A Theory of Semantic Program Embeddings

Jesse Michel
Massachusetts Institute of Technology
jmmichel@mit.edu

Logan Weber
Massachusetts Institute of Technology
loganweb@mit.edu

Saman Amarasinghe
Massachusetts Institute of Technology
samana@mit.edu

Michael Carbin
Massachusetts Institute of Technology
mcarbin@mit.edu

Abstract

Models that embed programs as vectors are commonly used to predict properties of programs. A common desiderata of a learned program embedding space is that similar programs are close in the embedding space. We formalize the notion of semantic similarity in terms of metrics over functions and embeddings and define a corresponding optimization problem. Interpreting embeddings as coefficients with respect to an orthonormal basis, we derive an optimal embedding with respect to an upper bound of the optimization objective and show how it may be efficiently approximated. We compare our implementation of this embedding to a neural program embedding. To make this comparison, we construct a data set from pairs of basic blocks found in numerical libraries such as TensorFlow and FFMPEG. We show that a transformer-based model can learn a limited amount about the semantics of assembly, but there is still a performance gap between it and the numerical baseline.

1 Introduction

Recent work uses natural language processing techniques to take in the syntax of a program and solve tasks such as method name prediction, variable misuse detection, and autocompletion [Allamanis et al., 2015, Alon et al., 2019, Raychev et al., 2014].

A key concept that many of these techniques develop is that of a *program embedding*: a real-valued vector representation of the program designed to capture elements of the syntax and semantics of the program [Allamanis et al., 2016, Mou et al., 2016, Allamanis et al., 2018, Wang et al., 2020]. These techniques first map the text of the program to an embedding and then use the embedding as input to a neural network or other computation to solve a given end task. For example, Jain et al. [2020] develop and leverage a program embedding to measure the similarity of two programs, which can then be used in the implementation of a code search tool that takes as input a query program and returns the most similar programs within a database of programs. Specifically, Jain et al. [2020] designed the embeddings such that the cosine similarity of the embeddings of two programs provides a measure of the syntactic and semantic similarity between them. They show that this approach learns embeddings that are more robust to label-preserving code edits than other approaches.

To date, work in the area of program embeddings has only informally suggested that program embeddings capture the semantics in the form of the true input-output behavior of the program [Alon et al., 2019, Wang and Su, 2020, Wang et al., 2020]. However, there are clear instances in which the program embeddings generated by existing techniques do not capture basic properties that follow from the semantics of a program [Srikant et al., 2021]. For example, the embeddings generated by several techniques [Srikant et al., 2021, Alon et al., 2019, Kanade et al., 2020] can change in value

Preprint. Under review.

depending on the variable names of the program despite the fact that renaming all occurrences of a variable with a new variable name yields a semantically equivalent program.

In this work, we develop a formal semantics of *semantic program embeddings* and investigate if it is possible to leverage natural language processing techniques to learn semantic program embeddings.

We motivate our definition of semantic program embeddings by the fact that programs denote mathematical functions (e.g., a numerical program denotes a real-valued function). We therefore define the semantic program embedding of a program such that the distance between the embedding of two programs is the same as the distance between the input-output behavior of the two program's respective functions. We formalize this relationship using metric spaces. Specifically, a semantic program embedding is designed to minimize the discrepency between:

$$d(\llbracket p \rrbracket, \llbracket q \rrbracket) \text{ and } d_e(e(p), e(q))$$

where p, q are numerical programs, $\llbracket p \rrbracket$ is the function denoted by p, e is an embedding function, and d, d_e compute the distance between functions and embeddings respectively. We frame the problem of finding an embedding function, e, as an optimization problem and propose a mathematically justified way to build a baseline semantic embedding function.

We compare the baseline semantic embedding function and a transformer-based model trained on the semantic embedding optimization problem. To do this, we create a dataset of x86 basic blocks from numerical programs taken from the BHive benchmark suite, which includes, for example, TensorFlow benchmarks [Chen et al., 2019]. We then explore the trade-offs between different embedding techniques both quantitatively and qualitatively.

Our contributions are as follows:

1. We define semantic program embeddings as a real vector modeling the semantics of a program and pose the semantic embedding optimization problem.
2. We present a numerical baseline semantic embedding and mathematically justify it.
3. We create a dataset of x86 basic blocks from numerical programs that we use to train a transformer-based model. We show that a transformer-based model can learn a limited amount about the semantics of assembly, but there is still a performance gap between it and the numerical baseline.

