Large Binary Models

Language Modeling of Compiled Bytecode, Without Language

Jovonni L. Pharr
Georgia Cyber Warfare Range
info@gacwr.org

August 9, 2026

v0.0.1

Abstract

The language model is built upon language. Its alphabet is the tokenized surface form of human prose, and its competence is competence in that surface form. Yet the true substrate of a program is not English; it is compiled binary, a finite alphabet of machine operations. This work asks whether a model can learn to predict and to generate that binary directly, without the intermediary of natural language, and finds that it can, and readily. We train a byte-level model directly upon the raw bytecode of compiled Python—the operation bytes themselves, carrying no source text and drawn from an alphabet of two hundred and fifty-eight symbols—and observe that compiled binary is an unusually tractable modeling substrate: a model of ten to forty million parameters drives the held-out cross-entropy from the random baseline of 8.01 bits per token to 0.46, and its generations, disassembled, reproduce the genuine control- and data-flow idioms of real programs. We argue that a finite and grounded operation alphabet is the natural primitive vocabulary for neural computation, connect the line to the general purpose neural computer, and state plainly the milestone that remains: the generation of programs that not only disassemble but *run*.

Contents

I Preliminaries **2**

1 The State of Today **2**

2 Related Work **2**
2.1 Neural models of code as language . . . 2
2.2 Modeling compiled and low-level representations . . . 2
2.3 Learned execution and neural program interpreters . . . 3
2.4 Program synthesis with execution and correctness feedback . . . 3
2.5 Compression as a measure of modeling . . . 3
2.6 Small and data-efficient language models . . . 4

3 Current Problems **4**

II Proposal **4**

4 The Corpus **4**

5 The Model and Its Behavior **5**

6 Connection to the General Purpose Neural Computer **9**

7 From Structural to Runnable Generation **9**

8 Open Problems **11**

9 Conclusion **11**

Part I
Preliminaries

1 The State of Today

The contemporary model of code is, almost without exception, a model of *source text*. It is trained on the characters a human types—identifiers, keywords, whitespace, comments—and it is evaluated on its ability to continue that text plausibly. This is a reasonable choice, for source text is what humans read and write, and it is abundant. But it is worth observing that source text is not what a machine executes. Between the text and the execution stands a compiler, and the object the compiler produces—a sequence of operations drawn from a fixed instruction set—is the program in the only sense that matters to the machine. To model the source is to model the human-facing shadow of computation; to model the compiled operations is to model computation itself.

2 Related Work

This work sits at the confluence of several lines: the neural modeling of code as language, the modeling of *compiled* and low-level artifacts, the learning of program execution, program synthesis under execution feedback, and the long-standing identification of sequence prediction with compression. We position our two distinctive contributions against each in turn: (i) modeling a program's *compiled bytecode directly as the token stream*, with no source text, and (ii) grounding evaluation of the model's own generations in an *execution-based correctness oracle* rather than in perplexity or surface match.

2.1 Neural models of code as language

The dominant paradigm treats a program as *source text* and applies the machinery of language modeling to it. Encoder models such as CodeBERT [2] and encoder–decoder models such as CodeT5 [3] pre-train bimodal representations of code and natural language. Decoder-only models scaled this to generation: Codex [4] synthesized functional code from docstrings, and open successors such as StarCoder [6] and Code Llama [7] trained on large public source corpora. AlphaCode [5] reached competitive-programming performance by generating and filtering large candidate populations, and a related strand learns to *translate* between languages via back-translation on source text [8]. In every case the object of study is the human-facing surface form, drawn from a subword vocabulary of tens of thousands of units. Our substrate is the opposite end of the compilation pipeline: the finite alphabet of operation bytes the compiler emits, with no identifiers and no source text at all.

2.2 Modeling compiled and low-level representations

A separate literature learns representations of *compiled* artifacts, almost always for *analysis* rather than generation. Binary code similarity detection embeds functions from control-flow

graphs [9] or from jump-aware and instruction-level transformers over disassembled assembly [11, 10]. Neural decompilation maps low-level code back to a higher-level form [13, 14], and Nova [12] is a *generative* language model over assembly trained with a decompilation objective. Closest in spirit to our raw-byte view, MalConv [15] classifies malware by consuming the raw bytes of an entire executable. Two distinctions define our contribution. First, prior work overwhelmingly operates on *textual assembly* (a disassembled, human-readable rendering) or on whole-file bytes for a classification label; we model the raw `co_code` operation-byte stream, and then the full serialized code object, as the generative token stream itself. Second, these systems are discriminative or translational, whereas we ask a model to *generate* novel compiled artifacts that reconstruct and run in the virtual machine.

