The General Purpose Neural Computer

A Learned, Grounded, and Deterministic Interpreter of a Turing-Complete Instruction Set

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

August 9, 2026
v0.0.1

Abstract

The prevailing artifact of machine intelligence today is a *stochastic* one: the language model, an extraordinary approximator of natural language whose computation is sampled, non-repeatable, and illegible. For the large class of problems that are fundamentally *algorithmic*, this is exactly the wrong set of properties. This work proposes a *General Purpose Neural Computer* (GPNC): a differentiable machine that learns to interpret a real, Turing-complete instruction set—the WebAssembly integer core—directly from the observation of its execution, runs deterministically and auditably, and generalizes exactly to program lengths, loop trip-counts, memory sizes, and operand magnitudes it never encountered during learning. We are precise about the boundary of the contribution. That neural networks can be made universal is settled and is not claimed here; hand-constructed weights already emulate one-instruction computers, and finite-precision Turing-completeness is established. The contribution is the intersection that no prior differentiable computer has occupied at once: a computer that is simultaneously *learned* from traces rather than hand-compiled, *grounded* in a real instruction set with a machine-checked formal semantics, *length-generalizing* through resolution-invariant operators, and *deterministic* through a discretize-and-continue membrane that arrests the multi-step drift which historically defeated neural memories. We formalize the machine, prove a label-free identifiability result (the operational semantics of all fifteen opcodes of a Turing-complete instruction set are recoverable from before-and-after state pairs alone, with no supplied instruction labels), and exhibit the recovered interpreter executing real loop programs. We are equally precise about what remains open, and treat those items as a roadmap rather than a wall.

Contents

I Preliminaries **2**

1 The State of Today **2**

2 Related Work **3**

3 Current Problems **4**

4 The Discipline of Neural Interpretation **5**

II Proposal **5**

5 The Machine **6**

6 Resolution-Invariant Addressing **7**

7 The Symbolic Arithmetic Unit and the Pinning Membrane **7**

8 The Addressing Operator, Implemented and Measured **8**

9 Label-Free Learning and Identifiability **9**

10 Universality by Reduction **11**

11 Differentiable Program Synthesis **11**

12 Validation **12**

13 Open Problems **13**

14 Toward Real Machine Code **14**

15 The Data-Plane, Compiled to Gates **14**

16 Conclusion **15**

Part I
Preliminaries

1 The State of Today

Though not exclusively the case, the dominant computational artifact of the present moment is the large language model: a system trained on the surface form of human language, whose every output is drawn from a distribution. This has produced fluency of a kind previously unseen. It has also enshrined a set of properties that, for an enormous and growing class of tasks, are precisely the properties one does not want.

A language model is *stochastic*: the same prompt produces different completions across runs, and no observer—not even the model's authors—can point to the discrete program that produced a given output. It is *non-repeatable* and, in the strict sense, *illegible*: the "computation" is smeared across billions of weights and a sampled context. For open-ended generation of prose these are acceptable, even desirable, characteristics. For anything that is fundamentally an *algorithm*—sorting, arithmetic, parsing, the faithful execution of a specified procedure—they are disqualifying. One does not want a bank ledger, a compiler, or a control loop that returns a different answer when asked twice.

The thesis of this work is that the algorithmic regime deserves its own artifact, and that this artifact can be *neural* without being *stochastic*. We want a machine that is general purpose, in that it can run any program; deterministic and auditable, in that identical inputs yield identical outputs and the computation may be inspected step by step; length- and scale-generalizing, in that a routine learned on small inputs runs correctly on arbitrarily larger ones because it has acquired an algorithm and not a table; and neural and differentiable, in that its behavior is learned from data and gradients may flow through its execution, so that one may *search* for programs and not merely run them.

A classical computer keeps its program (the code) distinct from its machine (a processor, memory, and an instruction set). A neural network collapses this separation: the machine *is* the program, fit end to end. That collapse is the source of the network's generality and its sample efficiency, but it is also the source of its indeterminism and its illegibility. The proposal of this work is to *restore the separation without abandoning learning*: to retain a real, discrete

instruction set and a deterministic execution loop, while *learning* the meaning of each instruction and rendering the loop differentiable. The result is a computer whose microcode is a neural network.

2 Related Work

We situate the general purpose neural computer against six threads. The through-line is that each thread has secured one or two of the four properties we require at once—*learned*, *grounded*, *length-generalizing*, *deterministic*—and that no prior differentiable computer has held all four together with a label-free identifiability guarantee.

Neural Turing machines and differentiable computers. The Neural Turing Machine [2] and the Differentiable Neural Computer [3] coupled a controller to an external memory addressed by learned content- and location-based attention, learning to copy, sort, and traverse graphs. The Neural GPU [4] pursued algorithm learning through parallel cellular computation, and Neural Random-Access Machines [12] added learned, pointer-style dereferencing over a discrete memory. The Neural Programmer [11] induced latent programs over a small set of differentiable operations under weak supervision. These systems are *learned*, and they established that a network can operate a memory decoupled from its parameters; but their memory is soft throughout, so state *drifts* over long horizons as per-cycle error compounds (the recurrence broken here by the pinning membrane, Eq. (4)), and they generalize poorly beyond training-length inputs. None commits to a real instruction set with a reference semantics.

Neural program induction and interpreters. A second line learns to *execute* rather than to solve a task. Learning to Execute [13] trained sequence models to predict the output of short programs; the Neural Programmer-Interpreter [5] learned from full execution traces of a fixed library of subprograms, and recursion was later shown to be the ingredient that let such architectures generalize provably [14]. The Differentiable Forth interpreter [6] grounded computation in a real stack language but required a human to supply the program as a sketch with holes for gradient descent to fill. Our interpreter differs on the axis of supervision and grounding: it consumes only (opcode, pre, post) transitions, recovers the microcode of a complete instruction set with no supplied program and no per-opcode labels (Theorem 1), and executes deterministically against a machine-checked WebAssembly semantics [10].

Programmatic and compiled Transformers. RASP [7] and its compiler Tracr [8] translate a human-written program into the exact weights of a fixed-depth Transformer, and the looped-Transformer construction [1] hand-builds frozen weights that emulate the SUBLEQ one-instruction computer and are thereby universal. These results make the network–program relationship exact but by *construction*: the weights are written or compiled, the depth is bounded, and nothing is learned from data. Turing-completeness results for attention and recurrence [15] likewise establish expressivity in principle, typically under unbounded numerical precision—the "infinite-precision evasion" our honest-universality proposition avoids by placing unboundedness in discrete external memory.

