Blended, Precise Semantic Program Embeddings

Ke Wang
Visa Research
U.S.A
kewang@visa.com

Zhendong Su
ETH
Switzerland
zhendong.su@inf.ethz.ch

Abstract

Learning neural program embeddings is key to utilizing deep neural networks in program languages research — precise and efficient program representations enable the application of deep models to a wide range of program analysis tasks. Existing approaches predominately learn to embed programs from their source code, and, as a result, they do not capture deep, precise program semantics. On the other hand, models learned from runtime information critically depend on the quality of program executions, thus leading to trained models with highly variant quality. This paper tackles these inherent weaknesses of prior approaches by introducing a new deep neural network, LiGer, which learns program representations from a mixture of symbolic and concrete execution traces. We have evaluated LiGer on two tasks: method name prediction and semantics classification. Results show that LiGer is significantly more accurate than the state-of-the-art static model code2seq in predicting method names, and requires on average around 10x fewer executions covering nearly 4x fewer paths than the state-of-the-art dynamic model DYPRO in both tasks. LiGer offers a new, interesting design point in the space of neural program embeddings and opens up this new direction for exploration.

CCS Concepts: • **Software and its engineering → General programming languages**; • **Computing methodologies → Learning latent representations**.

Keywords: Semantic Program Embedding, Attention Network, Static and Dynamic Program Features

ACM Reference Format:
Ke Wang and Zhendong Su. 2020. Blended, Precise Semantic Program Embeddings. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '20), June 15–20, 2020, London, UK*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3385412.3385999

PLDI '20, June 15–20, 2020, London, UK

ACM ISBN 978-1-4503-7613-6/20/06…$15.00
https://doi.org/10.1145/3385412.3385999

1 Introduction

Learning representations has been a major focus in deep learning research for the past several years. Mikolov et al. pioneered the field with their seminal work on learning word embeddings [18, 19]. The idea is to construct a vector space for a corpus of text such that words found in similar contexts in the corpus are located in close proximity to one another in the vector space. Word embeddings, along with other representation learning (*e.g.* doc2vec [17]), become vital in solving many downstream Natural Language Processing (NLP) tasks, such as language modeling [7] and sentiment classification [12].

Similar to word embeddings, the goal of this paper is to learn *program embeddings*, vector representations of program semantics. By learning program embeddings, the power of deep neural networks (DNNs) can be utilized to tackle many program analysis tasks. For example, Alon et al. [3] present a DNN to predict the name of a method given its body. Wang et al. [28] propose a deep model to guide the repair of student programs in Massive Open Online Courses (MOOCs). Despite such notable advances, an important challenge remains: How to tackle the precision and efficiency issues in learning program embeddings? As illustrated by Wang and Christodorescu [27], due to the inherent gap between program syntax and semantics, models learned from source code (*i.e.*, the *static models*) can be imprecise at capturing semantic properties. Consider, for example, the programs in Figure 1. State-of-the-art static models can neither recognize the equivalent semantics between programs in Figures 1a and 1c, nor the different semantics between programs in Figures 1a and 1b. The reason for this is quite obvious — static models base their predictions on the surface-level program syntax. Specifically, programs 1a and 1b are syntactically much more similar than programs 1a and 1c despite that programs 1a and 1c implement the same sorting strategy, namely *Bubble Sort*.

In parallel, a separate class of models has been proposed that embed programs from their concrete execution traces (*i.e.*, the *dynamic models*). Compared to source code, program executions capture accurate, deep program semantics, thus offering benefits beyond static models that reason over syntactic representations. Figure 2 shows the executions of the three programs with an input array `A = [8,5,1,4,3]` according to the state-based encoding proposed by Wang et al. [28]. Naturally, the semantic relationship among the three programs becomes much clearer. Despite their advantages,

```
static int SortI(int[] A)
{
    int left = 0;
    int right = A.Length - 1;

    for(int i=right;i>left;i--) {
        for(int j=left;j<i;j++) {
            if (A[j] > A[j + 1]) {
                int tmp = A[j];
                A[j] = A[j + 1];
                A[j + 1] = tmp;
    }}}

    return A;
}
```

(a) Bubble Sort

```
static int SortII(int[] A)
{
   int left = 0;
   int right = A.Length;

   for(int i=left;i<right;i++){
      for(int j=i-1;j>=left;j--){
          if (A[j] > A[j + 1]){
             int tmp = A[j];
             A[j] = A[j + 1];
             A[j + 1] = tmp;
   }}}

   return A;
}
```

(b) Insertion Sort

```
static int SortIII(int[] A)
{
   int swappbit = 1;
   while (swappbit != 0) {
      swappbit = 0;
      for (int i=0;i<A.Length-1;i++)
      {
          if (A[i + 1] > A[i]) {
              int tmp = A[i];
              A[i] = A[i + 1];
              A[i + 1] = tmp;
              swappbit = 1;
      }}}
   return A;
}
```

(c) Bubble Sort

Figure 1. Example programs that implement a sorting routine. Code highlighted within the shadow boxes depicts the syntactic differences between programs 1a and 1b.

{**A:[8, 5, 1, 4, 3]**; *left:0;* **right:⊥**}	{**A:[8, 5, 1, 4, 3]**; *left:0;* **right:⊥**}	{**A:[8, 5, 1, 4, 3]**; *swapbit:1*}
{**A:[8, 5, 1, 4, 3]**; **left:0**; *right:4*}	{**A:[8, 5, 1, 4, 3]**; **left:0**; *right:5*}	{**A:[8, 5, 1, 4, 3]**; *swapbit:0*}
{*A:[5, 5, 1, 4, 3]*; **left:0; right:4**}	{*A:[5, 5, 1, 4, 3]*; **left:0; right:5**}	{*A:[5, 5, 1, 4, 3]*; **swapbit:1**}
{*A:[5, 8, 1, 4, 3]*; **left:0; right:4**}	{*A:[5, 8, 1, 4, 3]*; **left:0; right:5**}	{*A:[5, 8, 1, 4, 3]*; **swapbit:1**}
{*A:[5, 1, 1, 4, 3]*; **left:0; right:4**}	{*A:[5, 1, 1, 4, 3]*; **left:0; right:5**}	{*A:[5, 1, 1, 4, 3]*; **swapbit:1**}
{*A:[5, 1, 8, 4, 3]*; **left:0; right:4**}	{*A:[5, 1, 8, 4, 3]*; **left:0; right:5**}	{*A:[5, 1, 8, 4, 3]*; **swapbit:1**}
{*A:[5, 1, 4, 4, 3]*; **left:0; right:4**}	{*A:[1, 1, 8, 4, 3]*; **left:0; right:5**}	{*A:[1, 1, 8, 4, 3]*; **swapbit:1**}
{*A:[5, 1, 4, 8, 3]*; **left:0; right:4**}	{*A:[1, 5, 8, 4, 3]*; **left:0; right:5**}	{*A:[1, 5, 8, 4, 3]*; **swapbit:1**}
{*A:[5, 1, 4, 3, 3]*; **left:0; right:4**}	{*A:[1, 5, 4, 4, 3]*; **left:0; right:5**}	{*A:[1, 5, 4, 4, 3]*; **swapbit:1**}
{*A:[5, 1, 4, 3, 8]*; **left:0; right:4**}	{*A:[1, 5, 4, 8, 3]*; **left:0; right:5**}	{*A:[1, 5, 4, 8, 3]*; **swapbit:1**}
{*A:[1, 1, 4, 3, 8]*; **left:0; right:4**}	{*A:[1, 4, 4, 8, 3]*; **left:0; right:5**}	{*A:[1, 4, 4, 8, 3]*; **swapbit:1**}
{*A:[1, 5, 4, 3, 8]*; **left:0; right:4**}	{*A:[1, 4, 5, 8, 3]*; **left:0; right:5**}	{*A:[1, 4, 5, 8, 3]*; **swapbit:1**}
{*A:[1, 4, 4, 3, 8]*; **left:0; right:4**}	{*A:[1, 4, 5, 3, 3]*; **left:0; right:5**}	{*A:[1, 4, 5, 3, 3]*; **swapbit:1**}
{*A:[1, 4, 5, 3, 8]*; **left:0; right:4**}	{*A:[1, 4, 5, 3, 8]*; **left:0; right:5**}	{*A:[1, 4, 5, 3, 8]*; **swapbit:1**}
{*A:[1, 4, 3, 3, 8]*; **left:0; right:4**}	{*A:[1, 4, 3, 3, 8]*; **left:0; right:5**}	{*A:[1, 4, 3, 3, 8]*; **swapbit:1**}
{*A:[1, 4, 3, 5, 8]*; **left:0; right:4**}	{*A:[1, 4, 3, 5, 8]*; **left:0; right:5**}	{*A:[1, 4, 3, 5, 8]*; **swapbit:1**}
{*A:[1, 3, 3, 5, 8]*; **left:0; right:4**}	{*A:[1, 3, 3, 5, 8]*; **left:0; right:5**}	{*A:[1, 3, 3, 5, 8]*; **swapbit:1**}
{*A:[1, 3, 4, 5, 8]*; **left:0; right:4**}	{*A:[1, 3, 4, 5, 8]*; **left:0; right:5**}	{*A:[1, 3, 4, 5, 8]*; **swapbit:1**}
(a)	**(b)**	**(c)**

Figure 2. Encoding the executions of the programs in Figure 1 with the input array `A = [8,5,1,4,3]`. At each step, the variable in italics is updated. Steps that are in red illustrate the semantic differences between bubble sort (program 1a) and insertion sort (program 1b) concerning only the manipulation of the input array `A`. Note that we have omitted all loop induction variables, variable `tmp` in both programs 1a and 1b as well as some steps that update the variable `swapbit` in program 1c to simply our presentation.

the performance of dynamic models heavily depends on the quality of program executions. In particular, dynamic models, similar to dynamic program analysis, can suffer from insufficient code coverage. Even for each covered path, dynamic models may need a large number of execution instances to generalize, resulting in a lengthy, expensive training process.

