Automated Mechanistic Interpretability for Neural Networks

by

Isaac C. Liao

B.S. Computer Science and Engineering and Physics, MIT, 2023

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2024

© 2024 Isaac C. Liao. This work is licensed under a CC BY-NC-ND 4.0 license.

The author hereby grants to MIT a nonexclusive, worldwide, irrevocable, royalty-free license to exercise any and all rights under copyright, including to reproduce, preserve, distribute and publicly display copies of the thesis, or release the thesis under an open-access license.

Authored by: Isaac C. Liao
Department of Electrical Engineering and Computer Science
May 18, 2024

Certified by: Max Tegmark
Professor of Physics, Thesis Supervisor

Accepted by: Katrina LaCurts
Chair
Master of Engineering Thesis Committee

Automated Mechanistic Interpretability for Neural Networks

by

Isaac C. Liao

Submitted to the Department of Electrical Engineering and Computer Science
on May 18, 2024 in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

ABSTRACT

Mechanistic interpretability research aims to deconstruct the underlying algorithms that neural networks use to perform computations, such that we can modify their components, causing them to change behavior in predictable and positive ways. This thesis details three novel methods for automating the interpretation process for neural networks that are too large to manually interpret. Firstly, we detect inherently multidimensional representations of data; we discover that large language models use circular representations to perform modular addition tasks. Secondly, we introduce methods to penalize complexity in neural circuitry; we discover the automatic emergence of interpretable properties such as sparsity, weight tying, and circuit duplication. Last but not least, we apply neural network symmetries to put networks into a simplified normal form, for conversion into human-readable python; we introduce a program synthesis benchmark with this and successfully convert 32 out of 62 of them.

Thesis supervisor: Max Tegmark
Title: Professor of Physics

Acknowledgments

I extend a great many thanks to my advisor, Max Tegmark, for giving me the wonderful opportunity to pursue research in the Tegmark AI Safety Group, and for his motivation and guidance. I would also like to thank Ziming Liu for helping me get to know the group, mentoring me, and bringing me up to speed with the group's most recent research activities. I also thank Joshua Engels for our wonderful close collaboration on the first of the three papers in this thesis, and the deep discussions we had about our hunt for multidimensional representations. This project has been made possible by the combined effort of many collaborators who contributed to the three sub-parts of this thesis; their names are listed in the relevant chapters.

I would like to acknowledge the MIT SuperCloud and Lincoln Laboratory Supercomputing Center for providing HPC resources that have contributed to the research results reported in this thesis. This work was also sponsored in part by the National Science Foundation under Cooperative Agreement PHY-2019786 (The NSF AI Institute for Artificial Intelligence and Fundamental Interactions, http://iaifi.org/), the Beneficial AI Foundation, Erik Otto, Jaan Tallinn, the Rothberg Family Fund for Cognitive Science, the NSF Graduate Research Fellowship (Grant No. 2141064), the NSF AI Institute for Artificial Intelligence and Fundamental Interactions through NSF grant PHY-2019786, and the MIT Department of Physics.

Lastly, I would like to thank my parents Hiu and Anita, my brother Herman, my girlfriend Grace, and all of my friends for their love which helped me greatly during my studies.

Contents

Title page 1

Abstract 3

Acknowledgments 5

List of Figures 11

List of Tables 17

1 Introduction 19

1.1 Thesis Overview 20

1.2 My Contributions 23

2 Not All Language Model Features Are Linear 25

2.1 Introduction 26

2.1.1 Contributions 27

2.2 Related Work 27

2.3 Definitions and Theory 29

2.3.1 Multi-Dimensional Features 30

2.3.2 Superposition 32

2.4 Sparse Autoencoders Find Multi-Dimensional Features 34

2.5 Circular Representations in Large Language Models 36

2.5.1 Intervening on Circular Day and Month Representations 37
2.5.2 Decomposing Hidden States 40
2.6 Broader Impact and Limitations 43
References 43
2.A Proofs 48
2.B More Plots 51

3 Generating Interpretable Networks using Hypernetworks 56
3.1 Introduction 57
3.2 L1 Network Experiments 60
3.2.1 Interpretation of Generated Networks 60
3.2.2 Order Parameters 62
3.2.3 Development of Algorithms Throughout Training 64
3.2.4 Generalization Capabilities 66
3.2.5 Baseline Algorithm 67
3.3 Related Work 68
3.4 Conclusion and Discussion 69
3.5 Acknowledgements 71
References 71
3.A Intuition of Why Hypernetworks Work 76
3.B Method 77
3.B.1 Force-Directed Graph Drawings 77
3.B.2 Attentional Hypernetworks 78

4 Opening the AI black box: Program Synthesis via Mechanistic Interpretability 84
4.1 Introduction 85
4.2 Related Work 85
4.3 MIPS, our program synthesis algorithm 86

4.3.1 AutoML optimizing for simplicity . . . 89
4.3.2 Auto-simplification . . . 90
4.3.3 Boolean and integer autoencoders . . . 92
4.4 Results . . . 95
4.4.1 Benchmark . . . 95
4.4.2 Evaluation . . . 96
4.4.3 Performance . . . 98
4.5 Conclusions . . . 99
4.5.1 Findings . . . 99
4.5.2 Outlook . . . 102
References . . . 103
4.A Lattice finding using generalized greatest common divisor (GCD) . . . 107
4.A.1 Problem formulation . . . 107
4.A.2 Regular GCD . . . 107
4.A.3 Generalized GCD in D dimensions . . . 108
4.B Linear lattice finder . . . 112
4.C Symbolic regression . . . 112
4.D Neural Network Normalization Algorithms . . . 113
4.D.1 Whitening Transformation . . . 115
4.D.2 Jordan Normal Form Transformation . . . 115
4.D.3 Toeplitz Transformation . . . 118
4.D.4 De-biasing Transformation . . . 119
4.D.5 Quantization Transformation . . . 119
4.E Supplementary training data details . . . 119
4.F Generated programs . . . 120
4.F.1 Formal Verification . . . 130

5 Conclusion **136**

A Notes on Jordan Normal Form **137**

References **139**

List of Figures

2.1 Examples of testing reducibility of different features. **Left in each subfigure:** Histograms of the distribution of $\mathbf{v} \cdot \mathbf{x}$ with red lines indicating a 2ϵ-wide region. **Right in each subfigure:** Distributions of x. For the first feature (a), 34.37% lies within the dotted lines, indicating that $\mathbf{f}$ is a mixture. For the second feature (b), 17.84% lies within, indicating that $\mathbf{f}$ is not a mixture. 32

2.2 Reconstructions from the sparse autoencoder search technique. 34

2.3 Projections onto top 2 PCA directions on modular arithmetic tasks on the starting day/month token. Colors are in order of days of week/months of year. 35

2.4 Visual representation of the intervention process and intervention results. 37

2.5 Off distribution interventions on Mistral layer 5 on the `Weekdays` task. The color corresponds to the highest logit after performing our circular subspace intervention process on that point. 39

2.6 Residual RGB plots from explanation via regression, on Mistral hidden states of the token that predicts γ in the `Weekdays` task, from layers 17 to 29. From top to bottom, we show each residual RGB plot after adding the function(s) $\mathbf{f}_i$ labelled just underneath, as well as the resulting r^2 value. We write a, b, c for α, β, γ, and "tmr" meaning "tomorrow" for $\beta = 1$. We also write "circle for x" meaning the inclusion of two functions $\mathbf{f}_i(x) = \{\cos, \sin\}(2\pi x/7)$. 42

2.7 An example of testing ϵ-irreducibility on an irreducible feature. **Left:** histogram of the distribution of $\mathbf{v} \cdot \mathbf{f}$. Red lines indicate a 2ϵ-wide region in which we maximize the probability mass. **Right:** The distribution of $\mathbf{f}$. 7.94% of the feature distribution lies within the dotted lines, which is roughly on the order of $\epsilon = 0.1$, indicating that this supposed "feature" is unlikely to be a mixture. 51
2.8 An example of testing ϵ-irreducibility on a grid dataset. **Left:** histogram of the distribution of $\mathbf{v} \cdot \mathbf{f}$. Red lines indicate a 2ϵ-wide region in which we maximize the probability mass. **Right:** The distribution of $\mathbf{f}$. 25.90% of the feature distribution lies within the dotted lines, much higher than zero, indicating that this supposed "feature" is actually a mixture. 52
2.9 The top two PCA dimensions of model hidden states on the α token show that circular representations of α are present in various layers. 53
2.10 Attention and MLP patching results. Results are averaged over 20 different runs with fixed α and varying β and 20 different runs with fixed β and varying α. 54
2.11 Residual RGB plots from explanation via regression, on Mistral hidden states of the token where α is input in the `Weekdays` task, from layers 17 to 29. From top to bottom, we show each residual RGB plot after adding the function(s) f_i labelled just underneath, as well as the resulting r^2 value. We write a, b, c for α, β, γ. 54
2.12 Residual RGB plots from explanation via regression, on LLAMA 3 hidden states of the token that predicts γ in the `Months` task, from layers 13 to 29. From top to bottom, we show each residual RGB plot after adding the function(s) f_i labelled just underneath, as well as the resulting r^2 value. We write a, b, c for α, β, γ. We also write "circle for x" meaning the inclusion of two functions $f_i(x) = \{\cos, \sin\}(2\pi x/7)$. 55

3.1 Three algorithms for computing the L1 norm, discovered by the hypernetwork. **Left:** the convexity algorithm. **Center top:** the negative pudding algorithm. **Center bottom:** the positive pudding algorithm. **Right:** the double-sided algorithm. The output neuron is in the center of all visualizations. The visualization method is included in Appendix 3.B.1. 59

3.2 Networks that try to compute the L1 norm cluster into three general algorithms in order parameter space, spanned by the "strongest connection", "double-sidedness" and "seed dependence" order parameters. Lines are generated by sweeping β from 10^{-12} to 1 in 30 increments logarithmically. The dotted lines represent hand-picked boundaries which determine when phase transitions between algorithms occur. Lines are grouped and colored by the phases where they start and end at. 63

3.3 Five ways to develop the three algorithms through training. The horizontal axis is the step number, while the vertical axis is the β parameter used to generate the network. **Red:** convexity algorithm. **Green:** pudding algorithm. **Blue:** double-sided algorithm. 64

3.4 Loss of neural network generated when the encoder is either used or ignored. The black dotted line indicates when the performance is unaffected by the presence of the encoder, i.e., when the encoder is unused during the weight generation process. Lines are generated by sweeping β from 10^{-12} to 1 in 30 logarithmic increments. . 65

3.5 Loss of neural network generated with various input dimensions and hidden dimensions using a hypernetwork. The red contour denotes a loss of 0.07, while the blue denotes a loss of 0.15. The hypernetwork was only trained to generate networks with input dimension 16 and hidden dimension 48 (yellow star), yet it can produce networks of diverse shapes which all compute the L1 norm with reasonable accuracy. 66

3.6 Visualization of a $(16, 48, 1)$ neural networks trained to compute the L1 norm of a vector. Input neurons in green, output neurons in magenta. Positive weights in red, and negative weights in blue. The Adam-trained network is messy, whereas the hypernetwork-generated network is much easier to interpret. 67

3.7 The full architecture of the hypernetwork. The hyperhypernetwork above generates weights for the hypernetwork, which generates weights for the network, on which the loss is evaluated. There are many components in the hypernetwork, each drawn individually on the left: graph attention, positional encodings, information bottleneck channels, learned hyperfeatures, and random variables. 83

4.1 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. 87

4.2 These hidden structures can be turned into discrete representations. Left: the hidden states for the bitstring addition task are seen to form four clusters, corresponding to 2 bits: the output bit and the carry bit. Right: the hidden states for the Sum_Last2 task are seen to form clusters on a 2D lattice corresponding to two integers. 92

4.3 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. 99

4.4 Comparison of code generated from an RNN trained on Sum_Last5, without (top) and with (bottom) normalizers. The whitening normalizer provided numerical stability to the Jordan normal form normalizer, which itself simplified the recurrent portion of the program. The Toeplitz and de-biasing normalizers jointly sparsified the occurrences of x in the program, and the number of terms required to compute y. The quantization normalizer enabled all variables to be represented as integers. . 100

4.5 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. 101

4.6 Both red and blue basis 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). 110

List of Tables

2.1 Aggregate model top-1 accuracy on days of the week and months of the year modular arithmetic tasks. GPT-2 is worse than random because it predicts non weekday/month tokens like “a” or “the”. 35

2.2 Notation table, in order encountered in the paper. Matrices are written in capital bold, distributions in caligraphy, and vectors and scalars in lowercase. 48

4.1 Benchmark results. For tasks with note “see text”, please refer to Appendix 4.E . . 97

4.2 AutoML architecture search results. All networks achieved 100% accuracy on at least one test batch. 135

Chapter 1

Introduction

Neural networks have recently become central to multiple safety-critical applications, despite our lack of understanding of their learned internal functioning. With the recent breakneck pace of machine learning research in generative pretrained models such as large language models (LLMs), safety considerations have quickly become a critical priority. The capabilities of models increase rapidly, and safety precautions struggle to keep pace. The worst possible outcome of this development is the creation of unsafe artificial general intelligence (AGI) with the capacity to do dangerous things, which may have a harmful impact on humanity. It follows that a major priority in AI safety research is to find methods to guarantee or encourage specific desired neural network behaviors and properties. With safer, more controlled methods of building intelligent models, we may prevent potential damages to society as model capabilities progress.

Most current work surrounding safety in large language models and other state-of-the-art systems aims to provide some heuristic form of encouragement or modification for models to abstain from producing harmful outputs. Examples include adverserial robustness training (Drenkow et al. 2021), fine-tuning, and RLHF (Christiano et al. 2017; Ziegler et al. 2019).

Amongst these top-down approaches to safety, which try to build safety properties into machine learning systems, is a bottom-up approach dubbed “mechanistic interpretability”, where we try to break machine learning systems down into safe components. In mechanistic interpretability,

we dissect neural networks to understand the learned internal structures, the same way that biologists dissect organisms to understand their evolved internal structures. And just as medical practitioners use this knowledge to treat patients, mechanistic interpretability researchers aim to use their knowledge to *modify* models to induce or prevent safety-relevant behaviors. More recently, researchers inlcuding myself have become more interested in automated mechanistic interpretability, whereby an algorithm is used to convert model weights into human-interpretable explanations. This is especially popular with LLMs, which have far too many weights for humans to possibly dissect and understand. (Sharkey, Braun, and Millidge 2022; Elhage et al. 2021)

1.1 Thesis Overview

The next three thesis chapters correspond to three research papers I have contributed to, each of which explores one research direction relating to automatic mechanistic interpretability.

1. **Circular Representations in Large Language Models**: Neural networks trained to perform modular addition have the curious tendency to create circular representations of numbers. This recent discovery is easy to reproduce and study in small experiments, and the theory surrounding these toy examples is rapidly advancing. But at the end of the day, this work is hardly useful for understanding real LLMs unless we can find circular representations in them, instead of in toy examples. So, the hunt has been on to discover the elusive circular representations "in the wild".

   A few of us in the Tegmark group have finally found them. We have piled up some evidence of their existence, and we did some experiments to understand what role they play. I have included a self-contained copy of this work in Chapter 2. A rough outline of the work is as follows:

   - We mathematically define the internal operation of LLMs in terms of "features", the fundamental unit of data manipulated in attention layers and MLPs. We aim for definitions and tests which will separate the behavior of conventionally understood

"one-dimensional representations" from "multidimensional representations" such as the circular representations that pop up in toy examples. We perform experiments to illustrate the properties of our definitions and how they classify toy datasets into one-dimensional and multidimensional representations.

- We introduce a new method, as an alternative to sparse auto-encoders (SAEs), for auto-extracting multidimensional features, like circular representations, from LLMs, on large datasets like The Pile (Gao et al. 2020).
- We test out various prompts which ask various language models to perform naturally-occurring modular addition tasks. The addition problems which the LLMs are most successful at solving involve measures of time, such as days of the week, months of the year, and minutes in an hour.
- We deconstruct the hidden states that show up in the middle layers of the LLMs, and show using a new technique that circular representations of the summands and sum are present in these hidden states. This technique casts a wide net that allows us to quickly identify every human-interpretable every feature in the hidden state, allowing us to catch the circlular representation in the mix.
- We show that circular representations can be probed out of the hidden states.
- We perform intervention and patching experiments which replace individual features in the addition problem representations, to test if the LLMs react according to how we expect. We also test what happens if we replace the circular feature by moving off the circle onto any point in the 2D plane.

2. **Hypernetworks for Generating Interpretable Neural Network Weights**: This research project is an exploration into the question of what it means for a neural network to be *machine-interpretable*, rather than human-interpretable. We suppose that machines can interpret weights by fitting a learnable generative probabilistic model, ie. a "hypernetwork", to the distribution of weights of well-performing networks. The learnable model represents the

machine's understanding of the distribution of "well-trained weights", thereby acting as the machine's interpretation. Ideally, the learned model becomes well-adapted to capture and encode various possible human-discovered interpretations of network weights.

In this project, we demonstrate that the learned model has a tendency to capture several human-interpretable properties, without explicit encouragement towards these properties. For example, we sample networks from the learned probabilistic model that exhibit circuit duplication, sparsity, and weight tying, even though we use no structural assumptions nor L1/sparsity penalties in our model. We also show that the generated networks achieve a low out-of-distribution loss.

A preprint of the paper from this project is available online in Liao, Liu, and Tegmark (2023), and a fully self-contained copy is also included in Chapter 3.

3. **Auto-Simplification of Neural Networks:** This is the idea of automatically reducing a network into a form that humans can more easily understand, without altering the reduced network's behavior, with the goal of making interpretation or safety verification easier. This is an umbrella research direction, which encompasses many classes of methods, such as formal verification for neural networks (Dalrymple et al. 2024; Carr, Jansen, and Topcu 2020; Ayache, Eyraud, and Goudian 2019; Mayr, Visca, and Yovine 2020; Mayr and Yovine 2018; Okudono et al. 2020; Omlin and Giles 1996; Weiss, Goldberg, and Yahav 2018), compilation from neural network to and from other programming languages (Lindner et al. 2024; Weiss, Goldberg, and Yahav 2021), and symmetry identification (Grigsby, Lindsey, and Rolnick 2023).

   My work on auto-simplification was part of a larger project focused on program synthesis via mechanistic interpretability, which involved many collaborators. A preprint of the paper from this project is available online in Michaud et al. (2024), and a fully self-contained copy is also included in Chapter 4. A rough outline of the project is as follows:

   - We introduce a new benchmark for the conversion of neural network weights into short

and interpretable python programs. This benchmark consists of training tasks which can be used to train the network weights before one tries to convert them into programs.

- We train small recurrent neural networks (RNNs) on the tasks in the benchmark.
- We identify symmetry transformations of the neural network weights, and apply various normalization algorithms to simplify the RNNs so that they would be easier to interpret.
- We attempt to find lattices in the hidden representations, and we apply symbolic, boolean, and integer regression to decode these lattice representations into python programs.
- We compare our method of program synthesis to the use of a large language model to write a program when prompted with raw data generated from benchmark tests.

1.2 My Contributions

All of the aforementioned research was done in collaboration with co-authors:

1. **Circular Representations in Large Language Models**: In this project, I discussed extensively with coauthors on how we can make a bulletproof definition of a "feature" for our work, which captures most currently well-accepted examples of features. I also performed our experiments which demonstrate the properties of our feature definitions on toy examples. I am responsible for coming up with our technique for quickly deconstructing hidden states into human-interpretable features. I extensively tested the properties of interactions between features in individual layers of the LLMs, guided by crucial patching experiments from my collaborator Josh Engels and with support and ideas from Max Tegmark.

2. **Hypernetworks for Generating Interpretable Neural Network Weights**: This was a fairly independent research project led by myself, with great input, discussion, and advice from my collaborator Ziming Liu and my supervisor Max Tegmark.

3. **Auto-Simplification of Neural Networks:** My role in the project was to come up with, refine, and operate the normalization algorithms. More specifically, I identified the symmetries

which we could exploit, constructed and refined programs to pick and apply the symmetry transformations, applied them to the trained networks, and measured interpretability-related metrics such as weight sparsity and activation sparsity. I am responsible for coming up with and implementing the algorithm for approximate Jordan Normal Form in Appendix 4.D.2. I also came up with 14 out of the 62 benchmark tasks.

Chapter 2

Not All Language Model Features Are Linear

Joshua Engels[*], Isaac Liao[*], Eric Michaud, Wes Gurnee, and Max Tegmark

MIT

{jengels,iliao,ericjm}@mit.edu,wesgurnee@gmail.com,tegmark@mit.edu

ABSTRACT

Recent work has proposed the *linear representation hypothesis*: that language models perform computation by manipulating one-dimensional representations of concepts ("features") in activation space. In contrast, we explore whether some language model representations may be inherently multi-dimensional. We begin by developing a rigorous definition for irreducible multi-dimensional features based on whether they can be decomposed into either independent or non-co-occurring lower dimensional features. Motivated by these definitions, we design a scalable method that uses sparse autoencoders to automatically find multi-dimensional features in GPT-2 and Mistral 7B. These auto-discovered features include many strikingly interpretable examples, e.g. *circular* features representing days of the week and months of the year. We

[*]Equal contribution.

identify tasks where these circles are used to solve computational problems involving modular arithmetic in days of the week and months of the year. Finally, we provide evidence that these circular features are indeed the fundamental unit of computation in these tasks with intervention experiments on Mistral 7B and Llama3 8B, and we find further circular representations by breaking down the residual stream for these tasks into interpretable components.

2.1 Introduction

Language models trained for next-token prediction on large text corpora have demonstrated remarkable capabilities, including coding, reasoning, and in-context learning (Bubeck et al. 2023; Achiam et al. 2023; Anthropic 2024; Team et al. 2023). However, the specific algorithms learned to achieve these capabilities remain largely a mystery to researchers; we do not understand how language models write poetry. Mechanistic interpretability is a field that seeks to address this gap in understanding by reverse-engineering trained models from the ground up into variables (features) and the programs (circuits) that process these variables (Olah et al. 2020).

One mechanistic interpretability research direction has focused on understanding toy models in detail. This work has found multi-dimensional representations of inputs such as lattices (Michaud et al. 2024) and circles (Z. Liu et al. 2022), and has successfully reverse-engineered the algorithms that models use to manipulate these representations. A separate direction has identified one-dimensional representations of high level concepts and quantities in large language models (Gurnee and Tegmark 2023; Marks and Tegmark 2023; Heinzerling and Inui 2024). These findings have led to the *linear representation hypothesis*: that all representations in pretrained langauge models are one-dimensional lines, and that we can understand model behavior as nonlinear manipulations of these linear representations (Park, Choe, and Veitch 2023; Bricken et al. 2023). In this work, we bridge the gap between these to regimes by providing evidence that also language models use multi-dimensional representations.

2.1.1 Contributions

1. In Section 2.3, we generalize the one-dimensional definition of a language model feature to multi-dimensional features, provide an updated superposition hypothesis to account for these new features, and analyze the reduction in a model's representation space implied by using multi-dimensional features. We also develop a theoretically grounded and empirically practical test for irreducible features, and run this test on some sample distributions.

2. In Section 2.4, we present a theoretically motivated and scalable method for finding multi-dimensional features using sparse autoencoders. Using this method, we auto-identify many multi-dimensional representations automatically in GPT-2 and Mistral 7B, including circular representations for the day of the week and month of the year. To the best of our knowledge, we are the first to find an emergent circular representation of a quantity in a large language model.

3. In Section 2.5, we propose two tasks, modular addition in days of the week and in months of the year, that we hypothesize will cause models to use these circular representations. We intervene on the circular representations in Mistral 7B and Llama 3 8B to show that the models do indeed use these circular representations for these tasks. Finally, we present novel methods for decomposing LLM hidden states, which we use to reveal circles in the computed day of the week and month of the year.

2.2 Related Work

Linear Representations: Early word embedding methods such as GloVe and Word2vec, although only trained using co-occurrence data, were found to contain directions in their vector spaces corresponding to semantic concepts, e.g. the well-known formula f(king) - f(man) + f(woman) = f(queen) (Tomáš Mikolov, Yih, and Zweig 2013; Pennington, Socher, and Manning 2014; Tomas Mikolov et al. 2013). More recent research has found similar evidence of linear representations in

sequence models trained only on next token prediction, including Othello board positions (Nanda, A. Lee, and Wattenberg 2023; K. Li et al. 2022), the truth value of assertions (Marks and Tegmark 2023), and numeric quantities such as longitude, latitude, birth year, and death year (Gurnee and Tegmark 2023; Heinzerling and Inui 2024). These results have inspired the linear representation hypothesis (Park, Choe, and Veitch 2023; Elhage et al. 2022) defined above. Recent theoretical work provides evidence for this hypothesis, assuming a latent (binary) variable- based model of language (Y. Jiang et al. 2024). Empirically, dictionary learning has shown success in breaking down a model's feature space into a sparse over-complete basis of linear features using sparse autoencoders (Bricken et al. 2023; Cunningham et al. 2023). These works assume that the number of linear features stored in superposition exceeds the model dimensionality (Elhage et al. 2022).

Nonlinear Representations: There has been comparatively little research on nonlinear features. One recent paper (Kim and Suzuki 2024) proves that a one-layer swapped order (MLP followed by attention) transformer can in-context-learn a nonlinear mapping function followed by a linear regression, implying that the "features" between the MLP and attention blocks are nonlinear. Another work (Shai et al. 2024) finds that a transformer trained on a hidden Markov model uses a fractal structure to represent the probability of each next token. These works analyze toy models, and so it is not clear if large language models will have similar nonlinear features. A separate idea (Black et al. 2022) argues for interpreting neural networks through the polytopes they split the input space into, and identifies regions of low polytope density as "valid" regions for a potential linear representation. Finally, recent work on dictionary learning (Bricken et al. 2023) has speculated about multi-dimensional *feature manifolds*; our work is most similar to this direction, and can be viewed as developing the idea of feature manifolds theoretically and empirically.

Circuits: Circuits research seeks to identify and understand a subset of a model (usually represented as a directed acyclic graph) that explains a specific behavior (Olah et al. 2020). The base units that form the circuits can be layers, neurons (Olah et al. 2020), or sparse autoencoder features (Marks, Rager, et al. 2024). The first circuits-style work looked at the InceptionV1 image model and found line features that were combined into curve detection features (Olah et al. 2020).

More recent work has examined language models, for example the indirect object identification circuit in GPT-2 (Wang et al. 2022). Given the difficulty of designing bespoke experiments, there has been increased research in automated circuit discovery methods (Marks, Rager, et al. 2024; Conmy et al. 2023; Syed, Rager, and Conmy 2023).

Interpretability for Arithmetic Problems: Prior work studies models trained on modular arithmetic problems of the form $a + b = c \pmod m$ and finds that models that generalize well have multi-dimensional circular representations for a and b (Z. Liu et al. 2022). Further work shows that models use these circular representations to compute c via a "clock" algorithm (Nanda, Chan, et al. 2023) and a separate "pizza" algorithm (Zhong et al. 2024). These papers are limited to the case of a small model trained only on modular arithmetic. Another direction has studied how large language models perform basic arithmetic, including a circuits level description of the greater-than operation in GPT-2 (Hanna, O. Liu, and Variengien 2024) and addition in GPT-J (Stolfo, Belinkov, and Sachan 2023). These works find that to perform a computation, models copy pertinent information to the token before the computed result and perform the computation in the subsequent MLP layers. Finally, recent work (Gould et al. 2023) investigates language models' ability to increment numbers and finds linear features that fire on tokens equivalent modulo 10.

