entropy

MDPI

Article

Opening the AI Black Box: Distilling Machine-Learned Algorithms into Code

Eric J. Michaud [1,2,†], Isaac Liao [3,†], Vedang Lad [3,†], Ziming Liu [1,2,†], Anish Mudide [3], Chloe Loughridge [4], Zifan Carl Guo [3], Tara Rezaei Kheirkhah [3], Mateja Vukelić [3] and Max Tegmark [1,2,*]

[1] Department of Physics, Massachusetts Institute of Technology, Cambridge, MA 02139, USA; ericjm@mit.edu (E.J.M.); zmliu@mit.edu (Z.L.)
[2] Institute for Artificial Intelligence and Fundamental Interactions, Cambridge, MA 02139, USA
[3] Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, USA; iliao@mit.edu (I.L.); vedang@mit.edu (V.L.); amudide@mit.edu (A.M.); carlguo@mit.edu (Z.C.G.); tarark@mit.edu (T.R.K.); mvukelic@mit.edu (M.V.)
[4] Harvard College, Cambridge, MA 02138, USA; cloughridge@college.harvard.edu
* Correspondence: tegmark@mit.edu
† These authors contributed equally to this work.

Abstract: Can we turn AI black boxes into code? Although this mission sounds extremely challenging, we show that it is not entirely impossible by presenting a proof-of-concept method, MIPS, that can synthesize programs based on the automated mechanistic interpretability of neural networks trained to perform the desired task, auto-distilling the learned algorithm into Python code. We test MIPS on a benchmark of 62 algorithmic tasks that can be learned by an RNN and find it highly complementary to GPT-4: MIPS solves 32 of them, including 13 that are not solved by GPT-4 (which also solves 30). MIPS uses an integer autoencoder to convert the RNN into a finite state machine, then applies Boolean or integer symbolic regression to capture the learned algorithm. As opposed to large language models, this program synthesis technique makes no use of (and is therefore not limited by) human training data such as algorithms and code from GitHub. We discuss opportunities and challenges for scaling up this approach to make machine-learned models more interpretable and trustworthy.

Keywords: mechanistic interpretability; program synthesis

Citation: Michaud, E.J.; Liao, I.; Lad, V.; Liu, Z.; Mudide, A.; Loughridge, C.; Guo, Z.C.; Kheirkhah, T.R.; Vukelić, M.; Tegmark, M. Opening the AI Black Box: Distilling Machine-Learned Algorithms into Code. *Entropy* **2024**, *26*, 1046. https://doi.org/10.3390/e26121046

Received: 7 September 2024
Revised: 2 November 2024
Accepted: 22 November 2024
Published: 2 December 2024

1. Introduction

Machine-learned algorithms now outperform traditional human-discovered algorithms on many tasks, from translation to general-purpose verbal reasoning. These learned algorithms tend to be black box neural networks, and we typically lack a full understanding of how they work. This is part of the reason why many leading AI researchers and business leaders have warned that the seemingly unstoppable race toward ever-more-powerful AI could end badly for humanity [1]. Regardless of one's views on this controversial matter, there is broad agreement that improved reliability and trustworthiness are valuable as AI becomes ever more capable [2,3].

Our research question is as follows: How can we ensure that AI models are truly implementing their intended functions? Without formal proof, we can never be certain. We propose transforming AI models into programs that can be formally verified. Yet formal verification, the gold standard for reliability, is widely viewed as unattainable for powerful AI systems because

1. Neural networks seem too messy to formally verify;
2. Mechanistic interpretability converting learned powerful neural network algorithms into formally verifiable code seems too hard and labor-intensive.

Our paper aims to inject optimism into this field by taking a small but nontrivial step toward showing that automatic mechanistic interpretability (AutoMI) is not impossible,

where AutoMI means black box models can be turned into programs automatically without human inspection. Specifically, we present a proof-of-concept method, MIPS (mechanistic-interpretability-based program synthesis), which can distill simple learned algorithms from neural networks into Python code, for small-scale algorithmic tasks. The main goal of this paper is not to present a method that fully solves AutoMI, but to demonstrate progress toward AutoMI with a simple proof of concept. The rest of this paper is organized as follows. After reviewing prior work in Section 2, we present our method in Section 3, test it on a benchmark in Section 4 and summarize our conclusions in Section 5.

2. Related Work

Program synthesis is a venerable field dating back to Alonzo Church in 1957; Zhou and Ding [4] and Odena et al. [5] provide recent reviews of the field. Large language models (LLMs) have become increasingly good at writing code based on verbal problem descriptions or auto-complete. We instead study the common alternative problem setting known as "programming by example" (PBE), where the desired program is specified by giving examples of input–output pairs [6]. The aforementioned papers review a wide variety of program synthesis methods, many of which involve some form of search over a space of possible programs. LLMs that synthesize code directly have recently become quite competitive with such search-based approaches [7]. Our work provides an alternative search-free approach where the program learning happens during neural network training rather than execution.

Our work builds on the recent progress in mechanistic interpretability (MI) of neural networks [8–11]. Much MI work has tried to understand how neural networks represent various types of information, e.g., geographic information [12–14], truth [15,16] and the state of board games [17–19]. Another major MI thrust has been to understand how neural networks perform algorithmic tasks, e.g., modular arithmetic [20–23] and other group operations [24], greater than [25], and greatest common divisor [26]. The key step of mechanistic interpretability is to look into discovering structures in hidden representations. In this paper, we manage to discover bit representations, integer representations, and clusters (finite state machines).

Whereas Lindner et al. [27] automatically convert traditional code into a neural network, we aim to do the opposite, as was also recently demonstrated in Friedman et al. [28]. One direction in automating mechanistic interpretability uses LLMs to label internal units of neural networks such as neurons [29] and features discovered by sparse autoencoders [30,31]. Another recent effort in automating MI involves automatically identifying which internal units causally influence each other and the network output for a given set of inputs, [32–34]. However, these methods do not automatically generate pseudocode or give a description of *how* the states of downstream units are computed from upstream units, which we aim to do in this work.

In this work, we focus on automating mechanistic interpretability for recurrent neural networks (RNNs), building on the rich existing literature on interpreting RNN internals [35,36] and on extracting finite state machines from trained RNNs [36–41]. In our work, we seek exceptionally simple descriptions of RNNs by factoring network hidden states into discrete variables and representing state transitions with symbolic formulae.

3. MIPS, Our Program Synthesis Algorithm

As summarized in Figure 1, MIPS involves the following key steps.

1. Neural network training;
2. Neural network simplification;
3. Finite state machine extraction;
4. Symbolic regression.

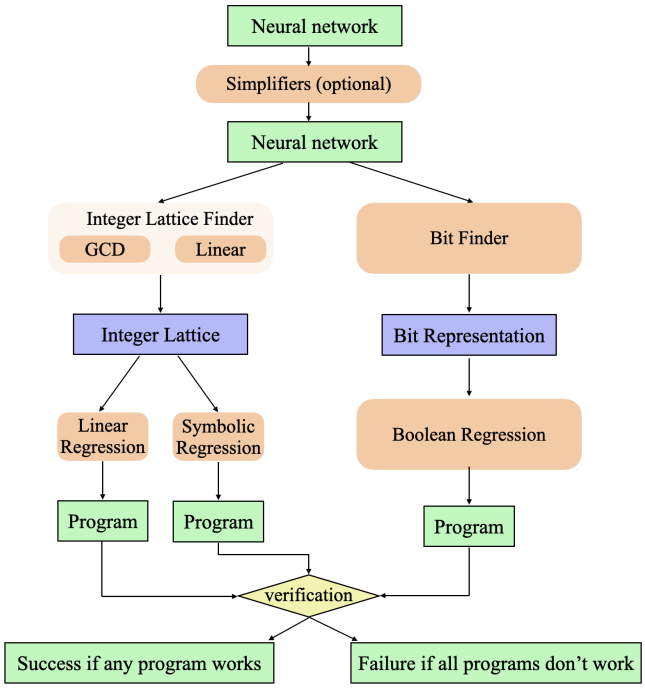

Figure 1. The pipeline of our program synthesis method. MIPS relies on discovering integer representations and bit representations of hidden states, which enable regression methods to figure out the exact symbolic relations between input, hidden, and output states.

Step 1 is to train a black box neural network to learn an algorithm that performs the desired task. In this paper, we use a recurrent neural network (RNN) of the general form

$$\mathbf{h}_i = f(\mathbf{h}_{i-1}, \mathbf{x}_i), \quad (1)$$

$$\mathbf{y}_i = g(\mathbf{h}_i), \quad (2)$$

that maps input vectors $\mathbf{x}_i$ into output vectors $\mathbf{y}_i$ via hidden states $\mathbf{h}_i$. The RNN is defined by the two functions f and g, which are implemented as feed-forward neural networks (MLPs) to allow more model expressivity than a vanilla RNN. The techniques described below can also be applied to more general neural network architectures.

Step 2 attempts to automatically simplify the learned neural network without reducing its accuracy. Steps 3 and 4 automatically distill this simplified learned algorithm into Python code. When the training data are discrete (consisting of, say, text tokens, integers, or pixel colors), the neural network will be a *finite state machine*: the activation vectors for each of its neuron layers define finite sets and the entire working of the network can be defined by look-up tables specifying the update rules for each layer. For our RNN, this means that the space of hidden states $\mathbf{h}$ is discrete, so the functions f and g can be defined by lookup tables. As we will see below, the number of hidden states that MIPS needs to keep track of can often be greatly reduced by clustering them, corresponding to learned representations. After this, the geometry of the cluster centers in the hidden space often reveals that they form either an incomplete multidimensional lattice whose points represent integer tuples, or a set whose cardinality is a power of two, whose points represent Boolean tuples. In both of these cases, the aforementioned lookup tables simply specify integer or Boolean functions, which MIPS attempts to discover via symbolic regression. Below, we present an *integer autoencoder* and a *Boolean autoencoder* to discover such integer/Boolean representations from arbitrary point sets.

We will now describe each of the three steps of MIPS in greater detail.

3.1. AutoML Optimizing for Simplicity

We wish to find the *simplest* RNN that can learn our task, to facilitate subsequent discovery of the algorithm that it has learned. We therefore implement an AutoML-style neural architecture search that tries to minimize network size while achieving perfect test accuracy. This search space is defined by a vector $\mathbf{p}$ of five main architecture hyperparameters: the five integers $\mathbf{p} = (n, w_f, d_f, w_g, d_g)$ corresponding to the dimensionality of hidden state $\mathbf{h}$, the width and depth of the f-network, and the width and depth of the g-network, respectively. Both the f- and g-networks have a linear final layer and ReLU activation functions for all previous layers. The hidden state $\mathbf{h}_0$ is initialized to zero.

To define the parameter search space, we define ranges for each parameter. For all tasks, we use $n \in \{1, 2, \ldots, 128\}$, $w_f \in \{1, 2, \ldots, 256\}$, $d_f \in \{1, 2, 3\}$, $w_g \in \{1, 2, \ldots, 256\}$ and $d_g \in \{1, 2, 3\}$, so the total search space consists of $128 \times 256 \times 3 \times 256 \times 3 = 75{,}497{,}472$ hyperparameter vectors $\mathbf{p}_i$. We order this search space by imposing a strict ordering on the importance of minimizing each hyperparameter –lower d_g is strictly more important than lower d_f, which is strictly more important than lower n, which is strictly more important than lower w_g, which is strictly more important than lower w_f. We aim to find the hyperparameter vector (integer 5-tuple) p_i in the search space that has the lowest rank i under this ordering.

We search the space in the following simple manner. We first start at index $i = 65{,}536$, which corresponds to parameters $(1, 1, 2, 1, 1)$. For each parameter tuple, we train networks using five different seeds. We use the loss function $\ell(x, y) = \frac{1}{2}\log[1 + (x - y)^2]$, finding that it leads to more stable training than using vanilla MSE loss. We train for either 10,000 or 20,000 steps, depending on the task, using the Adam optimizer, a learning rate of 10^{-3}, and batch size 4096. The test accuracy is evaluated with a batch of 65536 samples. If no networks achieve 100% test accuracy (on any test batch), we increase i by $2^{1/4}$. We proceed in this manner until we find a network where one of the seeds achieves perfect test accuracy or until the full range is exhausted. If we find a working network on this upwards sweep, we then perform a binary search using the interval halving method, starting from the successful i, to find the lowest i where at least one seed achieves perfect test accuracy.

3.2. Auto-Simplification

After finding a minimal neural network architecture that can solve a task, the resulting neural network weights typically seem random and un-interpretable. This is because there exist symmetry transformations of the weights that leave the overall input–output behavior of the neural network unchanged. The random initialization of the network has therefore caused random symmetry transformations to be applied to the weights. In other words, the learned network belongs to an equivalence class of neural networks with identical behavior and performance, corresponding to a submanifold of the parameter space. We exploit these symmetry transformations to simplify the neural network into a *normal form*, which in a sense is the simplest member of its equivalence class. Conversion of objects into a normal/standard form is a common concept in mathematics and physics (for example, conjunctive normal form, wavefunction normalization, reduced row echelon form, and gauge fixing).