To tackle those aforementioned issues of both strands of prior work, we introduce a novel, *blended* approach for learning precise and efficient representations of program semantics. Our insight is to blend the respective strengths of static and dynamic models to mitigate their respective weaknesses. Different from dynamic models that consider only program states created along an execution path, our blended model incorporates the additional symbolic representation of each statement (*i.e. symbolic trace*) whose execution leads to a corresponding program state.

The benefits of blending these feature dimensions are twofold. First, learning from symbolic program encodings is shown difficult for DNNs. Concrete program states, which give live illustrations of program behavior, provide explanations to DNNs about each symbolic statement's semantics. As a result, models trained on the combined features capture deeper semantic properties than symbolic traces alone (*cf.* Section 6.3.2). Second, a symbolic trace typically generalizes a large number of concrete executions. Therefore, symbolic traces present high-level, general descriptions of program meaning to DNNs. It is for this very reason that symbolic traces stand out as the major feature dimension from which models generalize. In the presence of the symbolic feature dimension, DNNs deemphasize the role of dynamic program features, leading to reduced demand on the number of concrete executions. As another benefit of our blended model, we observe that DNNs trained on both feature dimensions are also more resilient to the varying diversity on program executions. Indeed, when the path coverage on the targeted program is systematically reduced, our blended network largely maintains its accuracy, and thus has improved data reliance (*cf.* Section 6.1.2).

Dynamic vs. Static Embeddings. LiGer is grounded as a dynamic approach, and thus shares the well-known tradeoffs between dynamic and static approaches in programming languages, such as dynamic and static program analyses — while static approaches suffer from imprecision, dynamic approaches are weak in generality and coverage since they depend on executions. Thus, we do not claim that dynamic approaches are always superior to static approaches; rather, they both are valuable, complementary design choices. We design LiGer centered on being dynamic to learn deep, precise semantic program embeddings for settings where quality executions exist. The primary goal of this work is to reduce dynamic models' reliance on program executions. Indeed, by incorporating symbolic features into a pure dynamic model, we show that LiGer offers strong advantages over both purely static and dynamic models where it is applicable.

We have realized our approach in a new DNN, LiGer, and extensively evaluated it. To demonstrate LiGer's generality, we pick two downstream tasks: method name prediction and semantics classification. In method name prediction task, we measure LiGer's performance on two Java datasets, Java-med and Java-large, proposed in prior work [3]. We use Randoop [22], a feedback-directed, unit test generator for Java programs, to produce meaningful executions for LiGer to learn from. Our results show that LiGer is highly accurate. In particular, it significantly outperforms code2seq [3], the state-of-the-art static model. In addition, we find in both tasks LiGer yields comparable performance to DYPRO [26], the state-of-the-art dynamic model, when consuming on average around 10x fewer executions covering nearly 4x fewer code paths.

Contributions. We make the following main contributions:

- We propose a novel, blended approach that combines static and dynamic program features for learning precise, efficient representations of program semantics.
- We realize our approach in LiGer and evaluate it on the tasks of method name prediction and semantics classification. Results show that LiGer both significantly outperforms the state-of-the-art static model, code2seq, and requires far fewer program executions for both training and testing than the state-of-the-art dynamic model, DYPRO. Thus, LiGer offers an interesting, valuable design point in the space of program embeddings.
- We present the details of our extensive evaluation of LiGer, including an ablation study that analyzes the contributions of LiGer's several crucial components to its overall performance.

The remainder of this paper is organized as follows. Section 2 formalizes the notion of execution traces and several pertinent concepts. Section 3 provides our insight that motivates the design of LiGer. Section 4 describes the background of this work. Section 5 presents LiGer's architecture. Section 6 details our evaluation results. We survey related work in Section 7 and conclude in Section 8 with a discussion of future work.

2 Formalization

In general, given a program P and an input I, an execution trace is obtained by executing P on I. Its concept and notations are standard, which we formalize more precisely below.

Definition 2.1. (Execution Trace) An execution trace, denoted by π, is a sequence in the form of $s_0 \rightarrow (e_i \rightarrow s_i)^*$, where e_i denotes a statement encountered as P executes on an input I; s_i denotes a program state, which is a set of variable/memory and value pairs immediately after the execution of statement e_i; s_0 is the initial program state; and * denotes the Kleene star.

As an example, Figure 3 presents a graphical illustration of an execution trace of the program in Figure 4 with input `A = "abc"` and `B = "bca"`.

Definition 2.2. (Symbolic Trace) Given an execution trace, π, a symbolic trace, σ, is the sequence of statements visited in π in the form of $(e_i \rightarrow e_{i+1})^*$.

Similarly, Figure 3 also gives an example of symbolic trace, which is a projection of the execution trace *w.r.t.* the program statements.

Definition 2.3. (State Trace) Given an execution trace, π, a state trace, ϵ, is the sequence of program states created in π in the form of $(s_i \rightarrow s_{i+1})^*$.

Again, Figure 3 shows an example of state trace, which is a projection of the execution trace *w.r.t.* the program states.

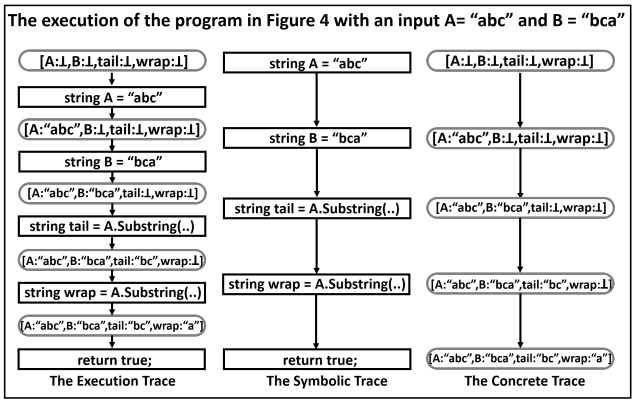

Figure 3. Execution traces, symbolic traces and concrete traces of the program in Figure 4.

```
public bool IsStringRotation(string A, string B)
{
   if (A.Length != B.Length)
        return false;

   for (int i = 1; i < A.Length; i++)
   {
       string tail = A.Substring(i,A.Length-i);
       string wrap = A.Substring(0, i);

       if (tail + wrap == B)
           return true;
   }

   return false;
}
```

Figure 4. An example program.

3 Motivation and Insight

Learning program embeddings from execution traces has been explored in the literature. The prior work can be divided into two categories: static and dynamic. The former refers to learning program embeddings exclusively on symbolic traces. As an example, Henkel et al. [14] train a model from symbolic traces for a code analogy task. Although learning from symbolic traces captures, to a certain degree, program properties more at the semantic level than purely the syntactic source code, it suffers from the same fundamental issue of all static models. That is such approaches leave the burden on the deep models to reason about program semantics through a syntactic representation, a task that is proven to be challenging even for state-of-the-art DNNs. To give a simple example, a capable neural network needs to recognize the identical semantics `i+=i` and `i*=2` denote because such variations are ubiquitous in real-world code. Note that Henkel et al.'s approach relies on user-defined abstraction templates, which unfortunately do not address the problem's root cause.