2.3 Definitions and Theory

In this section, we focus on transformer models M with L layers that take in token input $\mathbf{t} = (t_1, \ldots, t_n)$, have intermediate hidden states $\mathbf{x}_{1,l}, \ldots, \mathbf{x}_{n,l}$ for all hidden layers l, and output logit vectors $\mathbf{y}_1, \ldots, \mathbf{y}_n$. Given an input set of tokens T, we denote $X_{i,l}$ as the set of all corresponding $\mathbf{x}_{i,l}$. This section focuses on hypotheses that describe how hidden states $\mathbf{x}_{i,l}$ can be decomposed into sums of functions of the input (features). Note that while this is always possible when M is deterministic via the "trivial" evaluation of M itself, we are particularly interested in decomposable, interpretable hypotheses for the construction of $\mathbf{x}_{i,l}$.

We summarize all notation in our paper in Table 2.2 in Section 2.A. In general, matrices are

written in capital bold, vectors and vector valud functions are written in lowercase bold, distributions in capital caligraphy, and sets in capital non-bold.

2.3.1 Multi-Dimensional Features

Definition 2.3.1 (Feature)**.** We define a *d_f-dimensional feature of sparsity s* as a function $\mathbf{f}$ that maps a subset of the input space of probability $1 - s > 0$ into a d_f-dimensional point cloud in $\mathbb{R}^{d_f}$. We say that a feature is *active* on the aforementioned subset.

As an example, let the context length be $n = 1$ (so that inputs are single tokens) and consider a feature $\mathbf{f}$ defined on the set of tokens representing integers. If $\mathbf{f}$ maps integer tokens to equispaced points in $\mathbb{R}^1$, then $\mathbf{f}$ is a 1-dimensional feature that is active on integer tokens. If integer tokens occur 1% of the time across the input distribution, $\mathbf{f}$ has sparsity $s = 0.99$.

For features to be meaningful, we want them to be *irreducible*. In this work, we focus on a form of statistical irreducibly: $\mathbf{f}$ is reducible if we can find two lower-dimensional features that "combine" to create $\mathbf{f}$.

We consider two ways of combining lower dimensional features: composing two statistically independent co-occurring features (in which case $\mathbf{f}$ is "separable") or composing two non-co-occurring features (in which case $\mathbf{f}$ is a "mixture").

The probability distribution over input tokens t induces a d_f-dimensional probability distribution over feature vectors $\mathbf{f}(t)$ — Figure 1 shows two examples. Note that $\mathbf{f}(t)$ is a random vector since t is a random variable; we use $p(\mathbf{f})$ to denote the probability density function of $\mathbf{f}(t)$, suppressing the argument t for brevity.

Definition 2.3.2. A feature $\mathbf{f}$ is *reducible* into features $\mathbf{a}$ and $\mathbf{b}$ if there exists an affine transformation

$$\mathbf{f} \mapsto \mathbf{Rf} + \mathbf{c} \equiv \begin{pmatrix} \mathbf{a} \\ \mathbf{b} \end{pmatrix} \tag{2.1}$$

for some orthonormal $d_f \times d_f$ matrix $\mathbf{R}$ and additive constant $\mathbf{c}$, such that the transformed feature

probability distribution $p(\mathbf{a}, \mathbf{b})$ satisfies at least one of these conditions:

1. p is *separable*, i.e., factorizable as a product of its marginal distributions: $p(\mathbf{a}, \mathbf{b}) = p(\mathbf{a})p(\mathbf{b})$.

2. p is a *mixture* $p(\mathbf{a}, \mathbf{b}) = wp_1(\mathbf{a}, \mathbf{b}) + (1 - w)p_2(\mathbf{a}, \mathbf{b})$ of disjoint probability distributions for $w > 0$, and p_1 is lower-dimensional such that $p_1(\mathbf{a}, \mathbf{b}) = p_1(\mathbf{a})\delta(\mathbf{b})$.

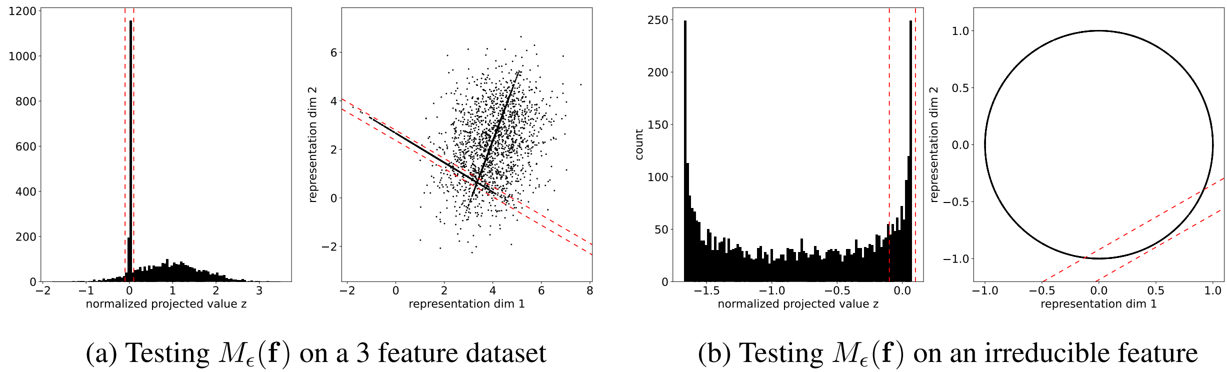
Here δ is the Dirac delta function. By two probability distributions being disjoint, we mean that they have disjoint support (there is no set where they both have positive probability measure, so the two features $\mathbf{a}$ and $\mathbf{b}$ cannot be active at the same time). In Eq. (2.1), $\mathbf{a}$ is simply defined as the first k components of the vector $\mathbf{Rf} + \mathbf{c}$ and $\mathbf{b}$ contains the remaining components. When p is separable or a mixture, we also say that $\mathbf{f}$ is separable or a mixture. We term a feature *irreducible* if it is not reducible, i.e., if no rotation and translation can make it separable or a mixture in the manner above.

Fig. 2.1a shows an example of a 2D feature that is a mixture, because it can be decomposed into features $\mathbf{a}$ and $\mathbf{b}$ where $\mathbf{b}$ is a 1D line distribution (marked in red) and $\mathbf{a}$ is the remainder (a 2D cloud and a line, which can in turn be decomposed). Another example of a feature that is a mixture is a one hot encoding along a simplex; an example of a feature that is separable is a normal distribution (since any multidimensional Gaussian can be rotated to have a diagonal covariance matrix). In natural language, a mixture might be a one hot encoding of "language of the current token", while a separable distribution might be the "latitude" and "longitude" of location tokens.

In practice, because of noise and finite sample size, an empirically observed feature may only be mostly a mixture or may not be exactly separable. Thus, we soften our definitions to permit degrees of reducibility:

Definition 2.3.3 (Separability Index and ϵ-Mixture Index)**.** Consider a feature $\mathbf{f}$. The **separability index** $S(\mathbf{f})$ measures the minimal mutual information between all possible $\mathbf{a}$ and $\mathbf{b}$ defined in Eq. (2.1):

$$S(\mathbf{f}) \equiv \min I(\mathbf{a}; \mathbf{b})$$



(a) Testing $M_\epsilon(\mathbf{f})$ on a 3 feature dataset

(b) Testing $M_\epsilon(\mathbf{f})$ on an irreducible feature

Figure 2.1: Examples of testing reducibility of different features. **Left in each subfigure:** Histograms of the distribution of $\mathbf{v} \cdot \mathbf{x}$ with red lines indicating a 2ϵ-wide region. **Right in each subfigure:** Distributions of x. For the first feature (a), 34.37% lies within the dotted lines, indicating that $\mathbf{f}$ is a mixture. For the second feature (b), 17.84% lies within, indicating that $\mathbf{f}$ is not a mixture.

where I denotes the mutual information. Smaller values of $S(\mathbf{f})$ mean that $\mathbf{f}$ is more separable and therefore more reducible. Note that we only need to minimize over how many of the d_f components to split off as $\mathbf{a}$ and over invertible matrices $\mathbf{R}$, since the additive offset $\mathbf{c}$ does not affect the mutual information.

The ϵ**-mixture index** $M_\epsilon(\mathbf{f})$ tests how often $\mathbf{f}$ can be projected near zero while it is active:

$$M_\epsilon(\mathbf{f}) = \max_{\mathbf{v}\in\mathbb{R}^{d_f},\, c\in\mathbb{R}} \mathbb{P}\left(|\mathbf{v}\cdot\mathbf{f} + c| < \epsilon\sqrt{\mathbb{E}[(\mathbf{v}\cdot\mathbf{f} + c)^2]}\right)$$

Larger values of $M_\epsilon(\mathbf{f})$ mean that $\mathbf{f}$ is more of a mixture and is therefore more reducible.

2.3.2 Superposition

Now that we have a definition for a multi-dimensional feature, we will examine the implications for the *superposition hypothesis* (Elhage et al. 2022).

Definition 2.3.4 (δ-orthogonal matrices)**.** Two matrices $\mathbf{A}_1 \in \mathbb{R}^{d\times d_1}$ and $\mathbf{A}_2 \in \mathbb{R}^{d\times d_2}$ are *δ-orthogonal* if $|\mathbf{x}_1 \cdot \mathbf{x}_2| \le \delta$ for all unit vectors $\mathbf{x}_1 \in \text{colspace}(\mathbf{A}_1)$ and $\mathbf{x}_2 \in \text{colspace}(\mathbf{A}_2)$.

Hypothesis 1 (One-Dimensional Superposition Hypothesis, paraphrased from (Elhage et al. 2022))**.** Hidden states $\mathbf{x}_{i,l}$ are the sum of many ($\gg d$) sparse one-dimensional features f_i and pairwise

δ-orthogonal vectors $\mathbf{v}_i$ such that $\mathbf{x}_{i,l}(t) = \sum_i f_i(t)\mathbf{v}_i$. We assign the value of $f_i(t)$ to zero when t is outside the domain of f_i.

We now present our superposition hypothesis, which instead of positing unknown levels of independence between features, explicitly decomposes the spaces using mixtures and separability until only irreducible features remain:

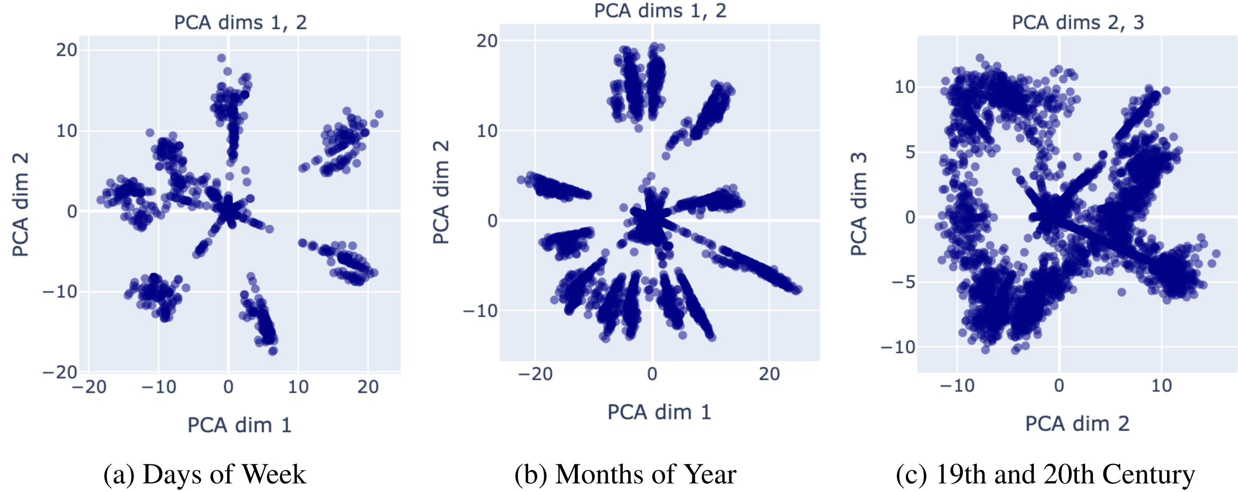
Hypothesis 2 (Our Superposition Hypothesis, changes underlined)**.** Hidden states $\mathbf{x}_{i,l}$ are the sum of many ($\gg d$) sparse low-dimensional irreducible features $\mathbf{f}_i$ and pairwise δ-orthogonal matrices $\mathbf{V}_i \in \mathbb{R}^{d \times d_{\mathbf{f}_i}}$ such that $\mathbf{x}_{i,l}(t) = \sum_i \mathbf{V}_i \mathbf{f}_i(t)$. We assign the value of $\mathbf{f}_i(t)$ to zero when t is outside the domain of $\mathbf{f}_i$. Any subset of features must be mutually independent on their shared domain.

The Johnson-Lindenstrauss (JL) Lemma (Johnson and Lindenstrauss 1984) implies that we can choose $e^{\Theta(d\delta^2)}$ pairwise one-dimensional δ-orthogonal vectors to satisfy Hypothesis 1, thus allowing us to build the model's feature space with a number of one-dimensional δ-orthogonal features exponential in d. In Appendix 2.A, we prove a similar result for low-dimensional projections:

Theorem 1. *For any δ, it is possible to choose $e^{\Theta((d/d_{\max}^2)\delta^2)}$ pairwise δ-orthogonal projection matrices $\mathbf{A}_i \in \mathbb{R}^{n_i \times d}$ where $1 \leq n_i \leq d_{\max}$.*

This exponential reduction in the number of features that are representable δ-orthogonally (as opposed to merely a multiplicative factor reduction) suggests that models will employ higher-dimensional features only for representations that necessitate multi-dimensional, detailed descriptions. Moreover, these representations will likely be highly compressed to fit within the smallest dimensional space possible, potentially leading to interesting encoding strategies; for example, recent work (Morwani et al. 2023) finds that maximum-margin solutions for problems like modular arithmetic consist of Fourier features.

Note that the proof assumes the "worst case" scenario that all of the features are dimension $d_{\max}$, while in practice many of the features may be 1 or low dimensional, so the effect on the capacity of a real model that represents multi-dimensional features is unlikely to be this extreme.



(a) Days of Week (b) Months of Year (c) 19th and 20th Century

Figure 2.2: Reconstructions from the sparse autoencoder search technique.

2.4 Sparse Autoencoders Find Multi-Dimensional Features

Sparse autoencoders (SAEs) are a recent technique to deconstruct a model's hidden states using sparse sums of vectors from an over-complete basis (Bricken et al. 2023; Cunningham et al. 2023). Given a set of hidden states $X_{i,l}$, a one-layer sparse autoencoder of size m and sparsity penalty λ seeks to minimize the following dictionary learning loss (Bricken et al. 2023; Cunningham et al. 2023):

$$\mathtt{DL}(X_{i,l}) = \underset{\mathbf{E}\in\mathbb{R}^{m\times d}, \mathbf{D}\in\mathbb{R}^{d\times m}}{\arg\min} \sum_{\mathbf{x}_{i,l}\in X_{i,l}} \left[\|\mathbf{x}_{i,l} - \mathbf{D}\cdot\mathtt{ReLU}(\mathbf{E}\cdot\mathbf{x}_{i,l})\|_2^2 + \lambda\|\mathtt{ReLU}(\mathbf{E}\cdot\mathbf{x}_{i,l})\|_0\right]$$

In practice, the L_0 loss on the last term is relaxed to L_1 to make the loss differentiable. We call the m columns of $\mathbf{D}$ (vectors in $\mathbb{R}^d$) **dictionary elements**.

We now claim that SAEs can help discover irreducible multi-dimensional features by identifying point sets that are not mixtures (i.e. have a low ϵ-mixture index). For example, assume that $X_{i,l}$ contains an irreducible two dimensional feature $\mathbf{f}$ in its sparse sum (see Hypothesis 2). Because $\mathbf{f}$ is not a mixture, if $\mathbf{D}$ contains just two dictionary elements that span the space of $\mathbf{f}$, then both of them must always have a nonzero weight after the $ReLU$ to perfectly reconstruct $\mathbf{f}$. Thus the Jaccard similarity of the sets of tokens that these two dictionary elements fire on is likely to be high. On the

Model	Weekdays	Months
Llama 3 8B	25 / 49	143 / 144
Mistral 7B	31 / 49	125 / 144
GPT-2	0 / 49	0 / 144

Table 2.1: Aggregate model top-1 accuracy on days of the week and months of the year modular arithmetic tasks. GPT-2 is worse than random because it predicts non weekday/month tokens like "a" or "the".

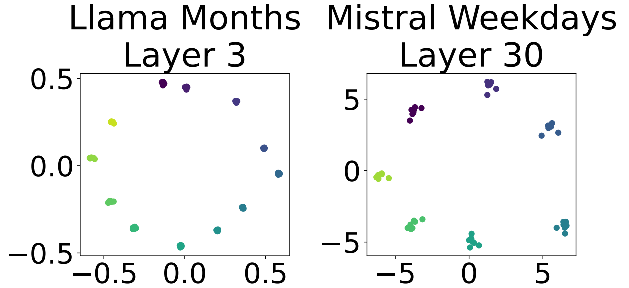


Figure 2.3: Projections onto top 2 PCA directions on modular arithmetic tasks on the starting day/month token. Colors are in order of days of week/months of year.

other hand, if $\mathbf{D}$ contains (many) more than two dictionary elements that span the space of $\mathbf{f}$, then the Jaccard similarity of the sets of tokens each dictionary element fires on may be lower. However, since there are now many dictionary elements with a high projection in the two dimensional feature space, the cosine similarity of the dictionary elements is likely to be high.

Thus for a two dimensional irreducible feature $\mathbf{f}$, we expect there to be groups of dictionary elements with either high cosine or Jaccard similarity corresponding to $\mathbf{f}$. We expect this observation to be true for higher dimensional irreducible features as well. Note that there also may be clusters that correspond to separable features $\mathbf{f}$, as this technique only finds features that are not mixtures. This suggests a natural approach to using sparse autoencoders to search for irreducible multi-dimensional features:

1. Cluster dictionary elements by their pairwise cosine similarity or Jaccard similarity. We find empirically that spectral clustering on cosine similarity works best.

2. For each cluster, run the SAEs on all $\mathbf{x}_{i,l} \in X_{i,l}$ and ablate all dictionary elements not in the cluster. This will give the reconstruction of each $\mathbf{x}_{i,l}$ restricted to the cluster found in step 1 (if no cluster dictionary elements are non-zero for a given point, we ignore the point).

3. Examine the resulting reconstructed activation vectors for irreducible multi-dimensional features, especially ensuring that the reconstruction is not separable. This step can be done manually be visually inspecting the PCA projections for known irreducible multi-dimensional structures (e.g. circles) or automatically by passing the PCA projections to the tests for Definition 2.3.3.

We apply this method to real language models using GPT-2 (Radford et al. 2019) residual stream SAEs trained by Bloom for every residual layer (Bloom 2024) and Mistral 7B (A. Q. Jiang et al. 2023) SAEs that we train on residual layers 16 and 24.

Strikingly, we find clusters of SAE features whose reconstructions form circles. Moreover, these circles are *interpretable*: in GPT-2, we find reconstructions where days of the week (Fig. 2.2a), months of the year (Fig. 2.2b), and years in the 19th and 20th century (Fig. 2.2c) are arranged circularly *in order*.

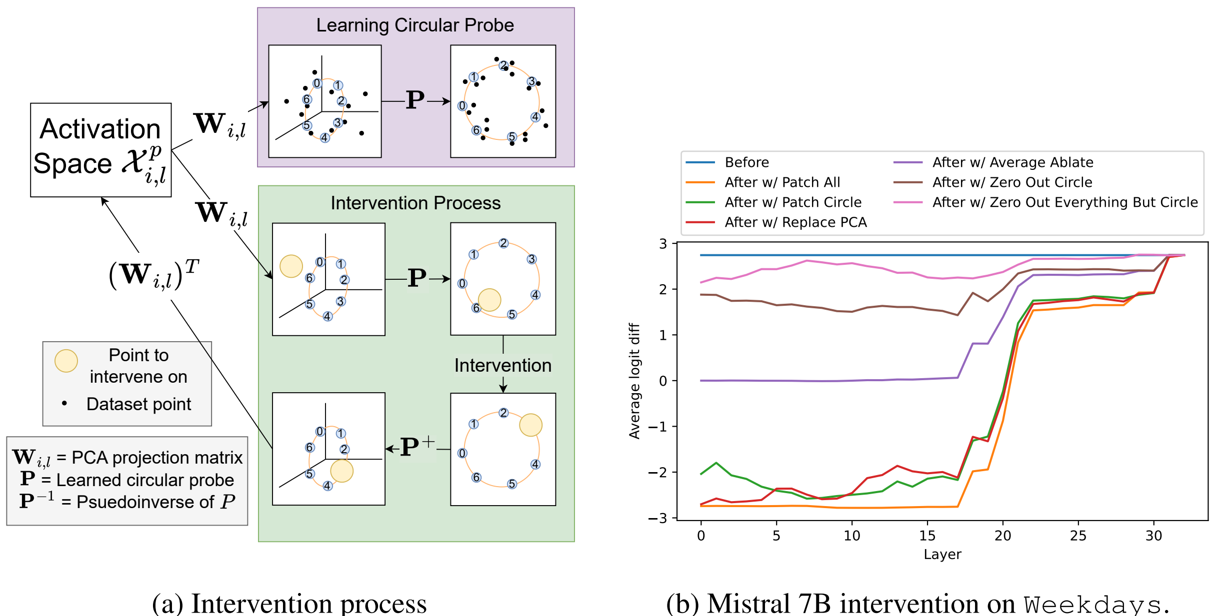
2.5 Circular Representations in Large Language Models

In this section, we aim to find tasks in which models *use* the multi-dimensional features we discovered in the last section, thereby providing evidence that these representations are indeed the fundamental unit of computation for some problems. For simplicity, we design tasks where the answer is a single token. Inspired by the circular representations we found in Section 2.4 and prior work studying circular representations in modular arithmetic (Z. Liu et al. 2022), we define two tasks that represent "natural" modular arithmetic with the following prompts:

`Weekdays` task: *"Let's do some day of the week math. Two days from Monday is"*

`Months` task: *"Let's do some calendar math. Four months from January is"*

For `Weekdays`, we range over the 7 days of the week and durations between 1 and 7 days to get 49 prompts. For `Months`, we range over the 12 months of the year and durations between 1 and 12 months to get 144 prompts. Mistral 7B and Llama 3 8B (AI@Meta 2024) both achieve reasonable performance on the `Weekdays` task and excellent performance on the `Months` task (measured by comparing the highest logit token against the ground truth answer), as summarized in Table 2.1. Interestingly, although these problems are equivalent to modular arithmetic problems $a + b \equiv\ ?\ (\text{mod}\ m)$ for $m = 7, 12$, we were not able to get more than trivial accuracy when using plain modular addition prompts, e.g. "$5 + 3\ (\text{mod}\ 7) \equiv$". Finally, even though we found



(a) Intervention process (b) Mistral 7B intervention on `Weekdays`.

Figure 2.4: Visual representation of the intervention process and intervention results.

circular representations for GPT-2, it failed on every problem instance for both the `Weekdays` and `Months` tasks.

2.5.1 Intervening on Circular Day and Month Representations

To simplify discussion, let α be the day of the week or month of the year token (e.g. “Monday” or “April”), β be the duration token (e.g. “four” or “eleven”), and γ be the target ground truth token the model should predict, such that (abusing notation) we have $\alpha + \beta = \gamma$. Let the prompts of the task be parameterized by j, such that the jth prompt asks about α_j, β_j, and γ_j. We first confirm that Llama 3 8B and Mistral 7B have circular representations of α by examining the PCA of the hidden state at various layers on the α token. We plot two of these in Fig. 2.3 and show all of them in Section 2.B. These plots clearly show circular representations as the highest varying two components in the model’s representation of α at many layers, even in the embedding layer.

We now experiment with *intervening* on these circular representations. We base our experiments on the common interpretability technique of activation patching, which replaces activations from a “dirty” run of the model with the corresponding activations from a “clean” run (Zhang and Nanda

2023). Activation patching empirically tests whether a specific model component, position, and/or representation has a causal influence on the model's output. We employ a custom subspace patching method to allow testing for whether a specific *circular subspace* of a hidden state is sufficient to causally explain model output. Specifically, our patching technique relies on the following steps (visualized in Fig. 2.4a):

1. **Find a subspace with a circle to intervene on**: Using a PCA reduced activation subspace to avoid overfitting, we train a "circular probe" to identify representations which exhibit strong circular patterns. More formally, let $\mathbf{x}_{i,l}^{j}$ be the hidden state at layer l token position i for prompt j. Let $\mathbf{W}_{i,l} \in \mathbb{R}^{k \times d}$ be the matrix consisting of the top k principal component directions of $\mathbf{x}_{i,l}^{j}$. In our experiments, we set $k = 5$. We learn a linear probe $\mathbf{P} \in \mathbb{R}^{2,k}$ from $\mathbf{W}_{i,l} \cdot \mathcal{X}_{i,l}$ to a unit circle in α. In other words, if $\texttt{circle}(\alpha) = [\cos(2\pi\alpha/7), \sin(2\pi\alpha/7)]$ for `Weekdays` and $\texttt{circle}(\alpha) = [\cos(2\pi\alpha/12), \sin(2\pi\alpha/12)]$ for `Months`, $\mathbf{P}$ is defined as follows:

$$\mathbf{P} = \underset{\mathbf{P}' \in \mathbb{R}^{2,k}}{\arg\min} \sum_{\mathbf{x}_{i,l}^{j}} \left[\mathbf{P}' \cdot \mathbf{W}_{i,l} \cdot \mathbf{x}_{i,l}^{j} - \texttt{circle}(\alpha)\right]^2$$

2. **Intervene on the subspace**: Say our initial prompt had $\alpha = \alpha_j$ and we are intervening with $\alpha = \alpha_{j'}$. In this step, we replace the model's projection on the subspace $\mathbf{P}\mathbf{W}_{i,l}$, which will be close to $\texttt{circle}(\alpha_j)$, with the "clean" point $\texttt{circle}(\alpha_{j'})$. Note that this step does not actually use any of the hidden states $\mathbf{x}_{i,l}^{j'}$ from the corresponding "clean" run, only the "clean" label $\alpha_{j'}$. In practice, other subspaces of $\mathbf{x}_{i,l}^{j}$ may be used by the model in alternate pathways to compute the answer. To avoid this affecting the intervention, we average out all subspaces not in the intervened subspace. Letting $\overline{\mathbf{x}_{i,l}}$ be the average of $\mathbf{x}_{i,l}^{j}$ across all prompts indexed by j, we intervene via the formula

$$\mathbf{x}_{i,l}^{j*} = \overline{\mathbf{x}_{i,l}} + {\mathbf{W}_{i,l}}^T \mathbf{P}^{+} (\texttt{circle}(\alpha_{j'}) - \overline{\mathbf{x}_{i,l}})$$

where $\mathbf{P}^{+}$ is the pseudoinverse of $\mathbf{P}$.