Two of our simplification strategies below exploit a symmetry of the RNN hidden state space $\mathbf{h}$. We can always write the MLP g in the form $g(\mathbf{h}) = G(\mathbf{U}\mathbf{h} + \mathbf{c})$ for some function G. So if f is affine, i.e., of the form $f(\mathbf{h}, \mathbf{x}) = \mathbf{W}\mathbf{h} + \mathbf{V}\mathbf{x} + \mathbf{b}$, then the symmetry transformation $\mathbf{W}' \equiv \mathbf{A}\mathbf{W}\mathbf{A}^{-1}$, $\mathbf{V}' = \mathbf{A}\mathbf{V}$, $\mathbf{U}' = \mathbf{U}\mathbf{A}^{-1}$, $\mathbf{h}' \equiv \mathbf{A}\mathbf{h}$, $\mathbf{b}' = \mathbf{A}\mathbf{b}$ keeps the RNN in the same form:

$$\begin{aligned} \mathbf{h}'_i &= \mathbf{A}\mathbf{h}_i = \mathbf{A}\mathbf{W}\mathbf{A}^{-1}\mathbf{A}\mathbf{h}_{i-1} + \mathbf{A}\mathbf{V}\mathbf{x}_i + \mathbf{A}\mathbf{b} \\ &= \mathbf{W}'^{-1}\mathbf{h}'_{i-1} + \mathbf{V}'\mathbf{x}_i + \mathbf{b}', \qquad (3) \\ \mathbf{y}_i &= G(\mathbf{U}\mathbf{h}_i + \mathbf{c}) = G(\mathbf{U}\mathbf{A}^{-1}\mathbf{h}'_i + \mathbf{c}) \\ &= G(\mathbf{U}'\mathbf{h}'_i + \mathbf{c}). \qquad (4) \end{aligned}$$

We think of neural networks as nails, which can be hit by various auto-normalization hammers. Each hammer is an algorithm that applies transformations to the weights to remove degrees of freedom caused by extra symmetries or cleans the neural network up in some other way. In this section, we describe five normalizers we use to simplify our trained networks, termed "Whitening", "Jordan normal form", "Toeplitz", "De-bias", and "Quantization". For every neural network, we always apply this sequence of normalizers in that specific order, for consistency. We describe them below and provide additional details about them in the Appendix D.

1. Whitening: Just as we normalize input data to use for training neural networks, we would like activations in the hidden state space $\mathbf{h}_i$ to be normalized. To ensure normalization in all directions, we feed the training dataset into the RNN, collect all the hidden states, compute the uncentered covariance matrix $\mathbf{C}$, and then apply a whitening transform $\mathbf{h} \mapsto \mathbf{C}^{-1/2}\mathbf{h}$ to the hidden state space so that its new covariance becomes the identity matrix. This operation exists purely to provide better numerical stability to the next step.
2. Jordan normal form: When the function g is affine, we can apply the aforementioned symmetry transformation to try to diagonalize $\mathbf{W}$, so that none of the hidden state dimensions interact with one another. Unfortunately, not all matrices $\mathbf{W}$ can be diagonalized, so we use a generalized alternative: the Jordan normal form, which allows elements of the superdiagonal to be either zero or one. To eliminate complex numbers, we also apply 2×2 unitary transformations to eigenvectors corresponding to conjugate pairs of complex eigenvalues afterward. The aforementioned whitening is now ruined, but it helped make the Jordan normal form calculation more numerically stable.
3. Toeplitz: Once $\mathbf{W}$ is in a Jordan normal form, we divide it up into Jordan blocks and apply upper-triangular Toeplitz transformations to the dimensions belonging to each Jordan block. There is now an additional symmetry, corresponding to multiplying each Jordan block by an upper triangular Toeplitz matrix, and we exploit the Toeplitz matrix that maximally simplifies the aforementioned $\mathbf{V}$-matrix.
4. De-bias: Sometimes $\mathbf{W}$ is not full rank, and $\mathbf{b}$ has a component in the direction of the nullspace. In this case, the component can be removed, and the bias $\mathbf{c}$ can be adjusted to compensate.
5. Quantization: After applying all the previous normalizers, many of the weights may have become close to integers, but not exactly due to machine precision and training imperfections. Sometimes, depending on the task, all of the weights can become integers. We therefore round any weights that are within $\epsilon \equiv 0.01$ of an integer to that integer.

3.3. Boolean and Integer Autoencoders

As mentioned, our goal is to convert a trained recurrent neural network (RNN) into a maximally simple (Python) program that produces equivalent input–output behavior. This means that if the RNN has 100% accuracy for a given dataset, so should the program, with the added benefit of being more interpretable, precise, and verifiable.

Once trained/written, the greatest difference between a neural network and a program implementing the same finite state machine is that the former is fuzzy and continuous, while the latter is precise and discrete. To convert a neural network to a program, some discretization ("defuzzification") process is needed to extract precise information from seemingly noisy representations. Fortunately, mechanistic interpretability research has shown that neural networks tend to learn meaningful, structured knowledge representations for algorithmic tasks [20,21]. Previous interpretability efforts typically involved case-by-case manual inspection and only gained algorithmic understanding at the level of pseudocode at best. We tackle this more ambitious question: Can we create an automated method that distills the learned representation and associated algorithms into an equivalent (Python) program?

Since the tasks in our benchmark involve bits and integers, which are already discrete, the only non-discrete parts in a recurrent neural network are its hidden representations. Here, we show two cases when hidden states can be discretized: they are (1) a bit representation or (2) a (typically incomplete) integer lattice. Generalizing to the mixed case of bits and integers is straightforward. Figure 2 shows all hidden state activation vectors $\mathbf{h}_i$ for all steps with all training examples for two of our tasks. The left panel shows that the 10^4 points $\mathbf{h}_i$ form $2^2 = 4$ tight clusters, which we interpret as representing two bits. The right panel reveals that the points $\mathbf{h}_i$ form an incomplete 2D lattice that we interpret as secretly representing a pair of integers.

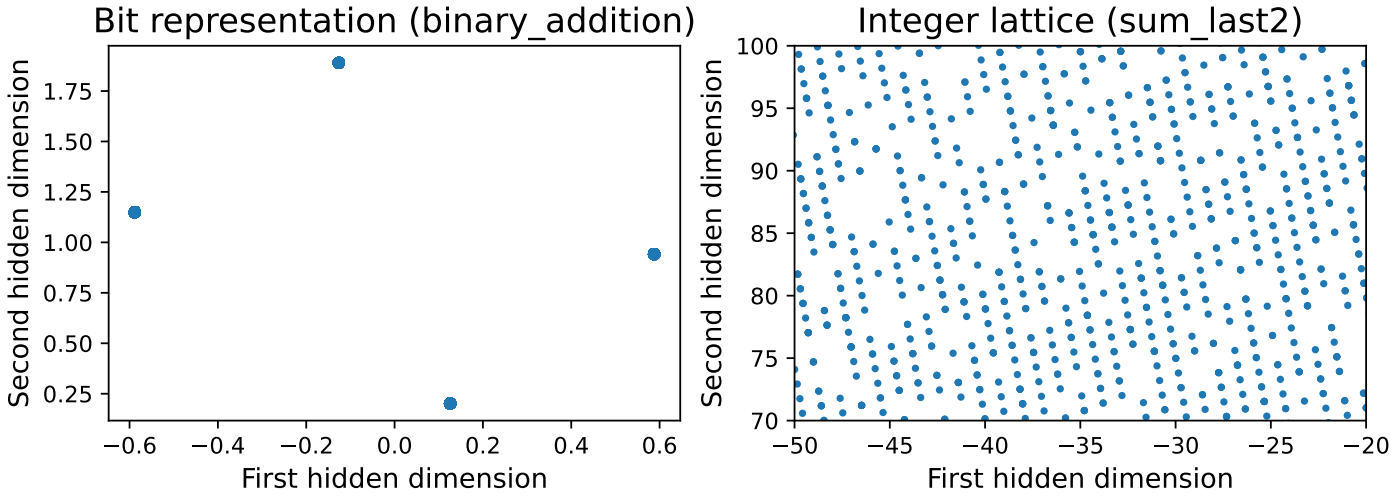

Figure 2. These hidden structures can be turned into discrete representations. Left: bitstring addition, corresponding to 2 bits: the output bit and the carry bit. Right: Sum_Last2 task, 2D lattice corresponding to two integers.

3.3.1. Bit Representations

The hidden states for the two bits in Figure 2 are seen to form a parallelogram. More generally, we find that hidden states encode b bits as 2^b clusters, which in some cases form b-dimensional parallelograms and in other cases look more random. Our algorithm tries all $(2^b)!$ possible assignments of the 2^b clusters to bitstrings of length b and selects the assignment that minimizes the length of the resulting Python program.

3.3.2. Integer Lattice

As seen in Figure 2, the learned representation of an integer lattice tends to be both non-square (deformed by a random affine transformation) and sparse (since not all integer tuplets occur during training). We thus face the following problem: given (possibly sparse) samples of points $\mathbf{h}_i$ from an n-dimensional lattice, how can we reconstruct the integer lattice in the sense that we figure out which integer tuple each lattice point represents? We call the solution an *integer autoencoder* since it compresses any point set into a set of integer tuples from which the original points can be at least approximately recovered as $\mathbf{h}_i = \mathbf{A}\mathbf{k}_i + \mathbf{b}$, where $\mathbf{A}$ is a matrix and $\mathbf{b}$ is a vector that defines the affine transformation and a set of integer vectors $\mathbf{k}_i$.

In the Appendix A, we present a solution that we call the *GCD lattice finder*. For the special case $n = 1$, its core idea is to compute the greatest common denominator of pairwise separations: for example, for the points $\{1.7, 3.2, 6.2, 7.7...\}$, all point separations are divisible by $A = 1.5$, from which one infers that $b = 0.2$ and the lattice can be rewritten as $1.5 \times \{1, 2, 4, 5\} + 0.2$. For multidimensional lattices, our algorithm uses the GCD of ratios of generalized cell volumes to infer the directions and lengths of the lattice vectors that form the columns of $\mathbf{A}$.

For the special case where the MLP defining the function f is affine or can be accurately approximated as affine, we use a simpler method we term the *Linear lattice finder*, also described in Appendix B. Here, the idea is to exploit the fact that the lattice is simply an affine transformation of a regular integer lattice (the input data), so we can simply "read off" the desired lattice basis vectors from this affine transformation.

3.3.3. Symbolic Regression

Once the hidden states $\mathbf{h}_i$ have been successfully mapped to Boolean or integer tuples as described above, the functions f and g that specify the learned RNN can be re-expressed as lookup tables, showing their Boolean/integer output tuple for each Boolean/integer input tuple. All that remains is now symbolic regression, i.e., discovering the simplest possible symbolic formulae that define f and g.

3.3.4. Boolean Regression

In the case where a function maps bits to a bit, our algorithm determines the following set of correct Boolean formulae and then returns the shortest one. The first candidate formula is the function written in disjunctive normal form, which is always possible. If the Boolean function is symmetric, i.e., invariant under all permutations of its arguments, then we also write it as an integer function of its bit sum.

3.3.5. Integer Regression

In the case when a function maps integers to an integer, we try the following two methods:

1. If the function is linear, then we perform simple linear regression, round the resulting coefficients to integers, and simplify, e.g., multiplications by 0 and 1.
2. Otherwise, we use the brute force symbolic solver from *AI Feynman* [42], including the six unary operations $\{>, <, \sim, H, D, A\}$ and four binary operations $\{+, -, *, \%\}$, whose meanings are explained in Appendix C, then convert the simplest discovered formula into Python format.

Once symbolic formulas have been separately discovered for each component of the vector-valued functions f and g, we insert them into a template Python program that implements the basic loop over inputs that are inherent in an RNN. We present examples of our auto-generated programs in Figures 3 and 4 and in Appendix G. Most hyperparameters are thresholds. For example, when a lattice has a reconstruction error ϵ below some threshold θ, we claim an integer lattice has been detected. Since we define the reconstruction errors to be dimensionless, it is clear that $\epsilon = 1$ means no structure is detected and $\epsilon = 0$ means perfect integer lattices. We find our results to be fairly robust with respect to θ; e.g., $\theta = 10^{-2} - 10^{-1}$ would yield the same results.

```
1
2  def f(s,t):
3      a = 0;b = 0;
4      ys = []
5      for i in range(10):
6          c = s[i]; d = t[i];
7          next_a = b ^ c ^ d
8          next_b = b+c+d>1
9          a = next_a;b = next_b;
10         y = a
11         ys.append(y)
12     return ys
```

Figure 3. The generated program for the addition of two binary numbers represented as bit sequences. Note that MIPS rediscovers the "ripple adder", where the variable b above is the carry bit.

```python
def f(s):
    a = 198;b = -11;c = -3;d = 483;e = 0;
    ys = []
    for i in range(20):
        x = s[i]
        next_a = -b+c+190
        next_b = b-c-d-e+x+480
        next_c = b-e+8
        next_d = -b+e-x+472
        next_e = a+b-e-187
        a = next_a;b = next_b;c = next_c;d = next_d;e = next_e;
        y = -d+483
        ys.append(y)
    return ys
```

```python
def f(s):
    a = 0;b = 0;c = 0;d = 0;e = 0;
    ys = []
    for i in range(20):
        x = s[i]
        next_a = +x
        next_b = a
        next_c = b
        next_d = c
        next_e = d
        a = next_a;b = next_b;c = next_c;d = next_d;e = next_e;
        y = a+b+c+d+e
        ys.append(y)
    return ys
```

Figure 4. Comparison of code generated from an RNN trained on Sum_Last5, without (**top**) and with (**bottom**) normalizers.