In contrast to the symbolic trace-based approaches, another line of work only considers concrete state traces for learning program representations. In particular, Wang et al. [28] propose a model learned from concrete state traces to predict the type of errors students make in their programming assignments. The intuition behind the approach is to capture the semantics of a program through the states that are created in an execution. The advantage of their approach is the canonicalization of syntactic variations as programs of equivalent semantics will always create identical program states regardless of their syntactic differences (*e.g.*, the earlier example involving `i+=i` and `i*=2`). Despite this strength, models that embed programs from concrete state traces have their own weaknesses. In principle, a symbolic trace can be instantiated to a large or arbitrary number of concrete states traces. Therefore, symbolic traces lay the foundation of feature representations. By completely disregarding the program syntax, deep models lose the high-level overview of the execution trace, therefore demanding a large number of concrete traces to compensate. Assuming that a model requires M concrete traces to estimate the semantics of one symbolic trace, learning a program yielding N total symbolic traces amounts to $M \times N$ concrete traces. This drastic increase in the amount of training data leads to lengthy and inefficient training.

The deficiencies of the static and dynamic models together motivate the design of LiGer. By simply exposing the entire execution traces (*i.e.* both symbolic and concrete state traces) as structured inputs, LiGer combines the strengths of both types of models and outperforms ones learned from either symbolic or concrete traces alone. On one hand, concrete traces help LiGer deal with the challenge of learning from symbolic program representations. Instead of generalizing from high-level program symbols, LiGer is also provided with low-level concrete explanations. Consider the earlier example with `i+=i` and `i*=2`. Although the two statements are represented differently in terms of symbolic traces, their identical program states force LiGer to inject the notion of equivalent semantics between the two statements. Ultimately, this allows LiGer's to reduce the difficulty of reasoning about program semantics from syntax.

Additionally, since the symbolic representation of an execution trace is still present in the feature representation, LiGer has a general, symbolic view of the execution trace, therefore does not need a large number of concrete traces to generalize. Thus, it needs less training data.

Through our extensive experiments, we observe that LiGer possesses an interesting benefit. As we systematically lower the path coverage of programs in both training and test sets, LiGer is able to maintain its accuracy. This property helps LiGer address the intrinsic limitations of the dynamic models. That is, even when programs are hard to cover, LiGer can still reason about their semantics from the limited available traces, thus further reducing its reliance on training data.

4 Preliminaries

This section reviews the necessary background, in particular, recurrent neural networks [15], TreeLSTM [24], and attention neural networks, the building blocks of LiGer.

4.1 Recurrent Neural Network

A recurrent neural network (RNN) [15] is a class of artificial neural networks that are distinguished from feedforward networks by their feedback loops. This allows RNNs to ingest their own outputs as inputs. It is often said that RNNs have memory, enabling them to process sequences of inputs.

Here we briefly describe the computation model of a vanilla RNN. Given an input sequence, embedded into a sequence of vectors $x = (x_1, \cdots, x_{T_x})$, an RNN with N inputs, a single hidden layer with M hidden units, and Q output units. We define the RNN's computation as follows:

$$\begin{aligned} h_t &= f(W * x_t + V * h_{t-1}) \\ o_t &= softmax(Z * h_t) \end{aligned} \tag{1}$$

where $x_t \in \mathbb{R}^N$, $h_t \in \mathbb{R}^M$, $o_t \in \mathbb{R}^Q$ is the RNN's input, hidden state and output at time t, f is a non-linear function (*e.g.* tanh or sigmoid), $W \in \mathbb{R}^{M*N}$ denotes the weight matrix for connections from input layer to hidden layer, $V \in \mathbb{R}^{M*M}$ is the weight matrix for the recursive connections (*i.e.* from hidden state to itself) and $Z \in \mathbb{R}^{Q*M}$ is the weight matrix from hidden to the output layer.

4.2 TreeLSTM

TreeLSTM [24], a variant of LSTM, was proposed to work on tree topology. The following equation sums up how TreeLSTM works at a high-level.

$$h_j = o_j \odot tanh(i_j \odot \tilde{c_j} + \sum_{k \in C_j} f_{jk} \odot c_k)$$

where C_j is the set of children of node j.

Like a standard LSTM, h_j, o_j and i_j denote the hidden state, output gate and input gate of node j. $\tilde{c_j}$ is new candidate value proposed for updating the cell state of node j. A significant adaption TreeLSTM introduced is the multiplication of forget gates, which allow itself to selectively incorporate information from each child to the parent node. In particular, the output of each forget gate of node j is multiplied with the cell state of the corresponding child ($f_{jk} \odot c_k$), which then are combined into a single output ($\sum_{k \in C_j}(\cdot)$). Another difference TreeLSTM makes is in how i_j, o_j, c_j get updated. In a traditional LSTM, the update stems from the hidden state, h_j, in the previous step whereas TreeLSTM uses the sum of the hidden states of its children.

4.3 Neural Attention Network

Before we describe attention neural networks, we give a brief overview of the underlying framework — *Encoder-Decoder* — proposed by Cho et al. [8] and Devlin et al. [11].

The encoder-decoder [8, 11] neural architecture was first introduced in the field of machine translation. An encoder neural network reads and encodes a source sentence into a vector, based on which a decoder outputs a translation. From a probabilistic point of view, the goal of translation is to find the target sentence L_t that maximizes the conditional probability of L_t given source sentence L_s (*i.e.* $\text{argmax}_{L_t} P(L_t|L_s)$).

Using the terminologies defined in Section 4.1, we explain the computation model of an encoder-decoder. Given an input sequence x, the encoder performs the computation defined in Equation 1 and spits out its final hidden state c (*i.e.* $c = h_{T_x}$). The decoder is responsible for predicting each word y_t given the vector c and all the previously predicted words $(y_1, \cdots, y_{t-1})$. In other words, the decoder outputs the probability distribution of $y = (y_1, \cdots, y_{T_y})$ by decomposing the joint probability into the ordered conditionals:

$$P(y) = \prod_{t=1}^{T_y} P(y_t|(y_1, \cdots, y_{t-1}), c)$$

With an RNN, each ordered conditional is defined as:

$$P(y_t|(y_1, \cdots, y_{t-1}), c) = g(y_{t-1}, d_t, c)$$

where d_t is the hidden state of the decoder RNN at time t.

An issue of this encoder–decoder architecture is that the encoder has to compress all the information from a source sentence into a vector to feed the decoder. This is especially problematic when the encoder has to deal with long sentences. To address this issue, Bahdanau et al. [5] introduced an attention mechanism on top of the standard encoder-decoder framework that learns to align and translate simultaneously. The proposed solution is to enable the decoder network to search the most relevant information from the source sentence to concentrate when decoding each target word. In particular, instead of fixing each conditional probability on the vector c in Equation 2, a distinct context vector c_t for each y_t is used:

$$P(y_t|(y_1, \cdots, y_{t-1}), x) = g(y_{t-1}, d_t, c_t)$$

where g is a nonlinear, potentially multi-layered function that outputs the probability of y_t, and d_t is the hidden state of the RNN.

To compute the context vector c_t, a bi-directional RNN is adopted which reads the input sequence x from both directions (*i.e.*, from x_1 to x_{T_x} and vice versa), and produces a sequence of forward hidden states $(\overrightarrow{h_1}, \cdots, \overrightarrow{h_{T_x}})$ and backward hidden states $(\overleftarrow{h_1}, \cdots, \overleftarrow{h_{T_x}})$. We obtain an annotation h_d for each word x_d by concatenating the forward hidden state $\overrightarrow{h_d}$ and the backward one $\overleftarrow{h_d}$. Now c_t can be computed as a weighted sum of these annotations h_d :

$$c_t = \sum_{d=1}^{T_x} \alpha_{td} h_d \tag{2}$$

The attention weight α_{td} of each annotation h_d is computed by

$$\alpha_{td} = \frac{exp(\mu_{td})}{\sum_{k=1}^{T_x} exp(\mu_{tk})}$$

where $\mu_{td} = a(d_{t-1}, h_d)$ is the attention score which reflects the importance of the annotation h_d *w.r.t.* the previous hidden state d_{t-1} in deciding the next state d_t and generating y_t. The parameter a stands for a feed-forward neural network that is jointly trained with the system's other components.

5 Model

This section presents the technical details of LiGer's architecture *w.r.t.* the problem of predicting method names.

5.1 LiGer's Architecture

LiGer's architecture is depicted in Figure 5. At a high level, LiGer employs a typical encoder-decoder architecture, in which the encoder learns the semantic embedding of a method while the decoder generates the method name as a sequence of words.

5.1.1 Encoder.

We split LiGer's encoder architecture into four layers and discuss each layer in detail.

