Learning and Evaluating Contextual Embedding of Source Code

Aditya Kanade [*,1,2] **Petros Maniatis** [*,2] **Gogul Balakrishnan** [2] **Kensen Shi** [2]

Abstract

Recent research has achieved impressive results on understanding and improving source code by building up on machine-learning techniques developed for natural languages. A significant advancement in natural-language understanding has come with the development of pre-trained contextual embeddings, such as BERT, which can be fine-tuned for downstream tasks with less labeled data and training budget, while achieving better accuracies. However, there is no attempt yet to obtain a high-quality contextual embedding of source code, and to evaluate it on multiple program-understanding tasks simultaneously; that is the gap that this paper aims to mitigate. Specifically, first, we curate a massive, deduplicated corpus of 6M Python files from GitHub, which we use to pre-train CuBERT, an open-sourced code-understanding BERT model; and, second, we create an open-sourced benchmark that comprises five classification tasks and one program-repair task, akin to code-understanding tasks proposed in the literature before. We fine-tune CuBERT on our benchmark tasks, and compare the resulting models to different variants of Word2Vec token embeddings, BiLSTM and Transformer models, as well as published state-of-the-art models, showing that CuBERT outperforms them all, even with shorter training, and with fewer labeled examples. Future work on source-code embedding can benefit from reusing our benchmark, and comparing against CuBERT models as a strong baseline.

1. Introduction

Modern software engineering places a high value on writing clean and readable code. This helps other developers understand the author's intent so that they can maintain and extend the code. Developers use meaningful identifier names and natural-language documentation to make this happen (Martin, 2008). As a result, source code contains substantial information that can be exploited by machine-learning algorithms. Indeed, sequence modeling on source code has been shown to be successful in a variety of software-engineering tasks, such as code completion (Hindle et al., 2012; Raychev et al., 2014), source code to pseudocode mapping (Oda et al., 2015), API-sequence prediction (Gu et al., 2016), program repair (Pu et al., 2016; Gupta et al., 2017), and natural language to code mapping (Iyer et al., 2018), among others.

The distributed vector representations of tokens, called token (or word) embeddings, are a crucial component of neural methods for sequence modeling. Learning useful embeddings in a supervised setting with limited data is often difficult. Therefore, many unsupervised learning approaches have been proposed to take advantage of large amounts of unlabeled data that are more readily available. This has resulted in ever more useful pre-trained token embeddings (Mikolov et al., 2013a; Pennington et al., 2014; Bojanowski et al., 2017). However, the subtle differences in the meaning of a token in varying contexts are lost when each word is associated with a single representation. Recent techniques for learning contextual embeddings (McCann et al., 2017; Peters et al., 2018; Radford et al., 2018; 2019; Devlin et al., 2019; Yang et al., 2019) provide ways to compute representations of tokens based on their surrounding context, and have shown significant accuracy improvements in downstream tasks, even with only a small number of task-specific parameters.

Inspired by the success of pre-trained contextual embeddings for natural languages, we present the first attempt to apply the underlying techniques to source code. In particular, BERT (Devlin et al., 2019) produces a bidirectional Transformer encoder (Vaswani et al., 2017) by training it to predict values of masked tokens and whether two sentences follow each other in a natural discourse. The pre-trained model can be fine-tuned for downstream supervised tasks and has been shown to produce state-of-the-art results on a number of natural-language understanding benchmarks. In this work, we derive a contextual embedding of source code by training a BERT model on source code. We call our model CuBERT, short for *Code Understanding BERT*.

[*]Equal contribution [1]Indian Institute of Science, Bangalore, India [2]Google Brain, Mountain View, USA. Correspondence to: Aditya Kanade <kanade@iisc.ac.in>, Petros Maniatis <maniatis@google.com>.

Proceedings of the 37^{th} International Conference on Machine Learning, Vienna, Austria, PMLR 119, 2020.

In order to achieve this, we curate a massive corpus of Python programs collected from GitHub. GitHub projects are known to contain a large amount of duplicate code. To avoid biasing the model to such duplicated code, we perform deduplication using the method of Allamanis (2018). The resulting corpus has 6.6M unique files with a total of 2 billion words. We also train Word2Vec embeddings (Mikolov et al., 2013a;b), namely, continuous bag-of-words (CBOW) and Skipgram embeddings, on the same corpus.

For evaluating CuBERT, we create a benchmark of five classification tasks, and a sixth localization and repair task. The classification tasks range from classification of source code according to presence or absence of certain classes of bugs, to mismatch between a function's natural language description and its body, to predicting the right kind of exception to catch for a given code fragment. The localization and repair task, defined for variable-misuse bugs (Vasic et al., 2019), is a pointer-prediction task. Although similar tasks have appeared in prior work, the associated datasets come from different languages and varied sources; instead we create a cohesive multiple-task benchmark dataset in this work. To produce a high-quality dataset, we ensure that there is no overlap between pre-training and fine-tuning examples, and that all of the tasks are defined on Python code.

