full data setting, with the commonly used train / test splits of Python150k and JavaScript150k datasets. For code completion with vocabulary size 50K and pointer network, using exactly the same setup as in (Li et al., 2018), we achieved accuracy of 69.39% / 80.92%, while the paper reports 71% / 81.0% for Python / JavaScript: the results are close to each other. In the variable misuse task, we achieved joint accuracy of 50.2% while Hellendoorn et al. (2020) report 44.4% (Python, JavaScript was not reported in the paper). Our result is higher, since we use 1500 hidden units while Hellendoorn et al. (2020) uses 256 hidden units. In addition, we use different preprocessing and different synthetically generated bugs.

6 Conclusion

In this work, we presented dynamic embeddings, a new approach for capturing the semantics of the variables in code processing tasks. The proposed approach could be used in any recurrent architecture. We incorporated dynamic embeddings in the RNN-based models in two tasks, namely code completion and variable misuse detection, and showed that using the proposed dynamic embeddings improves quality in both full data setting and the anonymized setting, when all user-defined identifiers are removed from the data.

Acknowledgments

I would like to thank Sergey Troshin, Irina Saparina, and the anonymous reviewers for the valuable feedback. The results for the anonymized setting presented in section 5.1 were supported by Samsung Research, Samsung Electronics. The results for the full data setting presented in section 5.2 were supported by the Russian Science Foundation grant №19-71-30020. The research was supported in part through the computational resources of HPC facilities at NRU HSE.

References

Umair Z. Ahmed, Pawan Kumar, Amey Karkare, Purushottam Kar, and Sumit Gulwani. 2018. Compilation error repair: For the student programs, from the student programs. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering Education and Training*, ICSE-SEET '18, page 78–87, New York, NY, USA. Association for Computing Machinery.

Miltiadis Allamanis. 2019. The adverse effects of code duplication in machine learning models of code. *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*.

Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019. code2seq: Generating sequences from structured representations of code. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net.

Dzmitry Bahdanau, Kyung Hyun Cho, and Yoshua Bengio. 2015. Neural machine translation by jointly learning to align and translate. 3rd International Conference on Learning Representations, ICLR 2015 ; Conference date: 07-05-2015 Through 09-05-2015.

Xinyun Chen, Chang Liu, and Dawn Song. 2018. Tree-to-tree neural networks for program translation. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 2552–2562, Red Hook, NY, USA. Curran Associates Inc.

Nadezhda Chirkova and Sergey Troshin. 2020. Empirical study of transformers for source code. *In CoRR*.

Nadezhda Chirkova and Sergey Troshin. 2021. A simple approach for handling out-of-vocabulary identifiers in deep learning for source code. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics*. Association for Computational Linguistics.

Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. Deepfix: Fixing common c language errors by deep learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, AAAI'17, page 1345–1351. AAAI Press.

Vincent J. Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. 2020. Global relational models of source code. In *International Conference on Learning Representations, ICLR 2020*.

Sepp Hochreiter and Jurgen Schmidhuber. 1997. Long short-term memory. *Neural Comput.*, 9(8):1735–1780.

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

Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Learning and evaluating contextual embedding of source code. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pages 5110–5121. PMLR.

Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, C. Sutton, and A. Janes. 2020. Big code != big vocabulary: open-vocabulary models for source code. *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*.

Diederik P. Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

Sosuke Kobayashi, Naoaki Okazaki, and Kentaro Inui. 2017. A neural language model for dynamically representing the meanings of unknown words and entities in a discourse. In *Proceedings of the Eighth International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 473–483, Taipei, Taiwan. Asian Federation of Natural Language Processing.

Jeremy Lacomis, Pengcheng Yin, Edward J. Schwartz, Miltiadis Allamanis, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. 2019. DIRE: A neural approach to decompiled identifier naming. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*, pages 628–639. IEEE.

Triet Le, Hao Chen, and Muhammad Ali Babar. 2020. Deep learning for source code modeling and generation: Models, applications and challenges. *Computing Research Repository*, arXiv:2002.05442.

Alexander LeClair and Collin McMillan. 2019. Recommendations for datasets for source code summarization. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 3931–3937, Minneapolis, Minnesota. Association for Computational Linguistics.

Jian Li, Yue Wang, Michael R. Lyu, and Irwin King. 2018. Code completion with neural attention and pointer networks. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, IJCAI'18, page 4159–25. AAAI Press.

Ilya Loshchilov and Frank Hutter. 2019. Decoupled weight decay regularization. In *International Conference on Learning Representations*.

Veselin Raychev, Pavol Bielik, and Martin Vechev. 2016a. Probabilistic model for code with decision trees. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, page 731–747, New York, NY, USA. Association for Computing Machinery.

Veselin Raychev, Pavol Bielik, Martin Vechev, and Andreas Krause. 2016b. Learning programs from noisy data. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, page 761–774, New York, NY, USA. Association for Computing Machinery.

Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1715–1725, Berlin, Germany. Association for Computational Linguistics.

Vighnesh Shiv and Chris Quirk. 2019. Novel positional encodings to enable tree-based transformers. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d' Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 12081–12091. Curran Associates, Inc.

Marko Vasic, Aditya Kanade, Petros Maniatis, David Bieber, and Rishabh Singh. 2019a. Neural program repair by jointly learning to localize and repair. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net.

Marko Vasic, Aditya Kanade, Petros Maniatis, David Bieber, and Rishabh singh. 2019b. Neural program repair by jointly learning to localize and repair. In *International Conference on Learning Representations*.

Shengbin Xu, Yuan Yao, Feng Xu, Tianxiao Gu, Hanghang Tong, and Jian Lu. 2019. Commit message generation for source code changes. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*, pages 3975–3981. International Joint Conferences on Artificial Intelligence Organization.