2 Related Work

Just as embeddings of natural language have enabled a wide variety of new tasks in natural language processing, embeddings of programs have enabled a wide variety of tasks that are either wholly new or have been difficult to accomplish using traditional symbolic program reasoning techniques. Such tasks include code completion [Raychev et al., 2014], code search [Raychev et al., 2014], type inference [Hellendoorn et al., 2018], method name prediction [Alon et al., 2019], documentation generation [Kanade et al., 2020], and code deobfuscation [Vasilescu et al., 2017]. Program embeddings have partially alleviated the need to develop handcrafted features of programs. Instead, learning-based systems are able to rely on representation learning to produce new state-of-the-art results in each of these areas.

Program Embedding Techniques. Researchers have applied deep learning models to program embeddings. Early work used models such as RNNs[Raychev et al., 2014]. Architectures using contexts, convolutions, attention, and neural networks progressed in developing richer representations of programs [Allamanis et al., 2015, 2016, Mou et al., 2016, Allamanis et al., 2018]. Other work, decomposes the AST into paths that are attended over to match the behavior of execution traces [Alon et al., 2019]. Increased specialization of architectures to code structures such as loops, and branching statements have further improved performance [Wang et al., 2020]. By combining information from execution traces with syntax, researchers have improved the expressiveness of their models [Wang and Su, 2020]. Building on the successes in natural language modeling, researchers have pretrained large transformer-based language models on program text [Kanade et al., 2020, Jain et al., 2020].

Semantic Program Embeddings. Recent work has begun to articulate the informal concept that program embeddings should capture the semantics of programs [Alon et al., 2019]. Contemporary

wisdom primarly roots the property of capturing the semantics of a program in the idea that embeddings should be robust to semantics-preserving changes in their syntax [Wang and Su, 2020, Wang et al., 2020, Jain et al., 2020]. Examples of such transformations include variable renaming, semantics preserving compiler optimizations (such as eliminating *dead code* – code that has no effect on the observable behavior of the program [Jain et al., 2020]), and structural transformations (such as replacing looping constructs with semantically equivalent recursive counterparts). The demonstrated benefit is that embeddings that better capture the semantics will return better predictions that are robust to irrelevant changes in the syntactic representation of the program. However, to date, no work has given a formal semantics to what it means for an embedding to be semantic beyond the passing phrase that denotationally similar programs should have similar embeddings [Wang et al., 2020].

3 A Theory of Semantic Program Embeddings

In this section, we concretize the statement that the embeddings of similar programs are similar. For convenience, we consider arbitrary programs $p, q \in \mathcal{P}$ throughout this section. We begin by covering the definition of a metric space, which we then use to formalize similarity for both functions and embeddings. We conclude by posing an optimization objective that measures to what degree an embedding function produces semantic embeddings.

3.1 Metric spaces

A metric space is a pair (X, d_X) where X is a set and $d_X : X \times X \to \mathbb{R}$ is a metric, which satisfies the following properties for all $x, y, z \in X$:

1. *identity of indiscernibles*: $d_X(x, y) = 0$ if and only if $x = y$
2. *symmetry*: $d_X(x, y) = d_X(y, x)$
3. *triangle inequality*: $d_X(x, z) \leq d_X(x, y) + d_X(y, z)$

3.2 Metrics on functions

We say two programs p, q are similar when the distance between their denotations $[\![p]\!], [\![q]\!] : \mathbb{R}^n \to \mathbb{R}^m$ is small with respect to some metric. A common class of metrics on functions is induced by the L_p norms, which we refer to as L_k norms to avoid conflict with the notation for programs $p \in \mathcal{P}$. For a fixed k and continuous support S, the distance $d([\![p]\!], [\![q]\!])$ induced by the $L_k(S)$ norm is defined as follows:

$$d([\![p]\!], [\![q]\!]) = \|[\![p]\!] - [\![q]\!]\|_k = \left(\int_{\vec{x} \in S} |[\![p]\!](\vec{x}) - [\![q]\!](\vec{x})|^k \right)^{1/k}$$

In order to express a distance in this way, the set of denotations of programs must be representable as a metric space and the domain S must be measurable.