We run our patching on α circles for all 49 `Weekday` problems and 144 `Month` problems and use as "clean" runs the 6 or 11 other possible values for β, resulting in a total of $49 * 6$ patching

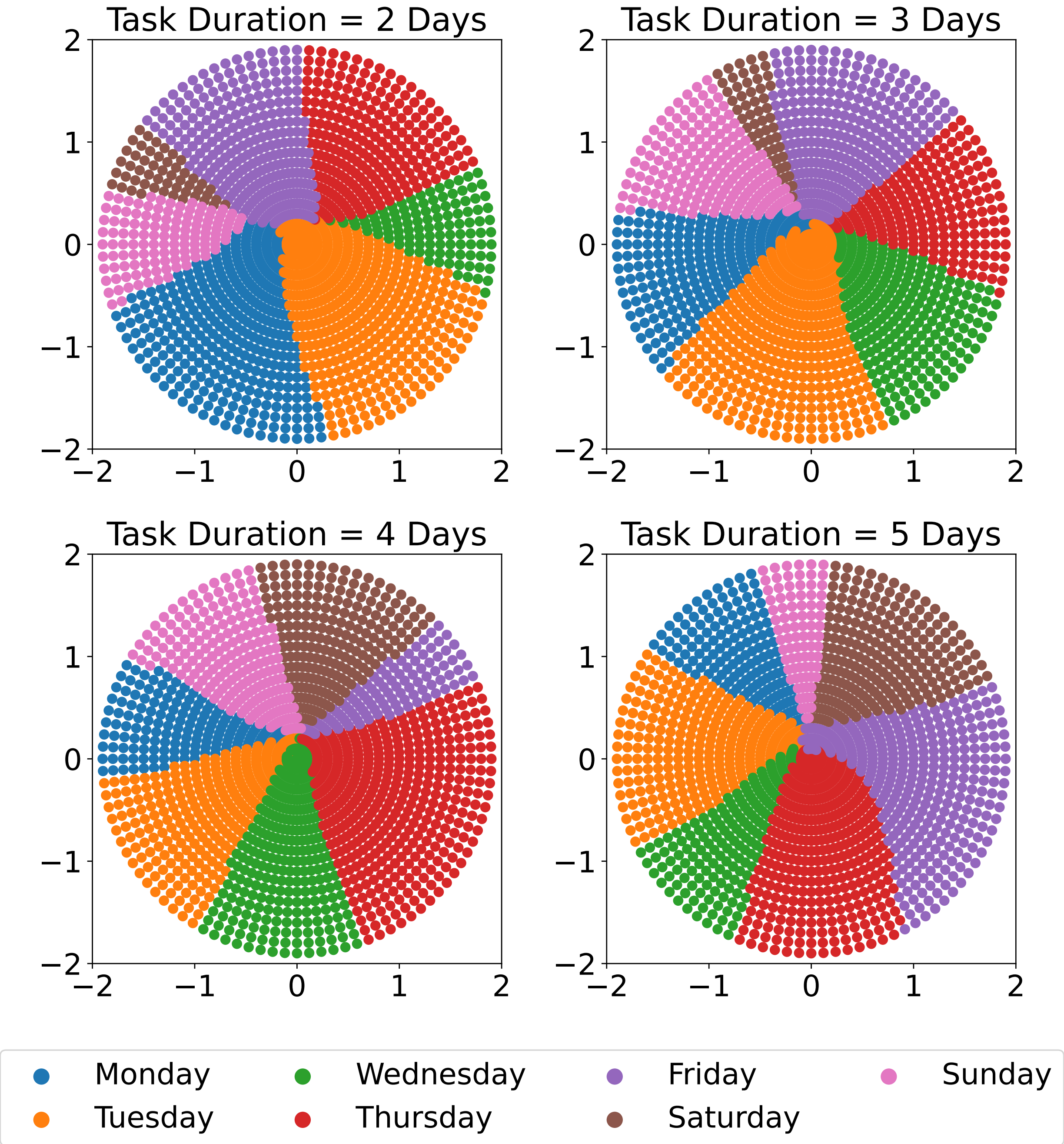


Figure 2.5: Off distribution interventions on Mistral layer 5 on the `Weekdays` task. The color corresponds to the highest logit after performing our circular subspace intervention process on that point.

experiments for `Weekdays` and $144 * 11$ patching experiments for `Months`. As baselines, we also run intervention experiments where we (1) replace the entire subspace corresponding to the first 5 PCA dimensions with the corresponding subspace from the clean run, (2) replace the entire layer with the corresponding layer from the clean run, (3) replace the entire layer with the average across the task, (4) zero ablate the circle subspace, and (5) zero ablate everything *but* the circle subspace. The metric we use is *average logit difference* across all patching experiments between the original correct token and the target token. See Fig. 2.4b for the average logit difference across all layers with Mistral 7B on the `Weekdays` task.

The main takeaway from Fig. 2.4b is that intervening on the circular subspace works! For Mistral 7B, intervening at early layers on the circular subspace is almost as good as patching the entire layer, and is sometimes even better than patching the top 5 PCA dimensions from the corresponding problem. Note that patching experiments in Section 2.B (Fig. 2.10) show that the value of α is copied to the penultimate token on layers 15 to 17, which is why the intervention effect of all methods decreases at around these layers. Moreover, zero ablating the circle plane hurts intervention performance far more than zero ablating everything but the circle plane, further evidence that the main pathway the model uses to compute γ relies on the circular representation of α.

As a final experiment to investigate how the model is using the subspace, we perform an *off distribution* intervention experiment, where instead of intervening on one of the 7 or 12 points on the circumference of the circle that the probe learned, we intervene on a grid of points *in* the circle. We plot the results of this experiment on layer 5 in Mistral 7B with $\beta \in [2, 3, 45]$ in Fig. 2.5. We believe due to these results that the model indeed treats the circle as though it is multi-dimensional representation with α encoded in the angle.

2.5.2 Decomposing Hidden States

To isolate the rough circuit for `Weekdays` and `Months`, we perform activation patching on 20 random problems with the same α and differing β and on 20 random problems with the same β

and differing α. The results, displayed in Section 2.B (Fig. 2.10), show that the circuit to compute γ consists of processing on top of the α and β tokens, followed by a copy to the token before γ, and finally further processing there (this circuit's structure is roughly similar to prior work studying arithmetic circuits (Stolfo, Belinkov, and Sachan 2023)). Moreover, the attention writes from the α and β token are restricted to a few attention layers.

We now introduce a new technique for empirically explaining hidden representations in algorithmic problems: **explanation via regression**. Given an input distribution $\mathcal{T}$ which goes through a model $\mathbf{f} : t \mapsto x$ to produce a distribution of activations $\mathcal{X}$, we try to explain away all the variance in x by adding together hand-computed functions of t, thereby producing an explanation of what $\mathbf{f}$ computes. When we know what hand-computed functions $f_i(t)$ to use, the r^2 value of a linear regression from $f_1(t), f_2(t), \ldots f_k(t)$ to x tells us how much of the variance in x has been explained. But what functions $\mathbf{f}_i$ should we choose?

We believe it is best to build a list of f_i *iteratively* and *greedily*. That is, at each iteration, we perform a linear regression with the current list $f_1 \ldots f_k$, try to visualize and interpret the residual prediction errors, and build a new function f_{k+1} representing these errors, to add to the list. Since $\mathcal{T}$ consists of modular addition problems with two inputs α and β, we can visualize the errors by making a heatmap with α and β on the two axes, where the color shows what kind of error is made. More specifically, we take the top 3 PCA components of the error distribution and assign them to the colors red, green, and blue. We call the resulting heatmap a **residual RGB plot**. Errors that depend primarily on α, β, or γ show up as horizontal, vertical, or diagonal stripes on the residual RGB plot, and signal that functions of α, β, or γ should be added to the list of f_i.

Once most of the variance has been explained, we can reasonably conclude that whatever $f_1, \ldots, f_k$ we have constitutes the entirety of what is represented in the hidden state x, allowing us to even make claims about what is *not* linearly extractable from x. Furthermore, the linear regression coefficients can tell us which directions in $\mathcal{X}$ each of these functions are represented in, if each function were a feature, connecting this technique to Definition 2.3.1 and Hypothesis 2.

Using explanation via regression, we can explain the second to last token's hidden states almost

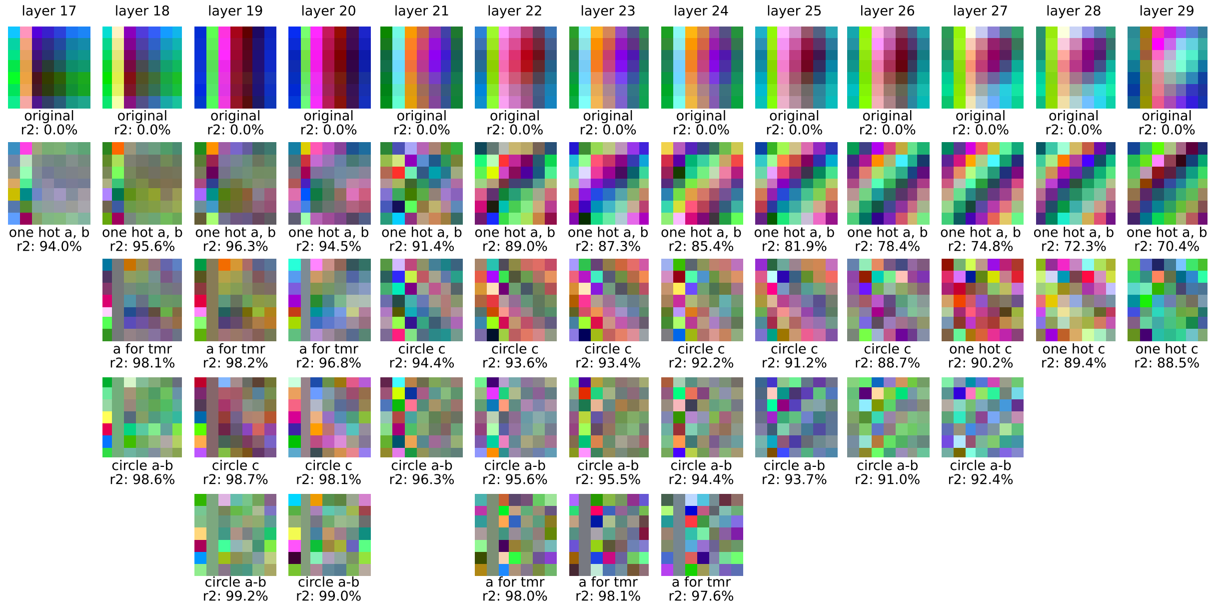


Figure 2.6: Residual RGB plots from explanation via regression, on Mistral hidden states of the token that predicts γ in the `Weekdays` task, from layers 17 to 29. From top to bottom, we show each residual RGB plot after adding the function(s) $\mathbf{f}_i$ labelled just underneath, as well as the resulting r^2 value. We write a, b, c for α, β, γ, and "tmr" meaning "tomorrow" for $\beta = 1$. We also write "circle for x" meaning the inclusion of two functions $\mathbf{f}_i(x) = \{\cos, \sin\}(2\pi x/7)$.

entirely with one hot encodings of α and β (see Section 2.B for these plots). This means that the computation of γ is happening in the MLPs on the last token.

With explanation via regression, we can also remove all linear influence from α and β on the last token's hidden state, to examine what remains. Incredibly, the leftover errors from the regression form a clear circle, which encodes the value of γ. This suggests that the models could be generating γ by using a trigonometry based algorithm like the "clock" (Nanda, Chan, et al. 2023) or "pizza" (Zhong et al. 2024) algorithm.

In Fig. 2.6, we perform explanation via regression on the hidden states between layers 17-29 of Mistral 7B on the `Weekdays` task.

2.6 Broader Impact and Limitations

As machine learning models become more capable, it is increasingly important to understand how they work to ensure they behave safely and reliably. This paper works towards this long term goal by enhancing our understanding of current models. We believe that the insights gained from our investigation into feature structures will facilitate better control, intervention, and ultimately the simplification of complex circuits into formally verifiable programs in future models. We do not anticipate adverse effects from our research, as it focuses solely on deepening our understanding of how language models internally represent concepts.

The main limitation of our work is that we were not able to find a specific small subset of MLP neurons that implement the "clock" algorithm by rotating the circles in α. In preliminary experiments exploring this direction, we found many neurons with large impact on the subspaces which we determined γ was encoded in, so we suspect that if the models use the "clock" algorithm, it is split across multiple MLP neurons. Nevertheless, we believe that the presence of circles encoding α in the first few PCA dimensions is strong evidence that models use these circular representations for many computational tasks.

References

Bubeck, Sébastien et al. (2023). "Sparks of artificial general intelligence: Early experiments with gpt-4". In: *arXiv preprint arXiv:2303.12712*.

Achiam, Josh et al. (2023). "Gpt-4 technical report". In: *arXiv preprint arXiv:2303.08774*.

Anthropic (2024). *The Claude 3 Model Family: Opus, Sonnet, Haiku*. Tech. rep. Anthropic.

Team, Gemini et al. (2023). "Gemini: a family of highly capable multimodal models". In: *arXiv preprint arXiv:2312.11805*.

Olah, Chris et al. (2020). "Zoom in: An introduction to circuits". In: *Distill* 5.3, e00024–001.

Michaud, Eric J et al. (2024). "Opening the AI black box: program synthesis via mechanistic interpretability". In: *arXiv preprint arXiv:2402.05110*.

Liu, Ziming et al. (2022). "Towards understanding grokking: An effective theory of representation learning". In: *Advances in Neural Information Processing Systems* 35, pp. 34651–34663.

Gurnee, Wes and Max Tegmark (2023). "Language models represent space and time". In: *arXiv preprint arXiv:2310.02207*.

Marks, Samuel and Max Tegmark (2023). "The geometry of truth: Emergent linear structure in large language model representations of true/false datasets". In: *arXiv preprint arXiv:2310.06824*.

Heinzerling, Benjamin and Kentaro Inui (2024). "Monotonic Representation of Numeric Properties in Language Models". In: *arXiv preprint arXiv:2403.10381*.

Park, Kiho, Yo Joong Choe, and Victor Veitch (2023). "The linear representation hypothesis and the geometry of large language models". In: *arXiv preprint arXiv:2311.03658*.

Bricken, Trenton et al. (2023). "Towards Monosemanticity: Decomposing Language Models With Dictionary Learning". In: *Transformer Circuits Thread*. https://transformer-circuits.pub/2023/monosemantic-features/index.html.

Mikolov, Tomáš, Wen-tau Yih, and Geoffrey Zweig (2013). "Linguistic regularities in continuous space word representations". In: *Proceedings of the 2013 conference of the north american chapter of the association for computational linguistics: Human language technologies*, pp. 746–751.

Pennington, Jeffrey, Richard Socher, and Christopher D Manning (2014). "Glove: Global vectors for word representation". In: *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pp. 1532–1543.

Mikolov, Tomas et al. (2013). "Distributed representations of words and phrases and their compositionality". In: *Advances in neural information processing systems* 26.

Nanda, Neel, Andrew Lee, and Martin Wattenberg (2023). "Emergent linear representations in world models of self-supervised sequence models". In: *arXiv preprint arXiv:2309.00941*.

Li, Kenneth et al. (2022). “Emergent world representations: Exploring a sequence model trained on a synthetic task”. In: *arXiv preprint arXiv:2210.13382*.

Elhage, Nelson et al. (2022). “Toy Models of Superposition”. In: *Transformer Circuits Thread*. https://transformer-circuits.pub/2022/toy_model/index.html.

Jiang, Yibo et al. (2024). “On the Origins of Linear Representations in Large Language Models”. In: *arXiv preprint arXiv:2403.03867*.

Cunningham, Hoagy et al. (2023). “Sparse autoencoders find highly interpretable features in language models”. In: *arXiv preprint arXiv:2309.08600*.

Kim, Juno and Taiji Suzuki (2024). “Transformers Learn Nonlinear Features In Context: Nonconvex Mean-field Dynamics on the Attention Landscape”. In: *arXiv preprint arXiv:2402.01258*.

Shai, Adam et al. (2024). *Transformers Represent Belief State Geometry in their Residual Stream*. https://www.alignmentforum.org/posts/gTZ2SxesbHckJ3CkF/transformers-represent-belief-state-geometry-in-their.

Black, Sid et al. (2022). “Interpreting neural networks through the polytope lens”. In: *arXiv preprint arXiv:2211.12312*.

Marks, Samuel, Can Rager, et al. (2024). “Sparse Feature Circuits: Discovering and Editing Interpretable Causal Graphs in Language Models”. In: *arXiv preprint arXiv:2403.19647*.

Wang, Kevin et al. (2022). “Interpretability in the wild: a circuit for indirect object identification in gpt-2 small”. In: *arXiv preprint arXiv:2211.00593*.

Conmy, Arthur et al. (2023). “Towards automated circuit discovery for mechanistic interpretability”. In: *Advances in Neural Information Processing Systems* 36, pp. 16318–16352.

Syed, Aaquib, Can Rager, and Arthur Conmy (2023). “Attribution Patching Outperforms Automated Circuit Discovery”. In: *arXiv preprint arXiv:2310.10348*.

Nanda, Neel, Lawrence Chan, et al. (2023). “Progress measures for grokking via mechanistic interpretability”. In: *arXiv preprint arXiv:2301.05217*.

Zhong, Ziqian et al. (2024). “The clock and the pizza: Two stories in mechanistic explanation of neural networks”. In: *Advances in Neural Information Processing Systems* 36.

Hanna, Michael, Ollie Liu, and Alexandre Variengien (2024). "How does gpt-2 compute greater-than?: Interpreting mathematical abilities in a pre-trained language model". In: *Advances in Neural Information Processing Systems* 36.

Stolfo, Alessandro, Yonatan Belinkov, and Mrinmaya Sachan (2023). "A mechanistic interpretation of arithmetic reasoning in language models using causal mediation analysis". In: *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 7035–7052.

Gould, Rhys et al. (2023). "Successor heads: Recurring, interpretable attention heads in the wild". In: *arXiv preprint arXiv:2312.09230*.

Johnson, William B. and Joram Lindenstrauss (1984). "Extensions of Lipschitz mappings into a Hilbert space". In: *Conference in modern analysis and probability (New Haven, Conn., 1982)*. Vol. 26. Contemporary Mathematics. Providence, RI: American Mathematical Society, pp. 189–206. ISBN: 0-8218-5030-X. DOI: 10.1090/conm/026/737400.

Morwani, Depen et al. (2023). "Feature emergence via margin maximization: case studies in algebraic tasks". In: *arXiv preprint arXiv:2311.07568*.

Radford, Alec et al. (2019). "Language Models are Unsupervised Multitask Learners". In.

Bloom, Joseph (2024). *Open Source Sparse Autoencoders for all Residual Stream Layers of GPT2 Small*. https://www.alignmentforum.org/posts/f9EgfLSurAiqRJySD/open-source-sparse-autoencoders-for-all-residual-stream.

Jiang, Albert Q et al. (2023). "Mistral 7B". In: *arXiv preprint arXiv:2310.06825*.

AI@Meta (2024). "Llama 3 Model Card". In: URL: https://github.com/meta-llama/llama3/blob/main/MODEL_CARD.md.

Zhang, Fred and Neel Nanda (2023). "Towards best practices of activation patching in language models: Metrics and methods". In: *arXiv preprint arXiv:2309.16042*.

Higham, Nicholas J. (May 2021). *Singular Value Inequalities*. https://nhigham.com/2021/05/04/singular-value-inequalities/.

Gershgorin, Semyon Aranovich (1931). "Über die Abgrenzung der Eigenwerte einer Matrix". In: *Izvestiya Rossiiskoi akademii nauk. Seriya matematicheskaya* 6, pp. 749–754.

(https://mathoverflow.net/users/2554/bill-johnson), Bill Johnson (n.d.). *Almost orthogonal vectors*. MathOverflow. URL:https://mathoverflow.net/q/24873 (version: 2010-05-16). eprint: https://mathoverflow.net/q/24873. URL: https://mathoverflow.net/q/24873.

Symbol	Description
M	Denotes a specific autoregressive language model.
n	Model context length.
d	Model hidden dimension. Known in the literature as $model_dim$.
$t = (t_1, \ldots, t_n)$	A specific sequence of token inputs to the model.
$\mathcal{T}$	The distribution of t.
$\mathbf{x}_{i,l}$	The residual stream state at token index i and layer l. $\mathbf{x}_{i,l} \in \mathbb{R}^d$
$\mathcal{X}_{i,l}$	Distribution of vectors in feature space induced by input distribution $\mathcal{T}$.
$\mathbf{f}$	A feature, i.e. a function from a subset of supp($\mathcal{T}$) to $\mathbb{R}^{d'}$.
d_f	Used to denote the dimension of a feature $\mathbf{f}$. $d_f \ll d$.
s	Sparsity of a feature $\mathbf{f}$. One minus the total probability that $\mathcal{T}$ assigns to domain$(\mathbf{f})$.
ϵ	Small threshold value used in definition of empricial reducibility.
δ	Measure of orthogonality in superposition hypothesis.
$\texttt{DL}(\mathcal{X}_{i,l})$	Dictionary learning loss function on residual stream distribution $\mathcal{X}_{i,l}$.
m	Dimension of a sparse autoencoder.
$\mathbf{E},\ \mathbf{D}$	Encoder and decoders for a sparse autoencoder. $E \in \mathbf{R}^{m \times d}, D \in \mathbf{R}^{d \times m}$.
α, β, γ	Tokens / quantities for modular addition prompts. $\alpha + \beta = \gamma \pmod 7$ for the `Weekdays` task and $\alpha + \beta = \gamma \pmod{12}$ for the `Months` task.
$\mathbf{W}_{i,l}$	The PCA projection matrix for $\mathcal{X}_{i,l}$.
$\mathbf{P}$	Learned projection for intervention experiments on a circle.

Table 2.2: Notation table, in order encountered in the paper. Matrices are written in capital bold, distributions in caligraphy, and vectors and scalars in lowercase.

2.A Proofs

We will first prove a lemma that will help us prove Theorem 1.

Lemma 2.A.1. *Pick n pairwise δ-orthogonal unit vectors in $\mathbf{v}_1, \ldots, \mathbf{v}_n \in \mathbb{R}^d$. Let $\mathbf{y} \in \mathbb{R}^d$ be a unit norm vector that is a linear combination of unit norm vectors $\mathbf{v}_1, \ldots, \mathbf{v}_n$ with coefficients $z_1 \ldots, z_n \in \mathbb{R}$. We can write $\mathbf{A} = [\mathbf{v}_1, \ldots, \mathbf{v}_n]$ and $\mathbf{y} = [z_1, \ldots, z_n]^T$, so that we have $\mathbf{y} =$*

$\sum_{k=1}^{n} z_k \mathbf{v}_k = \mathbf{A}\mathbf{y}$ *with* $\|\mathbf{y}\|_2 = 1$. *Then,*

$$\left| \sum_{k=1}^{n} z_k \right| = \|\mathbf{y}\|_1 \leq \sqrt{\frac{n}{1-\delta n}}$$

Proof. We will first bound the L_2 norm of $\mathbf{y}$. If σ_n is the minimum singular value of A, then we have via standard singular value inequalities (Higham 2021)

$$\sigma_n \leq \frac{\|\mathbf{y}\|_2}{\|\mathbf{y}\|_2} \implies \|\mathbf{y}\|_2 \leq \frac{\|\mathbf{y}\|_2}{\sigma_n} = \frac{1}{\sigma_n}$$

Thus we now lower bound σ_n. The singular values are the square roots of the eigenvalues of the matrix $\mathbf{A}^T\mathbf{A}$, so we now examine $\mathbf{A}^T\mathbf{A}$. Since all elements of $\mathbf{A}$ are unit vectors, the diagonal of $\mathbf{A}^T\mathbf{A}$ is all ones. The off diagonal elements are dot products of pairs of δ-orthogonal vectors, and so are within the range $[-\delta, \delta]$. Then by the Gershgorin circle theorem (Gershgorin 1931), all eigenvalues λ_i of $\mathbf{A}^T\mathbf{A}$ are in the range

$$(1-\delta(n-1), 1+\delta(n-1))$$

In particular, $\sigma_n^2 = \lambda_n \geq 1-\delta(n-1)$, and thus $\sigma_n \geq \sqrt{1-\delta(n-1)}$. Plugging into our upper bound for $\|\mathbf{y}\|_2$, we have that $\|\mathbf{y}\|_2 \leq 1/\sqrt{1-\delta(n-1)}$. Finally, the largest L_1 for a point on an n-hypersphere of radius r is when all dimensions are equal and such a point has magnitude $\sqrt{n}r$, so

$$\|\mathbf{y}\|_1 \leq \sqrt{\frac{n}{1-\delta(n-1)}} \leq \sqrt{\frac{n}{1-\delta n}}$$

□

Theorem 1. *For any δ, it is possible to choose $e^{\Theta((d/d_{\max}^2)\delta^2)}$ pairwise δ-orthogonal projection matrices $\mathbf{A}_i \in \mathbb{R}^{n_i \times d}$ where $1 \leq n_i \leq d_{\max}$.*

Proof. By the JL lemma (Johnson and Lindenstrauss 1984; (https://mathoverflow.net/users/2554/bill-johnson) n.d.), we can choose $e^{\Theta(d\delta^2)}$ δ-orthogonal unit vectors in $\mathbb{R}^d$ indexed as $\mathbf{v}_i$. Let $\mathbf{A}_i =$

$[\mathbf{v}_{d_{\max}*i}, \ldots, \mathbf{v}_{d_{\max}*i+n_i-1}]$ where each element in the brackets is a column. Then by construction all $\mathbf{A}_i$ are matrices composed of unique δ-orthogonal vectors and there are $\frac{1}{d_{\max}} e^{\Theta(d\delta^2)} = e^{\Theta(d\delta^2)}$ matrices $\mathbf{A}_i$.