2.3 Learned execution and neural program interpreters

A complementary program is to learn what code *does*. "Learning to Execute" [16] trained recurrent networks to predict the output of short programs; Neural Programmer-Interpreters [17] learned compositional execution traces; and neural algorithmic reasoning imitates the step-wise behavior of classical algorithms [18]. Within code pre-training, execution-aware methods inject dynamic signal: TRACED [19] pre-trains on execution traces and CodeExecutor [20] predicts execution traces of Python. These works learn a *map from program to behavior*. Ours runs in the opposite direction—generating the program—and it does not *predict* execution; it *invokes* the real CPython virtual machine on the generated code object and checks the result. Learned execution is the sequel we flag as differentiable execution in our open problems, not a claim we make here.

2.4 Program synthesis with execution and correctness feedback

The synthesis literature has long induced programs as structured objects checked against a specification rather than continued as text. DeepCoder [21] predicts program attributes to guide search; RobustFill [22] learns string programs under noisy I/O; and execution-guided synthesis conditions generation on intermediate states [23]. In the LLM era, correctness signal is folded back into training and decoding: CodeRL [24] optimizes against unit-test outcomes, and self-debugging [25] iterates on execution feedback. Evaluation by *functional correctness*—executing candidates against tests, as with the pass@k of Codex [4] and the filtering of AlphaCode [5]—is now standard. We adopt correctness as the metric of record, and we are explicit that execution-grounded evaluation is *not* itself novel. What differs is the object checked: a reconstructed `types.CodeType` executed under sandboxed resource limits, verified by a *differential oracle* that independently decompiles the generated operations to an expression and confirms the virtual machine's output equals that expression's value on probe inputs. We further separate memorization from generation with a *novel-correct* rate—programs that both run correctly and whose operation-and-constant signature is absent from the training corpus—rather than novelty-by-held-out-prompt.

2.5 Compression as a measure of modeling

That a good sequence model is a good compressor is a long-standing thesis, operationalized by the Hutter Prize [28] and revisited by "Language Modeling Is Compression" [26], which frames foundation models directly as lossless compressors; neural compressors such as NNCP [27] achieve state-of-the-art bits-per-byte on text. Our bits-per-token reporting and our comparison against `gzip`, `bzip2`, and an order-two n-gram on the identical held-out bytes are a direct instance of this framing; the novelty is not the yardstick but the substrate—compiled bytecode, which compresses far more sharply than the irregular human text these methods target.

2.6 Small and data-efficient language models

Finally, our models are small—ten to forty million parameters—placing them among efforts to show that competence need not require scale when the data are clean or structured. TinyStories [29] shows sub-ten-million-parameter models producing coherent text from a constrained corpus; the phi line argues that "textbook-quality" data yields strong small models [30]; and SmolLM2 [31] documents data-centric training of a compact model. Our result is a data-side analogue: the compiled substrate is so regular that a small model attains near-deterministic grammar and runnable generation on it, precisely because the alphabet is finite and the grammar rigid.

3 Current Problems

Two problems motivate the direct modeling of compiled binary. The first is a problem of *grounding*. A model of source text learns the statistics of a human notation; it does not, except incidentally, learn the semantics of execution. The compiled operations, by contrast, are the semantics—each is a defined transformation of machine state—so a model of the operations is a model of what the program does, not merely of how it is written. The second is a problem of *efficiency and regularity*. Source text is drawn from a vocabulary of tens of thousands of subword units and is riddled with the irregularities of human expression; compiled operations are drawn from a fixed alphabet of some hundreds of symbols and obey the rigid grammar of a virtual machine. The substrate is smaller and far more regular, and one may reasonably expect it to be more predictable.

Part II
Proposal

4 The Corpus

Definition 1 (Bytecode corpus)**.** *Compile every available Python source file of a standard installation—the standard library together with installed packages—to a code object. From each code object, recursively extract the raw operation-byte string of every nested code object it contains. Concatenate these strings, delimiting code objects and source files by two reserved symbols. The resulting alphabet is the two hundred and fifty-six byte values together with a code-object separator and a file separator, for a vocabulary of two hundred and fifty-eight symbols.*

Applied to a standard installation, this procedure yields a corpus of 124.2 million tokens over 414,522 code objects drawn from approximately twenty thousand source files, of which none failed to compile (Table 1). The final five percent of the corpus is held out for evaluation. No source text enters the model; the model observes only the compiled operations.

Table 1: The compiled-bytecode corpus and the three trained models. Bits-per-token is the held-out cross-entropy in base two; the uniform baseline is $\log_2 258 = 8.01$.

