Learning Self-Supervised Representations of Code Functionality

Paras Jain
Ajay Jain
Tianjun Zhang
Pieter Abbeel
Joseph Gonzalez
Ion Stoica

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2021-62
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2021/EECS-2021-62.html

May 13, 2021

Learning Self-Supervised Representations of Code Functionality

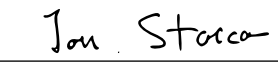
Paras Jagdish Jain

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee

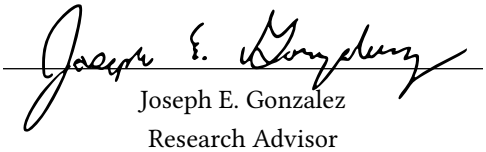


Ion Stoica
Research Advisor

5/12/2021

(Date)

★ ★ ★ ★ ★ ★ ★



Joseph E. Gonzalez
Research Advisor

5/12/21

(Date)

Learning Self-Supervised Representations of Code Functionality

Paras Jain [*1] **Ajay Jain** [*1] **Tianjun Zhang** [1] **Pieter Abbeel** [1] **Joseph E. Gonzalez** [1] **Ion Stoica** [1]

Abstract

Recent work learns contextual representations of source code by reconstructing tokens from their context. For downstream semantic understanding tasks like summarizing code in English, these representations should ideally capture program functionality. However, we show that the popular reconstruction-based BERT model is sensitive to source code edits, *even when the edits preserve semantics*. We propose ContraCode: a contrastive pre-training task that learns code functionality, not form. ContraCode pre-trains a neural network to identify functionally similar variants of a program among many non-equivalent distractors. We scalably generate these variants using an automated source-to-source compiler as a form of data augmentation. Contrastive pre-training improves JavaScript summarization and TypeScript type inference accuracy by 2% to 13%. We also propose a new zero-shot JavaScript code clone detection dataset, showing that ContraCode is both more robust and semantically meaningful. On it, we outperform RoBERTa by 39% AUROC in an adversarial setting and up to 5% on natural code.

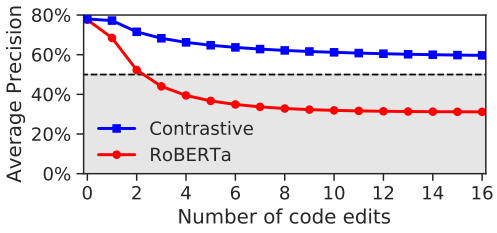


Figure 1. **Robust code clone detection:** When trained on source code, **BERT is not robust to simple label-preserving code edits** like renaming variables. Adversarially selecting between possible edits lowers performance below random guessing. Contrastive pre-training with ContraCode learns a more robust representation of functionality that is consistent across code transforms.

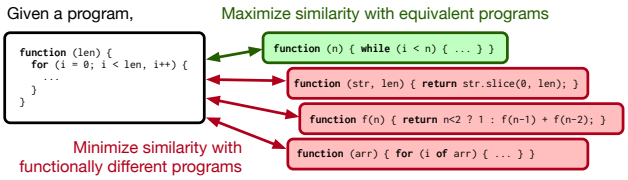


Figure 2. For many learned analyses, programs with the same functionality should have similar representations. ContraCode learns such representations by pre-training an encoder to retrieve equivalent, transformed programs among many distractors.

1. Introduction

Programmers increasingly rely on machine-aided programming tools to aid software development (Kim et al., 2012). These tools analyze or transform code automatically. Traditionally, code analysis uses hand-written rules, though the wide diversity of programs encountered in practice can limit their generality. Recent work uses machine learning to improve performance through richer language understanding, such as learning to detect bugs (Pradel & Sen, 2018) and predict performance (Mendis et al., 2019).

Still, program datasets suffer from scarce annotations due to the time and expertise needed to label code. Synthetic auto-generated labels are used for method naming (Alon et al., 2019a;b) and bug detection (Ferenc et al., 2018; Pradel & Sen, 2018; Benton et al., 2019). However, synthetic code datasets have duplication issues (Allamanis, 2019) and biases (Shin et al., 2019) that can degrade generalization. Moreover, supervised models are not robust to adversarial code edits, suffering significant accuracy loss when variables are renamed (Yefet et al., 2019), statements are permuted, or other semantics-preserving transformations are applied (Wang & Christodorescu, 2019; Wang & Su, 2019; Rabin & Alipour, 2020).

In contrast, self-supervised models can acquire knowledge from large open-source repositories such as GitHub without annotations. Inspired by the success of pre-training in natural language processing, Ben-Nun et al. (2018) use self-supervision to learn code token embeddings like word2vec. More recently, the popular BERT model family (Devlin et al., 2018) has been applied to code (Kanade et al., 2020; Feng et al., 2020; Guo et al., 2020). BERT pre-trains a

[*]Equal contribution [1]University of California, Berkeley. Correspondence to: Paras Jain <parasj@berkeley.edu>, Ajay Jain <ajayj@berkeley.edu>.

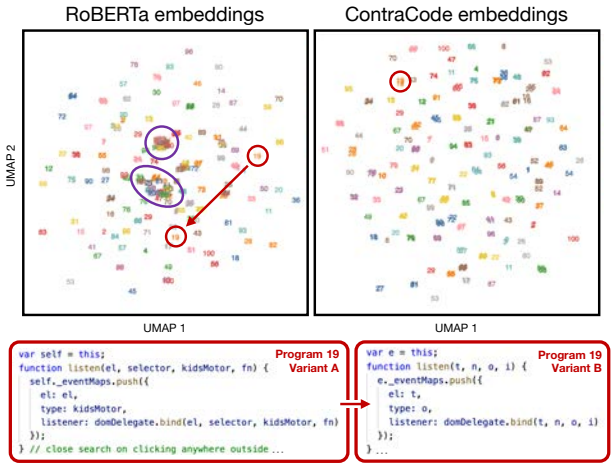


Figure 3. A UMAP visualization of JavaScript method representations learned by RoBERTa and ContraCode, in $\mathbb{R}^2$. Programs with the same functionality share color and number. RoBERTa's embeddings often do not cluster by functionality, suggesting that it is sensitive to implementation details. For example, many different programs **overlap**, and renaming the variables of Program 19 significantly **changes the embedding**. In contrast, variants of Program 19 cluster in ContraCode's embedding space.

Transformer (Vaswani et al., 2017) on programs by reconstructing masked or replaced tokens from context. This objective is called masked language modeling (MLM).

However, we find that BERT is sensitive to implementation details. Figure 1 shows the performance of two self-supervised models on a binary classification task: detecting whether two programs solve the same problem. We mine these programs from the HackerRank interview preparation website. While BERT has comparable performance on the original user-submitted programs (0 edits), BERT's performance greatly degrades when the programs are adversarially transformed *e.g.* by renaming variables and deleting dead code (1-16 edits). These transforms do not change the functionality of the programs, so are label-preserving. Qualitatively, program representations are not invariant to edits (Figure 3). This could be because accurate reconstructions during pre-training mostly depend on syntactic and program implementation details.

Motivated by the sensitivity of supervised learning and reconstruction-based pre-training, we develop ContraCode: a self-supervised learning algorithm that explicitly optimizes for representations of program functionality. We hypothesize that *programs with the same functionality should have similar underlying representations* for downstream code understanding tasks. ContraCode generates syntactically diverse but functionally similar programs with source-to-source compiler transformation techniques (*e.g.*, dead code elimination, obfuscation and constant folding). ContraCode uses these programs in a challenging discriminative pretext task that requires the model to identify equivalent programs out of a large dataset of distractors, illustrated in Figure 2. To solve this task, the model has to embed code semantics rather than syntax. In essence, we specify domain knowledge about desired invariances through code transformations. ContraCode improves robustness even under the most adversarial setting in Figure 1, and consistently improves downstream code understanding on other tasks. The contributions of our work include:

1. the novel use of compiler-based transformations as data augmentations for code,
2. the concept of program representation learning based on functional equivalence, and
3. a detailed analysis of architectures, code transforms and pre-training strategies, showing ContraCode improves type inference top-1 accuracy by 9%, learned inference by 2%–13%, summarization F1 score by up to 8% and clone detection AUROC by 2%–46%.

2. Related Work

Self-supervised learning (SSL) is a general representation learning strategy where some dimensions or attributes of a datapoint are predicted from the remaining parts. These methods are unsupervised in the sense that they do not rely on labels, but SSL tasks often adapt losses and architectures designed for supervised learning. Self-supervised pre-training has yielded large improvements in both NLP (Howard & Ruder, 2018; Devlin et al., 2018; Radford et al., 2018; 2019) and computer vision (Mahajan et al., 2018) by improving generalization (Erhan et al., 2010; Hao et al., 2019). Weak visual features, such as orientation (Gidaris et al., 2018), color (Zhang et al., 2016), and context (Pathak et al., 2016), are meaningful signals for representations (Mahajan et al., 2018).

Contrastive learning unifies many past SSL approaches that compare pairs or collections of similar and dissimilar items (Hadsell et al., 2006). Rather than training the network to predict labels or reconstruct data, contrastive methods minimize the distance between the representations of similar examples (positives) while maximizing the distance between dissimilar examples (negatives). Examples include Siamese networks (Bromley et al., 1994) and triplet losses (Schroff et al., 2015). Contrastive predictive coding (Oord et al., 2018; Hénaff et al., 2019) learns to encode chunks of sequential data to predict future chunks with the InfoNCE loss, a variational lower bound on mutual information between views of the data (Tian et al., 2019; Wu et al., 2020) inspired by noise-constrastive estimation (Gutmann & Hyvärinen, 2010). In instance discrimination tasks (Wu

et al., 2018), views and not pieces of an entire image are compared. SimCLR (Chen et al., 2020a) and Momentum Contrast (He et al., 2019; Chen et al., 2020b) recently made progress by using many negatives for dense loss signal. Beyond images, InfoNCE has been applied to NLP (Chuang et al., 2020; Giorgi et al., 2020), but may require supervision (Fang & Xie, 2020).