4. Results

We will now test the program synthesis abilities of our MIPS algorithm on a benchmark of algorithmic tasks specified by numerical examples. For comparison, we try the same benchmark on GPT-4 Turbo, which is currently (as of January 2024) described by OpenAI as their latest generation model, with a 128k context window and more capable than the original GPT-4.

4.1. Benchmark

Our benchmark consists of the 62 algorithmic tasks listed in Table 1. They each map one or two integer lists of length 10 or 20 into a new integer list. We refer to integers whose range is limited to $\{0, 1\}$ as bits. We generated this task list manually, attempting to produce a collection of diverse tasks that would in principle be solvable by an RNN. We also focused on tasks whose known algorithms involved majority, minimum, maximum, and absolute value functions because we believed they would be more easily learnable than other nonlinear algorithms due to our choice of the ReLU activation for our RNNs. The benchmark training data and project code are available at https://github.com/ejmichaud/neural-verification (accessed on 26 November 2024). The tasks are described in Table 1, with additional details in Appendix E. The benchmark aims to cover a diverse range of algorithmic tasks. To balance between different families, when a group of tasks is similar (e.g., summing up the last k bits), our convention is to keep (at most) six of them.

Since the focus of our paper is not on whether RNNs can learn algorithms, but on whether learned algorithms can be auto-extracted into Python, we discarded from our benchmark any generated tasks on which our RNN training failed to achieve 100% accuracy.

Our benchmark can never show that MIPS outperforms any large language model (LLM). Because LLMs are typically trained on GitHub, many LLMs can produce Python code for complicated programming tasks that fall outside of the class we study. Instead, the question that our MIPS-LLM comparison addresses is whether MIPS complements LLMs by being able to solve some tasks where an LLM fails.

Table 1. Benchmark results. For tasks with the note "see text", please refer to Appendix E. The last column highlights the MIPS module responsible for each task. BR = boolean regression, LR = linear regression, SR = symbolic regression, and NA means MIPS is not expected to solve this problem. Green means success and red means failure.

Task #	Input Strings	Element Type	Task Description	Task Name	Solved by GPT-4?	Solved by MIPS?	MIPS Module
1	2	bit	Binary addition of two bit strings	Binary_Addition	0	1	BR
2	2	int	Ternary addition of two digit strings	Base_3_Addition	0	0	SR
3	2	int	Base 4 addition of two digit strings	Base_4_Addition	0	0	SR
4	2	int	Base 5 addition of two digit strings	Base_5_Addition	0	0	SR
5	2	int	Base 6 addition of two digit strings	Base_6_Addition	1	0	SR
6	2	int	Base 7 addition of two digit strings	Base_7_Addition	0	0	SR
7	2	bit	Bitwise XOR	Bitwise_Xor	1	1	BR
8	2	bit	Bitwise OR	Bitwise_Or	1	1	BR
9	2	bit	Bitwise AND	Bitwise_And	1	1	BR
10	1	bit	Bitwise NOT	Bitwise_Not	1	1	BR
11	1	bit	Parity of last 2 bits	Parity_Last2	1	1	BR
12	1	bit	Parity of last 3 bits	Parity_Last3	0	1	BR
13	1	bit	Parity of last 4 bits	Parity_Last4	0	0	BR
14	1	bit	Parity of all bits seen so far	Parity_All	0	1	BR
15	1	bit	Parity of number of zeros seen so far	Parity_Zeros	0	1	BR
16	1	int	Cumulative number of even numbers	Evens_Counter	0	0	BR
17	1	int	Cumulative sum	Sum_All	1	1	LR
18	1	int	Sum of last 2 numbers	Sum_Last2	0	1	LR
19	1	int	Sum of last 3 numbers	Sum_Last3	0	1	LR
20	1	int	Sum of last 4 numbers	Sum_Last4	1	1	LR
21	1	int	Sum of last 5 numbers	Sum_Last5	1	1	LR
22	1	int	sum of last 6 numbers	Sum_Last6	1	1	LR
23	1	int	Sum of last 7 numbers	Sum_Last7	1	1	LR
24	1	int	Current number	Current_Number	1	1	LR
25	1	int	Number 1 step back	Prev1	1	1	LR
26	1	int	Number 2 steps back	Prev2	1	1	LR
27	1	int	Number 3 steps back	Prev3	1	1	LR
28	1	int	Number 4 steps back	Prev4	1	1	LR
29	1	int	Number 5 steps back	Prev5	1	1	LR
30	1	int	1 if last two numbers are equal	Previous_Equals_Current	0	1	SR
31	1	int	current − previous	Diff_Last2	0	1	SR
32	1	int	$\lvert$current − previous$\rvert$	Abs_Diff	0	1	SR
33	1	int	$\lvert$current$\rvert$	Abs_Current	1	1	SR
34	1	int	$\lvert$current$\rvert$ − $\lvert$previous$\rvert$	Diff_Abs_Values	1	0	SR
35	1	int	Minimum of numbers seen so far	Min_Seen	1	0	SR
36	1	int	Maximum of integers seen so far	Max_Seen	1	0	SR
37	1	int	integer in 0-1 with highest frequency	Majority_0_1	1	0	SR
38	1	int	Integer in 0-2 with highest frequency	Majority_0_2	0	0	SR
39	1	int	Integer in 0-3 with highest frequency	Majority_0_3	0	0	SR
40	1	int	1 if even, otherwise 0	Evens_Detector	1	0	SR
41	1	int	1 if perfect square, otherwise 0	Perfect_Square_Detector	0	0	SR
42	1	bit	1 if bit string seen so far is a palindrome	Bit_Palindrome	1	0	NA
43	1	bit	1 if parentheses balanced so far, else 0	Balanced_Parenthesis	0	0	SR
44	1	bit	Number of bits seen so far mod 2	Parity_Bits_Mod2	1	0	BR
45	1	bit	1 if last 3 bits alternate	Alternating_Last3	0	0	BR
46	1	bit	1 if last 4 bits alternate	Alternating_Last4	1	0	BR
47	1	bit	bit shift to right (same as prev1)	Bit_Shift_Right	1	1	LR
48	2	bit	Cumulative dot product of bits mod 2	Bit_Dot_Prod_Mod2	0	1	BR
49	1	bit	Binary division by 3 (see text)	Div_3	1	0	SR
50	1	bit	Binary division by 5 (see text)	Div_5	0	0	SR

Table 1. *Cont.*

Task #	Input Strings	Element Type	Task Description	Task Name	Solved by GPT-4?	Solved by MIPS?	MIPS Module
51	1	bit	Binary division by 7 (see text)	Div_7	0	0	SR
52	1	int	Cumulative addition modulo 3	Add_Mod_3	1	1	SR
53	1	int	Cumulative addition modulo 4	Add_Mod_4	0	0	SR
54	1	int	Cumulative addition modulo 5	Add_Mod_5	0	0	SR
55	1	int	Cumulative addition modulo 6	Add_Mod_6	0	0	SR
56	1	int	Cumulative addition modulo 7	Add_Mod_7	0	0	SR
57	1	int	Cumulative addition modulo 8	Add_Mod_8	0	0	SR
58	1	int	1D dithering, 4-bit to 1-bit (see text)	Dithering	1	0	NA
59	1	int	Newton's of - freebody (integer input)	Newton_Freebody	0	1	LR
60	1	int	Newton's law of gravity (see text)	Newton_Gravity	0	1	LR
61	1	int	Newton's law w. spring (see text)	Newton_Spring	0	1	LR
62	2	int	Newton's law w. magnetic field (see text)	Newton_Magnetic	0	0	LR
			Total solved		30	32	

4.2. Evaluation

For both our method and GPT-4 Turbo, a task is considered solved if and only if a Python program is produced that solves the task with 100% accuracy. GPT-4 Turbo is prompted using the "chain-of-thought" approach described below and illustrated in Figure 5.

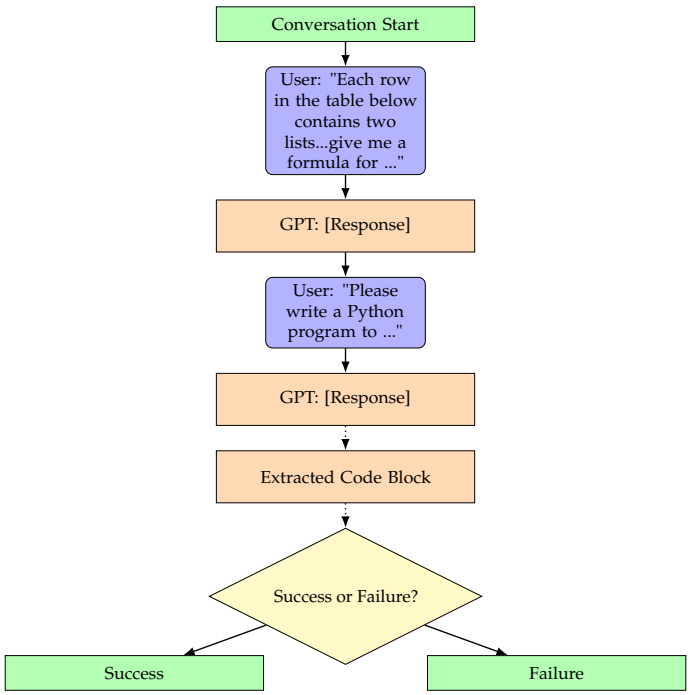

Figure 5. We compare MIPS against program synthesis with the large language model GPT-4 Turbo, prompted with a "chain-of-thought" approach. It begins with the user providing a task, followed by the model's response, and culminates in assessing the success or failure of the generated Python code based on its accuracy in processing the provided lists.

For a given task, the LLM receives two lists of length 10 sourced from the respective RNN training set. The model is instructed to generate a formula that transforms the elements of list "x" (features) into the elements of list "y" (labels). Subsequently, the model is instructed to translate this formula into Python code. The model is specifically asked to use elements of the aforementioned lists as a test case and print "Success" or "Failure"

if the generated function achieves full accuracy on the test case. An external program extracts a fenced markdown codeblock from the output, which is saved to a separate file and executed to determine if it successfully completes the task. To improve the chance of success, this GPT-4 Turbo prompting process is repeated three times, requiring only at least one of them to succeed. We run GPT using default temperature settings.

4.3. Performance

As seen in Table 1, MIPS is highly complementary to GPT-4 Turbo: MIPS solves 32 of our tasks, including 13 that are not solved by ChatGPT-4 (which solves 30).

The AutoML process of Section 3.1 discovers networks of varying task-dependent shape and size. Table A1 shows the parameters $\mathbf{p}$ discovered for each task. Across our 62 tasks, 16 tasks could be solved by a network with hidden dimensions $n = 1$, and the largest n required was 81. For many tasks, there was an interpretable meaning to the shape of the smallest network we discovered. For instance, on tasks where the output is the element occurring k steps earlier in the list, we found $n = k + 1$, since the current element and the previous k elements must be stored for later recall.

We found two main failure modes for MIPS:

1. Noise and non-linearity. The latent space is still close to being a finite state machine, but the non-linearity and/or noise present in an RNN is so dominant that the integer autoencoder fails, e.g., for *Diff_Abs_Values*. Humans can stare at the lookup table and regress the symbolic function with their brains, but since the lookup table is not perfect, i.e., it has the wrong integer in a few examples, MIPS fails to symbolically regress the function. This can probably be mitigated by learning and generalizing from a training subset with a smaller dynamic range.
2. Continuous computation. A key assumption of MIPS is that RNNs are finite-state machines. However, RNNs can also use continuous variables to represent information—the *Majority_0_X* tasks fail for this reason. This can probably be mitigated by identifying and implementing floating-point variables.

Figure 3 shows an example of a MIPS rediscovering the "ripple-carry adder" algorithm. The normalizers significantly simplified some of the resulting programs, as illustrated in Figure 4, and sometimes made the difference between MIPS failing and succeeding. We found that applying a small $L1$ weight regularization sometimes facilitated integer autoencoding by axis-aligning the lattice.