3.3 Metrics on embeddings

Now that we have defined a notion of similarity for programs, we define similarity for embeddings. A program embedding is a vector of real numbers $\vec{c} \in \mathbb{R}^n$. We say two embeddings are similar when the distance between them is small with respect to some metric. An embedding function is a function $e : \mathcal{P} \to \mathbb{R}^n$ that accepts a program as input and produces a program embedding. A metric over embeddings is a function $d_e(e(p), e(q)) : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$. Analogous to L_k norms on function spaces, the metrics induced by ℓ_p (which we refer to as ℓ_k) norms are commonly used for comparing finite-dimensional vectors. For a fixed k, the distance $d_e(e(p), e(q))$ induced by the $\ell_k(\mathbb{R})$ norm is defined as follows:

$$d_e(e(p), e(q)) = \|e(p) - e(q)\|_k = \left(\sum_{i=0}^{n} |e(p) - e(q)|^k \right)^{1/k}$$

3.4 Semantic metric embedding as optimization

Our goal in the design of a semantic metric embedding is to have the metric on functions and the metric on embeddings match as closely as possible. In complete generality, we pose the *semantic*

embedding objective as the sum of the squared errors of all program pairs $p, q \in \mathcal{P}$. The *semantic embedding optimization problem* is then a minimization of this objective over the set of embedding functions e:

$$\min_{e} \sum_{p,q \in \mathcal{P}} (d_e(e(p), e(q)) - d([\![p]\!], [\![q]\!]))^2 \tag{1}$$

where d and d_e are fixed. We present two approaches to solving this problem: (1) a baseline numerical technique that estimates the optimal solution for an upper bound of the loss (Section 4) and (2) a deep neural network approach that trains on data to achieve the objective (Section 5).

4 Numerical Baseline Embeddings

In this section, we introduce an interpretation for an embedding vector with respect to a basis and distances between embedding vectors. With this structure, we are able to characterize optimal embeddings with respect to an upper bound of the objective function. Using standard numerical techniques, we show how to efficiently estimate this optimal embedding. We use this approach as a baseline to compare with a transformer-based model.

4.1 Embedding with respect to a basis

We interpret a programming embedding by viewing each element of the vector as a coefficient of a function. Concretely, we select a linearly independent basis $\mathcal{B}$ that expresses programs such that extracting coefficients over the basis from a program is tractable and such that $[\![p]\!] = e(p) \cdot \vec{B}$. Note that linear independence ensures embeddings are unique and means the embeddings satisfy the additional property that $e(p) = e(q)$ if and only if $[\![p]\!] = [\![q]\!]$.

It is not possible to perfectly embed all programs for a Turing-complete language, because if this were the case, we could use the embedding to solve the halting problem. As a result, it is desirable to use an approximate embedding function $\tilde{e}$ such that $\tilde{[\![p]\!]} = \tilde{e}(p) \cdot \vec{B}$ and $d([\![p]\!], \tilde{[\![p]\!]}) < \epsilon$ for some $\epsilon > 0$. In this case, either the basis may be such that it cannot represent all functions $\mathcal{P}$ or the formula over coefficients may not be tractable to compute.

4.2 Optimal Embeddings for an Upper Bound

The optimization problem in Equation 1 was:

$$\min_{e} \sum_{p,q \in \mathcal{P}} (d_e(e(p), e(q)) - d([\![p]\!], [\![q]\!]))^2$$

If we assume that d is the L1 norm, d_e is the L1 norm with respect to the basis, and S is the set to be optimized over, then by Lemma 4.1 (with $a(x) = e(p) \cdot \vec{B}(x) - e(q) \cdot \vec{B}(x)$ and $b(x) = [\![p]\!](x) - [\![q]\!](x)$)

$$\min_{e} \sum_{p,q \in \mathcal{P}} \int_{x \in S} (e(p) \cdot \vec{B}(x) - e(q) \cdot \vec{B}(x) - ([\![p]\!](x) - [\![q]\!](x)))^2 \tag{2}$$

is greater than or equal to Equation 1 for any e.

