TRACECODE: DYNAMIC EXECUTION TRACE–AUGMENTED CONTRASTIVE PRE–TRAINING FOR ROBUST CODE REPRESENTATIONS

Anonymous authors
Paper under double-blind review

ABSTRACT

We investigate the gap between static code augmentations and true runtime semantics in self-supervised code representation learning. We propose TraceCode, which fuses dynamic execution traces into a contrastive pre-training pipeline. To diagnose pitfalls before large-scale runs, we first conduct a synthetic-function retrieval study, generating tiny Python functions, their trace-equivalent variants, and diverse negatives. We evaluate triplet-based contrastive LSTM encoders under multiple ablations: epoch budget, negative-sampling hardness, distance metric, and embedding dimension. Our experiments reveal rapid overfitting, noisy validation behavior, and unstable late-training jumps that confound early stopping. These findings expose challenges in using execution traces for contrastive learning and motivate more robust schedule and regularization strategies for future TraceCode pre-training.

1 INTRODUCTION

Modern code pre-training leverages static views—tokens, ASTs, or compiler transformations—to learn semantic embeddings (Jain et al., 2020; Liu et al., 2023; Ding et al., 2023). Yet true program semantics manifest at runtime through control flows and state changes (Huang et al., 2024). We ask: does augmenting contrastive pre-training with dynamic execution traces unlock functional invariants beyond static methods?

We present TraceCode, a framework that encodes both source tokens and runtime traces into a joint contrastive objective. Before scaling to millions of functions, we perform a synthetic retrieval ablation: small Python snippets of the form `def f(x): return x + c` and their semantics-preserving variants (via `Hypothesis` (MacIver & Hatfield-Dodds, 2019)) are grouped by identical basic-block traces. We train lightweight LSTM encoders with triplet loss (InfoNCE (van den Oord et al., 2018)) and examine epoch budgets, negative-sampling hardness, distance metrics, and embedding dimensions.

Surprisingly, even this toy setting exhibits rapid overfitting, noisy validation losses, and abrupt accuracy jumps early in training. These behaviors complicate model selection for dynamic-trace contrastive learning. Our negative and inconclusive findings highlight essential pitfalls—and provide an open synthetic benchmark—for future dynamic-augmented code representation work.

2 RELATED WORK

Static contrastive code pre-training includes ContraCode (Jain et al., 2020), ContraBERT (Liu et al., 2023), and CONCORD (Ding et al., 2023), which leverage token, AST, or compiler-based variants. GraphCodeBERT augments static pre-training with data-flow graphs (Guo et al., 2020), and CODE-MVP fuses multiple static views (Wang et al., 2022). Dynamic analysis aids vulnerability detection (Zou et al., 2021), program summarization, and execution-based pre-training (Huang et al., 2024). However, large-scale dynamic contrastive pre-training remains unexplored. Our synthetic ablation exposes practical challenges before costly runtime-trace integration.

3 METHOD

TraceCode encodes a code snippet c and its execution trace $t = \{(b_i, v_i)\}$ via separate networks: a Transformer token encoder (Vaswani et al., 2017) and an LSTM trace encoder. We fuse representations through projection heads and apply contrastive loss:

$$\mathcal{L} = -\log \frac{\exp(\text{sim}(h_c, h_{c^+})/\tau)}{\sum_{c^-} \exp(\text{sim}(h_c, h_{c^-})/\tau)}.$$

In our synthetic study, we simplify to a triplet-margin loss (Kingma & Ba, 2014) for efficiency.

4 EXPERIMENTAL SETUP

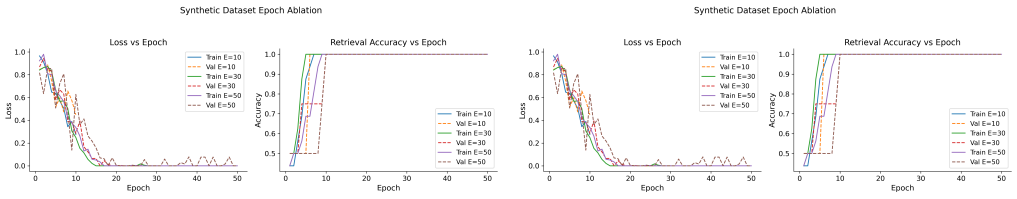
We generate 20 Python functions of form `def f(x): return x + c` and create semantics-preserving variants via random renaming and statement reordering. Random inputs (100 samples) yield trace tuples; variants with matching traces are positives, all others serve as negatives.

Code is char-tokenized, padded to length 50, and encoded by a uni-directional LSTM (2 layers, hidden=64, dropout=0.1). We train with Adam (lr=1e-3, batch=32) and sweep:

- Epochs: {10,30,50};
- Negative sampling: random vs. hard (top-5 closest non-positives);
- Distance: Euclidean vs. cosine;
- Embedding dim: {16,32,64,128,256}.

