Renamer: A Transformer Architecture Invariant to Variable Renaming

Zachary Ankner*[1], Alex Renda[1], and Michael Carbin[1]

[1]MIT CSAIL

Abstract

Many modeling tasks involve learning functions which are invariant to certain types of input transformations. We study a specific class of invariance: semantics-preserving variable renaming for models of code. We show that vanilla Transformers trained on renaming-invariant tasks do not exhibit renaming invariance. We propose Renamer, a Transformer architecture which is itself invariant to semantics-preserving variable renaming. On a CPU simulation task, Renamer reduces error by between 24.79% and 52.8% compared to a vanilla Transformer.

1 Introduction

Modeling tasks often require reasoning about *invariances* of the task, classes of input transformations for which the output of the task remains unchanged (Snavely, 2019; Bianchi et al., 2022). Ideally a model itself is invariant to the same classes of input transformations as the underlying task. Violations of this ideal hurt the model's generalization on downstream tasks (Alon et al., 2019; Gao et al., 2023).

Renaming invariance. A common invariance for models of code is *renaming invariance*, invariance to transformations that change variable names (or other identifiers) in code in a way that preserves the semantics of the overall code block. Renaming invariance arises when reasoning about formal languages including programming languages (Chen et al., 2021; Alon et al., 2019; Renda et al., 2020), mathematics (Lample & Charton, 2020; Polu et al., 2022), and synthetic grammars of natural languages (Marzoev et al., 2020; Berant & Liang, 2014).

To define renaming invariance, we first define a *view mapping* as a mapping from an input token to its *view*, where a view represents all the information about a token that is salient to the function other than its relation to other tokens. We then define a *referent relation* as a binary relation between input tokens that states whether or not the tokens refer to the same underlying entity. A renaming invariant function is a function that generates the same output for any bijection of tokens that does not change tokens' views and preserves the referent relation.

Consider an example input from a symbolic algebra task: `x + y`, which has three tokens: `x`, `+`, and `y`. In this task the information about `x` and `y` that is salient to the function is that each is a variable, while the salient information about `+` is that it is an addition operator. Thus the view mapping for this task maps `x` and `y` to the same view, and `+` to a different view. In this task, all of `x`, `+`, and `y` refer to different underlying entities (different variables and the addition operation), and thus are not related by the referent relation.

Renaming invariance in ML for systems. Renaming invariance is a common property of systems tasks. For example, compilers require accurate predictions of code execution speed, which is invariant to variable renaming (Mendis et al., 2019; Baghdadi et al., 2021). ML-based bug detection predicts whether a given code snippet has a bug, which, for closed terms (those with no free variables), is invariant to variable name choices (Liu et al., 2019; Allamanis et al., 2021; Liu et al., 2023). LLMs that interact with computer systems often do so through renaming-invariant analysis, generation, and execution of code (Gao et al., 2023).

*Correspondence to: ankner@mit.edu

Machine Learning for Systems Workshop at 37th NeurIPS Conference, 2023, New Orleans, LA, USA.

While general-purpose neural network architectures have shown impressive results on learning functions with renaming invariance (Alon et al., 2019; Renda et al., 2021), neural networks (including LLMs) are not themselves invariant to renaming (Alon et al., 2019; Gao et al., 2023).

Contributions. We present an approach to enforcing renaming invariance in Transformers:

- We introduce and formally characterize the renaming invariance problem.
- We propose the Renamer, a renaming invariant Transformer model architecture.
- We evaluate the Renamer on a renaming invariant x86 assembly processing task. Renamer reduces the error compared to a vanilla Transformer model by between 27.58% and 52.80%.

By defining renaming invariance and proposing a Transformer model invariant to renaming, our work takes a key step towards providing low-error models with provable guarantees.

2 Renaming Invariance in x86 Assembly

This section presents a case study of renaming invariance in a sequence processing task.

Task. Following Renda et al. (2021), we create a *neural network surrogate* that mimics the behavior of llvm-mca, a CPU simulator included with the LLVM infrastructure (Lattner & Adve, 2004). As input llvm-mca takes a *basic block* of x86-64 assembly, a sequence of assembly instructions with no jumps or loops. It then predicts the *throughput* of the basic block on the simulated CPU, which is the number of CPU clock cycles taken to execute the block when repeated for a fixed number of iterations. Learning a surrogate of llvm-mca results in faster or more accurate throughput prediction than using llvm-mca itself (Renda et al., 2021).

