Learning and Evaluating Contextual Embedding of Source Code

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

[1] Indian Institute of Science
[2] Google Brain

General-Purpose Representations of Source Code

- Success of learned representations (e.g., ELMo, GPT, BERT, etc.) in NLU
- Source code is a formal description of an executable task.
- Source code is a means to communicate developer intent.
  - Meaningful identifier names
  - Natural-language documentation
  - Convey a lot of semantic information
- Could the following code be buggy?

```
number_of_batches = batch_size / number_of_examples
```

CuBERT: Code Understanding BERT*

Can we exploit characteristics of source code to learn general-purpose representations that can be used effectively in downstream tasks?

Pre-train a deep bidirectional Transformer encoder from unlabeled code.

Use the pre-training objectives, masked language modeling (MLM) and next-sentence prediction (NSP), popularized by BERT.

Design and evaluate on a new benchmark of six code-understanding tasks -- including five classification and one multi-headed pointer prediction task.

*BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding

Experimental Results

Q1: How do contextual embeddings compare against word embeddings?

CuBERT outperforms BiLSTM models initialized with pre-trained source-code-specific Word2Vec embeddings by +2.9% to +22%.

Q2: Is Transformer (without pre-training) all you need?

CuBERT outperforms Transformers trained from scratch by +5.8% to +23%.

Q3: What is the effect of reduced supervision?

CuBERT achieves results comparable to the baselines with 1/3rd or 2/3rd of training data, and within 2 or 10 fine-tuning epochs (the default being 20 epochs).

Experimental Results

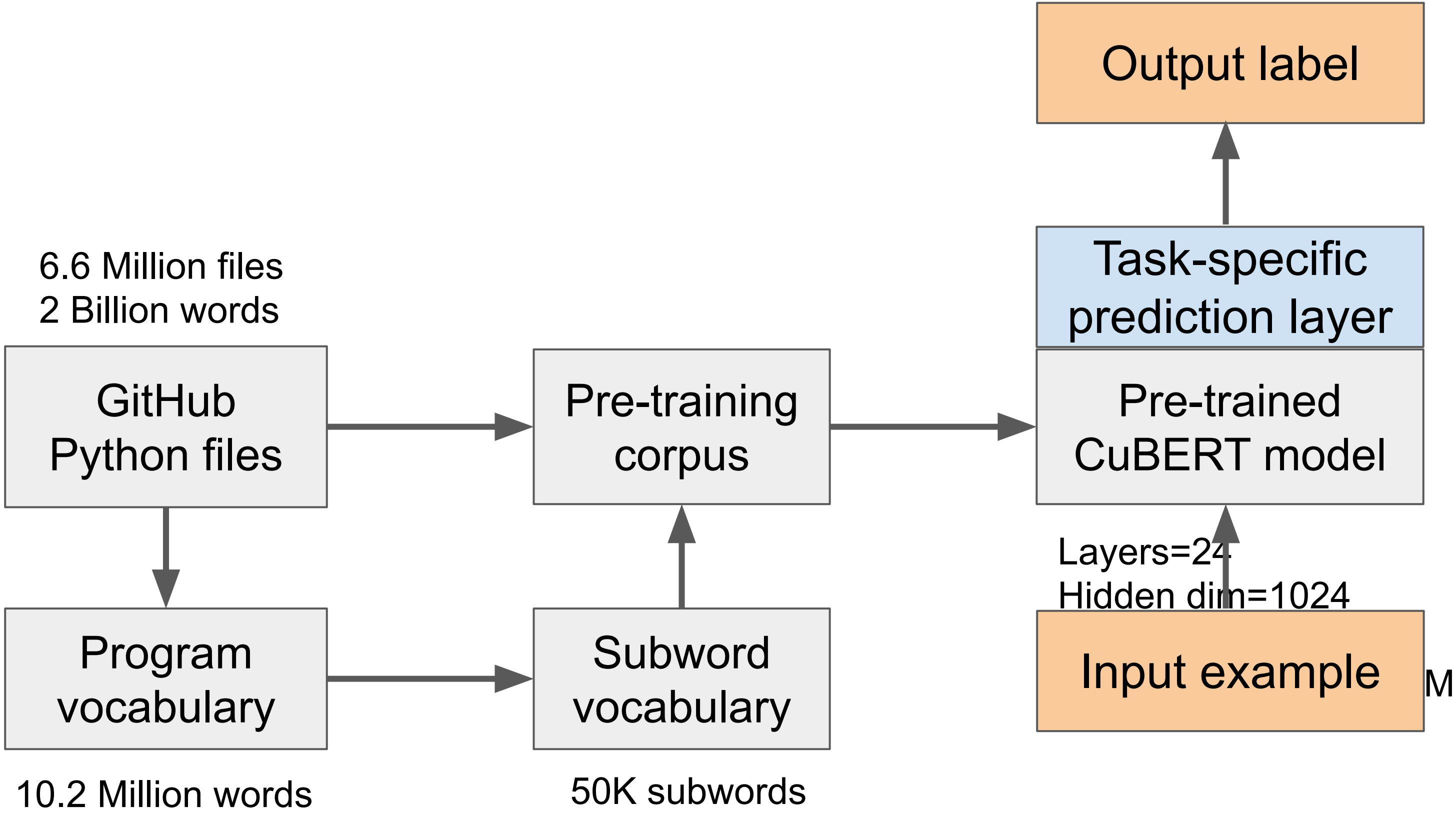
Q4: How does the context length affect CuBERT's performance?

Increasing context length (128 -> 256 -> 512) tends to improve the performance.

Q5: How does CuBERT perform on the more complex task of predicting a two-headed pointer in comparison to SOTA approaches?