Now, consider two of these matrices $\mathbf{A}_i = [\mathbf{v}_1, \ldots, \mathbf{v}_{n_i}]$ and $\mathbf{A}_j = [\mathbf{u}_1, \ldots, \mathbf{u}_{n_j}]$, $i \neq j$; we will prove that they are $f(\delta)$-orthogonal for some function f. Let $\mathbf{y}_i = \sum_{k=1}^{n_i} z_{i,k}\mathbf{v}_k$ be a vector in the colspace of $\mathbf{A}_i$ and $\mathbf{y}_j = \sum_{k=1}^{n_j} z_{j,k}\mathbf{u}_k$ be a vector in the colspace of $\mathbf{A}_j$, such that $\mathbf{y}_i$ and $\mathbf{y}_j$ are unit vectors. To prove $f(\delta)$-orthogonality, we must bound the absolute dot product between $\mathbf{y}_i$ and $\mathbf{y}_j$:

$$
\begin{aligned}
|\langle \mathbf{y}_i, \mathbf{y}_j \rangle| &= \left| \left\langle \sum_{k=1}^{n_i} z_{i,k}\mathbf{v}_k, \sum_{k=1}^{n_j} z_{j,k}\mathbf{u}_k \right\rangle \right| \\
&= \left| \sum_{k_1=1}^{n_i} \sum_{k_2=1}^{n_j} \langle z_{i,k_1}\mathbf{v}_{k_1}, z_{j,k_2}\mathbf{u}_{k_2} \rangle \right| \\
&\leq \sum_{k_1=1}^{n_i} \sum_{k_2=1}^{n_j} |z_{i,k_1} z_{j,k_2}| \, |\langle \mathbf{v}_{k_1}, \mathbf{u}_{k_2} \rangle| && \text{Triangle Inequality} \\
&\leq \sum_{k_1=1}^{n_i} \sum_{k_2=1}^{n_j} |z_{i,k_1} z_{j,k_2}| \delta && \text{All } \mathbf{v}_i, u_j \text{ are } \delta \text{ orthogonal} \\
&= \delta \sum_{k_1=1}^{n_i} \sum_{k_2=1}^{n_j} |z_{i,k_1} z_{j,k_2}| \\
&= \delta \left| \sum_{k=1}^{n_i} z_{i,k} \right| \left| \sum_{k=1}^{n_j} z_{j,k} \right| && \text{Factoring the product} \\
&\leq \delta \sqrt{\frac{n_i}{1-\delta n_i}} \sqrt{\frac{n_j}{1-\delta n_j}} && \text{By Lemma 2.A.1} \\
&\leq \frac{\delta d_{\max}}{1-\delta d_{\max}} && n_i, n_j \leq d_{\max} \text{ by assumption}
\end{aligned}
$$

Thus A_i and A_j are $f(\delta)$-orthogonal for $f(\delta) = \delta d_{\max}/(1-\delta d_{\max})$, and so it is possible to choose $e^{\Theta(d\delta^2)}$ pairwise $f(\delta)$-orthogonal projection matrices. Remapping the variable δ with $\delta \mapsto f^{-1}(\delta) = \delta/(d_{\max}(1+\delta))$, we find that it is possible to choose $e^{\Theta(d\delta^2/((1+\delta)^2 d_{\max}^2))}$ pairwise δ-orthogonal projection matrices. Because $\frac{\delta^2}{(1+\delta)^2}$ is within a factor of 4 to δ^2 on the valid interval of $\delta \in (0,1)$, we can further simplify the exponent and find that it is possible to choose $e^{\Theta((d/d_{\max}^2)\delta^2))}$ pairwise δ-orthogonal projection matrices. □

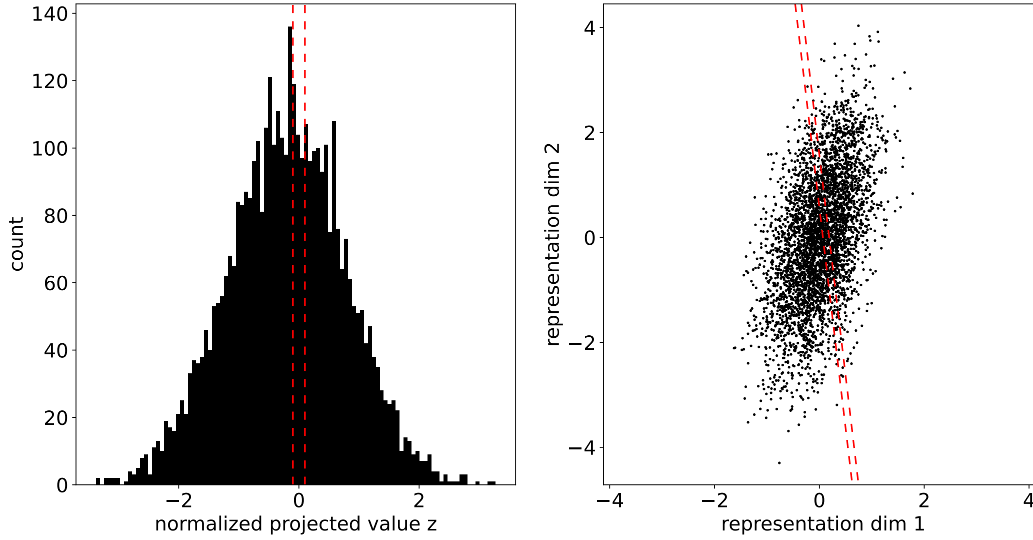


Figure 2.7: An example of testing ϵ-irreducibility on an irreducible feature. **Left:** histogram of the distribution of $\mathbf{v} \cdot \mathbf{f}$. Red lines indicate a 2ϵ-wide region in which we maximize the probability mass. **Right:** The distribution of $\mathbf{f}$. 7.94% of the feature distribution lies within the dotted lines, which is roughly on the order of $\epsilon = 0.1$, indicating that this supposed "feature" is unlikely to be a mixture.

2.B More Plots

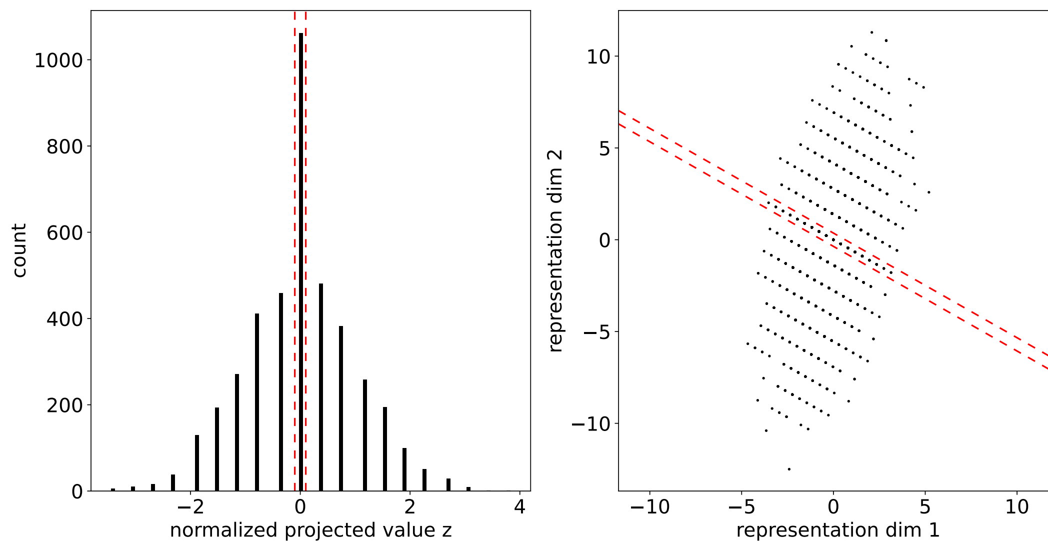
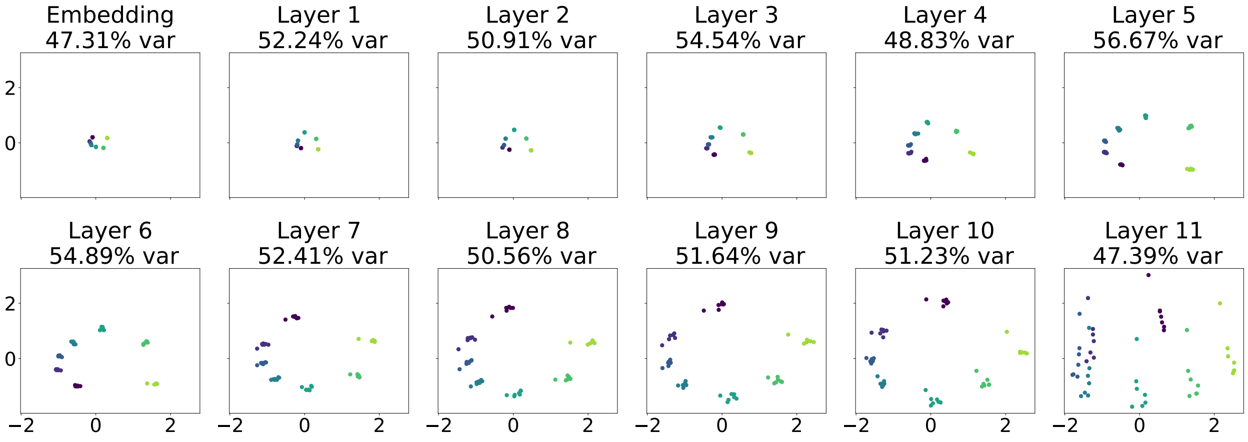
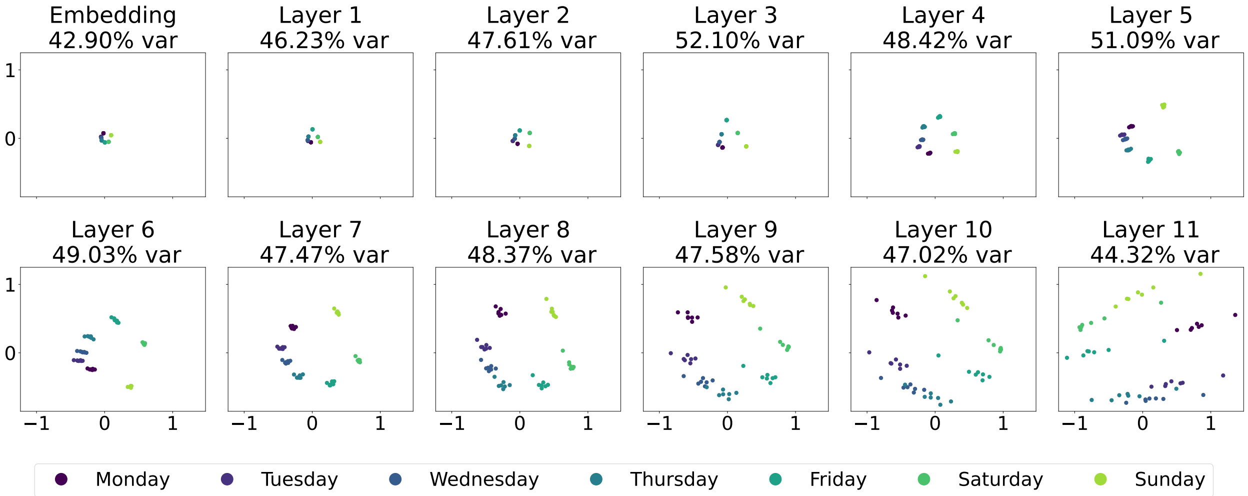


Figure 2.8: An example of testing ϵ-irreducibility on a grid dataset. **Left:** histogram of the distribution of $\mathbf{v} \cdot \mathbf{f}$. Red lines indicate a 2ϵ-wide region in which we maximize the probability mass. **Right:** The distribution of $\mathbf{f}$. 25.90% of the feature distribution lies within the dotted lines, much higher than zero, indicating that this supposed "feature" is actually a mixture.

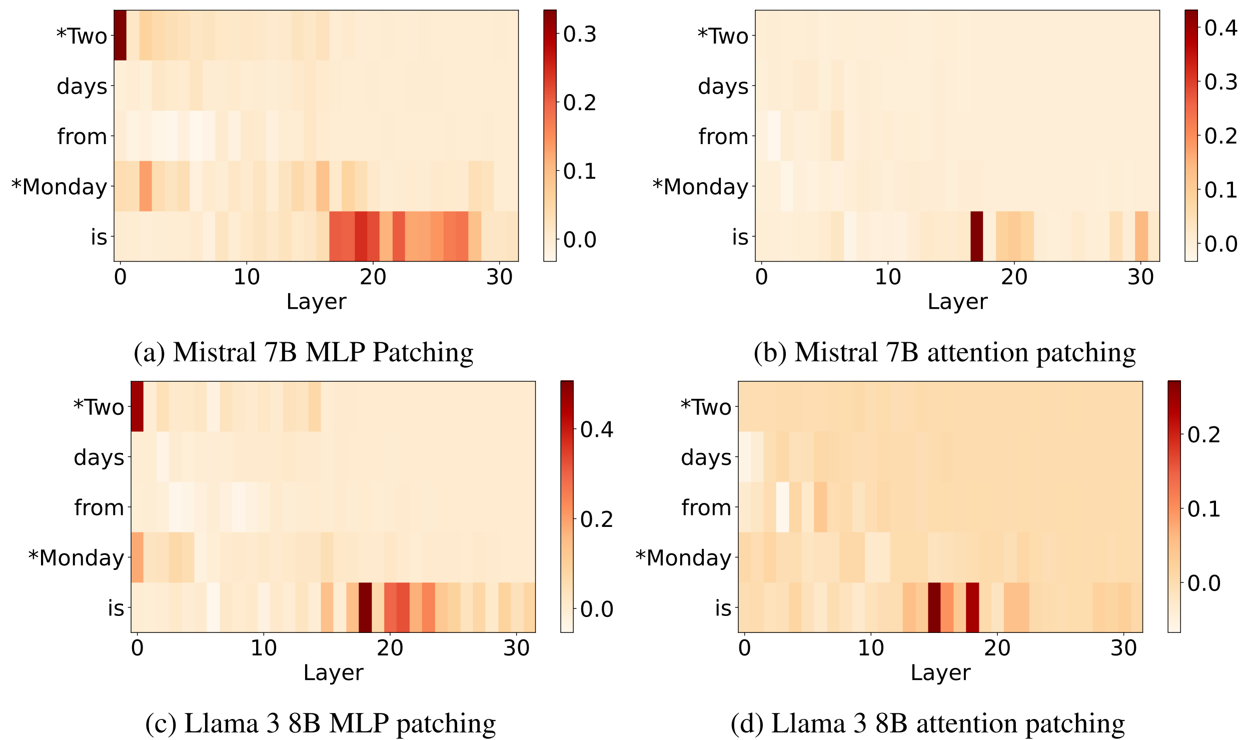


(a) Llama 3 8B



(b) Mistral 7B

Figure 2.9: The top two PCA dimensions of model hidden states on the α token show that circular representations of α are present in various layers.



(a) Mistral 7B MLP Patching (b) Mistral 7B attention patching

(c) Llama 3 8B MLP patching (d) Llama 3 8B attention patching

Figure 2.10: Attention and MLP patching results. Results are averaged over 20 different runs with fixed α and varying β and 20 different runs with fixed β and varying α.

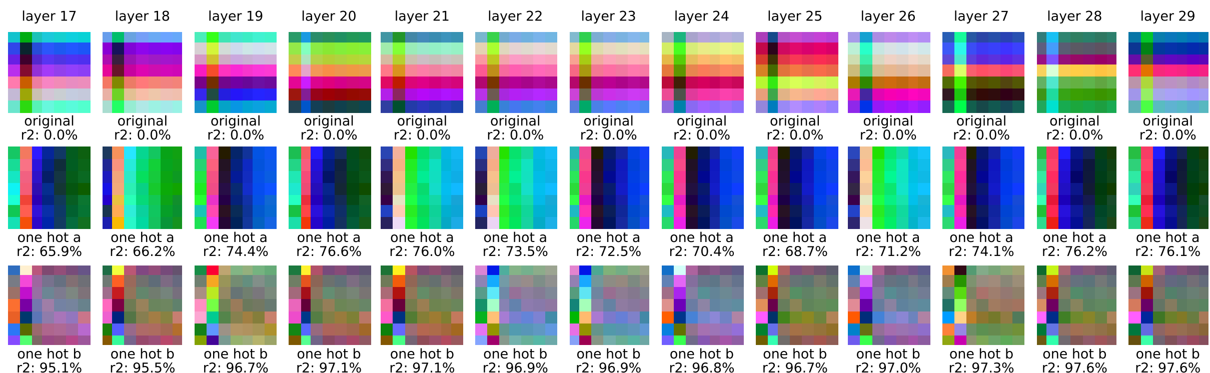


Figure 2.11: Residual RGB plots from explanation via regression, on Mistral hidden states of the token where α is input in the `Weekdays` task, from layers 17 to 29. From top to bottom, we show each residual RGB plot after adding the function(s) f_i labelled just underneath, as well as the resulting r^2 value. We write a, b, c for α, β, γ.

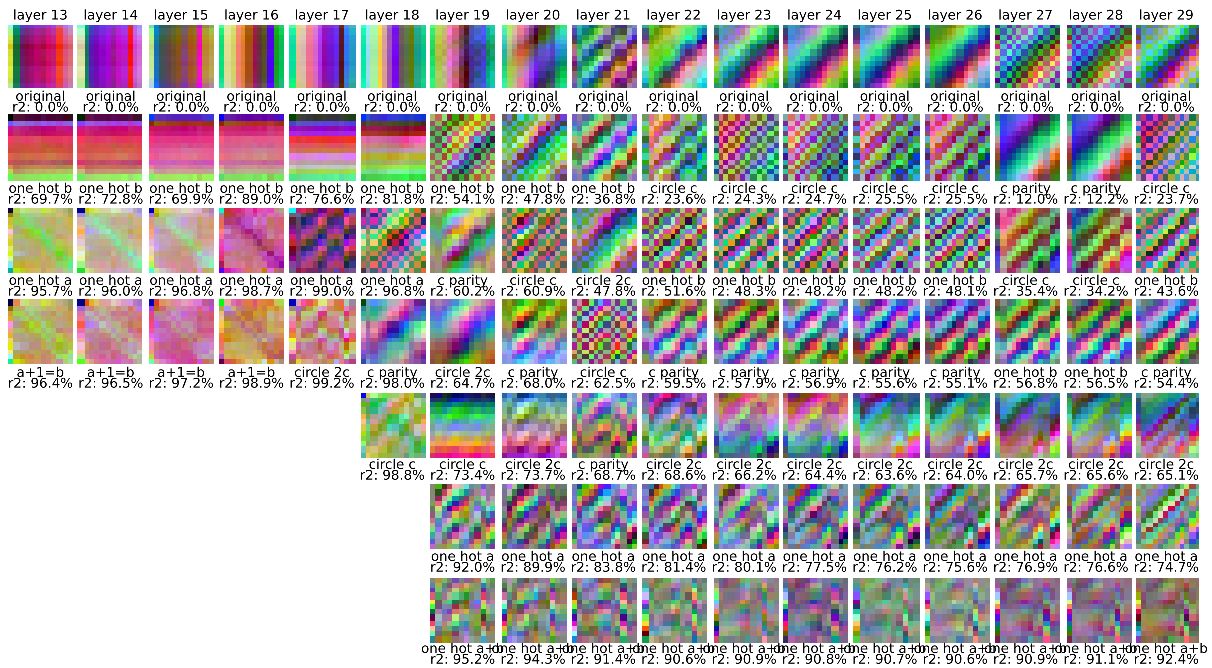


Figure 2.12: Residual RGB plots from explanation via regression, on LLAMA 3 hidden states of the token that predicts γ in the `Months` task, from layers 13 to 29. From top to bottom, we show each residual RGB plot after adding the function(s) f_i labelled just underneath, as well as the resulting r^2 value. We write a, b, c for α, β, γ. We also write "circle for x" meaning the inclusion of two functions $f_i(x) = \{\cos, \sin\}(2\pi x/7)$.

Chapter 3

Generating Interpretable Networks using Hypernetworks

Isaac Liao	**Ziming Liu**	**Max Tegmark**
MIT	MIT	MIT
iliao@mit.edu	zmliu@mit.edu	tegmark@mit.edu

ABSTRACT

An essential goal in mechanistic interpretability to *decode* a network, i.e., to convert a neural network's raw weights to an interpretable algorithm. Given the difficulty of the decoding problem, progress has been made to understand the easier *encoding* problem, i.e., to convert an interpretable algorithm into network weights. Previous works focus on encoding existing algorithms into networks, which are interpretable by definition. However, focusing on encoding limits the possibility of discovering new algorithms that humans have never stumbled upon, but that are nevertheless interpretable. In this work, we explore the possibility of using hypernetworks to generate interpretable networks whose underlying algorithms are not yet known. The hypernetwork is carefully designed such that it can control network complexity, leading to a diverse family of interpretable algorithms ranked by their complexity. All of them are interpretable in hindsight,

although some of them are less intuitive to humans, hence providing new insights regarding how to "think" like a neural network. For the task of computing L1 norms, hypernetworks find three algorithms: (a) the double-sided algorithm, (b) the convexity algorithm, (c) the pudding algorithm, although only the first algorithm was expected by the authors before experiments. We automatically classify these algorithms and analyze how these algorithmic phases develop during training, as well as how they are affected by complexity control. Furthermore, we show that a trained hypernetwork can correctly construct models for input dimensions not seen in training, demonstrating systematic generalization.

3.1 Introduction

Although large language models have demonstrated a number of surprising mathematical and algorithmic capabilities (Yuan et al. 2023; Wei et al. 2022), it remains unknown whether they rediscover algorithms familiar to humans, or if they create more alien forms of mathematics and algorithms that appear less intuitive to humans. This question can be partially answered by recent efforts to mechanistically interpret neural networks (Scherlis et al. 2022; O'Mahony et al. 2023; Schubert et al. 2021; Power et al. 2022; Nanda et al. 2023; Zhong et al. 2023). The holy grail of mechanistic interpretability is to *decode* model weights into interpretable algorithms. This is quite challenging because we have limited clues as to where to look and what to look for. Luckily, the inverse problem, how to *encode* an interpretable algorithm into model weights (Lindner et al. 2023), may shed light on what an interpretable model may look like. Models converted from existing algorithms are by definition interpretable, but their limitations are also obvious: they rule out the possibility of new interpretable algorithms that no human has never stumbled upon (for whatever reason) but are nevertheless interpretable.

This brings up a dilemma of mechanistic interpretability: a trained model is flexible but too uninterpretable, whereas a constructed model is interpretable but too inflexible. This raises the

question of whether there is a way to balance between interpretability and flexibility. Ideally, we hope for models to reveal new algorithms that are undiscovered but which remain within the reach of human understanding. We propose to use hypernetworks (Chauhan et al. 2023) for such a purpose. Intuitively, we find that hypernetworks are well-suited to this task because: (1) hypernetworks can generate "regular patterns of weights", which are similar to the notion of interpretability; and (2) hypernetworks enable control over model complexity, so instead of generating one interpretable network, hypernetworks can generate a diverse family of networks with varying degrees of complexity. [1]

We focus on the simple example task of computing the L_1 norm of a vector. Although this task seems extremely simple and feels fully understood, our hypernetwork is able to generate new algorithms which appear less intuitive to humans yet still remain interpretable with a little overhead of reverse engineering. These new algorithms shed light on how neural networks may do computation or process information in ways that are different from humans. In particular, we identify in our neural networks three types of algorithms for computing the L_1 norm: the double-sided algorithm, the pudding algorithm, and the convexity algorithm (see Figure 3.1), though the authors only expected the double-sided algorithm to be learned before experiments revealed otherwise. By contrast, a conventionally trained network appears to be highly uninterpretable, with no clear patterns in weights or activations (see Figure 3.6). We further define order parameters to auto-classify these algorithms and find intriguing phase transitions between their occurrences, either in training, or when model complexity is varied. By ablating our hypernetwork, we also find that hypernetworks produce the pudding algorithm in two main ways, only one of which is disrupted by the ablation. We also show that a trained hypernetwork can correctly construct models for input dimensions not seen in training, demonstrating algorithmic generalization.

Our results highlight the complexity of mechanistic descriptions even in models trained to perform extremely simple mathematical tasks. We encourage the use of hypernetworks to help future works explore full algorithmic spaces in a controlled and systematic way.

[1] See Appendix 3.A for more discussion.

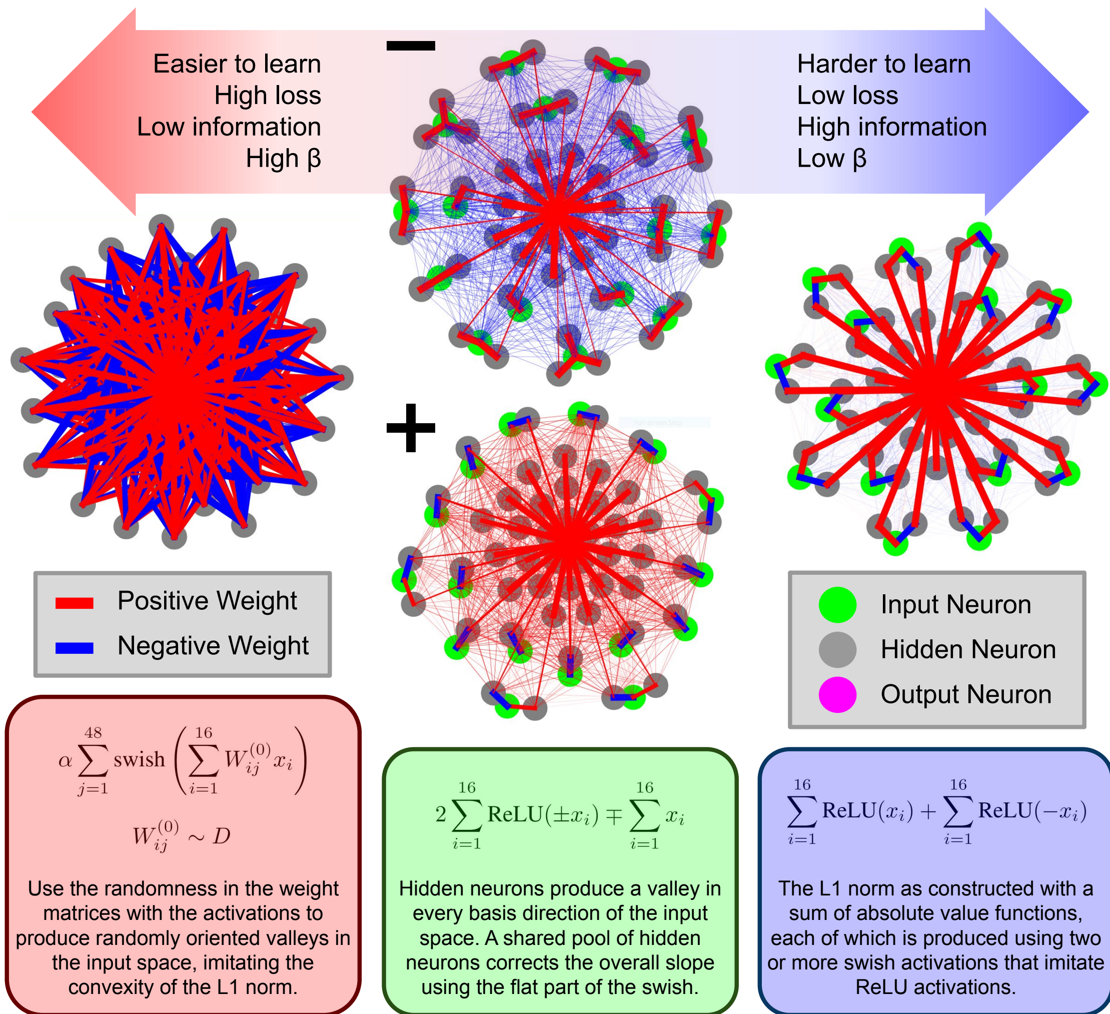


Figure 3.1: Three algorithms for computing the L1 norm, discovered by the hypernetwork. **Left:** the convexity algorithm. **Center top:** the negative pudding algorithm. **Center bottom:** the positive pudding algorithm. **Right:** the double-sided algorithm. The output neuron is in the center of all visualizations. The visualization method is included in Appendix 3.B.1.

3.2 L1 Network Experiments

In this section, we explain how all the networks we trained can compute the L1 norm. All of our networks are two-layer MLPs with 16 inputs, 48 hidden neurons, and 1 output, with swish activation (Ramachandran, Zoph, and Q. V. Le 2017) (Hendrycks and Gimpel 2016). The training data is randomly generated online by sampling the inputs from i.i.d standard normal distribution, and by shifting and rescaling the target L1 norm outputs so that they have zero mean and unit variance.