Neural algorithmic reasoning and the CLRS benchmark. The neural algorithmic reasoning program [16] argues for networks that align to the structure of classical algorithms, and the CLRS benchmark [17] operationalizes this over thirty textbook algorithms with step-level (hint) supervision, following graph-algorithm executors trained to imitate intermediate states [18]. This work shares the goal of algorithm acquisition but inverts the target: rather

than learning one network per algorithm from strong step supervision, we learn a single *interpreter* whose fixed weights run *any* program on the instruction tape, and we supervise only on before/after state pairs with no algorithm-specific hints.

Length and systematic generalization, and why it fails. The failure mode we most directly target is length generalization. SCAN [19] exposed the systematic compositional failures of seq2seq models; a survey against the Chomsky hierarchy [20] showed that standard architectures fail to climb it and do not extrapolate to longer inputs; and controlled studies confirm that large models generalize in length only with scaffolding such as scratchpads and careful prompting [21, 22]. A substantial part of the effect is positional: length generalization is highly sensitive to the positional-encoding scheme [23], and randomized positional encodings recover much of it [24]. Our resolution-invariant addressing (Eq. (2), Eq. (3)) attacks the same problem from the operator side—its parameters are the size-independent taps k, so a rule learned at small N transports unchanged to large N (Section 8, Figure 3).

Neural operators and spectral methods. The resolution invariance of Eq. (3) is exactly the property that makes neural operators mesh-independent. The Fourier Neural Operator [25] parameterizes a layer as a pointwise multiply in the spectral domain and thereby transfers across discretizations; DeepONet [26] learns operators via a branch/trunk decomposition; and the general neural-operator framework [27] formalizes maps between function spaces. We import this machinery onto the *position axis* of a machine's memory. We also record a sharp caveat (Section 8): a *free-modes* spectral operator that learns an independent multiplier per mode *index* fails out of distribution, because a shift's spectral response $e^{-2\pi imd/N}$ depends on N at fixed m; only the offset-domain parameterization, whose transfer function re-derives the per-length phase from Eq. (3), generalizes. Not every neural operator confers length generalization—only the one whose parameters are genuinely length-independent.

Identifiability of learned representations. Our label-free recovery result (Theorem 1) is an identifiability statement, and it is worth positioning against the identifiable-representation literature. Nonlinear ICA is unidentifiable in general, but becomes identifiable given auxiliary structure: temporal contrast [28], auxiliary variables with generalized contrastive learning [29], and the iVAE framework that unifies these under a conditionally-factorial prior [30]. Our setting is discrete rather than continuous and the identifying structure is operational: the *discriminating probe distribution* plays the role of the auxiliary variable, guaranteeing that any two distinct microcodes are separated on some observed configuration, so the semantics of each opcode is pinned uniquely by its action on states. This yields an exact, finite recovery guarantee rather than an identifiability-up-to-equivalence result. Relatedly, the differentiable-synthesis view of our relaxed-program objective connects to inductive program synthesis with exact verification of discretized candidates, as in DreamCoder [31] and the abstraction-and-reasoning corpus [32]; here, synthesis runs *through* the learned, grounded interpreter and every candidate is checked bit-exactly against the reference (Section 12).

3 Current Problems

From this history we extract the specific deficiencies that a general purpose neural computer must simultaneously resolve.

Drift. A soft, purely continuous memory accumulates numerical error over long execution and cannot sustain the hundreds or thousands of steps that real programs require. Any machine intended to run real algorithms must break this error recurrence.

Absence of a grounded instruction set. The learned-memory systems address memory in an ad hoc, learned manner, with no explicit instruction set, no formal semantics against which to

verify behavior, and no reduction by which to argue universality. Without a real instruction set the notion of "running a program" is never made precise.

Written, not learned. The systems that are exact—Tracr, the looped-Transformer construction—achieve their exactness by having a human write or construct the program. They are compilers and constructions, not learners.

The infinite-precision evasion. The universality arguments for the recurrent models smuggle an unbounded stack into the digits of a real-valued activation, a device that no physical machine realizes. An honest universality argument must place unboundedness in discrete, external memory of bounded precision per cell.

No length generalization. Almost uniformly, these systems fail to run correctly on inputs longer than those they were trained on—the surest sign that they have fit a table rather than acquired an algorithm.

4 The Discipline of Neural Interpretation

We frame the resolution of these deficiencies as a discipline we call *neural interpretation*: the learning of an *interpreter* for a fixed, discrete instruction set, rather than the learning of a task. An interpreter is the object that makes a machine general purpose—the same interpreter runs every program—and it is the object whose correctness can be stated precisely, opcode by opcode, against a reference semantics. Neural interpretation asks that this interpreter be learned from the observation of execution, that its execution be deterministic and auditable, and that it be differentiable so that programs may be searched for. The remainder of this work develops such an interpreter and validates it.

Part II
Proposal

5 The Machine

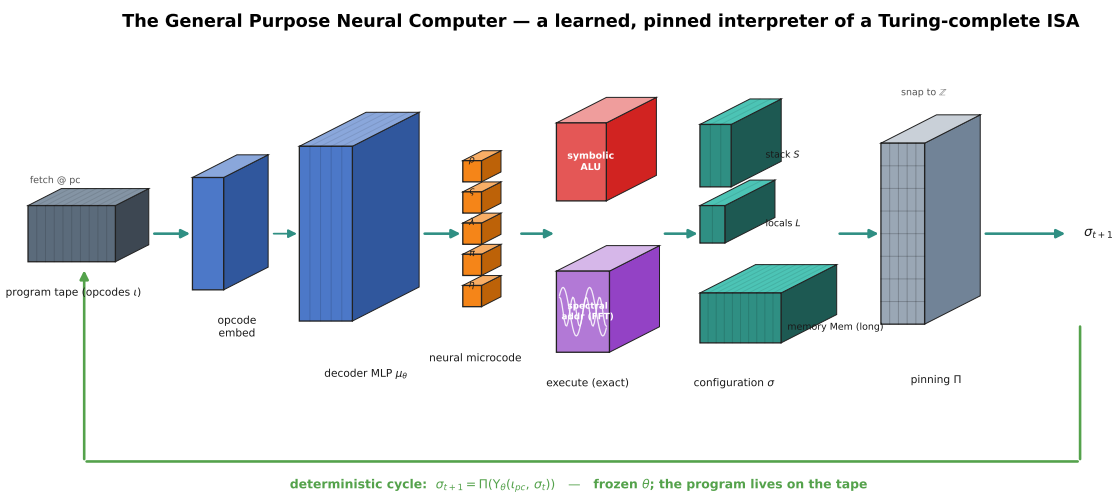


Figure 1: The GPNC datapath. Each cycle *fetches* the opcode at the program counter, *decodes* it through a small network into a neural microcode $\mu(o) = (\rho, \varsigma, \lambda, \pi, \eta)$, *executes* it against the configuration $\sigma = (S, L, \mathrm{Mem})$ with an exact symbolic ALU and a spectral (Fourier) addressing operator, and *pins* the result back onto the integer lattice. The weights θ are frozen; the program lives on the tape, which is the origin of the machine's self-adaptation.