Corpus		**Models (width / depth)**			
tokens	124.2M	metric	d320/L8	d448/L12	d512/L16
code objects	414,522	parameters	10M	18M	29M
source files	∼20,000	init. bits/token	8.01	8.01	8.01
compile failures	0	final bits/token	0.531	0.480	**0.460**
vocabulary	258	compression	15.1×	16.7×	**17.4**×

5 The Model and Its Behavior

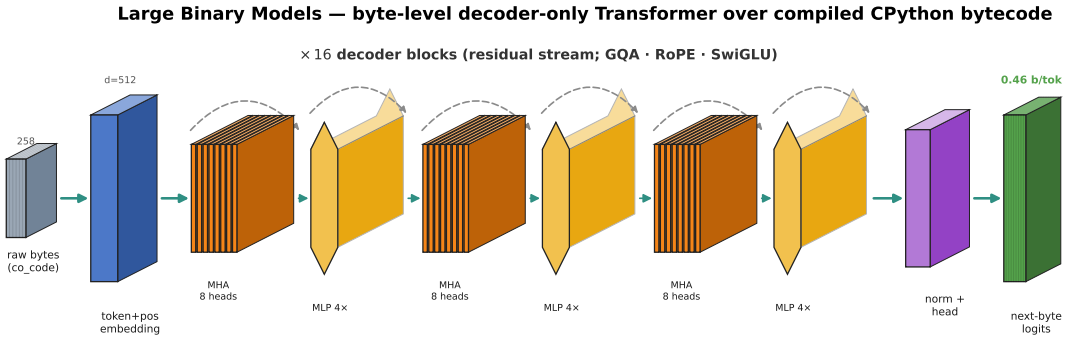


Figure 1: The binary model: a byte-level decoder-only Transformer over the two-hundred-and-fifty-eight-symbol alphabet of raw operation bytes. No natural language enters the model; it consumes compiled bytecode and predicts the next operation byte, driving held-out cross-entropy from the uniform 8.01 bits/token to 0.46.

The model is a standard decoder-only Transformer over the byte alphabet, of ten to forty million parameters according to width and depth. Training minimizes the mean next-token negative log-likelihood over the operation stream $b_{1:n}$,

$$\mathcal{L}(\theta) \;=\; -\frac{1}{n}\sum_{i=1}^{n}\log_2 p_\theta(b_i \mid b_{<i}) \quad \text{bits/token}, \qquad \mathcal{L}_{\text{uniform}} = \log_2 258 = 8.01, \tag{1}$$

and drives the held-out value from the uniform baseline of 8.01 down to 0.46 bits per token for the largest model—a compression ratio

$$\kappa \;=\; \frac{\mathcal{L}_{\text{uniform}}}{\mathcal{L}(\theta)} \;=\; \frac{8.01}{0.46} \;\approx\; 17.4, \tag{2}$$

and an indication that compiled binary is highly structured and highly learnable directly, without the mediation of source text (Figure 3). Compression improves monotonically with model scale, from $\kappa = 15.1$ at ten million parameters to $\kappa = 17.4$ at twenty-nine million (Figure 4). The model acquires the grammar of the virtual machine: the layout of the operation stream, the sequences of operations that constitute valid computations, and the regularities of their arguments.

We are careful that the comparison against a uniform prior, while exact, is a weak one: any model of a structured source beats the uniform baseline. The honest question is whether the model beats a serious classical compressor on the identical held-out bytes. It does. Measured on the same held-out five percent (Figure 2), an order-two byte n-gram attains 1.784 bits per token, `gzip` 1.716, and `bzip2`—a Burrows–Wheeler compressor with long-range reach, and the strongest of the three—1.246; the model's 0.460 is 2.7× better than `bzip2` and 3.9× better than the n-gram. A small neural model of the operation stream compresses real bytecode substantially beyond the best general-purpose compressors, which is the claim a uniform baseline cannot make.

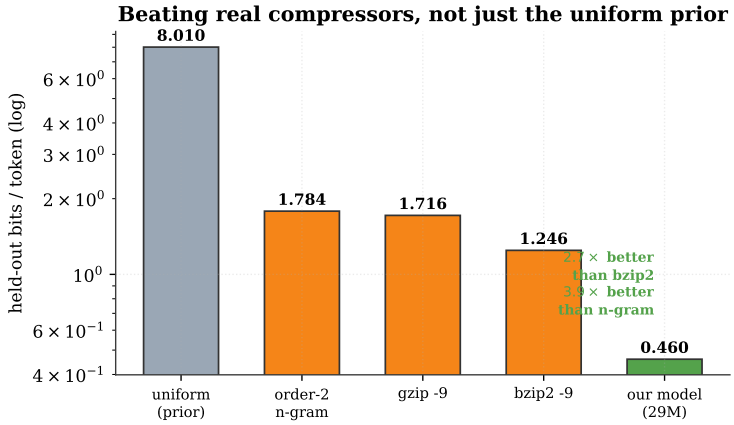


Figure 2: Held-out bits-per-token against honest baselines on the identical byte stream (log scale). The model at 0.460 beats an order-two n-gram (1.784), `gzip` (1.716), and `bzip2` (1.246)—2.7× below the strongest classical compressor. Baselines are computed by `compute_baselines.py`.