Terminology. To turn a program P in its source code form to the format LiGer requires, we symbolically execute P to obtain U distinct paths, where each path σ_i is associated with a condition ϕ_i. By solving ϕ_i, we obtain concrete traces $\epsilon_{i\_1}, \cdots, \epsilon_{i\_N_\epsilon}$. Each program state in a concrete trace, $\epsilon_{i\_i'}$, is a tuple $(v_1, \cdots, v_{N_v})$, where $v_i \in \mathcal{D}_d$ is the value of a variable in P. $\mathcal{D}_d$ refers to the set of all values any variable has ever been assigned in any concrete trace of any program in our dataset. Note that the order of variables are fixed across all program states in any concrete trace of P. We also define $\mathcal{D}_s$ to include all tokens extracted from all programs in our dataset together with all AST (non-leaf) node types of the language P is written in.

Object Types. Unlike Wang et al. [28] which only considers primitive data types, LiGer is capable of handling object variables. Specifically, it treats the value of an object, v_i, as an array $attr(v_i)$. In other words, LiGer flattens v_i into an array of primitive data types, $attr(v_i)[0], \cdots, attr(v_i)[-1]$, where $attr(v_i)[0]/attr(v_i)[-1]$ denotes the first/last value in the flattened array.

Vocabulary Embedding Layer. In this layer, each item in $\mathcal{D}_s$ and $\mathcal{D}_d$ will be assigned a vector. Consider the j-th statement in σ_i, the state it creates in $\epsilon_{i\_i'}$ will be encoded as $(x_{i\_i'\_j\_attr(v_0)[0]}, \cdots, x_{i\_i'\_j\_attr(v_0)[-1]}) \cdots (x_{i\_i'\_j\_attr(v_{N_v})[0]}, \cdots, x_{i\_i'\_j\_attr(v_{N_v})[-1]})$.

Fusion Layer. This layer embeds each statement in σ_i and each state in $\epsilon_{i\_i'}$ before fusing the two feature dimensions, which forms the core of our blended approach. To facilitate the later presentation, we introduce and formalize the notion of blended traces.

Definition 5.1. (Blended Trace) Given a symbolic trace σ and multiple concrete traces, $\epsilon_1, \cdots, \epsilon_{N_\epsilon}$ that traverse the same program path as σ, a blended trace, λ, is a sequence of the form $(\theta_i \rightarrow \theta_{i+1})^*$, where θ_i is an ordered pair $<e_i, S_i>$, where e_i is a statement in σ and $S_i = \{s_{i\_1}, \cdots, s_{i\_N_\epsilon}\}$ is the set of program states e_i created in $\epsilon_1, \cdots, \epsilon_{N_\epsilon}$.

For simplicity, we assume that a blended trace λ_i is composed of σ_i and two concrete traces, ϵ_{i_1} and ϵ_{i_2}. Given the j-th ordered pair in λ_i, LiGer employs a TreeLSTM to embed a statement via its abstract syntax tree. At each step, given the vector representation of the tokens (for terminal nodes) and AST node types (for non-terminal nodes) produced by the vocabulary embedding layer, we recursively updating the hidden states of parent nodes based on those of the child nodes. Finally, we extract the hidden state of the root to be the embedding vector of the statement denoted by $h_{i\_j\_sta}$.

As for embedding the program states in the j-th ordered pair, first, we compute the vector representation of each variable depending on its type. If the variable is an object type, we will use a RNN to embed its value:

$$h'_{i\_i'\_j\_v_{I_v}} = f_1(x_{i\_i'\_j\_attr(v_{I_v})[-1]}, h'_{i\_i'\_j\_attr(v_{I_v})[-1]}) \quad (3)$$

If the variable is a primitive type, we directly use the embedding of its value produced by the vocabulary embedding layer *s.t.* $h'_{i\_i'\_j\_v_{I_v}} = x_{i\_i'\_j\_v_{I_v}}$. Next, we use the second RNN to embed a program state with each variable embedding computed from Equation 3:

$$h_{i\_1\_j} = f_2(h'_{i\_1\_j\_v_{N_v}}, h_{i\_1\_j\_v_{N_v-1}})$$
$$h_{i\_2\_j} = f_2(h'_{i\_2\_j\_v_{N_v}}, h_{i\_2\_j\_v_{N_v-1}})$$

Now we present the idea key to LiGer's success. To combine the strengths of both approaches, we fuse the vector representations across the feature dimensions to compute a single embedding of each ordered pair in a blended trace. Specifically, we adopt the attention mechanism to allocate a weight for each feature vector $h_{i\_j\_sta}$, $h_{i\_1\_j}$ and $h_{i\_2\_j}$:

$$\alpha_{i\_j\_sta} = \frac{exp(\mu_{i\_j\_sta})}{exp(\mu_{i\_j\_sta}) + exp(\mu_{i\_1\_j}) + exp(\mu_{i\_2\_j})}$$
$$\alpha_{i\_1\_j} = \frac{exp(\mu_{i\_1\_j})}{exp(\mu_{i\_j\_sta}) + exp(\mu_{i\_1\_j}) + exp(\mu_{i\_2\_j})}$$
$$\alpha_{i\_2\_j} = \frac{exp(\mu_{i\_2\_j})}{exp(\mu_{i\_j\_sta}) + exp(\mu_{i\_1\_j}) + exp(\mu_{i\_2\_j})}$$

where $\mu_{i\_j\_sta}$, $\mu_{i\_1\_j}$ and $\mu_{i\_2\_j}$ are defined below:

$$\mu_{i\_j\_sta} = a_1(h_{i\_j\_sta} \oplus H^e_{i\_j-1})$$
$$\mu_{i\_1\_j} = a_1(h_{i\_1\_j} \oplus H^e_{i\_j-1})$$
$$\mu_{i\_2\_j} = a_1(h_{i\_2\_j} \oplus H^e_{i\_j-1})$$

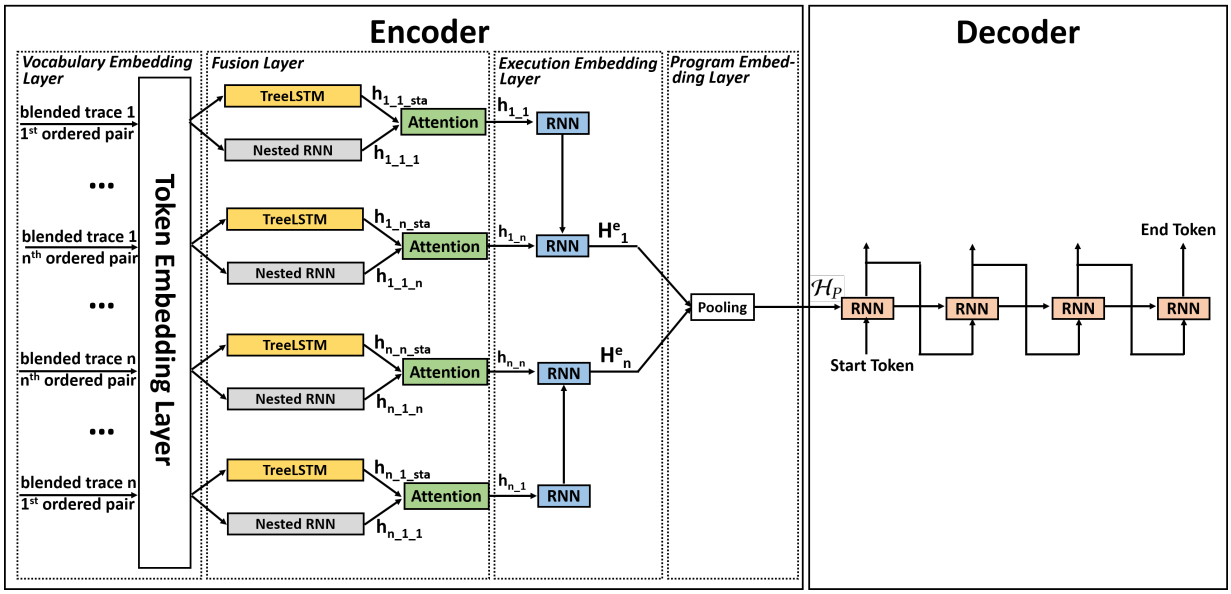

Figure 5. LiGer's architecture.

where $H^e_{i\_j-1}$ denotes the embedding that represents λ_i before j-th ordered pair,[1] $\oplus$ denotes vector concatenation and a_1 stands for a feedforward neural network.

Using the attention weights, we compute the embedding of the j-th ordered pair in λ_i as:

$$h_{i\_j} = \alpha_{i\_j\_sta} * h_{i\_j\_sta} + \alpha_{i\_1\_j} * h_{i\_1\_j} + \alpha_{i\_2\_j} * h_{i\_2\_j}$$

Note that we evenly distribute the weights across all feature vectors embedded from the first ordered pair in any blended trace.