Input specification. We evaluate using a dataset of AT&T-syntax x86-64 basic blocks (Chen et al., 2019). Figure 1 presents three such basic blocks. AT&T syntax basic blocks are sequences of instructions, where each instruction consists of an opcode (e.g., `mov`), a source operand (e.g., `64(%rsp)`), and a destination operand (e.g., `%rax`). Each operand may be a constant (e.g., `$1`), a register (e.g., `%rax`), or a memory address (e.g., `64(%rsp)`). In AT&T syntax x86, the final operand of an instruction is the destination to which the instruction writes (and may also be read from).

A given register operand consists of a *bitwidth* and a *base register*. The bitwidth is how many bits of the register data are addressed by the register. As an example, `%rax` addresses all 64 bits of the register data, `%eax` addresses the lowest 32 bits, and `%ax` addresses the lowest 16 bits. The base register is the location where register data is stored; this is typically indicated by the final several characters of the register (e.g., `ax` in `%rax`). Base registers belong to different classes (general-purpose, vector, floating-point, etc).

Simulation model. In llvm-mca's simulation model, predictions depend on the opcodes of each instruction in the block, the bitwidth and register class of each operand, and the *register dependency graph* with edges between instructions with source and destination operands that share the same base register.

Renaming invariance. Renaming invariance manifests in llvm-mca as invariance to register renaming. When the base register names are renamed in a given block such that none of the register bitwidths, classes, or dependency graph change, llvm-mca generates an identical prediction for this block.

To formalize this, we say that the *view* of a given register is its bitwidth and register class. We then define the *referent relation* as a binary relation that holds between two registers if they have the same base register (e.g., as `%rax` and `%eax` do); we say that any registers related by the referent relation are *coreferential*. Any transformation of registers that maintains both register views and also the referent relation is a renaming invariant transformation.

Example. Figure 1 presents three x86 basic blocks in AT&T syntax. Figure 1(a) shows the original block. This basic block has a throughput of 1.6 cycles per iteration in llvm-mca's model. There are four unique registers in the basic block: `%rsp`, `%eax`, `%rax`, and `%rbp`. Registers `%rsp`, `%rax`, and `%rbp` all have view `(64-bit, general-purpose)`. `%eax` has the view `(32-bit, general-purpose)`. Registers `%eax` and `%rax` are coreferential as they both share the `ax` base register. Registers `%rsp` and `%rbp` are not coreferential with any other registers in Figure 1(a).

Figure 1(b) shows a semantically equivalent version of the block. In the new block, `%rax` was renamed to `%rbx` and `%eax` was renamed to `%ebx`. Since `%rbx` has view `(64-bit, general-purpose)` and `%ebx` has view `(32-bit, general-purpose)`, this transformation preserves the views of all registers. Since `%rbx` and `%ebx` are coreferential (and are not coreferential with any other registers),

```
mov  64(%rsp), %rax
sub  $1,   56(%rbp)
mov  16(%rax), %eax
```

llvm-mca: 1.68 cycles

(a) Original block.

```
mov  64(%rsp),  %rbx
sub  $1,   56(%rbp)
mov  16(%rbx), %ebx
```

llvm-mca: 1.68 cycles

(b) Invariant renaming.

```
mov  64(%rax), %rax
sub  $1,   56(%ebp)
mov  16(%rax), %eax
```

llvm-mca: 10.03 cycles

(c) Non-invariant renaming.

Figure 1: Example renamings of an x86-64 basic block. Registers may be renamed as long as each is renamed to a register with the same bitwidth and the dependency graph is preserved.

this transformation preserves the referent relation. Thus, because the views and referent relation are preserved, llvm-mca outputs the same timing for the renamed block.

Figure 1(c) shows a version of the block with registers renamed in manner that is not semantically equivalent. First, views are not preserved: `%rbp`, which is `64-bit`, is renamed to `%ebp`, which is `32-bit`. Second, the referent relation is not preserved: `%rsp` is renamed to `%rax` on the first line while `%rax` and `%eax` are not renamed. Each change alters the semantics of the original block. Thus, llvm-mca outputs a different timing for the renamed block compared to the original block.

Vanilla Transformers are sensitive to register renaming, predicting 1.5 cycles on Figure 1(a) and 2.3 on Figure 1(b). The Renamer more accurately predicts 1.6 cycles for both.