We train a hypernetwork to generate good weights for this network. Our hypernetwork has a hyperparameter β which controls the balance between the objectives of loss and model complexity (which we measure using a KL divergence). A higher β values simplicity over reducing loss, and a lower β values reducing loss over simplicity. Since our goal is to interpret the generated networks, we mostly treat the hypernetwork as a black box. Details for the internal structure of our hypernetwork are in Appendix 3.B.2. For our purposes, it suffices to remember that a hypernetwork can generate networks whose complexity can be controlled via β. As a result of this training, we have many models saved at various points in training for many β values. Moreover, we tried 33 random seeds, so we have 33 independently sampled copies of all of these models. We find that three algorithms are discovered by the hypernetwork: the convexity algorithm, the pudding algorithm and the double-sided algorithm, shown in Figure 3.1.

3.2.1 Interpretation of Generated Networks

In this section, we deconstruct the algorithms performed by the networks generated by our hypernetworks. We determined these algorithms by looking at force-directed graph drawings of the learned networks (Kobourov 2012). Force-directed graph drawings are a way to visualize graphs of computations by organizing the nodes on the plane of a drawing, to make diagrams of these graphs more intuitive to read. Our force-directed graph drawings assign a position on a drawing to every neuron to minimize an energy consisting of mutual repulsion, connection strength weighted attraction, and central attraction (Bannister et al. 2013). Full details of how we generate these

drawings can be found in Appendix 3.B.1. One can alternatively choose other visualization methods, but we find force-directly graph drawings especially useful since they make symmetries explicit (see Figure 3.1) hence one can simply distinguish different algorithms by noticing their symmetries and other visual traits. By looking at these drawings, we found that networks generally compute the L1 norm using one of three main algorithms:

- **The Double-sided Algorithm** An absolute value function can be constructed with two ReLU neurons, i.e., $|x| = \mathrm{ReLU}(x) + \mathrm{ReLU}(-x)$. It is thus reasonable to expect a neural network to perform L1 computation by summing absolute values of all dimensions [2]:

$$||\boldsymbol{x}||_1 = \sum_{i=1}^{16} \mathrm{ReLU}(x_i) + \sum_{i=1}^{16} \mathrm{ReLU}(-x_i) \tag{3.1}$$

  Indeed, this is one possible algorithm that the hypernetwork produces.

- **The (Signed) Pudding Algorithm** It turns out that there is another method of computing the L1 norm using ReLUs. The hypernetworks that generate the pudding algorithm have learned to take advantage of the following fact:

$$\begin{aligned} ||\boldsymbol{x}||_1 =& 2\sum_{i=1}^{16} \mathrm{ReLU}(\mp x_i) \pm \sum_{i=1}^{16} x_i & (3.2) \\ \approx& \lim_{c\to\infty} 2\sum_{j=1}^{16} \mathrm{ReLU}(\mp x_j \pm \sum_{i=1}^{16} x_i) + \sum_{j=1}^{32} \left(\mathrm{ReLU}(c \pm \sum_{i=1}^{16} x_i) - c \right) & (3.3) \end{aligned}$$

  which holds for both signs $\pm$ (hence the name "signed pudding"), where i iterates through input neurons and j through hidden neurons. The signed pudding algorithm assigns one hidden neuron to each input neuron i to compute the first summation, and uses the leftover hidden neurons to compute the second summation term. Note that only one neuron is actually needed to compute the second term, and so this algorithm can actually be implemented with $n + 1$ hidden neurons, which is more efficient than the $2n$ needed for the double-sided

[2]Note that we are actually using the SiLU activation, but SiLU and ReLU share similar qualitative behavior: a zoomed out plot of a swish function looks like a ReLU.

algorithm. The pudding algorithm is almost always implemented imperfectly, as in Equation 3.3. The hypernetwork uses a hard-coded assignment of pairs of hidden neurons to input neurons; changing the generation seed does not change the order of the hidden neurons. This is the most common algorithm found in our experiments.

- **The Convexity Algorithm** This is an imperfect random algorithm that is easy for the hypernetwork to produce. This algorithm notices that the L1 norm is a convex function, and it tries to match the convexity using randomly oriented swish functions:

$$||\boldsymbol{x}||_1 \approx \alpha \sum_{j=1}^{48} \text{swish} \left(\sum_{i=1}^{16} W_{ij}^{(0)} x_i \right), \quad W_{ij}^{(0)} \sim D \tag{3.4}$$

  with α some constant and D some distribution that is typically symmetric. Sometimes the distribution of D is unimodal, and sometimes it is bimodal.

3.2.2 Order Parameters

Having identified that our experiments mainly consist of three algorithms, we construct a number of order parameters below to distinguish the three algorithms apart from one another.

Double Sidedness: Given the weight matrix $W \in \mathbb{R}^{n_0 \times n_1}$ for the first linear layer (bias not included) where n_0 is the number of input neurons and n_1 is the number of hidden neurons, the double sidedness order parameter α_1 is defined as:

$$\alpha_1 = \frac{\min_i \min(-\min_j W_{ij}, \max_j W_{ij})}{\text{median}_{i,j} \, \text{abs}(W_{ij})} \tag{3.5}$$

The double sidedness measures the degree to which the solution uses the double-sided algorithm.

Strongest Connection: The strongest connection order parameter α_2 is defined as:

$$\alpha_2 = \max_{i,j} \text{abs}(W_{ij}) \tag{3.6}$$

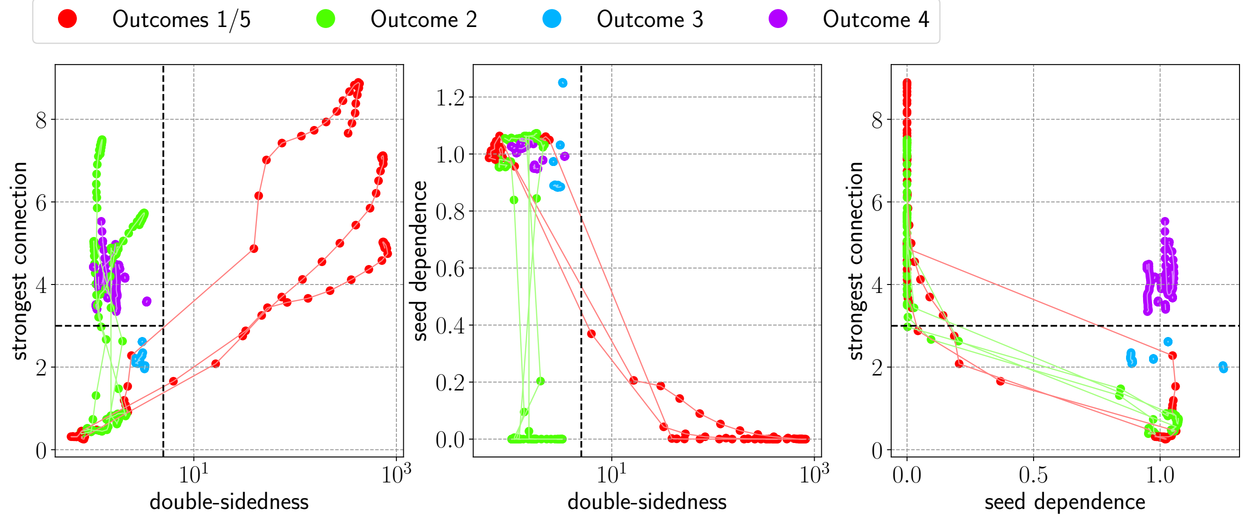


Figure 3.2: Networks that try to compute the L1 norm cluster into three general algorithms in order parameter space, spanned by the "strongest connection", "double-sidedness" and "seed dependence" order parameters. Lines are generated by sweeping β from 10^{-12} to 1 in 30 increments logarithmically. The dotted lines represent hand-picked boundaries which determine when phase transitions between algorithms occur. Lines are grouped and colored by the phases where they start and end at.

A weak strongest connection is indicative of the convexity solution.

Seed Dependence: The seed dependence order parameter α_3 is defined as:

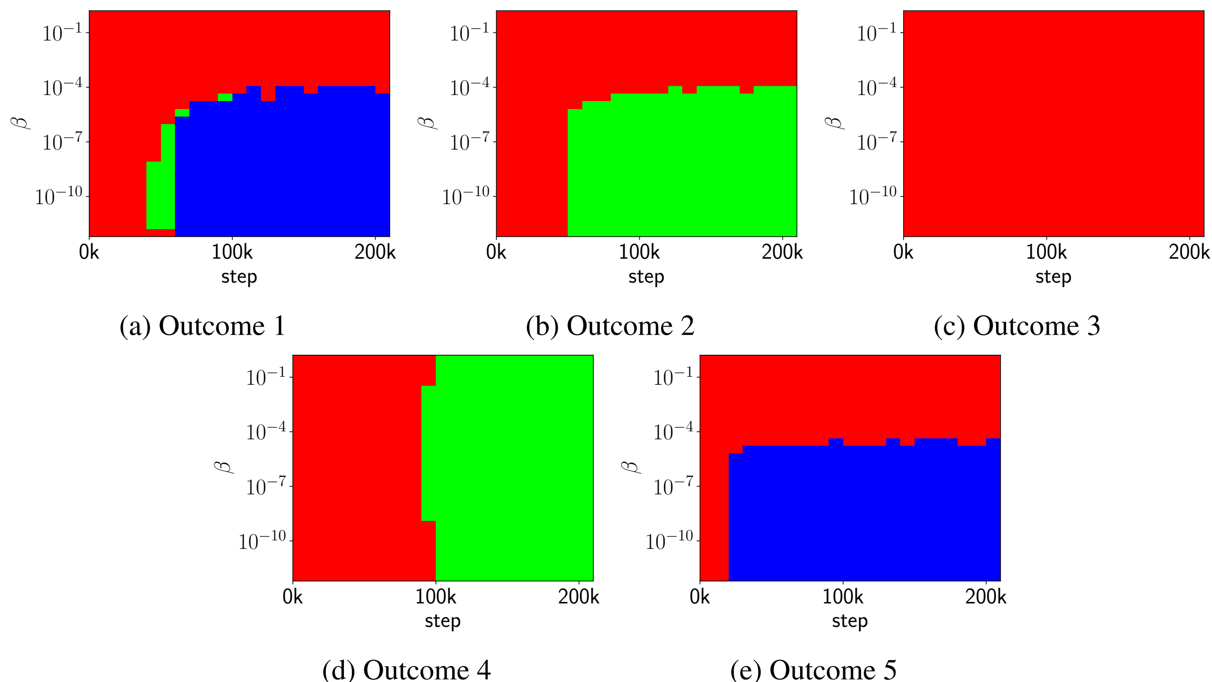
$$\frac{||W - V||_F^2}{||W||_F^2 + ||V||_F^2} \tag{3.7}$$

where W and V are weight matrices for the first linear layer (bias not included) generated with different randomization seeds using the same hypernetwork. A low seed dependence indicates that the hypernetwork is using memorized information about configurations of weights rather than generating this information randomly.

The networks divide themselves roughly into three clusters in order parameter space, each corresponding to one algorithm, as shown in Figure 3.2. While the seed dependence is not immediately relevant since it does not differentiate between algorithms, we will explain later that it can be used to investigate the way that the hypernetwork constructs the pudding algorithm.

3.2.3 Development of Algorithms Throughout Training

Using the order parameters, we automatically classified the algorithms which developed at various points during training for various β values, as in Figure 3.3. We find that the convexity algorithm always develops first, and that other algorithms differentiate away from there. The convexity algorithm can evolve into either the pudding or double-sided algorithms, and the pudding algorithm sometimes also transitions into the double-sided algorithm. Transitions to the pudding algorithm can happen either for only low β or for all β. Oftentimes, the high β regime retains the convexity algorithm while the low β regime evolves though multiple algorithms, and the β value of the transition boundary increases over time and then stabilizes.



(a) Outcome 1 (b) Outcome 2 (c) Outcome 3

(d) Outcome 4 (e) Outcome 5

Figure 3.3: Five ways to develop the three algorithms through training. The horizontal axis is the step number, while the vertical axis is the β parameter used to generate the network. **Red:** convexity algorithm. **Green:** pudding algorithm. **Blue:** double-sided algorithm.

It is worth noting that in Figure (3.3d) the hypernetwork becomes insensitive to β. In this case, the hypernetwork's accumulated KL divergence sums up to nearly zero for all β, causing the Pareto frontier between loss and simplicity to collapse to a single point. This is not the case for the other cases that develop the pudding algorithm; the KL divergence usually increases

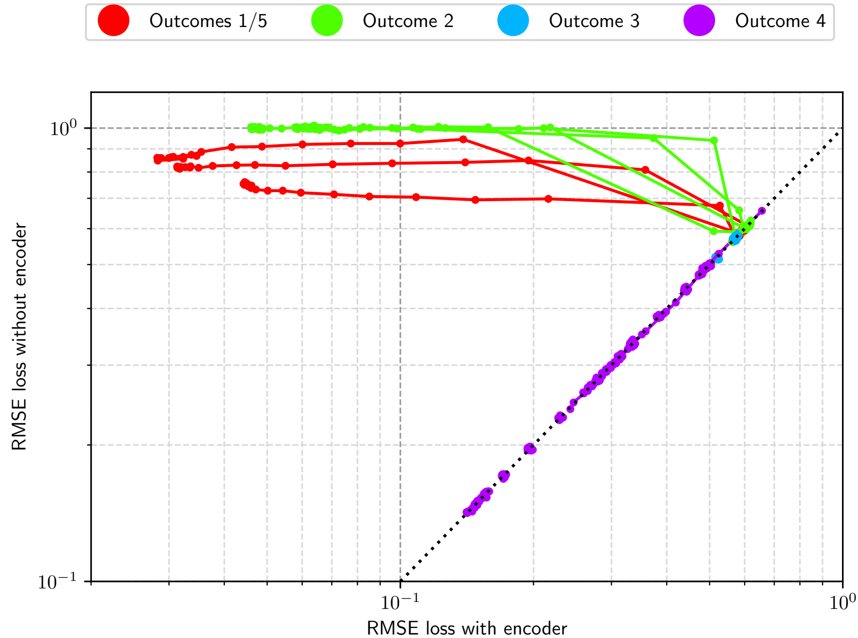


Figure 3.4: Loss of neural network generated when the encoder is either used or ignored. The black dotted line indicates when the performance is unaffected by the presence of the encoder, i.e., when the encoder is unused during the weight generation process. Lines are generated by sweeping β from 10^{-12} to 1 in 30 logarithmic increments.

considerably for lower β. We believe that in case (3.3d), the hypernetwork randomly generates an assignment of hidden neurons to input neurons, while in the other cases, the hypernetwork stores a memorized assignment on the decoder side and passes it through to the encoder for output. Thus, the hypernetwork accumulates KL in all cases except (3.3d). To test this hypothesis, we generated another network using only the decoder side of the hypernetwork without the encoder side[3], and evaluated its loss on the L1 problem again. Removing the encoder should prevent the hypernetwork from using any memorized assignments, without affecting its ability to randomly generate an assignment. Indeed, Figure 3.4 shows that when we remove the encoder, hypernetworks in case (3.3d) are still able to construct working L1 networks that implement the pudding algorithm, but hypernetworks in other cases are not.

[3]by drawing latents from the distribution defined by the decoder side instead of the encoder side

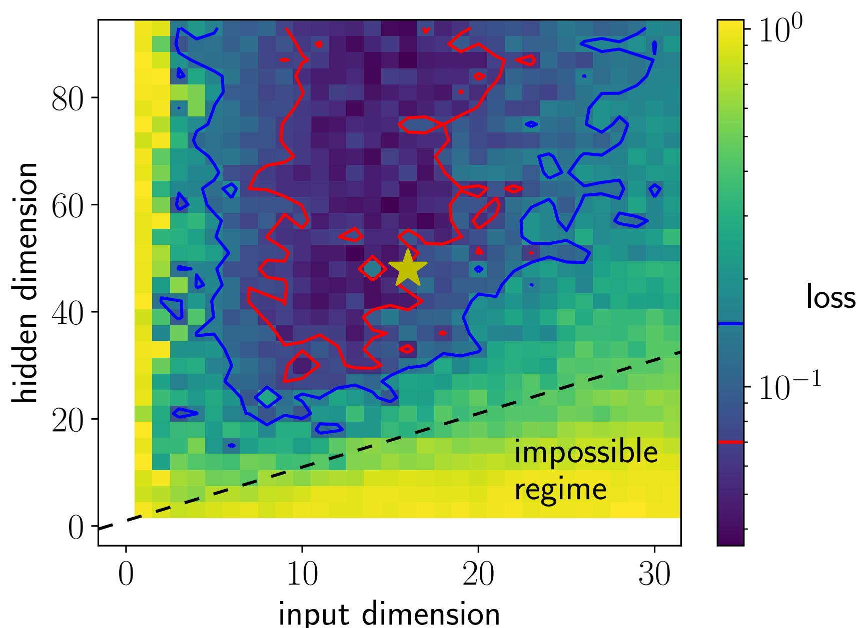


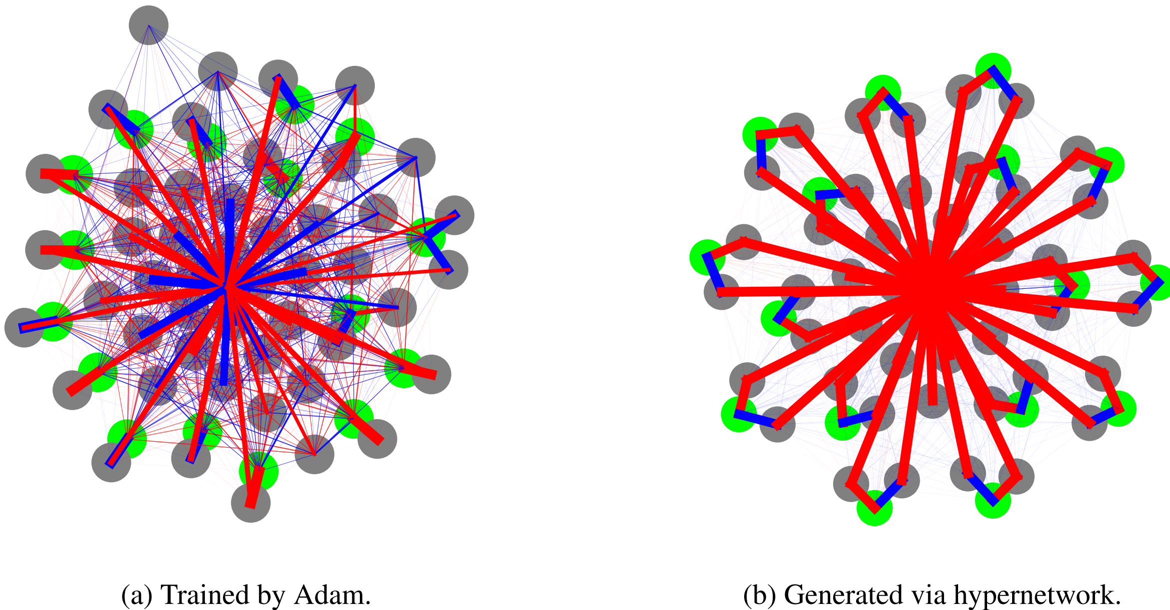
Figure 3.5: Loss of neural network generated with various input dimensions and hidden dimensions using a hypernetwork. The red contour denotes a loss of 0.07, while the blue denotes a loss of 0.15. The hypernetwork was only trained to generate networks with input dimension 16 and hidden dimension 48 (yellow star), yet it can produce networks of diverse shapes which all compute the L1 norm with reasonable accuracy.

3.2.4 Generalization Capabilities

A key feature of the assignment generation hypothesis is that implies hypernetworks in case (3.3d) can generate L1 networks of different layer sizes, because all the configurations of weights are automatically randomly generated instead of being memorized specifically for the $(16, 48, 1)$ layer size structure. We can therefore use an existing hypernetwork to generate networks that compute the L1 norm in a wide range of dimensions, including dimensions larger than 16 which is what the hypernetwork was trained for. The hidden layer size can also be modified to be larger or smaller in the same way. Figure 3.5 shows that many of these L1 networks perform similarly to the original $(16, 48, 1)$ network. We find that there is a region of low loss which extends in the direction of increasing input dimension and hidden dimension, leading us to believe that the hypernetwork has found a general algorithm for computing L1 norms of vectors of any arbitrarily large size, demonstrating systematic generalization.

3.2.5 Baseline Algorithm

As a baseline, we used Adam (Kingma and Ba 2014) to train the same L1 norm network instead of generating the weights using a hypernetwork. The resulting network has a lower loss than the networks generated via hypernetwork, but it is much more difficult to interpret. This is already visible in the cleanliness of the visualization for the hypernetwork in comparison to Adam, as shown in Figure 3.6.



(a) Trained by Adam.

(b) Generated via hypernetwork.

Figure 3.6: Visualization of a $(16, 48, 1)$ neural networks trained to compute the L1 norm of a vector. Input neurons in green, output neurons in magenta. Positive weights in red, and negative weights in blue. The Adam-trained network is messy, whereas the hypernetwork-generated network is much easier to interpret.

Since the Adam-trained L1 norm network is much harder to interpret, we only have a hypothesis about how it computes the L1 norm, which is as follows. In the Adam-trained L1 norm network, hidden neurons are split into two groups: some that are assigned to a single input neuron, and the leftover hidden neurons that are attached to all input neurons. Each input neuron then uses any of its assigned hidden neuron(s) to create a kink of appropriate difference in slope between the positive and negative sides, and the leftover hidden neurons are used to correct the average slope between the two sides, forming an absolute value function for that input neuron. The absolute value functions

for each input neuron are then added together. This is like the pudding algorithm except that there can be many hidden neurons assigned to one input neuron (e.g., 5 hidden neurons, $+ + + - -$) and that the corresponding weights can differ in their sign. Signs for all weights connecting to any given leftover hidden neuron are randomized, and each input neuron connects to every leftover hidden neuron with a different random strength, provided that all the weights satisfy the constraints above. The takeaway is that the network trained by Adam is unnecessarily complex and variable due to the randomness involved, whereas a much simpler and more interpretable solution exists that can be found via hypernetworks.

3.3 Related Work

Hypernetworks Most current hypernetwork research focuses on how we can use hypernetworks to predict different attributes of a model in a particular setting without having to train it in that setting, for example it's loss, accuracy, or trained parameters (Zhang, Ren, and Urtasun 2018) (Knyazev, Drozdzal, et al. 2021). This information can be used to design an architecture with lower loss, skip some training steps (Knyazev, Hwang, and Lacoste-Julien 2023), adjust the parameters based on different loss functions (Navon et al. 2020), and more. Our work instead uses hypernetworks to compress and generate data that comes in the form of neural network weights. Training hypernetworks typically involves computing hypergradients (Baydin et al. 2017), and our work is no exception to this.

Minimum description length The minimum description length (MDL) principle is a mathematical version of Occam's razor that prefers the most compressed explanation for a given dataset (Grünwald 2007; Grünwald and Roos 2019; Rissanen 1978; Solomonoff 2009). The MDL principle can treat a neural network's learning as the process of information compression, so that the complexity of a model is expressed by various KL divergences that form the loss to be minimized (Chaitin 1975; Polyanskiy and Y. Wu n.d.). The MDL principle implies that our hypernetwork is trying to find weights which are as simple as possible for solving the task. The weights' simplicity

makes them easier for humans to pick apart: the resulting model should be very interpretable, and this fact is the main driving force of this paper.

Neural network compression While we were building a compressor to compress neural networks, we noticed that most machine learning systems compress sets of weights with techniques (Tibshirani 1996; Frankle and Carbin 2018; Tan and Q. Le 2019; White et al. 2023; Redmon and Farhadi 2018) that are different to how data from a dataset is typically compressed (Kingma and Welling 2013; Kobyzev, Prince, and Brubaker 2020; Vahdat and Kautz 2020; Sønderby et al. 2016; Child 2020; Vaswani et al. 2017; OpenAI 2023; Touvron et al. 2023). When we tried to use techniques for data compression to compress weights instead, it turned out we were constructing systems that are commonly known as hypernetworks (Chauhan et al. 2023). Our hypernetwork's architecture is based on graph neural networks, self-attention, and deep hierarchical VAEs, all in combination.

Mechanistic Interpretability Research on mechanistic interpretability helps us explain how neural networks operate at the individual neuron level, so that we can understand why they produce certain outputs. This lets us build safer models that we can have better trust in for applications that need this trust. Landmark works in interpretability have included the discovery of polysemantic neurons (Scherlis et al. 2022; O'Mahony et al. 2023), high-low frequency detectors (Schubert et al. 2021), edge detectors, arithmetic representations (Power et al. 2022), modularity (Liu, Gan, and Tegmark 2023), and algorithmic circuits (Nanda et al. 2023; Zhong et al. 2023; Wang et al. 2022).

3.4 Conclusion and Discussion

In this paper, we have introduced a novel hypernetwork-based method for constructing neural network which makes them mechanistically interpretable. We then used this method to construct networks which compute the L1 norm and mechanistically interpreted them.

We found that the hypernetwork-generated L1 norm networks implement three main algorithms for computing the L1 norm, and they represent different tradeoffs between their errors and the

model simplicity. This tradeoff can be manipulated by the β hyperparameter, which controls the relative weight of error vs. simplicity. The algorithm most expected by humans is the double-sided algorithm, which is the hardest to learn and the most accurate. We constructed three order parameters, two of which we use to automatically classify neural networks according to the three algorithms. We find that the three algorithms develop in different β regimes and at different times during training. Namely, the convexity algorithm is the easiest to learn, simplest, and the one that develops at greatest β, followed by the pudding algorithm, followed by the double-sided algorithm. The pudding algorithm is even more simple than the expected double-sided, in that it can be used to compute the L1 norm with fewer hidden neurons, than what the authors originally thought was possible.

There is also value in developing explanations for how the hypernetworks themselves learn to build neural networks. We find that the hypernetworks develop two main methods for constructing networks which operate via the pudding algorithm: one which constructs a random assignment of hidden neurons to input neurons and another which uses a memorized assignment whose information content is penalized. We demonstrate that hypernetworks which construct random assignments can be used to generate working L1 networks to operate on different, sometimes even larger input sizes and hidden dimensions. This works completely in inference time, without any retraining. The hypernetwork's ability to generalize to other input dimensions signals that the hypernetwork has learned an algorithm for computing L1 norms in general for any input vector size, rather than just a circuit that computes an L1 norm with a fixed input size that cannot generalize to other sizes. This work is a preliminary report that showcases the potential of hypernetworks for interpretability research. In the future we hope to generalize the analysis of hypernetwork to more complicated realistic problems.

3.5 Acknowledgements

We would like to acknowledge the MIT SuperCloud and Lincoln Laboratory Supercomputing Center for providing HPC resources that have contributed to the research results reported within this paper. This work was also sponsored in part by the National Science Foundation under Cooperative Agreement PHY-2019786 (The NSF AI Institute for Artificial Intelligence and Fundamental Interactions, http://iaifi.org/) as well as the Beneficial AI Foundation.

References

Yuan, Zheng et al. (2023). "How well do Large Language Models perform in Arithmetic tasks?" In: *arXiv preprint arXiv:2304.02015*.