Code representation learning Many works apply machine learning to code (Allamanis et al., 2018). We address code clone detection (White et al., 2016), variable type inference (Hellendoorn et al., 2018), and summarization (Alon et al., 2019a). Others have also explored ML for summarization (Movshovitz-Attias & Cohen, 2013; Allamanis et al., 2016; Iyer et al., 2016) and type inference (Pradel et al., 2019; Pandi et al., 2020; Wei et al., 2020; Allamanis et al., 2020; Bielik & Vechev, 2020) with various languages and datasets. The tree or graph structure of code can be exploited to encode invariances in the representation. Inst2vec (Ben-Nun et al., 2018) locally embeds individual statements in LLVM IR by processing a contextual flow graph with a context prediction objective (Mikolov et al., 2013). Tree-Based CNN embeds the Abstract Syntax Tree (AST) nodes of high-level source code. Code2seq (Alon et al., 2019a) embeds AST paths with an attention-based encoder and LSTM decoder for supervised sequence-to-sequence tasks. Kanade et al. (2020) and Feng et al. (2020) pre-train a Transformer (Vaswani et al., 2017) on code using variants of the masked language modeling objective (Devlin et al., 2018), an instance of the cloze task (Taylor, 1953) for reconstructing corrupted tokens. Recurrent networks have also been pre-trained on code (Hussain et al., 2020) as language models (Peters et al., 2018; Karampatsis & Sutton, 2020).

3. Approach

Understanding global program functionality is important for difficult semantic tasks. For these problems, learned representations should be similar for functionally equivalent programs and dissimilar for non-equivalent programs. The principle of contrastive learning offers a simple approach for learning such representations if data can be organized into pairs of *similar positives* and *dissimilar negatives* (Arora et al., 2019). We use these to shape representation space, drawing positives together and pushing negatives apart. A major question remains: *given an unlabeled corpus of programs, how do we identify or generate similar programs for positives?* We address this question in §3.1 and §3.2, then introduce our learning framework in §3.3.

3.1. Compilation as data augmentation

Modern programming languages afford great flexibility to software developers, allowing them to implement the same desired functionality in different ways. Crowdsourced datasets mined from developers, such as GitHub repositories, have many near-duplicates in terms of textual similarity (Allamanis, 2019), and are bound to contain even more functional equivalences for common tasks. Satisfiability solvers can identify these equivalent programs (Joshi et al., 2002; Bansal & Aiken, 2006), but functional equivalence is undecidable in general (Rice, 1953). Also, formal documentation of semantics is required. Programs can instead be compared approximately using test-cases (Massalin, 1987), but this is costly and requires executing untrusted code.

	Code compression		**Identifier modification**
✓	Reformatting (R)	✓	Variable renaming (VR)
✓	Beautification (B)	✓	Identifier mangling (IM)
✓	Compression (C)		**Regularization**
✓	Dead-code elimination (DCE)	✓	Dead-code insertion (DCI)
✓	Type upconversion (T)	✓	Subword regularization (SW)
✓	Constant folding (CF)	✗	Line subsampling (LS)

✓ = semantics-preserving transformation ✗ = lossy transformation

Table 1. We augment programs with 11 automated source-to-source compiler transformations. 10 are correct-by-construction and preserve operational semantics. More details are in Sec. A.

Instead of searching for equivalences, we propose correct by construction data augmentation. Our insight is to apply source-to-source compiler transformations to unlabeled code to generate many variants with the same functionality. For example, dead-code elimination (DCE) is a common compiler optimization that removes operations that leave the output of a function unchanged. While DCE preserves program functionality, Wang & Christodorescu (2019) find that up to 12.7% of the predictions of current supervised algorithm classification models change after DCE. Supervised datasets were insufficient to acquire the domain knowledge that DCE does not change the algorithm.

We unambiguously parse a particular source code sequence, e.g. `W*x + b` into a tree-structured representation `(+ (* W x) b)` called an Abstract Syntax Tree (AST). This tree is then transformed by automated traversal algorithms. A rich body of prior programming language work explores parsing then transforming ASTs to optimize a program prior to machine code generation. If source code is emitted by the compiler rather than machine code, this is called source-to-source transformation. Source-to-source transformations are common for optimization and obfuscation purposes in dynamic languages like JavaScript. Further, if each transformation preserves code functionality, then any composition also preserves code functionality.

We leverage the Babel and Terser compiler infrastructure tools for JavaScript (McKenzie et al., 2020; Santos et al., 2020) to perform different transformations on method bodies. Example transformations are shown in Figure 4. Table 1 and the supplement list all transformations, but we broadly

```
function x(maxLine) {
  const section = {
    text: '',
    data
  };

  for (; i < maxLine; i += 1) {
    section.text += `${lines[i]}\n`;
  }

  if (section) {
    parsingCtx.sections.push(section);
  }
}
```

Original JavaScript method

```
function x(t) {
  const n = {
    'text': '',
    'data': data
  };
  for (;i < t; i += 1) {
    n.text += lines[i] + '\n';
  }
  n && parsingCtx.sections.push(n);
}
```

Renamed variables, explicit object style, explicit concatenation, inline conditional

```
function x(t){const
n={'text':'','data':data};for(;i<t;i+=
1)n.text+=lines[i]
+'\n';n&&parsingCtx.sections.push(n)}
```

Mangled source with compressed whitespace

Figure 4. A JavaScript method from our unlabeled training set with two automatically generated semantically-equivalent programs. The method is from the StackEdit Markdown editor.

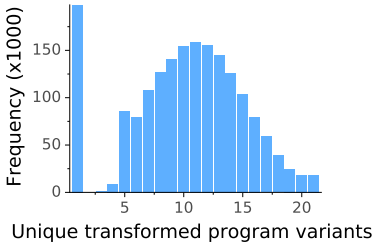


Figure 5. Histogram of the number of unique transformed variants per JavaScript method during pre-training.

group program transformations into three categories.

Code compression passes change the syntactic structure of code and perform correct-by-construction transformations such as pre-computing constant expressions at compile time. **Identifier modification** transformations substitute method and variable names with random or short tokens, masking part of the human-readable information in a program but leaving its functionality unchanged. Finally, transformations for **Regularization** improve model generalization by reducing the number of trivial positive pairs with high text overlap. The line subsampling pass in this group potentially modifies program semantics.

3.2. Transformation dropout for view diversity

Computer vision datasets are often augmented with altered images like crops. Back-translations have been used as data augmentations for natural language (Sennrich et al., 2016). Similarly, each compiler transformation is an augmentation to a program. Each transformation is a function $\tau : \mathcal{P} \to \mathcal{P}$, where the space of programs $\mathcal{P}$ is composed of both the set of valid ASTs and the set of programs in source form.

Stochastic augmentations like random crops generate many views of an image, but most of our compiler-based transformations are deterministic. To produce a diverse set of transformed programs, we randomly apply a subset of available compiler passes in a pre-specified order, applying transform τ_i with probability p_i. Intermediate programs are converted between AST and source form as needed for the compiler. Algorithm 1 details our transformation dropout procedure.

Figure 5 measures the resulting diversity in programs. We precompute up to 20 augmentations of 1.8M JavaScript methods from GitHub. Algorithm 1 deduplicates method variants before pre-training since some transforms will leave the program unchanged. 89% of the methods have more than one alternative after applying 20 random sequences of transformations. The remaining methods without syntactically distinct alternatives include one-line functions that are obfuscated. We apply subword regularization (Kudo, 2018) as a final transformation to derive different tokenizations every batch, so pairs derived from the same original method will still differ. All transformations are fast; our compiler transforms 300 functions per second on a single CPU core.

Algorithm 1 Transformation dropout: Stochastic program augmentation with two encodings (AST or source).

Input: Program source x, transformation functions $\tau_1, \ldots \tau_k$, transform probabilities $p_1, \ldots p_k$, count N
Returns: N variants of x
$\mathcal{V} \leftarrow \{x\}$, a set of augmented program variants
for SAMPLE $i \leftarrow 1 \ldots N-1$ **do**
  $x' \leftarrow x$
  for transform $t \leftarrow 1 \ldots k$ **do**
    Sample $y_t \sim$ Bernoulli(p_t)
    if $y_t = 1$ **then**
      if REQUIRESAST$(\tau_t(\cdot))$ and $\neg$ISAST(x') **then** $x' \leftarrow$ PARSETOAST(x')
      else if $\neg$REQUIRESAST$(\tau_t(\cdot))$ and ISAST(x') **then** $x' \leftarrow$ LOWERTOSOURCE(x')
      $x' \leftarrow \tau_t(x')$
    end if
  end for
  if ISAST(x') **then** $x' \leftarrow$ LOWERTOSOURCE(x')
  $\mathcal{V} \leftarrow \mathcal{V} \cup \{x'\}$
end for
return $\mathcal{V}$

3.3. Learning an encoder with contrastive pre-training

While BERT pre-trains a neural program encoder by reconstructing tokens (a generative task), we apply constrative learning to code by shaping the representation at the method level. Contrastive learning is a natural framework to induce invariances into a model by attracting positives while repelling negatives. To adapt recent contrastive learning objectives for images to code representation learning, we leverage the augmentations discussed in Section 3.1-3.2 to define the positive program pairs. Dissimilar negatives are randomly sampled from other programs. We extend the Momentum Contrast method (He et al., 2019) that was designed for image representation learning. In our case, we learn a program encoder f_q that maps a sequence of program tokens to a single, fixed dimensional embedding. This embedding is projected with a small MLP before computing

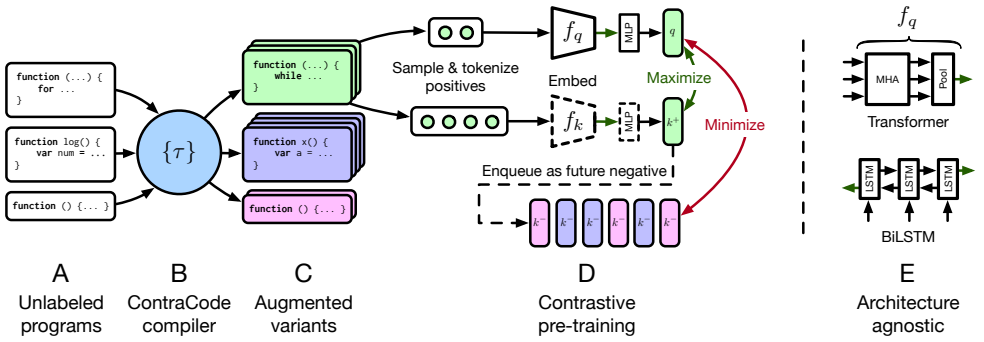


Figure 6. ContraCode pre-trains a neural program encoder f_q and transfers it to downstream tasks. **A-B.** Unlabeled programs are transformed **C.** into augmented variants. **D.** We pre-train f_q by maximizing similarity of projected embeddings of *positive* program pairs–variants of the same program–and minimizing similarity with a queue of cached negatives. **E.** ContraCode supports any architecture for f_q that produces a global program embedding such as Transformers and LSTMs. f_q is then fine-tuned on smaller labeled datasets.

the pre-training objective.