Executions Embedding Layer. Given $h_{i\_1}, \cdots, h_{i\_|\lambda_i|}$, the embeddings for all ordered pairs in λ_i, we use the third RNN to model the flow of the blended trace.

$$H^e_i = f_3(H^e_{i\_|\lambda_i|-1}, h_{i\_|\lambda_i|})$$

where $H^e_{i\_j}$ is the embedding that represents the partial blended trace from the first ordered pair to the j-th ordered pair (including the j-th ordered pair). In other words, H^e_i represents the entire λ_i.

Programs Embedding Layer. We design a pooling layer to compress the embeddings of all the blended traces, $H^e_1, \cdots, H^e_U$, one for each path to a program embedding $\mathcal{H}_P$.

$$\mathcal{H}_P = max\_pooling(H^e_1, \cdots, H^e_U)$$

5.1.2 Decoder. Given the encoder outputs $\mathcal{H}_P$, and $\{\{H^e_{i\_j} | j \in [1, |\lambda_i|]\} | i \in [1, U]\}$, the set of embeddings for each blended trace of program P, we use another RNN to decode the method names. For initialization, we provide the decoder with the program embedding $\mathcal{H}_P$. The decoder also receives a special token to begin, and emits another to end the generation.

Attention. As explained in Section 4.3, we incorporate the attention mechanism to the extended architecture to aid the decoding process. Unlike the attention neural network introduced previously where the decoder attends over the input symbols from a single source sentence, our decoder attends over the flow of all blended traces (*i.e.* $\{\{H^e_{i\_j} | j \in [1, |\lambda_i|]\} | i \in [1, U]\}$).

We recompute the context vector c_t (defined in Equation 2) for each generated word y_t as:

$$c_t = \sum_{i=1}^{U} \sum_{j=1}^{|\lambda_i|} \alpha_{t\_i\_j} H^e_{i\_j}$$

Each attention weight $\alpha_{t\_i\_j}$ is computed by

$$\alpha_{t\_i\_j} = \frac{exp(\mu_{t\_i\_j})}{\sum_{i=1}^{U} \sum_{j=1}^{|\lambda_i|} exp(\mu_{t\_i\_j})}$$

where $\mu_{t\_i\_j} = a_2(H^d_{t-1}, H^e_{i\_j})$ is the attention score which measures how well each $H^e_{i\_j}$ correlates with the previous hidden state of the decoder, H^d_{t-1}. The parameter a_2 stands for a feedforward neural network that is jointly trained with other components in LiGer's architecture.

6 Evaluation

This section presents the details of our evaluation. First, we evaluate how well LiGer predicts method names. Then, we evaluate LiGer on COSET [27] for a semantics classification task. Finally, we perform substantial ablation studies to evaluate LiGer's internal design and realization.

[1] We give the formal definition of $H^e_{i\_j-1}$ in the next paragraph.

6.1 Method Name Prediction Task

Datasets. We use Java-med and Java-large, two datasets proposed by Alon et al. [3] for the task of method name prediction. We parse programs into token sequences and abstract syntax trees (ASTs) using JavaParser. To collect execution traces, we rely on Randoop [22], a random unit test generator for Java program, to trigger high-coverage executions. Ideally, we would like to use Java-med and Java-large in their entirety. However, there are several engineering issues that force us to settle for a subset of both datasets: (1) some programs do not compile. (2) some programs reference external packages that Randoop does not have access to. To alleviate this issue, we have made many popular Java libraries available to Randoop such as Apache Commons and Google Core Libraries for Java. Nevertheless, the list won't be inclusive, and some libraries will be missed out. (3) some programs take too long (exceeding a pre-defined timeout) for Randoop to generate test inputs. (4) some programs are too small to be considered (a couple of lines). Table 1 shows the detailed statistics of both datasets before and after the filtering. Following the protocol proposed by Alon et al. [3], methods in training, validation and test sets are extracted from distinct projects. We serialize each object to register its value during instrumentation; we reserve a special symbol for the value of the objects whose definitions are not accessible. For symbolic traces, we group concrete executions that traverse the same program path, and then decompose each path into a list of statements. In the end, we collected on average 20 symbolic traces, each of which is coupled with 5 concrete executions for each method in both datasets.

Baselines. We choose code2seq [2], the state-of-the-art static model for predicting method names, and code2vec [3], another static model that has achieved good results on the same problem. We re-train both models using their implementations open-sourced at GitHub repositories. To ensure the models we trained yield comparable performance, we perform a pre-test of both models on the entire Java-med and Java-large, and results[2] show the re-trained models achieve either better or very similar results to those reported in Alon et al. [2]. We also include in this experiment a slightly modified version of DYPRO [26], a dynamic model that supposedly learns from pure execution traces. In particular, we feed the variable names together with their values for DYPRO to embed execution traces.

Implementation. LiGer is implemented in Tensorflow. All RNNs have one single recurrent layer with 100 hidden units. We adopt random initialization for weight initialization. Our vocabulary has 9,641 unique tokens (for both static and dynamic feature dimensions), each of which is embedded into a 100-dimensional vector. We have experimented with other model parameters and included the results in Appendix for readers' perusal. All networks are trained using the Adam optimizer [16] with learning and decay rates set to their default values (*i.e.*, learning rate = 0.0001, beta1 = 0.9, beta2 = 0.999) and a mini-batch size of 100. We use a Red Hat Linux server hosting four Tesla V100 GPUs (with 32GB GPU memory).

Table 1. Dataset statistics. Original column denotes the # of methods in the original datasets. Filtered column denotes the remaining # of methods after the filtering.

Datasets	Java-med		Java-large	
	Original	Filtered	Original	Filtered
Training	3,004,536	74,951	15,344,512	338,126
Validation	410,699	5,000	320,866	5,000
Test	411,751	5,000	417,003	5,000
Total	3,826,986	84,951	16,082,381	438,126

Table 2. Compare LiGer against other models.

Models	Java-med			Java-large		
	Precision	Recall	F1	Precision	Recall	F1
code2vec	14.64	13.18	13.87	19.85	14.26	16.60
code2seq	32.95	20.23	25.07	36.49	22.51	27.84
DYPRO	37.84	24.31	29.60	41.57	26.69	32.51
LiGer	**39.88**	**27.14**	**32.30**	**43.28**	**31.43**	**36.42**

6.1.1 Accuracy.

We adopt the metric used by prior work [3] to measure precision, recall, and F1 score over case insensitive sub-tokens. The idea is that the prediction of a whole method name highly depends on that of the sub-words. For example, given a method named `computeDiff`, a prediction of `diffCompute` is considered a perfect answer (*i.e.*, the order of the sub-words does not matter), a prediction of `compute` has a full precision, but low recall, and a prediction of `compute- FileDiff` has full recall, but low precision. Table 2 shows the results all models achieve on our datasets (Table 1). LiGer is the most accurate model among all according to the F1 score, albeit not a substantial improvement over DYPRO. In contrast, static models, both code2vec and code2seq, perform significantly worse, and far away from what they achieved on the original Java-med and Java-large due to the substantial reduction of the training data.

Remarks. We also conduct a manual inspection on the results of code2seq and LiGer to reveal their respective strengths and weaknesses. Apart from the methods of simple, straight-forward data manipulations on which both models perform well, LiGer is more accurate in handling those that present greater algorithmic complexity.[3] Moreover, we discover that code2seq's prediction is more of a keywords mining process. That is, finding the most relevant words in the method body that describes its functionality. We did not

[2]See Appendix at https://kbwang.bitbucket.io/Appendices/PLDI20.pdf.

[3]We have provided a couple of example methods in the Appendix.