We fine-tune CuBERT on each of the classification tasks and compare the results with multi-layered bidirectional LSTM (Hochreiter & Schmidhuber, 1997) models. We train the LSTM models from scratch and also using pre-trainined Word2Vec embeddings. Our results show that CuBERT consistently outperforms these baseline models by 2.9–22% across the tasks. We perform a number of additional studies by varying the sampling strategies used for training Word2Vec models, by varying program lengths, and by comparing against Transformer models trained from scratch. In addition, we also show that CuBERT can be fine-tuned effectively using only 33% of the task-specific labeled data and with only 2 epochs, and that it attains results competitive to the baseline models trained with the full datasets and many more epochs. CuBERT, when fine-tuned on the variable-misuse localization and repair task, produces high classification, localization and localization+repair accuracies and outperforms published state-of-the-art models (Hellendoorn et al., 2020; Vasic et al., 2019). Our contributions are as follows:

- We present the first attempt at pre-training a BERT contextual embedding of source code.
- We show the efficacy of the pre-trained contextual embedding on five classification tasks. Our fine-tuned models outperform baseline LSTM models (with/without Word2Vec embeddings), as well as Transformers trained from scratch, even with reduced training data.
- We evaluate CuBERT on a pointer prediction task.
- We will make the models and datasets publicly available. We hope that future work would benefit by reusing our benchmark tasks and comparing against the strong baseline, in the form of CuBERT, that we provide.

2. Related Work

Given the abundance of natural-language text, and the relative difficulty of obtaining labeled data, much effort has been devoted to using large corpora to learn about language in an unsupervised fashion, before trying to focus on tasks with small labeled training datasets. Word2Vec (Mikolov et al., 2013a;b) computed word embeddings based on word co-occurrence and proximity, but the same embedding is used regardless of the context. The continued advances in word (Pennington et al., 2014) and subword (Bojanowski et al., 2017) embeddings led to publicly released pre-trained embeddings, used in a variety of tasks.

To deal with varying word context, contextual word embeddings were developed (McCann et al., 2017; Peters et al., 2018; Radford et al., 2018; 2019), in which an embedding is learned for the *context* of a word in a particular sentence, namely the sequence of words preceding it and possibly following it. BERT (Devlin et al., 2019) improved natural-language pre-training by using a de-noising autoencoder. Instead of learning a language model, which is inherently sequential, BERT optimizes for predicting a noised word within a sentence. Such prediction instances are generated by choosing a word position and either keeping it unchanged, removing the word, or replacing the word with a random wrong word. It also pre-trains with the objective of predicting whether two sentences can be next to each other. These pre-training objectives, along with the use of a Transformer-based architecture, gave BERT an accuracy boost in a number of NLP tasks over the state-of-the-art. BERT has been improved upon in various ways, including modifying training objectives, utilizing ensembles, combining attention with autoregression (Yang et al., 2019), and expanding pre-training corpora and time (Liu et al., 2019). However, the main architecture of BERT seems to hold up as the state-of-the-art, as of this writing.

In the space of programming languages, attempts have been made to learn embeddings in the context of specific software-engineering tasks (Chen & Monperrus, 2019). These include embeddings of variable and method identifiers using local and global context (Allamanis et al., 2015), abstract syntax trees or ASTs (Mou et al., 2016; Zhang et al., 2019), paths in ASTs (Alon et al., 2019), memory heap graphs (Li et al., 2016), and ASTs enriched with data-flow information (Allamanis et al., 2018; Hellendoorn et al., 2020). These approaches require analyzing source code beyond simple tokenization. In this work, we derive a pre-trained contextual embedding of tokenized source code without explicitly modeling source-code-specific information, and show that the resulting embedding can be effectively fine-tuned for downstream tasks.

Upcoming related work (Feng et al., 2020) aims at solving paired natural-language (NL) and programming-language (PL) tasks, such as code search and generation of documentation for code, in a multi-lingual setting. Towards this, it pre-trains a Transformer encoder by treating a natural-language description of a function and its body as separate sentences in the sentence-pair representation of BERT. We also handle natural language directly, but do not require such a separation. Natural-language tokens can be mixed with source-code tokens both within and across sentences in our encoding. One of our benchmark tasks, function-docstring mismatch, illustrates the ability of CuBERT to handle NL-PL tasks.

3. Experimental Setup

We now outline our benchmarks and experimental study. The supplementary material contains deeper detail aimed at reproducing our results. [1]

3.1. Code Corpus for Fine-Tuning Tasks

We use the ETH Py150 corpus (Raychev et al., 2016) to generate datasets for the fine-tuning tasks. This corpus consists of 150K Python files from GitHub, and is partitioned into a training split (100K files) and a test split (50K files). We held out 10K files from the training split as a validation split. We deduplicated the dataset in the fashion of Allamanis (2018), resulting in a slightly smaller dataset of 85K, 9.5K, and 47K files in train, validation, and test. This is our Python fine-tuning code corpus.

3.2. The GitHub Python Pre-Training Code Corpus

We used the public GitHub repository hosted on Google's BigQuery platform (the `github_repos` dataset under BigQuery's public-data project, `bigquery-public-data`). We extracted all files ending in `.py`, under open-source, redistributable licenses, removed symbolic links, and retained only files reported to be in the `refs/heads/master` branch. This resulted in about 16.1M files.

[1]The datasets and results presented in this section and the next represent the original data we collected for this project. In the process of open-sourcing our artifacts and addressing reviewer comments, we have had to adjust our data and corresponding results. The final version of this paper will include links to the open-sourced artifacts, and updated quantitative results reflecting those artifacts. The statistics and experimental results in this paper version were current as of April 2020.

To avoid duplication between pre-training and fine-tuning data, we removed files that had high similarity to the files in the ETH Py150 dataset, using the method of Allamanis (2018). In particular, two files are considered similar to each other if the Jaccard similarity between the sets of tokens (identifiers and string literals) is above 0.8 and in addition, it is above 0.7 for multi-sets of tokens. This brought the dataset to 13.5M files. We then further deduplicated the remaining files, by clustering them into equivalence classes holding similar files according to the same similarity metric, and keeping only one exemplar per equivalence class. This helps avoid biasing the pre-trained embedding. Finally, we removed files that could not be tokenized. In the end, we were left with 6.6M Python files containing over 2 billion words. This is our Python pre-training code corpus.