Pre-training objective The contrastive objective maximizes the similarity of positives without collapsing onto a single representation. Like He et al. (2019), we use InfoNCE (Oord et al., 2018), a tractable objective that frames contrastive learning as a classification task: can the positives be identified among a batch of sampled negatives? InfoNCE computes the probability of selecting the positive (transformed program) by taking the softmax of projected embedding similarities across a batch of negatives. Eq. (1) shows the InfoNCE loss for instance discrimination, a function whose value is low when q is similar to the positive key embedding k^+ and dissimilar to negative key embeddings k^-. t is a temperature hyperparameter (Wu et al., 2018).

$$\mathcal{L}_{q,k^+,k^-} = -\log \frac{\exp(q \cdot k^+/t)}{\exp(q \cdot k^+/t) + \sum_{k^-} \exp(q \cdot k^-/t)} \tag{1}$$

The query representation $q = f_q(x^q)$ is computed by the encoder network f_q, and x^q is a query program. Likewise, $k = f_k(x^k)$ using the EMA key encoder f_k. Views x^q, x^k depend on the specific domain and pretext task. In our case, the views are tokenized representations of the augmented programs, and the summation $\sum_{k^-}$ in the normalizing denominator is taken over the queue of pre-computed negatives as well as other non-matching keys in the batch.

To reduce memory consumption, we enqueue past batches to cache activations for negative samples. These cached samples are valid negatives if the queue is smaller than the dataset size. Following He et al. (2019), the query encoder f_q is trained via gradient descent while the key encoder f_k is trained slowly via an exponential moving average (EMA) of the query encoder parameters. The EMA update stabilizes the pre-computed key embeddings across training iterations. Since keys are only embedded once per epoch, we use a very large set of negatives, over 100K, with minimal additional computational cost and no explicit hard negative mining.

ContraCode is agnostic to the architecture of the program encoder f_q. We evaluate contrastive pre-training of 6-layer Transformer (Vaswani et al., 2017) and 2-layer BiLSTM (Schuster & Paliwal, 1997; Huang et al., 2015) architectures, with specific details in Section 4.

Transfer learning After pre-training converges, the encoder f_q is transferred to downstream tasks. For code clone detection, we transfer the representation $f_q(x)$ in zero-shot, without fine-tuning. For tasks where the output space differs from the encoder, we add a task-specific MLP or Transformer decoder after f_q, then fine-tune the resulting network end-to-end on labeled task data.

4. Evaluation

We evaluate whether self-supervised pre-training with ContraCode improves JavaScript and TypeScript code analysis through (1) code clone detection (Baker, 1992), (2) extreme code summarization (Allamanis et al., 2016) and (3) type inference (Hellendoorn et al., 2018) tasks.

Clone detection experiments show that contrastive and hybrid representations with our compiler-based augmentations are predictive of program functionality in-the-wild, and that contrastive representations are the most robust to adversarial edits (§4.1). Contrastive pre-training outperforms baseline supervised and self-supervised methods on all three tasks (§4.1-4.3). Finally, ablations suggest it is better to augment unlabeled programs during pre-training rather than augmenting smaller supervised datasets (§4.4).

Experimental setup Models are pre-trained on CodeSearchNet, a large corpus of methods extracted from popular GitHub repositories (Husain et al., 2019). CodeSearchNet contains 1,843,099 JavaScript programs. Only 81,487 methods have both a documentation string and a method name. The asymmetry between labeled and unlabeled programs stems from JavaScript coding practices where anonymous

Table 2. **Zero-shot code clone detection** with cosine similarity probe. Contrastive and hybrid representations improve clone detection AUROC on unmodified (natural) HackerRank programs by $+8\%$ and $+10\%$ AUROC over a heuristic textual similarity probe, respectively, suggesting they are predictive of functionality. Contrastive representations are also the most robust to adversarial code transformations.

	Natural code		**Adversarial** (N=4)		**Adversarial** (N=16)	
	AUROC	**AP**	**AUROC**	**AP**	**AUROC**	**AP**
Edit distance heuristic	69.55±0.81	73.75	31.63±0.82	42.85	12.11±0.54	32.46
Randomly initialized Transformer	72.31±0.79	75.82	22.72±0.20	37.73	3.09±0.28	30.95
+ RoBERTa MLM pre-train	74.04±0.77	77.65	25.83±0.21	39.46	4.51±0.33	31.17
+ ContraCode pre-train	75.73±0.75	78.02	**64.97**±0.24	**66.23**	**58.32**±0.88	**59.66**
+ ContraCode + RoBERTa MLM	**79.39**±0.70	**81.47**	37.81±0.24	51.42	10.09±0.50	32.52

functions are widespread. The pre-training dataset described in Section 3.1 is the result of augmenting all 1.8M programs.

As our approach supports any encoder, we evaluate two architectures: a 2-layer Bidirectional LSTM with 18M parameters, similar to the supervised model used by Hellendoorn et al. (2018), and a 6-layer Transformer with 23M parameters. For a baseline self-supervised approach, we pre-train both architectures with the RoBERTa MLM objective, then transfer it to downstream tasks.

4.1. Evaluating Functionality and Robustness: Zero-shot Code Clone Detection

ContraCode learns to match variants of programs with similar functionality. While transformations produce highly diverse token sequences (quantified in the supplement), they are artificial and do not change the underlying algorithm. Human programmers can solve a problem with many data structures, algorithms and programming models. Are pre-trained representations consistent across programs written by different people? We benchmark on *code clone detection*, a binary classification task to detect whether two programs solve the same problem or different ones. This is useful for deduplicating, refactoring and retrieving code, as well as checking approximate code correctness.

Datasets exist like BigCloneBench (Svajlenko et al., 2014), but to the best of our knowledge, there is no benchmark for the JavaScript programming language. We collected 274 in-the-wild JavaScript programs correctly solving 33 problems from the HackerRank interview preparation website. There are 2065 pairs solving the same problem and 70K pairs solving different problems, which we randomly subsample to 2065 to balance the classes. Since we probe zero-shot performance based on pre-trained representations, there is no training set. Instead, we threshold cosine similarity of pooled representations of the programs u and v: $u^T v / \|u\| \|v\|$. Many traditional code analysis methods for clone detection measure textual similarity (Baker, 1992). As a baseline heuristic, we threshold the dissimilarity score, a scaled Levenshtein edit distance between normalized and tokenized programs that excludes formatting changes.

Table 2 reports the area under the ROC curve (AUROC) and average precision (AP, area under Precision-Recall). All continuous representations improve clone detection over the heuristic on natural code. Self-supervision through RoBERTa MLM pre-training improves over a randomly initialized network by +1.7% AUROC. Contrastive pre-training achieves +3.4% AUROC over the same baseline. A hybrid objective combining both the contrastive loss and MLM has the best performance with +7.0% AUROC (+5.4% over MLM alone). This indicates that ContraCode learns a a more useful representation of functionality than MLM, though both objectives are useful for natural code.

However, are these representations robust to code edits? We adversarially edit one program in each pair by applying the loss-maximizing code compression and identifier modification transformation among N samples from Algorithm 1. These transformations preserve program functionality, so ground-truth labels are unchanged. With only 4 possible edits, RoBERTa performs worse than the heuristic (-5.8% AUROC) and worse than random guessing (50% AUROC), indicating it is highly sensitive to these kinds of implementation details. ContraCode retains much of its performance (+39% AUROC over RoBERTa) as it explicitly optimizes for invariance to code edits. Surprisingly, the hybrid model is less robust than ContraCode alone, perhaps indicating that MLM learns non-robust features (Ilyas et al., 2019).

4.2. Fine-tuning for Type Inference

JavaScript is a dynamically typed language, where variable types are determined at runtime based on the values they represent. Manually annotating code with types helps tools flag possible bugs before runtime by detecting incompatible types. Annotations also help programmers document code. However, annotations are tedious to maintain. Type inference tools automatically predict types from context.

To *learn* to infer types, we use the same annotated dataset of TypeScript programs from DeepTyper (Hellendoorn et al., 2018), without GitHub repositories that were made private or deleted since publication. The training set contains 15,570 TypeScript files from 187 repositories with

Table 3. **Type inference accuracy on TypeScript programs** in the Hellendoorn et al. (2018) dataset. ContraCode (BiLSTM) outperforms baseline top-1 accuracies by 2.28% to 13.16%. As ContraCode does not modify model architecture, contrastive pre-training can be combined with each baseline. Compared with TypeScript's built-in type inference, we improve accuracy by 8.9%.