5. Discussion

We have presented MIPS, a novel method for program synthesis based on the automated mechanistic interpretability of neural networks trained to perform the desired task, auto-distilling the learned algorithm into Python code. Its essence is to first train a recurrent neural network to learn a clever finite state machine that performs the task and then automatically figure out how this machine works.

5.1. Findings

We found MIPS to be highly complementary to LLM-based program synthesis with GPT-4 Turbo, with each approach solving many tasks that stumped the other. Please note that our motivation is not to outcompete other program synthesis methods, but instead to provide a proof of principle that fully automated distillation of machine-learned algorithms is not impossible.

Whereas LLM-based methods have the advantage of drawing upon a vast corpus of human training data, MIPS has the advantage of discovering algorithms from scratch without human hints, with the potential to discover entirely new algorithms. As opposed to genetic programming approaches, MIPS leverages the power of deep learning by exploiting gradient information.

Program synthesis aside, our results shed further light on mechanistic interpretability, specifically on how neural networks represent bits and integers. We found that n

integers tend to get encoded linearly in n dimensions, but generically in non-orthogonal directions with an additive offset. This is presumably because there are many more such messy encodings than simple ones, and the messiness can be easily (linearly) decoded. We saw that n bits sometimes get encoded as an n-dimensional parallelogram, but not always——possibly because linear decodability is less helpful when the subsequent bit operations to be performed are nonlinear anyway.

5.2. Outlook

Our work is merely a modest first attempt at mechanistic-interpretability-based program synthesis, and there are many obvious generalizations worth trying in future work, for example,

1. Improvements in training and integer autoencoding (since many of our failed examples failed only just barely);
2. Generalization from RNNs to other architectures such as transformers;
3. Generalization from bits and integers to more general extractable data types such as floating-point numbers and various discrete mathematical structures and knowledge representations;
4. Scaling to tasks requiring much larger neural networks;
5. Automated formal verification of synthesized programs (we perform such verification with Dafny in Appendix G—Formal Verification to show that our MIPS-learned ripple adder correctly adds *any* binary numbers, not merely those in the test set, but such manual work should ideally be fully automated).

LLM-based coding co-pilots are already highly useful for program synthesis tasks based on verbal problem descriptions or auto-complete and will only get better. MIPS instead tackles program synthesis based on test cases alone. This makes it analogous to symbolic regression [42,43], which has already proven useful for various science and engineering applications [44,45] where one wishes to approximate data relationships with symbolic formulae. The MIPS framework generalizes symbolic regression from feed-forward formulae to programs with loops, which are in principle Turing-complete. If this approach can be scaled up, it may enable promising opportunities for making machine-learned algorithms more interpretable, verifiable, and trustworthy.

Author Contributions: Conceptualization M.T.; software E.J.M., I.L., V.L., Z.L., A.M., C.L., Z.C.G., T.R.K. and M.V.; writing: M.T., E.J.M., I.L., V.L., Z.L., A.M. and C.L.; investigation E.J.M., I.L., V.L., Z.L., A.M. and C.L.; supervision: M.T., E.J.M. and Z.L. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by Erik Otto, Jaan Tallinn, the Rothberg Family Fund for Cognitive Science, the NSF Graduate Research Fellowship (Grant No. 2141064), and IAIFI through NSF grant PHY-2019786.

Institutional Review Board Statement: Not applicable

Data Availability Statement: Code for reproducing our experiments can be found at https://github.com/ejmichaud/neural-verification (accessed on 26 November 2024).

Acknowledgments: We thank Wes Gurnee, James Liu, and Armaun Sanayei for helpful conversations and suggestions.

Conflicts of Interest: The authors declare no conflicts of interest.

Appendix A. Lattice Finding Using a Generalized Greatest Common Divisor (Gcd)

Our method often encounters cases where hidden states secretly form an affine transformation of an integer lattice. However, not all lattice points are observed in training samples, so our goal is to recover the hidden integer lattice from sparse observations.

Appendix A.1. Problem Formulation

Suppose we have a set of lattice points in $\mathbb{R}^D$ spanned by D independent basis vectors, $\mathbf{b}_i$ $(i = 1, 2, \cdots, D)$. Each lattice point j has the position

$$\mathbf{x}_j = \sum_{i=1}^{D} a_{ji}\mathbf{b}_i + \mathbf{c}, \tag{A1}$$

where $\mathbf{c}$ is a global translation vector, and the coefficients a_{ji} are integers.

Our problem: given N such data points $(\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_N)$, how can we recover the integer coefficients a_{ji} for each point data point as well as $\mathbf{b}_i$ and $\mathbf{c}$?

Note that even when the whole lattice is given, there are still degrees of freedom for the solution. For example, $\{\mathbf{c} \mapsto \mathbf{c} + \mathbf{b}_i, a_{ji} \mapsto a_{ji} - 1\}$ remains a solution, and $\{\mathbf{b}_i \to \sum_{j=1}^{D} \Lambda_{ij}\mathbf{b}_j\}$ remains a solution if $\mathbf{\Lambda}$ is an integer matrix whose determinant is ± 1. So our success criteria are a s follows: (1) a_{ji} are integers; (2) the discovered bases and the true bases have the same determinant (the volume of a unit cell). Once a set of bases is found, we can simplify them by minimizing their total norms over valid transformations $(\mathbf{\Lambda} \in \mathbb{Z}^{D\times D}, \det(\mathbf{\Lambda}) = \pm 1)$.

Appendix A.2. Regular GCD

As a reminder, given a list of n numbers $\{y_1, y_2, \cdots, y_n\}$, a common divisor d is a number such that for all i, $\frac{y_i}{d}$ is an integer. All common divisors are the set $\{d | y_i/d \in \mathbb{Z},$ and the greatest common divisor (GCD) is the largest number in this set. Because

$$\mathrm{GCD}(y_1, \cdots, y_n) = \mathrm{GCD}(y_1, \mathrm{GCD}(y_2, \mathrm{GCD}(y_3, ...))), \tag{A2}$$

it, without loss of generality, suffices to consider the case $n = 2$. A common algorithm to compute GCD of two numbers is the so-called Euclidean algorithm. We start with two numbers r_0, r_1 and $r_0 > r_1$, which is step 0. For the k^{th} step, we perform division-with-remainder to find the quotient q_k and the remainder r_k so that $r_{k-2} = q_k r_{k-1} + r_k$ with $|r_{k-1}| > |r_k|$ (We are considering a general case where r_0 and r_1 may be negative. Otherwise r_k can always be positive numbers, hence no need to use the absolute function). The algorithm will eventually produce a zero remainder $r_N = 0$, and the other non-zero remainder r_{N-1} is the greatest common divisor. For example, $\mathrm{GCD}(55, 45) = 5$, because

$$\begin{aligned} 55 &= 1 \times 45 + 10, \\ 45 &= 4 \times 10 + 5, \\ 10 &= 2 \times 5 + 0. \end{aligned} \tag{A3}$$

Appendix A.3. Generalized GCD in D Dimensions

Given a list of n vectors $\{\mathbf{y}_1, \mathbf{y}_2, \cdots, \mathbf{y}_n\}$ where $\mathbf{y}_i \in \mathbb{R}^D$, and assuming that these vectors are in the lattice described by Equation (A1), we can, without loss of generality, set $\mathbf{c} = 0$, since we can always redefine the origin. In D dimensions, the primitive of interest is the D-dimensional parallelogram: a line segment for $D = 1$ (one basis vector), a parallelogram for $D = 2$ (two basis vectors), parallelepiped for $D = 3$ (three basis vectors), etc.

One can construct an D-dimensional parallelogram by constructing its basis vectors as a linear integer combination of $\mathbf{y}_j$, i.e.,

$$\mathbf{q}_i = \sum_{j=1}^{n} m_{ij}\mathbf{y}_j, m_{ij} \in \mathbb{Z}, i = 1, 2, \cdots, D. \tag{A4}$$

The goal of D-dimensional GCD is to find a "minimal" parallelogram, such that its volume (which is $\det(\mathbf{q}_1, \mathbf{q}_2, \cdots, \mathbf{q}_D)$) is the GCD of volumes of other possible parallelograms. Once the minimal parallelogram is found (There could be many minimal parallelograms,

but finding one is sufficient), we can also determine $\mathbf{b}_i$ in Equation (A1), since $\mathbf{b}_i$ is exactly $\mathbf{q}_i$! To find the minimal parallelogram, we need two steps: (1) figure out the unit volume; (2) figure out $\mathbf{q}_i (i = 1, 2, \cdots)$ whose volume is the unit volume.

Step 1: Compute unit volume V_0. We first define *representative* parallelograms as one where all $i = 1, 2, \cdots, D$, $\mathbf{m}_i \equiv (m_{i1}, m_{i2}, \cdots, m_{iD})$ are one-hot vectors, i.e., with only one element being 1 and 0 otherwise. It is easy to show that the volume of any parallelogram is a linear integer combination of volumes of representative parallelograms, so in WLOG, we can focus on representative parallelograms. We compute the volumes of all representative parallelograms, which gives a volume array. Since volumes are just scalars, we can obtain the unit volume V_0 by calling the regular GCD of the volume array.

Step 2: Find a minimal parallelogram (whose volume is the unit volume computed in step 1). Recall that in regular GCD, we are dealing with two numbers (scalars). To leverage this in the vector case, we need to create scalars out of vectors and make sure that the vectors share the same linear structure as the scalars so that we can extend division and remainder to vectors. A natural scalar is volume. Now consider two parallelograms P1 and P2, which share $D - 1$ basis vectors $(\mathbf{y}_3, \ldots, \mathbf{y}_{D+1})$, but the last basis vector is different: $\mathbf{y}_1$ for P1 and $\mathbf{y}_2$ for P2. Denote their volume as V_1 and V_2:

$$\begin{aligned} V_1 &= \det(\mathbf{y}_1, \mathbf{y}_3, \mathbf{y}_4, \ldots, \mathbf{y}_D) \\ V_2 &= \det(\mathbf{y}_2, \mathbf{y}_3, \mathbf{y}_4, \ldots, \mathbf{y}_D) \end{aligned} \tag{A5}$$

Since

$$aV_1 + bV_2 = \det(a\mathbf{y}_1 + b\mathbf{y}_2, \mathbf{y}_3, \mathbf{y}_4, \ldots, \mathbf{y}_D), \tag{A6}$$

which shows that (V_1, V_2) and $(\mathbf{y}_1, \mathbf{y}_2)$ share the same linear structure. We can simply apply division and remainder to V_1 and V_2 as in regular GCD:

$$V_1', V_2' = \mathrm{GCD}(V_1, V_2), \tag{A7}$$

whose quotients in all iterations are saved and transferred to $\mathbf{y}_1$ and $\mathbf{y}_2$:

$$\mathbf{y}_1', \mathbf{y}_2' = \mathrm{GCD\_with\_predefined\_quotients}(\mathbf{y}_1, \mathbf{y}_2). \tag{A8}$$

If $V_1 = V_0$ (which is the condition for minimal parallelogram), the algorithm terminates and returns $(\mathbf{y}_1', \mathbf{y}_3, \mathbf{y}_4, \cdots, \mathbf{y}_D)$. If $V_1 > V_0$, we need to repeat step 2 with the new vector list $\{\mathbf{y}_1', \mathbf{y}_3, \cdots, \mathbf{y}_N\}$.

Why can we remove $\mathbf{y}_2'$ for the next iteration? Note that although eventually, $V_1' > 0$ and $V_2' = 0$, typically, $\mathbf{y}_2 \neq 0$. However, since

$$0 = V_2' = \det(\mathbf{y}_2', \mathbf{y}_3, \mathbf{y}_4, \cdots, \mathbf{y}_D), \tag{A9}$$

this means $\mathbf{y}_2'$ is a linear combination of $(\mathbf{y}_3, \cdots, \mathbf{y}_D)$ and hence can be removed from the vector list.

Step 3: Simplification of basis vectors. We want to further simplify basis vectors. For example, the basis vectors obtained in step 2 may have large norms. For example, $D = 2$, the standard integer lattice, has $\mathbf{b}_1 = (1, 0)$ and $\mathbf{b}_2 = (0, 1)$, but there are infinitely many possibilities after step 2, as long as $pt - sq = \pm 1$ for $\mathbf{b}_1 = (p, q)$ and $\mathbf{b}_2 = (s, t)$, e.g., $\mathbf{b}_1 = (3, 5)$ and $\mathbf{b}_2 = (4, 7)$.

To minimize ℓ_2 norms, we choose a basis and project and subtract for other bases. Note that (1) again, we are only allowed to subtract integer times of the chosen basis; (2) the volume of the parallelogram does not change since the project-and-subtract matrix has a determinant of 1 (suppose $\mathbf{b}_i (i = 2, 3, \cdots, D)$ are projected to $\mathbf{b}_1$ and subtracted by multiples of $\mathbf{b}_1$ and p_* represents projection integers):

$$\begin{pmatrix} 1 & p_{2\to 1} & p_{3\to 1} & \cdots & p_{D\to 1} \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{pmatrix} \tag{A10}$$

We do this iteratively until no norm can become shorter via the project-and-subtract procedure. Please see Figure A1 for an illustration of how simplification works for a 2D example.