3 Formalizing Renaming Invariance

Let $x \in X$ be an input token from the vocabulary, let $[x_i] \in X^n$ be a length n sequence of tokens, and let $f : X^n \to \mathbb{R}$ be a function over a sequence of tokens.

Let V be a set of *views*. A view represents the information about a token that is salient to the function other than its relation to other tokens. The *view mapping* $v : X \to V$ associates tokens with a view.

Let $R \subseteq X \times X$ be the *referent relation*, a reflexive, symmetric, and transitive relation on X. The referent relation is a relation which holds between two tokens if they refer to the same object. We use the notation $x\ R\ y$ to denote that R holds between x and y. We also refer to such tokens as *coreferential*.

Let $\sigma : X \to X$ be a permutation (i.e., a bijection) over the input tokens. σ is *view-constrained* if $\forall x \in X.v(x) = v(\sigma(x))$, meaning that all tokens are mapped to a token with the same semantic role in the function. σ is *referent-constrained* if $\exists\ R.\forall x, y \in X.x\ R\ y \Leftrightarrow \sigma(x)\ R'\ \sigma(y)$. That is, σ is referent-constrained if it preserves the referent relation between tokens. Furthermore, σ is *semantics-preserving* if it is both view-constrained and referent-constrained. We use $\sigma^n : X^n \to X^n$ to denote an element-wise extension of σ to sequences. A function f is *renaming invariant* for a given view mapping and referent relation if for all semantics-preserving permutations σ^n, $f([x_i]) = f(\sigma^n([x_i]))$.

4 Renamer Architecture

In this section we present the Renamer, a renaming invariant Transformer architecture.

Transformers background. A Transformer takes a length-n sequence of tokens $[x_i]$ as input, *embeds* each individual token, then repeatedly performs *self-attention* on the resulting embedding sequence. The sequence content embedding $C^{[x_i]} \in \mathbb{R}^{n \times d}$ is computed token-wise as $C^{[x_i]}_j = C([x_i]_j)$, where $C : X \to \mathbb{R}^d$ is an embedding table. Self attention is computed as $\text{softmax}(M(QK^T))V$ where Q, K, and V are linear projections of the token representations, and $M : \mathbb{R}^{n \times n} \to \mathbb{R}^{n \times n}$ is an attention mask operator. The attention mask operator M, computed from $[x_i]$, is defined such that $M(A)_{j,k} = -\infty$ if $[x_i]_j$ should not attend to $[x_i]_k$ and $M(A)_{j,k} = A_{j,k}$ otherwise.

4.1 Renamer Architecture Modifications

Name-to-view anonymization. We anonymize the name of a token to the model by mapping each token in the input sequence to an embedding that represents only its view. If all input tokens which share the same view share the same representation when input to the network, then a sequence and its renamed counterpart will share the same representation. This can be seen as $\forall x_j \in [x_i].\ C(x_j) = C(\sigma(x_j)) \Rightarrow C^{[x_i]} = C^{\sigma^n([x_i])}$. Thus the embeddings of the input sequence and the renamed input sequence are identical, meaning that after name-to-view anonymization the Renamer is renaming invariant.

Referent binding. Name-to-view anonymization alone limits the class of functions the Transformer can represent: all tokens with the same view share the same embedding, so the network can't

Table 1: MAPE on original test set

Model	Tiny	Mini	Small
Vanilla	3.30%	1.13%	1.25%
Augmented	3.34%	2.25%	2.36%
Canonicalized	3.03%	**0.96%**	0.76%
Renamer	**2.39%**	**0.85%**	**0.59%**

Table 2: MAPE on test set with renamed registers

Model	Tiny	Mini	Small
Vanilla	5.26%	2.76%	2.89%
Augmented	3.44%	2.55%	2.36%
Canonicalized	3.03%	0.96%	0.76%
Renamer	**2.39%**	**0.85%**	**0.59%**

differentiate which tokens in the input are coreferential and which are not. To allow this differentiation, the first layer of the Renamer only applies self-attention between coreferential tokens. This breaks symmetry between tokens that share the same view but that are not coreferential. We only intervene on the first layer of the Transformer as once the view symmetry is broken, the Transformer's representational capacity is fully restored for future layers so no other interventions are needed.

Given an input $[x_i]$ and a referent relation R, we construct the *referent attention mask operator* as $M^r(A)_{j,k} = A_{j,k}$ if $[x_i]_j \, R \, [x_i]_k$ and $M^r(A)_{j,k} = -\infty$ otherwise, meaning we only compute attention between coreferential tokens. The representation is then $\text{softmax}(M^r(QK^T))V$.