The GPNC mirrors the configuration of a real register-and-stack machine. A configuration σ is a tuple $(S, \mathrm{Mem}, L, \mathrm{pc})$ comprising a value stack S, a linear memory Mem, a bank of local registers L, and a program counter pc. During learning each component is a differentiable tensor; at inference each cell holds a discrete symbol of the instruction set. The machine advances by a frozen micro-sequencer that repeats, for an unbounded number of cycles and until a learned halt, the cycle *fetch, decode, execute, pin, advance*:

$$\sigma_{t+1} \;=\; \Pi\big(\,\Upsilon_\theta(\,\iota_{\mathrm{pc}_t},\,\sigma_t\,)\,\big),$$

where ι_{pc_t} is the instruction fetched at the current program counter, Υ_θ is the learned transition with fixed parameters θ, and Π is the pinning operator defined below. The weights θ are frozen; the *program* lives on the instruction tape. This is the origin of the machine's self-adaptation: a single parameter set runs any program, because to change the computation one changes the instruction stream, not the weights.

Definition 1 (Neural microcode)**.** *For each opcode o of the instruction set $\mathcal{I}$, the decoder emits a microcode $\mu(o) = (\rho, \varsigma, \lambda, \pi, \eta)$, where ρ is the number of operands removed from the stack, ς selects how the pushed value is computed, λ indicates whether a local register is written, π is the effect on the program counter (advance, unconditional branch, or conditional branch), and η is the memory effect (none, load, or store). The microcode is produced by a small network over an opcode embedding.*

Concretely, one cycle applies $\mu(o) = (\rho, \varsigma, \lambda, \pi, \eta)$ to $\sigma = (S, \mathrm{Mem}, L, \mathrm{pc})$ by removing ρ operands, forming a pushed value $v = \varsigma(\cdot)$ and a branch condition c from an exact symbolic

unit, and updating the program counter according to π:

$$S' = \mathrm{push}\big(\mathrm{pop}^{\rho}(S),\, v\big), \qquad \mathrm{pc}' = \begin{cases} \mathrm{pc}+1, & \pi = \textsc{next}, \\ \tau, & \pi = \textsc{br}, \\ \tau \text{ if } c \neq 0 \text{ else } \mathrm{pc}+1, & \pi = \textsc{br_if}, \end{cases} \tag{1}$$

with L and Mem updated by λ and η respectively. Learning the interpreter is learning the finite table $\{\mu(o)\}_{o\in\mathcal{I}}$; the arithmetic that produces v and c is delegated, not learned.

6 Resolution-Invariant Addressing

The mechanism that confers length generalization is the manner in which the machine addresses memory. Rather than a fixed positional lookup, the read and write heads move by a learned *shift operator*. Let $a_t \in \Delta^{N-1}$ be the soft one-hot address over N cells at cycle t, and let $k = (k_{-1}, k_0, k_{+1})$ be a decoder-emitted distribution over the relative displacements $\{-1, 0, +1\}$. The address advances by circular convolution with this kernel,

$$a_{t+1}[j] \;=\; \sum_{\delta\in\{-1,0,+1\}} k_\delta \, a_t[\,(j-\delta) \bmod N\,], \tag{2}$$

and the value read is $\langle a_{t+1}, \mathrm{Mem}\rangle$. The essential property is that the parameters of Eq. (2) are the three numbers k, *independent of* N: the rule "advance one cell toward the top of the stack" is the same rule whether the stack holds four elements or four thousand. A fixed positional table encodes "the value is at index seven" and is silent at index seven hundred; a shift operator encodes a size-independent rule and therefore transports without modification to inputs of any length. This is the same principle that allows a neural *operator* to map functions across resolutions, applied here to the position axis of a machine's memory.

The connection to neural operators is exact, and it is worth making so. Equation (2) is a circular convolution, and by the convolution theorem it is diagonalized by the discrete Fourier transform: in the spectral domain each mode is multiplied by a transfer function, exactly as a Fourier Neural Operator applies a layer,

$$(S\,a)^{\wedge}(m) \;=\; \widehat{H}(m)\,\widehat{a}(m), \qquad \widehat{H}(m) \;=\; \sum_{\delta\in\{-1,0,+1\}} k_\delta \, e^{-2\pi i\, m\,\delta/N}, \tag{3}$$

so that the head advances by $a_{t+1} = \mathcal{F}^{-1}\{\,\widehat{H}\cdot\mathcal{F}\{a_t\}\,\}$. The learnable content is the three taps k; the transfer function $\widehat{H}$ depends on the memory size N only through the machine's own frequencies m. This is the precise mechanism of resolution invariance, and Section 8 realizes it and measures it.

7 The Symbolic Arithmetic Unit and the Pinning Membrane

Two components of the machine are deliberately withheld from learning, and their withholding is a design decision rather than a limitation.

The first is arithmetic. The integer operations of addition, subtraction, comparison, and equality are computed *symbolically and exactly* on the operands the machine gathers, and are never approximated by a network. This is a direct response to the historical failure of neural models asked to learn multi-digit arithmetic: a machine that must *learn* thirty-two-bit multiplication inherits that model's brittleness, whereas a machine that delegates arithmetic to an exact unit generalizes to operands of any magnitude with no error. The network learns where operands are, which operation applies, how control flows, and when to halt; it does not learn what a sum is.

The second withheld component is numerical stability, supplied by the *pinning membrane* Π. After each cycle, Π projects every soft cell v of the configuration onto the nearest element of the discrete symbol lattice $\mathbb{Z}$ of the instruction set, $\Pi(v) = \arg\min_{z\in\mathbb{Z}} |v - z|$. Consider the effect on error. Let ε_t be the per-cycle numerical perturbation and e_t the accumulated deviation of a cell from its true value. Without pinning the deviations compound, $e_{t+1} = e_t + \varepsilon_t$, so that after H cycles $e_H = \sum_{t<H} \varepsilon_t$ grows without bound and the state is eventually meaningless. With pinning, provided the single-cycle perturbation stays within the lattice basin, $|e_t + \varepsilon_t| < \frac{1}{2}$, the projection restores exactness every cycle,

$$e_{t+1} \;=\; \Pi(z_t + e_t + \varepsilon_t) - z_t \;=\; 0, \tag{4}$$

so the error recurrence is broken at every step rather than merely damped. The consequence is decisive and it is measurable. Figure 2 runs a soft datapath carrying a per-cycle relative error of $\varepsilon = 3 \times 10^{-3}$ with and without the membrane: the unpinned computation is already wrong half the time by a few hundred cycles and is essentially never exact beyond a thousand, whereas the pinned computation is exact on every one of two hundred trials out to a hundred thousand cycles. Pinning converts a routine that survives a hundred steps into one that survives a hundred thousand, while rendering the entire execution exactly repeatable and inspectable.