Baseline	Method	Acc@1	Acc@5
Static analysis	TypeScript CheckJS (Bierman et al., 2014)	45.11%	—
	Name only (Hellendoorn et al., 2018)	28.94%	70.07%
Transformer	Transformer (supervised)	45.66%	80.08%
	+ ContraCode pre-train	**46.86%**	**81.85%**
RoBERTa-6	Transformer (RoBERTa MLM pre-train)	40.85%	75.76%
	+ ContraCode pre-train (hybrid)	**47.16%**	**81.44%**
DeepTyper (BiLSTM)	DeepTyper (supervised)	51.73%	82.71%
	+ RoBERTa MLM pre-train (10K steps)	50.24%	82.85%
	+ ContraCode pre-train	52.65%	84.60%
	+ ContraCode pre-train (+ SW reg ft)	**54.01%**	**85.55%**

6,902,642 total tokens. Validation and test sets are from held-out repositories. For additional supervision, missing types are inferred by static analysis to augment user-defined types as targets. All types are removed from model input. A 2-layer MLP head predicts types from output token embeddings. We perform early stopping based on validation set top-1 accuracy. For our remaining experiments, the baseline RoBERTa models are pre-trained on the same augmented dataset as ContraCode for fair comparison.

Benefiting from unlabeled JavaScript programs is challenging because TypeScript is a different dialect. TypeScript supports a superset of JavaScript's grammar, adding type annotations and syntactic sugar that must be learned during fine-tuning. Further, the pre-training dataset contains methods, while DeepTyper's dataset includes entire modules.

Table 3 summarizes results. Contrastive pre-training outperforms all baseline learned methods, showing meaningful transfer. ContraCode can be applied in a drop-in fashion to each of the baselines. Simply pre-training each baseline with the contrastive objective and data augmentations yields absolute accuracy improvements of +1.2%, +6.3%, +2.3% top-1 and +1.8%, +5.7%, +2.8% top-5 over the Transformer, RoBERTa, and DeepTyper, respectively.

The RoBERTa baseline may perform poorly since the MLM objective focuses on token reconstruction that is overly sensitive to local syntactic structure, or because sufficient fine-tuning data is available, described as weight "ossification" by Hernandez et al. (2021). To combine the approaches, we minimized our loss in addition to MLM as a hybrid local-global objective to pre-training a Transformer, improving accuracy by +6.31% over the RoBERTa Transformer.

Learning outperforms static analysis by a large margin. Overall, our best model has +8.9% higher top-1 accuracy than the built-in TypeScript CheckJS type inference system, showing the promise of learned code analysis. Surfacing multiple candidate types can also be useful to users. While CheckJS only produces a single prediction, one of our top-5 predictions is correct for 85.6% of labeled tokens.

Table 4. Results for different settings of the **code summarization task**: supervised training with 81K functions, masked language model pre-training, training from scratch and contrastive pre-training with fine-tuning.

Method	Precision	Recall	F1
code2vec (Alon et al., 2019b)	10.78%	8.24%	9.34%
code2seq (Alon et al., 2019a)	12.17%	7.65%	9.39%
RoBERTa MLM (Liu et al., 2019)	15.13%	11.47%	12.45%
Transformer (Vaswani et al., 2017)	18.11%	15.78%	16.86%

4.3. Fine-tuning for Extreme Code Summarization

The extreme code summarization task asks a model to predict the name of a method given its body (Allamanis et al., 2016). These names often summarize the method, such as `reverseString(...)`. Summarization models could help programmers interpret poorly documented code. We create a JavaScript summarization dataset using the 81,487 labeled methods in the CodeSearchNet dataset. The name is masked in the method declaration. A sequence-to-sequence model with an autoregressive decoder is trained to maximize log likelihood of the ground-truth name, a form of abstractive summarization. All models overfit, so stop early according to validation loss. As proposed by Allamanis et al. (2016), we evaluate model predictions by precision, recall and F1 scores over the set of method name tokens.

Table 4 shows code summarization results in four settings: (1) supervised training using baseline tree-structured architectures that analyze the AST (code2vec, code2seq), (2) pre-training on all 1.8M programs using MLM followed by fine-tuning on the labeled programs (RoBERTa), (3) training a supervised Transformer from scratch and (4) contrastive pre-training followed by fine-tuning with augmentations.

Contrastive pre-training outperforms code2seq by +8.2% test precision, +7.3% recall, and +7.9% F1 score. The tree-based code2seq architecture is a way to encode code-specific invariances into the model, while contrastive pre-training learns invariances through data augmentation. ContraCode outperforms self-supervised pre-training with RoBERTa by +4.8% F1. ContraCode also achieves slightly higher performance than the Transformer learned from scratch with the same network architecture. While this improvement is relatively smaller, code summarization is a difficult task. Naming conventions are not consistent between programmers, and the metric measures exact token matches.

Table 5. On two tasks, compiler data augmentations degrade performance when training supervised models *from scratch*.

Method for code summarization	F1
Transformer (Table 4)	**16.86**
+ LS,SW,VR,DCI augmentations	15.65

Method for type inference	Acc@1
Transformer (Table 3)	**45.66**
+ SW regularization	43.96
+ LS,SW augmentations	44.14
DeepTyper (Table 3)	**51.73**
+ SW regularization	49.93
+ LS,SW augmentations	50.93
+ stronger LS,SW augmentations	50.33

4.4. Understanding the importance of augmentation

We analyze the effect of our proposed augmentations on supervised learning from scratch. We then study the importance of individual augmentations during pre-training.

Supervised learning with data augmentation As a baseline, we re-train models from scratch with compiler transforms during *supervised learning* rather than pre-training. Data augmentation artificially expands labeled training sets. For sequence-to-sequence summarization, we apply a variety of augmentations; these all preserve the method name label. For type inference, labels are aligned to input tokens, so they must be realigned after transformation. We only apply token-level transforms as we can track label locations.

Table 5 shows results. Compiler-based data augmentations degrade supervised models, perhaps by creating a training distribution not reflective of evaluation programs. However, as shown in §4.1–4.3, augmenting during ContraCode pre-training yields a more accurate model. Our contrastive learning framework also allows learning over large numbers of unlabeled programs that supervised learning alone cannot leverage. The ablation indicates that augmentations do not suffice, and contrastive learning is important.

Ablating pre-training augmentations Some data augmentations could be more valuable than others. Empirically, pre-training converges faster with a smaller set of augmentations at the same batch size since the positives are syntactically more similar, but this hurts downstream performance. Table 6 shows that type inference accuracy degrades when different groups of augmentations are removed. Semantics-preserving code compression passes that require code analysis are the most important, improving top-1 accuracy by 1.95% when included. Line subsampling serves as a regularizer, but changes program semantics. LS is relatively less important, but does help accuracy. Identifier modifications preserve semantics, but change useful naming information. Removing these hurts the least.

Table 6. Ablating compiler transformations used during contrastive pre-training. The DeepTyper BiLSTM is pre-trained with constrastive learning for 20K steps, then fine-tuned for type inference. Augmentations are only used during pre-training. Each transformation contributes to accuracy.

Augmentations used for pre-training	Acc@1	Acc@5
All augmentations (Table 3)	**52.65%**	**84.60%**
w/o identifier modification (-VR, -IM)	51.94%	84.43%
w/o line subsampling (-LS)	51.05%	81.63%
w/o code compression (-T,C,DCE,CF)	50.69%	81.95%

Additional results We perform additional ablations in the supplement by transferring different parts of the network to downstream tasks, computing the contrastive objective with representations taken from different encoder layers, varying architecture, and tuning the pre-training procedure. These experiments suggest that as many parameters as possible should be transferred to the downstream task. Details of the pre-training strategy are also important. For an LSTM, computing the contrastive objective using a global representation q summarizing the whole input sequence x^q outperforms a more local representation based on pooling across tokens. Further, a large batch size is helpful to stabilize pre-training. The supplement also includes qualitative results.

5. Conclusion

Large-scale unannotated repositories of code like GitHub are a powerful resource for learning machine-aided programming tools. However, most current approaches to code representation learning do not leverage unannotated data, and popular self-supervised learning approaches like BERT that learn to reconstruct the text of code are not robust. Instead of reconstructing the text of code, learning *what it says*, we learn *what programs do*. We propose ContraCode, a contrastive self-supervised algorithm that learns representations invariant to code transformations. Our method optimizes for this invariance via novel compiler-based data augmentations for code. In experiments on JavaScript, ContraCode learns effective representations of code functionality, and is robust to adversarial code edits. We find that ContraCode significantly improves performance on three downstream JavaScript code understanding tasks.

Acknowledgments

We thank Koushik Sen, Jonathan Ho, Aravind Srinivas and Rishabh Singh. In addition to NSF CISE Expeditions Award CCF-1730628, the NSF GRFP under Grant No. DGE-1752814, and ONR PECASE N000141612723, this research is supported by gifts from Amazon Web Services, Ant Financial, Ericsson, Facebook, Futurewei, Google, Intel, Microsoft, NVIDIA, Scotiabank, Splunk and VMware.

References

Allamanis, M. The adverse effects of code duplication in machine learning models of code. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2019, pp. 143–153, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450369954. doi: 10.1145/3359591.3359735.

Allamanis, M., Peng, H., and Sutton, C. A convolutional attention network for extreme summarization of source code. In *International Conference on Machine Learning (ICML)*, 2016.

Allamanis, M., Barr, E. T., Devanbu, P., and Sutton, C. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):81, 2018.

Allamanis, M., Barr, E. T., Ducousso, S., and Gao, Z. Typilus: Neural type hints. In *Programming Language Design and Implementation (PLDI)*, 2020.

Alon, U., Brody, S., Levy, O., and Yahav, E. code2seq: Generating sequences from structured representations of code. In *International Conference on Learning Representations*, 2019a.

Alon, U., Zilberstein, M., Levy, O., and Yahav, E. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 2019b.

Arora, S., Khandeparkar, H., Khodak, M., Plevrakis, O., and Saunshi, N. A theoretical analysis of contrastive unsupervised representation learning. In Chaudhuri, K. and Salakhutdinov, R. (eds.), *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pp. 5628–5637. PMLR, 09–15 Jun 2019. URL http://proceedings.mlr.press/v97/saunshi19a.html.

Baker, B. S. A program for identifying duplicated code. *Computing Science and Statistics*, 1992.

Bansal, S. and Aiken, A. Automatic generation of peephole superoptimizers. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pp. 394–403, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595934510. doi: 10.1145/1168857.1168906.

Ben-Nun, T., Jakobovits, A. S., and Hoefler, T. Neural code comprehension: A learnable representation of code semantics. In *NeurIPS*, 2018.