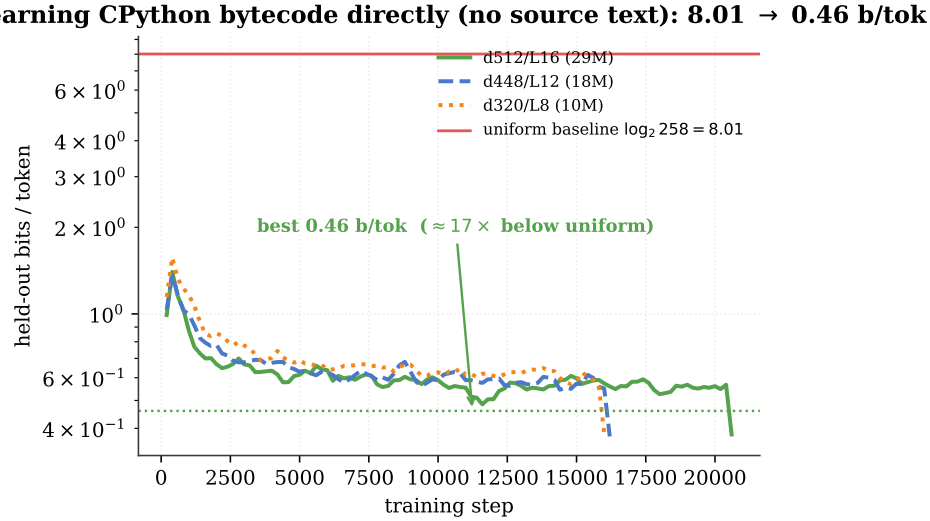


Figure 3: Held-out bits-per-token (Eq. 1, log scale) against training step for the three model sizes, from the uniform baseline of 8.01 toward 0.46.

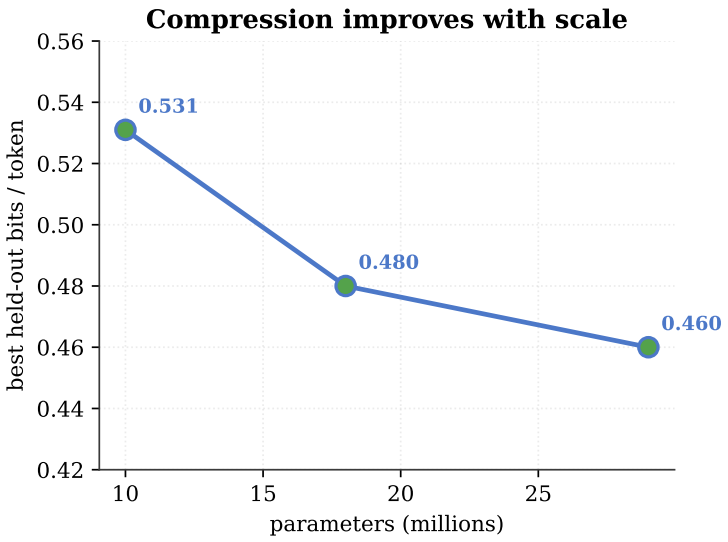


Figure 4: Best held-out bits-per-token against parameter count: compression improves monotonically with scale.

The character of what the model *generates* is more telling than the perplexity. Sampled generations, disassembled, are not noise of low perplexity but genuine idioms of real programs. A generation may open with the prologue operation that begins every code object, proceed through the operations that load a global name and then access an attribute upon it—the pattern of a qualified reference such as a call to a library function—and close with the operation that returns a value. The model reproduces even the fine structure of the operation stream, including the approximately correct counts of the inline-cache placeholders that follow certain operations in the modern virtual machine.