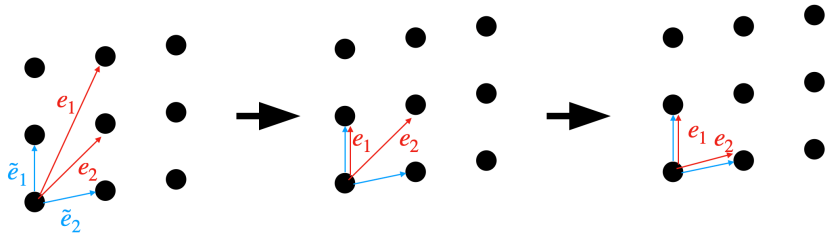

Figure A1. Both red and blue bases form a minimal parallelogram (in terms of cell volume) but one can further simplify red to blue by linear combination (simplicity in the sense of small ℓ_2 norm).

Computation overhead is actually surprisingly small. In typical cases, we only need to call $O(1)$ times of GCD.

Dealing with noise: Usually, the integer lattice in the hidden space is not perfect, i.e., vulnerable to noise. How do we extract integer lattices in a robust way in the presence of noise? Note that the terminating condition for the GCD algorithm is when the remainder is exactly zero–we relax this condition such that the absolute value of the remainder to be smaller than a threshold ϵ_{gcd}. Another issue regarding noise is that noise can accumulate in the GCD iterations, so we hope that GCD can converge in a few steps. To achieve this, we select hidden states in a small region with a data fraction p% of the whole data. Since both ϵ_{gcd} and p depend on data and neural network training, which we do not know a priori, we choose to grid sweep $\epsilon_{\text{gcd}} \in [10^{-3}, 1]$ and $p \in (0.1, 100)$; for each $(\epsilon_{\text{gcd}}, p)$, we obtain an integer lattice and compute its description length. We select the $(\epsilon_{\text{gcd}}, p)$ that gives the lattice with the smallest description length. The description length includes two parts: integer descriptions of hidden states $\log(1 + |Z|^2)$ and residual of reconstruction $\log(1 + (\frac{AZ+b-X}{\epsilon_{\text{dl}}})^2)$ with $\epsilon_{\text{dl}} = 10^{-4}$.

Appendix B. Linear Lattice Finder

Although our RNN can represent general nonlinear functions, in the special case when the RNN actually performs linear functions, program synthesis can be much easier. So if the hidden MLP is linear, we would expect the hidden states to be an integer lattice, because inputs are integer lattices and the mappings are linear. Effectively, the hidden MLP works as a linear function: $\mathbf{h}^{(t)} = W_h \mathbf{h}^{(t-1)} + W_i \mathbf{x}^{(t)}$ (we neglected the bias term since it is not relevant to finding basis vectors of a lattice).

Suppose we have input series $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, ..., \mathbf{x}^{(t)}$; then $\mathbf{h}^{(t)}$ is

$$\mathbf{h}^{(t)} = \sum_{j=1}^{t} W_h^{t-j} W_i \mathbf{x}_j, \tag{A11}$$

Since $\mathbf{x}_j$ values themselves are integer lattices, we could then interpret the following as basis vectors:

$$W_h^{t-j} W_i, j = 1, 2, \cdots, t, \tag{A12}$$

which are not necessarily independent. For example, for the task of summing up the last two numbers, $W_h W_i$ and W_i are non-zero vectors and are independent, while others

$W_h^n W_i \approx 0, n \geq 2$. Then, $W_h W_i$ and W_i are the two basis vectors for the lattice. In general, we measure the norm of all the candidate basis vectors and select the first k vectors with the highest norms, which are exactly basis vectors of the hidden lattice.

Appendix C. Symbolic Regression

The formulation of symbolic regression is that one has a data pair $(\mathbf{x}_i, y_i), i = 1, 2, \ldots, N$ with N data samples. The goal is to find a symbolic formula f such that $y_i = f(\mathbf{x}_i)$. A function is expressed in reverse Polish notation (RPN); for example, $|a| - c$ is expressed as `aAc-` where A stands for the absolute value function. We have three types of variables:

- type-0 operator. We include input variables and constants.
- type-1 operator (takes in one type-0 to produce one type-0). We include operations $\{>, <, \sim, H, D, A\}$. $>$ means $+1$; $<$ means -1; $\sim$ means negating the number; D is dirac delta which outputs 1 only when taking in 0; A is the absolute value function;
- type-2 operator (takes in two type-0 to produce on type-0). We include operations $\{+, *, -, \%\}$. $+$ means the addition of two numbers; $*$ means the multiplication of two numbers; $-$ means the subtraction of two numbers; % is the remainder of one number module the other.

There are only certain types of templates (a string of numbers consisting of 0, 1, 2) that are syntactically correct. For example, 002 is correct while 02 is incorrect. We iterate over all the templates not longer than six symbols, and for each template, we try all the variable combinations. Each variable combination corresponds to a symbolic equation f, for which we can check whether $f(\mathbf{x}_i) = y_i$ for 100 data points. If successful, we terminate the brute force program and return the successful formula. If a brute force search does not find any correct symbolic formula within the computing budget, we will simply return the formula a, to make sure that the program can still be synthesized but simply fail to make correct predictions.

Appendix D. Neural Network Normalization Algorithms

It is well known that neural networks exhibit a large amount of symmetry. That is, there are many transformations that can be applied to networks without affecting the map $y = f(x)$ that they compute. A classic example is to permute the neurons within layers.

In this section, we describe a suite of normalizers that we use to transform our networks into a standard form, such that the algorithms that they learn are easier to interpret. We call our five normalizers "Whitening", "Jordan normal form (JNF)", "Toeplitz", "Debias", and "Quantization".

The main symmetry that we focus on is a linear transformation of the hidden space $\mathbf{h} \mapsto \mathbf{Ah}$, which requires the following changes to f and g:

$$f(\mathbf{h}, \mathbf{x}) = \mathbf{Wh} + \mathbf{Vx} + \mathbf{b} \implies f(\mathbf{h}, \mathbf{x}) = \mathbf{AWA}^{-1}\mathbf{h} + \mathbf{AVx} + \mathbf{Ab}$$
$$g(\mathbf{h}) = G(\mathbf{Uh} + \mathbf{c}) \implies f(\mathbf{h}) = G(\mathbf{UA}^{-1}\mathbf{h} + \mathbf{c})$$

and is implemented by changing the weights:

$$\mathbf{W} \implies \mathbf{AWA}^{-1}$$
$$\mathbf{V} \implies \mathbf{AV}$$
$$\mathbf{b} \implies \mathbf{Ab}$$
$$\mathbf{U} \implies \mathbf{UA}^{-1}$$

For this symmetry, we can apply an arbitrary invertible similarity transformation $\mathbf{A}$ to $\mathbf{W}$, which is the core idea underlying our normalizers, three of which have their own unique ways of constructing $\mathbf{A}$, as we describe in the sections below. Most importantly, one of our normalizers exploits $\mathbf{A}$ to convert the hidden-to-hidden transformation $\mathbf{W}$ into a Jordan normal form, in the case where f is linear. Recent work has shown that large

recurrent networks with linear hidden-to-hidden transformations, such as state space models [46], can perform just as well as transformer-based models in language modeling on a large scale. A major advantage of using linear hidden-to-hidden transformations is the possibility of expressing the hidden space in its eigenbasis. This causes the hidden-to-hidden transformation to become diagonal so that it can be computed more efficiently. In practice, modern state space models assume diagonality and go further to assume the elements on the diagonal are real; they fix the architecture to be this way during training.

By carrying this out, we ignore the possibility of linear hidden-to-hidden transformations that cannot be transformed into a real diagonal matrix via diagonalization. Such examples include rotation matrices (whose eigenvalues may be complex) and shift matrices (whose eigenvalues are degenerate and whose eigenvectors are duplicated). A more general form than the diagonal form is the Jordan normal form, which consists of Jordan blocks along the diagonal, each of which has the form $\lambda \mathbf{I} + \mathbf{S}$ for an eigenvalue λ, and the shift matrix $\mathbf{S}$, with ones on the superdiagonal and zeros elsewhere. The diagonalization is a special case of the Jordan normal form, and all matrices can be transformed to the Jordan normal form. A simple transformation can also be applied to the Jordan normal forms that contain pairs of complex generalized eigenvectors to convert them into real matrices.

For nonlinear hidden-to-hidden transformations, we compute $\mathbf{W}$ as though the nonlinearities have been removed.

Appendix D.1. Whitening Transformation

Similar to normalizing the means and variances of a dataset before feeding it into a machine learning model, a good first preprocessing step is to normalize the distribution of hidden states. We therefore choose to apply a whitening transformation to the hidden space. To compute the transformation, we compute the covariance matrix of hidden activations across the dataset and use the singular value decomposition (SVD) of this covariance matrix to find the closest transformation to the identity that will bring this covariance matrix to the identity. We ignore any directions with covariance less than $\epsilon = 0.1$, which cause more instability when normalized. We then post-apply this transformation to the last linear layer of the hidden-to-hidden transformation and its biases and pre-apply its inverse to the first layers of the hidden-to-hidden and hidden-to-output transformations. This leaves the net behavior of the network unchanged. Other transformations that we use in other normalizers operate in a similar manner by post-applying and pre-applying a transformation and its inverse transformation to the first and last layers that interact with the hidden space.

Appendix D.2. Jordan Normal Form Transformation

Critically, the hidden-to-hidden transformations that we would like to convert into Jordan normal form are imperfect because they are learned. Eigenvectors belonging to each Jordan block must be identical, whereas this will only be approximately true of the learned transformation.

The Jordan normal form of a matrix is unstable; consider a matrix

$$\mathbf{W} = \begin{pmatrix} 0 & 1 \\ \delta & 0 \end{pmatrix}$$

which, when $\delta \neq 0$, can be transformed into a Jordan normal form by

$$\begin{pmatrix} 0 & 1 \\ \delta & 0 \end{pmatrix} = \underbrace{\begin{pmatrix} 1 & 1 \\ \sqrt{\delta} & -\sqrt{\delta} \end{pmatrix}}_{\mathbf{T}} \begin{pmatrix} \sqrt{\delta} & 0 \\ 0 & -\sqrt{\delta} \end{pmatrix} \begin{pmatrix} 1 & 1 \\ \sqrt{\delta} & -\sqrt{\delta} \end{pmatrix}^{-1} \tag{A13}$$

but when $\delta = 0$, is transformed into a Jordan normal form by

$$\begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} = \underbrace{\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}}_{\mathbf{T}} \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}^{-1} \tag{A14}$$

As we can see, all of the matrices in the decomposition are unstable near $\delta = 0$, so the issue of error thresholding is not only numerical but is mathematical in nature as well.

We would like to construct an algorithm that computes the Jordan normal form with an error threshold $|\delta| < \epsilon = 0.7$ within which the algorithm will pick the transformation $\mathbf{T}$ from Equation (A14) instead of from Equation (A13).

Our algorithm first computes the eigenvalues λ_i and then iteratively solves for the generalized eigenvectors that lie in $\ker((\mathbf{W} - \lambda\mathbf{I})^{\mathbf{k}})$ for increasing k. The approximation occurs whenever we compute the kernel (of unknown dimension) of a matrix $\mathbf{X}$; we take the SVD of $\mathbf{X}$ and treat any singular vectors as part of the nullspace if their singular values are lower than the threshold ϵ, calling the result ϵ-$\ker(\mathbf{X})$.

Spaces are always stored in the form of a rectangular matrix $\mathbf{F}$ of orthonormal vectors, and their dimension is always the width of the matrix. We build projections using $\text{proj}(\mathbf{F}) = \mathbf{F}\mathbf{F}^H$, where $\mathbf{F}^H$ denotes the conjugate transpose of $\mathbf{F}$. We compute kernels $\ker(\mathbf{X})$ of known dimension of matrices $\mathbf{X}$ by taking the SVD $\mathbf{X} = \mathbf{V}_1 S \mathbf{V}_2^H$ and taking the last singular vectors in $\mathbf{V}_2^H$. We compute column spaces of projectors of known dimension by taking the top singular vectors of the SVD.