3.3. Source-Code Modeling

We first tokenize a Python program using the standard Python tokenizer (the `tokenize` package). We leave language keywords intact and produce special tokens for syntactic elements that have either no string representation (e.g., `DEDENT` tokens, which occur when a nested program scope concludes), or ambiguous interpretation (e.g., new line characters inside string literals, at the logical end of a Python statement, or in the middle of a Python statement result in distinct special tokens). We split identifiers according to common heuristic rules (e.g., snake or Camel case). Finally, we split string literals using heuristic rules, on whitespace characters, and on special characters. We limit all thus produced tokens to a maximum length of 15 characters. We call this the *program vocabulary*. Our Python pre-training code corpus contained 10.2M unique tokens, including 12 reserved tokens.

We greedily compress the program vocabulary into a *subword vocabulary* (Schuster & Nakajima, 2012) using the `SubwordTextEncoder` from the Tensor2Tensor project (Vaswani et al., 2018), resulting in about 50K tokens. All words in the program vocabulary can be losslessly encoded using one or more of the subword tokens.

We encode programs first into program tokens, as described above, and then encode those tokens one by one in the subword vocabulary. The objective of this encoding scheme is to preserve syntactically meaningful boundaries of tokens. For example, the identifier "`snake_case`" could be encoded as "`sna ke _ ca se`", preserving the snake case split of its characters, even if the subtoken "`e_c`" were very popular in the corpus; the latter encoding might result in a smaller representation but would lose the intent of the programmer in using a snake-case identifier. Similarly, "`i=0`" may be very frequent in the corpus, but we still force it to be encoded as separate tokens `i`, `=`, and `0`, ensuring that we preserve the distinction between operators and operands.

Both the BERT model and the Word2Vec embeddings are built on the subword vocabulary.

3.4. Fine-Tuning Tasks

To evaluate CuBERT, we design five classification tasks and a multi-headed pointer task. These are motivated by prior work, but unfortunately, the associated datasets come from different languages and varied sources. We want the tasks to be on Python code, and for accurate results, we ensure that there is no overlap between pre-training and fine-tuning datasets. We therefore create all the tasks on the ETH Py150 corpus (see Section 3.1). As discussed in Section 3.2, we ensure that there is no duplication between this and the pre-training corpus. We hope that our datasets for these tasks will be useful to others as well. The fine-tuning tasks are described below. A more detailed discussion is presented in the supplementary material.

Variable-Misuse Classification Allamanis et al. (2018) observed that developers may mistakenly use an incorrect variable in the place of a correct one. These mistakes may occur when developers copy-paste similar code but forget to rename all occurrences of variables from the original fragment, or when there are similar variable names that can be confused with each other. These can be subtle errors that remain undetected during compilation. The task by Allamanis et al. (2018) is to predict a correct variable name at a location within a function and was devised on C# programs. We take the classification version restated by Vasic et al. (2019), wherein, given a function, the task is to predict whether there is a variable misuse at some location in the function, without specifying a particular location to consider. Here, the classifier has to consider all variables and their usages to make the decision. In order to create negative (buggy) examples, we replace a variable use at some location with another variable that is defined within the function.

Wrong Binary Operator Pradel & Sen (2018) proposed the task of detecting whether a binary operator in a given expression is correct. They use features extracted from limited surrounding context. We use the entire function with the goal of detecting whether any binary operator in the function is incorrect. The negative examples are created by randomly replacing some binary operator with another type-compatible operator.

Swapped Operand Pradel & Sen (2018) propose the wrong binary operand task where a variable or constant is used incorrectly in an expression, but that task is quite similar to the variable-misuse task we already use. We therefore define another class of operand errors where the operands of non-commutative binary operators are swapped. The operands can be arbitrary subexpressions, and are not restricted to be just variables or constants. To simplify example generation, we restrict examples for this task to those in which the binary operator and its operands all fit within a single line.

Function-Docstring Mismatch Developers are encouraged to write descriptive docstrings to explain the functionality and usage of functions. This provides parallel corpora between code and natural language sentences that have been used for machine translation (Barone & Sennrich, 2017), detecting uninformative docstrings (Louis et al., 2018) and to evaluate their utility to provide supervision in neural code search (Cambronero et al., 2019). We prepare a sentence-pair classification problem where the function and its docstring form two distinct sentences. Similar to the other fine-tuning tasks, we use the ETH Py150 corpus to create this dataset. The positive examples come from the correct function-docstring pairs. We create negative examples by replacing correct docstrings with docstrings of other functions, randomly chosen from the dataset. For this task, the existing docstring is removed from the function body.

Exception Type While it is possible to write generic exception handlers (e.g., "`except Exception`" in Python), it is considered a good coding practice to catch and handle the precise exceptions that can be raised by a code fragment. We identified the 20 most common exception types from the GitHub dataset, excluding the catch-all `Exception` (full list in Table 1 in the supplementary material). Given a function with an `except` clause for one of these exception types, we replace the exception with a special "hole" token. The task is the multi-class classification problem of predicting the original exception type.

Variable-Misuse Localization and Repair As an instance of a non-classification task, we consider the joint classification, localization, and repair version of the variable-misuse task from (Vasic et al., 2019). Given a function, the task is to predict one pointer (called the localization pointer) to identify a variable-misuse location, and another pointer (called the repair pointer) to identify a variable from the same function that is the right one to use at the faulty location. The model is also trained to classify functions that do not contain any variable misuse as bug-free by making the localization pointer point to a special location in the function. We create negative examples using the same method as used in the Variable-Misuse Classification task.