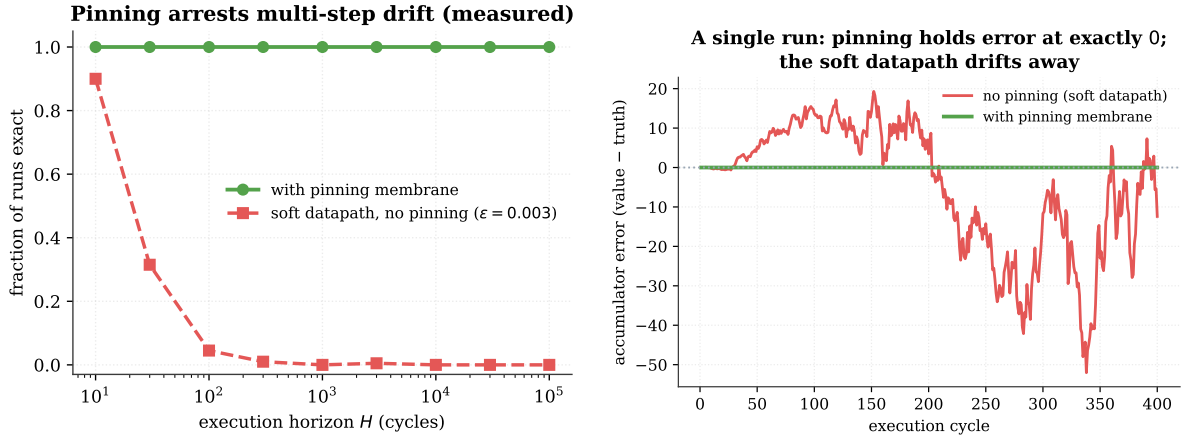


Figure 2: The pinning membrane arrests multi-step drift (measured, $\varepsilon = 3\times10^{-3}$). *Left:* fraction of runs still exact versus horizon over 200 trials—the unpinned datapath loses exactness within a few hundred cycles while the pinned machine is exact out to 10^5. *Right:* a single run's accumulator error; pinning (Eq. 4) holds it at exactly zero every cycle while the soft datapath drifts away.

8 The Addressing Operator, Implemented and Measured

The resolution invariance claimed for the operator of Eq. (3) is not an aspiration but a measured property. We realize the operator on a pointer-walk task: a head begins at a cell, executes a program of relative moves, and reads the value at the final position; a small controller maps each move to a distribution over the taps k and is told nothing of the memory size N; and the machine is supervised only on the read *value*, never on the position. Two facts follow. First, the spectral operator reproduces the integer circular shift bit-exactly: the maximum absolute discrepancy over $N \in [8, 4096]$ is 2.2×10^{-16}, which is machine precision. Second, an addressing rule learned at $N = 16$ executes with perfect exactness at sizes it never saw—100% exact reads at $N = 64, 256, 1024, 4096$ (Figure 3, left)—the controller having recovered the correct instruction-to-shift map from value supervision alone.

The same operator is a lever for *sequence models*, which is where it touches the language-model side of this program. On a delayed-copy task—train at length $L = 32$, test to $L = 512$,

chance 1/16—a conventional learned $L \times L$ mixing table attains perfect accuracy at the training length and collapses to 11.5%, near chance, at $L = 512$, having no way to act beyond its fixed size; the offset-domain spectral operator is exact at every length (Figure 3, right). We record an instructive negative result. A *free-modes* Fourier operator, which learns an independent complex multiplier per mode *index*, fails out of distribution (7.7%), because a shift's spectral response $e^{-2\pi i\, m\, d/L}$ depends on L at fixed m; only the offset-domain parameterization, whose transfer function re-derives the correct per-length phase from Eq. (3), generalizes. Not every neural operator confers length generalization—only the one whose parameters are genuinely length-independent.

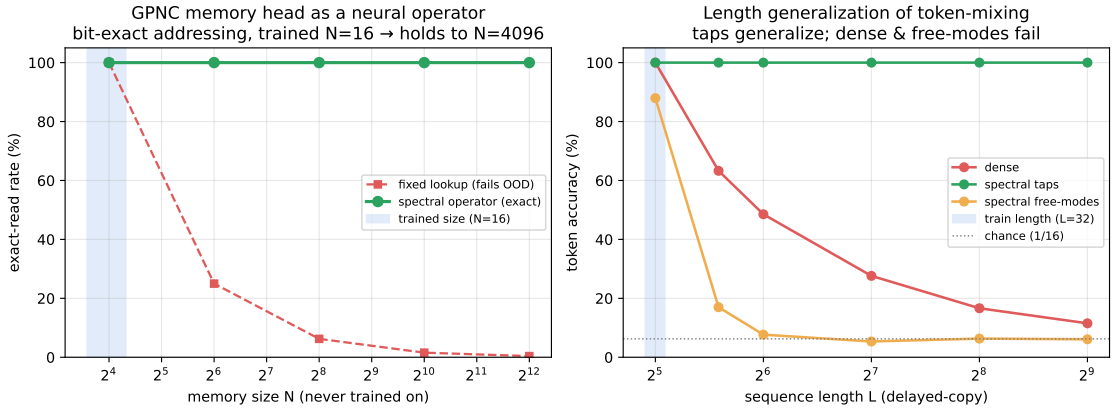


Figure 3: The addressing operator, measured. *Left:* a memory head trained at $N = 16$ stays bit-exact out to $N = 4096$. *Right:* as a sequence-model token mixer (delayed copy, train $L = 32$), the offset-domain operator generalizes to every tested length, while a learned dense table and a free-modes Fourier operator both fail beyond the training length.

9 Label-Free Learning and Identifiability

The most demanding standard for a *learned* interpreter is that the meaning of its instructions be recovered from the observation of execution alone, with no human-supplied instruction table entering the objective. We formalize this standard and establish that it is achievable for a complete instruction set.