5 Evaluation on llvm-mca

We evaluate Renamer by learning a surrogate of llvm-mca, the CPU simulator discussed in Section 2.

5.1 Task

The task under study is to take an x86-64 basic block as input, and output a prediction of the timing that llvm-mca would output for this basic block. We evaluate Renamer on the BHive dataset (Chen et al., 2019), which is a collection of x86-64 basic blocks from a variety of real-world programs. We remove the 8% of the dataset with instructions with implicit operands.

The views and referent relation for this task are as described in Section 2. The view for each register is its bitwidth and register class. All other tokens have unique singleton views. The referent relation holds between any two register tokens that share the same base register (i.e., that point to the same data). All other tokens are only coreferential with themselves.

5.2 Models

We use the Tiny, Mini, and Small BERT architectures (Devlin et al., 2019). We compare Renamer against three baselines: *Vanilla Transformer* is an unmodified BERT architecture. *Augmented Transformer* is an unmodified BERT architecture trained on a semantics-preserving register renamed version of the dataset. *Canonicalized Transformer* is an unmodified BERT architecture with a pre-processing step that canonicalizes each basic block before feeding it into the model.

5.3 Results

Standard test set. Table 1 shows the error of each model across BERT sizes on the standard test set. We find that across all model sizes, the Renamer model matches or outperforms the vanilla, augmented, and canonicalized models on the original test set.

Renamed test set. We also evaluate the models on a register-renamed version of the test set to test out-of-distribution generalization. Table 2 shows the error of each model across BERT sizes on the permuted version of the test set. The performance of the vanilla model is significantly affected by permuting the registers. In contrast to the vanilla model, Renamer is provably invariant to register perturbations.

6 Conclusion

Renaming invariance is an important property of a range of tasks, from x86 assembly throughput prediction to symbolic differentiation. We formalize the concept of renaming invariance, and present the Renamer, a renaming invariant Transformer architecture. We find that the Renamer results in commensurate or lower in-distribution error than baseline models and is more robust to out-of-distribution variable names than baseline models. Our work takes a key step towards the goal of providing low-error models with provable guarantees for tasks with input invariances.

Acknowledgements

We would like to thank Tian Jin and the anonymous reviewers for their helpful comments and suggestions. This work was supported in-part by the National Science Foundation (CCF-1918839 and CCF-2217064), the Defense Advanced Research Projects Agency (#HR00112190046), and an Intel Research Award.

References

Allamanis, M., Jackson-Flux, H. R., and Brockschmidt, M. Self-supervised bug detection and repair. In Beygelzimer, A., Dauphin, Y., Liang, P., and Vaughan, J. W. (eds.), *Advances in Neural Information Processing Systems*, 2021. URL https://openreview.net/forum?id=EQbyD_KG6w-1h.

Alon, U., Zilberstein, M., Levy, O., and Yahav, E. Code2vec: Learning distributed representations of code. In *Principles of Programming Languages*, 2019.

Baghdadi, R., Merouani, M., Leghettas, M.-H., Abdous, K., Arbaoui, T., Benatchba, K., and Amarasinghe, S. A deep learning based cost model for automatic code optimization. In *Proceedings of the Fourth Conference on Machine Learning and Systems (MLSys)*, MLSys 2021, Apr 2021. URL http://groups.csail.mit.edu/commit/papers/21/tiramisu_autoscheduler.pdf.

Berant, J. and Liang, P. Semantic parsing via paraphrasing. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2014.

Bianchi, F., Nozza, D., and Hovy, D. Language invariant properties in natural language processing. In *Proceedings of NLP Power! The First Workshop on Efficient Benchmarking in NLP*, 2022.

Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H. P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F. P., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W. H., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saunders, W., Hesse, C., Carr, A. N., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., and Zaremba, W. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Chen, Y., Brahmakshatriya, A., Mendis, C., Renda, A., Atkinson, E., Sykora, O., Amarasinghe, S., and Carbin, M. BHive: A benchmark suite and measurement framework for validating x86-64 basic block performance models. In *IEEE International Symposium on Workload Characterization*, 2019. doi: 10.1109/IISWC47752.2019.9042166.

Dernoncourt, F., Lee, J. Y., Uzuner, O., and Szolovits, P. De-identification of patient notes with recurrent neural networks. *Journal of the American Medical Informatics Association*, 24(3), 2016.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2019. doi: 10.18653/v1/N19-1423.