Table 1 lists the sizes of the resulting benchmark datasets extracted from the fine-tuning corpus. The Exception Type task contains fewer examples than the other tasks, since examples for this task only come from functions that catch one of the chosen 20 exception types.

	Train	Validation	Test
Variable-Misuse Classification	796 020	8 192 (86 810)	429 854
Wrong Binary Operator	537 244	8 192 (59 112)	293 872
Swapped Operand	276 116	8 192 (30 818)	152 248
Function-Docstring	391 049	8 192 (44 029)	213 269
Exception Type	21 694	2 459 (2 459)	12 036
Variable-Misuse Localization and Repair	796 020	8 192 (86 810)	429 854

Table 1. Benchmark fine-tuning datasets. Note that for validation, we have subsampled the original datasets (in parentheses) down to 8192 examples, except for exception classification, which only had 2459 validation examples, all of which are included.

3.5. BERT for Source Code

The BERT model (Devlin et al., 2019) consists of a multi-layered Transformer encoder. It is trained with two tasks: (1) to predict the correct tokens in a fraction of all positions, some of which have been replaced with incorrect tokens or the special `[MASK]` token (the Masked Language Model task, or *MLM*) and (2) to predict whether the two sentences separated by the special `[SEP]` token follow each other in some natural discourse (the Next-Sentence Prediction task, or *NSP*). Thus, each example consists of one or two *sentences*, where a sentence is the concatenation of contiguous lines from the source corpus, sized to fit the target example length. To ensure that every sentence is treated in multiple instances of both MLM and NSP, BERT by default duplicates the corpus 10 times, and generates independently derived examples from each duplicate. With 50% probability, the second example sentence comes from a random document (for NSP). A token is chosen at random for an MLM prediction (up to 20 per example), and from those chosen, 80% are masked, 10% are left undisturbed, and 10% are replaced with a random token.

CuBERT is similarly formulated, but a CuBERT line is a logical code line, as defined by the Python standard. Intuitively, a logical code line is the shortest sequence of consecutive lines that constitutes a legal statement, e.g., it has correctly matching parentheses. We count example lengths by counting the subword tokens of both sentences (see Section 3.3).

We train the BERT Large model having 24 layers with 16 attention heads and 1024 hidden units. Sentences are created from our pre-training dataset. Task-specific classifiers pass the embedding of a special start-of-example `[CLS]` token through feedforward and softmax layers. For the pointer prediction task, the pointers are computed over the sequence of outputs generated by the last layer of the BERT model.

3.6. Baselines

3.6.1. Word2Vec

We train Word2Vec models using the same pre-training corpus as the BERT model. To maintain parity, we generate the dataset for Word2Vec using the same pipeline as BERT but by disabling masking and generation of negative examples for NSP. The dataset is generated without any duplication. We train both CBOW and Skipgram models using GenSim (Řehůřek & Sojka, 2010). To deal with the large vocabulary, we use negative sampling and hierarchical softmax (Mikolov et al., 2013a;b) to train the two versions. In all, we obtain four types Word2Vec embeddings.

3.6.2. Bidirectional LSTM and Transformer

In order to obtain context-sensitive encodings of input sequences for the fine-tuning tasks, we use multi-layered bidirectional LSTMs (Hochreiter & Schmidhuber, 1997) (BiLSTMs). These are initialized with the pre-trained Word2Vec embeddings. To further evaluate whether LSTMs alone are sufficient without pre-training, we try initializing the BiLSTM with an embedding matrix that is trained from scratch. We also trained Transformer models (Vaswani et al., 2017) for our fine-tuning tasks. We used BERT's own Transformer implementation, to ensure comparability of results. For comparison with prior work, we use the unidirectional LSTM and pointer model from Vasic et al. (2019) for the Variable-Misuse Localization and Repair task.

4. Experimental Results

4.1. Training Details

As stated above, CuBERT's dataset generation duplicates the corpus 10 times, whereas Word2Vec is trained without duplication. To compensate for this difference, we trained Word2Vec for 10 epochs and CuBERT for 1 epoch.

We pre-train CuBERT with the default configuration of the BERT Large model, one model per example length (128, 256, and 512 subword tokens) with batch sizes of 8192, 4096, and 2048, respectively. For Word2Vec, when training with negative samples, we choose 10 negative samples. The embedding size for all the Word2Vec pre-trained models is set at 1024. We used TPU pods for training our models. The model evaluations were performed on P100 GPUs.

For the baseline BiLSTM models, we did extensive ex-

	Setting		Misuse	Operator	Operand	Docstring	Exception
BiLSTM (100 epochs)	From scratch		76.05%	82.00%	87.77%	78.43%	40.37%
	CBOW	ns	**77.66**%	**84.42**%	88.66%	**89.13**%	48.85%
		hs	77.01%	84.11%	**89.69**%	86.74%	46.73%
	Skipgram	ns	71.58%	83.06%	87.67%	84.69%	48.54%
		hs	77.21%	83.06%	89.01%	82.56%	**49.68**%
CuBERT	2 epochs		90.09%	85.15%	88.67%	95.81%	52.38%
	10 epochs		92.73%	88.43%	88.67%	95.81%	62.55%
	20 epochs		**94.61**%	**90.24**%	**92.56**%	**96.85**%	**71.74**%
Transformer	(100 epochs)		79.37%	78.66%	86.21%	91.10%	48.60%