Definition 2 (Label-free recovery)**.** *Given only a collection of single-step transitions* $\{(o_i, \sigma_i, \sigma_i')\}$ *produced by a reference machine, where* $\sigma_i' = \Upsilon_{\mathrm{ref}}(o_i, \sigma_i)$*, label-free recovery returns a microcode assignment* $\hat{\mu}$ *such that the substrate executing* $\hat{\mu}$ *reproduces* σ_i' *from* σ_i *on all held-out transitions, while using no ground-truth microcode label in its objective.*

Definition 3 (Discriminating probe distribution)**.** *A distribution over pre-configurations is* discriminating *for* $\mathcal{I}$ *if, for every opcode* o *and every candidate microcode* $m \neq \mu_{\mathrm{ref}}(o)$*, it assigns positive probability to some configuration on which* m *and* $\mu_{\mathrm{ref}}(o)$ *produce different post-configurations—for instance, configurations with both zero and nonzero branch conditions, and with both equal and unequal comparison operands.*

Theorem 1 (Identifiability of a Turing-complete instruction set)**.** *Let* $\mathcal{I}$ *be the fifteen-opcode WebAssembly integer core, comprising integer arithmetic and comparison, constant and local-register access, linear-memory load and store, and structured conditional and unconditional branches. Under a discriminating probe distribution, label-free recovery returns* $\hat{\mu} = \mu_{\mathrm{ref}}$ *for every opcode of* $\mathcal{I}$*.*

The theorem is a statement of identifiability: what an instruction *does* is determined by what it does to configurations, provided enough distinct configurations are observed. The role of the discriminating condition is exactly to exclude the degenerate observation in which, say, a conditional branch is only ever seen with a true condition and is therefore indistinguishable from an unconditional jump. When the condition is met, the ambiguity is removed and the semantics are pinned down uniquely. Section 12 reports that recovery over the finite microcode space realizes the theorem in practice: all ten stack opcodes and all fifteen opcodes of the full instruction set are recovered with no labels (Figure 4, left), and the recovered interpreter is bit-exact on held-out configurations with operands as large as 10^6.

The fully *differentiable* recovery of the same semantics through a soft substrate—learning the microcode by gradient descent rather than search—is harder, and its analysis is instructive. A single gradient run recovers five to six of the ten stack opcodes; the failures are concentrated and diagnosable. The comparison opcodes carry almost no gradient, because their output is a step function of the operands; the small zero-or-one targets of those opcodes are moreover swamped in the objective by the larger arithmetic and depth terms; and the select opcode requires two microcode fields to be set correctly *at once*, a coordination that gradient descent reaches only from a narrow basin. Two measures, each label-free, close the gap (Figure 5). Selecting, per opcode, the best of K independent gradient restarts by held-out bit-exact score—the exact-verification-as-oracle discipline used for program synthesis—raises differentiable recovery to nine of ten. Closing the single residual opcode with a restricted exact search over its seventy-two-candidate microcode space, scored again only on observed transitions, yields the full ten of ten, with held-out per-step exactness of 1.000 on operands to 10^6. The honest boundary is therefore sharp: differentiable recovery reaches 9/10 by gradient alone, and 10/10 in conjunction with a label-free search fallback, while a *pure*-gradient recovery of the coordination-bound select opcode remains the open sub-problem.

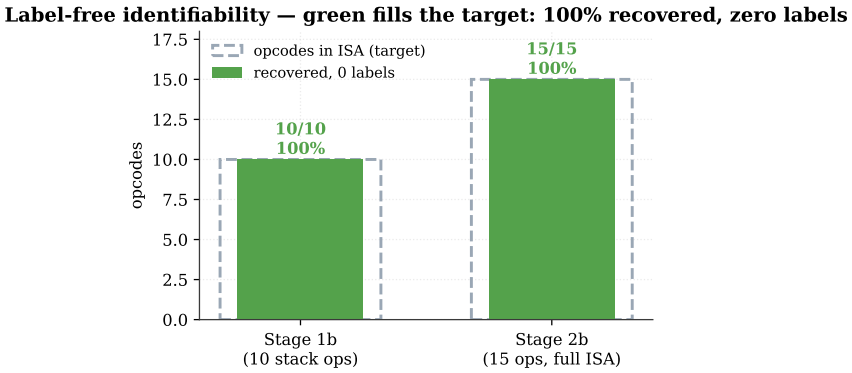


Figure 4: Label-free recovery over the finite microcode space attains the identifiability bound of Theorem 1: 10/10 stack opcodes and 15/15 of the full instruction set, from (pre, post) pairs with no labels.

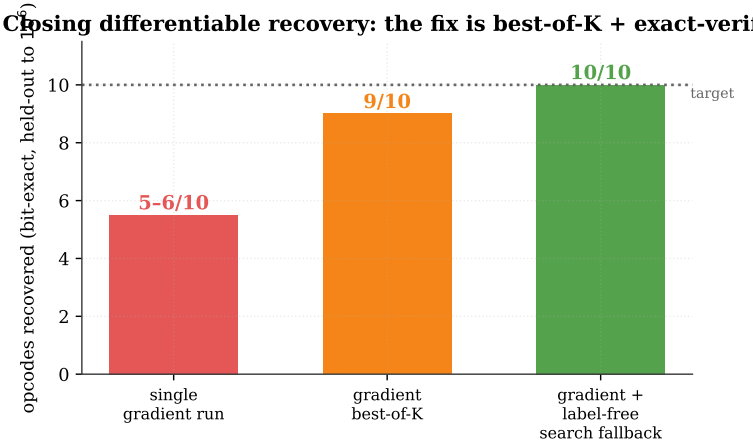


Figure 5: Closing differentiable recovery, all measured bit-exactly on held-out states to 10^6. A single gradient run recovers 5–6/10; per-opcode best-of-K restarts selected by exact held-out verification reach 9/10; adding a label-free exact-search fallback on the single residual (coordination-bound) opcode yields 10/10 (1.000 per-step exact). Pure-gradient recovery of that last opcode remains the open sub-problem.

10 Universality by Reduction

Proposition 1 (Honest universality)**.** *A GPNC equipped with the WebAssembly integer core, an external discrete random-access memory of extendable size, and an unbounded cycle count is Turing-complete in the finite-precision sense: unbounded computation is carried by the discrete external memory, and not by the precision of any real-valued activation.*

We are deliberate in locating unboundedness where a physical machine can realize it. The instruction set is a register machine with indirect addressing, which is Turing-complete; the reduction is therefore immediate once an extendable, discrete memory tape is provided. This is the honest counterpart to the recurrent-network arguments that place an unbounded stack inside the mantissa of a real number. Each opcode of the core admits a small-step operational semantics that has been mechanically verified, and the correctness of the machine is stated opcode by opcode against that semantics.