The steps in our algorithm are as follows:

1. Solve for the eigenvalues λ_i of $\mathbf{W}$, and check that eigenvalues that are within ϵ of each other form group, i.e., that $|\lambda_i - \lambda_j| \leq \epsilon$ and $|\lambda_j - \lambda_k| \leq \epsilon$ always imply $|\lambda_k - \lambda_i| \leq \epsilon$. Compute the mean eigenvalue for every group.
2. Solve for the approximate kernels of $\mathbf{W} - \lambda\mathbf{I}$ for each mean eigenvalue λ. We will denote this operation by ϵ-$\ker(\mathbf{W} - \lambda\mathbf{I})$. We represent these kernels by storing the singular vectors whose singular values are lower than ϵ. Also, we construct a "corrected matrix" of $\mathbf{W} - \lambda\mathbf{I}$ for every λ by taking the SVD, discarding the low singular values, and multiplying the pruned decomposition back together again.
3. Solve for successive spaces $\mathbf{F}_k$ of generalized eigenvectors at increasing depths k along the set of Jordan chains with eigenvalue λ for all λ. In other words, find chains of mutually orthogonal vectors that are mapped to zero after exactly k applications of the map $\mathbf{W} - \lambda\mathbf{I}$. We first solve for $\mathbf{F}_0 = \ker(\mathbf{W} - \lambda\mathbf{I})$. Then for $k > 0$, we first solve for $\mathbf{J}_k = \epsilon\text{-}\ker((I - \text{proj}(\mathbf{F}_{k-1}))(\mathbf{W} - \lambda\mathbf{I}))$ and deduce the number of chains which reach depth k from the dimension of $\mathbf{J}_k$, and then solve for $\mathbf{F}_k = \text{col}(\text{proj}(\mathbf{J}_k) - \text{proj}(\mathbf{F}_0))$.
4. Perform a consistency check to verify that the dimensions of $\mathbf{F}_k$ always stay the same or decrease with k. Go through the spaces $\mathbf{F}_k$ in reverse order, and whenever the dimension of $\mathbf{F}_k$ decreases, figure out which direction(s) are not mapped to by applying $\mathbf{W} - \lambda\mathbf{I}$ to $\mathbf{F}_{k+1}$. Carry this out by building a projector $\mathbf{J}$ from mapping vectors representing $\mathbf{F}_{k+1}$ through $\mathbf{W} - \lambda\mathbf{I}$ and taking $\text{col}(\text{proj}(\mathbf{F}_k) - \mathbf{J})$. Solve for the Jordan chain by repeatedly applying $\text{proj}(\mathbf{F}_i)(\mathbf{W}_i - \lambda\mathbf{I})$ for i starting from $k - 1$ and going all the way down to zero.
5. Concatenate all the Jordan chains together to form the transformation matrix $\mathbf{T}$.

The transformation $\mathbf{T}$ consists of generalized eigenvectors that need not be completely real but may also include pairs of generalized eigenvectors that are complex conjugates of each other. Since we do not want the weights of our normalized network to be complex, we also apply a unitary transformation that changes any pair of complex generalized eigenvectors into a pair of real vectors and the resulting block of $\mathbf{W}$ into a

multiple of a rotation matrix. As an example, for a real 2-by-2 matrix $\mathbf{W}$ with complex eigenvectors, we have

$$\begin{aligned}\mathbf{W} &= \mathbf{T}\begin{pmatrix} a+bi & 0 \\ 0 & a-bi \end{pmatrix}\mathbf{T}^{-1} \\ &= \mathbf{T}\mathbf{T}'\begin{pmatrix} a & -b \\ b & a \end{pmatrix}(\mathbf{T}\mathbf{T}')^{-1}, \quad \mathbf{T}' = \frac{\mathbf{1}}{\sqrt{\mathbf{2}}}\begin{pmatrix} 1 & i \\ 1 & -i \end{pmatrix}\end{aligned}$$

Appendix D.3. Toeplitz Transformation

Once $\mathbf{W}$ is in Jordan-normal form, each Jordan block is an upper triangular Toeplitz matrix. Upper-triangular Toeplitz matrices, including Jordan blocks, will always commute with each other because they are all polynomials of the shift matrix (which has ones on the superdiagonal and zeros elsewhere), and therefore, these transformations will leave $\mathbf{W}$ unchanged but will still affect $\mathbf{V}$. We split $\mathbf{V}$ up into parts operated on by each Jordan block and use these Toeplitz transformations to reduce the most numerically stable columns of each block of $\mathbf{V}$ to one-hot vectors. The numerical stability of a column vector is determined by the absolute value of the bottom element of that column vector, since its inverse will become the degenerate eigenvalues of the resulting Toeplitz matrix. If no column has a numerical stability above $\epsilon = 0.0001$, we pick the identity matrix for our Toeplitz transformation.

Appendix D.4. De-Biasing Transformation

Oftentimes, $\mathbf{W}$ is not full-rank, and has a nontrivial nullspace. The bias $\mathbf{b}$ will have some component in the direction of this nullspace, and eliminating this component only affects the behavior of the output network g, and the perturbation cannot carry on to the remainder of the sequence via f. Therefore, we eliminate any such component and compensate accordingly by modifying the bias in the first affine layer of g. We identify the nullspaces by taking an SVD and identifying components whose singular value is less than $\epsilon = 0.1$.

Appendix D.5. Quantization Transformation

After applying all of the previous transformations to the RNN, it is common for many of the weights to become close to zero or some other small integer. Treating this as a sign that the network is attempting to implement discrete operations using integers, we snap any weights and biases that are within a threshold $\epsilon = 0.01$ of an integer to that integer. For certain simple tasks, this sometimes allows the entire network to become quantized.

Appendix E. Supplementary Training Data Details

Here, we present additional details on the benchmark tasks marked "see text" in Table 1:

- Div_3/5/7: This is a long division task for binary numbers. The input is a binary number, and the output is that binary number divided by 3, 5, or 7, respectively. The remainder is discarded. For example, we have 1000011/11 = 0010110 (67/3 = 22). The most significant bits occur first in the sequence.
- Dithering: This is a basic image color quantization task, for 1D images. We map 4-bit images to 1-bit images such that the cumulative sum of pixel brightnesses of both the original and dithered images remains as close as possible.
- Newton_Gravity: This is a Euler forward-propagation technique that follows the equation $F = input - 1, v \mapsto v + F, x \mapsto x + v$.
- Newton_Spring: This is a Euler forward-propagation technique that follows the equation $F = input - x, v \mapsto v + F, x \mapsto x + v$.
- Newton_Magnetic: This is a Euler forward-propagation technique that follows the equation $F_x = input_1 - v_y, F_y = input_2 + v_x, \mathbf{v} \mapsto \mathbf{v} + \mathbf{F}, \mathbf{x} \mapsto \mathbf{x} + \mathbf{v}$.

Appendix F. Architecture Search Results

Table A1 shows the parameters **p** discovered for each task by our architecture search.

Table A1. AutoML architecture search results. All networks achieved 100% accuracy on at least one test batch.

Task #	Task Name	n	w_f	d_f	w_g	d_g	Train Loss	Test Loss
1	Binary_Addition	2	4	2	–	1	0	0
2	Base_3_Addition	2	5	2	–	1	0	0
3	Base_4_Addition	2	5	2	–	1	0	0
4	Base_5_Addition	2	5	2	–	1	0	0
5	Base_6_Addition	2	6	2	–	1	2.45×10^{-9}	2.53×10^{-9}
6	Base_7_Addition	2	10	2	–	1	2.32×10^{-6}	2.31×10^{-6}
7	Bitwise_Xor	1	2	2	–	1	0	0
8	Bitwise_Or	1	–	1	–	1	3.03×10^{-2}	3.03×10^{-2}
9	Bitwise_And	1	–	1	–	1	3.03×10^{-2}	3.03×10^{-2}
10	Bitwise_Not	1	–	1	–	1	0	0
11	Parity_Last2	1	229	2	–	1	1.68×10^{-2}	1.69×10^{-2}
12	Parity_Last3	2	5	2	–	1	1.62×10^{-4}	1.64×10^{-4}
13	Parity_Last4	3	29	2	–	1	3.07×10^{-7}	2.99×10^{-7}
14	Parity_All	1	2	2	–	1	0	0
15	Parity_Zeros	1	2	2	–	1	0	0
16	Evens_Counter	4	73	3	–	1	8.89×10^{-5}	8.88×10^{-5}
17	Sum_All	1	–	1	–	1	6.09×10^{-8}	6.13×10^{-8}
18	Sum_Last2	2	–	1	–	1	0	0
19	Sum_Last3	3	–	1	–	1	6.34×10^{-7}	6.35×10^{-7}
20	Sum_Last4	4	–	1	–	1	2.10×10^{-4}	2.11×10^{-4}
21	Sum_Last5	5	–	1	–	1	8.86×10^{-3}	8.87×10^{-3}
22	Sum_Last6	6	–	1	–	1	1.82×10^{-2}	1.81×10^{-2}
23	Sum_Last7	7	–	1	–	1	3.03×10^{-2}	3.01×10^{-2}
24	Current_Number	1	–	1	–	1	0	0
25	Prev1	2	–	1	–	1	0	0
26	Prev2	3	–	1	–	1	0	0
27	Prev3	4	–	1	–	1	0	0
28	Prev4	5	–	1	–	1	2.04×10^{-7}	2.05×10^{-7}
29	Prev5	6	–	1	–	1	6.00×10^{-5}	5.96×10^{-5}
30	Previous_Equals_Current	2	5	2	–	1	6.72×10^{-5}	6.61×10^{-5}
31	Diff_Last2	2	–	1	–	1	0	0
32	Abs_Diff	2	–	1	2	2	1.84×10^{-7}	1.84×10^{-7}
33	Abs_Current	1	2	2	–	1	4.51×10^{-8}	5.71×10^{-8}
34	Diff_Abs_Values	2	4	2	–	1	3.15×10^{-6}	2.96×10^{-6}
35	Min_Seen	1	2	2	–	1	0	0
36	Max_Seen	1	2	2	–	1	1.46×10^{-12}	0
37	Majority_0_1	1	63	2	–	1	4.03×10^{-3}	4.05×10^{-3}
38	Majority_0_2	4	98	2	–	1	1.64×10^{-4}	1.71×10^{-4}
39	Majority_0_3	21	132	3	–	1	6.94×10^{-5}	6.86×10^{-5}
40	Evens_Detector	5	163	2	–	1	8.18×10^{-4}	8.32×10^{-4}
41	Perfect_Square_Detector	48	100	2	–	1	1.92×10^{-3}	1.97×10^{-3}
42	Bit_Palindrome	18	86	2	–	1	3.81×10^{-5}	3.69×10^{-5}
43	Balanced_Parenthesis	1	16	2	–	1	7.44×10^{-3}	7.10×10^{-3}
44	Parity_Bits_Mod2	1	–	1	–	1	0	0
45	Alternating_Last3	2	3	2	–	1	1.85×10^{-2}	1.87×10^{-2}
46	Alternating_Last4	2	3	2	–	1	8.24×10^{-6}	8.09×10^{-6}
47	Bit_Shift_Right	2	–	1	–	1	0	0
48	Bit_Dot_Prod_Mod2	1	3	2	–	1	0	0
49	Div_3	2	59	2	–	1	6.40×10^{-3}	6.43×10^{-3}
50	Div_5	4	76	2	–	1	1.50×10^{-4}	1.55×10^{-4}
51	Div_7	4	103	2	–	1	6.65×10^{-4}	6.63×10^{-4}
52	Add_Mod_3	1	149	2	–	1	1.02×10^{-3}	1.04×10^{-3}
53	Add_Mod_4	2	33	2	–	1	1.53×10^{-4}	1.44×10^{-4}
54	Add_Mod_5	3	43	2	–	1	1.02×10^{-3}	1.03×10^{-3}
55	Add_Mod_6	4	108	2	–	1	6.14×10^{-4}	6.12×10^{-4}
56	Add_Mod_7	4	199	2	–	1	3.96×10^{-4}	4.07×10^{-4}
57	Add_Mod_8	67	134	2	–	1	8.53×10^{-4}	8.34×10^{-4}
58	Dithering	81	166	2	–	1	7.72×10^{-4}	7.75×10^{-4}
59	Newton_Freebody	2	–	1	–	1	2.61×10^{-7}	2.62×10^{-7}
60	Newton_Gravity	2	–	1	–	1	1.81×10^{-7}	1.87×10^{-7}
61	Newton_Spring	2	–	1	–	1	0	0
62	Newton_Magnetic	4	–	1	–	1	8.59×10^{-5}	8.60×10^{-5}

Appendix G. Generated Programs

This section includes all successfully generated Python programs.