Lemma 4.1. Let $a, b : \mathbb{R} \to \mathbb{R}$ then

$$\left(\int_{x \in S} |a(x)| - \int_{x \in S} |b(x)| \right)^2 \leq \int_{x \in S} (a(x) - b(x))^2$$

Proof.

$$\begin{aligned}
\left(\int_{x \in S} |a(x)| - \int_{x \in S} |b(x)| \right)^2 &= \left(\int_{x \in S} |a(x)| - |b(x)| \right)^2 \\
&\leq \int_{x \in S} (|a(x)| - |b(x)|)^2 \quad \text{by Jensen's inequality} \\
&\leq \int_{x \in S} (a(x) - b(x))^2 \quad \text{since } a(x)b(x) \leq |a(x)||b(x)| \ \forall x \in S
\end{aligned}$$

□

We compute the minimum of Equation 2 by taking its derivative and setting it to 0. Simplifying and using orthonormality, we see that:

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial e(p)_k} &= 2 \cdot 2 \sum_{q \in \mathcal{P}} \int_{x \in S} \left[(e(p) - e(q)) \cdot \vec{B}(x) - (\llbracket p \rrbracket(x) - \llbracket q \rrbracket(x)) \right] \vec{B}_k(x) \\ &= 4 \sum_{q \in \mathcal{P}} \int_{x \in S} (e(p)_k - e(q)_k) \vec{B}_k^2(x) - (\llbracket p \rrbracket(x) - \llbracket q \rrbracket(x)) \vec{B}_k(x) \\ &= 4 \sum_{q \in \mathcal{P}} (e(p)_k - e(q)_k) - \langle \llbracket p \rrbracket - \llbracket q \rrbracket, B_k \rangle\end{aligned}$$

where $\langle h_1, h_2 \rangle = \int_{x \in S} h_1(x) h_2(x)$ is an inner product. Setting the result to zero shows that an optimal embedding should satisfy:

$$\sum_{q \in \mathcal{P}} e(p)_k - e(q)_k = \sum_{q \in \mathcal{P}} \langle \llbracket p \rrbracket - \llbracket q \rrbracket, B_k \rangle$$

for $k = 1, 2, \ldots, n$ where n is the embedding size. We can derive an embedding that satisfies this constraint

$$e(p)_k = \langle \llbracket p \rrbracket, B_k \rangle \tag{3}$$

by linearity of the inner product. This definition of an embedding also coincides with a solution to the least squared problem:

$$\min_e \int_{x \in S} (e(p) \cdot \vec{B}(x) - \llbracket p \rrbracket(x))^2$$

In the case of a polynomial basis up to some bounded degree, the weight-basis vector products correspond to Legendre polynomials. Similarly, the weight-basis construction coincides with the computation of the Fourier coefficients for the Fourier basis.

4.3 Computing the Baseline Embedding

Thus far, the solution we presented is a mathematical formula. We now describe how to compute an estimate of our solution to the embedding problem.

Orthonormal basis. In this regime, in order to compute an embedding, we must first select a set of basis functions. The choice of basis functions has implications on the quality of an approximation and on the rate of convergence in the limit of embedding size.

Embedding computation. The optimal embedding is mathematically expressed by the integral:

$$e(p)_k = \int_{x \in S} \llbracket p \rrbracket(x) B_k(x)$$

for $k = 1, 2, \ldots, n$. Computing this result exactly is generally infeasible because there is no guarantee of a closed-form solution to integrals for arbitrary numerical programs.

We list three techniques to estimate this integral. If an executable is provided, integral estimation techniques such as quadrature and Monte Carlo apply. If the derivative is also provided, then more advanced techniques may be applied such as the Hamiltonian Monte Carlo algorithm. Given only the syntax of a program, the embedding will be a map from the syntax to a functional estimation. Transformer-based language models are a natural candidate model for this task [Vaswani et al., 2017].

5 Evaluation

In our evaluation we answer the following research questions:

RQ1: Can a neural network learn to semantically embed programs using only the program text (when compared to outputting the mean distance)?

RQ2: How does the mean loss of the neural network compare to the numerical baseline?

RQ3: What are the qualitative differences between the neural network and the numerical baseline?

5.1 Methodology

Neural Network Approach. We optimize the semantic embedding objective via gradient descent on a loss function parameterized by embeddings of program pairs generated by a neural network and the distance between their denotations:

$$\sum_{p,q \in \mathcal{P}} (d_e(e(p), e(q)) - d([\![p]\!], [\![q]\!]))^2 \tag{4}$$

For the evaluation, we instantiate the program space $\mathcal{P}$ with a collected dataset of x86 basic blocks and we form denotations of these blocks by considering their floating-point inputs and outputs. We set the distance in the embedding space to be the ℓ_1 norm

$$d_e(\vec{u}, \vec{v}) = \|\vec{u} - \vec{v}\|_1$$

which may also be written as $\sum_i |u_i - v_i|$. The distance over functions is the L_1 norm with ℓ_1 norm on the output space:

$$d([\![p]\!], [\![q]\!]) = \int_{\vec{x} \in S} \|[\![p]\!](\vec{x}) - [\![q]\!](\vec{x})\|_1 \tag{5}$$