Benton, S., Ghanbari, A., and Zhang, L. Defexts: A curated dataset of reproducible real-world bugs for modern jvm languages. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pp. 47–50. IEEE, 2019.

Bielik, P. and Vechev, M. Adversarial robustness for code. *CoRR*, 2020.

Bierman, G., Abadi, M., and Torgersen, M. Understanding typescript. In *European Conference on Object-Oriented Programming (ECOOP)*, 2014.

Bromley, J., Guyon, I., LeCun, Y., Säckinger, E., and Shah, R. Signature verification using a" siamese" time delay neural network. In *Advances in neural information processing systems*, pp. 737–744, 1994.

Chen, T., Kornblith, S., Norouzi, M., and Hinton, G. A simple framework for contrastive learning of visual representations. In *International Conference on Macnine Learning*, 2020a.

Chen, X., Fan, H., Girshick, R., and He, K. Improved baselines with momentum contrastive learning. *arXiv preprint arXiv:2003.04297*, 2020b.

Chuang, C.-Y., Robinson, J., Yen-Chen, L., Torralba, A., and Jegelka, S. Debiased contrastive learning, 2020.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Annual Conference of the North American Chapter of the Association for Computational Linguistics*, 2018.

Erhan, D., Bengio, Y., Courville, A., Manzagol, P.-A., Vincent, P., and Bengio, S. Why does unsupervised pretraining help deep learning? *Journal of Machine Learning Research*, 11(Feb):625–660, 2010.

Fang, H. and Xie, P. CERT: Contrastive self-supervised learning for language understanding. *arXiv preprint arXiv:2005.12766*, May 2020.

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.

Ferenc, R., Tóth, Z., Ladányi, G., Siket, I., and Gyimóthy, T. A public unified bug dataset for Java. In *Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering*, pp. 12–21, 2018.

Fogarty, J., Baker, R. S., and Hudson, S. E. Case studies in the use of roc curve analysis for sensor-based estimates in human computer interaction. In *Proceedings of Graphics Interface 2005*, GI '05, pp. 129–136, Waterloo, CAN, 2005. Canadian Human-Computer Communications Society. ISBN 1568812655.

Gidaris, S., Singh, P., and Komodakis, N. Unsupervised representation learning by predicting image rotations. *arXiv preprint arXiv:1803.07728*, 2018.

Giorgi, J. M., Nitski, O., Bader, G. D., and Wang, B. DeCLUTR: Deep contrastive learning for unsupervised textual representations. *arXiv preprint arXiv:2006.03659*, 2020.

Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., Zhou, L., Duan, N., Svyatkovskiy, A., Fu, S., Tufano, M., Deng, S. K., Clement, C., Drain, D., Sundaresan, N., Yin, J., Jiang, D., and Zhou, M. Graphcodebert: Pre-training code representations with data flow, 2020.

Gutmann, M. and Hyvärinen, A. Noise-contrastive estimation: A new estimation principle for unnormalized statistical models. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pp. 297–304, 2010.

Hadsell, R., Chopra, S., and LeCun, Y. Dimensionality reduction by learning an invariant mapping. In *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, volume 2, pp. 1735–1742. IEEE, 2006.

Hao, Y., Dong, L., Wei, F., and Xu, K. Visualizing and understanding the effectiveness of bert. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pp. 4134–4143, 2019.

He, K., Fan, H., Wu, Y., Xie, S., and Girshick, R. Momentum contrast for unsupervised visual representation learning. *arXiv preprint arXiv:1911.05722*, 2019.

Hellendoorn, V. J., Bird, C., Barr, E. T., and Allamanis, M. Deep learning type inference. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 152–162, 2018.

Hénaff, O. J., Srinivas, A., Fauw, J. D., Razavi, A., Doersch, C., Eslami, S. M. A., and Oord, A. v. d. Data-efficient image recognition with contrastive predictive coding. In *International Conference on Machine Learning*, 2019.

Hernandez, D., Kaplan, J., Henighan, T., and McCandlish, S. Scaling laws for transfer, 2021.

Howard, J. and Ruder, S. Universal language model fine-tuning for text classification. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics*, pp. 328–339, 2018.

Huang, Z., Xu, W., and Yu, K. Bidirectional lstm-crf models for sequence tagging. *arXiv preprint arXiv:1508.01991*, 2015.

Husain, H., Wu, H.-H., Gazit, T., Allamanis, M., and Brockschmidt, M. CodeSearchNet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019.

Hussain, Y., Huang, Z., Zhou, Y., and Wang, S. Deep transfer learning for source code modeling. *International Journal of Software Engineering and Knowledge Engineering*, 30(05):649–668, May 2020. ISSN 1793-6403. doi: 10.1142/s0218194020500230.

Ilyas, A., Santurkar, S., Tsipras, D., Engstrom, L., Tran, B., and Madry, A. Adversarial examples are not bugs, they are features. In Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems*, volume 32, pp. 125–136. Curran Associates, Inc., 2019.

Iyer, S., Konstas, I., Cheung, A., and Zettlemoyer, L. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 2073–2083, 2016.

Joshi, R., Nelson, G., and Randall, K. Denali: A goal-directed superoptimizer. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pp. 304–314, New York, NY, USA, 2002. Association for Computing Machinery. ISBN 1581134630. doi: 10.1145/512529.512566.

Kanade, A., Maniatis, P., Balakrishnan, G., and Shi, K. Pre-trained contextual embedding of source code. *ArXiv*, abs/2001.00059, 2020.

Karampatsis, R.-M. and Sutton, C. SCELMo: Source code embeddings from language models. *arXiv preprint arXiv:2004.13214*, 2020.

Kim, M., Zimmermann, T., and Nagappan, N. A field study of refactoring challenges and benefits. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pp. 1–11, 2012.

Kudo, T. Subword regularization: Improving neural network translation models with multiple subword candidates. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 66–75, 2018.

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. *arXiv preprint arXiv:1907.11692*, 2019.

Maaten, L. v. d. and Hinton, G. Visualizing data using t-sne. *Journal of machine learning research*, 9(Nov): 2579–2605, 2008.

Mahajan, D., Girshick, R., Ramanathan, V., He, K., Paluri, M., Li, Y., Bharambe, A., and van der Maaten, L. Exploring the limits of weakly supervised pretraining. In *Proceedings of the European Conference on Computer Vision*, pp. 181–196, 2018.

Massalin, H. Superoptimizer: A look at the smallest program. In *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems*, ASPLOS II, pp. 122–126, Washington, DC, USA, 1987. IEEE Computer Society Press. ISBN 0818608056. doi: 10.1145/36206.36194.

McKenzie, S. et al. Babel: compiler for writing next generation javascript. https://github.com/babel/babel, 2020.

Mendis, C., Renda, A., Amarasinghe, S., and Carbin, M. Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks. In *International Conference on Machine Learning*, pp. 4505–4515, 2019.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., and Dean, J. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pp. 3111–3119, 2013.

Movshovitz-Attias, D. and Cohen, W. Natural language models for predicting programming comments. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pp. 35–40, 2013.

Oord, A. v. d., Li, Y., and Vinyals, O. Representation learning with contrastive predictive coding. *arXiv preprint arXiv:1807.03748*, 2018.

Pandi, I. V., Barr, E. T., Gordon, A. D., and Sutton, C. Opttyper: Probabilistic type inference by optimising logical and natural constraints, 2020.

Pathak, D., Krahenbuhl, P., Donahue, J., Darrell, T., and Efros, A. A. Context encoders: Feature learning by inpainting. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2536–2544, 2016.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

Peters, M. E., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K., and Zettlemoyer, L. Deep contextualized word representations. In *Proc. of NAACL*, 2018.

Pradel, M. and Sen, K. Deepbugs: A learning approach to name-based bug detection. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–25, 2018.

Pradel, M., Gousios, G., Liu, J., and Chandra, S. Typewriter: Neural type prediction with search-based validation. *arXiv preprint arXiv:1912.03768*, 2019.

Rabin, M. R. I. and Alipour, M. A. Evaluation of generalizability of neural program analyzers under semantic-preserving transformations, 2020.

Radford, A., Narasimhan, K., Salimans, T., and Sutskever, I. Improving language understanding by generative pre-training. *ArXiv*, 2018.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. Language models are unsupervised multitask learners. *ArXiv*, 2019.

Rice, H. G. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

Santos, F. et al. Terser: Javascript parser, mangler and compressor toolkit for es6+. https://github.com/terser/terser, 2020.

Schroff, F., Kalenichenko, D., and Philbin, J. Facenet: A unified embedding for face recognition and clustering. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 815–823, 2015.

Schuster, M. and Paliwal, K. Bidirectional recurrent neural networks. *Signal Processing, IEEE Transactions on*, 45: 2673 – 2681, 12 1997. doi: 10.1109/78.650093.

Sennrich, R., Haddow, B., and Birch, A. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*, 2015.

Sennrich, R., Haddow, B., and Birch, A. Improving neural machine translation models with monolingual data. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 86–96, Berlin, Germany, August 2016. Association for Computational Linguistics. doi: 10.18653/v1/P16-1009. URL https://www.aclweb.org/anthology/P16-1009.

Shin, R., Kant, N., Gupta, K., Bender, C., Trabucco, B., Singh, R., and Song, D. Synthetic datasets for neural program synthesis. *arXiv preprint arXiv:1912.12345*, 2019.

Svajlenko, J., Islam, J. F., Keivanloo, I., Roy, C. K., and Mia, M. M. Towards a big data curated benchmark of inter-project code clones. In *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*, ICSME '14, pp. 476–480, USA, 2014. IEEE Computer Society. ISBN 9781479961467. doi: 10.1109/ICSME.2014.77. URL https://doi.org/10.1109/ICSME.2014.77.

Taylor, W. L. "Cloze procedure": A new tool for measuring readability. *Journalism Quarterly*, 30(4):415–433, 1953.

Tian, Y., Krishnan, D., and Isola, P. Contrastive multiview coding. *CoRR*, 2019.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. In *Advances in Neural Information Processing Systems*, pp. 5998–6008, 2017.