Wei, Jason et al. (2022). "Emergent abilities of large language models". In: *arXiv preprint arXiv:2206.07682*.

Scherlis, Adam et al. (2022). "Polysemanticity and capacity in neural networks". In: *arXiv preprint arXiv:2210.01892*.

O'Mahony, Laura et al. (2023). "Disentangling Neuron Representations with Concept Vectors". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 3769–3774.

Schubert, Ludwig et al. (2021). "High-Low Frequency Detectors". In: *Distill*. https://distill.pub/2020/circuits/frequer edges. DOI: 10.23915/distill.00024.005.

Power, Alethea et al. (2022). "Grokking: Generalization beyond overfitting on small algorithmic datasets". In: *arXiv preprint arXiv:2201.02177*.

Nanda, Neel et al. (2023). "Progress measures for grokking via mechanistic interpretability". In: *arXiv preprint arXiv:2301.05217*.

Zhong, Ziqian et al. (2023). "The clock and the pizza: Two stories in mechanistic explanation of neural networks". In: *arXiv preprint arXiv:2306.17844*.

Lindner, David et al. (2023). "Tracr: Compiled transformers as a laboratory for interpretability". In: *arXiv preprint arXiv:2301.05062*.

Chauhan, Vinod Kumar et al. (2023). "A Brief Review of Hypernetworks in Deep Learning". In: *arXiv preprint arXiv:2306.06955*.

Ramachandran, Prajit, Barret Zoph, and Quoc V Le (2017). "Searching for activation functions". In: *arXiv preprint arXiv:1710.05941*.

Hendrycks, Dan and Kevin Gimpel (2016). "Gaussian error linear units (gelus)". In: *arXiv preprint arXiv:1606.08415*.

Kobourov, Stephen G (2012). "Spring embedders and force directed graph drawing algorithms". In: *arXiv preprint arXiv:1201.3011*.

Bannister, Michael J et al. (2013). "Force-directed graph drawing using social gravity and scaling". In: *Graph Drawing: 20th International Symposium, GD 2012, Redmond, WA, USA, September 19-21, 2012, Revised Selected Papers 20*. Springer, pp. 414–425.

Kingma, Diederik P and Jimmy Ba (2014). "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980*.

Zhang, Chris, Mengye Ren, and Raquel Urtasun (2018). "Graph hypernetworks for neural architecture search". In: *arXiv preprint arXiv:1810.05749*.

Knyazev, Boris, Michal Drozdzal, et al. (2021). "Parameter prediction for unseen deep architectures". In: *Advances in Neural Information Processing Systems* 34, pp. 29433–29448.

Knyazev, Boris, Doha Hwang, and Simon Lacoste-Julien (2023). "Can We Scale Transformers to Predict Parameters of Diverse ImageNet Models?" In: *arXiv preprint arXiv:2303.04143*.

Navon, Aviv et al. (2020). "Learning the pareto front with hypernetworks". In: *arXiv preprint arXiv:2010.04104*.

Baydin, Atilim Gunes et al. (2017). "Online learning rate adaptation with hypergradient descent". In: *arXiv preprint arXiv:1703.04782*.

Grünwald, Peter (Jan. 2007). *The Minimum Description Length Principle*. ISBN: 9780262256292. DOI: 10.7551/mitpress/4643.001.0001.

Grünwald, Peter and Teemu Roos (2019). "Minimum description length revisited". In: *International journal of mathematics for industry* 11.01, p. 1930001.

Rissanen, J. (1978). "Modeling by shortest data description". In: *Automatica* 14.5, pp. 465–471. ISSN: 0005-1098. DOI: https://doi.org/10.1016/0005-1098(78)90005-5. URL: https://www.sciencedirect.com/science/article/pii/0005109878900055.

Solomonoff, Ray J. (2009). "Algorithmic Probability: Theory and Applications". In: *Information Theory and Statistical Learning*. Ed. by Frank Emmert-Streib and Matthias Dehmer. Boston, MA: Springer US, pp. 1–23. ISBN: 978-0-387-84816-7. DOI: 10.1007/978-0-387-84816-7_1. URL: https://doi.org/10.1007/978-0-387-84816-7_1.

Chaitin, Gregory J. (July 1975). "A Theory of Program Size Formally Identical to Information Theory". In: *J. ACM* 22.3, pp. 329–340. ISSN: 0004-5411. DOI: 10.1145/321892.321894. URL: https://doi.org/10.1145/321892.321894.

Polyanskiy, Yury and Yihong Wu (n.d.). "Information Theory: From Coding to Learning". preprint on webpage at https://people.lids.mit.edu/yp/homepage/papers.html.

Tibshirani, Robert (1996). "Regression Shrinkage and Selection via the Lasso". In: *Journal of the Royal Statistical Society. Series B (Methodological)* 58.1, pp. 267–288. ISSN: 00359246. URL: http://www.jstor.org/stable/2346178 (visited on 10/10/2023).

Frankle, Jonathan and Michael Carbin (2018). "The lottery ticket hypothesis: Finding sparse, trainable neural networks". In: *arXiv preprint arXiv:1803.03635*.

Tan, Mingxing and Quoc Le (2019). "Efficientnet: Rethinking model scaling for convolutional neural networks". In: *International conference on machine learning*. PMLR, pp. 6105–6114.

White, Colin et al. (2023). "Neural architecture search: Insights from 1000 papers". In: *arXiv preprint arXiv:2301.08727*.

Redmon, Joseph and Ali Farhadi (2018). "Yolov3: An incremental improvement". In: *arXiv preprint arXiv:1804.02767*.

Kingma, Diederik P and Max Welling (2013). "Auto-encoding variational bayes". In: *arXiv preprint arXiv:1312.6114*.

Kobyzev, Ivan, Simon JD Prince, and Marcus A Brubaker (2020). “Normalizing flows: An introduction and review of current methods”. In: *IEEE transactions on pattern analysis and machine intelligence* 43.11, pp. 3964–3979.

Vahdat, Arash and Jan Kautz (2020). “NVAE: A deep hierarchical variational autoencoder”. In: *Advances in neural information processing systems* 33, pp. 19667–19679.

Sønderby, Casper Kaae et al. (2016). “Ladder variational autoencoders”. In: *Advances in neural information processing systems* 29.

Child, Rewon (2020). “Very deep vaes generalize autoregressive models and can outperform them on images”. In: *arXiv preprint arXiv:2011.10650*.

Vaswani, Ashish et al. (2017). “Attention is all you need”. In: *Advances in neural information processing systems* 30.

OpenAI (2023). *GPT-4 Technical Report*. arXiv: 2303.08774 [cs.CL].

Touvron, Hugo et al. (2023). “Llama 2: Open foundation and fine-tuned chat models”. In: *arXiv preprint arXiv:2307.09288*.

Liu, Ziming, Eric Gan, and Max Tegmark (2023). “Seeing is Believing: Brain-Inspired Modular Training for Mechanistic Interpretability”. In: *arXiv preprint arXiv:2305.08746*.

Wang, Kevin et al. (2022). “Interpretability in the wild: a circuit for indirect object identification in gpt-2 small”. In: *arXiv preprint arXiv:2211.00593*.

Ying, Chengxuan et al. (2021). “Do transformers really perform badly for graph representation?” In: *Advances in Neural Information Processing Systems* 34, pp. 28877–28888.

Higgins, Irina et al. (2017). “beta-VAE: Learning Basic Visual Concepts with a Constrained Variational Framework”. In: *International Conference on Learning Representations*. URL: https://openreview.net/forum?id=Sy2fzU9gl.

Wu, Lingfei et al. (2022). *Graph Neural Networks: Foundations, Frontiers, and Applications*. Singapore: Springer Singapore, p. 725.

Scarselli, Franco et al. (2009). “The Graph Neural Network Model”. In: *IEEE Transactions on Neural Networks* 20.1, pp. 61–80. DOI: 10.1109/TNN.2008.2005605.

Sanchez-Lengeling, Benjamin et al. (2021). “A Gentle Introduction to Graph Neural Networks”. In: *Distill*. https://distill.pub/2021/gnn-intro. DOI: 10.23915/distill.00033.

Daigavane, Ameya, Balaraman Ravindran, and Gaurav Aggarwal (2021). “Understanding Convolutions on Graphs”. In: *Distill*. https://distill.pub/2021/understanding-gnns. DOI: 10.23915/distill.00032.

Ho, Jonathan, Ajay Jain, and Pieter Abbeel (2020). “Denoising diffusion probabilistic models”. In: *Advances in neural information processing systems* 33, pp. 6840–6851.

3.A Intuition of Why Hypernetworks Work

In this section, we seek to provide a rough understanding of how we use our hypernetwork to generate weights and why we expect the resulting networks to be interpretable. Most important is the fact that the hypernetwork's outputs are used as the weights of the neural network (Chauhan et al. 2023). Altogether, our hypernetwork architecture consists of two graph transformers (Ying et al. 2021) which operate on the computation graph of the target network, serving as the encoder and decoder side of a hierarchical variational autoencoder (HVAE) (Vahdat and Kautz 2020; Sønderby et al. 2016; Child 2020). The weights of the hypernetwork are generated by a Pareto hyperhypernetwork (Navon et al. 2020) which receives the HVAE β hyperparameter (Higgins et al. 2017) as input. The exact details of our hypernetwork and hyperhypernetwork architectures can be found in Appendix 3.B.2. Since our hypernetwork is a merge between a graph transformer and a HVAE, there are multiple lenses through which we understand our hypernetwork design:

The Hypernetwork Interpretation. The hypernetwork is a neural network whose output is used as the weights for a neural network (Chauhan et al. 2023). For input, the hypernetwork is given information about every weight it needs to generate, such as the layer number and indices to identify the input and output neurons that it is connected to. The hypernetwork can then learn a general process for configuring each individual weight depending on its location within the network. Simpler configurations are easier to learn, and thus the resulting networks tend to be simpler, and thus more interpretable.

The GNN Interpretation. A neural network is a computation graph, so the most natural way to manipulate weight data is through a graph neural network (GNN) (L. Wu et al. 2022; Scarselli et al. 2009; Sanchez-Lengeling et al. 2021; Daigavane, Ravindran, and Aggarwal 2021). The hypernetwork is a graph transformer, whereby information is stored at the edges and is sent to and from adjacent nodes where attention heads operate. This allows the hypernetwork to compute based on how weights are mutually related to one another within the architecture (which itself does not need to be fixed either). This allows the hypernetwork to form structures of weights which are

connected to the way the architecture is structured and are thus more likely to be interpretable.

The MDL/Compression Interpretation. Our hypernetwork is an application of the minimum description length (MDL) principle, which treats learning as a data compression procedure described by a mathematical version of Occam's razor (Grünwald 2007; Grünwald and Roos 2019; Rissanen 1978; Solomonoff 2009). The MDL principle treats a neural network as a compressor, so that we can speak of the "Occam simplicity" of a model through KL divergences, which measure how well that model compresses a dataset in an information theoretic sense. (Chaitin 1975; Polyanskiy and Y. Wu n.d.) For our case, the dataset consists of the neural network weights and the compressor is the hypernetwork; the hypernetwork is an encoding/decoding system for compressing neural network weights into as simple a latent representation as possible, and networks derived from simpler representations are more interpretable.

The Generative Model Interpretation. Our hypernetwork is a generative model for data in the form of neural network weights. In the past, generative models were developed for and have been successful in generating human-interpretable forms of data, such as natural language (OpenAI 2023; Touvron et al. 2023) and images (Ho, Jain, and Abbeel 2020). Thus, we may expect that a generative model for data in the form of neural network weights would naturally have an inductive bias for human-interpretable weights.

3.B Method

3.B.1 Force-Directed Graph Drawings

Throughout this paper, we use a force-directed graph drawing algorithm to visualize neural networks as computation graphs. Force-directed graph drawing algorithms try to position every node in the plane to minimize clutter and account for several visual quality measures, such as edge overlaps, drawing area, and symmetry. Typically, such an algorithm defines an energy consisting of a sum of several components which each depend on the node positions, and the node positions are adjusted to minimize this energy via gradient descent. In our case, we apply three components: $1/r$ pairwise

repulsion between all neurons, kr^2 pairwise attraction for weights of absolute value k, and r^2 attraction pulling all neurons towards the origin.

This graph drawing algorithm will help us to observe the structure of weights in the neural networks which we train in this paper, so that we may more easily understand how they function. Most importantly, it allows us to observe modularity, which is when individual parts of a learned neural network compute their operations individually without interaction. This is because the modules may form separate connected components which stay self-connected but repel each other in the drawn graph, making components easy to identify.

One of the main issues with force-directed graph drawing is that the gradient descent often falls into local minima of the energy, since two separate modules in the network can become tangled up, after which point the modules can no longer slide past one another and separate properly. To remedy this, we position the nodes in four dimensions instead of two (this provides mode connectivity), and we apply an increasingly strong decay to the two excess dimensions during gradient descent until they disappear, leaving a fully two-dimensional arrangement. This arrangement can be rescaled as needed for the visualization.

3.B.2 Attentional Hypernetworks

In this section, we explain in more detail how we use a hypernetwork to generate the weights of the MLP which we would like to train. Instead of learning the MLP weights directly, we learn the "hyperweights" of this hypernetwork. We generate the MLP weights every iteration as part of the forward pass when doing gradient descent. Figure 3.7 fully depicts all the components of the hypernetwork, and in the following text and sections below, we will explore each of the components in detail.

We design the hypernetwork in a way so that it has an inductive bias to create certain structures and formations of weights with more ease than others. For example, we may want the hypernetwork to tell the weights how to self-organize into many duplicates of a specific circuit, which are then connected together in a formation, rather than many unrelated circuits connected in a more complex

manner. Given that these structures are organized, with a limited number of hyperweights, it might be easier to learn a method of constructing such structures rather than the structures themselves. We believe that and inductive bias for learning such methods should best arise from clever forms of parameter reuse, just as convolutional filters are best for translationally equivariant computations. As such, our hypernetwork operates much like a graph transformer, whose computations are duplicated across all nodes and edges.

For every component of the MLP, there is an analogous component for our hypernetwork, which itself can somewhat be thought of like an MLP. For example, the hypernetwork has hyperfeatures, hyperlayers, hyperwidth, hyperdepth, and hyperactivations in the same way that the MLP (or "network") has features, layers, width, depth, and activations.

For our network architecture, we restrict ourselves to a neural network consisting of weights $W^{(\ell)} \in \mathbb{R}^{n_{i+1} \times n_i}$ and biases $b^{(\ell)} \in \mathbb{R}^{n_{i+1}}$ with N layers:

$$a_j^{(\ell+1)} = \sigma(b_j^{(\ell)} + \sum_i W_{ij}^{(\ell)} a_i^{(\ell)})$$

with $a_i^{(1)}$ the input vector and $b_j^{(N)} + \sum_i W_{ij}^{(N)} a_i^{(N)}$ the output vector.

We will now describe our hypernetwork architecture. The fundamental unit of data processed by the hypernetwork is a high-dimensional ragged tensor of "hyperactivations" containing information about every weight in the network. Each "hyperlayer" operates by applying many operations which serve to perform computations that only conduct information along individual axes of the tensor. For example, if we have a hyperactivation tensor pertaining to a 4 dimensional CNN filter tensor with an x axis, we might apply a blur filter along the corresponding x axis of the hyperactivation tensor while treating all other dimensions as batch dimensions, duplicating this operation for all indices.

Returning from the CNN example to our MLP network, we designate the last axis as special, and we call it the "hyperfeature" dimension, analogous to the feature dimension in the MLP. The hyperactivation tensor is sliced into many blocks along the hyperfeature axis, and each block under-

goes its own operation along its own designated axis (all other axes treated as batch dimensions), then the results are concatenated back together in the hyperfeature dimension. A linear operation in the hyperfeature axis can then be used to control the movement of data between these individual blocks, allowing for movement of data in every direction of the tensor. A good analogy is the way that trains can be shunted onto different tracks, where they may travel in different directions or undergo different procedures. This all can then be repeated multiple times by stacking together multiple of these hyperlayers. We will index hyperlayers using the variable ℓ' since we are indexing layers with ℓ, and let us call the number of hyperlayers the "hyperdepth" N', which we set to 4.

We will now roughly describe how we construct a hyperactivation tensor from the MLP weights. We can summarize the MLP weights using a ragged tensor $W_{ij}^{(\ell)}$ indexed by input neuron i, output neuron j, and layer ℓ. The hyperactivation tensor then has the same shape, except that it has an additional hyperfeature axis indexed by a variable i'.

The operations performed on blocks of the hyperactivation tensor are treated like activation functions for the hypernetwork. For a hyperactivation tensor with indices i, j, ℓ, and i', a block is processed with each of the following operations:

- $x \rightarrow \sigma(x)$. Elementwise activation function. Used on a block of 20 hyperfeatures.
- Nothing $\rightarrow$ Positional encoding of i if $\ell = 1$, else a zero tensor.
- Nothing $\rightarrow$ Positional encoding of j if $\ell = N$, else a zero tensor.
- Nothing $\rightarrow$ Positional encoding of ℓ.
- Nothing $\rightarrow$ 5 hyperfeatures of random i.i.d. samples from a standard normal distribution.
- A self-attention head for every neuron in the network. Each edge feeds 3 sub-blocks of the hyperactivation tensor—the keys, queries, and values—to the neuron in front and another 3 sub-blocks to the neuron behind, and concatenates the results received from both sides. The keys, queries, and values are all 5 hyperfeatures in size.

The output of the final hyperlayer has two hyperfeatures—one is used as the weight tensor and the other is averaged along the input neuron axis i and is used as the bias tensor.

Notice that this hypernetwork architecture as it stands does not contain that many learned parameters, compared to how many MLP parameters it can generate. Even with so few parameters, it still takes a huge amount of computation, which is necessary for the amount of parameter reuse we want. Remember that parameter reuse is useful for the hypernetwork to produce highly patterned structures in the network's weights.

KL Attentional Hypernetworks

Plain attentional hypernetworks like the one described above have a certain defect: there might be some networks that are learned more easily by gradient descent, but which are hard for the hypernetwork to capture because they are not structured in any obvious fashion. To allow the attentional hypernetwork to capture these circuits, we introduce another axis to the hyperactivation tensor, which we call the KL axis.

The KL axis has two indices, representing the encoder and decoder side of an information channel. Information is passed through this axis via two operations:

- Nothing $\rightarrow$ 5 hyperfeatures of learned variables which go along with the hyperweights during training. These hyperfeatures are intended to capture information that can be learned by standard gradient descent, but they are only given on the encoder side and are set to zero for the decoder side.

- Start with two blocks of 4 hyperfeatures representing μ and σ values for normal distributions. The total KL divergence $D_{\text{KL}}(q||p)$ between the encoder-side and decoder-side distributions q and p is computed and added to an accumulator variable. The output is 4 hyperfeatures of samples from q; a copy of these samples for each of the encoder and decoder sides to use.

The weights and biases are constructed using only the output from the decoder side.[4]

[4]By introducing the KL axis, we have essentially turned our hypernetwork into a kind of conditional hierarchical VAE.

In this additional axis, learning via standard gradient descent is allowed to take place, since the hypernetwork can pass the learned variables provided to the encoder through the KL channel to the decoder, where the variables can be output to be treated as weights and biases. But notice that any information passing through the KL channel has its information content measured and accounted for in an accumulator variable. This means we can suppress the usage of gradient descent algorithms by regularizing the quantity of raw weight information which passes through. [5] Tuning the suppression factor β allows us to control the balance between a hypernetwork which only designs very structured patterns of weights and a hypernetwork which regurgitates unstructured patterns of learned and memorized weights. The balance between loss and KL implicitly encourages the neural network to develop structures like modules and duplicated circuits of neurons, as these are structures that the hypernetwork can encode in a more compressed manner and will be penalized less for.

Hyperhypernetworks for Multi-Objective Optimization

We are now left with a multi-objective optimization problem where we would like to jointly optimize for the network's loss and hypernetwork's total accumulated KL. To solve this, we use a hyperhypernetwork to generate the hyperweights in every step during the forward pass, using something like a Pareto Hypernetwork. This hyperhypernetwork takes $\log \beta$ (rescaled and shifted) as input, has two hidden layers of size 100 and 10 with swish activation, and outputs two vectors a and b where $a\sigma(b)$ is treated as the flattened vector of hyperweights. At every iteration, we sample β from a distribution, use the hyperhypernetwork to generate the hyperweights, use the hypernetwork to generate the weights, and use $\log(L + \beta D_{\text{KL}})$ as the objective, where L is the network's loss and D_{KL} is the hypernetwork's accumulated KL.

[5] Note that the feedback in the channel lets the hypernetwork perform simpler computations without destroying the ELBO bound in the VAE interpretation of our hypernetwork.

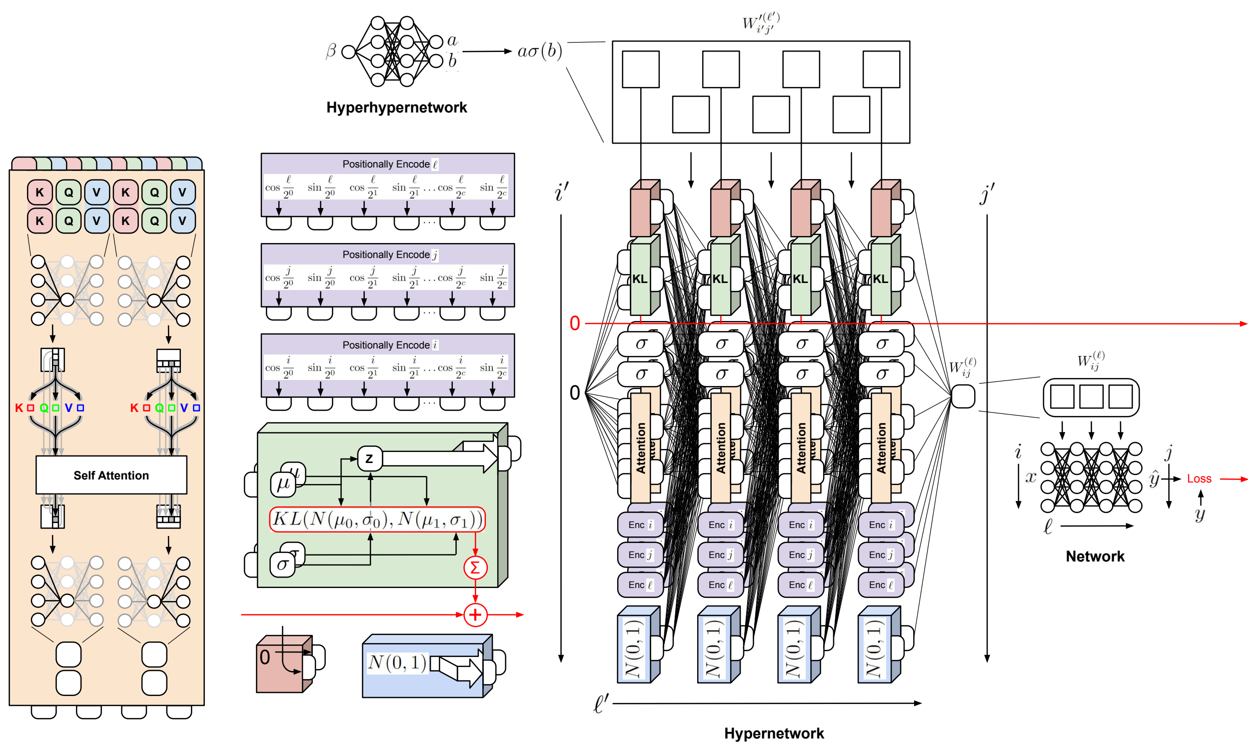


Figure 3.7: The full architecture of the hypernetwork. The hyperhypernetwork above generates weights for the hypernetwork, which generates weights for the network, on which the loss is evaluated. There are many components in the hypernetwork, each drawn individually on the left: graph attention, positional encodings, information bottleneck channels, learned hyperfeatures, and random variables.

Chapter 4

Opening the AI black box: Program Synthesis via Mechanistic Interpretability

Eric J. Michaud[*23] **Isaac Liao**[*2] **Vedang Lad**[*2] **Ziming Liu**[*23]
Anish Mudide[2] **Chloe Loughridge**[4] **Zifan Carl Guo**[2] **Tara Rezaei Kheirkhah**[2]
Mateja Vukelić[2] **Max Tegmark**[23]

Abstract

We present MIPS, a novel method for program synthesis based on 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

[*]Equal contribution.
[2]Massachusetts Institute of Technology, Cambridge, MA, USA
[3]Institute for Artificial Intelligence and Fundamental Interactions
[4]Harvard University, Cambridge, MA, USA

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.

4.1 Introduction

Machine-learned algorithms now outperform traditional human-discovered algorithms on many tasks, from translation to general-purpose reasoning. These learned algorithms tend to be black-box neural networks, and we typically lack a full understanding of how they work. This has motivated the growing field of *mechanistic interpretability*, aiming to assess and improve their trustworthiness. Major progress has been made in interpreting and understanding smaller models, but this work has involved human effort, which raises questions about whether it can scale to larger models. This makes it timely to investigate whether mechanistic interpretability can be fully automated (Tegmark and Omohundro 2023).

The goal of the present paper is to take a modest first step in this direction by presenting and testing MIPS (**M**echanistic-**I**nterpretability-based **P**rogram **S**ynthesis), a fully automated method that can distill simple learned algorithms from neural networks into Python code. The rest of this paper is organized as follows. After reviewing prior work in Section II, we present our method in Section III, test it on a benchmark in Section IV and summarize our conclusions in Section V.

4.2 Related Work

Program synthesis is a venerable field dating back to Alonzo Church in 1957; Zhou and Ding (2023) and Odena et al. (2020) 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 (Wu et al.

2023). 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 (Sobania, Briesch, and Rothlauf 2022). 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 (Olah et al. 2020; Cammarata et al. 2020; Wang et al. 2022; Olsson et al. 2022). Much MI work has tried to understand how neural networks represent various types of information, *e.g.*, geographic information (Goh et al. 2021; Gurnee and Tegmark 2023), truth (Burns et al. 2022; Marks and Tegmark 2023) and the state of board games (McGrath et al. 2022; Toshniwal et al. 2022; Li et al. 2022). Another major MI thrust has been to understand how neural networks perform algorithmic tasks, *e.g.*, modular arithmetic (Nanda et al. 2023; Z. Liu et al. 2022; Zhong et al. 2023; Quirke et al. 2023), greater-than (Hanna, O. Liu, and Variengien 2023), and greatest-common-divisor (Charton 2023).

Whereas Lindner et al. (2023) automatically convert traditional code into a neural network, we aim to do the opposite. Other recent efforts to automate MI include identifying a sparse subgraph of the network whose units are causally relevant to a behavior of interest (Conmy et al. 2023; Syed, Rager, and Conmy 2023) and using LLMs to label internal components of neural networks, for instance neurons (Bills et al. 2023) and features discovered by sparse autoencoders (Cunningham et al. 2023; Bricken et al. 2023).

4.3 MIPS, our program synthesis algorithm

As summarized in Figure 4.1, MIPS involves the following key steps.