Fuchs, F., Worrall, D., Fischer, V., and Welling, M. Se(3)-transformers: 3d roto-translation equivariant attention networks. In *Advances in Neural Information Processing Systems*, 2020.

Gao, L., Madaan, A., Zhou, S., Alon, U., Liu, P., Yang, Y., Callan, J., and Neubig, G. PAL: Program-aided language models. In Krause, A., Brunskill, E., Cho, K., Engelhardt, B., Sabato, S., and Scarlett, J. (eds.), *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pp. 10764–10799. PMLR, 23–29 Jul 2023. URL https://proceedings.mlr.press/v202/gao23f.html.

Johnson, A. E. W., Bulgarelli, L., and Pollard, T. J. Deidentification of free-text medical records using pre-trained bidirectional transformers. In *Proceedings of the ACM Conference on Health, Inference, and Learning*, CHIL '20, pp. 214–221, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370462. doi: 10.1145/3368555.3384455. URL https://doi.org/10.1145/3368555.3384455.

Lample, G. and Charton, F. Deep learning for symbolic mathematics. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=S1eZYeHFDS.

Lattner, C. and Adve, V. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization*, 2004. doi: 10.1109/CGO.2004.1281665.

Lee, J., Lee, Y., Kim, J., Kosiorek, A., Choi, S., and Teh, Y. W. Set transformer: A framework for attention-based permutation-invariant neural networks. In *Proceedings of the 36th International Conference on Machine Learning*, 2019.

Liu, B. Anonymized BERT: An augmentation approach to the gendered pronoun resolution challenge. In *Workshop on Gender Bias in Natural Language Processing*, 2019.

Liu, D., Metzman, J., and Chang, O. AI-powered fuzzing: Breaking the bug hunting barrier, 2023. URL https://security.googleblog.com/2023/08/ai-powered-fuzzing-breaking-bug-hunting.html.

Liu, K., Koyuncu, A., Bissyandé, T. F., Kim, D., Klein, J., and Le Traon, Y. You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, 2019.

Liu, Z., Tang, B., Wang, X., and Chen, Q. De-identification of clinical notes via recurrent neural network and conditional random field. *Journal of Biomedical Informatics*, 75:S34–S42, 2017. ISSN 1532-0464. doi: https://doi.org/10.1016/j.jbi.2017.05.023. URL https://www.sciencedirect.com/science/article/pii/S1532046417301223. Supplement: A Natural Language Processing Challenge for Clinical Records: Research Domains Criteria (RDoC) for Psychiatry.

Marzoev, A., Madden, S., Kaashoek, M. F., Cafarella, M. J., and Andreas, J. Unnatural language processing: Bridging the gap between synthetic and natural language data. *arXiv preprint arXiv:2004.13645*, abs/2004.13645, 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*, 2019.

Minot, J. R., Cheney, N., Maier, M., Elbers, D. C., Danforth, C. M., and Dodds, P. S. Interpretable bias mitigation for textual data: Reducing genderization in patient notes while maintaining classification performance. *ACM Trans. Comput. Healthcare*, 2022.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, 2019.

Polu, S., Han, J. M., Zheng, K., Baksys, M., Babuschkin, I., and Sutskever, I. Formal mathematics statement curriculum learning. *arXiv preprint arXiv:2202.01344*, 2022.

Renda, A., Chen, Y., Mendis, C., and Carbin, M. Difftune: Optimizing cpu simulator parameters with learned differentiable surrogates. In *IEEE/ACM International Symposium on Microarchitecture*, 2020. doi: 10.1109/MICRO50266.2020.00045.

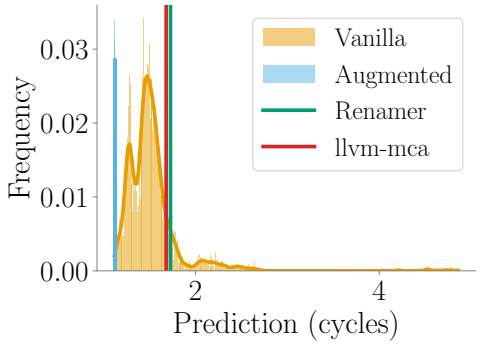
Renda, A., Ding, Y., and Carbin, M. Programming with neural surrogates of programs. In *ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, 2021.