Table 2. Test accuracies of fine-tuned CuBERT against BiLSTM (with and without Word2Vec embeddings) and Transformer trained from scratch on the classification tasks. “ns” and “hs” respectively refer to negative sampling and hierarchical softmax settings used for training CBOW and Skipgram models. “From scratch” refers to training with freshly initialized token embeddings, that is, without pre-trained Word2Vec embeddings.

perimentation on the Variable-Misuse task by varying the number of layers (1–3) and the number of hidden units (128, 256, 512). We also tried LSTM output dropout probability (0.1, 0.5) and learning rates (1e-3, 1e-4, 1e-5) with the Adam (Kingma & Ba, 2014) optimizer. The most promising combination was a 3-layered BiLSTM with 512 hidden units per layer, LSTM output dropout probability of 0.1 and learning rate of 1e-3. We use this set of parameters for all the tasks except the Exception-Type task. Due to the much smaller dataset size of the latter (Table 1), we did a separate search and chose a single-layer BiLSTM with 256 hidden units. We used the batch size of 8192 for the larger tasks and 64 for the Exception-Type task. For the baseline Transformer models, we originally attempted to train a Transformer model of the same configuration as CuBERT. However, the sizes of our fine-tuning datasets seemed too small to train that large a Transformer. Instead, we performed a hyperparameter search over transformer layers (1–6), hidden units (128, 256, 512), learning rates (5e-5, 1e-4, 5e-4, 1e-3) and batch sizes (64, 256, 1024, 2048, 4096, 8192) on the Variable-Misuse task. The best architecture (4 layers, 512 hidden units, 16 attention heads, learning rate of 5e-4, batch size of 4096) is used for all the tasks except the Exception-Type task. A separate experimentation for the smaller Exception-Type dataset resulted in the best configuration of 3 layers, 512 hidden units, 16 attention heads, learning rate of 5e-5, and batch size of 2048.

Finally, for our baseline pointer model (referred to as LSTM+pointer below) we searched over the following hyperparameter choices: hidden sizes of 512 and 1024, token embedding sizes of 512 and 1024, learning rates of 1e-1, 1e-2, and 1e-3, and the AdaGrad and Gradient Descent optimizers. In contrast to the original work (Vasic et al., 2019), we generated one pair of buggy/bug-free examples per function (rather than one per variable use, per function, which would bias towards longer functions), and use CuBERT's subword-tokenized vocabulary of 50K subtokens (rather than a limited full-token vocabulary, which leaves many tokens out of vocabulary).

4.2. Research Questions

We set out to answer the following research questions. We will address each with our results.

1. Do contextual embeddings help with source-code analysis tasks, when pre-trained on an unlabeled code corpus? We compare CuBERT to BiLSTM models with and without pre-trained Word2Vec embeddings on the classification tasks (Section 4.3).
2. Does fine-tuning actually help, or is the Transformer model behind CuBERT the main power behind the approach? We compare fine-tuned CuBERT models to Transformer-based models trained from scratch on the classification tasks (Section 4.4).
3. How does the performance of CuBERT on the classification tasks scale with the amount of labeled training data? We compare the performance of fine-tuned CuBERT models when fine-tuning with one third, two thirds, or the full training dataset for each task (Section 4.5).
4. How does example length affect the benefits of CuBERT? We compare fine-tuning performance for different example lengths on the classification tasks (Section 4.6).
5. How does CuBERT perform on complex tasks, against state-of-the-art methods? We implemented and fine-tuned a model for a multi-headed pointer prediction task, namely, the Variable-Misuse Localization and Repair task (Section 4.7). We compare it to the models from (Vasic et al., 2019) and (Hellendoorn et al., 2020).

Except for Section 4.6, all the results are presented for sequences of length 512. We give examples of classification instances in the supplementary material and include visualizations of attention weights for them.

4.3. Contextual vs. Word Embeddings

The purpose of this analysis is to understand how much pre-trained contextual embeddings help, compared to word embeddings. For each classification task, we trained BiLSTM models starting with each of the Word2Vec embeddings, namely, continuous bag of words (CBOW) and Skip-gram trained with negative sampling or hierarchical softmax. The Word2Vec embeddings can be refined during training. Within the first 100 epochs, the performance of the BiLSTM models stopped improving. The best model weights per task were selected by finding the minimum validation loss on the corresponding dataset (Table 1) over the first 100 epochs. On the CuBERT side, we fine-tuned the pre-trained model for 20 epochs, with similar model selection.

The resulting test accuracies are shown in Table 2. CuBERT consistently outperforms BiLSTM (with the best task-wise Word2Vec configuration) on all tasks, by a margin of 2.9–22%. Thus, the pre-trained contextual embedding provides superior results even with a smaller budget of 20 epochs, compared to the 100 epochs used for BiLSTMs. The Exception Type classification task has an order of magnitude less training data than the other tasks (see Table 1). The difference between the performance of BiLSTM and CuBERT is the highest for this task. Thus, fine-tuning is of much value for tasks with limited labeled training data.

We analyzed the performance of CuBERT with the reduced fine-tuning budget of only 2 and 10 epochs (see Table 2). Except for the Operand task, CuBERT outperforms BiLSTM within 2 fine-tuning epochs. On the Operand task, the performance difference between CuBERT with 2 or 10 fine-tuning epochs and BiLSTM is about 1%. For the rest of the tasks, CuBERT with only 2 fine-tuning epochs outperforms BiLSTM (with the best task-wise Word2Vec configuration) by a margin of 0.7–12%. This shows that CuBERT can reach accuracies that are comparable to or better than those of BiLSTMs trained with Word2Vec embeddings within only a few epochs.