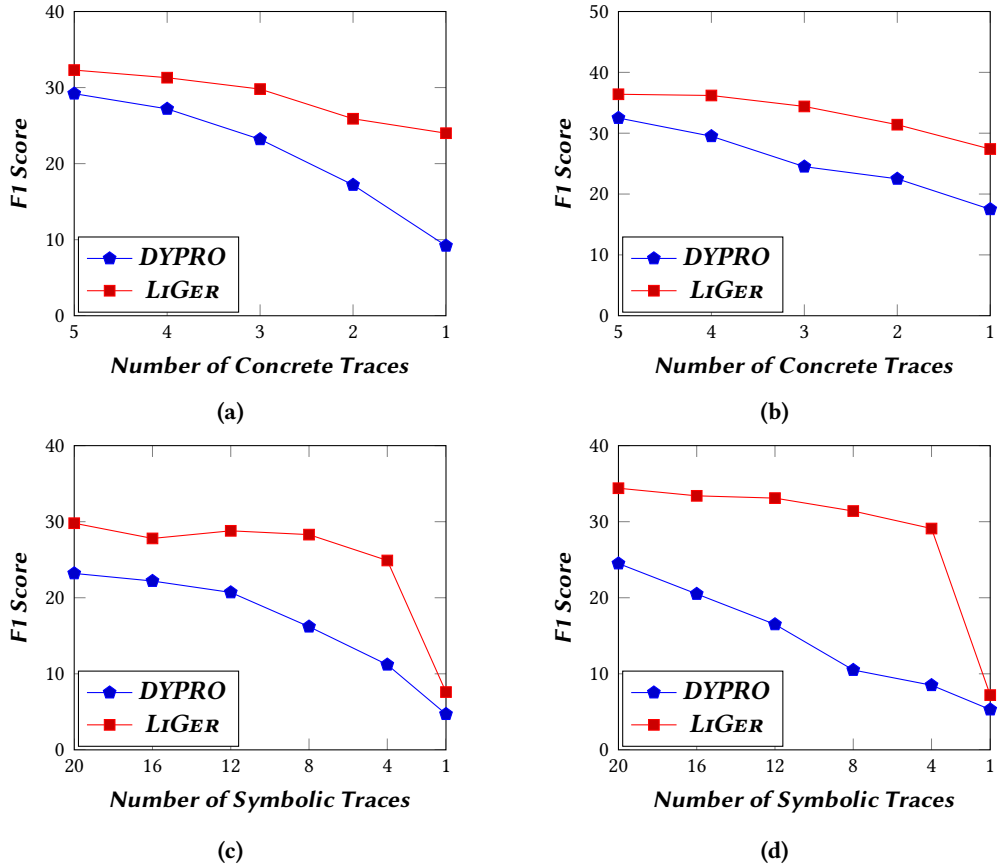

Figure 6. Evaluate LiGer's performance when either concrete or symbolic traces are down-sampled. Hereinafter, for the method name prediction task, charts on the left/right depict the results models achieved on Java-med/Java-large.

manage to catch one single method code2seq correctly predicts whose name does not appear in its body. Additionally, replacing keywords with less informative names for variable identifiers sways code2seq's previous correct predictions most of the time. On the contrary, LiGer incorporates the information of program states, which captures the essence of the program behavior. As a result, it is able to learn program properties at the semantic rather than syntactic level. Despite the weaknesses, static models enjoy an important advantage over LiGer in their applicability, namely, they will always function without executing or even compiling programs.

6.1.2 Data Reliance. More importantly, we examine LiGer *w.r.t.* one of our main goals. That is, how reliant is LiGer on executions to produce precise program embeddings. Reusing the method name prediction task, we evaluate LiGer from two aspects. First, we randomly reduce the number of concrete traces used to construct a blended trace while keeping the total number of symbolic traces constant for each method in Java-med and Java-large. In other words, we aim to understand how LiGer would perform when each symbolic trace is accompanied by fewer concrete traces. For comparison, we also evaluate the performance trend of DYPRO on the same concrete traces that LiGer is trained and tested on.

Figures 6a (6b) shows our results on Java-med (Java-large). In general, reducing concrete traces in a blended trace path has a small effect on LiGer's performance. Perhaps more unexpectedly, LiGer exhibits almost the same accuracy when it is supplied with no fewer than three concrete traces. For a more in-depth understanding, we also investigate how attention weights of each symbolic trace change when executions are down-sampled. We observe that, upon model convergence, the attention weight for each statement along each symbolic trace is 0.598 on average, and the weight stays largely unchanged throughout the reduction. Furthermore, the rest of the attention weight (*i.e.*, 0.412) is almost evenly split into the concrete traces regardless of their number. This finding shows that LiGer relies more on the symbolic feature dimension while generalizing from the training programs. Meanwhile, it views concrete traces as parallel instantiations of the same symbolic trace. Most importantly, LiGer is capable of compensating the loss of concrete traces by increasing the importance of the remaining ones, therefore preserving its accuracy.

Next, we investigate how LiGer reacts when the number of symbolic traces decreases. We emphasize that any dynamic technique will be (severely) affected if the given executions only cover a small part of a program under test. Therefore, we assume the preservation of line coverage throughout the reduction of symbolic traces, and study how decreased path coverage impacts LiGer's performance. In this regard, we first identify a minimum set of symbolic traces for each method in Java-med and Java-large that achieve the same line coverage as before, and then gradually remove symbolic traces that are not in the minimum set.

In this experiment, we randomly select three out of the original five concrete traces, from which we generate a minimum set of blended traces. As a baseline, we show how LiGer compares against DYPRO when given the concrete traces out of the blended ones.

When line coverage is preserved during path reduction, LiGer's performance is largely unaffected (Figure 6c/6d for Java-med/Java-large), indicating its strong resilience on the reduced program paths. Even if training and testing on the minimum set of blended traces, LiGer is comparable to DYPRO trained and tested on the entire set of concrete traces (25.88 vs. 29.60 F1 score on Java-med, and 30.42 vs. 32.51 F1 score on Java-large). As the average size of the minimum set of symbolic traces is calculated to be 5.3 (*i.e.*, 15.9 concrete executions) for each program, LiGer used almost 7x fewer executions covering nearly 4x fewer program paths. By learning from the minimum set of blended traces, LiGer also reduces the training time by more than ten times on both Java-med and Java-large with the same model architecture. It is worth mentioning that LiGer's accuracy drops sharply when it is provided with a single symbolic trace as expected. To a certain extent, LiGer predicts programs that are almost unrelated to those programs it is trained on due to the excessively low code coverage. Therefore, LiGer displays as poor a performance as DYPRO.

To conclude, compared to DYPRO, LiGer relies much less on program executions. First, LiGer has shown its resilience against the loss of concrete execution traces given that the path coverage of the targeted programs is maintained. In addition, LiGer is also largely unaffected even when the path coverage decreases given that the line coverage is maintained, arguably a prerequisite for any dynamic technique to be effective.

6.2 Semantics Classification

Experiment Setup. We use COSET, a dataset recently proposed by Wang and Christodorescu [27], to perform the semantics classification task. It consists of close to 85K programs developed by a large number of programmers while solving ten coding problems. The challenge for models to resolve is to differentiate a variety of algorithms applied for solving each coding problem (*e.g.* bubble sort vs. insertion sort vs. merge sort). Since we mostly deal with C# and Python programs, we use the Microsoft Roslyn compiler framework and IronPython for extracting a program's abstract syntax tree (AST) and instrumenting the program's source code. To collect execution traces, we run each program with randomly generated inputs. Because all programs in COSET only take inputs of primitive data types, we forgo Randoop and implement a random input generation engine on our own. We remove programs that fail to pass all test cases (*i.e.* crashes or having incorrect results) from the dataset. In the end, we are left with 63,596 programs which we split into a training set containing 45,596 programs, a validation set of 9,000 programs and a test set with the remaining 9,000.

Since LiGer is presented with a classification problem in this setting, we remove decoder from its architecture (Figure 5), and directly feed the learned program embedding to a linear transformation layer. Then, we add a one layer softmax regression to serve the prediction task. The rest of LiGer's architecture is kept intact. For comparison, we use DYPRO [26] as a baseline in this experiment.

Table 3. LiGer's results on COSET.

Models	Accuracy	F1 Score
DYPRO	81.6%	0.81
LiGer	**85.4%**	**0.85**

Results. As depicted in Table 3, LiGer outperforms DYPRO in both accuracy and F1 score. Similar to the experiment documented in Section 6.1.2, we study the performance trend of both models against the decreased number of traces. Specifically, we reduce the number of concrete and symbolic traces while preserving the path and line coverage, and evaluate how the model accuracy vary under both circumstances. As shown in Figure 7, LiGer is far more resilient in weathering the loss of training data, in particular, it achieves slightly better results than DYPRO (82.3% vs. 81.6% in accuracy, and 0.82 vs. 0.81 in F1 score) when trained on almost 10x fewer executions covering nearly 4x fewer program paths (4.7 symbolic traces × 2 concrete traces vs. 18 symbolic traces × 5 concrete traces). To sum up, our results show that LiGer is not only more accurate but also far less data dependent than DYPRO in classifying the program semantics.

6.3 Ablation Study

In this section, we conduct an ablation study to understand the contribution of each component in LiGer's architecture. Since all layers except the fusion layer are essential to LiGer's functionality, our ablation study will only examine the causality among components in the fusion layer, more precisely, the effect of both feature dimensions as well as the attention mechanism that fuses the feature dimensions. We pick the method name prediction task for our ablation study, and evaluate each new configuration of LiGer from the same aspects of model accuracy and data reliance.