Snavely, N. Lecture notes: Cs5670: Computer vision – lecture 5: Feature invariance, 2019. URL https://www.cs.cornell.edu/courses/cs5670/2019sp/lectures/lec05_invariance.pdf.

Su, J., Lu, Y., Pan, S., Wen, B., and Liu, Y. Roformer: Enhanced transformer with rotary position embedding. *arXiv preprint arXiv:2104.09864*, 2021.

Wennberg, U. and Henter, G. E. The case for translation-invariant self-attention in transformer-based language models. *arXiv preprint arXiv:2106.01950*, 2021.

Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., Davison, J., Shleifer, S., von Platen, P., Ma, C., Jernite, Y., Plu, J., Xu, C., Scao, T. L., Gugger, S., Drame, M., Lhoest, Q., and Rush, A. M. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 38–45, Online, October 2020. Association for Computational Linguistics. URL https://www.aclweb.org/anthology/2020.emnlp-demos.6.




```
mov  64(%rsp), %rax
sub  $1,    56(%rbp)
mov  16(%rax), %eax
```

llvm-mca: 1.68 cycles Vanilla: 1.51 cycles
Augmented: 1.12 cycles Renamer: **1.72** cycles

```
mov  64(%r8),  %rax
sub  $1,    56(%rbp)
mov  16(%rax), %eax
```

llvm-mca: 1.68 cycles Vanilla: 2.32 cycles
Augmented: 1.12 cycles Renamer: **1.72** cycles

Figure 2: The range of generated predictions for renamings of the basic block in Figure 1(a).

A Related Work

Anonymization and canonicalization. Anonymizing and the canonicalization of training data is an area of much focus in fields ranging from ethical AI to privacy-preserving AI. In an effort to reduce gender and region bias in gendered pronoun resolution Liu (2019) mask individual names by drawing from a set of canonical names. Similarly, for debiasing and preserving privacy in clinical ML, de-identification of data is a prevalent technique (Dernoncourt et al., 2016; Liu et al., 2017; Johnson et al., 2020; Minot et al., 2022). Most work, however, is focused on the process of automatic de-identification and not on the result of training on de-identified data. Minot et al. (2022) investigate the result of training on a canonicalized version of medical records. While canonicalization and de-identification reduce the bias of the model, they suffer from the fact that not every canonical representation of the same entity is guaranteed to have the same representation, and that inputs which have more entities than the number of canonical representations trained can't be represented. Furthermore, training on de-identified data is often associated with a degradation to network performance.

Transformer invariances. A wide variety of invariances and equivariances have been encoded into Transformer architectures. Lee et al. (2019) propose Set Transformer, which is invariant to permutations of the ordering of the input sequence. Fuchs et al. (2020) propose SE (3), which is equivariant to 3D translations and rotations, and evaluate on a variety of domains ranging from n-body simulations to point-cloud object classification. Su et al. (2021); Wennberg & Henter (2021) explore translation invariance in the context of natural language tasks. While these works enforce invariances, they all deal with spatial or positional invariances. To our knowledge, there is limited prior work on encoding invariances regarding the content of individual tokens.

B Renaming Sensitivity Case Study

Figure 2 presents a case study of each model's predictions on the basic block presented in Figure 1(a). The figure on the left is a histogram and corresponding density plot of predictions on semantically equivalent renamings of the basic block. To generate this plot, we uniformly sample 100,000 valid semantics-preserving register permutations σ, apply the permutation to the original block, then evaluate each model on the permuted block. The models under study are single trials of the best-performing BERT-Tiny models. The ground-truth timing for this basic block as output by llvm-mca is 1.68 cycles.

The vanilla model generates a range of predictions for permutations for this block, ranging from 1.11 cycles to 4.86 cycles. The predictions generated by the vanilla model are multimodal, though there are no clear indicators for which mode a given block will induce. The augmented model generates a significantly smaller range of predictions – though there is some variation on the order of one thousandth of a cycle, the predictions are essentially constant at 1.12 cycles. By construction, Renamer generates constant predictions for this basic block of 1.72 cycles.

C Training Details of llvm-mca

System. We evaluate using Pytorch-1.2.0 (Paszke et al., 2019), HuggingFace 4.17.0 (Wolf et al., 2020). Training is performed using an NVIDIA Tesla-V100.