Binary-Addition

```
def f(s,t):
    a = 0;b = 0;
    ys = []
    for i in range(10):
        c = s[i]; d = t[i];
        next_a = b ^ c ^ d
        next_b = b+c+d>1
        a = next_a;b = next_b;
        y = a
        ys.append(y)
    return ys
```

Bitwise-Xor

```
def f(s,t):
    a = 0;
    ys = []
    for i in range(10):
        b = s[i]; c = t[i];
        next_a = b ^ c
        a = next_a;
        y = a
        ys.append(y)
    return ys
```

Bitwise-Or

```
def f(s,t):
    a = 0;
    ys = []
    for i in range(10):
        b = s[i]; c = t[i];
        next_a = b+c>0
        a = next_a;
        y = a
        ys.append(y)
    return ys
```

Bitwise-And

```
def f(s,t):
    a = 0;b = 1;
    ys = []
    for i in range(10):
        c = s[i]; d = t[i];
        next_a = (not a and not b and c and d) or (not a and b and not c
    and d) or (not a and b and c and not d) or (not a and b and c and d) or
     (a and not b and c and d) or (a and b and c and d)
        next_b = c+d==0 or c+d==2
        a = next_a;b = next_b;
        y = a+b>1
        ys.append(y)
    return ys
```

Bitwise-Not

```
def f(s):
    a = 1;
    ys = []
    for i in range(10):
        x = s[i]
        next_a = x
        a = next_a;
        y = -a+1
        ys.append(y)
    return ys
```

Parity-Last2

```
def f(s):
    a = 0;b = 0;
    ys = []
    for i in range(10):
        c = s[i]
        next_a = c
        next_b = a ^ c
        a = next_a;b = next_b;
        y = b
        ys.append(y)
    return ys
```

Parity-Last3

```
def f(s):
    a = 0;b = 0;c = 0;
    ys = []
    for i in range(10):
        d = s[i]
        next_a = d
        next_b = c
        next_c = a
        a = next_a;b = next_b;c = next_c;
        y = a ^ b ^ c
        ys.append(y)
    return ys
```

Parity-All

```
def f(s):
    a = 0;
    ys = []
    for i in range(10):
        b = s[i]
        next_a = a ^ b
        a = next_a;
        y = a
        ys.append(y)
    return ys
```

Parity-Zeros

```
def f(s):
    a = 0;
    ys = []
    for i in range(10):
        b = s[i]
        next_a = a+b==0 or a+b==2
        a = next_a;
        y = a
        ys.append(y)
    return ys
```

Sum-All

```
def f(s):
    a = 884;
    ys = []
    for i in range(10):
        x = s[i]
        next_a = a-x
        a = next_a;
        y = -a+884
        ys.append(y)
    return ys
```

Sum-Last2

```
def f(s):
    a = 0;b = 99;
    ys = []
    for i in range(10):
        x = s[i]
        next_a = -b+x+99
        next_b = -x+99
        a = next_a;b = next_b;
        y = a
        ys.append(y)
    return ys
```

Sum-Last3

```
def f(s):
    a = 0;b = 198;c = 0;
    ys = []
    for i in range(10):
        x = s[i]
        next_a = x
        next_b = -a-x+198
        next_c = -b+198
        a = next_a;b = next_b;c = next_c;
        y = a+c
        ys.append(y)
    return ys
```

Sum-Last4

```
def f(s):
    a = 0;b = 99;c = 0;d = 99;
    ys = []
    for i in range(10):
        x = s[i]
        next_a = c
        next_b = -x+99
        next_c = -b-d+198
        next_d = b
        a = next_a;b = next_b;c = next_c;d = next_d;
        y = a-b-d+198
        ys.append(y)
    return ys
```

Sum-Last5

```
def f(s):
    a = 198;b = -10;c = -2;d = 482;e = 1;
    ys = []
    for i in range(20):
        x = s[i]
        next_a = -b+c+190
        next_b = b-c-d-e+x+480
        next_c = b-e+8
        next_d = -b+e-x+472
        next_e = a+b-e-187
        a = next_a;b = next_b;c = next_c;d = next_d;e = next_e;
        y = -d+483
        ys.append(y)
    return ys
```

Sum-Last6

```
def f(s):
    a = 0;b = 295;c = 99;d = 0;e = 297;f = 99;
    ys = []
    for i in range(20):
        x = s[i]
        next_a = -b+295
        next_b = b-c+f
        next_c = b-c+d-97
        next_d = -f+99
        next_e = -a+297
        next_f = -x+99
        a = next_a;b = next_b;c = next_c;d = next_d;e = next_e;f = next_f;
        y = -b+c-e-f+592
        ys.append(y)
    return ys
```

Sum-Last7

```
def f(s):
    a = 297;b = 198;c = 0;d = 99;e = 0;f = -15;g = 0;
    ys = []
    for i in range(20):
        x = s[i]
        next_a = -a+d-f+g+480
        next_b = a-d
        next_c = d+e-99
        next_d = -c+99
        next_e = -b+198
        next_f = -c+f+x
        next_g = x
        a = next_a;b = next_b;c = next_c;d = next_d;e = next_e;f = next_f;g
     = next_g;
        y = -d+f+114
        ys.append(y)
    return ys
```

Current-Number

```
def f(s):
    a = 99;
    ys = []
    for i in range(10):
        x = s[i]
        next_a = -x+99
        a = next_a;
        y = -a+99
        ys.append(y)
    return ys
```

Prev1

```
def f(s):
    a = 0;b = 99;
    ys = []
    for i in range(10):
        x = s[i]
        next_a = -b+99
        next_b = -x+99
        a = next_a;b = next_b;
        y = a
        ys.append(y)
    return ys
```

Prev2

```
def f(s):
    a = 99;b = 0;c = 0;
    ys = []
    for i in range(10):
        x = s[i]
        next_a = -x+99
        next_b = -a+99
        next_c = b
        a = next_a;b = next_b;c = next_c;
        y = c
        ys.append(y)
    return ys
```

Prev3

```
def f(s):
    a = 0;b = 0;c = 99;d = 99;
    ys = []
    for i in range(10):
        x = s[i]
        next_a = b
        next_b = -c+99
        next_c = d
        next_d = -x+99
        a = next_a;b = next_b;c = next_c;d = next_d;
        y = a
        ys.append(y)
    return ys
```

Prev4

```
def f(s):
    a = 0;b = 99;c = 0;d = 99;e = 0;
    ys = []
    for i in range(10):
        x = s[i]
        next_a = c
        next_b = -a+99
        next_c = -d+99
        next_d = -e+99
        next_e = x
        a = next_a;b = next_b;c = next_c;d = next_d;e = next_e;
        y = -b+99
        ys.append(y)
    return ys
```

Prev5

```
def f(s):
    a = 0;b = 0;c = 99;d = 99;e = 99;f = 99;
    ys = []
    for i in range(20):
        x = s[i]
        next_a = -c+99
        next_b = -d+99
        next_c = -b+99
        next_d = e
        next_e = f
        next_f = -x+99
        a = next_a;b = next_b;c = next_c;d = next_d;e = next_e;f = next_f;
        y = a
        ys.append(y)
    return ys
```

Previous-Equals-Current

```
def f(s):
    a = 0;b = 0;
    ys = []
    for i in range(10):
        c = s[i]
        next_a = delta(c-b)
        next_b = c
        a = next_a;b = next_b;
        y = a
        ys.append(y)
    return ys
```

Diff-Last2

```
def f(s):
    a = 199;b = 100;
    ys = []
    for i in range(10):
        x = s[i]
        next_a = -a-b+x+498
        next_b = a+b-199
        a = next_a;b = next_b;
        y = a-199
        ys.append(y)
    return ys
```

Abs-Diff

```
def f(s):
    a = 100;b = 100;
    ys = []
    for i in range(10):
        c = s[i]
        next_a = b
        next_b = c+100
        a = next_a;b = next_b;
        y = abs(b-a)
        ys.append(y)
    return ys
```

Abs-Current

```
def f(s):
    a = 0;
    ys = []
    for i in range(10):
        b = s[i]
        next_a = abs(b)
        a = next_a;
        y = a
        ys.append(y)
    return ys
```

Bit-Shift-Right

```
def f(s):
    a = 0;b = 1;
    ys = []
    for i in range(10):
        x = s[i]
        next_a = -b+1
        next_b = -x+1
        a = next_a;b = next_b;
        y = a
        ys.append(y)
    return ys
```

Bit-Dot-Prod-Mod2

```
def f(s,t):
    a = 0;
    ys = []
    for i in range(10):
        b = s[i]; c = t[i];
        next_a = (not a and b and c) or (a and not b and not c) or (a and not b and c) or (a and b and not c)
        a = next_a;
        y = a
        ys.append(y)
    return ys
```

Add-Mod-3

```
def f(s):
    a = 0;
    ys = []
    for i in range(10):
        b = s[i]
        next_a = (b+a)%3
        a = next_a;
        y = a
        ys.append(y)
    return ys
```

Newton-Freebody

```
def f(s):
    a = 82;b = 393;
    ys = []
    for i in range(10):
        x = s[i]
        next_a = a-x
        next_b = -a+b+82
        a = next_a;b = next_b;
        y = -a+b-311
        ys.append(y)
    return ys
```

Newton-Gravity

```
def f(s):
    a = 72;b = 513;
    ys = []
    for i in range(10):
        x = s[i]
        next_a = a-x+1
        next_b = -a+b+x+71
        a = next_a;b = next_b;
        y = b-513
        ys.append(y)
    return ys
```

Newton-Spring

```
def f(s):
    a = 64;b = 57;
    ys = []
    for i in range(10):
        x = s[i]
        next_a = a+b-x-57
        next_b = -a+121
        a = next_a;b = next_b;
        y = -a+64
        ys.append(y)
    return ys
```

Formal Verification

The Dafny programming language is designed so that programs can be formally verified for correctness. The desired behavior of a program can be explicitly specified via preconditions, postconditions, and invariants, which are verified via automated theorem proving. These capabilities make Dafny useful in fields where correctness and safety are crucial.

We leverage Dafny's robust verification capabilities to prove the correctness of the bit addition Python program synthesized by MIPS. The bit addition Python program was first converted to Dafny, then annotated with specific assertions, preconditions, and postconditions that defined the expected behavior of the code. Each annotation in the code was then formally verified by Dafny, ensuring that under all possible valid inputs, the code's output would be consistent with the expected behavior. On line 79, we show that the algorithm found by MIPS is indeed equivalent to performing bit addition with length 10 bitvectors in Dafny.

Dafny-Code

```
function ArrayToBv10(arr: array<bool>): bv10 // Converts boolean array to bitvector
  reads arr
  requires arr.Length == 10
{
  ArrayToBv10Helper(arr, arr.Length - 1)
}

function ArrayToBv10Helper(arr: array<bool>, index: nat): bv10
  reads arr
  requires arr.Length == 10
  requires 0 <= index < arr.Length
  decreases index
  ensures forall i :: 0 <= i < index ==> ((ArrayToBv10Helper(arr, i) >> i) & 1) == (if arr[i] then 1 else 0)
{
  if index == 0 then
    (if arr[0] then 1 else 0) as bv10
  else
    var bit: bv10 := if arr[index] then 1 as bv10 else 0 as bv10;
    (bit << index) + ArrayToBv10Helper(arr, index - 1)
}

method ArrayToSequence(arr: array<bool>) returns (res: seq<bool>) // Converts boolean array to boolean sequence
  ensures |res| == arr.Length
  ensures forall k :: 0 <= k < arr.Length ==> res[k] == arr[k]
{
    res := [];
    var i := 0;
    while i < arr.Length
        invariant 0 <= i <= arr.Length
        invariant |res| == i
        invariant forall k :: 0 <= k < i ==> res[k] == arr[k]
    {
        res := res + [arr[i]];
        i := i + 1;
    }
}

function isBitSet(x: bv10, bitIndex: nat): bool
  requires bitIndex < 10
  ensures isBitSet(x, bitIndex) <==> (x & (1 << bitIndex)) != 0
{
  (x & (1 << bitIndex)) != 0
}

function Bv10ToSeq(x: bv10): seq<bool> // Converts bitvector to boolean sequence
  ensures |Bv10ToSeq(x)| == 10
  ensures forall i: nat :: 0 <= i < 10 ==> Bv10ToSeq(x)[i] == isBitSet(x, i)
{
  [isBitSet(x, 0), isBitSet(x, 1), isBitSet(x, 2), isBitSet(x, 3),
   isBitSet(x, 4), isBitSet(x, 5), isBitSet(x, 6), isBitSet(x, 7),
   isBitSet(x, 8), isBitSet(x, 9)]
}

function BoolToInt(a: bool): int {
    if a then 1 else 0
}

function XOR(a: bool, b: bool): bool {
    (a || b) && !(a && b)
}
```

```
function BitAddition(s: array<bool>, t: array<bool>): seq<bool> // Performs traditional bit addition
  reads s
  reads t
  requires s.Length == 10 && t.Length == 10
{
    var a: bv10 := ArrayToBv10(s);
    var b: bv10 := ArrayToBv10(t);
    var c: bv10 := a + b;

    Bv10ToSeq(c)
}

method f(s: array<bool>, t: array<bool>) returns (sresult: seq<bool>) // Generated program for bit addition
  requires s.Length == 10 && t.Length == 10
  ensures |sresult| == 10
  ensures forall i :: 0 <= i && i < |sresult| ==> sresult[i] == ((s[i] != t[i]) != (i > 0 && ((s[i-1] || t[i-1]) && !(sresult[i-1] && (s[i-1] != t[i-1])))))
  ensures BitAddition(s, t) == sresult // Verification of correctness
{
  var a: bool := false;
  var b: bool := false;
  var result: array<bool> := new bool[10];
  var i: int := 0;

  while i < result.Length
    invariant 0 <= i <= result.Length
    invariant forall j :: 0 <= j < i ==> result[j] == false
  {
        result[i] := false;
        i := i + 1;
  }

  i := 0;

  assert forall j :: 0 <= j < result.Length ==> result[j] == false;

  while i < result.Length
    invariant 0 <= i <= result.Length
    invariant b == (i > 0 && ((s[i-1] || t[i-1]) && !(result[i-1] && (s[i-1] != t[i-1]))))
    invariant forall j :: 0 <= j < i ==> result[j] == ((s[j] != t[j]) != (j > 0 && ((s[j-1] || t[j-1]) && !(result[j-1] && (s[j-1] != t[j-1])))))
  {
    assert b == (i > 0 && ((s[i-1] || t[i-1]) && !(result[i-1] && (s[i-1] != t[i-1]))));

    result[i] := XOR(b, XOR(s[i], t[i]));
    b := BoolToInt(b) + BoolToInt(s[i]) + BoolToInt(t[i]) > 1;
    assert b == ((s[i] || t[i]) && !(result[i] && (s[i] != t[i])));

    i := i + 1;
  }

  sresult := ArrayToSequence(result);
}
```