We also trained the BiLSTM models from scratch, that is, without using the Word2Vec embeddings. The results are shown in the first row of Table 2. Compared to those, the use of Word2Vec embeddings performs better by a margin of 1.5–10.5%. Though no single Word2Vec configuration is the best, CBOW trained with negative sampling gives the most consistent results overall.

4.4. Is Transformer All You Need?

One may wonder if CuBERT's promising results derive more from using a Transformer-based model for its classification tasks, and less from the actual, unsupervised pre-training. Here we compare our results on the classification tasks to a Transformer-based model trained from scratch, i.e., without the benefit of a pre-trained embedding. As discussed in Section 4.1, the size of the training data limited us to try out Transformers that were substantially smaller than the CuBERT model (which is equivalent to the BERT Large model). All the Transformer models were trained for 100 epochs during which their performance stopped improving. We selected the best model within the chosen hyperparameters for each task based on least validation loss. As seen from the last row of Table 2, the performance of CuBERT is substantially higher than the Transformer models trained from scratch. Thus, for the same choice of architecture (i.e., Transformer) pre-training seems to help by enabling training of a larger and better model.

4.5. The Effects of Little Supervision

The big draw of unsupervised pre-training followed by fine-tuning is that some tasks have small labeled datasets. We study here how CuBERT fares with reduced training data. We sampled uniformly the training dataset to $2/3$rds and $1/3$rd its size, and produced corresponding training datasets for each classification task. We then fine-tuned the pre-trained CuBERT model with each of the 3 different training splits. Validation and testing were done with the same original datasets. Table 3 shows the results.

The Function Docstring task seems robust to the reduction of the training dataset, both early and late in the fine-tuning process (that is, within 2 vs. 20 epochs), whereas the Exception Classification task is heavily impacted by the dataset reduction, given that it has relatively few training examples to begin with. Interestingly enough, for some tasks, even fine-tuning for only 2 epochs and only using a third of the training data outperforms the baselines. For example, for both Variable Misuse and Function Docstring, CuBERT at 2 epochs and $1/3$rd training data outperforms the BiLSTM with Word2Vec and the Transformer baselines.

4.6. The Effects of Context

Context size is especially useful in code tasks, given that some relevant information may lie many "sentences" away from its locus of interest. Here we study how reducing the context length (i.e., the length of the examples used to pre-train and fine-tune) affects performance. We produce data with shorter example lengths by following the standard BERT mechanism. Table 4 shows the results.

Although context seems to be important to most tasks, the

Best of # Epochs	**Train Fraction**	**Misuse**	**Operator**	**Operand**	**Docstring**	**Exception**
2	100%	90.09%	85.15%	88.67%	95.81%	52.38%
	66%	89.52%	83.26%	88.66%	95.17%	34.70%
	33%	88.64%	82.28%	87.45%	95.29%	26.87%
10	100%	92.73%	88.43%	88.67%	95.81%	62.55%
	66%	92.06%	87.06%	90.39%	95.64%	64.59%
	33%	91.23%	84.44%	87.45%	95.48%	54.22%
20	100%	94.61%	90.24%	92.56%	96.85%	71.74%
	66%	94.19%	89.36%	92.01%	96.17%	70.11%
	33%	93.54%	87.67%	91.30%	96.37%	67.72%

Table 3. Effects of reducing training-split size on fine-tuning performance on the classification tasks.

Length	**Misuse**	**Operator**	**Operand**	**Docstring**	**Exception**
128	85.89%	77.92%	77.17%	97.10%	55.95%
256	92.69%	86.52%	87.26%	97.08%	65.38%
512	94.61%	90.24%	92.56%	96.85%	71.74%

Table 4. Best out of 20 epochs of fine-tuning, for three example lengths, on the classification tasks.

Model	**Setting**	**True Positive**	**Classification Accuracy**	**Localization Accuracy**	**Loc+Repair Accuracy**
LSTM+pointer	100 epochs	81.63%	78.76%	63.83%	56.37%
CuBERT+pointer	2 epochs	**97.18%**	89.37%	79.05%	75.84%
	10 epochs	94.94%	93.05%	88.52%	85.91%
	20 epochs	96.83%	**94.85%**	**91.11%**	**89.35%**

Table 5. Comparison of the fine-tuned CuBERT+pointer model and the LSTM+pointer model from Vasic et al. (2019) on the variable-misuse localization and repair task.

Function Docstring task seems to improve with reduced context. This may be because the task primarily depends on comparison between the docstring and the function signature, and including more context dilutes the model's focus.

For comparison, we also evaluated the BiLSTM model on sequences of length 128 and 256 for the Variable-Misuse task with CBOW and negative sampling. We obtained accuracies of 71.34% and 73.63% respectively, which are lower than the best BiLSTM accuracy on sequence length 512 and also lower than the accuracies of CuBERT for the corresponding lengths (see Table 4).

4.7. Evaluation on a Multi-Headed Pointer Task

We now discuss the results of fine-tuning CuBERT to predict the localization and repair pointers for the variable-misuse task. For this task, we implement the multi-headed pointer model from Vasic et al. (2019) on top of CuBERT. The baseline consists of the same pointer model on a unidirectional LSTM as used by Vasic et al. (2019). We refer to these respectively as CuBERT+pointer and LSTM+pointer models, respectively. Due to limitations of space, we omit the details of the pointer model and refer the reader to the above paper. As reported in Section 4 of that work, to enable comparison with an enumerative approach, the evaluation was performed only on 12K test examples. In comparison, we report the numbers on all 430K test examples (Table 1) for both the models.