1. Neural network training
2. Neural network simplification
3. Finite state machine extraction

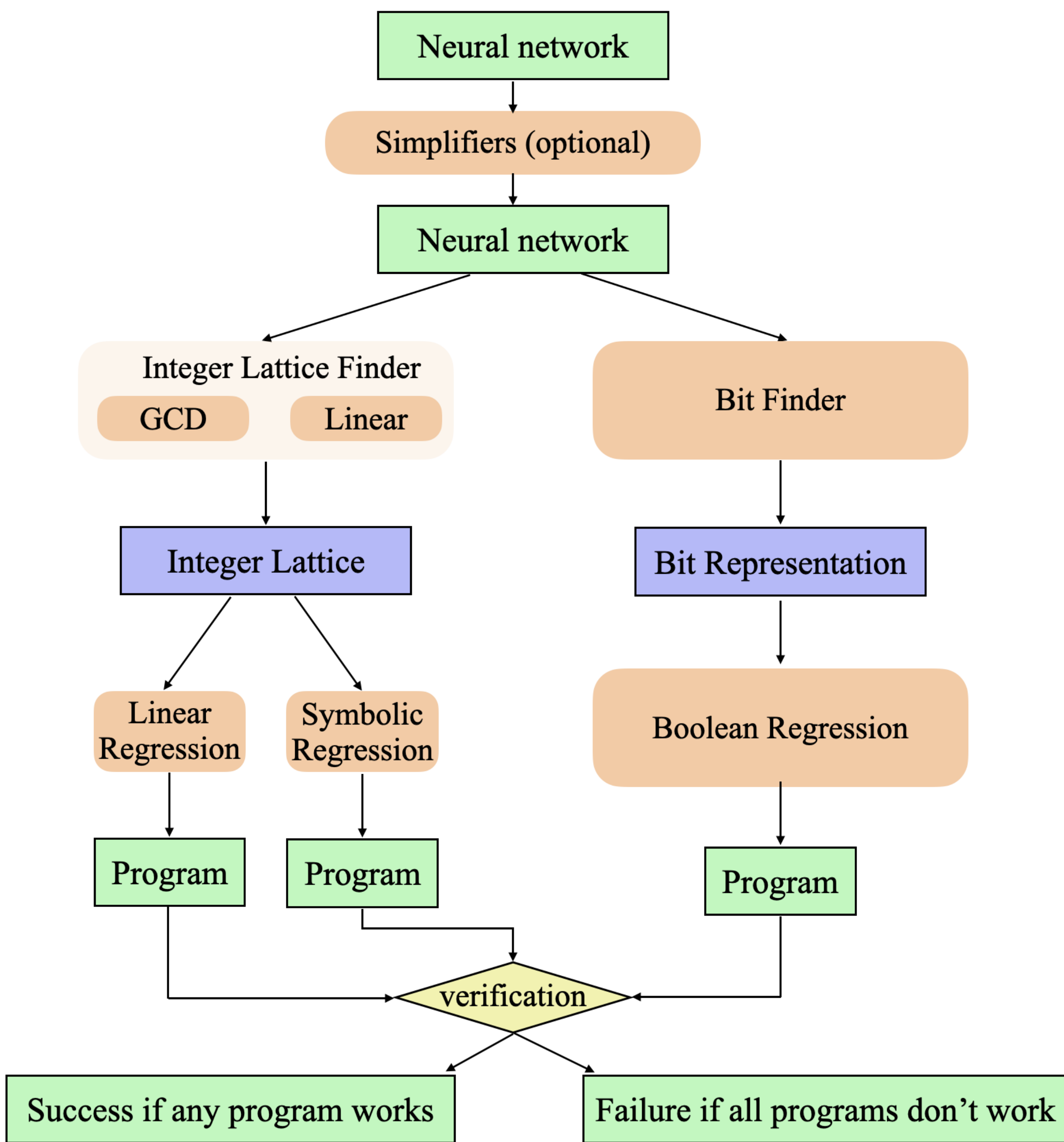


Figure 4.1: The pipeline of our program synthesis method. MIPS relies on discovering integer representations and bit representations of hidden states, which enable regression methods to figure out the exact symbolic relations between input, hidden, and output states.

4. Symbolic regression

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

$$\mathbf{h}_i = f(\mathbf{h}_{i-1}, \mathbf{x}_i), \tag{4.1}$$

$$\mathbf{y}_i = g(\mathbf{h}_i), \tag{4.2}$$

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 for 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 is 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 that 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.

4.3.1 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 which has 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 5 different seeds. We use the loss function $\ell(x, y) = \frac{1}{2}\log[1 + (x - y)^2]$, finding that it led 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.

4.3.2 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:

$$\begin{aligned} \mathbf{h}'_i &= \mathbf{A}\mathbf{h}_i = \mathbf{A}\mathbf{W}\mathbf{A}^{-1}\mathbf{A}\mathbf{h}_{i-1} + \mathbf{A}\mathbf{V}\mathbf{x}_i + \mathbf{A}\mathbf{b} \\ &= \mathbf{W}'^{-1}\mathbf{h}'_{i-1} + \mathbf{V}'\mathbf{x}_i + \mathbf{b}', \qquad (4.3) \\ \mathbf{y}_i &= G(\mathbf{U}\mathbf{h}_i + \mathbf{c}) = G(\mathbf{U}\mathbf{A}^{-1}\mathbf{h}'_i + \mathbf{c}) \\ &= G(\mathbf{U}'\mathbf{h}'_i + \mathbf{c}). \qquad (4.4) \end{aligned}$$

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 4.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 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

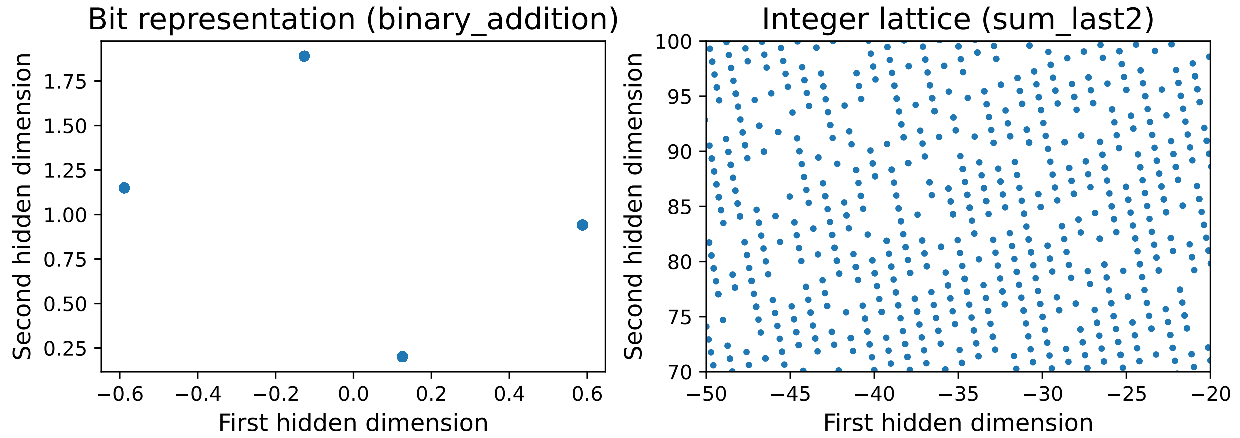


Figure 4.2: These hidden structures can be turned into discrete representations. Left: the hidden states for the bitstring addition task are seen to form four clusters, corresponding to 2 bits: the output bit and the carry bit. Right: the hidden states for the Sum_Last2 task are seen to form clusters on a 2D lattice corresponding to two integers.

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.

4.3.3 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 (Z. Liu et al. 2022; Nanda et al. 2023). 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 4.2 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 2 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.

Bit representations

The hidden states for the 2 bits in Figure 4.2 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.

Integer lattice

As seen in Figure 4.2, 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 4.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 4.B. Here the idea is to exploit 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.

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.

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.

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* (Udrescu et al. 2020), including the 6 unary operations $\{>, <, \sim, H, D, A\}$ and 4 binary operations $\{+, -, *, \%\}$ whose meanings are explained in Appendix 4.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 4.3 and 4.4 and in Appendix 4.F.

4.4 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.

4.4.1 Benchmark

Our benchmark consists of the 62 algorithmic tasks listed in Table 4.1. 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 is available at https://github.com/ejmichaud/neural-verification. The tasks are described in Table 4.1, with additional details in Appendix 4.E.

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.

4.4.2 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 4.5.

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.

Table 4.1: Benchmark results. For tasks with note "see text", please refer to Appendix 4.E

Task #	Input Strings	Element Type	Task Description	Task Name	Solved by GPT-4?	Solved by MIPS?
1	2	bit	Binary addition of two bit strings	Binary_Addition	0	1
2	2	int	Ternary addition of two digit strings	Base_3_Addition	0	0
3	2	int	Base 4 addition of two digit strings	Base_4_Addition	0	0
4	2	int	Base 5 addition of two digit strings	Base_5_Addition	0	0
5	2	int	Base 6 addition of two digit strings	Base_6_Addition	1	0
6	2	int	Base 7 addition of two digit strings	Base_7_Addition	0	0
7	2	bit	Bitwise XOR	Bitwise_Xor	1	1
8	2	bit	Bitwise OR	Bitwise_Or	1	1
9	2	bit	Bitwise AND	Bitwise_And	1	1
10	1	bit	Bitwise NOT	Bitwise_Not	1	1
11	1	bit	Parity of last 2 bits	Parity_Last2	1	1
12	1	bit	Parity of last 3 bits	Parity_Last3	0	1
13	1	bit	Parity of last 4 bits	Parity_Last4	0	0
14	1	bit	Parity of all bits seen so far	Parity_All	0	1
15	1	bit	Parity of number of zeros seen so far	Parity_Zeros	0	1
16	1	int	Cumulative number of even numbers	Evens_Counter	0	0
17	1	int	Cumulative sum	Sum_All	1	1
18	1	int	Sum of last 2 numbers	Sum_Last2	0	1
19	1	int	Sum of last 3 numbers	Sum_Last3	0	1
20	1	int	Sum of last 4 numbers	Sum_Last4	1	1
21	1	int	Sum of last 5 numbers	Sum_Last5	1	1
22	1	int	sum of last 6 numbers	Sum_Last6	1	1
23	1	int	Sum of last 7 numbers	Sum_Last7	1	1
24	1	int	Current number	Current_Number	1	1
25	1	int	Number 1 step back	Prev1	1	1
26	1	int	Number 2 steps back	Prev2	1	1
27	1	int	Number 3 steps back	Prev3	1	1
28	1	int	Number 4 steps back	Prev4	1	1
29	1	int	Number 5 steps back	Prev5	1	1
30	1	int	1 if last two numbers are equal	Previous_Equals_Current	0	1
31	1	int	current − previous	Diff_Last2	0	1
32	1	int	|current − previous|	Abs_Diff	0	1
33	1	int	|current|	Abs_Current	1	1
34	1	int	|current| − |previous|	Diff_Abs_Values	1	0
35	1	int	Minimum of numbers seen so far	Min_Seen	1	0
36	1	int	Maximum of integers seen so far	Max_Seen	1	0
37	1	int	integer in 0-1 with highest frequency	Majority_0_1	1	0
38	1	int	Integer in 0-2 with highest frequency	Majority_0_2	0	0
39	1	int	Integer in 0-3 with highest frequency	Majority_0_3	0	0
40	1	int	1 if even, otherwise 0	Evens_Detector	1	0
41	1	int	1 if perfect square, otherwise 0	Perfect_Square_Detector	0	0
42	1	bit	1 if bit string seen so far is a palindrome	Bit_Palindrome	1	0
43	1	bit	1 if parentheses balanced so far, else 0	Balanced_Parenthesis	0	0
44	1	bit	Number of bits seen so far mod 2	Parity_Bits_Mod2	1	0
45	1	bit	1 if last 3 bits alternate	Alternating_Last3	0	0
46	1	bit	1 if last 4 bits alternate	Alternating_Last4	1	0
47	1	bit	bit shift to right (same as prev1)	Bit_Shift_Right	1	1
48	2	bit	Cumulative dot product of bits mod 2	Bit_Dot_Prod_Mod2	0	1
49	1	bit	Binary division by 3 (see text)	Div_3	1	0
50	1	bit	Binary division by 5 (see text)	Div_5	0	0
51	1	bit	Binary division by 7 (see text)	Div_7	0	0
52	1	int	Cumulative addition modulo 3	Add_Mod_3	1	1
53	1	int	Cumulative addition modulo 4	Add_Mod_4	0	0
54	1	int	Cumulative addition modulo 5	Add_Mod_5	0	0
55	1	int	Cumulative addition modulo 6	Add_Mod_6	0	0
56	1	int	Cumulative addition modulo 7	Add_Mod_7	0	0
57	1	int	Cumulative addition modulo 8	Add_Mod_8	0	0
58	1	int	1D dithering, 4-bit to 1-bit (see text)	Dithering	1	0
59	1	int	Newton's of - freebody (integer input)	Newton_Freebody	0	1
60	1	int	Newton's law of gravity (see text)	Newton_Gravity	0	1
61	1	int	Newton's law w. spring (see text)	Newton_Spring	0	1
62	2	int	Newton's law w. magnetic field (see text)	Newton_Magnetic	0	0
			Total solved		30	32

4.4.3 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 4.3.1 discovers networks of varying task-dependent shape and size. Table 4.2 shows the parameters $\mathbf{p}$ discovered for each task. Across our 62 tasks, 16 tasks could be solved by a network with hidden dimension $n = 1$, 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.*, 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 4.3 shows an example of a MIPS rediscovering the "ripple-carry adder" algorithm. The normalizers significantly simplified some of the resulting programs, as illustrated in Fig. 4.4, 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.

```python

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
```

Figure 4.3: 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.

4.5 Conclusions

We have presented MIPS, a novel method for program synthesis based on 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.

4.5.1 Findings

We found MIPS highly complementary to LLM-based program synthesis with GPT-4 Turbo, with each approach solving many tasks that stumped the other. 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

```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 4.4: Comparison of code generated from an RNN trained on Sum_Last5, without (top) and with (bottom) normalizers. The whitening normalizer provided numerical stability to the Jordan normal form normalizer, which itself simplified the recurrent portion of the program. The Toeplitz and de-biasing normalizers jointly sparsified the occurrences of x in the program, and the number of terms required to compute y. The quantization normalizer enabled all variables to be represented as integers.

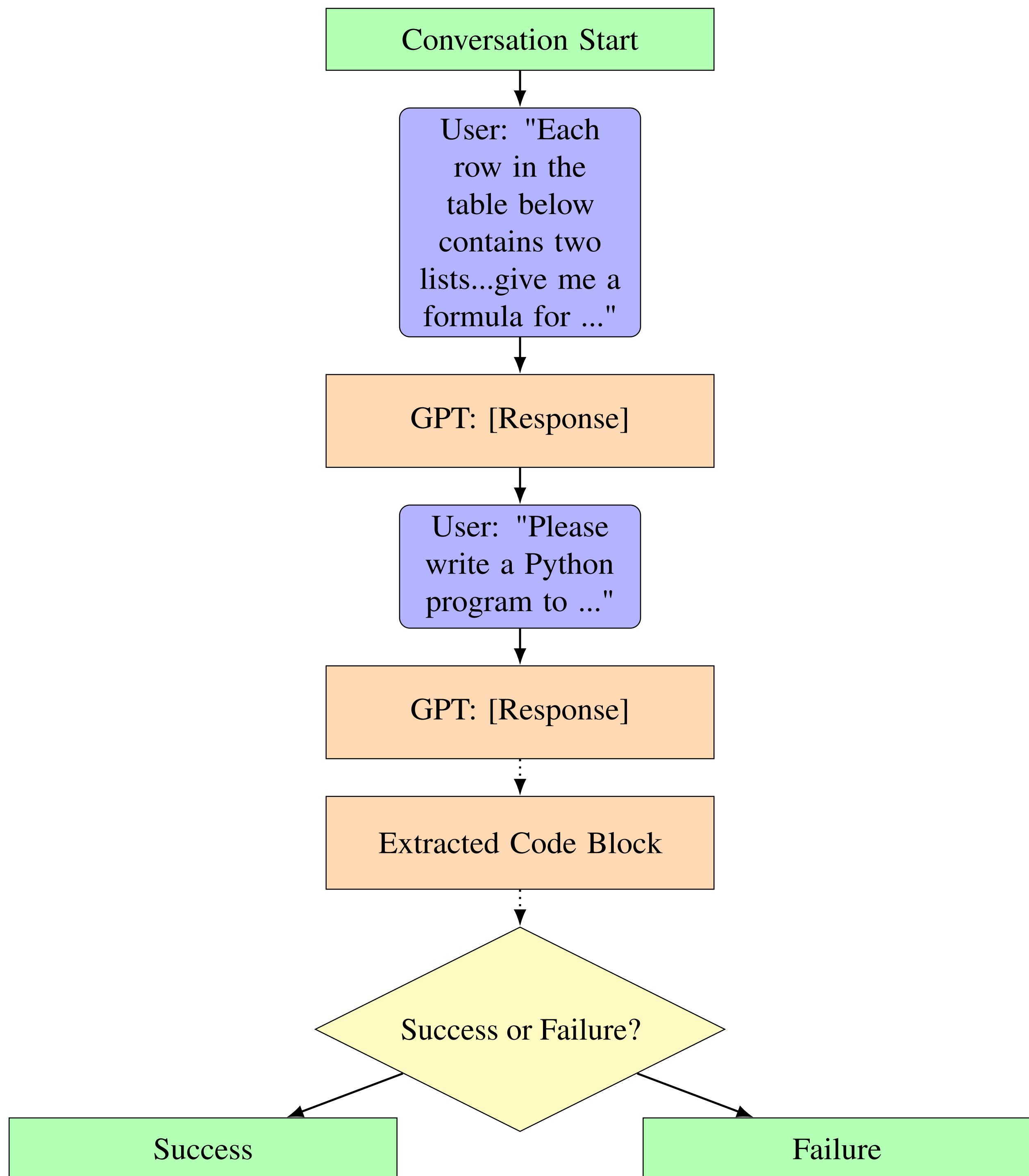


Figure 4.5: We compare MIPS against program synthesis with the large language model GPT-4 Turbo, prompted with a “chain-of-thought" approach. It begins with the user providing a task, followed by the model’s response, and culminates in assessing the success or failure of the generated Python code based on its accuracy in processing the provided lists.

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.

4.5.2 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 Section 4.F.1 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* (Udrescu et al. 2020; Cranmer 2023), which has already proven useful for various science and engineering applications (Cranmer et al. 2020; Ma et al. 2022) 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.

Broader Impact

Because machine-learned algorithms now outperform traditional human-discovered algorithms on many tasks, there are incentives to deploy them even without a full understanding of how they work and of whether they are biased, unsafe, or otherwise problematic. The aspirational broader impact motivating this paper is to help automate the process of making AI systems more transparent, robust, and trustworthy, with the ultimate goal of developing provably safe AI systems (Tegmark and Omohundro 2023).

Acknowledgements

We thank Wes Gurnee, James Liu, and Armaun Sanayei for helpful conversations and suggestions. This work is supported 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.

References

Tegmark, Max and Steve Omohundro (2023). "Provably safe systems: the only path to controllable agi". In: *arXiv preprint arXiv:2309.01933*.

Zhou, Baishun and Gangyi Ding (2023). "Survey of intelligent program synthesis techniques". In: *International Conference on Algorithms, High Performance Computing, and Artificial Intelligence (AHPCAI 2023)*. Vol. 12941. SPIE, pp. 1122–1136.

Odena, Augustus et al. (2020). “BUSTLE: Bottom-Up program synthesis through learning-guided exploration”. In: *arXiv preprint arXiv:2007.14381*.

Wu, Jiarong et al. (2023). “Programming by Example Made Easy”. In: *ACM Transactions on Software Engineering and Methodology* 33.1, pp. 1–36.

Sobania, Dominik, Martin Briesch, and Franz Rothlauf (2022). “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*, pp. 1019–1027.

Olah, Chris et al. (2020). “Zoom in: An introduction to circuits”. In: *Distill* 5.3, e00024–001.

Cammarata, Nick et al. (2020). “Curve Detectors”. In: *Distill*. https://distill.pub/2020/circuits/curve-detectors. DOI: 10.23915/distill.00024.003.

Wang, Kevin et al. (2022). “Interpretability in the wild: a circuit for indirect object identification in gpt-2 small”. In: *arXiv preprint arXiv:2211.00593*.

Olsson, Catherine et al. (2022). “In-context Learning and Induction Heads”. In: *Transformer Circuits Thread*. https://transformer-circuits.pub/2022/in-context-learning-and-induction-heads/index.html.

Goh, Gabriel et al. (2021). “Multimodal Neurons in Artificial Neural Networks”. In: *Distill*. https://distill.pub/2021/multimodal-neurons. DOI: 10.23915/distill.00030.

Gurnee, Wes and Max Tegmark (2023). “Language models represent space and time”. In: *arXiv preprint arXiv:2310.02207*.

Burns, Collin et al. (2022). “Discovering latent knowledge in language models without supervision”. In: *arXiv preprint arXiv:2212.03827*.

Marks, Samuel and Max Tegmark (2023). “The geometry of truth: Emergent linear structure in large language model representations of true/false datasets”. In: *arXiv preprint arXiv:2310.06824*.

McGrath, Thomas et al. (2022). “Acquisition of chess knowledge in alphazero”. In: *Proceedings of the National Academy of Sciences* 119.47, e2206625119.

Toshniwal, Shubham et al. (2022). “Chess as a testbed for language model state tracking”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 36. 10, pp. 11385–11393.

Li, Kenneth et al. (2022). “Emergent world representations: Exploring a sequence model trained on a synthetic task”. In: *arXiv preprint arXiv:2210.13382*.

Nanda, Neel et al. (2023). “Progress measures for grokking via mechanistic interpretability”. In: *arXiv preprint arXiv:2301.05217*.

Liu, Ziming et al. (2022). “Towards understanding grokking: An effective theory of representation learning”. In: *Advances in Neural Information Processing Systems* 35, pp. 34651–34663.

Zhong, Ziqian et al. (2023). “The clock and the pizza: Two stories in mechanistic explanation of neural networks”. In: *arXiv preprint arXiv:2306.17844*.

Quirke, Philip et al. (2023). “Understanding Addition in Transformers”. In: *arXiv preprint arXiv:2310.13121*.

Hanna, Michael, Ollie Liu, and Alexandre Variengien (2023). “How does GPT-2 compute greater-than?: Interpreting mathematical abilities in a pre-trained language model”. In: *arXiv preprint arXiv:2305.00586*.

Charton, Franccois (2023). “Can transformers learn the greatest common divisor?” In: *arXiv preprint arXiv:2308.15594*.

Lindner, David et al. (2023). “Tracr: Compiled transformers as a laboratory for interpretability”. In: *arXiv preprint arXiv:2301.05062*.

Conmy, Arthur et al. (2023). “Towards automated circuit discovery for mechanistic interpretability”. In: *arXiv preprint arXiv:2304.14997*.

Syed, Aaquib, Can Rager, and Arthur Conmy (2023). “Attribution Patching Outperforms Automated Circuit Discovery”. In: *arXiv preprint arXiv:2310.10348*.

Bills, Steven et al. (2023). *Language models can explain neurons in language models*. https://openaipublic.blob.core.windows.net/neuron-explainer/paper/index.html.

Cunningham, Hoagy et al. (2023). “Sparse autoencoders find highly interpretable features in language models”. In: *arXiv preprint arXiv:2309.08600*.

Bricken, Trenton et al. (2023). “Towards Monosemanticity: Decomposing Language Models With Dictionary Learning”. In: *Transformer Circuits Thread*. https://transformer-circuits.pub/2023/monosemantic-features/index.html.

Udrescu, Silviu-Marian et al. (2020). “AI Feynman 2.0: Pareto-optimal symbolic regression exploiting graph modularity”. In: *Advances in Neural Information Processing Systems* 33, pp. 4860–4871.

Cranmer, Miles (2023). “Interpretable machine learning for science with PySR and SymbolicRegression. jl”. In: *arXiv preprint arXiv:2305.01582*.

Cranmer, Miles et al. (2020). “Discovering symbolic models from deep learning with inductive biases”. In: *Advances in Neural Information Processing Systems* 33, pp. 17429–17442.

Ma, He et al. (2022). “Evolving symbolic density functionals”. In: *Science Advances* 8.36, eabq0279.

Gu, Albert and Tri Dao (2023). “Mamba: Linear-time sequence modeling with selective state spaces”. In: *arXiv preprint arXiv:2312.00752*.

4.A Lattice finding using 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.

4.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

$$\mathbf{x}_j = \sum_{i=1}^{D} a_{ji}\mathbf{b}_i + \mathbf{c}, \tag{4.5}$$

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 criterion is: (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 ($\mathbf{\Lambda} \in \mathbb{Z}^{D\times D}, \det(\mathbf{\Lambda}) = \pm 1$).

4.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

$$\mathrm{GCD}(y_1, \cdots, y_n) = \mathrm{GCD}(y_1, \mathrm{GCD}(y_2, \mathrm{GCD}(y_3, ...))), \tag{4.6}$$

it without loss of generality suffices to consider the case $n = 2$. A common algorithm to compute GCD of two number 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 kth 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 $|r_{k-1}| > |r_k|$ [1]. 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

$$\begin{aligned} &55 = 1 \times 45 + 10, \\ &45 = 4 \times 10 + 5, \\ &10 = 2 \times 5 + 0. \end{aligned} \tag{4.7}$$

4.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 Eq. (4.5), 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 a D-dimensional parallelogram by constructing its basis vectors as a linear integer combination of $\mathbf{y}_j$, i.e.,

$$\mathbf{q}_i = \sum_{j=1}^{n} m_{ij} \mathbf{y}_j, m_{ij} \in \mathbb{Z}, i = 1, 2, \cdots, D. \tag{4.8}$$

[1] 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 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 GCD of volumes of other possible parallelograms. Once the minimal parallelogram is found [2], we can also determine $\mathbf{b}_i$ in Eq. (4.5), since $\mathbf{b}_i$ are 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 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 get 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 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:

$$\begin{aligned} V_1 &= \det(\mathbf{y}_1, \mathbf{y}_3, \mathbf{y}_4, \ldots, \mathbf{y}_D) \\ V_2 &= \det(\mathbf{y}_2, \mathbf{y}_3, \mathbf{y}_4, \ldots, \mathbf{y}_D) \end{aligned} \tag{4.9}$$

Since

$$aV_1 + bV_2 = \det(a\mathbf{y}_1 + b\mathbf{y}_2, \mathbf{y}_3, \mathbf{y}_4, \ldots, \mathbf{y}_D), \tag{4.10}$$

which shows that (V_1, V_2) and $(\mathbf{y}_1, \mathbf{y}_2)$ share the same linear structure. We can simply apply

[2] There could be many minimal parallelograms, but finding one is sufficient.

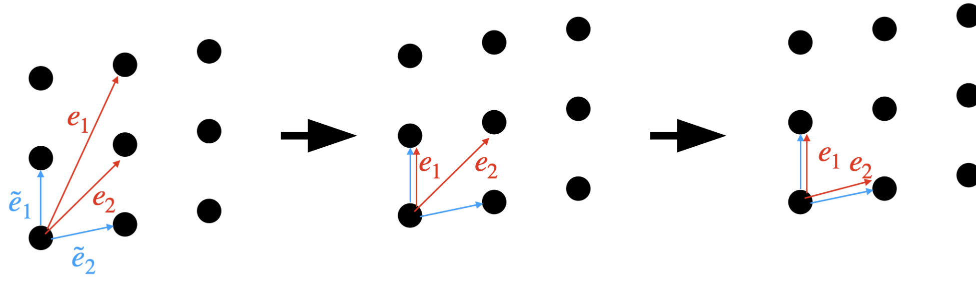


Figure 4.6: Both red and blue basis form a minimal parallelogram (in terms of cell volume), but one can further simplify red to blue by linear combination (simplicity in the sense of small ℓ_2 norm).

division-and-remainder to V_1 and V_2 as in regular GCD:

$$V_1', V_2' = \mathrm{GCD}(V_1, V_2), \tag{4.11}$$

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

$$\mathbf{y}_1', \mathbf{y}_2' = \mathrm{GCD_with_predefined_quotients}(\mathbf{y}_1, \mathbf{y}_2). \tag{4.12}$$

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

$$0 = V_2' = \det(\mathbf{y}_2', \mathbf{y}_3, \mathbf{y}_4, \cdots, \mathbf{y}_D), \tag{4.13}$$

this means $\mathbf{y}_2'$ is a linear combination of $(\mathbf{y}_3, \cdots, \mathbf{y}_D)$, 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 they 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 determinant 1 (suppose $\mathbf{b}_i (i = 2, 3, \cdots, D)$ are projected to $\mathbf{b}_1$ and subtracted by multiples of $\mathbf{b}_1$. p_* represents projection integers):

$$\begin{pmatrix} 1 & p_{2\to 1} & p_{3\to 1} & \cdots & p_{D\to 1} \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{pmatrix} \tag{4.14}$$

We do this iteratively, until no norm can become shorter via project-and-subtract. Please see Figure 4.6 for an illustration of how simplification works for a 2D example.

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 to that the absolute value of the remainder to be smaller than a threshold $\epsilon_{\rm gcd}$. Another issue regarding noise is that noise can be accumulated 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 data fraction $p\%$ of the whole data. Since both $\epsilon_{\rm gcd}$ and p depends on data and neural network training which we do not know a priori, we choose to grid sweep $\epsilon_{\rm gcd} \in [10^{-3}, 1]$ and $p \in (0.1, 100)$; for each $(\epsilon_{\rm gcd}, p)$, we obtain an integer lattice and compute its description length. We select the $(\epsilon_{\rm gcd}, p)$ which 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 $\log(1 + (\frac{AZ+b-X}{\epsilon_{\rm dl}})^2)$ with $\epsilon_{\rm dl} = 10^{-4}$.

4.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

$$\mathbf{h}^{(t)} = \sum_{j=1}^{t} W_h^{t-j} W_i \mathbf{x}_j, \tag{4.15}$$

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

$$W_h^{t-j} W_i, j = 1, 2, \cdots, t, \tag{4.16}$$

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 highest norms, which are exactly basis vectors of the hidden lattice.

4.C Symbolic regression

The formulation of symbolic regression is that one has 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 addition of two numbers; $*$ means multiplication of two numbers; $-$ means 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 6 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 success, we terminate the brute force program and return the successful formula. If brute force search does not find any correct symbolic formula within compute 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.

4.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", "De-bias", and "Quantization".

The main symmetry which we focus on is a linear transformation of the hidden space $\mathbf{h} \mapsto \mathbf{A}\mathbf{h}$,

which requires the following changes to f and g:

$$f(\mathbf{h}, \mathbf{x}) = \mathbf{W}\mathbf{h} + \mathbf{V}\mathbf{x} + \mathbf{b} \implies f(\mathbf{h}, \mathbf{x}) = \mathbf{A}\mathbf{W}\mathbf{A}^{-1}\mathbf{h} + \mathbf{A}\mathbf{V}\mathbf{x} + \mathbf{A}\mathbf{b}$$
$$g(\mathbf{h}) = G(\mathbf{U}\mathbf{h} + \mathbf{c}) \implies f(\mathbf{h}) = G(\mathbf{U}\mathbf{A}^{-1}\mathbf{h} + \mathbf{c})$$

and is implemented by changing the weights:

$$\mathbf{W} \implies \mathbf{A}\mathbf{W}\mathbf{A}^{-1}$$
$$\mathbf{V} \implies \mathbf{A}\mathbf{V}$$
$$\mathbf{b} \implies \mathbf{A}\mathbf{b}$$
$$\mathbf{U} \implies \mathbf{U}\mathbf{A}^{-1}$$

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 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 (Gu and Dao 2023) can perform just as well as transformer-based models in language modeling on a large scale. A main advantage of using linear hidden-to-hidden transformations is the possibility of expressing the hidden space in it's 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 doing this, 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 Jordan normal form, and all matrices can be transformed to Jordan normal form. A simple transformation can also be applied to 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.

4.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 which 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.

4.D.2 Jordan Normal Form Transformation

Critically, the hidden-to-hidden transformations which 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 Jordan normal form by:

$$\begin{pmatrix} 0 & 1 \\ \delta & 0 \end{pmatrix} = \underbrace{\begin{pmatrix} 1 & 1 \\ \sqrt{\delta} & -\sqrt{\delta} \end{pmatrix}}_{\mathbf{T}} \begin{pmatrix} \sqrt{\delta} & 0 \\ 0 & -\sqrt{\delta} \end{pmatrix} \begin{pmatrix} 1 & 1 \\ \sqrt{\delta} & -\sqrt{\delta} \end{pmatrix}^{-1} \tag{4.17}$$

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

$$\begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} = \underbrace{\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}}_{\mathbf{T}} \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}^{-1} \tag{4.18}$$

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 which 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 (4.18) instead of from Equation (4.17).

Our algorithm first computes the eigenvalues λ_i, and then iteratively solves for the generalized eigenvectors which 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 $\epsilon\text{-}\ker(\mathbf{X})$.

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 groups, ie. that $|\lambda_i - \lambda_j| \leq \epsilon$ and $|\lambda_j - \lambda_k| \leq \epsilon$ always implies $|\lambda_k - \lambda_i| \leq \epsilon$. 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, 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 which are mapped to zero after exactly k applications of the map $\mathbf{W} - \lambda\mathbf{I}$. We first solve for $\mathbf{F}_0 = \text{ker}(\mathbf{W} - \lambda\mathbf{I})$. 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 $\mathbf{F}_{k+1}$. Do this 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 which 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 which 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

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

4.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 numerically stability of a column vector is determined by the absolute value of the bottom element of that column vector, since it's inverse will become the degenerate eigenvalues of the resulting Toeplitz matrix. If no column has a numerically stability above $\epsilon = 0.0001$, we pick the identity matrix for our Toeplitz transformation.

4.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 perturabtion 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 $\epsilon = 0.1$.

4.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, sometimes this allows the entire network to become quantized.

4.E Supplementary training data details

Here we present additional details on the benchmark tasks marked "see text" in Table 4.1:

- **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 an euler forward propagation technique which follows the equation $F = \textit{input} - 1, v \mapsto v + F, x \mapsto x + v$.

- **Newton_Spring:** This is an euler forward propagation technique which follows the equation $F = \textit{input} - x, v \mapsto v + F, x \mapsto x + v$.

- **Newton_Magnetic:** This is an euler forward propagation technique which follows the equation $F_x = \textit{input}_1 - v_y, F_y = \textit{input}_2 + v_x, \mathbf{v} \mapsto \mathbf{v} + \mathbf{F}, \mathbf{x} \mapsto \mathbf{x} + \mathbf{v}$.

4.F Generated programs

This section includes all successfully generated Python programs.

Binary-Addition