Wang, K. and Christodorescu, M. COSET: A benchmark for evaluating neural program embeddings. *arXiv preprint arXiv:1905.11445*, 2019.

Wang, K. and Su, Z. Learning blended, precise semantic program embeddings. *ArXiv*, 2019.

Wei, J., Goyal, M., Durrett, G., and Dillig, I. Lambdanet: Probabilistic type inference using graph neural networks. In *International Conference on Learning Representations*, 2020.

White, M., Tufano, M., Vendome, C., and Poshyvanyk, D. Deep learning code fragments for code clone detection. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 87–98. IEEE, 2016.

Wu, M., Zhuang, C., Mosse, M., Yamins, D., and Goodman, N. On mutual information in contrastive learning for visual representations, 2020.

Wu, Z., Xiong, Y., Yu, S. X., and Lin, D. Unsupervised feature learning via non-parametric instance discrimination. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 3733–3742, 2018.

Yefet, N., Alon, U., and Yahav, E. Adversarial examples for models of code. *arXiv preprint arXiv:1910.07517*, 2019.

Zhang, R., Isola, P., and Efros, A. A. Colorful image colorization. In *European Conference on Computer Vision*, pp. 649–666. Springer, 2016.

Appendices

A. Program transformation details

We use the Babel compiler infrastructure (McKenzie et al., 2020) and the `terser` JavaScript library for AST-based program transformations. We perform variable renaming and dead code insertion (variable declaration insertion) using custom Babel transforms, subword regularization with `sentencepiece` Python tokenization library, line subsampling using JavaScript string manipulation primatives and other transformations with `terser`. Terser has two high-level transformation modes, mangling and compression, each with finer grained controls such as formatting, comment and log removal, and dead code elimination. We show an example merge sort with variants in Figure 7.

Reformatting, beautification, compression (R, B, C): Personal coding conventions do not affect the semantics of code; auto-formatting normalizes according to a style convention.

Dead-code elimination (DCE): In this pass, all unused code with no side effects are removed. Various statements can be inlined or removed as stale or unneeded functionality.

Type upconversion (T): In JavaScript, some types are polymorphic & can be converted between each other. As an example, booleans can be represented as `true` or as `1`.

Constant folding (CF): During constant folding, all expressions that can be pre-computed at compilation time can be inlined. For example, the expression `(2 + 3) * 4` is replaced with `20`.

Variable renaming, identifier mangling (VR, IM): Arguments can be renamed with random word sequences and identifiers can be replaced with short tokens to make the model robust to naming choices. Program behavior is preserved despite obfuscation.

Dead-code insertion (DCI): Commonly used no-ops such as comments and logging are inserted.

Subword regularization (SW): From Kudo (2018), text is tokenized in several different ways, with a single word (`_function`) or subtokens (`_func tion`).

Line subsampling (LS): We randomly sample ($p = 0.9$) lines from a method body. While not semantics-preserving, line subsampling serves as a regularizer.

```
// Split the array into halves and merge
    them recursively
function mergeSort (arr) {
  if (arr.length === 1) {
    // return once we hit an array with a
        single item
    return arr
  }
  const middle = Math.floor(arr.length /
      2)
  // get the middle item of the array
      rounded down
  const left = arr.slice(0, middle)
  // items on the left side
  const right = arr.slice(middle)
  // items on the right side
  return merge(
    mergeSort(left),
    mergeSort(right)
  )
}
```

Original merge sort program

```
function mergeSort(e) {
  if (e.length === 1) {
    return e;
  }
  const t = Math.floor(e.length / 2);
  const l = e.slice(0, t);
  const n = e.slice(t);
  return merge(mergeSort(l),
      mergeSort(n));
}
```

Variable renaming, comment removal, reformatting

```
function mergeSort(e) {
  if (1 === e.length) return e;
  const t = Math.floor(e.length / 2), r
      = e.slice(0, t), n = e.slice(t);
  return merge(mergeSort(r),
      mergeSort(n));
}
```

Combining variable declarations, inlining conditional

Figure 7. Given a JavaScript code snippet implementing the merge sort algorithm, we apply semantics-preserving transformations to produce functionally-equivalent yet textually distinct code sequences. Variable renaming and identifier mangling passes change variable names. Compression passes eliminate unnecessary characters such as redundant variable declarations and brackets.

B. How similar are transformed programs?

To understand the diversity created by program transformations, we compute the Levenshtein minimum edit distance between positive pairs in the precomputed pre-training dataset, *i.e.* transformed variants of the same source method. For comparison, we also compute the edit distance between negative pairs: transformed variants of different programs.

The edit distance $D(x_q, x_k)$ computes the minimum number of token insertions, deletions or substitutions needed to transform the tokenized query progrm x_q into the key program x_k. To normalize by sequence length $|\cdot|$, let

$$\text{dissimilarity}_D(x_q, x_k) = \frac{D(x_q, x_k)}{\max(|x_q|, |x_k|)} \tag{2}$$

Dissimilarity ranges from 0% for programs with the same sequence of tokens, to 100% for programs without any shared tokens. Note that whitespace transformations do not affect the metric because the tokenizer collapses repeated whitespace. For the positives, we estimate dissimilarity by sampling one pair per source program in the CodeSearchNet dataset (1.6M source programs with at least one pair). We sample the same number of negative pairs.

Figure 8 shows a histogram of token dissimilarity. Positive pairs have 65% mean dissimilarity, while negatives have 86% mean dissimilarity. Negatives are more dissimilar on average as source sequences could have different lengths, idioms and functionality. Still, the transformations generated quite different positive sequences, with less than half of their tokens shared. The 25th, median and 75th percentile dissimilarity is 59%, 66% and 73% for positives, and 82%, 87% and 90% for negatives.

C. Experimental setup

Architectures The Transformer encoder has 6 layers (23M parameters) in all experiments. For code summarization experiments, we add 4 decoder layers with causal masking to generate the natural language summary. We leverage the default positional embedding function (sin, cos) as used in the original Transformer architecture. The network originally proposed in DeepTyper (Hellendoorn et al., 2018) had 11M parameters with a 300 dimensional hidden state. We increase the hidden state size to 512 to increase model capacity, so our BiLSTM for type prediction has 17.5M parameters. During fine-tuning, across all experiments, we optimize parameters using Adam with linear learning rate warmup and decay. For the Transformer, the learning rate is linearly increased for 5,000 steps from 0 to a maximum of 10^{-4}. For the bidirectional LSTM, the learning rate is increased for between 2,500 and 10,000 steps to a maximum of 10^{-3}. Type inference hyperparameters are selected by validation top-1 accuracy.

ContraCode pre-training The InfoNCE objective is minimized with temperature $t = 0.07$ following He et al. (2019). Also following He et al. (2019), the key encoder's parameters are computed with the momentum update equation $\theta_k \leftarrow m\theta_k + (1-m)\theta_q$, equivalent to an EMA of the query

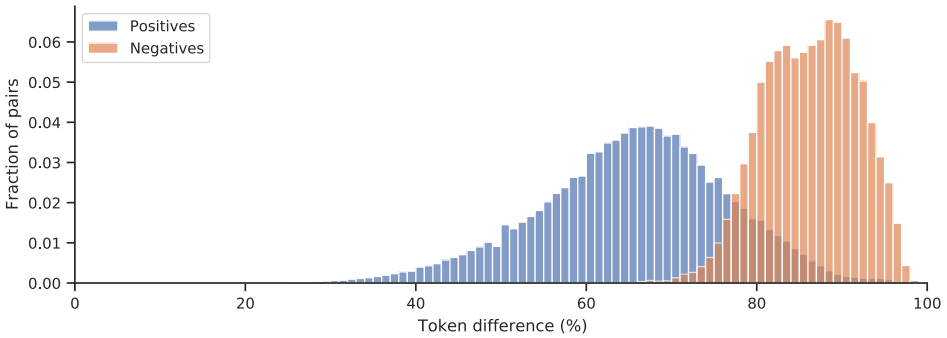


Figure 8. Histogram of pairwise token dissimilarity for contrastive positives (transformed variants of the same method) and negatives (transformed variants of different methods). Code transformations produce positives with dissimilar token sequences.

```
function processData(input) {
    var parse_fun = function (s) { return parseInt(s, 10); };

    var lines = input.split('\n');
    var A = parse_fun(lines[0]);
    var B = parse_fun(lines[1])

    console.log(A + B);
}


process.stdin.resume();
process.stdin.setEncoding("ascii");
var _input = "";
process.stdin.on("data", function (input) { _input += input; });
process.stdin.on("end", function () { processData(_input); });
```

```
(function() {
  var input;

  process.stdin.setEncoding('ascii');

  input = "";

  var sum = function(a,b){return a+b}

  process.stdin.on('data', function(data) {
    if (data === "\n")
        process.stdin.emit("end");
    input += data;
  });

  process.stdin.on('end', function() {
    var sum = input.split("\n").reduce(function(a,b){return (+a)+(+b)});
    process.stdout.write(sum);
    process.exit(0);
  });
}).call(global);
```

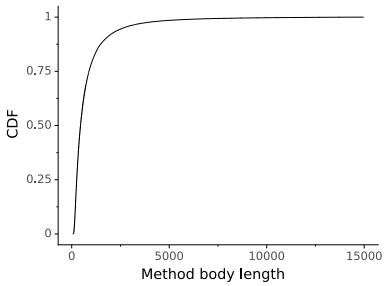
Figure 9. Code clone detection example. These programs solve the same HackerRank coding challenge (reading and summing two integers), but use different coding conventions. The neural code clone detector should classify this pair as a positive, *i.e.* a clone.

encoder parameters θ_q. To pretrain a Transformer using the ContraCode objective, we first embed each token in the program using the Transformer. However, the InfoNCE objective is defined in terms of a single embedding for the full program. The ContraCode Transformer is pre-trained with a batch size of 96. Our model averages the 512-dimensional token embeddings across the sequence, then applies a two-layer MLP with 512 hidden units and a ReLU activation to extract a 128-dimensional program embedding for the loss.