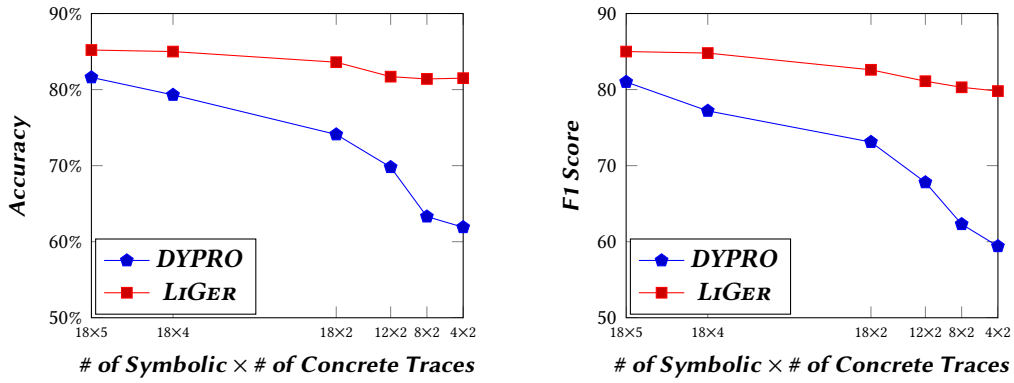

Figure 7. LiGer's performance trend when concrete and symbolic traces are down-sampled.

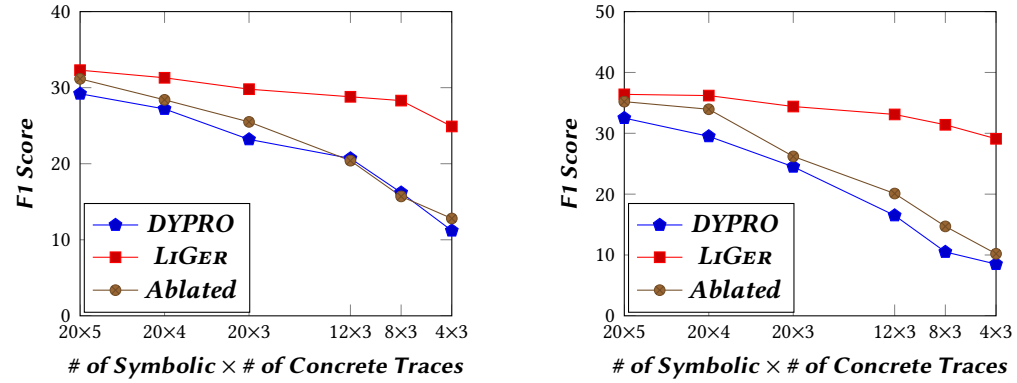

Figure 8. Evaluate LiGer (w/o static feature) when either concrete or symbolic traces are down-sampled.

6.3.1 Removing Static Feature Dimension. First, we remove the symbolic trace from the feature representation. As a result, the component for embedding statements is no longer needed and removed from LiGer's architecture. Note that the resulting configuration is not identical to DYPRO's architecture where representations of concrete traces are learned separately before pooled into a single vector as the program embedding. In contrast, LiGer first learns an embedding for a multitude of concrete traces along the same program path, and then performs a pooling operation among all path embeddings to obtain the program embedding. We reuse the concrete traces for each program in Java-med and Java-large for this experiment.

After removing the static feature dimension, LiGer exhibits almost the same prediction accuracy (31.16 on Java-med and 35.21 on Java-large in F1 score). This indicates that, when given abundant concrete traces to learn, LiGer is able to generalize from the dynamic features alone, therefore, symbolic traces becomes expendable. In addition, even without the static feature dimension, LiGer still significant outperforms both static models. Next, we aim to understand how dependent LiGer becomes on executions after removing the static feature dimension. In particular, we provide LiGer with the same concrete traces that DYPRO was trained and tested on. For brevity, we combine the results of reducing symbolic and concrete traces in the same diagram. As shown in Figures 8, after removing the static feature dimension, LiGer displays a similar performance trend to DYPRO. In other words, LiGer becomes more dependent on program executions, manifested in the significantly poorer results while learning from few concrete traces. This finding reveals that it is the static feature dimension that contributes to the moderate reliance LiGer has on program executions.

6.3.2 Removing Dynamic Feature Dimension. We remove the dynamic feature dimension from LiGer to reveal its contribution to the entire network. Since LiGer is left with the symbolic traces only, each statement in the trace will receive the full attention weight in the fusion layer. Like the prior experiment, we first measure LiGer's F1 score in the method name prediction task.

Removing dynamic feature has a much larger impact on LiGer's precision (20.23/22.95 on Java-med/Java-large). In particular, LiGer becomes a less accurate model than code2seq without the dynamic features. This finding confirms the challenges of learning precise program embeddings from symbolic program features directly. Even though symbolic traces

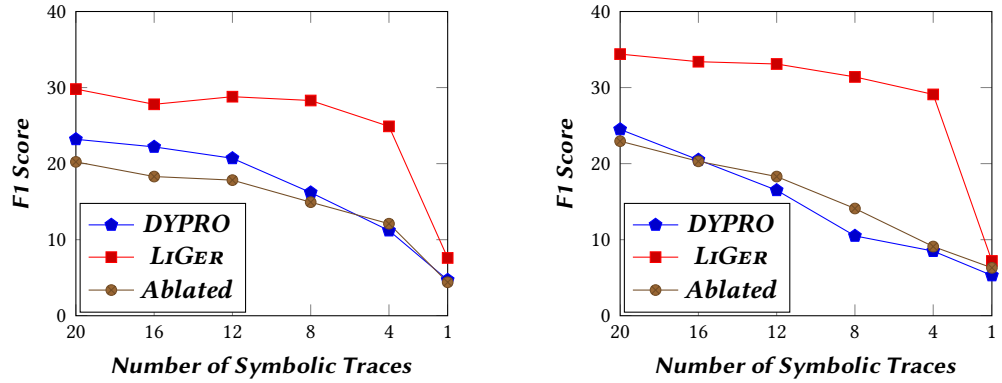

Figure 9. Evaluate LiGer (w/o dynamic feature) when symbolic traces are down-sampled.

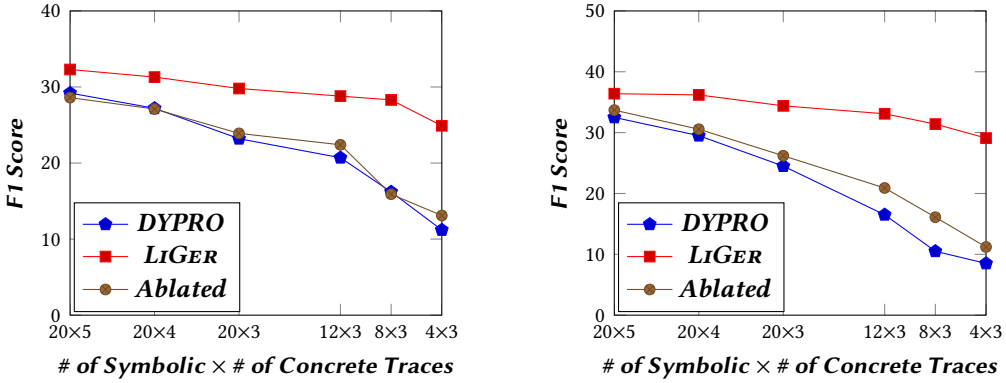

Figure 10. Evaluate LiGer (w/o attention) when either concrete or symbolic traces are down-sampled.

reflect a certain level of the runtime information, LiGer does not manage to learn precise program embeddings. Next, we study how LiGer's reliance on symbolic traces changes.

After removing the dynamic feature dimension, LiGer is still shown quite robust against trace reduction. Even though LiGer starts at a lower F1 score, it outperforms DYPRO as the number of symbolic traces decreases (Figures 9). In general, the accuracy trend LiGer displays correlates well before and after the removal of dynamic features. Thanks to the static feature dimension, LiGer does not suffer a significant accuracy drop.

6.3.3 Removing Attention. Finally, we remove the attention mechanism that controls the fusion of the two feature dimensions. To keep other components intact in the fusion layer, we evenly distribute the weights across all traces in a blended trace.

Removing attention has a notable impact on LiGer, which drop its F1 score from 32.30 (36.42) to 28.63 (33.71) on Java-med (Java-large). This is an unexpected result. As concrete traces are still abundant, an increase in their attention weights should at least leads to a similar performance. We hypothesize that allocating constant weights disrupts the balance LiGer strikes for the two feature dimensions. Although the weights for the dynamic feature increase, the presence of static features limits LiGer's ability to generalize. In terms of its reliance on program executions, LiGer becomes less accurate overall (Figure 10). The explanation is that, without the attention mechanism, symbolic program features will be allocated with lower weights. Therefore, the static feature dimension is unable to issue as strong signals as before to help LiGer learn, thus causing the drop in LiGer's accuracy.

6.3.4 Summary. Finally, we summarize the role of each component in the fusion layer. To provide a direct comparison, we show the results of each new configuration of LiGer on the same diagram (Figure 11).