CuBERT achieves +33% (absolute) localization+repair accuracy in comparison to (Vasic et al. 2019) and +6.2% (absolute) in comparison to (Hellendoorn et al., 2020) on the corresponding datasets.

Experimental Setup



New Benchmark of Code-understanding Tasks

Built using the ETH Py150 corpus (Raychev et al. 2016).

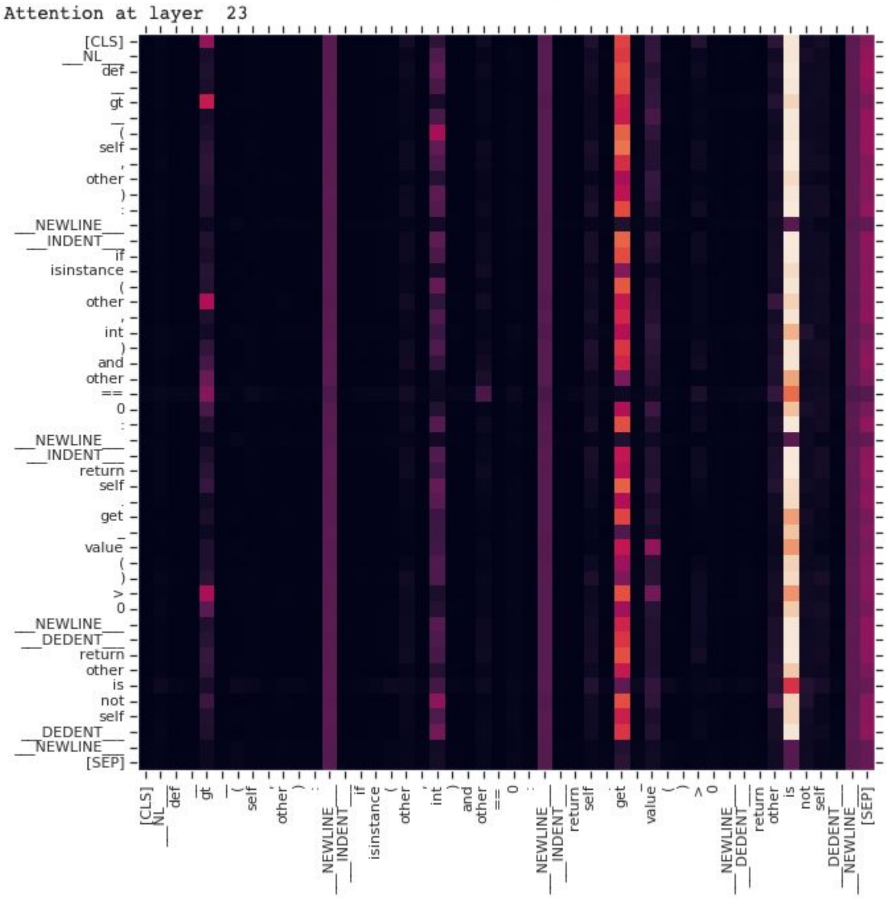
Motivated in part by code-understanding tasks studied in the literature.

- Swapped operands (binary classification) (Pradel & Sen 2018)
- Wrong binary operator (binary classification) (Pradel & Sen 2018)
- Exception-type (multi-class classification)
- Function-docstring mismatch (sentence-pair classification) (Louis et al. (2018)
- Variable-misuse (binary classification) (Allamanis et al. 2018)
- Variable-misuse localization and repair (multi-headed pointer prediction) (Vasic et al. 2019)

Example of Wrong Binary Operator Classification

```
def__gt__(self,other):
  if isinstance(other,int) and other==0:
    return self.get_value()>0
  return other is not self
```

Correct operator: <



Visualization of attention weights at the last layer

Example of Exception Type Classification

```
try:
  subprocess.call(hook_value)
  return jsonify(success=True), 200
except __HOLE__ as e:
  return jsonify(success=False,
                 error=str(e)), 400
```

Expected label: `OSError`

Multi-class classification with 20 top exception types as class labels.

Example of Function Docstring Classification

```
Sentence #1:
'Get form initial data.'

Sentence #2:
def __add__(self, cov):
  return SumOfKernel(self, cov)
```

Sentence-pair classification problem

Example of Variable Misuse Tasks

Repair pointer

```
def on_resize(self, event):
  event.apply_zoom()
```

Localization pointer

Variable `event` is used incorrectly instead of `self`.

Dealing with Code Duplicates

Open-source projects are replete with code duplicates. This can:

- Affect the reported model performance.
- Result in information leak between pre-training and fine-tuning corpora.
- Bias pre-training towards duplicated code.

Remedy code duplication by:

- Deduplicating the fine-tuning corpus in the fashion of Allamanis (2018) using Jaccard similarity over sets/multi-sets of tokens.
- Remove files with duplicates in the fine-tuning corpus from pre-training.
- Deduplicate the pre-training corpus.

Related Work

Representation learning for programs

- Structured representations like abstract syntax trees (Alon et al., 2019) and data-flow/control-flow information (Allamanis et al., 2018; Hellendoorn et al., 2020) used in specific software engineering tasks.
- An upcoming work by Feng et al. (2020) aims at solving NL-PL tasks by pre-training a BERT model on paired NL description and code, in a multi-lingual setting. CuBERT pre-training and fine-tuning (e.g., function-docstring task) also involves both code and natural language.

Conclusions and Future Work

We present the first pre-trained contextual embedding of source code.

Our model, CuBERT, shows strong performance against baselines.

We hope that our models and benchmarks will be useful to the community.

Pre-training using structured representations of code, such as ASTs and graphs, that encode different types of information (e.g., data-flow and control-flow) will be an interesting future direction.

We envision more innovations on the pre-training setup, reduction in model size and pre-training cost, and novel applications of the pre-trained models.