We trained the baseline model for 100 epochs and fine-tuned CuBERT for up to 20 epochs. Table 5 gives the results along the same metrics as Vasic et al. (2019). The metrics are defined as follows: 1) True Positive is the percentage of bug-free functions classified as bug-free. 2) Classification Accuracy is the percentage of correctly classified examples (between bug-free and buggy). 3) Localization Accuracy is the percentage of buggy examples for which the localization pointer correctly identifies the bug location. 4) Localization+Repair Accuracy is the precentage of buggy examples for which both the localization and repair pointers make correct predictions. As seen from Table 5, the CuBERT+pointer model outperforms the LSTM+pointer model consistently across all the metrics, and even within 2 and 10 epochs.

More recently, Hellendoorn et al. (2020) evaluated hybrid models for the same task, combining graph neural networks, Transformers, and RNNs. The test classification and localization+repair accurracies of that work (computed on their test data) were 81.9% and 73.8%, respectively. To compare our models to those of Hellendoorn et al. (2020), we obtained the same test dataset from the authors, and evaluated our best CuBERT fine-tuned model on it. Our corresponding accuracies are 90.5% and 80%, respectively. Interestingly, the models by Hellendoorn et al. (2020) make use of richer input representations, involving the data flow and control flow of programs. Nevertheless, CuBERT outperforms them while using only a lexical representation of the input program.

5. Conclusions and Future Work

We present the first attempt at pre-trained contextual embedding of source code by training a BERT model, called CuBERT, which we fine-tuned on five classification tasks and compared against BiLSTM with Word2Vec embeddings and Transformer models. As a more challenging task, we also evaluated CuBERT on a multi-headed pointer prediction task. CuBERT outperformed the baseline models consistently. We evaluated CuBERT with less data and fewer epochs, highlighting the benefits of pre-training on a massive code corpus.

We use only source-code tokens and leave it to the underlying Transformer model to infer any structural interactions between them through self-attention. Prior work (Allamanis et al., 2018; Hellendoorn et al., 2020) has argued for explicitly using structural program information (e.g., control flow and data flow). It is an interesting avenue of future work to incorporate such information in pre-training using relation-aware Transformers (Shaw et al., 2018). However, our improved results in comparison to Hellendoorn et al. (2020) show that CuBERT is a simple yet powerful technique and provides a strong baseline for future work on source-code representations.

While surpassing the accuracies achieved by CuBERT with newer models and pre-training/fine-tuning methods would be a natural extension to this work, we also envision other follow-up work. There is increasing interest in developing pre-training methods that can produce smaller models more efficiently and that trade-off accuracy for reduced model size. Further, our benchmark could be valuable to techniques that explore other program representations (e.g., trees and graphs), in multi-task learning, and to develop related tasks such as program synthesis.

Acknowledgements

We are indebted to Daniel Tarlow for his guidance and generous advice throughout the development of this work. Our work has also improved thanks to feedback, use cases, and proofs of concept offered by Vincent Hellendoorn, Ben Lerner, Hyeontaek Lim, Rishabh Singh, Charles Sutton, and Manushree Vijayvergiya. Finally, we are grateful to the anonymous reviewers, who gave useful, constructive comments and helped us improve our presentation and results.

References

Allamanis, M. The adverse effects of code duplication in machine learning models of code. *CoRR*, abs/1812.06469, 2018. URL http://arxiv.org/abs/1812.06469.

Allamanis, M., Barr, E. T., Bird, C., and Sutton, C. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pp. 38–49, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3675-8. doi: 10.1145/2786805.2786849. URL http://doi.acm.org/10.1145/2786805.2786849.

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

Alon, U., Zilberstein, M., Levy, O., and Yahav, E. Code2vec: Learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL):40:1–40:29, January 2019. ISSN 2475-1421. doi: 10.1145/3290353. URL http://doi.acm.org/10.1145/3290353.

Barone, A. V. M. and Sennrich, R. A parallel corpus of python functions and documentation strings for automated code documentation and code generation. *arXiv preprint arXiv:1707.02275*, 2017.

Bojanowski, P., Grave, E., Joulin, A., and Mikolov, T. Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics*, 5: 135–146, 2017.

Cambronero, J., Li, H., Kim, S., Sen, K., and Chandra, S. When deep learning met code search. *arXiv preprint arXiv:1905.03813*, 2019.

Chen, Z. and Monperrus, M. A literature study of embeddings on source code. *arXiv preprint arXiv:1904.03061*, 2019.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pp. 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1423. URL https://www.aclweb.org/anthology/N19-1423.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

Gu, X., Zhang, H., Zhang, D., and Kim, S. Deep api learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pp. 631–642, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4218-6. doi: 10.1145/2950290.2950334. URL http://doi.acm.org/10.1145/2950290.2950334.

Gupta, R., Pal, S., Kanade, A., and Shevade, S. Deepfix: Fixing common c language errors by deep learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, AAAI'17, pp. 1345–1351. AAAI Press, 2017. URL http://dl.acm.org/citation.cfm?id=3298239.3298436.

Hellendoorn, V. J., Sutton, C., Singh, R., Maniatis, P., and Bieber, D. Global relational models of source code. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=B1lnbRNtwr.

Hindle, A., Barr, E. T., Su, Z., Gabel, M., and Devanbu, P. On the naturalness of software. In *2012 34th International Conference on Software Engineering (ICSE)*, pp. 837–847, June 2012. doi: 10.1109/ICSE.2012.6227135.

Hochreiter, S. and Schmidhuber, J. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997. ISSN 0899-7667. doi: 10.1162/neco.1997.9.8.1735. URL http://dx.doi.org/10.1162/neco.1997.9.8.1735.