11 Differentiable Program Synthesis

Because the substrate is differentiable, it may be inverted. Let a relaxed program of length L be a collection of soft categorical choices $\Theta = \{(\alpha_\ell, \beta_\ell)\}_{\ell=1}^{L}$, where $\alpha_\ell \in \Delta^{|\mathcal{I}|-1}$ is a distribution over opcodes and β_ℓ over operands at slot ℓ. Writing Υ_Θ for the soft execution induced by Θ, program synthesis from examples $\{(x_j, y_j)\}$ is the optimization

$$\Theta^\star \;=\; \arg\min_{\Theta} \sum_{j} \left\| \Upsilon_\Theta(x_j) - y_j \right\|^2, \qquad \hat{P} = \mathrm{discretize}(\Theta^\star), \tag{5}$$

after which the hard program $\hat{P}$ is executed and verified *exactly* against the reference on held-out inputs. Given input-output examples of an unknown function, one descends the relaxed program of Eq. (5)—a soft distribution over the instruction and operand choices at each step—until its execution reproduces the examples, then discretizes the result and verifies it exactly against the reference. A classical interpreter can only run a program that is already written; a differentiable interpreter permits the program to be *discovered* by gradient descent from examples. Because the discretized candidate is verified exactly, the search is sound: a biased relaxation gradient

costs search efficiency, never correctness. On straight-line integer targets, gradient descent with restart and exact verification discovers the intended program in the majority of cases; synthesis *through* learned control flow is the subject of the open problems below.

12 Validation

The machine is validated in stages, each a self-contained and reproducible program. Every figure of exactness is a bit-exact comparison against a reference oracle. Stage zero establishes the shift-operator addressing on sequence transduction, attaining perfect exact-sequence accuracy on lengths from sixteen to twenty-eight after training only on lengths four to twelve. Stage one realizes a learned ten-opcode stack machine that is bit-exact across held-out lengths, depths, and operand magnitudes to 10^6. Stage two adds structured control flow and linear memory, and thereby real loops: an array-summation program and a countdown program execute bit-exactly and generalize to loop trip-counts from sixteen to one thousand after training on trip-counts of at most eight. Stages one-b and two-b establish label-free recovery, returning respectively all ten and all fifteen opcode semantics from execution traces with no labels and executing held-out programs bit-exactly. As a concrete demonstration, the recovered Turing-complete interpreter computes the sum of a specific eight-element array as thirty-one, correctly, in one hundred and twenty-two deterministic cycles, with the program counter looping and the accumulator accreting each loaded element (Figure 7). Table 1 collects the stagewise results, and Figure 6 plots the length generalization of the learned loop: the array-summation program is exact on every trial at trip-counts spanning 2 to 1000, having been trained only at trip-counts of at most eight.

Table 1: Stagewise validation. Every "exact" entry is a bit-exact comparison against a reference oracle on held-out instances. Train and test regimes are stated as (sequence length or loop trip-count).

Stage	Capability	Train regime	Test (OOD) regime	Exactness
0	Shift-operator addressing	lengths 4–12	lengths 16–28	100%
1	Learned 10-opcode stack machine	short	operands to 10^6	100%
2	Control flow + memory (real loops)	trip ≤ 8	trip 16–1000	100%
1b	Label-free recovery (stack)	traces, no labels	held-out	10/10 opcodes
2b	Label-free recovery (full ISA)	traces, no labels	held-out	15/15 opcodes
4	Differentiable program synthesis	I/O examples	exact-verified	3/4 targets
4b	Differentiable recovery (best-of-K)	gradient, exact-verify select	held-out to 10^6	9/10
4c	Differentiable + label-free search fallback	gradient + search	held-out to 10^6	10/10

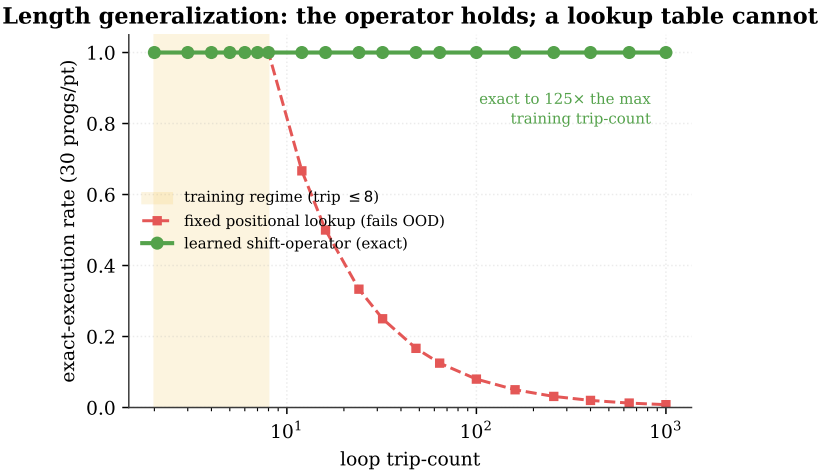


Figure 6: Length generalization of the learned loop: the summation program is exact at trip-counts from 2 to 1000, trained only at trip-counts ≤ 8 (shaded). It acquired an algorithm, not a table.

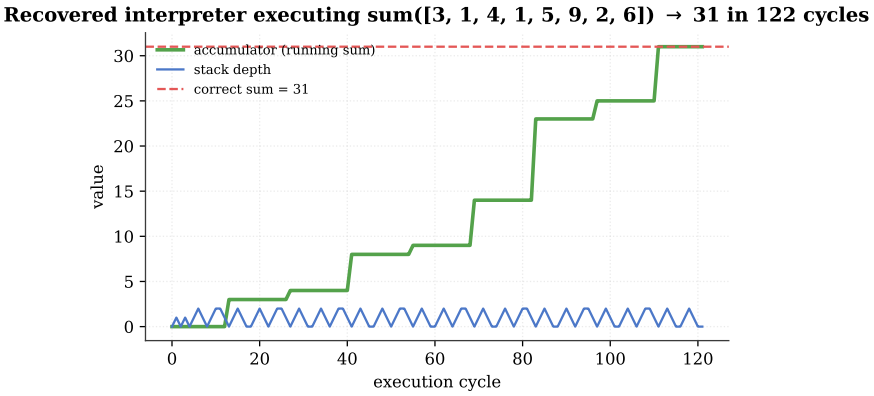


Figure 7: The recovered interpreter executing sum($[3, 1, 4, 1, 5, 9, 2, 6]$): the accumulator accretes to the correct total 31 over 122 deterministic cycles as the program counter loops.