Dataset. We collect a dataset of numerical benchmarks that we will embed. BHive is a benchmark suite of 330018 basic blocks that includes data from OpenBLAS, TensorFlow, etc. [Chen et al., 2019]. We use AT&T syntax for the textual representation of these basic blocks. We select the 2969 unique basic blocks that manipulate floating point numbers (those that use either single- or double-precision floating point or SIMD instructions). To execute and analyze basic blocks, we use the STOKE infrastructure [Schkufza et al., 2013].

These numerical basic blocks do not explicitly specify inputs or outputs. To determine the inputs and outputs, we first identify the live-ins (registers used before being assigned to) and def-outs (registers assigned to) of each block. We select the floating point registers thereof as the inputs and outputs, respectively.

Most of the basic block code references memory, so as-is, they would either fail to execute or use undefined values from memory. We replace these memory references with the immediate value 0.

SIMD floating-point registers (e.g., `%ymm*` and `%xmm*`) can contain single- or double-precision values, so the expected precision for a register in a basic block must be determined before we can determine the basic block's signature. We identify a block as single-precision or double-precision by checking whether the suffix of the first floating point instruction in the block ends with "s" (e.g., `vsubps`) or "d" (e.g., `vmulsd`), corresponding to single- and double-precision respectively. We use this precision assignment a specification for the number of inputs and outputs represented by separate variables. Some basic blocks (258 of the 2969) involve mixed-precision computations, which may lead to unspecified executions. Next, we will discuss how to filter problematic blocks such as these.

Without manually inspecting a block, we do not know whether it is numerically well-behaved with respect to our construction of a denotation. To filter blocks with degenerate behavior, we run each basic block with each input sampled uniformly at random from $[-0.5, 0.5]$. If any outputs are `NaN` or ∞, we discard the block. Averaging over 10 trials, this procedure discards ≈ 630 basic blocks with std. dev. ≈ 4, leaving 2339 basic blocks.

The basic blocks in the dataset have different numbers of floating-point inputs and outputs. Computing the distance between pairs requires operating over the same number of inputs to share samples and the same number of outputs to compute the ℓ_1 norm. To resolve this issue, we pad the inputs and outputs with zeros to a fixed size so that all functions map from $\mathbb{R}^n \to \mathbb{R}^m$. For example, if we want to extend $f(x) = x^2$ to have two inputs and two outputs, we would transform it to $f(x_1, x_2) = (x_1^2, 0)$.

To compute the distance between pairs, we need to evaluate the integral in Equation 5. Since the dimensionality of the integral can be large when there are many inputs, we opt for Monte Carlo integration. We evaluate each pair of basic blocks at 1000 random samples on the support $[-0.5, 0.5]^n$, where n is the number of inputs.

To generate data for the semantic embedding loss, we partition the data into two random groups and subsample pairs (to compute distances between them) randomly without replacement. We build train and test sets of 40000 and 10000 programs respectively. To guarantee the problem is tractable we

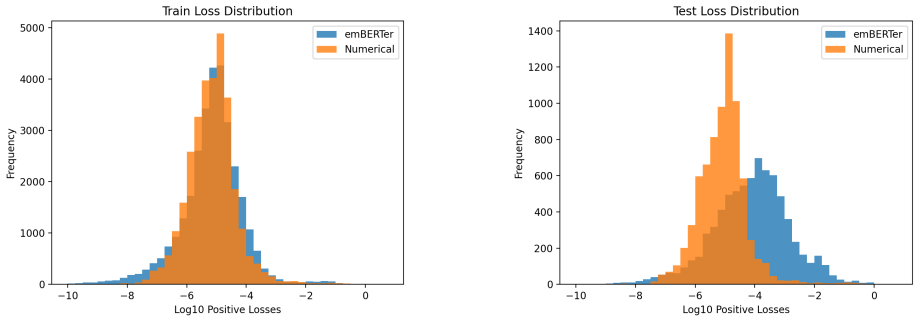

Figure 1: Train and test loss distributions for emBERTer and the numerical baseline.

bound the distances at 1000, which is also convenient for normalizing the data. To avoid outliers with many inputs and outputs, we discard programs with inputs or outputs more than one standard deviation above the mean (36 inputs and 52 outputs).