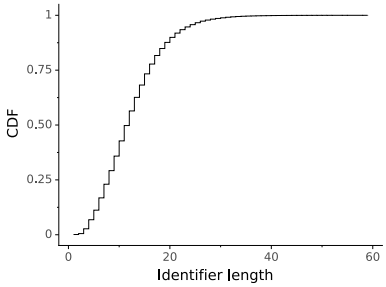
The DeepTyper bidirectional LSTM architecture offers two choices for extracting a global program representation. We aggregate a 1024-dimensional global representation of the program by concatenating its four terminal hidden states (from two sequence processing directions and two stacked LSTM layers), then apply the same MLP architecture as before to extract a 128-dimensional program representation. Alternatively, we can average the hidden state concatenated from each direction across the tokens in the sequence before applying the MLP head. We refer to the hidden-state configuration as a global representation and the sequence averaging configuration as a local representation in Table 8. We pre-train the BiLSTM with large batch size of 512 and apply weight decay.

Code clone detection on HackerRank programs Figure 9 shows two programs sampled from the HackerRank clone detection dataset. These programs successfully solve the same problem, so they are clones. We report metrics that treat code clone detection as a binary classification task given a pair of programs. 2065 pairs of programs solving the same HackerRank problem and 2065 pairs of programs solving different problems are sampled to construct an evaluation dataset. We use the area under the Receiver Operating Characteristic (AUROC) metric and Average Precision (AP) metrics. The standard error of the AUROC is reported according to the Wilcoxon statistic (Fogarty et al., 2005). Average Precision is the area under the Precision-Recall curve. AUROC and AP are both computed using the `scikit-learn` library (Pedregosa et al., 2011).

A Transformer predicts contextual embeddings of each token in a program, but our thresholded cosine similiarity classifier requires fixed length embeddings of whole programs. To determine if two programs that may differ in length are



(a) Character length per code sample



(b) Character length per method name

Figure 10. CodeSearchNet code summarization dataset statistics: (a) The majority of code sequences are under 2000 characters, but there is long tail of programs that span up to 15000 characters long, (b) JavaScript method names are relatively short compared to languages like C♯ and Java.

clones, we pool the token representations across the sequence. We evaluated both mean pooling and max pooling the representation. For the hybrid model pre-trained with both RoBERTa (MLM) and contrastive objectives, mean pooling achieved the best AUROC and AP. For other models, max pooling performed the best.

Type prediction Following DeepTyper (Hellendoorn et al., 2018), our regenerated dataset for type prediction has 187 training projects with 15,570 TypeScript files, totaling 6,902,642 tokens. We tune hyperparameters on a validation set of 23 distinct projects with 1,803 files and 490,335 tokens, and evaluate on a held-out test set of 24 projects with 2,206 files and 958,821. The training set is smaller than originally used in DeepTyper as several projects were made private or deleted from GitHub before May 2020 when we downloaded the data, but we used the same commit hashes for available projects so our splits are a subset of the original. We have released the data with our open-source code to facilitate further work on a stable benchmark as more repositories are deleted over time. We perform early stopping to select the number of training epochs. We train each model for 100 epochs and select the checkpoint with the minimum accuracy@1 metric (all types, including `any`) on the validation set. Except for the model learned from scratch, the Transformer architectures are pre-trained for 240K steps. Models with the DeepTyper architecture converge faster on the pre-training tasks and are pre-trained for 20K iterations (unless otherwise noted).

Extreme code summarization by method name prediction We train method prediction models using the labeled subset of CodeSearchNet. Neither method names nor docstrings are provided as input to the model: the docstring is deleted, and the method name is replaced with the token '`x`'. Thus, the task is to predict the method name using the method body and comments alone.

To decode method names from all models except the code2vec and code2seq baselines which implement their own decoding procedures, we use a beam search with a beam of size 5 and a maximum target sequence length of 20 subword tokens. We detail the cumulative distribution of program lengths in Figure 10. The ContraCode summarization Transformer only needed to be pre-trained for 20K iterations, with substantially faster convergence than RoBERTa (240K iterations). During fine-tuning, we apply the LS,SW,VR,DCI augmentations to ContraCode.

D. Baselines

Baselines for code summarization and type prediction trained their models on an inconsistent set of programming languages and datasets. In order to normalize the effect of datasets, we selected several diverse state-of-the-art baselines and reimplemented them on the JavaScript dataset.

AST-based models The authors of code2vec (Alon et al., 2019b) and code2seq (Alon et al., 2019a), AST-based code understanding models, made both data and code available, but train their model on the Java programming language. In order to extend the results in their paper to JavaScript for comparison with our approach, we generated an AST path dataset for the CodeSearchNet dataset. The sensitivity of path-mining embeddings to different datasets is documented in prior work, so published F1 scores are not directly comparable; F1 scores for code2vec (Alon et al., 2019b) vary between 19 (Alon et al., 2019a) and 43 (Alon et al., 2019b) depending on the dataset used. Therefore, we use the same dataset generation code as the authors for fair comparison. We first parse the source functions using the Babel compiler infrastructure. Using the original code on these ASTs, up to 300 token-to-token (leaf-to-leaf) paths are extracted from each function's AST as a precomputed dataset. Then, we generate a token and AST node vocabulary using the same author-provided code, and train the models for 20 epochs, using early stopping for code2seq. We observed that code2vec overfits after 20 epochs, and longer training was not beneficial.

DeepTyper (Hellendoorn et al., 2018) DeepTyper uses a two layer GRU with a projection over possible classes,

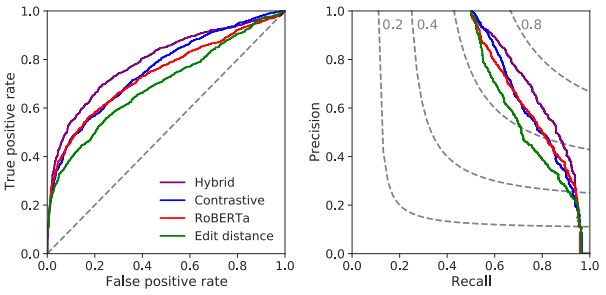


Figure 11. Receiver Operating Characteristic (ROC) and Precision-Recall (PR) curves for non-adversarial classifiers on the code clone detection task. Equal F1 score curves are shown on right.

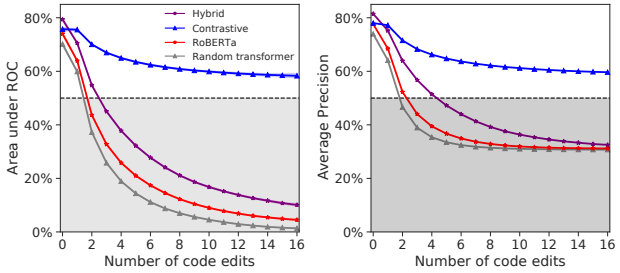


Figure 12. Adversarial AUROC and Average Precision for four models on the code clone detection task: a randomly initialized transformer, and transformers pre-trained on code with the RoBERTa MLM objective, our contrastive objective, or both. Representations learned by the contrastive model transfer robustly.

with an embedding size of 300 and hidden dimension of 650. However, we found improved performance by replacing the GRU with a bidirectional LSTM (BiLSTM). We normalize the LSTM parameter count to match our model, and therefore use a hidden dimension size of 512. We also use subword tokenization rather than space delimited tokens according to Kudo (2018), as subwords are a key part of state-of-the-art models for NLP (Sennrich et al., 2015).

RoBERTa We pre-trained an encoder using RoBERTa's masked language modeling loss on our augmented version of CodeSearchNet, the same data used to pre-train ContraCode. This model is then fine-tuned on downstream datasets. Unlike the original BERT paper which cuBERT (Kanade et al., 2020) is based on, hyperparameters from RoBERTa have been found to produce better results during pre-training. RoBERTa pre-trains using a masked language modeling (MLM) objective, where 15% of tokens in a sentence are masked or replaced and are reconstructed by the model. We did not use the BERT Next Sentence Prediction (NSP) loss which RoBERTa finds to be unnecessary. We normalize baseline parameter count by reducing the number of Transformer layers from 24 to 6 for a total of 23M parameters.

E. Additional results and ablations

Code clone detection ROC and PR curves Figure 11 plots true postive rate vs false positive rate and precision vs recall for different zero-shot classifiers on the code clone detection downstream tasks. These classifiers threshold a similarity score given by token-level edit distance for the heuristic approach or cosine similarity for the neural network representations. The hybrid self-supervised model combining ContraCode's contrastive objective and masked language modeling achieves better tradeoffs than the other approaches. Figure 12 shows the AUROC and Average Precision of four Transformer models on the same task under adversarial transformations of one input program. Untrained models as well as models pre-trained with RoBERTa's MLM objective are not robust to these code transformations. However, the model pre-trained with ContraCode preserves much of its performance as the adversarial attack is strengthened.

Which part of the model should be transferred? SimCLR (Chen et al., 2020a) proposed using a small MLP head to reduce the dimensionality of the representation used in the InfoNCE loss during pre-training, and did not transfer the MLP to the downstream image-classification task. In contrast, we find it beneficial to transfer part of the contrastive MLP head to type inference, showing a 2% improvement in top-5 accuracy over transferring the encoder only (Table 7). We believe the improvement stems from fine-tuning both the encoder and MLP which allows feature adaptation, while SimCLR trained a linear model on top of frozen features. We only transferred the MLP when contrasting the mean of token embeddings during pre-training, not the terminal hidden states, as the dimensionality of the MLP head differs. These representations are compared next.

Should we pre-train global or local representations? We compare pre-training DeepTyper with two variants of ContraCode. We either use the mean of token hidden states across the program (averaging local features), or the terminal hidden states as input to the MLP used to extract the contrastive representation $q = f_q(x)$ (global features). Token-level features might capture more syntactic details, but averaging pooling ignores order. Table 8 shows the accuracy of a BiLSTM pre-trained with each strategy.

Table 7. If local representations are learned, transferring part of the Contrastive MLP head improves type inference. The encoder is a 2-layer BiLSTM (d=512), with a 2-layer MLP head for both pre-training purposes and type inference. The mean hidden state representation is optimized for 10K iterations for the purposes of this ablation.