Iyer, S., Konstas, I., Cheung, A., and Zettlemoyer, L. Mapping language to code in programmatic context. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, October 31 - November 4, 2018*, pp. 1643–1652, 2018. URL https://www.aclweb.org/anthology/D18-1192/.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Li, Y., Tarlow, D., Brockschmidt, M., and Zemel, R. S. Gated graph sequence neural networks. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016. URL http://arxiv.org/abs/1511.05493.

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., and Stoyanov, V. Roberta: A robustly optimized BERT pretraining approach. *CoRR*, abs/1907.11692, 2019. URL http://arxiv.org/abs/1907.11692.

Louis, A., Dash, S. K., Barr, E. T., and Sutton, C. Deep learning to detect redundant method comments. *arXiv preprint arXiv:1806.04616*, 2018.

Martin, R. C. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1 edition, 2008. ISBN 0132350882, 9780132350884.

McCann, B., Bradbury, J., Xiong, C., and Socher, R. Learned in translation: Contextualized word vectors. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 30*, pp. 6294–6305. Curran Associates, Inc., 2017.

Mikolov, T., Chen, K., Corrado, G., and Dean, J. Efficient estimation of word representations in vector space. In *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*, 2013a. URL http://arxiv.org/abs/1301.3781.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., and Dean, J. Distributed representations of words and phrases and their compositionality. In Burges, C. J. C., Bottou, L., Welling, M., Ghahramani, Z., and Weinberger, K. Q. (eds.), *Advances in Neural Information Processing Systems 26*, pp. 3111–3119. Curran Associates, Inc., 2013b.

Mou, L., Li, G., Zhang, L., Wang, T., and Jin, Z. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, pp. 1287–1293. AAAI Press, 2016. URL http://dl.acm.org/citation.cfm?id=3015812.3016002.

Oda, Y., Fudaba, H., Neubig, G., Hata, H., Sakti, S., Toda, T., and Nakamura, S. Learning to generate pseudo-code from source code using statistical machine translation (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 574–584. IEEE, 2015.

Pennington, J., Socher, R., and Manning, C. D. Glove: Global vectors for word representation. In *In EMNLP*, 2014.

Peters, M. E., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K., and Zettlemoyer, L. Deep contextualized word representations. In *Proceedings of NAACL-HLT*, pp. 2227–2237, 2018.

Pradel, M. and Sen, K. Deepbugs: A learning approach to name-based bug detection. *Proc. ACM Program. Lang.*, 2(OOPSLA):147:1–147:25, October 2018. ISSN 2475-1421. doi: 10.1145/3276517. URL http://doi.acm.org/10.1145/3276517.

Pu, Y., Narasimhan, K., Solar-Lezama, A., and Barzilay, R. 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 Companion 2016, pp. 39–40, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4437-1. doi: 10.1145/2984043.2989222. URL http://doi.acm.org/10.1145/2984043.2989222.

Radford, A., Narasimhan, K., Salimans, T., and Sutskever, I. Improving language understanding by generative pre-training. *URL https://s3-us-west-2. amazonaws. com/openai-assets/researchcovers/languageunsupervised/language understanding paper. pdf*, 2018.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. Language models are unsupervised multitask learners. *OpenAI Blog*, 1(8), 2019.

Raychev, V., Vechev, M., and Yahav, E. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pp. 419–428, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2784-8. doi: 10.1145/2594291.2594321. URL http://doi.acm.org/10.1145/2594291.2594321.

Raychev, V., Bielik, P., and Vechev, M. T. Probabilistic model for code with decision trees. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, pp. 731–747, 2016.

Řehůřek, R. and Sojka, P. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, pp. 45–50, Valletta, Malta, May 2010. ELRA. http://is.muni.cz/publication/884893/en.

Schuster, M. and Nakajima, K. Japanese and korean voice search. In *International Conference on Acoustics, Speech and Signal Processing*, pp. 5149–5152, 2012.

Shaw, P., Uszkoreit, J., and Vaswani, A. Self-attention with relative position representations. *arXiv preprint arXiv:1803.02155*, 2018.

Vasic, M., Kanade, A., Maniatis, P., Bieber, D., and Singh, R. Neural program repair by jointly learning to localize and repair. *CoRR*, abs/1904.01720, 2019. URL http://arxiv.org/abs/1904.01720.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L. u., and Polosukhin, I. Attention is all you need. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 30*, pp. 5998–6008. Curran Associates, Inc., 2017. URL http://papers.nips.cc/paper/7181-attention-is-all-you-need.pdf.

Vaswani, A., Bengio, S., Brevdo, E., Chollet, F., Gomez, A. N., Gouws, S., Jones, L., Kaiser, L., Kalchbrenner, N., Parmar, N., Sepassi, R., Shazeer, N., and Uszkoreit, J. Tensor2tensor for neural machine translation. In *Proceedings of the 13th Conference of the Association for Machine Translation in the Americas, AMTA 2018, Boston, MA, USA, March 17-21, 2018 - Volume 1: Research Papers*, pp. 193–199, 2018. URL https://www.aclweb.org/anthology/W18-1819/.

Yang, Z., Dai, Z., Yang, Y., Carbonell, J. G., Salakhutdinov, R., and Le, Q. V. Xlnet: Generalized autoregressive pretraining for language understanding. *CoRR*, abs/1906.08237, 2019. URL http://arxiv.org/abs/1906.08237.

Zhang, J., Wang, X., Zhang, H., Sun, H., Wang, K., and Liu, X. A novel neural source code representation based on abstract syntax tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 783–794. IEEE, 2019.