Evaluation methodology. Each reported metric is the mean and the standard error of that metric across five trials with different random seeds. In each table of reported results, we use a Kruskal-Wallis test to determine if there is a significant difference between the means of the results of the models (with $p = 0.05$), and then a post-hoc Conover test to determine which models have the best error (again with $p = 0.05$), which are then bolded. [2]

Hyper-parameters. Across all models, we use the AdamW optimizer with a β_1, β_2 of 0.9 and 0.999 respectively.

We empirically determine the learning-rate, weight-decay, and dropout through a grid search over the hyper-parameters, selecting the hyper-parameter configuration which has the lowest validation error for the vanilla model. The hyper-parameters swept over are: learning-rate $\{3 \times 10^{-4}, 1 \times 10^{-4}, 5 \times 10^{-5}, 1 \times 10^{-5}\}$, weight-decay $\{0.0, 0.01\}$, and dropout $\{0.0, 0.1\}$.

Based on this sweep, the Tiny and Mini models use a learning rate of 3×10^{-4} and the Small models use 1×10^{-4}.

All models have a weight decay of 0.01, a dropout of 0, a batch size of 64, max sequence length of 128, and are trained for 500 epochs following Renda et al. (2021).

Objective. Following Chen et al. (2019) the loss and error metric are identical and are defined as the mean absolute percentage error (MAPE): $\mathcal{L}_{\text{MAPE}} = \sum_{x,y \in D} \frac{|f(x)-y|}{y}$.

D Evaluation on llvm-mca: Training Efficiency

In this section we evaluate the efficiency with which the vanilla, augmented, canonicalized, and Renamer models achieve various test errors on the llvm-mca task.

The time to process an input is the same across all models as they all have the same number of parameters, and Renamer only leverages components of the Transformer architecture (i.e. embedding table, attention mask) which the vanilla, augmented, and canonicalized architectures also employ. Since the time to train for an epoch is the same for all architectures, we directly compare the number of epochs needed to achieve comparable test error.

Figure 3 plots efficiency curves for the various model sizes on the BHive dataset. Each line shows the epochs required (on the y axis) to reach a given test error (on the x axis) for a given modeling approach. A more efficient model requires fewer epochs to reach a given target test error (i.e., lower is better).

Across all model sizes, Renamer achieves the same performance with significantly fewer training steps than the vanilla, augmented, and canonicalized models. This speedup is reflected in the gap between the MAPE versus epochs of training curves for the vanilla, augmented, canonicalized, and Renamer models.

We evaluate the speedup quantitatively by comparing the minimum number of epochs required to achieve the same best test error. Renamer reaches the best error of the vanilla model with 213, 123, and 290 fewer epochs for the Tiny, Mini, and Small architectures respectively, which corresponds to a relative decrease in steps to achieve the same performance of 42.77%, 24.75%, and 59.06%. As compared to the augmented model, Renamer requires 214, 403, and 419 fewer epochs for the Tiny, Mini, and Small variants respectively, which corresponds to a relative decrease in steps to achieve the same performance of 42.89%, 81.25%, and 83.80%. Finally, Renamer achieves the same error as the canonicalized model with 155 and 59 fewer epochs for the Tiny and Mini variants respectively, which corresponds to a relative decrease in training steps of 31.00% and 11.82%.

[2] `https://scikit-posthocs.readthedocs.io/en/latest/tutorial/#non-parametric-anova-with-post-hoc-tests`

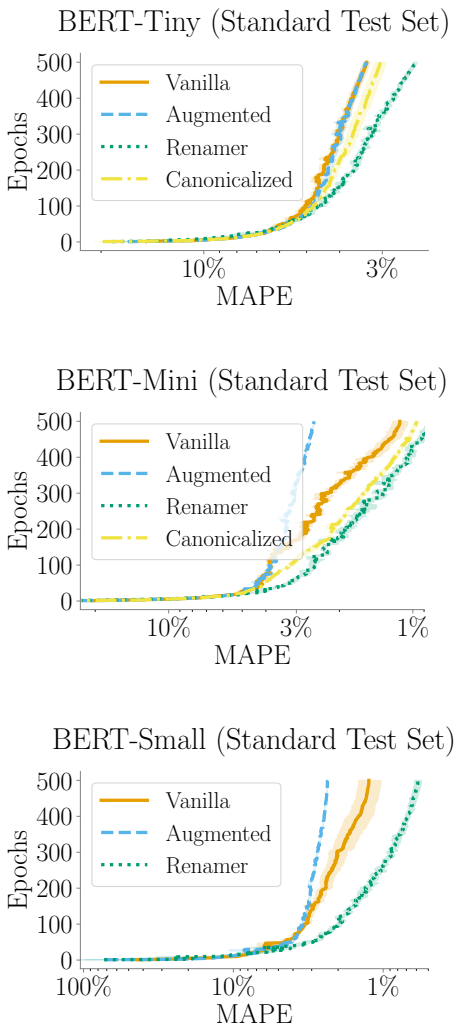


Figure 3: llvm-mca: Model efficiency on the original (unperturbed) test set. Each point on a curve corresponds to the number of epochs needed to reach a given test error for the specified architecture.