The claim that the model has learned the true distribution over operations, rather than a plausible-looking caricature of it, admits a direct quantitative test. We compute the frequency of each opcode in the real corpus, measured on the raw operation bytes exactly as the model observes them, and the frequency of each opcode in a sample of the model's own generations, and we compare the two. The distributions coincide almost exactly: over the full opcode alphabet the Pearson correlation between the corpus and generated frequencies is $r = 0.998$ (Figure 5). We note that this test required care—an earlier comparison that measured the corpus via the disassembler's *logical* instruction stream, while measuring generations from raw bytes, reported a spurious $r = 0.14$, because the disassembler hides the inline-cache placeholders that dominate the raw stream; once both sides are measured on the same raw representation, the agreement is near-perfect. It is the mismatch, not the model, that the low number had measured.

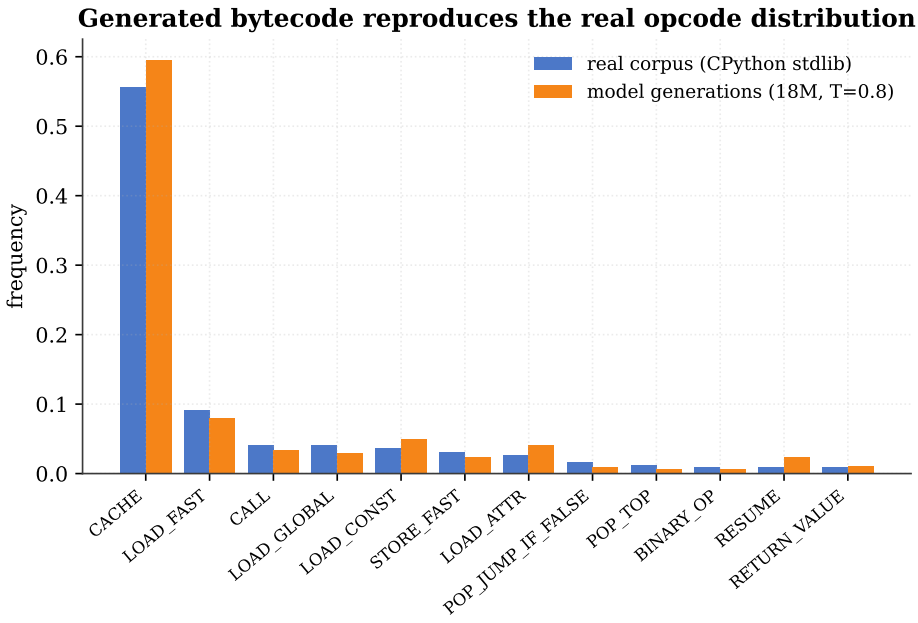


Figure 5: Opcode frequency in the real corpus (dark) and in the model's own generations (light), for the twelve most common operations, measured identically on the raw byte stream. The generated distribution reproduces the corpus with correlation $r = 0.998$ over the full alphabet.

Formally, writing f_o^{corp} and f_o^{gen} for the corpus and generated frequency of opcode o, the agreement is $r = \text{corr}\big(f^{\text{corp}}, f^{\text{gen}}\big) = 0.998$. That the match is so tight is a consequence of how *regular* the substrate is: the bytecode obeys a near-deterministic grammar in which each operation sharply constrains its successor. Figure 6 exhibits that grammar as the matrix of opcode-to-opcode transition probabilities $P(o_{t+1} \mid o_t)$ measured on the corpus. Its rows are close to one-hot, and the mean conditional (next-opcode) entropy,

$$H(o_{t+1} \mid o_t) \;=\; -\sum_{o} P(o) \sum_{o'} P(o' \mid o)\, \log_2 P(o' \mid o), \tag{3}$$

is small—the low-entropy structure that a compact model learns so readily, and that a model of irregular human source text cannot exploit.

That the model has learned this grammar, and not merely the marginal frequencies, is shown directly by comparing the transition matrix of the real corpus to the transition matrix of the model's *own* generations (Figure 6). The two are near-identical: the Pearson correlation between the corpus and generated transition matrices is $r = 0.956$—a bigram-level agreement far stronger than, and not implied by, the unigram-level $r = 0.998$ of the previous section. We stress that this is a *transition* matrix, not a confusion matrix: it records which operation follows which, so its structure is the columnar and block pattern of the bytecode grammar (every attribute access routing through an inline-cache placeholder, every return preceded by a value load), and no diagonal is expected or meaningful.