In the end, the dataset consists of 31,199 rows of training data and 7,562 rows of test data. Each row of the dataset is a pair of basic blocks and an estimated functional distance between basic blocks.

Model. Kanade et al. [2020] showed that the BERT (large) is capable of learning program embeddings that perform well across a number of syntactic tasks. Since our dataset is small when compared to the dataset size that BERT is trained on, we opt for a smaller model – TinyBERT [Jiao et al., 2020]. We use the `transformers` library [Wolf et al., 2020] and modify the `BertForSequenceClassification` model by having the forward pass return the logits corresponding to the desired embedding size. We normalize the distances so that the neural network need only predict a value between 0 and 1. We also remove all dropout from the model. We then train the model on the loss in Equation 4 for 100 epochs with a batch size of 512 on a NVIDIA Tesla v100 GPU, which takes about an hour.

Naive Baselines. To gain intuition for what constitutes a good loss on the semantic embedding objective, we include several baselines which produce the same output for every input. We set this constant to be a few different statistics: the mean, median, and mode of $d(\llbracket p \rrbracket, \llbracket q \rrbracket)$ over all program pairs in the *training* set. The values of these quantities are as follows:

Statistic	Value
Mean	10.20
Median	3.03
Mode	0

Numerical Baseline. We implemented the numerical baseline in Python and parallelized the execution. We set the embedding size to 1000 and used all 1000 samples of input/output pairs. The implementation ran to completion in about 2 hours on the training set and 1 hour on the test set on a computer with 224 CPUs and 224GB of memory.

5.2 Results

We use the same embedding size of 1000 for the numerical baseline and EmBERTer. Table 5.2 presents the train and test losses for each of the techniques we use to solve the semantic embedding problem.

RQ1. EmBERTer outperforms all of the statistical baselines on both the training and test set. As a result, we say that the neural network is able to learn a semantic embedding. However, it does not learn to perfectly semantically embed.

RQ2. Although EmBERTer is able to to achieve lower training loss than the numerical baseline, the test loss is significantly worse.

Model	Train Loss	Test Loss
Mean baseline	2962	4812
Median baseline	3013	4905
Mode baseline	3065	4970
Numerical baseline	747	1117
EmBERTer	612	4172

RQ3. To explore the characteristics of emBERTer and the numerical baseline, we analyze the loss distributions on the train and test set shown in Figure 5.1. The variance of the loss is lower for the numerical baseline than for the neural network. Note that we remove zero loss points in these plots because the log is undefined at 0. A further analysis of the pairs of programs with loss 0 predictions by both models shows that these are precisely the pairs that are syntactically equivalent. Thus, we conclude that both models correctly identify syntactically equivalent programs.

5.3 Discussion

Both the numerical baseline and the neural network are able to outperform the simple baseline, meaning that there's a payoff in performance for the increase in compute and complexity for both models. The fact that the numerical technique performs better than the neural network demonstrates that even in the absence of control flow and function calls, learning the semantics of programs from extrinsic measures (e.g., $d(\llbracket p \rrbracket, \llbracket q \rrbracket)$) is a challenging task.

There are a number of trade-offs between using the numerical baseline and the neural network. For one, the numerical baseline and the neural network have different inputs. The numerical baseline is obtained from input/output pairs $(x, \llbracket p \rrbracket(x))$, which are used to estimate the inner product $\langle \llbracket p \rrbracket, B_k \rangle$; the neural network uses triples $(p, q, d(\llbracket p \rrbracket, \llbracket q \rrbracket))$. In particular, the EmBERTer model receives the 1000 sample Monte Carlo estimate of the integral $d(\llbracket p \rrbracket, \llbracket q \rrbracket)$, whereas the numerical baseline receives all of the input/output pairs for the integrals it estimates. The numerical baseline requires repeated execution of the program, which may be time-consuming. However, the representation is interpretable – coefficients over a polynomial basis. Both techniques require significant computational resources, but parallelization on CPUs is more natural for the numerical technique. In contrast, EmBERTer operates over text and the primary interpretation of the embedding is with respect to other embeddings. GPUs are better match than CPUs for executing EmBERTer.