```
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
```

Bitwise-Xor

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

Bitwise-Or

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

Bitwise-And

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

Bitwise-Not

```python

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-Last3

```python

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-Last2

```python

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-All

```python

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

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

Sum-Last2

```
1
2 def f(s):
3     a = 0;b = 99;
4     ys = []
5     for i in range(10):
6         x = s[i]
7         next_a = -b+x+99
8         next_b = -x+99
9         a = next_a;b = next_b;
10        y = a
11        ys.append(y)
12    return ys
```

Sum-All

```
1
2 def f(s):
3     a = 884;
4     ys = []
5     for i in range(10):
6         x = s[i]
7         next_a = a-x
8         a = next_a;
9         y = -a+884
10        ys.append(y)
11    return ys
```

Sum-Last3

```
1
2 def f(s):
3     a = 0;b = 198;c = 0;
4     ys = []
5     for i in range(10):
6         x = s[i]
7         next_a = x
8         next_b = -a-x+198
9         next_c = -b+198
10        a = next_a;b = next_b;c =
   next_c;
11        y = a+c
12        ys.append(y)
13    return ys
```

Sum-Last4

```python
1
2 def f(s):
3     a = 0;b = 99;c = 0;d = 99;
4     ys = []
5     for i in range(10):
6         x = s[i]
7         next_a = c
8         next_b = -x+99
9         next_c = -b-d+198
10        next_d = b
11        a = next_a;b = next_b;c =
   next_c;d = next_d;
12        y = a-b-d+198
13        ys.append(y)
14    return ys
```

Sum-Last5

```python
1
2 def f(s):
3     a = 198;b = -10;c = -2;d = 482;e =
      1;
4     ys = []
5     for i in range(20):
6         x = s[i]
7         next_a = -b+c+190
8         next_b = b-c-d-e+x+480
9         next_c = b-e+8
10        next_d = -b+e-x+472
11        next_e = a+b-e-187
12        a = next_a;b = next_b;c =
   next_c;d = next_d;e = next_e;
13        y = -d+483
14        ys.append(y)
15    return ys
```

Sum-Last6

```python

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

```python

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

```python

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

```python

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

```python

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

```python

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

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

Abs-Diff

```
1
2 def f(s):
3     a = 100;b = 100;
4     ys = []
5     for i in range(10):
6         c = s[i]
7         next_a = b
8         next_b = c+100
9         a = next_a;b = next_b;
10        y = abs(b-a)
11        ys.append(y)
12    return ys
```

Diff-Last2

```
1
2 def f(s):
3     a = 199;b = 100;
4     ys = []
5     for i in range(10):
6         x = s[i]
7         next_a = -a-b+x+498
8         next_b = a+b-199
9         a = next_a;b = next_b;
10        y = a-199
11        ys.append(y)
12    return ys
```

Abs-Current

```
1
2 def f(s):
3     a = 0;
4     ys = []
5     for i in range(10):
6         b = s[i]
7         next_a = abs(b)
8         a = next_a;
9         y = a
10        ys.append(y)
11    return ys
```

Bit-Shift-Right

```python

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
```

Add-Mod-3

```python

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
```

Bit-Dot-Prod-Mod2

```python

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
```

Newton-Freebody

```python

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

```python

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

```python

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
```

4.F.1 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

```

```

```
2 function ArrayToBv10(arr: array<bool>): bv10 // Converts boolean array to
    bitvector
3   reads arr
4   requires arr.Length == 10
5 {
6   ArrayToBv10Helper(arr, arr.Length - 1)
7 }
8
9 function ArrayToBv10Helper(arr: array<bool>, index: nat): bv10
10   reads arr
11   requires arr.Length == 10
12   requires 0 <= index < arr.Length
13   decreases index
14   ensures forall i :: 0 <= i < index ==> ((ArrayToBv10Helper(arr, i) >> i) &
    1) == (if arr[i] then 1 else 0)
15 {
16   if index == 0 then
17     (if arr[0] then 1 else 0) as bv10
18   else
19     var bit: bv10 := if arr[index] then 1 as bv10 else 0 as bv10;
20     (bit << index) + ArrayToBv10Helper(arr, index - 1)
21 }
22
23 method ArrayToSequence(arr: array<bool>) returns (res: seq<bool>) // Converts
    boolean array to boolean sequence
24   ensures |res| == arr.Length
25   ensures forall k :: 0 <= k < arr.Length ==> res[k] == arr[k]
26 {
27     res := [];
28     var i := 0;
29     while i < arr.Length
30         invariant 0 <= i <= arr.Length
31         invariant |res| == i
```

```
32          invariant forall k :: 0 <= k < i ==> res[k] == arr[k]
33      {
34          res := res + [arr[i]];
35          i := i + 1;
36      }
37  }
38
39  function isBitSet(x: bv10, bitIndex: nat): bool
40    requires bitIndex < 10
41    ensures isBitSet(x, bitIndex) <==> (x & (1 << bitIndex)) != 0
42  {
43    (x & (1 << bitIndex)) != 0
44  }
45
46  function Bv10ToSeq(x: bv10): seq<bool> // Converts bitvector to boolean
      sequence
47    ensures |Bv10ToSeq(x)| == 10
48    ensures forall i: nat :: 0 <= i < 10 ==> Bv10ToSeq(x)[i] == isBitSet(x, i)
49  {
50    [isBitSet(x, 0), isBitSet(x, 1), isBitSet(x, 2), isBitSet(x, 3),
51     isBitSet(x, 4), isBitSet(x, 5), isBitSet(x, 6), isBitSet(x, 7),
52     isBitSet(x, 8), isBitSet(x, 9)]
53  }
54
55  function BoolToInt(a: bool): int {
56      if a then 1 else 0
57  }
58
59  function XOR(a: bool, b: bool): bool {
60      (a || b) && !(a && b)
61  }
62
63  function BitAddition(s: array<bool>, t: array<bool>): seq<bool> // Performs
```

```
    traditional bit addition
64   reads s
65   reads t
66   requires s.Length == 10 && t.Length == 10
67 {
68     var a: bv10 := ArrayToBv10(s);
69     var b: bv10 := ArrayToBv10(t);
70     var c: bv10 := a + b;
71 
72     Bv10ToSeq(c)
73 }
74 
75 method f(s: array<bool>, t: array<bool>) returns (sresult: seq<bool>) //
    Generated program for bit addition
76   requires s.Length == 10 && t.Length == 10
77   ensures |sresult| == 10
78   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])))))
79   ensures BitAddition(s, t) == sresult // Verification of correctness
80 {
81   var a: bool := false;
82   var b: bool := false;
83   var result: array<bool> := new bool[10];
84   var i: int := 0;
85 
86   while i < result.Length
87     invariant 0 <= i <= result.Length
88     invariant forall j :: 0 <= j < i ==> result[j] == false
89   {
90         result[i] := false;
91         i := i + 1;
92   }
```

```
93
94    i := 0;
95
96    assert forall j :: 0 <= j < result.Length ==> result[j] == false;
97
98    while i < result.Length
99      invariant 0 <= i <= result.Length
100     invariant b == (i > 0 && ((s[i-1] || t[i-1]) && !(result[i-1] && (s[i-1]
       != t[i-1]))))
101     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])))))
102   {
103     assert b == (i > 0 && ((s[i-1] || t[i-1]) && !(result[i-1] && (s[i-1] != t
       [i-1]))));
104
105     result[i] := XOR(b, XOR(s[i], t[i]));
106     b := BoolToInt(b) + BoolToInt(s[i]) + BoolToInt(t[i]) > 1;
107     assert b == ((s[i] || t[i]) && !(result[i] && (s[i] != t[i])));
108
109     i := i + 1;
110   }
111
112   sresult := ArrayToSequence(result);
113 }
```

Table 4.2: 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	1	1	4	2	0	0
2	Base_3_Addition	2	1	1	5	2	0	0
3	Base_4_Addition	2	1	1	5	2	0	0
4	Base_5_Addition	2	1	1	5	2	0	0
5	Base_6_Addition	2	1	1	6	2	2.45e-09	2.53e-09
6	Base_7_Addition	2	1	1	10	2	2.32e-06	2.31e-06
7	Bitwise_Xor	1	1	1	2	2	0	0
8	Bitwise_Or	1	1	1	1	1	3.03e-02	3.03e-02
9	Bitwise_And	1	1	1	1	1	3.03e-02	3.03e-02
10	Bitwise_Not	1	1	1	1	1	0	0
11	Parity_Last2	1	1	1	229	2	1.68e-02	1.69e-02
12	Parity_Last3	2	1	1	5	2	1.62e-04	1.64e-04
13	Parity_Last4	3	1	1	29	2	3.07e-07	2.99e-07
14	Parity_All	1	1	1	2	2	0	0
15	Parity_Zeros	1	1	1	2	2	0	0
16	Evens_Counter	4	1	1	73	3	8.89e-05	8.88e-05
17	Sum_All	1	1	1	1	1	6.09e-08	6.13e-08
18	Sum_Last2	2	1	1	1	1	0	0
19	Sum_Last3	3	1	1	1	1	6.34e-07	6.35e-07
20	Sum_Last4	4	1	1	1	1	2.10e-04	2.11e-04
21	Sum_Last5	5	1	1	1	1	8.86e-03	8.87e-03
22	Sum_Last6	6	1	1	1	1	1.82e-02	1.81e-02
23	Sum_Last7	7	1	1	1	1	3.03e-02	3.01e-02
24	Current_Number	1	1	1	1	1	0	0
25	Prev1	2	1	1	1	1	0	0
26	Prev2	3	1	1	1	1	0	0
27	Prev3	4	1	1	1	1	0	0
28	Prev4	5	1	1	1	1	2.04e-07	2.05e-07
29	Prev5	6	1	1	1	1	6.00e-05	5.96e-05
30	Previous_Equals_Current	2	1	1	5	2	6.72e-05	6.61e-05
31	Diff_Last2	2	1	1	1	1	0	0
32	Abs_Diff	2	2	2	1	1	1.84e-07	1.84e-07
33	Abs_Current	1	1	1	2	2	4.51e-08	5.71e-08
34	Diff_Abs_Values	2	1	1	4	2	3.15e-06	2.96e-06
35	Min_Seen	1	1	1	2	2	0	0
36	Max_Seen	1	1	1	2	2	1.46e-12	0
37	Majority_0_1	1	1	1	63	2	4.03e-03	4.05e-03
38	Majority_0_2	4	1	1	98	2	1.64e-04	1.71e-04
39	Majority_0_3	21	1	1	132	3	6.94e-05	6.86e-05
40	Evens_Detector	5	1	1	163	2	8.18e-04	8.32e-04
41	Perfect_Square_Detector	48	1	1	100	2	1.92e-03	1.97e-03
42	Bit_Palindrome	18	1	1	86	2	3.81e-05	3.69e-05
43	Balanced_Parenthesis	1	1	1	16	2	7.44e-03	7.10e-03
44	Parity_Bits_Mod2	1	1	1	1	1	0	0
45	Alternating_Last3	2	1	1	3	2	1.85e-02	1.87e-02
46	Alternating_Last4	2	1	1	3	2	8.24e-06	8.09e-06
47	Bit_Shift_Right	2	1	1	1	1	0	0
48	Bit_Dot_Prod_Mod2	1	1	1	3	2	0	0
49	Div_3	2	1	1	59	2	6.40e-03	6.43e-03
50	Div_5	4	1	1	76	2	1.50e-04	1.55e-04
51	Div_7	4	1	1	103	2	6.65e-04	6.63e-04
52	Add_Mod_3	1	1	1	149	2	1.02e-03	1.04e-03
53	Add_Mod_4	2	1	1	33	2	1.53e-04	1.44e-04
54	Add_Mod_5	3	1	1	43	2	1.02e-03	1.03e-03
55	Add_Mod_6	4	1	1	108	2	6.14e-04	6.12e-04
56	Add_Mod_7	4	1	1	199	2	3.96e-04	4.07e-04
57	Add_Mod_8	67	1	1	134	2	8.53e-04	8.34e-04
58	Dithering	81	1	1	166	2	7.72e-04	7.75e-04
59	Newton_Freebody	2	1	1	1	1	2.61e-07	2.62e-07
60	Newton_Gravity	2	1	1	1	1	1.81e-07	1.87e-07
61	Newton_Spring	2	1	1	1	1	0	0
62	Newton_Magnetic	4	1	1	1	1	8.59e-05	8.60e-05

Chapter 5

Conclusion

In this thesis, I have explored three ways that we can automate our efforts to interpret and understand the internal functioning components of trained neural networks. The first technique from Chapter 2 detects irreducible multidimensional features; I presented a wealth of evidence that such multidimensional features are indeed fundamental and are most accurately though of in this way. The second technique from Chapter 3 trains generative models of neural networks with a simplicity regularization, such that the simplicity of the generated model translates into general human-interpretable structures. The third technique from Chapter 4 removes random symmetry transformations that have been applied to learned networks, simplifying their weights without changing their behavior, before converting these networks into short pieces of python code. These techniques will help us to modify networks to induce predictable and reliable changes in their behavior. The scalability of these techniques gestures towards the long-term goal of gaining fine-grained control over language models that are too large for manual interpretation. We hope that our techniques encourage further progress in advancing our ability to control neural networks, to make them safer to use, in situations where their unpredictable behavior has the potential to do great harm.

Appendix A

Notes on Jordan Normal Form

In Chapter 4, we make heavy use of the Jordan normal form (Brechenmacher 2006) in order to simplify recurrent neural networks. To understand Jordan normal form, we must first discuss diagonalization. When we diagonalize a square matrix $\mathbf{W}$, we find a basis in which $\mathbf{W}$ represents elementwise multiplication. If we write the basis vectors in a matrix $\mathbf{A}$, this means we find a matrix $\mathbf{A}$ such that the product

$$\mathbf{A}^{-1}\mathbf{W}\mathbf{A} = \mathbf{D}$$

is a diagonal matrix; we say that it is $\mathbf{W}$ in "diagonal form". We call the product

$$\mathbf{A}\mathbf{D}\mathbf{A}^{-1} = \mathbf{W}$$

the "eigendecomposition" of $\mathbf{W}$. Not all square matrices can be diagonalized in this way; matrices that cannot are called "defective". Matrices become defective when their eigenvalues are duplicated multiple times, more times than the associated eigenspace. An example is the matrix

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

which has two eigenvalues of zero, but maps the $(0,1)^T$ vector to $(1,0)^T$ and the $(1,0)^T$ vector to zero.

The Jordan normal form (JNF) is a **generalization** of the diagonal form, whose main purpose here is to let us still talk about "two unique eigenvectors" even though there is only one. Converting a matrix $\mathbf{W}$ into JNF is defined by finding a matrix $\mathbf{A}$ such that $\mathbf{A}^{-1}\mathbf{W}\mathbf{A}$ is diagonal, *except* that it can optionally have ones on the superdiagonal, whenever the two adjacent eigenvalues are the same. All square matrices can be converted into JNF.

The JNF of a matrix is divided into "Jordan blocks", each of which is a square matrix with identical eigenvalues on the diagonal and ones on the superdiagonal. These Jordan blocks can be written as $\lambda I + \mathbf{S}$, where $\mathbf{S}$ is the "shift matrix" with ones on the superdiagonal and zeros elsewhere. The columns of the matrix $\mathbf{A}$ which converts $\mathbf{W}$ into JNF, are known as the "generalized eigenvectors" of $\mathbf{W}$. The generalized eigenvectors corresponding to each Jordan block of eigenvalue λ can be called a "Jordan chain", since they each map to the previous generalized eigenvector in the chain under $(W - \lambda I)$, until the first generalized eigenvector maps to zero (Bronson 1991).

References

Drenkow, Nathan et al. (2021). "A systematic review of robustness in deep learning for computer vision: Mind the gap?" In: *arXiv preprint arXiv:2112.00639*.

Christiano, Paul F et al. (2017). "Deep reinforcement learning from human preferences". In: *Advances in neural information processing systems* 30.

Ziegler, Daniel M et al. (2019). "Fine-tuning language models from human preferences". In: *arXiv preprint arXiv:1909.08593*.

Sharkey, Lee, Dan Braun, and Beren Millidge (Dec. 2022). *[interim research report] taking features out of superposition with sparse autoencoders - AI alignment forum*. URL: https://www.alignmentforum.org/posts/z6QQJbtpkEAX3Aojj/interim-research-report-taking-features-out-of-superposition.

Elhage, Nelson et al. (2021). "A Mathematical Framework for Transformer Circuits". In: *Transformer Circuits Thread*. https://transformer-circuits.pub/2021/framework/index.html.

Gao, Leo et al. (2020). "The pile: An 800gb dataset of diverse text for language modeling". In: *arXiv preprint arXiv:2101.00027*.

Liao, Isaac, Ziming Liu, and Max Tegmark (2023). "Generating interpretable networks using hypernetworks". In: *arXiv preprint arXiv:2312.03051*.

Dalrymple, David et al. (2024). "Towards Guaranteed Safe AI: A Framework for Ensuring Robust and Reliable AI Systems". In: *arXiv preprint arXiv:2405.06624*.

Carr, Steven, Nils Jansen, and Ufuk Topcu (2020). "Verifiable RNN-based policies for POMDPs under temporal logic constraints". In: *arXiv preprint arXiv:2002.05615*.

Ayache, Stéphane, Rémi Eyraud, and Noé Goudian (2019). "Explaining black boxes on sequential data using weighted automata". In: *International Conference on Grammatical Inference*. PMLR, pp. 81–103.

Mayr, Franz, Ramiro Visca, and Sergio Yovine (2020). "On-the-fly black-box probably approximately correct checking of recurrent neural networks". In: *Machine Learning and Knowledge Extraction: 4th IFIP TC 5, TC 12, WG 8.4, WG 8.9, WG 12.9 International Cross-Domain Conference, CD-MAKE 2020, Dublin, Ireland, August 25–28, 2020, Proceedings 4*. Springer, pp. 343–363.

Mayr, Franz and Sergio Yovine (2018). "Regular inference on artificial neural networks". In: *Machine Learning and Knowledge Extraction: Second IFIP TC 5, TC 8/WG 8.4, 8.9, TC 12/WG 12.9 International Cross-Domain Conference, CD-MAKE 2018, Hamburg, Germany, August 27–30, 2018, Proceedings 2*. Springer, pp. 350–369.

Okudono, Takamasa et al. (2020). "Weighted automata extraction from recurrent neural networks via regression on state spaces". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 34. 04, pp. 5306–5314.

Omlin, Christian W and C Lee Giles (1996). "Extraction of rules from discrete-time recurrent neural networks". In: *Neural networks* 9.1, pp. 41–52.

Weiss, Gail, Yoav Goldberg, and Eran Yahav (2018). "Extracting automata from recurrent neural networks using queries and counterexamples". In: *International Conference on Machine Learning*. PMLR, pp. 5247–5256.

Lindner, David et al. (2024). "Tracr: Compiled transformers as a laboratory for interpretability". In: *Advances in Neural Information Processing Systems* 36.

Weiss, Gail, Yoav Goldberg, and Eran Yahav (2021). "Thinking like transformers". In: *International Conference on Machine Learning*. PMLR, pp. 11080–11090.

Grigsby, Elisenda, Kathryn Lindsey, and David Rolnick (2023). "Hidden symmetries of ReLU networks". In: *International Conference on Machine Learning*. PMLR, pp. 11734–11760.

Michaud, Eric J et al. (2024). “Opening the AI black box: program synthesis via mechanistic interpretability”. In: *arXiv preprint arXiv:2402.05110*.

Brechenmacher, Frederic (Mar. 2006). “Histoire du théorème de Jordan de la décomposition matricielle (1870-1930).
Formes de représentation et méthodes de décomposition.” Theses. Ecole des Hautes Etudes en Sciences Sociales (EHESS). URL: https://theses.hal.science/tel-00142786.

Bronson, R. (1991). *Matrix Methods: An Introduction*. Elsevier Science. ISBN: 9780121352516. URL: https://books.google.com/books?id=-sncYzdl9WEC.