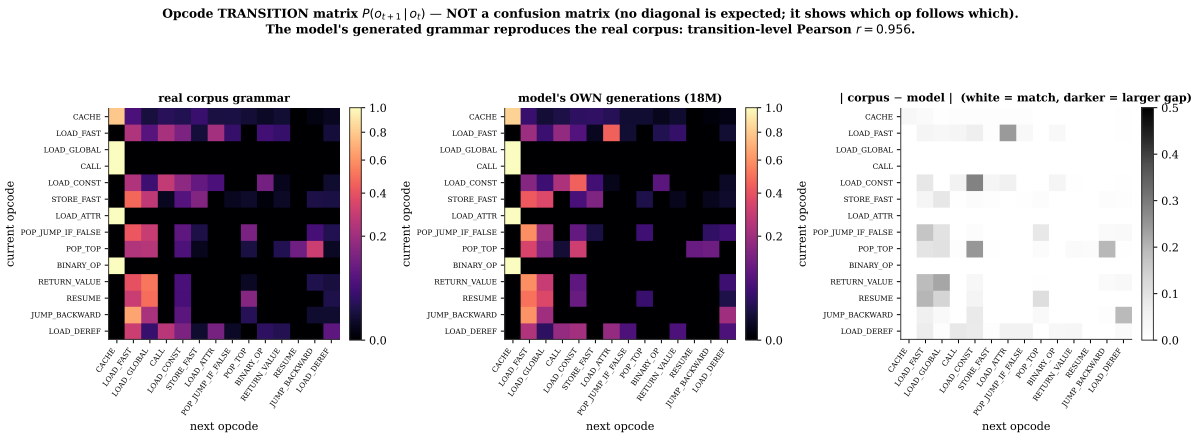


Figure 6: Opcode-to-opcode transition probabilities $P(o_{t+1} \mid o_t)$ over the fourteen most frequent operations. *Left:* the real corpus grammar. *Centre:* the same matrix computed from the model's own generations. *Right:* their absolute difference (light = agreement). The model reproduces the corpus grammar at transition-level correlation $r = 0.956$. This is a transition matrix, not a confusion matrix—no diagonal is expected; the bright columns are the near-deterministic successors (e.g. every CALL and LOAD_GLOBAL followed by an inline-cache placeholder), the low-entropy regularity of Eq. 3.

We are careful, nonetheless, not to overstate the significance of a superficial validity statistic. That essentially all generated operation bytes are legal operations is inflated by the frequency of the cache placeholder, which is always legal; the honest signal is the combination of the distributional match above with structure, in that generations open with the correct prologue, employ real load, store, call, and return operations, and terminate correctly. This is learned structure, and it is not yet a verified-runnable program.

6 Connection to the General Purpose Neural Computer

A finite operation alphabet is precisely the primitive vocabulary that a neural computer requires. In the companion work on the general purpose neural computer, the *semantics* of such an alphabet are learned from the observation of execution; here the *distribution* over operation sequences that real programs occupy is learned from a corpus of compiled code. The union of the two is a neural system that both writes operation sequences and knows what those sequences do—a system that writes and runs real, auditable programs.

7 From Structural to Runnable Generation

The generations described so far are structurally plausible but not runnable: the raw operation bytes, spliced into a code object and executed, fail, because their arguments index the constant and name tables of the code object, which a model of `co_code` alone does not co-generate. The natural response is to stop discarding that metadata. We serialize the *entire* code object—operation stream together with its constant table, name table, argument counts, and stack size—into a single byte-token stream, framed by a small set of structural marker tokens (raising the vocabulary from two hundred and fifty-eight to two hundred and seventy-eight), with integer constants encoded as sign-and-varint byte sequences. A model trained on this representation emits, for each generation, both the operations and the tables they reference.

To measure the result we built a verification harness that reconstructs a genuine `types.CodeType` from a generation and executes it in an isolated subprocess under strict resource limits—bounded

processor time and memory, no file writes, and an emptied builtin namespace so that no external primitive is reachable—so that a malformed or hostile generation is contained to a disposable child rather than trusted, and a differential correctness oracle that symbolically decompiles the generated operations to an arithmetic expression and checks that the virtual machine's output equals the expression's value on a battery of eight input pairs. The oracle scores the ground-truth corpus at one hundred and fifty of one hundred and fifty, so a failure on a generation is a real defect. On a first target of straight-line integer functions, a small fourteen-million-parameter model trained on this serialization moves the runnable rate from a categorical *zero*—unrunnable by construction, since the metadata was absent—to one hundred percent by a thousand optimizer steps (Figure 7). We report the model on a corpus of two rungs of the difficulty ladder at once: straight-line integer functions and, added on top, functions with a conditional branch (an `if/else` that the oracle checks as a ternary). Over four hundred sampled generations, more than ninety-nine percent are *correct*: the executed bytecode computes exactly what its own disassembly says, on every probe input. And the generations are not memorized: seventy-five percent are *novel*, their operation-and-constant signature absent from the training corpus, so that the model's novel-correct rate—new programs that both run and compute what they claim—is **seventy-four percent** (a Wilson ninety-five-percent interval of $[69.5, 78.0]$), up from sixty-two percent on the straight-line rung alone.