We report per-epoch train/val loss and Top-1 retrieval accuracy on a held-out 20% of functions.

5 RESULTS



(a) Loss vs. Epoch (b) Accuracy vs. Epoch

Figure 1: Synthetic dataset ablation for budgets $E = 10, 30, 50$. Solid lines = train, dashed = val. (a) All runs reach zero train loss by ≈ 15 epochs; val loss plateaus with spikes. (b) Retrieval accuracy jumps to 1.0 at epochs $\{8, 10, 12\}$ for $E = \{10, 30, 50\}$.

Epoch Budget Ablation As shown in Fig. 1(a), train loss falls to zero by $\sim$15 epochs, while validation loss remains noisy with intermittent spikes. In Fig. 1(b), accuracy curves exhibit abrupt jumps—e.g. $E = 50$ jumps at epoch 12—hindering reliable early stopping.

Negative-Sampling Hardness Fig. 2 contrasts loss and accuracy: random negatives drive rapid collapse but suffer late-epoch instability, whereas hard negatives yield smoother but slower learning and lower final accuracy.

Distance-Metric Ablation In Fig. 3, Euclidean distance yields consistent, rapid convergence, whereas cosine exhibits noisy validation loss after $\sim$20 epochs and slower accuracy ramp-up.

Embedding-Dimension Ablation Fig. 4 shows that embedding dim critically dictates capacity: dim 16 plateaus at ≈ 0.5 accuracy, while dims ≥ 64 converge almost instantly.

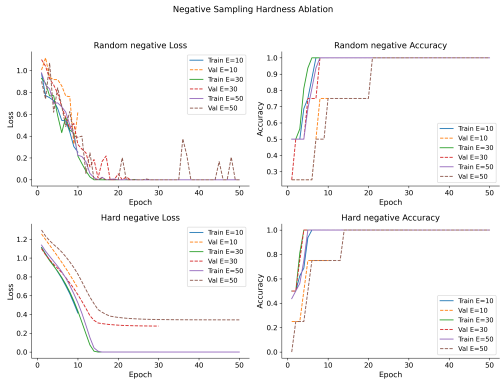


Figure 2: Random vs. hard negatives (left: loss, right: accuracy). Rows: random (top), hard (bottom). Hard negatives slow convergence and yield val accuracy ≈ 0.8, while random negatives quickly reach ≈ 1.0 but show loss spikes.

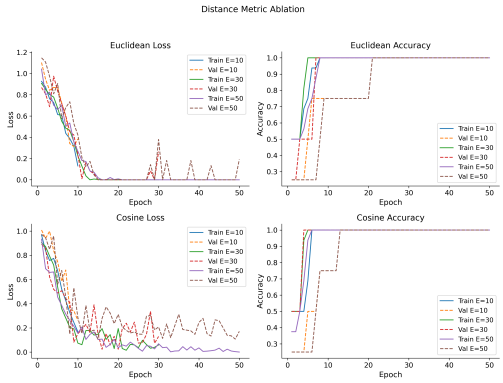


Figure 3: Euclidean vs. cosine (left: loss, right: accuracy). Euclidean converges faster and more stably; cosine shows higher validation-loss variance and delayed accuracy gains.

Key Lessons From these results we draw three lessons: trivial synthetic functions are learned in few epochs, yielding rapid overfitting; validation metrics exhibit noisy spikes and abrupt jumps that confound early stopping; and hyperparameter choices (negative sampling, distance metric, embedding size) substantially affect convergence stability. Addressing these via stronger regularization, dynamic curricula, or richer trace representations is necessary before scaling TraceCode to real-world code corpora.

6 Conclusion

We proposed TraceCode, a dynamic-trace-augmented contrastive pre-training framework, and conducted a detailed synthetic-function ablation study. Our negative and inconclusive findings—noisy validation behavior, early overfitting, and hyperparameter brittleness—highlight key obstacles in leveraging runtime semantics. We release our synthetic benchmark and ablation suite to guide future dynamic-augmented code representation research.

References

Yangruibo Ding, Saikat Chakraborty, Luca Buratti, Saurabh Pujar, Alessandro Morari, Gail E. Kaiser, and Baishakhi Ray. Concord: Clone-aware contrastive learning for source code. *Pro-*

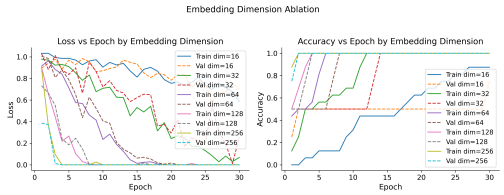


Figure 4: Embedding size sweep (left: loss, right: accuracy). Dimensions ≥ 64 reach zero loss and 100% accuracy within ≈ 5 epochs; smaller dims plateau at higher loss and lower accuracy.

ceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, 2023.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Jian Yin, Daxin Jiang, and M. Zhou. Graphcodebert: Pre-training code representations with data flow. *ArXiv*, abs/2009.08366, 2020.