6 Limitations

We present the semantics of programming embeddings and evaluate a numerical technique and a transformer-based model on the semantic embedding optimization problem. We evaluate these models on basic block assembly code, which lacks control flow and lacks loop structures. To semantically model program in general, it is necessary to model both control flow and loops. On a similar note, we only model floating-point inputs and outputs of basic blocks. In practice, programs operate over different data types (e.g., integers, booleans, etc.) and different data structures – a program may take sets, higher-order functions, etc. as inputs or return them as outputs.

7 Conclusions

In this paper, we defined the semantic program embeddings problem and proposed a corresponding semantic embedding optimization problem. Although it is impossible to embed of arbitrary programs from a Turing complete language, we show both a numerical and mathematically justified way and an transformer-based natural language technique. Compare these techniques we introduce a dataset of x86 numerical programs. We find that a transformer-based model can learn a limited amount about the semantics of assembly, but there is still a performance gap between it and the numerical baseline.

8 Societal Impacts

We do not believe that this research poses significant risk of societal harm. This research aims to lay a foundation for formally understanding the semantics of programming embeddings and for evaluating

models in the future. Our evaluation uses a TinyBERT model Jiao et al. [2020], which requires non-trivial training resources and our work could inspire future work that uses a significant amount of resources to train large models on the program embedding task.

References

Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. Suggesting accurate method and class names. In *Foundations of Software Engineering*, 2015.

Miltiadis Allamanis, Hao Peng, and Charles Sutton. A convolutional attention network for extreme summarization of source code. In *International Conference on Machine Learning*, 2016.

Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. In *International Conference on Learning Representations*, 2018.

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. Code2vec: Learning distributed representations of code. In *Principles of Programming Languages*, 2019.

Yishen Chen, Ajay Brahmakshatriya, Charith Mendis, Alex Renda, Eric Atkinson, Ondrej Sykora, Saman Amarasinghe, and Michael Carbin. Bhive: A benchmark suite and measurement framework for validating x86-64 basic block performance models. In *IEEE International Symposium on Workload Characterization*, 2019.

Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. Deep learning type inference. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018.

Paras Jain, Ajay Jain, Tianjun Zhang, Pieter Abbeel, Joseph E. Gonzalez, and Ion Stoica. Contrastive code representation learning. *arXiv preprint*, 2020.

Xiaoqi Jiao, Yichun Yin, Lifeng Shang, Xin Jiang, Xiao Chen, Linlin Li, Fang Wang, and Qun Liu. TinyBERT: Distilling BERT for natural language understanding. In *Empirical Methods in Natural Language Processing*, 2020.

Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. Learning and evaluating contextual embedding of source code. In *International Conference on Machine Learning*, 2020.

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

Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. In *Programming Language Design and Implementation*, 2014.

Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In *Architectural Support for Programming Languages and Operating Systems*, 2013.

Shashank Srikant, Sijia Liu, Tamara Mitrovska, Shiyu Chang, Quanfu Fan, Gaoyuan Zhang, and Una-May O'Reilly. Generating adversarial computer programs using optimized obfuscations. In *International Conference on Learning Representations*, 2021.

Bogdan Vasilescu, Casey Casalnuovo, and Premkumar T. Devanbu. Recovering clear, natural identifiers from obfuscated JS names. In *Foundations of Software Engineering*, 2017.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, 2017.

Ke Wang and Zhendong Su. Blended, precise semantic program embeddings. In *Programming Language Design and Implementation*, 2020.

Yu Wang, Ke Wang, Fengjuan Gao, and Linzhang Wang. Learning semantic program embeddings with graph interval neural network. In *Object-Oriented Programming, Systems, Languages, and Applications*, 2020.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-art natural language processing. In *Empirical Methods in Natural Language Processing: System Demonstrations*, 2020.

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes] Section 3, 4, 5 justify our three core claims.
   (b) Did you describe the limitations of your work? [Yes] See Section 6.
   (c) Did you discuss any potential negative societal impacts of your work? [Yes] See Section 8.
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [Yes] See Section 4.2.
   (b) Did you include complete proofs of all theoretical results? [Yes] See Section 4.2.
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [No] We will open-source add all code and data before the camera ready submision deadline.
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes] Section 5.1
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [No] We had insufficient time to develop and incorporate a methodology including different random seeds. However, we found that our results we robust for multiple runs.
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes] See Section 5.1.
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes]
   (b) Did you mention the license of the assets? [No] They are well-known libraries with permissive licenses (e.g., `transformers`).
   (c) Did you include any new assets either in the supplemental material or as a URL? [No] We will before the camera ready submission deadline.
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A]
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A]
5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]