To summarize, dynamic feature dimension is proven to be the main reason that LiGer is the most precise model in predicting method names. This also explains why LiGer is not a significant upgrade over DYPRO. However, by incorporating the symbolic feature dimension and the attention mechanism that coordinates the feature fusion, LiGer significantly reduces its reliance on program executions. Again, we clarify a sufficient line coverage is assumed, and LiGer is shown to be resilient against the reduction of concrete

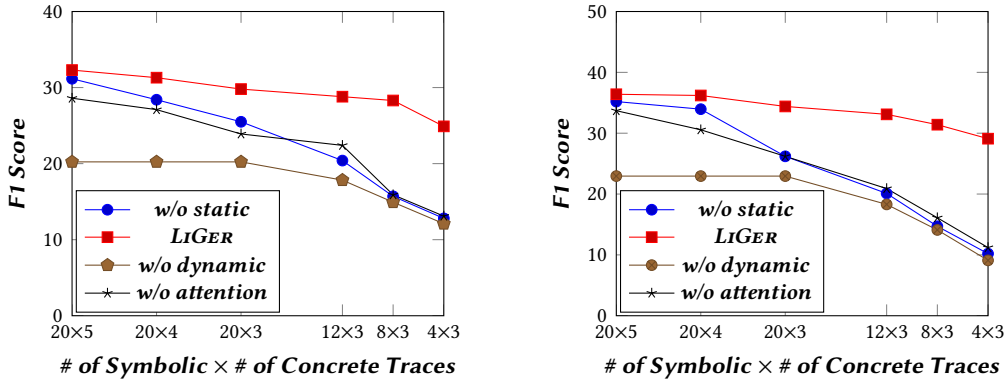

Figure 11. Compare different ablation configurations for LiGer.

traces when path coverage is preserved. More importantly, LiGer is also resilient against the decrease in path coverage when line coverage is preserved.

7 Related Work

This section surveys related work from three aspects: neural program embeddings, attention, and word embeddings.

7.1 Neural Program Embeddings

Recently, learning neural program representations has generated significant interest in the program languages community. As a first step, early methods [13, 21, 23] primarily focus on learning syntactic features. Despite these pioneering efforts, these approaches do not precisely represent program semantics. More recently, a number of new deep neural architectures have been developed to tackle this issue [1, 3, 10, 26, 28]. This line of work can be divided into two categories: dynamic and static. The former [26, 28] learns from concrete program executions, while the latter [1, 3, 10] attempts to dissect program semantics from source code. Unlike these prior efforts, this paper presents an effective blended approach of learning program embeddings from both concrete and symbolic traces.

7.2 Attention

Attention has achieved ground-breaking results in many NLP tasks, such as neural machine translation [5, 25], computer vision [4, 20], image captioning [29] and speech recognition [6, 9]. Attention models work by selectively choosing parts of the input to focus on while producing the output. code2vec is among the most notable that incorporate attention in their neural network architectures. Specifically, they attend over multiple AST paths and assign different weights for each before aggregating them into a program embedding. This paper uses attention to coordinate the combination of the two feature dimensions as well as to decode the method name as a sequence of words.

7.3 Word Embeddings

The seminal work word2vec [18, 19] stimulated the field of learning continuous representation. They propose to embed words into a numerical space where those of similar meanings would appear in close proximity. By embedding words into vectors, they also discover that simple arithmetic operations can reflect the analogies among words (*e.g.* $US - Washington = China - Beijing$). word2vec, along with later efforts on learning representations of sentences and documents [17], have greatly contributed to state-of-the-art results in many downstream tasks [7, 12].

8 Conclusion

This paper has introduced a novel, blended approach of learning program embeddings from the combination of symbolic and concrete execution traces. We demonstrate the effectiveness of our approach by applying it to solve method name prediction and semantics classification task.

Through an extensive evaluation, we have shown that our approach is the most accurate in both tasks, especially it outperforms the state-of-the-art, code2seq, in method name prediction by a wide margin. An important takeaway of our work is that concrete executions when supplied in large quantities help train highly precise models. Symbolic traces, on the other hand, reduce dynamic models' heavy reliance on executions.

For its strong distinct benefits, we believe that our blended approach, when applicable, can be adapted to tackle a wide range of problems in program analysis, program comprehension, and developer productivity.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Madan Musuvathi, for insightful comments. We also thank Ke Wang's Visa Research colleagues, Mihai Christodorescu, Rohit Sinha, Gaven Watson, and Shashank Agrawal for their constructive feedback on the paper.

References

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

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

[3] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. Code2Vec: Learning Distributed Representations of Code. *Proc. ACM Program. Lang.* 3, POPL, Article 40 (Jan. 2019), 29 pages.

[4] Jimmy Ba, Volodymyr Mnih, and Koray Kavukcuoglu. 2015. Multiple object recognition with visual attention. *International Conference on Learning Representations* (2015).

[5] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural machine translation by jointly learning to align and translate. *International Conference on Learning Representations* (2015).

[6] Dzmitry Bahdanau, Jan Chorowski, Dmitriy Serdyuk, Philemon Brakel, and Yoshua Bengio. 2016. End-to-end attention-based large vocabulary speech recognition. In *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 4945–4949.

[7] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin. 2003. A Neural Probabilistic Language Model. *J. Mach. Learn. Res.* 3 (March 2003), 1137–1155. http://dl.acm.org/citation.cfm?id=944919.944966

[8] Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 1724–1734.

[9] Jan K Chorowski, Dzmitry Bahdanau, Dmitriy Serdyuk, Kyunghyun Cho, and Yoshua Bengio. 2015. Attention-based models for speech recognition. In *Advances in neural information processing systems*. 577–585.

[10] Daniel DeFreez, Aditya V. Thakur, and Cindy Rubio-González. 2018. Path-based Function Embedding and Its Application to Error-handling Specification Mining. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. 423–433.

[11] Jacob Devlin, Rabih Zbib, Zhongqiang Huang, Thomas Lamar, Richard Schwartz, and John Makhoul. 2014. Fast and robust neural network joint models for statistical machine translation. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 1370–1380.

[12] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. 2011. Domain Adaptation for Large-scale Sentiment Classification: A Deep Learning Approach. In *International Conference on Machine Learning (ICML)*. 513–520.

[13] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. DeepFix: Fixing Common C Language Errors by Deep Learning. In *Thirty-First AAAI Conference on Artificial Intelligence*.

[14] Jordan Henkel, Shuvendu K. Lahiri, Ben Liblit, and Thomas Reps. 2018. Code Vectors: Understanding Programs Through Embedded Abstracted Symbolic Traces. In *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 163–174.

[15] L. C. Jain and L. R. Medsker. 1999. *Recurrent Neural Networks: Design and Applications* (1st ed.). CRC Press, Inc., USA.

[16] Diederik P Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. *International Conference on Learning Representations* (2015).

[17] Quoc Le and Tomas Mikolov. 2014. Distributed Representations of Sentences and Documents. In *International Conference on Machine Learning*. 1188–1196.

[18] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. *International Conference on Learning Representations* (2013).

[19] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Distributed Representations of Words and Phrases and Their Compositionality. In *Neural Information Processing Systems (NIPS)*. 3111–3119.

[20] Volodymyr Mnih, Nicolas Heess, Alex Graves, et al. 2014. Recurrent models of visual attention. In *Advances in neural information processing systems*. 2204–2212.

[21] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional neural networks over tree structures for programming language processing. In *Thirtieth AAAI Conference on Artificial Intelligence*.

[22] Carlos Pacheco and Michael D. Ernst. 2007. Randoop: Feedback-Directed Random Testing for Java. In *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion (OOPSLA '07)*. Association for Computing Machinery.

[23] Yewen Pu, Karthik Narasimhan, Armando Solar-Lezama, and Regina Barzilay. 2016. Sk_P: A Neural Program Corrector for MOOCs. In *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity (SPLASH)*. 39–40.

[24] Kai Sheng Tai, Richard Socher, and Christopher D. Manning. 2015. Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Association for Computational Linguistics.

[25] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*. 5998–6008.

[26] Ke Wang. 2019. Learning Scalable and Precise Representation of Program Semantics. *arXiv preprint arXiv:1905.05251* (2019).

[27] Ke Wang and Mihai Christodorescu. 2019. COSET: A Benchmark for Evaluating Neural Program Embeddings. *arXiv preprint arXiv:1905.11445* (2019).

[28] Ke Wang, Rishabh Singh, and Zhendong Su. 2018. Dynamic Neural Program Embedding for Program Repair. *International Conference on Learning Representations* (2018).

[29] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhudinov, Rich Zemel, and Yoshua Bengio. 2015. Show, attend and tell: Neural image caption generation with visual attention. *International Conference on Machine Learning* (2015).