Because the correctness oracle is perfect and cheap, a single generation is not the operative quantity: one may sample repeatedly and keep an oracle-verified program. With a per-sample novel-correct probability $p \approx 0.74$, the probability that at least one of k independent samples is a valid novel program is $1-(1-p)^k$, which we confirm empirically—0.94 at two samples, 0.99 at three. We are careful about what this does and does not mean: generation here is unconditional (there is no posed problem), so this measures the reliable *yield* of valid, novel programs under repeated sampling, not problem-solving `pass@`k. It is nonetheless the practical statement—paired with the execution oracle, the model reliably *delivers* runnable, novel bytecode at a handful of samples—and it is the sampling budget that any downstream self-improvement loop would spend.

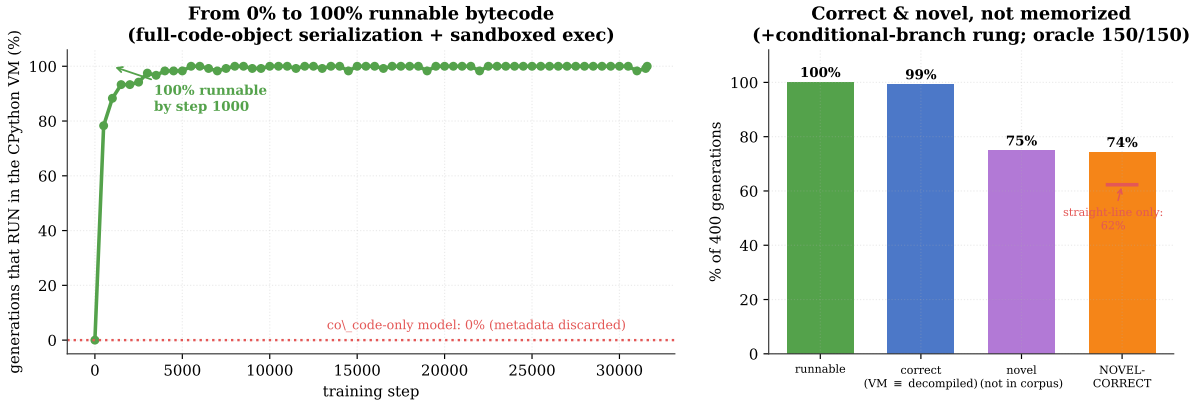


Figure 7: Runnable bytecode generation. *Left:* the fraction of generations that reconstruct into a real code object and execute in the CPython virtual machine, from the categorical zero of a `co_code`-only model (metadata discarded) to one hundred percent by a thousand steps. *Right:* over four hundred generations spanning straight-line *and* conditional-branch functions, more than ninety-nine percent are correct (virtual-machine output equals the independently decompiled expression on eight probe inputs), and seventy-five percent are novel programs absent from the training corpus—so novel-correct is seventy-four percent (Wilson 95% interval $[69.5, 78.0]$), up from sixty-two percent on the straight-line rung alone. The correctness oracle scores the ground-truth corpus at 150/150.

8 Open Problems

The runnable result of Section 7 now covers straight-line integer functions and a first control-flow rung, the conditional branch; the agenda is to keep climbing the difficulty ladder one virtual-machine feature at a time—names and global lookups over a whitelist, loops with their exception tables, nested code objects, and finally real library functions—holding novel-correct as the metric at each rung. Beyond it lie the learning of a translation from source to bytecode, which is a learned compiler; the prediction of behaviour from bytecode, which is differentiable execution; and the extension of the substrate beyond one virtual machine to others, including the stack machine of WebAssembly and the instruction sets of physical processors.

9 Conclusion

Compiled binary is a viable and unusually efficient substrate for neural modeling. A small model learns the grammar of a real virtual machine's bytecode directly, from bytes alone and with no natural language, and generates the genuine idioms of computation. Coupled with a neural computer that grounds the *meaning* of the operations, the direct modeling of compiled binary points toward neural systems that operate upon the true substrate of computation rather than upon its human-facing shadow.

References

[1] J. L. Pharr. *The General Purpose Neural Computer.* Georgia Cyber Warfare Range, 2026.

[2] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, M. Zhou. *CodeBERT: A Pre-Trained Model for Programming and Natural Languages.* Findings of EMNLP, 2020. arXiv:2002.08155.

[3] Y. Wang, W. Wang, S. Joty, S. C. H. Hoi. *CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation.* EMNLP, 2021. arXiv:2109.00859.

[4] M. Chen et al. *Evaluating Large Language Models Trained on Code.* 2021. arXiv:2107.03374.