References

1. Center for AI Safety. Statement on AI Risk. 2023. Available online: https://www.safe.ai/work/statement-on-ai-risk (accessed on 4 September 2024).
2. Tegmark, M.; Omohundro, S. Provably safe systems: The only path to controllable agi. *arXiv* **2023**, arXiv:2309.01933.
3. Dalrymple, D.; Skalse, J.; Bengio, Y.; Russell, S.; Tegmark, M.; Seshia, S.; Omohundro, S.; Szegedy, C.; Goldhaber, B.; Ammann, N.; et al. Towards Guaranteed Safe AI: A Framework for Ensuring Robust and Reliable AI Systems. *arXiv* **2024**, arXiv:2405.06624.
4. Zhou, B.; Ding, G. Survey of intelligent program synthesis techniques. In Proceedings of the International Conference on Algorithms, High Performance Computing, and Artificial Intelligence (AHPCAI 2023), Yinchuan, China, 18–19 August 2023 ; SPIE; Springer: Bellingham, WA, USA, 2023; Volume 12941, pp. 1122–1136.
5. Odena, A.; Shi, K.; Bieber, D.; Singh, R.; Sutton, C.; Dai, H. BUSTLE: Bottom-Up program synthesis through learning-guided exploration. *arXiv* **2020**, arXiv:2007.14381.
6. Wu, J.; Wei, L.; Jiang, Y.; Cheung, S.C.; Ren, L.; Xu, C. Programming by Example Made Easy. *ACM Trans. Softw. Eng. Methodol.* **2023**, *33*, 1–36. [CrossRef]
7. Sobania, D.; Briesch, M.; Rothlauf, F. Choose your programming copilot: A comparison of the program synthesis performance of github copilot and genetic programming. In Proceedings of the Genetic and Evolutionary Computation Conference, Boston, MA, USA, 9–13 July 2022; pp. 1019–1027.
8. Olah, C.; Cammarata, N.; Schubert, L.; Goh, G.; Petrov, M.; Carter, S. Zoom in: An introduction to circuits. *Distill* **2020**, *5*, e00024-001. [CrossRef]
9. Cammarata, N.; Goh, G.; Carter, S.; Schubert, L.; Petrov, M.; Olah, C. Curve Detectors. *Distill* **2020**. Available online: https://distill.pub/2020/circuits/curve-detectors (accessed on 26 November 2024). [CrossRef]
10. Wang, K.; Variengien, A.; Conmy, A.; Shlegeris, B.; Steinhardt, J. Interpretability in the wild: A circuit for indirect object identification in gpt-2 small. *arXiv* **2022**, arXiv:2211.00593.
11. Olsson, C.; Elhage, N.; Nanda, N.; Joseph, N.; DasSarma, N.; Henighan, T.; Mann, B.; Askell, A.; Bai, Y.; Chen, A.; et al. In-context Learning and Induction Heads. *Transform. Circuits Thread* **2022**. Available online: https://transformer-circuits.pub/2022/in-context-learning-and-induction-heads/index.html (accessed on 26 November 2024).
12. Goh, G.; Cammarata, N.; Voss, C.; Carter, S.; Petrov, M.; Schubert, L.; Radford, A.; Olah, C. Multimodal Neurons in Artificial Neural Networks. *Distill* **2021**. Available online: https://distill.pub/2021/multimodal-neurons (accessed on 26 November 2024). [CrossRef]
13. Gurnee, W.; Tegmark, M. Language models represent space and time. *arXiv* **2023**, arXiv:2310.02207.
14. Vafa, K.; Chen, J.Y.; Kleinberg, J.; Mullainathan, S.; Rambachan, A. Evaluating the World Model Implicit in a Generative Model. *arXiv* **2024**, arXiv:2406.03689.
15. Burns, C.; Ye, H.; Klein, D.; Steinhardt, J. Discovering latent knowledge in language models without supervision. *arXiv* **2022**, arXiv:2212.03827.
16. Marks, S.; Tegmark, M. The geometry of truth: Emergent linear structure in large language model representations of true/false datasets. *arXiv* **2023**, arXiv:2310.06824.
17. McGrath, T.; Kapishnikov, A.; Toma**v**sev, N.; Pearce, A.; Wattenberg, M.; Hassabis, D.; Kim, B.; Paquet, U.; Kramnik, V. Acquisition of chess knowledge in alphazero. *Proc. Natl. Acad. Sci. USA* **2022**, *119*, e2206625119. [CrossRef] [PubMed]
18. Toshniwal, S.; Wiseman, S.; Livescu, K.; Gimpel, K. Chess as a testbed for language model state tracking. In Proceedings of the AAAI Conference on Artificial Intelligence, Virtually, 22 February–1 March 2022; Volume 36, pp. 11385–11393.
19. Li, K.; Hopkins, A.K.; Bau, D.; Viégas, F.; Pfister, H.; Wattenberg, M. Emergent world representations: Exploring a sequence model trained on a synthetic task. *arXiv* **2022**, arXiv:2210.13382.
20. Nanda, N.; Chan, L.; Liberum, T.; Smith, J.; Steinhardt, J. Progress measures for grokking via mechanistic interpretability. *arXiv* **2023**, arXiv:2301.05217.
21. Liu, Z.; Kitouni, O.; Nolte, N.; Michaud, E.J.; Tegmark, M.; Williams, M. Towards Understanding Grokking: An Effective Theory of Representation Learning. In Proceedings of the Thirty-Sixth Conference on Neural Information Processing Systems, New Orleans, LA, USA, 28 November 2022.
22. Zhong, Z.; Liu, Z.; Tegmark, M.; Andreas, J. The clock and the pizza: Two stories in mechanistic explanation of neural networks. In *Advances in Neural Information Processing Systems: 37th Conference on Neural Information Processing Systems (NeurIPS 2023), New Orleans, LA, USA, 10–16 December 2023*; Volume 36.
23. Quirke, P.; Barez, F. Understanding Addition in Transformers. *arXiv* **2023**, arXiv:2310.13121.
24. Chughtai, B.; Chan, L.; Nanda, N. A Toy Model of Universality: Reverse Engineering how Networks Learn Group Operations. In Proceedings of the 40th International Conference on Machine Learning, Honolulu, HI, USA, 23–29 July 2023; Krause, A., Brunskill, E., Cho, K., Engelhardt, B., Sabato, S., Scarlett, J., Eds.; PMLR (Proceedings of Machine Learning Research) 2023; Volume 202, pp. 6243–6267.
25. Hanna, M.; Liu, O.; Variengien, A. How does GPT-2 compute greater-than?: Interpreting mathematical abilities in a pre-trained language model. *arXiv* **2023**, arXiv:2305.00586.
26. Charton, F. Can transformers learn the greatest common divisor? *arXiv* **2023**, arXiv:2308.15594.
27. Lindner, D.; Kramár, J.; Farquhar, S.; Rahtz, M.; McGrath, T.; Mikulik, V. Tracr: Compiled transformers as a laboratory for interpretability. *arXiv* **2023**, arXiv:2301.05062.

28. Friedman, D.; Wettig, A.; Chen, D. Learning transformer programs. *Adv. Neural Inf. Process. Syst.* **2023**, *36*, 49044–49067.
29. Bills, S.; Cammarata, N.; Mossing, D.; Tillman, H.; Gao, L.; Goh, G.; Sutskever, I.; Leike, J.; Wu, J.; Saunders, W. Language Models Can Explain Neurons in Language Models. 2023. Available online: https://openaipublic.blob.core.windows.net/neuron-explainer/paper/index.html (accessed on 26 November 2024).
30. Cunningham, H.; Ewart, A.; Riggs, L.; Huben, R.; Sharkey, L. Sparse autoencoders find highly interpretable features in language models. *arXiv* **2023**, arXiv:2309.08600.
31. Bricken, T.; Templeton, A.; Batson, J.; Chen, B.; Jermyn, A.; Conerly, T.; Turner, N.; Anil, C.; Denison, C.; Askell, A.; et al. Towards Monosemanticity: Decomposing Language Models with Dictionary Learning. *Transform. Circuits Thread* **2023**. Available online: https://transformer-circuits.pub/2023/monosemantic-features/index.html (accessed on 26 November 2024).
32. Conmy, A.; Mavor-Parker, A.N.; Lynch, A.; Heimersheim, S.; Garriga-Alonso, A. Towards automated circuit discovery for mechanistic interpretability. *arXiv* **2023**, arXiv:2304.14997.
33. Syed, A.; Rager, C.; Conmy, A. Attribution Patching Outperforms Automated Circuit Discovery. *arXiv* **2023**, arXiv:2310.10348.
34. Marks, S.; Rager, C.; Michaud, E.J.; Belinkov, Y.; Bau, D.; Mueller, A. Sparse feature circuits: Discovering and editing interpretable causal graphs in language models. *arXiv* **2024**, arXiv:2403.19647.
35. Karpathy, A.; Johnson, J.; Fei-Fei, L. Visualizing and understanding recurrent networks. *arXiv* **2015**, arXiv:1506.02078.
36. Strobelt, H.; Gehrmann, S.; Pfister, H.; Rush, A.M. LSTMVis: A Tool for Visual Analysis of Hidden State Dynamics in Recurrent Neural Networks. *IEEE Trans. Vis. Comput. Graph.* **2018**, *24*, 667–676. [CrossRef]
37. Giles, C.L.; Horne, B.G.; Lin, T. Learning a class of large finite state machines with a recurrent neural network. *Neural Netw.* **1995**, *8*, 1359–1365. [CrossRef]
38. Wang, Q.; Zhang, K.; Ororbia II, A.G.; Xing, X.; Liu, X.; Giles, C.L. An empirical evaluation of rule extraction from recurrent neural networks. *arXiv* **2017**, arXiv:1709.10380. [CrossRef] [PubMed]
39. Weiss, G.; Goldberg, Y.; Yahav, E. Extracting automata from recurrent neural networks using queries and counterexamples. In Proceedings of the 35th International Conference on Machine Learning, Stockholm, Sweden, 10–15 July 2018; pp. 5247–5256.
40. Oliva, C.; Lago-Fernández, L.F. On the interpretation of recurrent neural networks as finite state machines. In Proceedings of the Artificial Neural Networks and Machine Learning–ICANN 2019: Theoretical Neural Computation: 28th International Conference on Artificial Neural Networks, Munich, Germany, 17–19 September 2019; Proceedings, Part I 28; Springer: Berlin, Germany, 2019; pp. 312–323.
41. Muvskardin, E.; Aichernig, B.K.; Pill, I.; Tappler, M. Learning finite state models from recurrent neural networks. In Proceedings of the International Conference on Integrated Formal Methods, Lugano, Switzerland, 7–10 June 2022; Springer: Berlin, Germany, 2022; pp. 229–248.
42. Udrescu, S.M.; Tan, A.; Feng, J.; Neto, O.; Wu, T.; Tegmark, M. AI Feynman 2.0: Pareto-optimal symbolic regression exploiting graph modularity. *Adv. Neural Inf. Process. Syst.* **2020**, *33*, 4860–4871.
43. Cranmer, M. Interpretable machine learning for science with PySR and SymbolicRegression. jl. *arXiv* **2023**, arXiv:2305.01582.
44. Cranmer, M.; Sanchez Gonzalez, A.; Battaglia, P.; Xu, R.; Cranmer, K.; Spergel, D.; Ho, S. Discovering symbolic models from deep learning with inductive biases. *Adv. Neural Inf. Process. Syst.* **2020**, *33*, 17429–17442.
45. Ma, H.; Narayanaswamy, A.; Riley, P.; Li, L. Evolving symbolic density functionals. *Sci. Adv.* **2022**, *8*, eabq0279. [CrossRef]
46. Gu, A.; Dao, T. Mamba: Linear-time sequence modeling with selective state spaces. *arXiv* **2023**, arXiv:2312.00752.