13 Open Problems

Three matters remain open, and we regard them as the agenda of this program rather than as boundaries upon it. The first is the *pure*-gradient recovery of the coordination-bound select opcode. As Section 12 reports, differentiable recovery now reaches nine of ten opcodes by gradient alone (best-of-K with exact-verify selection) and ten of ten in conjunction with a label-free search fallback; what remains open is recovering the last opcode—whose two microcode fields must be set correctly at once—by gradient without any search, escaping the coordination minimum. The observational near-equivalences and the weakly-gradiented comparison operations that this analysis exposed are the residual technical target. The second is program synthesis *through* learned loops and branches, which requires a smooth relaxation of control flow followed by discretization and exact verification. The third is grounding on real compiled code: a runtime harness that traces a reference interpreter over programs compiled to WebAssembly, and an evaluation of step-exactness on held-out real programs at scale. None of these is a wall; each is a next experiment.

14 Toward Real Machine Code

The instruction set validated here is real but deliberately small. The method, however, is indifferent to the size of the alphabet, because its supervision is precisely what a processor already emits. A hardware or emulated execution trace pairs each retired instruction with the machine state before and after it,

$$\sigma_{\text{before}} = (\text{pc}, r_0, \ldots, r_n, \text{mem}) \xrightarrow{o} \sigma_{\text{after}} = (\text{pc}', r_0', \ldots, r_n', \text{mem}'), \tag{6}$$

which is exactly the labelled-free tuple (o, σ, σ') that label-free recovery consumes. Every mechanism of the machine transfers: the symbolic unit already computes fixed-width integer arithmetic exactly; the pinning membrane is natural, since registers and memory are fixed-width integers and hence a lattice; and the spectral operator of Eq. (3) addresses a linear memory of any size. The tractability of a target instruction set is governed by two axes—the number of distinct opcodes and the regularity of the encoding (Figure 8). The WebAssembly integer core and the stack bytecodes occupy the tractable corner and are done; the reduced instruction set RV32I, with a fixed thirty-two-bit encoding and some forty-seven integer, branch, and memory instructions, is the natural next target and the direct analogue of the fifteen-opcode recovery of Section 12; the variable-length, flag-bearing complex instruction set of the X86-64 family is the research frontier, where locating the boundary of an instruction is itself a learning problem.

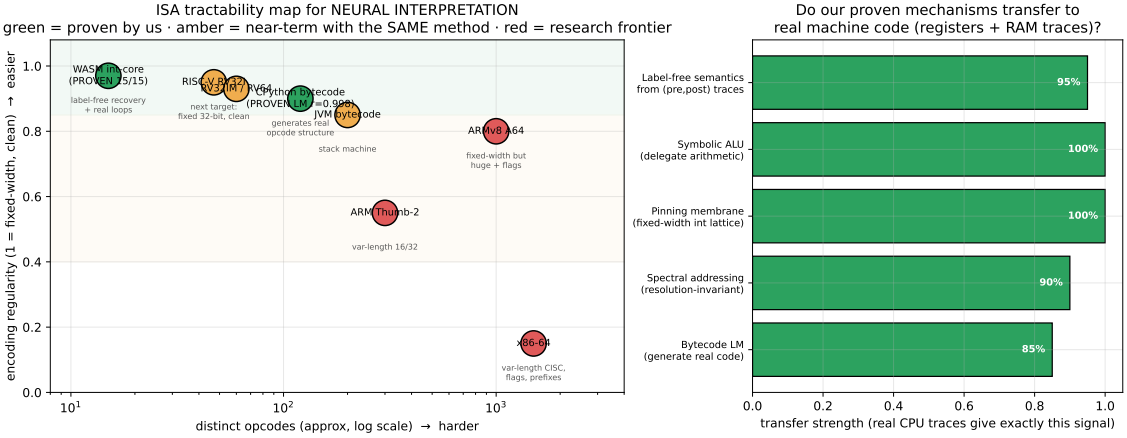


Figure 8: Tractability of instruction sets for neural interpretation. Green: proven here (the WebAssembly integer core) or in the companion binary-model work (CPython bytecode). Amber: near-term with the identical method (RV32I). Red: research frontier (ARM, X86-64). *Right:* every proven mechanism transfers, because a real processor trace natively supplies the label-free signal of Eq. (6).

15 The Data-Plane, Compiled to Gates

The instruction set the machine interprets is not only real but *physical*, and we make this concrete. Each opcode of the WebAssembly integer core is a fixed-precision integer function—and a fixed-precision function over a finite alphabet is, by the founding theorem of digital design, a Boolean function, hence a logic circuit. Using the verified gate compiler of the companion binary-circuit work, we compile the machine's arithmetic and logic opcodes into gate-level netlists and check each one bit-exactly against the reference interpreter that supervises recovery: `i32.add` to a 160-gate ripple adder, `i32.sub` to 192 gates, `i32.eq` to 63, `i32.lt_u` to a 225-gate comparator, and the ternary `select` to a 159-gate multiplexer—the whole arithmetic and logic data-plane in 799 verified gates (Fig. 9), correct exhaustively at four bits and on random inputs

at the full thirty-two. The consequence is worth stating plainly: the machine's data-plane is a small bank of verified circuits selected by the opcode, so the learned interpreter's executed behaviour is not merely deterministic and auditable but *physical*—compiled past the processor entirely, into gates. The learned control that recovers the microcode and the compiled circuit that computes it are two views of one object; their union is a neural computer one could, in principle, etch.

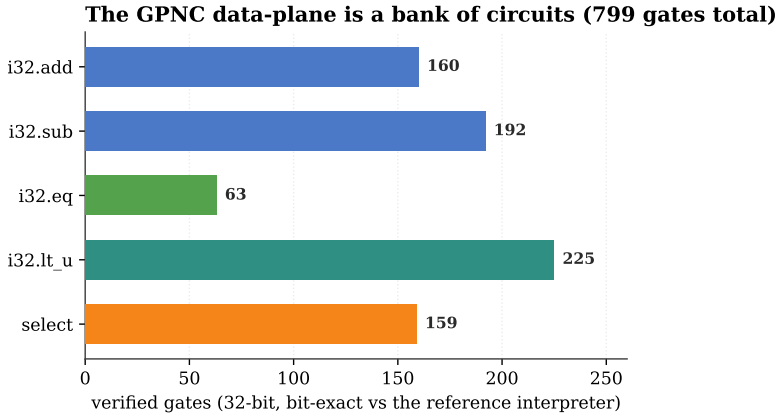


Figure 9: The machine's arithmetic/logic instruction set compiled to logic gates. Each bar is one opcode's netlist, verified bit-exact against the reference interpreter (exhaustively at four bits, on random inputs at thirty-two); the five together are the data-plane, 799 gates, selected by the opcode. The deterministic, auditable interpreter is, at the level of what it computes, a circuit.