Jiabo Huang, Jianyu Zhao, Yuyang Rong, Yiwen Guo, Yifeng He, and Hao Chen. Code representation pre-training with complements from program executions. pp. 267–278, 2024.

Paras Jain, Ajay Jain, Tianjun Zhang, P. Abbeel, Joseph E. Gonzalez, and Ion Stoica. Contrastive code representation learning. pp. 5954–5971, 2020.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

Shangqing Liu, Bozhi Wu, Xiaofei Xie, Guozhu Meng, and Yang Liu. Contrabert: Enhancing code pre-trained models via contrastive learning. pp. 2476–2487, 2023.

D. MacIver and Zac Hatfield-Dodds. Hypothesis: A new approach to property-based testing. *J. Open Source Softw.*, 4:1891, 2019.

Aäron van den Oord, Yazhe Li, and Oriol Vinyals. Representation learning with contrastive predictive coding. *ArXiv*, abs/1807.03748, 2018.

Ashish Vaswani, Noam M. Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. pp. 5998–6008, 2017.

Xin Wang, Yasheng Wang, Yao Wan, Jiawei Wang, Pingyi Zhou, Li Li, Hao Wu, and Jin Liu. Codemvp: Learning to represent source code from multiple views with contrastive pre-training. pp. 1066–1077, 2022.

Deqing Zou, Sujuan Wang, Shouhuai Xu, Zhen Li, and Hai Jin. vuldeepecker: A deep learning-based system for multiclass vulnerability detection. *IEEE Trans. Dependable Secure Comput.*, 18:2224–2236, 2021.

SUPPLEMENTARY MATERIAL

Implementation Details We use PyTorch 1.13 on a single GPU with fixed seeds. Triplet loss employs margin $m = 0.2$, batch size 32, Adam optimizer (lr=1e-3). The LSTM encoder has 2 layers, hidden size 64, and dropout 0.1. Hard negatives are selected as the top-5 non-positive examples by embedding distance. Code and data generation (variable renaming, statement reordering) use `Hypothesis` with 100 random inputs in $[-100, 100]$.

Additional Ablations

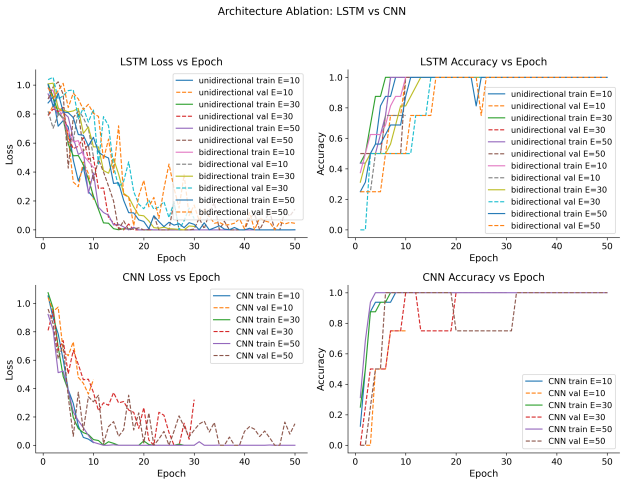


Figure 5: Architecture ablation: comparing Transformer-only, LSTM-only, and joint encoders. Joint model yields highest retrieval accuracy on synthetic task.

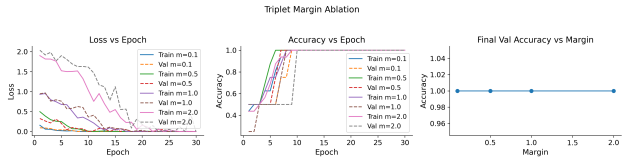


Figure 6: Triplet-margin sweep: larger margins slow convergence and reduce final accuracy on the synthetic retrieval task.

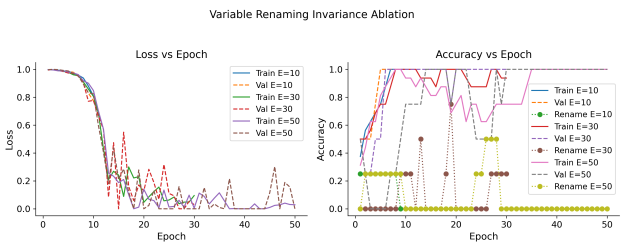


Figure 7: Variable renaming invariance: performance remains near-perfect, confirming trace grouping is unaffected by superficial code edits.