[5] Y. Li et al. *Competition-level Code Generation with AlphaCode.* Science 378(6624):1092–1097, 2022.

[6] R. Li et al. *StarCoder: May the Source Be With You!* 2023. arXiv:2305.06161.

[7] B. Rozière et al. *Code Llama: Open Foundation Models for Code.* 2023. arXiv:2308.12950.

[8] B. Rozière, M.-A. Lachaux, L. Chanussot, G. Lample. *Unsupervised Translation of Programming Languages.* NeurIPS, 2020. arXiv:2006.03511.

[9] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, D. Song. *Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection.* ACM CCS, 2017.

[10] X. Li, Y. Qu, H. Yin. *PalmTree: Learning an Assembly Language Model for Instruction Embedding.* ACM CCS, 2021. arXiv:2103.03809.

[11] H. Wang, W. Qu, G. Katz, W. Zhu, Z. Gao, H. Qiu, J. Zhuge, C. Zhang. *jTrans: Jump-Aware Transformer for Binary Code Similarity Detection.* ISSTA, 2022. arXiv:2205.12713.

[12] N. Jiang, C. Wang, K. Liu, X. Xu, L. Tan, X. Zhang, P. Babkin. *Nova: Generative Language Models for Assembly Code with Hierarchical Attention and Contrastive Learning.* ICLR, 2025. arXiv:2311.13721.

[13] C. Fu, H. Chen, H. Liu, X. Chen, Y. Tian, F. Koushanfar, J. Zhao. *Coda: An End-to-End Neural Program Decompiler.* NeurIPS, 2019.

[14] H. Tan, Q. Luo, J. Li, Y. Zhang. *LLM4Decompile: Decompiling Binary Code with Large Language Models.* EMNLP, 2024. arXiv:2403.05286.

[15] E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro, C. K. Nicholas. *Malware Detection by Eating a Whole EXE.* AAAI Workshops, 2018. arXiv:1710.09435.

[16] W. Zaremba, I. Sutskever. *Learning to Execute.* 2014. arXiv:1410.4615.

[17] S. Reed, N. de Freitas. *Neural Programmer-Interpreters.* ICLR, 2016. arXiv:1511.06279.

[18] P. Veličković, R. Ying, M. Padovano, R. Hadsell, C. Blundell. *Neural Execution of Graph Algorithms.* ICLR, 2020. arXiv:1910.10593.

[19] Y. Ding, B. Steenhoek, K. Pei, G. Kaiser, W. Le, B. Ray. *TRACED: Execution-aware Pre-training for Source Code.* ICSE, 2024. arXiv:2306.07487.

[20] C. Liu, S. Lu, W. Chen, D. Jiang, A. Svyatkovskiy, S. Fu, N. Sundaresan, N. Duan. *Code Execution with Pre-trained Language Models.* Findings of ACL, 2023. arXiv:2305.05383.

[21] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, D. Tarlow. *DeepCoder: Learning to Write Programs.* ICLR, 2017.

[22] J. Devlin, J. Uesato, S. Bhupatiraju, R. Singh, A. Mohamed, P. Kohli. *RobustFill: Neural Program Learning under Noisy I/O.* ICML, 2017. arXiv:1703.07469.

[23] X. Chen, C. Liu, D. Song. *Execution-Guided Neural Program Synthesis.* ICLR, 2019.

[24] H. Le, Y. Wang, A. D. Gotmare, S. Savarese, S. C. H. Hoi. *CodeRL: Mastering Code Generation through Pretrained Models and Deep Reinforcement Learning.* NeurIPS, 2022. arXiv:2207.01780.

[25] X. Chen, M. Lin, N. Schärli, D. Zhou. *Teaching Large Language Models to Self-Debug.* ICLR, 2024. arXiv:2304.05128.

[26] G. Delétang et al. *Language Modeling Is Compression.* ICLR, 2024. arXiv:2309.10668.

[27] F. Bellard. *NNCP v2: Lossless Data Compression with Transformer.* Technical report, bellard.org, 2021.

[28] M. Hutter. *The Hutter Prize for Lossless Compression of Human Knowledge.* 2006. http://prize.hutter1.net/.

[29] R. Eldan, Y. Li. *TinyStories: How Small Can Language Models Be and Still Speak Coherent English?* 2023. arXiv:2305.07759.

[30] S. Gunasekar et al. *Textbooks Are All You Need.* 2023. arXiv:2306.11644.

[31] L. Ben Allal et al. *SmolLM2: When Smol Goes Big—Data-Centric Training of a Small Language Model.* 2025. arXiv:2502.02737.