Warm-started layers	Acc@1	Acc@5
BiLSTM	**49.32%**	80.03%
BiLSTM, 1 layer of MLP	49.15%	**82.58%**

Table 8. Contrasting global, sequence-level representations outperforms contrasting local representations. We compare using the terminal (global) hidden states of the DeepTyper BiLSTM and the mean pooled token-level (local) hidden states.

Representation	Optimization	Acc@1	Acc@5
Global	InfoNCE with terminal hidden state, 20K steps	**52.65%**	**84.60%**
	InfoNCE with terminal hidden state, 10K steps	51.70%	83.03%
Local	InfoNCE with mean token rep., 10K steps	49.32%	80.03%

Table 9. Training time and decoder depth ablation on the method name prediction task. Longer pre-training significantly improves downstream performance when a shallow, 1 layer decoder is used.

Decoder	Pre-training (1.8M programs)	Supervision (81k programs)	Precision	Recall	F1
Transformer, 1 layer	MoCo, 10k steps	Original set	11.91%	5.96%	7.49%
Transformer, 1 layer	MoCo, 45k steps	Original set	**17.71%**	**12.57%**	**13.79%**
Transformer, 4 layers	MoCo, 45k steps	Original set	**18.21%**	**13.21%**	**14.56%**

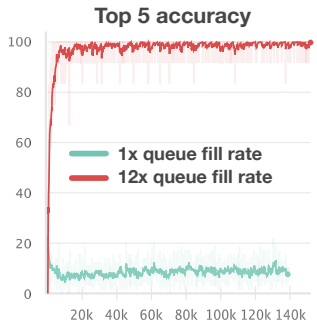


Figure 13. Pre-training quickly converges if negative programs in the queue are frequently changed.

Using the global features for pre-training yields significantly improved performance, +2.38% acc@1 after 10K iterations of pre-training (not converged for the purposes of ablation). The global pre-training strategy achieves our best results.

Do pre-trained encoders help more with shallow decoders? For the sequence-to-sequence code summarization task, ContraCode only pre-trains the encoder of the Transformer. In Table 9, we ablate the depth of the decoder to understand how much shallow decoders benefit from contrastive pre-training of the encoder. Similar experiments were performed in a vision context by (Erhan et al., 2010), where different numbers of layers of a classifier are pre-trained. After 45k pre-training steps, the 4-layer decoder achieves 0.50% higher precision, 0.64% higher recall and 0.77% higher F1 score than the 1-layer model, so additional decoder depth is helpful for the downstream task. The 1-layer decoder model also benefits significantly from longer pre-training, with a 6.3% increase in F1 from 10k to 45k iterations. This large of an improvement indicates that ContraCode could be more helpful for pre-training when the number of randomly initialized parameters at the start of fine-tuning is small. For larger decoders, more parameters must be optimized during-finetuning, and the value of pre-training is diminished.

Contrastive representation learning strategies In Figure 13, we compare two strategies of refreshing the MoCo queue of key embeddings (the dictionary of negative program representations assumed to be non-equivalent to the batch of positives). In the first strategy, we add 8 items out of the batch to the queue ($1\times$), while in the second we add 96 items ($12\times$). In addition, we use a larger queue (65k versus 125k keys) and a slightly larger batch size (64 versus 96). We observe that for the baseline queue fill rate, the accuracy decreases for the first 8125 iterations as the queue fills. This decrease in accuracy is expected as the task becomes more difficult due to the increasing number of negatives during queue warmup. However, it is surprising that accuracy grows so slowly once the queue is filled. We suspect this is because the key encoder changes significantly over thousands of iterations: with a momentum term $m = 0.999$, the original key encoder parameters are decayed by a factor of 2.9×10^{-4} by the moving average. If the queue is rapidly refreshed, queue embeddings are predicted by recent key encoders, not old parameters. This also indicates that a large diversity of negative, non-equivalent programs are helpful for rapid convergence of ContraCode pre-training.

F. Qualitative results

t-SNE visualization of representations We qualitatively inspect the structure of the learned representation space by visualizing self-supervised representations of variants of 28 programs using t-SNE (Maaten & Hinton, 2008) in Figure 15. Representations of transformed variants of the same program are plotted with the same color. ContraCode (BiLSTM) clusters variants closely together. Indeed, contrastive learning learns representations that are invariant to a wide

```
import {
  write,
  categories,
  messageType
} from "s";
export const animationsTraceCategory = "s";
export const rendererTraceCategory = "s";
export const viewUtilCategory = "s";
export const routerTraceCategory = "s";
export const routeReuseStrategyTraceCategory = "s";
export const listViewTraceCategory = "s";
export function animationsLog ( message: string 100.0% ): void 99.9% {
  write(message, animationsTraceCategory);
}
export function rendererLog (msg): void 53.7% {
  write(msg, rendererTraceCategory);
}
export function rendererError ( message: string 99.5% ): void 99.7% {
  write(message, rendererTraceCategory, messageType.error);
}
export function viewUtilLog (msg): void 100.0% {
  write(msg, viewUtilCategory);
}
export function routerLog ( message: string 99.9% ): void 100.0% {
  write(message, routerTraceCategory);
}
export function routeReuseStrategyLog ( message: string 99.8% ): void 99.98% {
  write(message, routeReuseStrategyTraceCategory);
}
export function styleError ( message: string 99.97% ): void 100.0% {
  write(message, categories.Style, messageType.error);
}
export function listViewLog ( message: string 100.0% ): void 100.0% {
  write(message, listViewTraceCategory);
}
export function listViewError ( message: string 99.93% ): void 100.0% ...
```

```
import {
  ComponentRef,
  ComponentFactory,
  ViewContainerRef,
  Component,
  Type,
  ComponentFactoryResolver,
  ChangeDetectorRef
} from "s";
import {
  write
} from "s";
export const CATEGORY = "s";

function log( message: string 56.95 ) {
  write(message, CATEGORY);
}
@ Component({
  selector: "s",
  template: `template`
}) export class DetachedLoader {
  constructor(private resolver: ViewContainerRef 63.85% (GT: ComponentFactoryResolver) ,
              private changeDetector: ChangeDetectorRef 100.0% ,
              private containerRef: ViewContainerRef 100.0% ) {}
  private loadInLocation (
        componentType<any>: TemplateRef 99.6% (GT: Type)) <ComponentRef<any>>: Promise 100.0% {
    const factory = this.resolver.resolveComponentFactory(componentType);
    const componentRef = this.containerRef.createComponent(
      factory, this.containerRef.length, this.containerRef.parentInjector);
    log("s");
    return Promise.resolve(componentRef);
  }
  public detectChanges() {
    this.changeDetector.markForCheck();
  }
  public loadComponent (
        componentType<any>: TemplateRef 99.9% (GT: Type)) <ComponentRef<any>>: Promise 100.0% {
    log("s");
    return this.loadInLocation(componentType);
  } ...
```

Figure 14. Our model, a variant of DeepTyper pretrained with ContraCode, generates type annotations for two programs in the held-out set. The model consistently predicts the correct return type of functions, and even predicts project-specific types imported at the top of the file. The model corresponds to the top row of Table 8, though is not our best performing model.

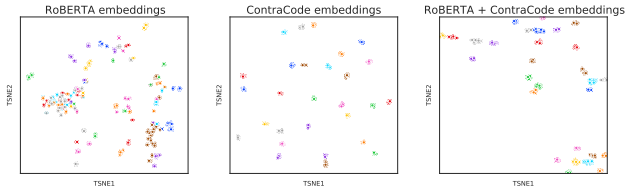


Figure 15. t-SNE (Maaten & Hinton, 2008) plot of mean pooled program representations learned with masked language modeling (RoBERTa), contrastive learning (ContraCode), and a hybrid loss (RoBERTa + ContraCode). Transformed variants of the same program share the same color. Note that colors may be similar across different programs.

```
function x(url, callback,
      error) {
  var img = new Image();
  img.src = url;
  if(img.complete){
    return callback(img);
  }
  img.onload = function(){
    img.onload = null;
    callback(img);
  };
  img.onerror = function(e){
    img.onerror = null;
    error(e);
  };
}
```

Ground truth: `loadImage`
Prediction: `loadImage`

Other predictions:

1. `getImageItem`
2. `createImage`
3. `loadImageForBreakpoint`
4. `getImageSrcCSS`

Figure 16. A JavaScript program from the CodeSearchNet dataset not seen during training and the predicted method names from a Transformer pre-trained with ContraCode. ContraCode predicts the correct method name as its most likely decoding.

class of automated compiler-based transformations. In comparison, the representations learned by masked language modeling (RoBERTa) show more overlap between different programs, and variants do not cleanly cluster. With a hybrid loss combining masked language modeling and contrastive learning, representations of variants of the same program once again cluster.

Code summaries Figure 16 shows a qualitative example of predictions for the code summarization task. The JavaScript method is not seen during training. A Transformer pre-trained with ContraCode predicts the correct method name through beam search. The next four predictions are reasonable, capturing that the method processes an image. The 2nd and 3rd most likely decodings, `getImageItem` and `createImage`, use `get` and `create` as synonyms for `load`, though the final two unlikely decodings include terms not in the method body.

Type inferences We can also visualize outputs of the type inference model. Figure 14 shows two TypeScript programs from the held-out test set. User-provided type annotations are removed from the programs, and the model is provided with a tokenized form without access to dependencies. We visualize predictions from a variant of DeepTyper pre-trained with ContraCode. This corresponds to the best-performing model in Table 8.

In the first program, our model consistently predicts the correct return and parameter type. While a tool based on static analysis could infer the `void` return types, the type of the `message` argument is ambiguous without access to the imported `write` method signature. Still, the model correctly predicts with high confidence that the variable `message` is a string. In the second program, ContraCode correctly predicts 4 of 8 types including the `ViewContainerRef` and `ChangeDetectorRef` types, each imported from the AngularJS library. As this sample is held-out from the training set, these predictions show generalization from other repositories using AngularJS.