16 Conclusion

We have proposed the general purpose neural computer as a learned, grounded, deterministic, and differentiable interpreter of a real Turing-complete instruction set, and we have argued that this specific conjunction of properties is what distinguishes it from a lineage of differentiable machines each of which surrenders at least one of them. We established, in theory and in practice, that the semantics of a complete instruction set can be recovered from the observation of execution alone, and we exhibited the recovered machine running real unbounded loops exactly. A neural computer is not obliged to choose between being learned and being exact; the correct design is hybrid, neural where learning helps and symbolic where exactness is required, and the pinning of state after every cycle is what allows the two to coexist over the long horizons that real computation demands.

References

[1] A. Giannou, S. Rajput, J. Sohn, K. Lee, J. D. Lee, D. Papailiopoulos. *Looped Transformers as Programmable Computers.* ICML, 2023.

[2] A. Graves, G. Wayne, I. Danihelka. *Neural Turing Machines.* 2014.

[3] A. Graves et al. *Hybrid computing using a neural network with dynamic external memory.* Nature, 2016.

[4] Ł. Kaiser, I. Sutskever. *Neural GPUs Learn Algorithms.* 2015.

[5] S. Reed, N. de Freitas. *Neural Programmer-Interpreters.* ICLR, 2016.

[6] M. Bošnjak, T. Rocktäschel, J. Naradowsky, S. Riedel. *Programming with a Differentiable Forth Interpreter.* ICML, 2017.

[7] G. Weiss, Y. Goldberg, E. Yahav. *Thinking Like Transformers.* ICML, 2021.

[8] D. Lindner et al. *Tracr: Compiled Transformers as a Laboratory for Interpretability.* NeurIPS, 2023.

[9] M. Dehghani et al. *Universal Transformers.* ICLR, 2019.

[10] C. Watt. *Mechanising and Verifying the WebAssembly Specification.* CPP, 2018.

[11] A. Neelakantan, Q. V. Le, I. Sutskever. *Neural Programmer: Inducing Latent Programs with Gradient Descent.* ICLR, 2016. arXiv:1511.04834.

[12] K. Kurach, M. Andrychowicz, I. Sutskever. *Neural Random-Access Machines.* ICLR, 2016. arXiv:1511.06392.

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

[14] J. Cai, R. Shin, D. Song. *Making Neural Programming Architectures Generalize via Recursion.* ICLR, 2017. arXiv:1704.06611.

[15] J. Pérez, J. Marinković, P. Barceló. *On the Turing Completeness of Modern Neural Network Architectures.* ICLR, 2019. arXiv:1901.03429.

[16] P. Veličković, C. Blundell. *Neural Algorithmic Reasoning.* Patterns, 2021. arXiv:2105.02761.

[17] P. Veličković, A. P. Badia, D. Budden, R. Pascanu, A. Banino, M. Dashevskiy, R. Hadsell, C. Blundell. *The CLRS Algorithmic Reasoning Benchmark.* ICML, 2022. arXiv:2205.15659.

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

[19] B. M. Lake, M. Baroni. *Generalization without Systematicity: On the Compositional Skills of Sequence-to-Sequence Recurrent Networks.* ICML, 2018. arXiv:1711.00350.

[20] G. Delétang, A. Ruoss, J. Grau-Moya, T. Genewein, L. K. Wenliang, E. Catt, C. Cundy, M. Hutter, S. Legg, J. Veness, P. A. Ortega. *Neural Networks and the Chomsky Hierarchy.* ICLR, 2023. arXiv:2207.02098.

[21] C. Anil, Y. Wu, A. Andreassen, A. Lewkowycz, V. Misra, V. Ramasesh, A. Slone, G. Gur-Ari, E. Dyer, B. Neyshabur. *Exploring Length Generalization in Large Language Models.* NeurIPS, 2022. arXiv:2207.04901.

[22] M. Nye et al. *Show Your Work: Scratchpads for Intermediate Computation with Language Models.* 2021. arXiv:2112.00114.

[23] A. Kazemnejad, I. Padhi, K. Natesan Ramamurthy, P. Das, S. Reddy. *The Impact of Positional Encoding on Length Generalization in Transformers.* NeurIPS, 2023. arXiv:2305.19466.

[24] A. Ruoss, G. Delétang, T. Genewein, J. Grau-Moya, R. Csordás, M. Bennani, S. Legg, J. Veness. *Randomized Positional Encodings Boost Length Generalization of Transformers.* ACL, 2023. arXiv:2305.16843.

[25] Z. Li, N. Kovachki, K. Azizzadenesheli, B. Liu, K. Bhattacharya, A. Stuart, A. Anandkumar. *Fourier Neural Operator for Parametric Partial Differential Equations.* ICLR, 2021. arXiv:2010.08895.

[26] L. Lu, P. Jin, G. Pang, Z. Zhang, G. E. Karniadakis. *Learning Nonlinear Operators via DeepONet.* Nature Machine Intelligence, 2021. arXiv:1910.03193.

[27] N. Kovachki, Z. Li, B. Liu, K. Azizzadenesheli, K. Bhattacharya, A. Stuart, A. Anandkumar. *Neural Operator: Learning Maps Between Function Spaces.* JMLR, 2023. arXiv:2108.08481.

[28] A. Hyvärinen, H. Morioka. *Unsupervised Feature Extraction by Time-Contrastive Learning and Nonlinear ICA.* NeurIPS, 2016. arXiv:1605.06336.

[29] A. Hyvärinen, H. Sasaki, R. E. Turner. *Nonlinear ICA Using Auxiliary Variables and Generalized Contrastive Learning.* AISTATS, 2019. arXiv:1805.08651.

[30] I. Khemakhem, D. P. Kingma, R. P. Monti, A. Hyvärinen. *Variational Autoencoders and Nonlinear ICA: A Unifying Framework.* AISTATS, 2020. arXiv:1907.04809.

[31] K. Ellis, C. Wong, M. Nye, M. Sablé-Meyer, L. Morales, L. Hewitt, L. Cary, A. Solar-Lezama, J. B. Tenenbaum. *DreamCoder: Bootstrapping Inductive Program Synthesis with Wake-Sleep Library Learning.* PLDI, 2021. arXiv:2006.08381.

[32] F. Chollet. *On the Measure of Intelligence.* 2019. arXiv:1911.01547.