Table 3: Symbolic Algebra: Error of different model variants on the original test set.

	Model size
Model	BERT-Small
Vanilla	**0.79% ±0.13%**
Canoncalized	**0.71% ±0.14%**
Renamer	**0.80% ±0.06%**

Table 4: Symbolic Algebra: Error of different model variants on the augmented test set.

	Model size
Model	BERT-Small
Vanilla	4.68% ±0.69%
Canoncalized	4.98% ±0.56%
Renamer	**2.38% ±0.16%**

E Evaluation on Symbolic Algebra

We next evaluate Renamer on a symbolic algebra task, comparing against a suite of baselines. We show that on this task, Renamer matches the in-distribution error, and is able to generalize to out-of-distribution inputs better than a suite of baseline models.

E.1 Task

We evaluate Renamer on an invariant modification of Lample & Charton (2020)'s Backward dataset. Each input in the dataset is composed of a pair of expressions and the corresponding label is whether one expression is the partial derivative of the other with respect to the variable `x`. We present two examples below:

$$(\texttt{sin x} \stackrel{?}{=} \texttt{cos x}, \texttt{true})$$

$$(\texttt{mul a0 cos x} \stackrel{?}{=} \texttt{add a0 x}, \texttt{false})$$

Tokens in the dataset include standard mathematical operators (`add`, `sub`, `mul`, `pow`, `sin`, `cos`, etc.), *coefficient variables* (`a0`, `a1`, and `a2`), and *input variables* (`x`, `y`, and `z`).

Variable renaming invariance in Backward dataset. For this task, coefficient variables can be renamed to any other coefficient variable, input variables other than `x` can be renamed to any other input variable other than `x`. The variable `x` and operators cannot be renamed.

Thus, the view mapping maps: coefficient variables to the view `coefficient`, input variables other than `x`, to the view `input-nonx`, the variable `x` to the view `input-x`, and operators to the view `operator`.

We define two tokens to be coreferential if they are the same variable – that is, every token is only coreferential with itself.

Dataset. We use the same dataset generation technique as Lample & Charton (2020)'s Backward dataset, but randomly pair each expression with its derivative with probability 0.5, and a random other expression from the dataset with probability 0.5. This turns the task into an invariant classification task, rather than the equivariant generation task it is in Lample & Charton (2020). The generated modified Backward dataset is composed of a training set of $300{,}000$ examples, and validation and test set of 9128 and 9139 examples respectively.

E.2 Evaluation Methodology

Models. The details of the models used are the same as those defined in Section 5.2. However, for this symbolic algebra task we only use the BERT-Small model size (the largest model evaluated in Section 5).

System. The system details and other evaluation methodology are the same as those in Appendix C.

Hyper-parameters. We use the same hyper-parameters as the those for BERT-Small reported in Appendix C, with the exception of training for 50 epochs (rather than 500).

Objective. We train the model using the cross entropy loss, and report the classification error on the test set.

E.3 Results

We again present results on the vanilla, canonicalized, and Renamer models on the original test set and an extended version of the test set. We again define test error as the test error of the epoch with the lowest validation error.

Standard test set. Table 4 presents the test errors of all models on the standard test set, achieving between 0.71% and 0.80% error. We find that all models perform similarly well on the original test set: the statistical test discussed in Appendix C does not distinguish between the errors from any model.

Extended test set. We also evaluate on a version of the test set extended with an additional coefficient variable, labeled `a3`. This experiment tests the hypothesis that the Renamer generalizes better to unseen variable names than all other models.

While the performance of all models decreases on the extended test set, we find that on the extended test set Renamer significantly outperforms all other models. The vanilla model achieves an error of 4.68% on the extended test set while Renamer reaches an error of 2.38%, a 49.1% decrease in error compared to the vanilla model. Similarly, the canonicalized model achieves an error of 4.98%, meaning the Renamer has a 52.2% decrease in error compared to the canonicalized model.